POLITECHNIKA ŚLĄSKA

WYDZIAŁ AUTOMATYKI, ELEKTRONIKI I INFORMATYKI

KIERUNEK INFORMATYKA

Praca dyplomowa magisterska

Rozmyte grupowanie danych w środowisku Apache Hadoop

Autor: Mateusz Piątkowski

Kierujący pracą: dr inż. Bożena Małysiak-Mrozek

Gliwice, październik 2017

O Ś W I A D C Z E N I E

Wyrażam zgodę/nie wyrażam* zgody na udostępnienie mojej pracy dyplomowej/rozprawy doktorskiej*

....., dnia .....

.....

(podpis)

.....

(poświadczenie wiarygodności podpisu przez Dziekanat)

* *właściwe podkreślić*

Spis treści

Spis tabel

Spis listingów

Spis równań

Spis rysunków

1. Wstęp

Logika rozmyta ma wiele zastosowań, dzięki jej właściwościom. Szczególnie wykorzystywana jest w systemach sterowania jak i przy rozpoznawaniu wzorców. Logikę rozmytą można przyrównać do sposobu rozwiązywania problemów przez człowieka. Pomaga rozważyć wszystkie, często nieprecyzyjne dane i podjąć odpowiednią decyzję.

Ważną dziedziną, w której ma zastosowanie logika rozmyta jest *fuzzy matching*, polega ono na określeniu stopnia podobieństwa pomiędzy dwoma elementami. Wartość tego podobieństwa jest określana za pomocą dedykowanej funkcji. Następnie wartość ta jest w odpowiedni dla rodzaju zadania sposób interpretowana. Szczególnie często wykorzystywane jest to w operacjach na łańcuchach znaków. Działania te mają na celu na przykład znalezienie potencjalnych „duplikatów" rekordów w bazie. „Duplikaty" te nie są dokładnym wzajemnym odwzorowaniem, są to rekordy, które dla człowieka są jedną i tą samą wartością. Powodem powstania takich duplikatów mogą być na przykład błędy w pisowni. Innym zastosowaniem logiki rozmytej może być porównywanie rekordów, które ze względu na swój charakter, obarczone są pewną niekonsekwencją w zapisie. Przykładem takiej niekonsekwencji w zapisie mogą być adresy z ulicami – na przykład ulica imienia Marii Curie-Skłodowskiej może zostać zapisana jako ulica Skłodowskiej lub M. C. Skłodowskiej. Dla człowieka takie dwa rekordy są jednoznaczne, natomiast przy wykorzystaniu standardowej binarnej logiki rekordy te zdecydowanie nie są sobie równe.

Zadanie grupowania z użyciem logiki rozmytej sprowadza się do wielokrotnego wykorzystania procesu *fuzzy matching*. Po tym procesie następuje wyznaczenie grupy, poprzez zastosowanie funkcji klasyfikującej. Działanie to polega na wykorzystaniu wartości określającej stopień podobieństwa wyznaczonej przez funkcję klasyfikującą. Grupowanie podobnie jak dopasowywanie, może być wykorzystywane do znajdowania wielokrotnych powtórzeń w bazie danych powstałych ze względu na błędy w pisowni, niekonsekwentnego zapisu. Szczególnie przydatne może być to przy integracji danych pochodzących z różnych źródeł. Pochodzenie danych z różnych źródeł często oznacza różną konwencję zapisu rekordów, dzięki wykorzystaniu grupowania rozmytego, przy odpowiedniej implementacji można w zautomatyzowany sposób osiągnąć spójną nową bazę danych.

Niemal zawsze jednak jest potrzebne utworzenie dedykowanego rozwiązania dla danego problemu, aby osiągnąć zadowalający wynik. Wymaga to ekstensywnej analizy danych wejściowych i odpowiedniego przygotowania funkcji określającej podobieństwo, jak i później funkcji klasyfikującej. Brak ogólnych rozwiązań dla zadań tej klasy wynika z wysokiej różnorodności danych gromadzonych w dzisiejszych czasach.

2. Wybór rozwiązania

Istnieje wiele bibliotek i algorytmów pozwalających rozwiązywać zadania związane z logiką rozmytą. Najczęstszym typem danych stosowanych w tych zadaniach to łańcuchy znaków. Mogące reprezentować różne dane, takie jak struktury białek, dane teleadresowe, dane geolokalizacyjne. Ogrom ilości powstających takich danych, wymaga powstania odpowiednich narzędzi, by móc je sprawnie przetwarzać i analizować. Często się zdarza, że dane te pochodzą z różnych źródeł, które nie posiadają wspólnego standardu zapisu takich danych. Do analizy takich danych mogą być wówczas zastosowane różne algorytmy wykorzystujące logikę rozmytą.

2.1. Dane

Algorytmy zawierające logikę rozmytą znajdują głównie zastosowanie w analizie łańcuchów znaków. Często wykorzystywane są jako wspomaganie tłumaczenia, wykrywanie błędów w pisowni, wyszukiwanie rekordów o takim samym znaczeniu przy innym zapisie w bazie danych. Są to głównie rozwiązania dedykowane dla konkretnej klasy danych. Przykładem takiego dopasowania może być algorytm Soundex, którego zastosowanie ogranicza się tylko wyłącznie dla angielskich słów [1].

Dane do przebadania muszą posiadać wcześniej już przedstawioną relację, by móc później w wyniku eksperymentów ocenić skuteczność działania zaproponowanych rozwiązań. Częste błędy w pisowni są dobrym zestawem danych do testów. Dzięki jasnej relacji poprawny wyraz - wiele niepoprawnych sposobów zapisu owego wyrazu, można przygotować statystyki opisujące wyniki eksperymentu, w wyniku którego powstaje podobny zestaw danych z wykorzystaniem logiki rozmytej. Dobrym źródłem takich danych jest Wikipedia. Dzięki ogromnej liczbie artykułów i użytkowników, możliwe było utworzenie listy zawierającej pospolite pomyłki przy zapisie wyrazów [2]. Lista taka jest dostępna pod adresem URL:

https://en.wikipedia.org/wiki/Wikipedia:Lists_of_common_misspellings.

Przygotowana została także wersja tej listy do zastosowań informatycznych, która jest dostępna pod adresem:

https://en.wikipedia.org/wiki/Wikipedia:Lists_of_common_misspellings/For_machines.

Dane zastosowane w eksperymentach wykorzystują tę listę (stan na dzień 29.05.2017).

Drugim zestawem danych wykorzystanym w eksperymentach stanowi lista chorób i objawów wraz z wygenerowanymi literówkami. Zestaw ten został zbudowany za pomocą listy dostępnej pod adresem URL:

http://www.diseasesdatabase.com/.

Zestaw te zawiera oryginalną pisownię oraz literówki wygenerowane za pomocą skryptu. Dla każdego rekordu z listy chorób wygenerowano pięć błędnych zapisów: brak jednego losowego znaku, podwojony losowy znak, odwrócona kolejność dwóch losowych sąsiednich liter, zastąpienie losowego znaku poprzez inny losowy znak, wstawienie dodatkowego znaku. Dane zaczerpnięto z listy w dniu 21.09.2017.

2.2. Algorytmy podobieństwa

Istnieje wiele algorytmów pozwalających określić podobieństwo pomiędzy dwoma ciągami znaków [3]. Różnią się one sposobem wyznaczania pokrewieństwa pomiędzy ciągami znaków. Do najczęstszego sposobu wyznaczania pokrewieństwa wykorzystuje się metryki, które pozwalają określić odległość jednego ciągu znaków od drugiego. Innym przykładem mogą algorytmy bazujące na wektorach, które przekształcają ciąg znaków w wektory, a następnie określają podobieństwo na ich bazie.

2.2.1. Adjacent Pairing – Współczynnik Sørensena

Jest to algorytm wykorzystujący do ustalenia podobieństwa liczbę sąsiednich par [4]. Pierwszym elementem działania algorytmu jest utworzenie dla dwóch ciągów znaków wektora z parami znaków. Przykładowo dla „drukarka” byłby to wektor {dr, ru, uk, ka, ar, rk, ka}, a dla słowa „drukarz” – {dr, ru, uk, ka, ar, rz}. Następnie znajdowana jest część

wspólna z powstałych w ten sposób zbiorów. Same podobieństwo wyrażane jest równaniem 1.

$$similarity(s1, s2) = \frac{2 * |pairs(s1) \cap pairs(s2)|}{|pairs(s1)| + |pairs(s2)|}$$

Równanie 1 Adajacent Pairing

Dla podanego wyżej przykładu wartość podobieństwa po podstawieniu do wzoru wynosi:

$$similaririty(drukarka, drukarz) = \frac{2 * |\{dr, ru, uk, ka, ar\}|}{|\{\text{dr, ru, uk, ka, ar, rk, ka}\}| + |\{\text{dr, ru, uk, ka, ar, rz}\}|} = \frac{2 * 6}{7 + 6} \approx 0{,}923$$

Metryka ta zawsze zwraca wartość między 0 - zero wspólnych par, a 1 – dokładnie te same ciągi znaków, można traktować tę wartość jako procent podobieństwa [4]. Metryka ta uwzględnia zarówno wzajemne położenie znaków jak i częstość ich występowania do określenia podobieństwa.

2.2.2. Cosine similarity

Jest to metryka wskazująca na podobieństwo pomiędzy dwoma niezerowymi wektorami w przestrzeni unitarnej, mierząca cosinus kąta pomiędzy nimi [5]. Aby móc wykorzystać tak zdefiniowaną metrykę do porównania podobieństwa dwóch łańcuchów znaków, należy przekształcić te ciągi do postaci wektorowej. Można to realizować na wieloraki sposób. Na przykład dla ciągów znaków, które mają postać zdania należy zliczyć unikalne słowa występujące w danych dwóch zdaniach. Liczebność tych unikalnych słów, stanowi wymiar stworzonych wektorów [6]. Następnie dla każdego zdania zliczamy liczbę słów i przypisujemy odpowiedniej współrzędnej w wektorze. Aby przystosować ten algorytm do przetwarzania pojedynczych słów, postępujemy analogicznie, zamiast zliczanych słów, zliczamy poszczególne litery budujące słowo. Przykładowo dla słów „cosinus” i „sinus” budujemy te wektory w sposób przedstawiony w tabeli 1.

Litera	„cosinus”	„sinus”
c	1	0
o	1	0
s	2	2
i	1	1
n	1	1
u	1	1

Tabela 1 Przykład budowy wektorów

Mając zdefiniowane wektory $\vec{a} = (1, 1, 2, 1, 1, 1)$ i $\vec{b} = (0, 0, 2, 1, 1, 1)$, możemy wykorzystać metrykę określoną równaniem 2.

$$similarity(\vec{a}, \vec{b}) = cos\theta = \frac{\vec{a} \cdot \vec{b}}{\|\vec{a}\|\|\vec{b}\|}$$

Równanie 2 Cosine Similarity

Podstawiając dane do wzoru otrzymujemy wartość:

$$similarity(\vec{a}, \vec{b}) = \frac{7}{\sqrt{63}} \approx 0{,}882$$

Ważnym do odnotowania faktem jest to że metryka ta pomija kolejność występowania znaku w określaniu podobieństwa [6]. Tak sformułowany algorytm zwraca wartości pomiędzy 0 – żadnych wspólnych znaków, a 1 – dokładnie taka sama liczba takich samych znaków [6].

2.2.3. Levenshtein

Jest to metryka pozwalająca określić odległość pomiędzy dwoma łańcuchami znaków. Określona jest ona przez minimalną liczbę operacji pozwalających przekształcić

jeden ciąg w drugi. W podstawowej wersji algorytmu operacje te są sprecyzowane jako operacja insercji – wstawienia znaku, operacja usunięcia znaku oraz substytucji – operacja zamienienia znaku na inny [7]. Metryka ta wyznaczana jest równaniem 3.

$$lev_{a,b}(i,j) = \begin{cases} \max(i,j) \ jeżeli \ \min(i,j) = 0 \\ min \begin{cases} lev_{a,b}(i-1,j) + 1 \\ lev_{a,b}(i,j-1) + 1 \\ lev_{a,b}(i-1,j-1) + 1_{(a_i \neq b_j)} \end{cases} \end{cases}$$

Równanie 3 Levenshtein

Gdzie $1_{(a_i \neq b_j)}$ jest funkcją przyjmującą wartość 0, gdy $a_i = b_j$, a w pozostałych przypadkach 1. Symbole a i b reprezentują dwa porównywane ciągi znaków, a odpowiednio i i j reprezentują ich długości. Algorytm ten zwraca wartość, która jest równa 0, gdy łańcuchy są takie same, gdy są zupełnie różne wartość ta jest równa długości dłuższego łańcucha. Dla przykładu dla wyrazów „kitten” i „sitting” odległość wynosi 3. Można to skrótowo przedstawić w trzech krokach:

$$kitten \rightarrow sitten - zamiana \text{ „s” na „k”}$$
$$sitten \rightarrow sittin - zamiana \text{ „e” na „i”}$$
$$sittin \rightarrow sitting - insercja \text{ "g" na końcu}$$

Aby przekształcić wartości zwracane przez algorytm do przedziału 0 do 1 wykorzystywana jest równanie 4.

$$similarity(s1, s2) = 1 - \left(\frac{lev_{s1,s2}(|s1|, |s2|)}{\sqrt{\max(|s1|, |s2|) * \sqrt{|s1| * |s2|}}} \right)$$

Równanie 4 Podobieństwo Levenshtein

Gdzie $s1$ i $s2$ reprezentują dwa porównywane łańcuchy znaków.

2.2.4. Damerau-Levenshtein

Metryka ta jest rozszerzeniem algorytmu Levenshteina [8]. Definiuje ona dodatkową możliwą operację – transpozycję, poza trzema dozwolonymi w podstawowej wersji algorytmu (insercja, usunięcie i substytucja). Podobnie jak metryka Levenshteina jej

wartość wynosi tyle ile minimalnie trzeba przeprowadzić zdefiniowanych operacji, by przekształcić jeden ciąg znaków w drugi. Metryka ta zdefiniowana jest równaniem 5.

$$d_{a,b}(i,j) = \begin{cases} \max(i,j) \ jeżeli \ \min(i,j) = 0 \\ min \begin{cases} d_{a,b}(i-1,j)+1 \\ d_{a,b}(i,j-1)+1 \\ d_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \\ d_{a,b}(i-2,j-2) \end{cases} dla \ i,j > 1 \ i \ a_i = b_{j-1} \ i \ a_{i-1} = b_j \\ min \begin{cases} d_{a,b}(i-1,j)+1 \\ d_{a,b}(i,j-1)+1 \\ d_{a,b}(i-1,j-1)+1_{(a_i \neq b_j)} \end{cases} \end{cases}$$

Równanie 5 Damerau-Levenshtein

Gdzie $1_{(a_i \neq b_j)}$ jest funkcją przyjmującą wartość 0, gdy $a_i = b_j$, a w pozostałych przypadkach 1. Symbole a i b reprezentują dwa porównywane ciągi znaków, a odpowiednio i i j reprezentują ich długości.

Tak samo jak w algorytmie Levenshteina by przekształcić wartości zwracane przez algorytm do przedziału od 0 do 1, wykorzystywana jest funkcja przedstawiona równaniem 6.

$$similarity(s1,s2) = 1 - \left(\frac{d_{s1,s2}(|s1|,|s2|)}{\sqrt{\max(|s1|,|s2|) * \sqrt{|s1| * |s2|}}} \right)$$

Równanie 6 Podobieństwo Damerau-Levenshtein

Gdzie $s1$ i $s2$ reprezentują dwa porównywane łańcuchy znaków.

2.2.5. Jaro-Winkler

Algorytm Jaro-Winklera opiera się na algorytmie Jaro wyrażonym równaniem 7.

$$similarity_j(s1,s2) = \begin{cases} 0 \ dla \ m = 0 \\ \frac{1}{3} * \left(\frac{m}{|s1|} + \frac{m}{|s2|} + \frac{m-t}{m} \right) \end{cases}$$

Równanie 7 Jaro

Gdzie m to liczba dopasowanych znaków, t to liczba transpozycji, a $|s1|$ i $|s2|$ to odpowiednio długość wyrazu pierwszego i drugiego. Dwa znaki są według algorytmu dopasowane, gdy ten pochodzący z pierwszego ciągu znaków jest taki sam jak ten z drugiego i nie jest ulokowany dalej niż $\left\lfloor \frac{\max(|s1|,|s2|)}{2} \right\rfloor - 1$ [9]. Dla każdej pary dopasowanych znaków z inną sekwencją znaków, liczba transpozycji t zwiększana jest o 1. Rozwinięcie Winklera opierało się o dodaniu do wartości podobieństwa otrzymanej z algorytmu Jaro czynnika, który premiuje ciągi znaków zaczynające się tym samym symbolem [9]. Wyraża się on przez równanie 8.

$$similarity_{jw}(s1, s2) = similarity_j(s1, s2) + l * p * (1 - similarity_j(s1, s2))$$

Równanie 8 Jaro-Winkler

Gdzie $similarity_j(s1, s2)$ to wartość wyliczona z algorytmu Jaro, l to długość sekwencji takich samych znaków na początku dwóch słów, którego maksymalna wartość wynosi 4, a p to współczynik zawierający się w przedziale $p\epsilon[0,1]$. Standardowa wartość współczynnika p wynosi $p = 0{,}1$ [10]. Przykład działania algorytmu dla łańcuchów „mathematics" i „matematica". Długość pierwszego wyrazu wynosi 11 drugiego 10. Obliczamy maksymalną odległość, by móc określić czy znak jest dopasowany : $d_m = \left\lfloor \frac{\max(11,10)}{2} \right\rfloor - 1 = 4$. Pierwsze cztery znaki w obu ciągach są zgodne więc liczba $m = 3$. Następny znaki są różne, zgodnie z maksymalną odległością szukamy znaku „h", w odległości 4 znajdują się znaki odpowiednio po lewej stronie „mat" a po prawej „mati", dlatego znak „h" nie jest uznany za dopasowany. Następnie znak „e" uznawany jest za dopasowany, gdyż zawiera się w odległości, zwiększana jest liczba $m = 4$.

Kolejne kroki wykonywane są w ten sam sposób. Po zakończeniu wszystkich kroków otrzymujemy liczbę $m = 9$. Następnie przeprowadzane jest obliczanie liczby transpozycji. Sporządzana jest lista jak występują wszystkie dopasowane znaki w każdym ze słów, odpowiednio „matematic" dla pierwszego łańcucha i „matematic" dla drugiego. W tym przypadku wszystkie dopasowane znaki występują na tej samej pozycji więc liczba $t = 0$. Przykładowo dla wyrazów „house" i „houes" liczba t wynosiła by 1. Następnie po podstawieniu do wzoru otrzymujemy wartość:

$$similarity_j(s1, s2) = \frac{1}{3} * \left(\frac{9}{11} + \frac{9}{10} + \frac{9}{9}\right) \approx 0{,}906$$

Następnie wyznaczana jest liczba l, która w tym przypadku wynosi 3 (znaki „mat"). Po podstawieniu do wzoru otrzymujemy wartość, przy współczynniku $p = 0{,}1$:

$$similarity_{jw}(s1, s2) = \ 0{,}906 + 3 * 0{,}1 * (1 - 0{,}906) = 0{,}9342$$

Na podstawie przykładu można zauważyć, że podobieństwo między dwoma łańcuchami jest większe w algorytmie Jaro-Winkler niż przy samej odległości Jaro, dzięki uwzględnieniu tożsamych znaków występujących na początku obydwu wyrazów. Przy ograniczeniu liczby l do 4 i współczynniku p nie większym niż 0,25 algorytm zwraca wartości z przedziału pomiędzy 0 a 1 [10].

2.2.6. Ratcliff/Obershelp

Algorytm korzysta z równania 9.

$$similarity(s1, s2) = \frac{2 * K_m}{|s1| + |s2|}$$

Równanie 9 Ratcliff/Obershelp

Gdzie K_m to liczba dopasowanych znaków, a $|s1|$ i $|s2|$ to odpowiednio długość wyrazu pierwszego i drugiego. Liczba dopasowanych znaków wyliczana jest poprzez znalezienie najdłuższego wspólnego podciągu znaków, długość tego podciągu nazywana jest *anchor* [9]. Następnie wartość K_m jest zwiększana o długość *anchor*. Później pozostałe części ciągu znaków znajdujące się po prawej i lewej strony *anchor* muszą przeanalizowane w taki sam sposób. Proces ten trwa, dopóki nie zostaną przetworzone wszystkie znaki w każdym ze dwóch wejściowych ciągów znaków. Przykład działania algorytmu dla słów „mathematics" i „matematica" wygląda następująco. Pierwsza wyliczana jest długość wyrazów, dla przykładu to odpowiednio $|s1| = 11$ i $|s2| = 10$. Następnie wyznaczany jest najdłuższy wspólny podciąg, w tym przypadku to „ematic" o długości równej 6. Wartość K_m ustalana jest na 6. Po lewej stronie *anchor* znajdują się odpowiednio dla $s1$ i $s2$ ciągi „math" i „mat". Najdłuższy wspólny podciąg dla nich to „mat" o długości 3. Wartość K_m powiększana jest o nią i teraz wynosi 9. Po lewej stronie ciągu „mat" nie znajdują się żadne znaki, a po prawej stronie w pierwszym wyrazie znajduje się znak „e", natomiast w drugim nie znajduje się

żaden znak, wartość K_m pozostaje niezmienna. Następnie analizowana jest część po prawej stronie od oryginalnej *anchor*. W tym przypadku dla pierwszego ciągu jedynym znakiem znajdującym się po prawej stronie jest „s”, podobnie dla drugiego wyrazu to znak „a”. Wartość K_m pozostaje na poziomie 9. Podstawiając do wzoru otrzymujemy wartość:

$$similarity(s1, s2) = \frac{2 * 9}{10 + 11} = \frac{18}{21} \approx 0{,}857$$

Tak samo jak poprzednie algorytmy zwraca wartości z przedziału pomiędzy 0 a 1, gdzie 1 otrzymywane jest, gdy porównywane ciągi są całkowicie zgodne [9].

2.3. Biblioteki i narzędzia

Istotnym problemem, która stwarza analiza takich danych i wykorzystanie podanych algorytmów jest czas potrzebny na ich wykonanie. Przy tak ogromnej ilości danych powstających każdego dnia, potrzebne są narzędzia mogące jest sprawnie i niezawodnie przetworzyć. Często przy wykorzystaniu takich narzędzi, kosztem otrzymanej lepszej szybkości wykonania, jest trudniejsza implementacja problemu.

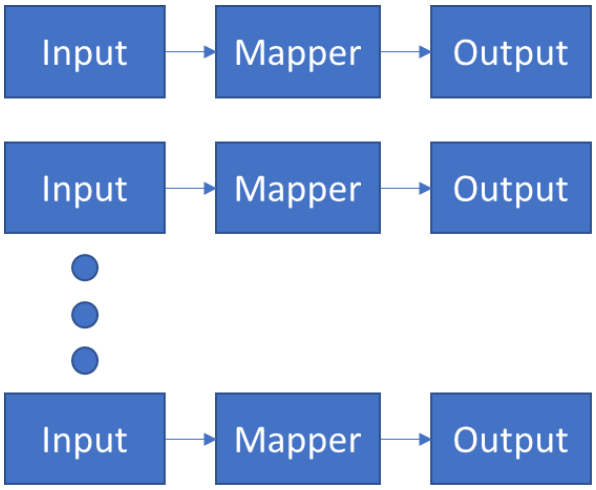
2.3.1. Hadoop

Apache Hadoop to narzędzie używane do składowania i przetwarzania rozproszonego danych [11]. Działa on na podstawie klastrów. Wszystkie części składowe biblioteki zbudowane są z myślą, że awarie sprzętowe są powszechne i takie zdarzenia powinny być obsługiwane przez bibliotekę [12]. Podstawowa wersja Apache Hadoop zbudowana jest z następujących modułów:

- Hadoop Common – zawiera biblioteki używane przez inne moduły
- Hadoop Distributed File System (HDFS) – rozproszony system przechowywania danych na urządzeniach tworzących klastry
- Hadoop YARN – platforma odpowiedzialna za zarządzanie zasobami obliczeniowymi klastrów i szeregowaniem aplikacji użytkownika
- Hadoop MapReduce – implementacja modelu programowania *MapReduce* [11].

Apache Hadoop napisany jest głównie w języku programowania Java. Powszechnym jest stosowanie tego języka do definiowania zdań w modelu *MapReduce*.

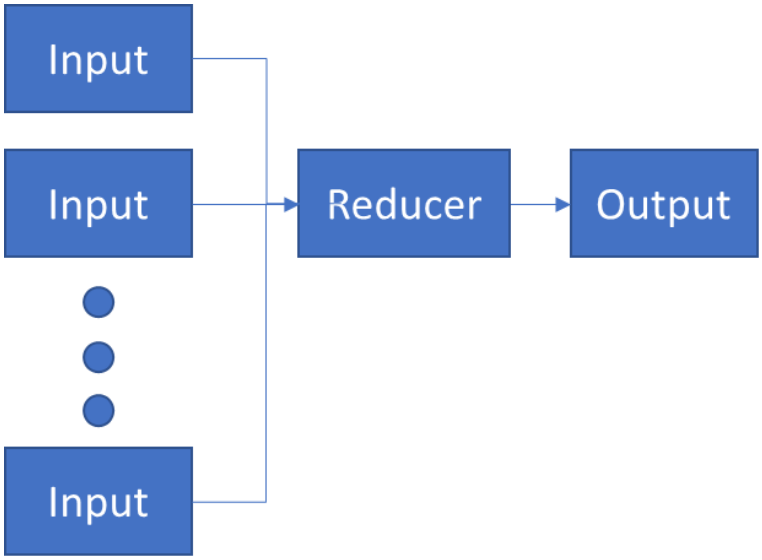
MapReduce to model programowania, który pomaga przetwarzać dużą ilość danych poprzez podział zadania na niezależne mniejsze części i wykonywanie ich równolegle [13]. Model *MapReduce* jest wzorowany na występujących w funkcyjnych językach programowania konstruktach map i reduce, które są powszechnie używane do przetwarzania list. Model *MapReduce* zaimplementowany w bibliotece Hadoop składa się z trzech faz: *map*, *shuffle and sort* i reduce. W pierwszej fazie zwanej *map* funkcja – *mapper* przetwarza serie danych w formacie klucz-wartość. Funkcja ta sekwencyjnie przetwarza każdą z par osobno i wytwarza zero lub więcej par klucz-wartość. Działanie tej fazy przedstawia ono na rysunku 1.



Rysunek 1 Rysunek przedstawiający fazę Map. Źródło: [13]

Drugą fazą działania jest część *shuffle and sort*. Po zakończeniu pracy *mapper*, pośrednie rezultaty przenoszone są do *reducers*. Proces ten jest nazywany *shuffling*, wykonywany jest on przez element zwany *partitioner*. Otrzymuje on z każdego *mapper* klucz oraz posiada informację na temat liczebności *reducers* w następnej fazie. Zwraca natomiast indeks przydzielonego do klucza *reducer*. Zapewnia on, że każda otrzymana wartość trafi do dobrego *reducer*. Domyślnie jego działanie oparte jest na funkcji haszującej. Po wykonaniu tego działania przez partitioner, wykonywane jest działanie zwane *sort*. Pośrednie rezultaty są sortowane przez Hadoop, zanim przekazane zostaną do obróbki w następnej fazie.

W fazie *reduce* funkcja zwana reducer otrzymuje wszystkie wartości o takim samym kluczu. Sama funkcja agreguje odpowiednio wartości i zwraca zero lub więcej par klucz wartość. Działanie tej fazy przedstawione zostało na rysunku 2.



Rysunek 2 Rysunek przedstawiający fazę Reduce. Źródło: [13]

W pakiecie Hadoop znajduje się narzędzie zwane *Hadoop Streaming*, które pozwala na definiowanie zadań w modelu *MapReduce* jako dowolny skrypt, plik wykonywalny. Zarówno *mapper* jak i *reducer* są w tym przypadku plikami wykonywalnymi, które odczytują ze standardowego wejścia(stdin), przetwarzają i wypisują na standardowe wyjście(stdout). Narzędzie Hadoop tworzy zadanie *MapReduce*, wysyła je do klastra i monitoruje progres do zakończenia [13]. Po fazie inicjalizacji pracy, powoływane są zadania map, każde uruchamiane jest jako osobny proces. Do każdego z tych zadań przekazywana jest pojedyncza linia tekstu z pliku wejściowego. Po przetworzeniu przez każdy z procesów, dane są pobierane ze standardowego wyjścia i konwertowane do formatu klucz-wartość. Jako klucz interpretowana jest wartość przed pierwszym znakiem tabulatora. Reszta uznawana jest jako wartość. Podobnie w fazie *reduce*, dla każdego *reducer* powoływany jest osobny proces. Przetwarzanie danych z wejścia i wyjścia jest prawie tak samo zorganizowane jak w fazie *map*, z tym, że na wejście może być podane wiele linii.

2.3.2. mrjob

To biblioteka utworzona przez firmę Yelp, która bazuje na *Hadoop Streaming*. Pozwala konstruować zadania *MapReduce* w języku Python w prostszy sposób [13]. Pozwala definiować wielokrokowe zadania. Zawiera też elementy pozwalające w łatwy sposób uruchomić napisane zadania na takich usługach chmurowych jak *Amazon Elastic MapReduce* i *Google Cloud Dataproc* [14]. Przydatną funkcją tej biblioteki jest opcja testowania zadań lokalnie. Dużą zaletą tej biblioteki jest możliwość przechowywania kodu całości zadania w jednej klasie, a także łatwe zarządzanie zależnościami. Dzięki opcji uruchamiania zadań lokalnie jest ułatwione ich debugowanie. Biblioteka ta jest aktywnie rozwijana, a jej źródła ogólnie dostępne pod adresem URL https://github.com/Yelp/mrjob.

Przykładowe zadanie zliczające słowa zawarte w pliku tekstowym za pomocą biblioteki przedstawiono na listingu 1.

```
from mrjob.job import MRJob

class wordCount(MRJob):

    def mapper(self, _, line):
        for word in line.split():
            yield(word, 1)

    def reducer(self, word, counts):
        yield(word, sum(counts))

if __name__ == '__main__':
    wordCount.run()
```

Listing 1 Przykładowe zadnie MapReduce

Aby uruchomić tak zdefiniowane zadanie lokalnie należy w terminalu wpisać polecenie:

```
python <nazwa_pliku_z_zadaniem> <plik wejściowy>
```

Wynikiem tak uruchomionego zadania będzie wypisanie na terminal słów i ich liczebności w danym pliku wejściowym w postaci: <słowo> <liczebność>. Aby uruchomić zadanie z użyciem *Amazon Elastic MapReduce* należy w poleceniu uwzględnić parametr *-r (runner),* który wówczas powinien przyjąć wartość *emr*. Analogicznie, żeby uruchomić na klastrze hadoop parametr *-r* powinien przyjąć wartość *hadoop*, a dla *Google Cloud Dataproc* - dataproc. Oczywiście dla usług w chmurze wcześniej musi być skonfigurowana

odpowiednio autoryzacja. Służy do tego plik *mrjob.conf*. Przykładowa najprostsza konfiguracja dla *Amazon Elastic MapReduce* przedstawiono na listingu 2.

```
runners:
  emr:
    aws_access_key_id: <your key ID>
    aws_secret_access_key: <your secret>
```

Listing 2 Przykładowa konfiguracja

Biblioteka posiada też możliwość uruchamiana zadań dla biblioteki *Spark* z wykorzystaniem *PySpark* [14].

2.3.3. Jellyfish

Biblioteka zawierające funkcje porównujące łańcuchy znaków [15]. Ich implementacja jest zarówno w języku C jak i Python. Przy typowej instalacji interpretera CPython wykorzystywane są funkcje napisane w języku C, co gwarantuje lepszą wydajność. W ramach pracy wykorzystana została implementacja algorytmu Damerau-Levenshtein oraz Jaro-Winkler.

3. Opis rozwiązania

Całość rozwiązania od ekstrakcji danych, poprzez ich przetworzenie w modelu MapReduce oraz analizę została zrealizowana za pomocą wielu skryptów, wykorzystujących język Python w wersji 3. Było to możliwie dzięki bardzo dużej liczbie modułów i bibliotek pozwalających na stworzenie między innymi wykresów, histogramów, konstruowania zadań zleconych do wykonania na klastrze obliczeniowym ulokowanym w chmurze.

Do wykonania obliczeń wykorzystany został klaster, wykorzystujący usługę *DataProc*, która znajduje się w ofercie *Google Cloud*. Wykorzystany klaster posiadał jeden węzeł główny w konfiguracji *n1-standard-4*, posiadający 4 procesory wirtualne, 15 GB pamięci RAM oraz dysk twardy o pojemności 1000 GB. Główny węzeł zawierał menedżera zasobów YARN, HDFS NameNode i wszystkie sterowniki zadania. Do klastra należały 4 węzły robocze, każdy z nich w konfiguracji *n1-standard-1*. Każdy z nich posiadał jeden procesor wirtualny, 3,75 GB pamięci RAM, dysk podstawowy o pojemności 10 GB oraz dysk SSD o pojemności 375 GB. Węzły robocze zawierały YARN NodeMenager i HDFS DataNode, a współczynnik replikacji HDFS wynosił 2. Dodatkową usługą wykorzystywaną w rozwiązaniu jest *Google Storage,* potrzebna jako miejsce do przechowywania plików wejściowych, pośrednich i wyjściowych.

3.1. Ekstrakcja danych

Dane zaczerpnięte ze źródła:

https://en.wikipedia.org/wiki/Wikipedia:Lists_of_common_misspellings/For_machines

mają postać:

```
Błędnie zapisany wyraz->poprawna forma lub formy
```

Przekształcane są za pomocą skryptu do czterech zestawów danych. Skrypt jako parametry przyjmuje plik tekstowy zawierający pierwotne dane oraz parametr liczbowy określający minimalną liczebność grupy stworzonej z pierwotnych danych. Za grupę uważany jest zestaw klucz wartość, gdzie kluczem jest poprawnie zapisany wyraz, wartościami jest

tablica z niepoprawnymi zapisami, liczebność tej tablicy musi być większa od wartości parametru podanego przy stracie skryptu.

Po odczytaniu i przefiltrowaniu danych, następuje zapisanie czterech zestawów danych. Pierwszy z nich o nazwie *„corrSet<wartość_parametru_filtrującego>.txt”* zawiera wszystkie klucze, czyli wszystkie poprawne formy zapisu, każda w nowej linii. Drugi o nazwie *„missSet<wartość_parametru_filtrującego>.txt”* zawiera natomiast wszystkie wartości, czyli wszystkie błędne wyrazy, podobnie jak w przypadku pierwszego zestawu każdy w nowej linii. Pliki te są wyłączni używane do analizy danych wyjściowych, pozwalają określić liczebność wyrazów z obydwu grup, która jest potrzebna przy obliczaniu statystyk takich jak wartość *true negative*. Pozwalają też zweryfikować poprawność utworzonego pliku wejściowego do zadania w modelu *MapReduce.* Używane są także do określenia statystyk opisujących wejściowy zbiór danych.

Trzeci zestaw o nazwie *input<wartość_parametru_filtrującego>.txt”* zawiera iloczyn kartezjański z powstałych poprzednio zestawów danych. W każdej linii tego zestawu znajduje się jeden element zestawu pierwszego oraz oddzielony spacją element zestawu drugiego. Ma to na celu utworzenie zestawu przystosowania do przetworzenia go jako wartość wejściowa w zadaniach w modelu *MapReduce*. Dzięki takiemu przekształceniu możliwe jest porównanie każdego wyrazu z pierwszego zestawu z każdym wyrazem z zestawu drugiego w łatwy i zorganizowany sposób. Ostatni natomiast zestaw o nazwie *„valiSet<wartość_parametru_filtrującego>.json”* zawiera wartości w schemacie JSON o formacie przedstawionym na listingu 3.

```
{
  "poprawny_zapis_1": [
    "niepoprawny_zapis1_1",
    "niepoprawny_zapis1_2",
    "niepoprawny_zapis1_3",
    "niepoprawny_zapis1_4"
  ],
  "poprawny_zapis_2": [
    "niepoprawny_zapis2_1",
    "niepoprawny_zapis2_2",
    "niepoprawny_zapis2_3"
  ]
}
```

Listing 3 Schemat Json

Jest on wyłącznie wykorzystywany na etapie analizy danych wyjściowych, gdzie oceniane są powstałe w wyniku eksperymentu grupy.

Do eksperymentów zastosowany został zestaw danych powstały w wyniku uruchomienia skryptu z parametrem 2, czyli znajdują się w nim tylko takie poprawne wyrazy, które mają przypasowane minimum 3 niepoprawne zapisy. Liczebność tak skonstruowanego zestawu danych wynosi 267 poprawnych wyrazów i w sumie 1002 niepoprawnych wyrazów. Co daje do przetworzenia z użyciem modelu *MapReduce* 267 534 rekordów.

Drugi zestaw danych zaczerpnięty ze źródła:

http://www.diseasesdatabase.com/

zawiera spis chorób, objawów i innych powiązanych rzeczy. W każdej nowej linii znajduje się nowy rekord. Rekord ten także może zawierać odwołanie, jeśli posiada synonim. Ma wtedy postać:

```
rekord see synonim
```

Rekord może także zawierać znaki specjalne i dopiski zawarte w nawiasach. Przekształcany jest tak samo jak pierwszy zestaw danych do czterech zbiorów. Dodatkowymi elementami w ekstrakcji tych danych są ich czyszczenie ze znaków specjalnych, dopisków, rozbijanie rekordów zawierających synonimy na dwa osobne rekordy. Takim wyczyszczenie danych jest realizowane przez skrypt, który przyjmuje trzy argumenty: nazwę zbioru wejściowego stworzonego na podstawie źródła, maksymalną liczbę rekordów zawartych w nowym zbiorze oraz nazwę zbioru wyjściowego, który stanowi ekwiwalent zbioru *„corrSet.txt”* z pierwszego zestawu danych. Następny skrypt realizuje wygenerowanie ekwiwalentów zbiorów *„missSet.txt”, „input.txt”, „valiSet.json”*. Przyjmuje on za argument wejściowy nazwę pliku z wyczyszczonymi danymi. Następnie dla każdego rekordu generowana jest grupa 5 rekordów zawierających literówki. Są to: brak jednego losowego znaku, podwojony losowy znak, odwrócona kolejność dwóch losowych sąsiednich liter, zastąpienie losowego znaku poprzez inny losowy znak, wstawienie dodatkowego losowego znaku na losowej pozycji. Po wygenerowaniu błędnych zapisów tworzone są zbiory danych *missSet.txt”, „input.txt”, „valiSet.json”,* których zawartość jest taka sama jak w przypadku pierwszego

zbioru danych. Jedyną różnicą jest znak rozdzielający dwa rekordy w zbiorze *„input.txt”*, jest to znak *„;”*, jest to spowodowane tym, że rekord może się składać z wielu wyrazów.

Do eksperymentów został utworzony zbiór o liczebności 2000 prawidłowych rekordów, co daje 10 000 błędnych zapisów oraz 20 000 000 rekordów do przetworzenia w modelu *MapReduce*. Taki rozmiar danych został podyktowany możliwościami dostępnego klastra obliczeniowego.

Zasadność wykorzystania modelu *MapReduce* i zastosowania narzędzia Hadoop do rozwiązania takiej grupy problemów może być fakt znacznego przyrostu liczby porównań przy dodaniu wyrazu dla jednej z tych grup. Na przykład w pierwszym zbiorze danych, przy dodaniu wyrazu do grupy wyrazów niepoprawnych, konieczne jest przeprowadzenie dodatkowych 267 porównań. Podobnie przy dodaniu jedynie jednego wyrazu do grupy wyrazów poprawnych, dodatkowo musi być przeprowadzone 1002 porównań.

3.2. Implementacja z użyciem MapReduce

Przygotowano w sumie 6 zadań z wykorzystaniem modelu *MapReduce*. Każde z tych zadań wykorzystuje inny algorytm porównywania podobieństwa między łańcuchami znaków. Trzy z nich, to jest C*osine Similarity*, *Adjacent Pairing* oraz algorytm *Levnshteina* są zaimplementowane bezpośrednio w kodzie zadania. Trzy pozostałe, czyli *Damerau-Levenshtein, Jaro-Winkler* oraz *Ratcliff/Obershelp* wykorzystują bibliotekę jako źródło implementacji. Algorytmy *Damerau-Levenshtein* oraz *Jaro-Winkler* pochodzą z biblioteki *jellyfish*. Natomiast *Ratcliff/Obershelp* pochodzi z modułu *difflib* zawartego w standardowej bibliotece Python. Dodatkowo algorytmy *Levenshtein* i *Damerau-Levenshtein* zawierają dodatkową metodę pozwalającą przetworzyć odległość, którą zwraca wartość znajdującą się w przedziale od 0 do 1.

Każde z zadań zawiera podział na dwa kroki przetwarzania. Model działania każdego z zadań można zademonstrować na przykładzie kodu zadania z użyciem algorytmu *Ratcliff/Obershelp*. Kod tego zadania wygląda przedstawiono na listingu 4.

```
1 from difflib import SequenceMatcher
2
3 from mrjob.job import MRJob
```

```
4 from mrjob.protocol import JSONValueProtocol
5 from mrjob.step import MRStep
6
7
8 class ratcliffSimilarity(MRJob):
9
10     OUTPUT_PROTOCOL = JSONValueProtocol
11     def mapperSimilarity(self, _, line):
12         SIMILARITY_THRESHOLD = 0.9
13         words = line.split(' ')
14         sim = SequenceMatcher(_,words[0], words[1]).ratio()
15         if(sim > SIMILARITY_THRESHOLD):
16             yield(words[0], [words[1], sim])
17
18     def reducerConstructDictionaries(self, key, values):
19         yield(None, {key : list(values)})
20
21     def reducerConstructJSON(self, _, dictionaries):
22         yield(None, self.mergeDicts(dictionaries))
23
24     def steps(self):
25         return[
26         MRStep(mapper = self.mapperSimilarity,
27         reducer = self.reducerConstructDictionaries),
28         MRStep(reducer = self.reducerConstructJSON)
29     ]
30
31     def mergeDicts(self, dict_args):
32         result = {}
33         for dictionary in dict_args:
34             result.update(dictionary)
35         return result
36
37
38
39 if __name__ == '__main__':
40     ratcliffSimilarity.run()
```

Listing 4 Kod zadania

Konstrukcja każdego zadania jest taka sama. Przed definicją klasy, czyli w liniach od 1 do 5 znajdują się załączenia bibliotek. Ważnym elementem są linie od 3 do 5, zawierają one definicje pochodzące z biblioteki mrjob. Dzięki nim możliwie jest odpowiednie skonfigurowanie zadania. Do tej konfiguracji wykorzystywana jest wartość

JSONValueProtocol, przypisana do zmiennej OUTPUT_PROTOCOL w linii 10, co ustala jaką postać mają otrzymywane z zadania dane. W każdym z eksperymentów otrzymywane są dane w postaci JSON.

Drugą ważną częścią konfiguracji jest ustalenie kroków zadania. Każdy z nich jest traktowany jako osobne zadanie wykonywane w klastrze obliczeniowym. Odbywa się to za pomocą metody *steps*, zdefiniowanej w linii numer 24. W ramach tej metody tworzona jest lista z obiektami klasy *MRStep.* Konstruktor takiego obiektu przyjmuje za parametry metody z klasy definiującej zadanie. Parametry te określają rolę metody w zadaniu, jak można to zaobserwować w linii 26, rolę *mapper* przyjmuje metoda *mapperSimilarity* zdefiniowana w linii 10. Podobnie rolę *reducer* określonej w linii 27 przyjmuje metoda *reducerConstructDictionaries*, znajdująca się w 21 linii. Możliwe jest definiowanie kroków zawierających tylko jedną fazę.

Pierwszym elementem zadania jest przekształcenie linii tekstu otrzymanej na wejściu, jest to jedna z linii pochodzących z pliku wejściowego. Następnie z linii tej wyłuskiwane są dwa wyrazy poprzez przedzielenie łańcucha znaków w miejscu znaku spacji w przypadku pierwszego zbioru danych lub znaku „;” w przypadku drugiego zbioru danych, taka operacja wykonywana jest w linii 13. Tak powstałe słowa są następnie porównywane poprzez jeden z algorytmów, w tym przypadku jest to wykonywane w linii 14. W większości przypadków tego rodzaju funkcja przyjmuje 2 parametry, czyli porównywane słowa. Wartość otrzymana z algorytmu jest następnie porównywana do minimalnego ustalonego poziomu decydującego czy dołączyć dany wyraz do grupy wyrazów podobnych. Jeżeli ten minimalny poziom został zaspokojony funkcja emituje parę klucz-wartość, gdzie kluczem jest wyraz pierwszy, czyli ten z poprawnym zapisem. Natomiast wartością jest lista dwuelementowa, gdzie pierwszym elementem jest wyraz drugi – niepoprawne zapisane słowo, a drugim wartość podobieństwa otrzymana z algorytmu, jest to realizowane poprzez linię 16. Całość tego działania pełni rolę *mapper* pierwszego kroku zadania.

W ramach pierwszego kroku następuje faza *reduce*, do każdego *reducer* trafia lista wartości przypisana do konkretnego klucza. Tworzony jest obiekt typu słownik, gdzie kluczem jest poprawne zapisane słowo, a wartością jest lista wartości, gdzie pojedynczą

wartością jest dwu elementowa lista zawierająca niepoprawnie zapisane słowo oraz wynik działania algorytmu podobieństwa. Przykładowy wynik przedstawiono na listingu 5.

```
{
  "year": [
    [
      "allready",
      0.7216878364870323
    ],
    [
      "relaly",
      0.7071067811865475
    ],
    [
      "yeasr",
      0.8944271909999159
    ],
    [
      "yera",
      1.0
    ],
    [
      "yeras",
      0.8944271909999159
    ],
    [
      "yersa",
      0.8944271909999159
    ],
    [
      "yrea",
      1.0
    ]
  ]
}
```

Listing 5 Wynik działania kroku

Jako wyjście z fazy *reduce* emitowane jest para klucz-wartość, gdzie wartością jest obiekt podobny jak przedstawiony wyżej, a kluczem staje się wartość *None,* co powoduje, że w następnym kroku wszystkie wartości otrzymane z *reducers* w tym kroku trafiają do jednej metody. Całość realizowana tej fazy jest realizowana w linii numer 19. To działanie konkluduje pierwszy krok.

Zadaniem kroku drugiego jest zespolenie wszystkich wyników w jeden obiekt. Obiekt ten ma format słownika klucz-wartość, gdzie kluczem jest poprawne zapisane słowo, a wartością jest lista tak jak w poprzednim kroku, całość jest realizowana w 22 linii. W tym kroku występuje tylko faza *reduce*.

4. Badania i eksperymenty

Do analizy danych pochodzących z eksperymentu utworzono skrypt, który przyjmuje za parametry plik z danymi pochodzącymi z eksperymentu, plik walidujący powstały wcześniej zawierający elementy klucz-wartość – poprawne słowo-lista niepoprawnych zapisów danego słowa oraz adres folderu, do którego zapisane będą wyniki działania skryptu. W wyniku działania tego skryptu powstaje pięć plików. Trzy z nich to histogramy powstałe z użyciem bibliotek *matplotlib* i *astropy*. Pozostałe dwa pliki zawierają statystyki zapisane w postaci JSON.

4.1. Statystyki

Dla każdego dobrze zapisanego słowa – klucza, wyliczane są wyznaczane statystki przedstawione na listingu 6.

```
"about":{
   "matchNumber":1,
   "correctMatches":1,
   "excessMatches":0,
   "maxCorrectMatches":4,
   "coverage":0.25,
   "correctRatio":1.0,
   "wrongRatio":0.0,
   "truePositive":1,
   "falsePositive":0,
   "falseNegative":3,
   "trueNegative":998,
   "recall":0.25,
   "specifity":1.0,
   "falsePositiveRate":0.0,
   "precision":1.0,
   "f_score":0.4,
   "accurancy":0.9970059880239521
}
```

Listing 6 Statystyki

„mathNumer” to liczba przypasowanych do tego słowa wyrazów. Wartość „correctMatches” natomiast mówi o liczbie przypasowanych słów, które znajdują się zarówno w pliku walidującym, jak i pliku otrzymanym z eksperymentu. „excessMatches” – informuje o liczbie nadmiarowo dopasowanych wyrazów, to jest wyrazów, których nie ma w pliku walidującym, a „maxCorrectMatches” mówi o liczbie wyrazów znajdujących się dla

danego klucza w pliku walidującym. Wartości „coverage”, „correctRatio” i „wrongRatio” wyrażane są odpowiednio przez równania 10, 11 i 12.

$$coverage = \frac{correctMatches}{maxCorrectMatches}$$

Równanie 10 Coverage

$$correctRatio = \frac{correctMatches}{matchNumber}$$

Równanie 11 Correct Ratio

$$wrongRatio = \frac{excessMatches}{matchNumber}$$

Równanie 12 Wrong Ratio

Pozostałe statystki wyliczone są odpowiednio z wartości *True Positive, False Positive, True Negative, False Negative*. W ich wyliczeniu, jako że problem można traktować jako *multi-label-classifiction*, należało przeprowadzić binaryzację otrzymanych wyników eksperymentu [16]. Warto zauważyć, że dla każdej klasy/grupy jest wykonywana liczba porównań równa liczbie niepoprawnych wyrazów, a każdy z tych wyrazów może zostać przypisany do wielu grup, w wyniku jednego eksperymentu [17].

Statystki te zapisane są w pliku *„experimentStats.json”*.

Drugi plik nazwany *„globalStats.json”* zawiera agregaty statystyk „coverage”, „correctRatio” i „wrongRatio”. Przedstawiony jest na listingu 7.


```
{
  "coverage": {
    "mean": 0.7364990190832887,
    "median": 0.8821428571428571,
    "mean_deviation": 0.2671705846557508,
    "standard_deviation": 0.3289265178955095
  },
  "correctRatio": {
    "mean": 0.6862740927538994,
    "median": 0.875,
    "mean_deviation": 0.31948712092463305,
    "standard_deviation": 0.35529071153249764
  },
  "wrongRatio": {
```

```
    "mean": 0.22383826679666244,
    "median": 0.03503184713375797,
    "mean_deviation": 0.26433799765892096,
    "standard_deviation": 0.2907025224042517
  }
}
```


Listing 7 Agregaty statystyk

Są one wyliczane na podstawie wszystkich wyników eksperymentu. „mean” to średnia wartość danej statystyki.

Wartość „median” to mediana wyliczana z równania 13.

$$median = l + \frac{interval * \left(\frac{n}{2} - cf\right)}{f}$$

Równanie 13 Mediana

Gdzie l to dolny limit przedziału, n – całkowita liczba punktów pomiarowych, cf – liczba punktów pomiarowych poniżej przedziału mediany oraz f – liczba punktów w przedziale mediany. Wartość $interval$ jest równa 0,1.

Wartość „mean_deviation” to średnie odchylenie bezwzględne wyznaczane z równania 14.

$$D = \frac{\sum_{i=1}^{n} |x_i - \bar{x}|}{n}$$

Równanie 14 Średnie odchylenie bezwzględne

Gdzie n to liczebność zbioru danych, $\bar{x}$ to średnia, a x_i – wartość *i*-tego elementu zbioru danych.

Ostatnia z wartości „standard_deviation” – to odchylenie standardowe wyliczane z równania 15.

$$\sigma = \sqrt{\frac{\sum_{i=1}^{n} (x_i - \bar{x})^2}{n}}$$

Równanie 15 Odchylenie standardowe

Gdzie n to liczebność zbioru danych, $\bar{x}$ to średnia, a x_i – wartość *i*-tego elementu zbioru danych.

4.2. Zestawy danych

Dla każdego z wcześniej przygotowanych zestawów danych sporządzono statystyki je opisujące. Wykorzystano do tego dwa skrypty. Jeden z nich oblicza średnią długość łańcucha znaków zawartych w pliku tekstowym. Jest on wykorzystywany do obliczenia średniej długości rekordów zawartych w plikach z poprawnymi zapisami oraz średniej długości rekordów z pliku, w którym zawarte są błędne zapisy. Na podstawie tych dwóch wartości obliczana jest średnia długość łańcucha zawartego w obydwu tych zbiorach. Drugi skrypt zwraca średnią odległość edycyjną pomiędzy parami łańcuchów znaków zawartymi w pliku zawierającym iloczyn kartezjański. Skrypt ten bazuje na implementacji zadania *MapReduce* z wykorzystaniem algorytmu *Levenshtein Distance*. W fazie *map* dla każdej pary obliczana jest odległość, następnie w fazie *reduce* obliczana jest średnia z wartości otrzymanych w fazie *map.* Statystki dla zbiorów danych przedstawiono w tabeli 2.

	Średnia odległość edycyjna	Średnia długość łańcucha dla poprawnych wyrażeń	Średnia długość łańcucha dla błędnych wyrażeń	Średnia długość łańcucha znaków
Pierwszy zbiór danych	8,916452	10,32584	8,99002	9,6579313
Drugi zbiór danych	20,57940	19,116	20,55554	19,835772

Tabela 2 Statystki zbiorów danych

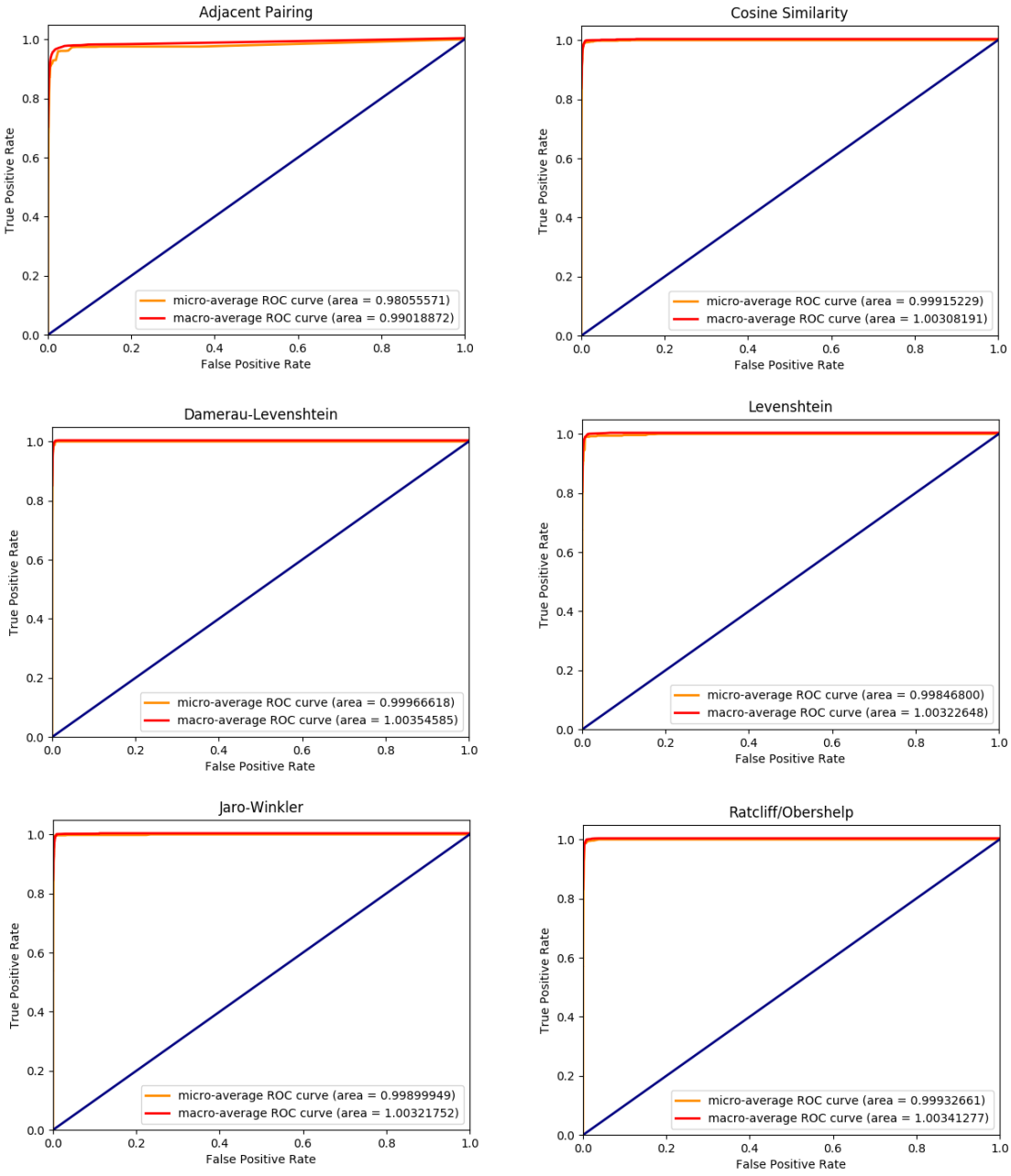
4.3. Pierwszy zestaw danych

Zestaw ten zbudowanych jest z pojedynczych słów. Ich średnia długość wynosi około 10 znaków. Słowa te są dosyć do siebie podobne, świadczy o tym niska średnia odległość edycyjna, sprawia to, że trudniej wyznaczyć odpowiednie grupy. Ważną cechą tego zestawu danych jest jego naturalność, czyli zawiera rzeczywiste literówki popełniane przez ludzi.

4.3.1. Krzywa ROC

Dla każdego z algorytmów przeprowadzono operację wykreślenia krzywej ROC. W tym celu przeprowadzono eksperyment z parametrem `SIMILARITY_THRESHOLD` równym -1.0, by otrzymać dla każdej klasy/grupy wynik algorytmu podobieństwa dla każdej możliwej kombinacji. Następnie korzystając z metodyki przedstawionej w [16], wykreślono dla każdego algorytmu dwie krzywe ROC oraz wyliczono AUC, wykorzystując do tego skrypt *generateRoc.py*. Skrypt ten przyjmuje jako parametry plik walidacyjny, plik z wynikami eksperymentu oraz nazwę pliku wyjściowego.

Z analizy przedstawionych na Rysunku 3 krzywych wynika, że badane algorytmy można traktować jako bardzo dobre metody do klasyfikacji badanych danych. Wskazuje na to sam kształt krzywej oraz bardzo wysoki wynik AUC dla każdego z algorytmów. Jedynym algorytmem, który otrzymał słabszy wynik to *Adjacent Pairing*, jest to jednak wartość z zakresu bardzo dobrych, w żaden sposób niedyskwalifikująca tego algorytmu w problemach takiej klasy. Wartości AUC nieco powyżej 1 dla wyznaczania krzywej ROC z wykorzystaniem metodyki *macro-average,* wynikają z niedokładności działań na liczbach zmiennoprzecinkowych



Rysunek 3 Krzywe ROC

.

4.3.2. Porównanie ze względu na poziom podobieństwa

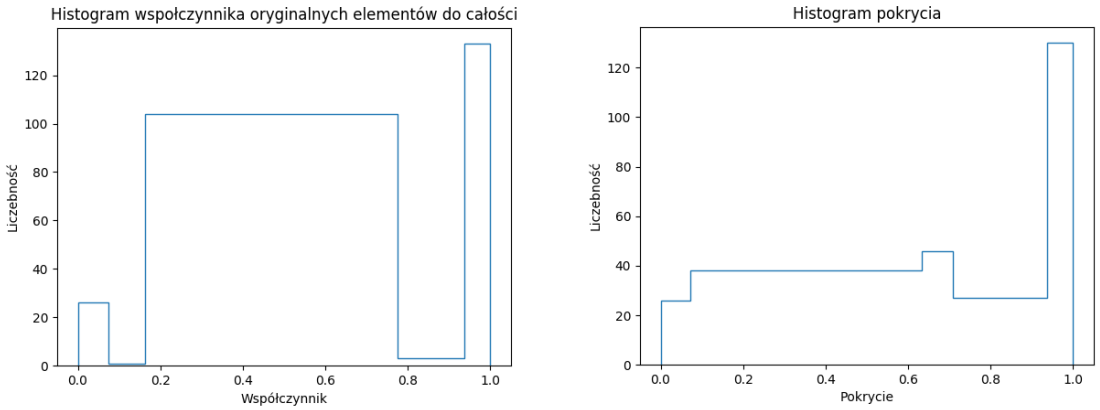
W ramach pracy przeprowadzono trzy eksperymenty, w których ustalono różne poziomy parametru `SIMILARITY_THRESHOLD`, który odpowiada za decyzję czy dane wyrazy są wystarczająco podobne by uznać je za należące do tej samej grupy. Badania przeprowadzono dla `SIMILARITY_THRESHOLD` równego:

- 0,7
- 0,8
- 0,9

Powyższe wartości ustalono tak, by otrzymane grupy zawierały jak najmniejszą liczbę wyrazów nadmiarowych, wyższe wartości mogą powodować natomiast bardzo niewiele zaakceptowanych wyrazów do grup. Dzięki tak zdefiniowanym przedziałom można było ustalić, jak to wpływa grupy otrzymywane w ramach eksperymentów oraz na statystki je opisujące.

4.3.2.1. Eksperyment przy minimalnym poziomie podobieństwa równym 0,7

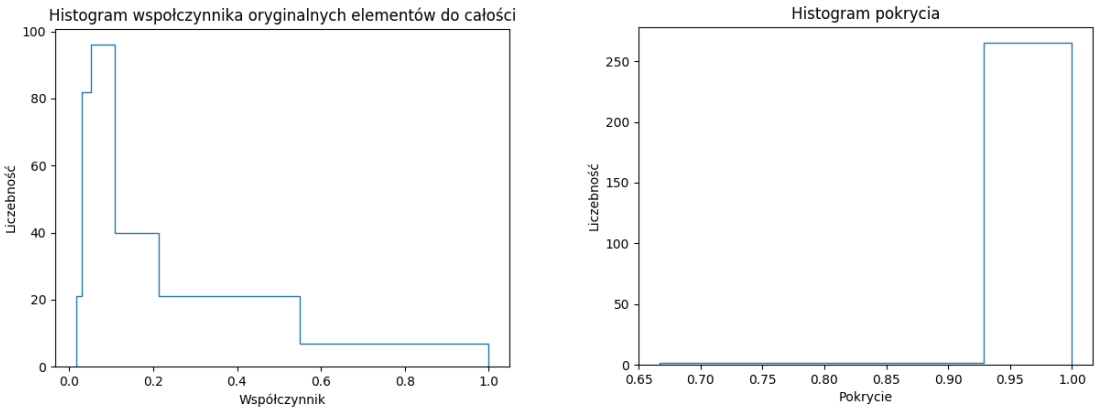
Wyniki dla algorytmu *adjacent pairing* są przedstawione na rysunku 4.



Rysunek 4 Histogramy dla algorytmu adjacent pairing przy minimalnym poziomie podobieństwa równym 0,7

Z powstałych histogramów można zauważyć, że ten algorytm, przy takiej wartości minimalnej, dobiera grupy bardzo podobne do zdefiniowanych w pliku walidującym. Najbardziej liczebną jest klasa osiągająca współczynnik oryginalnych elementów do całości równy 1, przy mierze pokrycia również równej 1. Zaś w większości przypadków nadmiarowe słowa nie stanowią więcej niż 50% liczebności nadmiarowych grup.

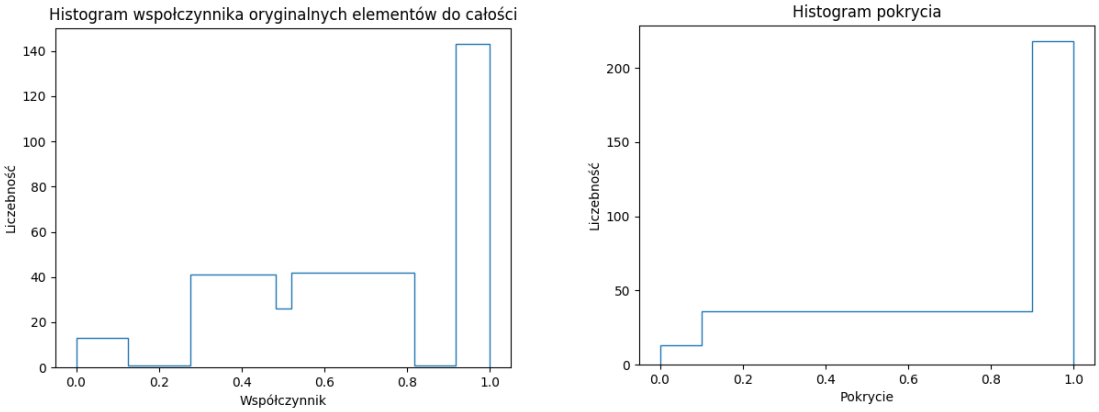
Wyniki dla algorytmu *cosine similarity* przedstawione są na rysunku 5.



Rysunek 5 Histogramy dla algorytmu cosine similarity przy minimalnym poziomie podobieństwa równym 0,7

Przy tak dobranym minimalnym poziomie, algorytm ten definiuje bardzo wiele wyrazów jako podobne, o czym świadczy duża liczebność dla klas z małą wartością na pierwszym z histogramów przestawionym na rysunku 5. Algorytm ten niemal nie pominął żadnych wyrazów, które miały się znaleźć w odpowiednich grupach, świadczy o tym drugi z histogramów.

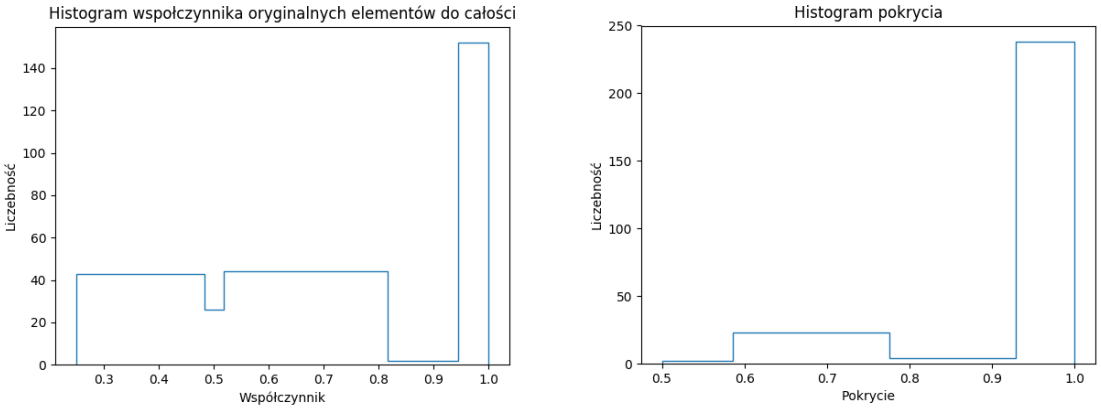
Wyniki dla algorytmu *Levenshtein* przedstawione są na rysunku 6.



Rysunek 6 Histogramy dla algorytmu Levenshtein przy minimalnym poziomie podobieństwa równym 0,7

Algorytm przy tak dobranym minimalnym poziomie podobieństwa wskazuje na podobne grupy do tych znajdujących się w pliku walidującym. Większość grup nie zawiera także zbyt dużo nadmiarowych elementów.

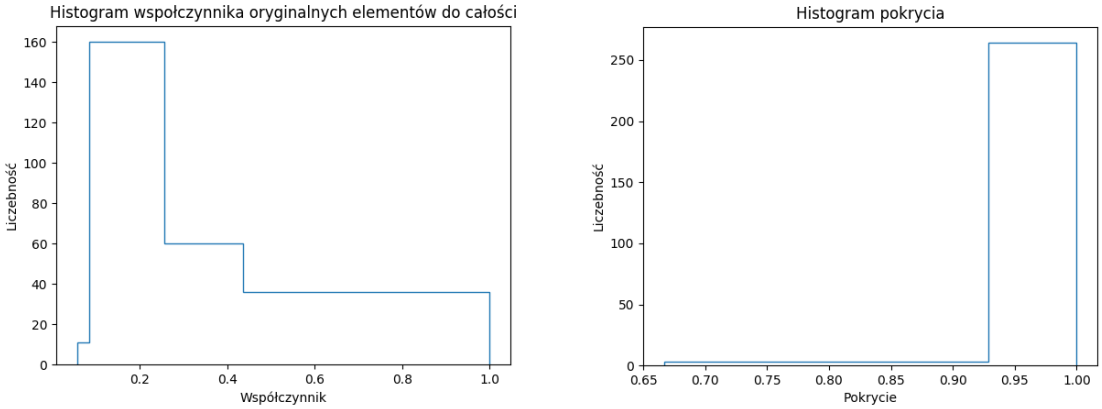
Wyniki dla algorytmu *Damerau-Levenshtein* przedstawione są na rysunku 7.



Rysunek 7 Histogramy dla algorytmu Damerau-Levenshtein przy minimalnym poziomie podobieństwa równym 0,7

Podobnie jak algorytm Levenshteina wyniki wskazują na duże podobieństwo powstałych grup do tych oryginalnych. Zwiększony został ogólny wskaźnik pokrycia, kosztem jednak większej liczby nadmiarowo przypisanych elementów.

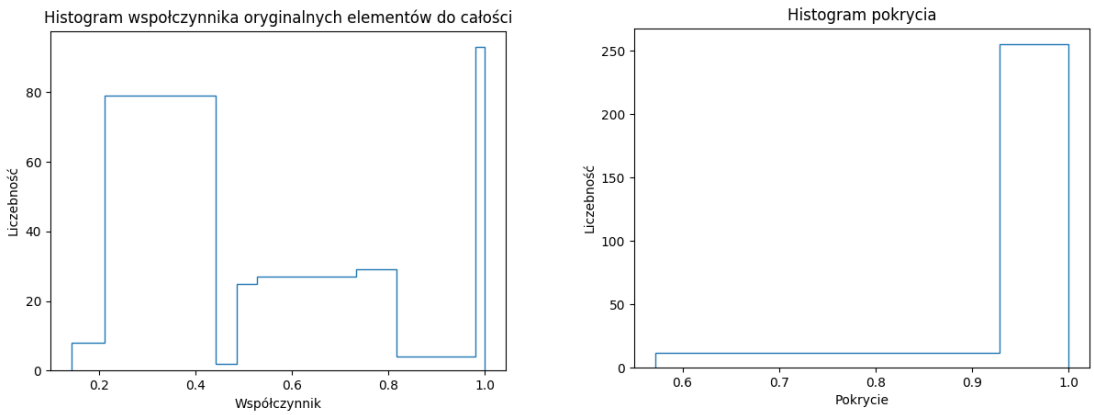
Wyniki dla algorytmu *Jaro-Winkler* przedstawia rysunek 8.



Rysunek 8 Histogramy dla algorytmu Jaro-Winkler przy minimalnym poziomie podobieństwa równym 0,7

Przy tak dobranym współczynniku algorytm przypisuje do grup znaczne liczby nadmiarowych wyrazów, jak i niemal wszystkie wyrazy zawarte oryginalnie w grupach.

Wyniki dla algorytmu *Ratcliff/Obershelp* przedstawione są na rysunku 9.

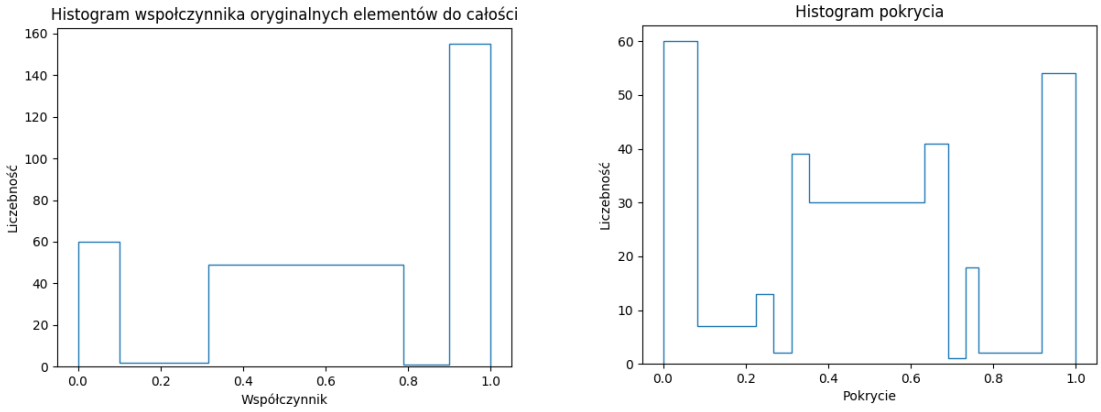


Rysunek 9 Histogramy dla algorytmu Ratcliff/Obershelp przy minimalnym poziomie podobieństwa równym 0,7

Wskaźnik pokrycia niemal w wszystkich przypadkach wynosi niemal 1, co oznacza, że wszystkie oryginalne elementy znalazły się w grupach utworzonych w ramach eksperymentu. Interesujący jest fakt bardzo dużego rozrzutu dla stosunku liczby oryginalnych elementów do liczby całości elementów grupy. Bardzo wiele przypadków ma ten współczynnik równy niemal 1, a druga największą klasa przypadków ma ten stosunek bardzo niski.

4.3.2.2. Eksperyment przy minimalnym poziomie podobieństwa równym 0,8

Wyniki dla algorytmu *adjacent pairing* przedstawia rysunek 10.

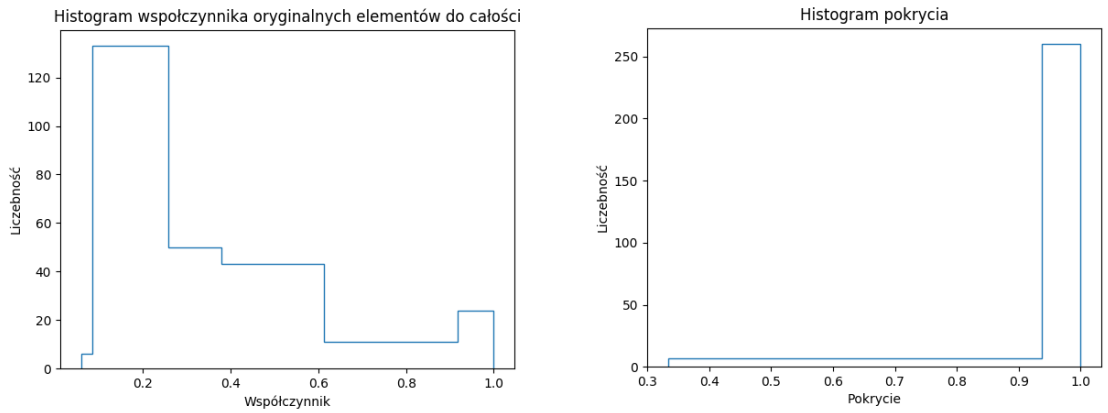


Rysunek 10 Histogramy dla algorytmu adjacent pairing przy minimalnym poziomie podobieństwa równym 0,8

Na powstałych histogramach można zauważyć znaczne zróżnicowanie współczynnika pokrycia i raczej korzystny rozkład liczebności klas stosunku oryginalnych elementów do całości. Wynika z tego, że algorytm raczej nie przydziela do grup elementów,

które wcześniej do niej nie należały. Gorzej wygląda sytuacja dla współczynnika pokrycia. Algorytm powoduje powstanie bardzo licznej klasy, która nie zawiera żadnych elementów oryginalnych. Jednak drugą najliczniejszą klasą jest taka, która zawiera wszystkie elementy zawarte w oryginalnej grupie. Duża jest liczba rekordów zawierająca pokrycie na poziomie 50%.

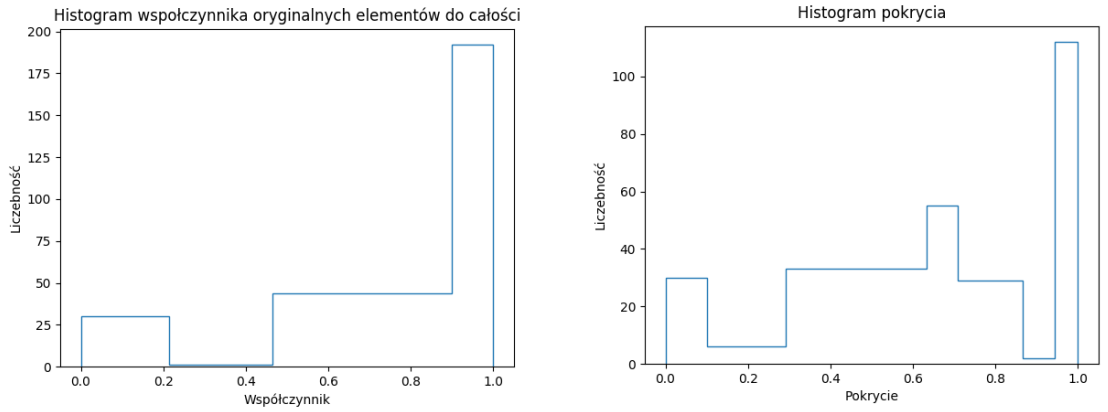
Wyniki dla algorytmu *cosine similarity* przedstawia rysunek 11.



Rysunek 11 Histogramy dla algorytmu cosine similarity przy minimalnym poziomie podobieństwa równym 0,8

Algorytm ten przypisuje znaczną liczbę nadmiarowych wyrazów do grup, nie pomija wyrazów oryginalnie ulokowanych w tych grupach. Z tych wyników można wywnioskować, że wartość minimalnego poziomu podobieństwa dla tego algorytmu jest zbyt mała, gdyż akceptuje on jako podobne bardzo szerokie spektrum elementów.

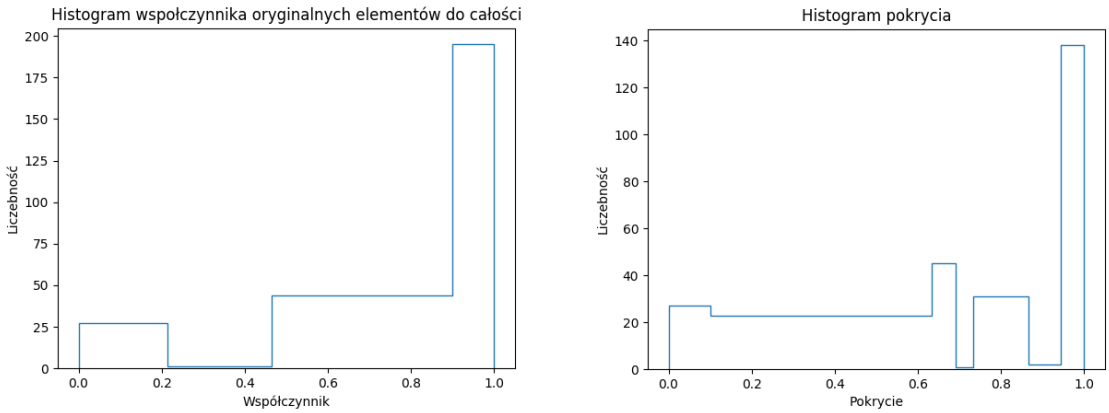
Wyniki dla algorytmu *Levenshtein* przedstawione są na rysunku 12.



Rysunek 12 Histogramy dla algorytmu Lebenstein przy minimalnym poziomie podobieństwa równym 0,8

Przy tak dobranym poziome podobieństwa algorytm ustala jako podobne głównie wyrazy należące do oryginalnej grupy znajdującej się w pliku walidacyjnym. Jednak część wyrazów zawartych w ów grupach jest pomijana.

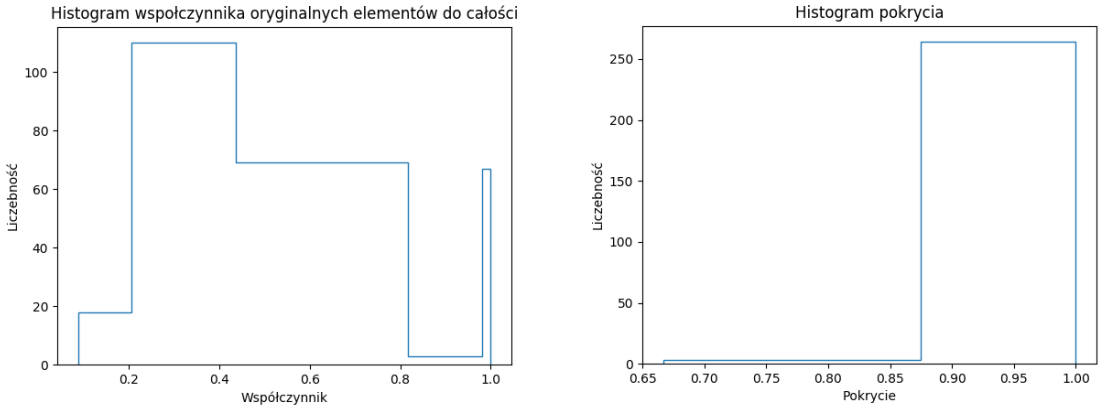
Wyniki dla algorytmu *Damerau-Levenshtein* przedstawia rysunek 13.



Rysunek 13 Histogramy dla algorytmu Damerau-Levenshtein przy minimalnym poziomie podobieństwa równym 0,8

Otrzymane histogramy są bardzo podobne do tych otrzymanych z Levenshteina. Widać jednak widać przyrost wskaźnika pokrycia. Algorytm ten lepiej sprawdził się w przypisaniu do grup wyrazów, które miały znajdować się w danej grupie, według zbioru walidującego.

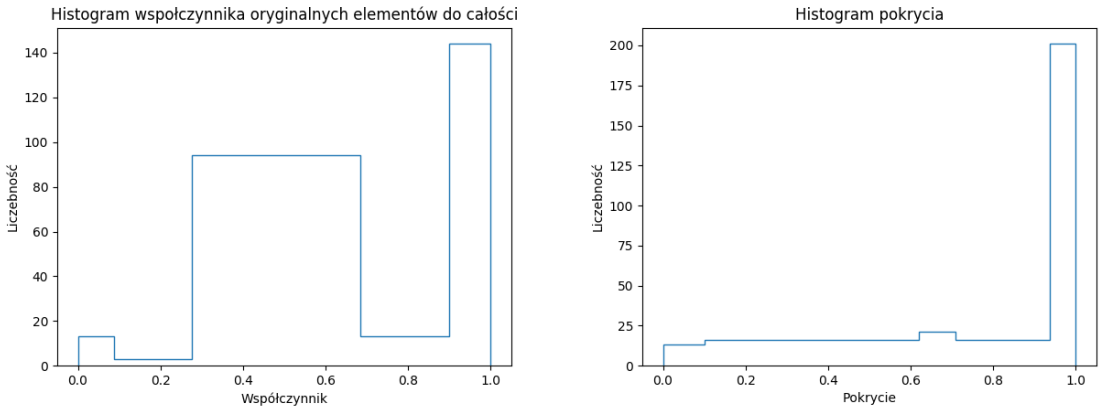
Wyniki dla algorytmu Jaro-Winkler przedstawione są rysunku 14.



Rysunek 14 Histogramy dla algorytmu Jaro-Winkler przy minimalnym poziomie podobieństwa równym 0,8

Algorytm przypisuje znaczną część nadmiarowych elementów do grup, zachowując przy tym wszystkie elementy jakie w danej grupie miały się znaleźć. Winna jest temu zbyt mała wartość progu podobieństwa.

Wyniki dla algorytmu *Ratlcliff/Obershelp* przedstawia rysunek numer 15.

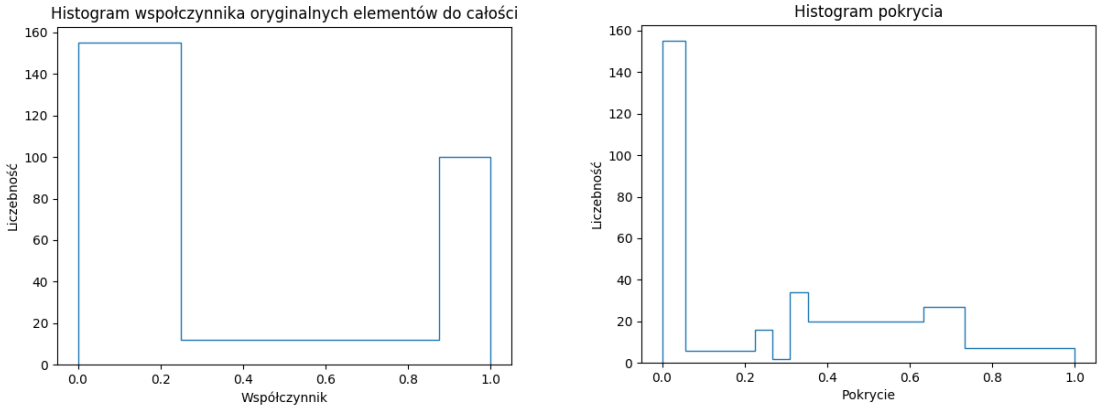


Rysunek 15 Histogramy dla algorytmu Ratcliff/Obershelp przy minimalnym poziomie podobieństwa równym 0,8

Z otrzymanych wyników można wywnioskować, że algorytm w większości przypadków rozpoznaje, który element powinien należeć do tworzonych grup, ale dopasowuje również dużą liczbę nadmiarowych wyrazów.

4.3.2.3. Eksperyment przy minimalnym poziomie podobieństwa równym 0,9

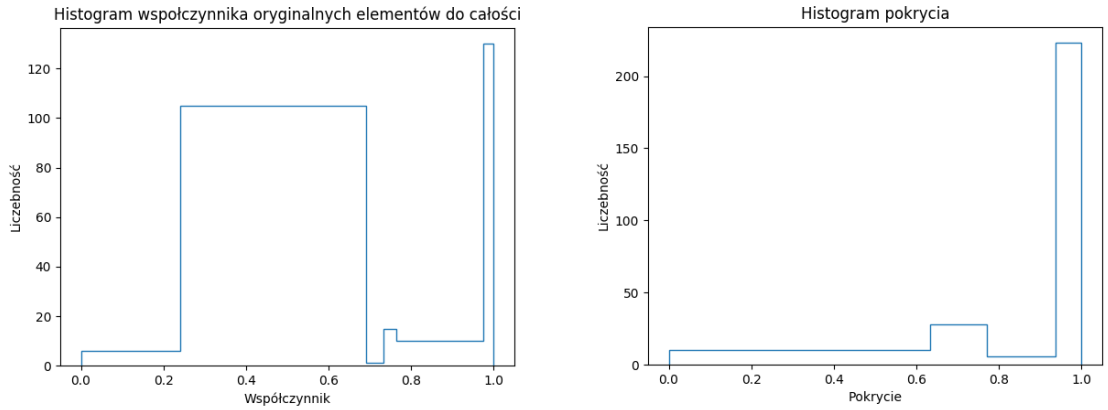
Wyniki dla algorytmu *adjacent pairing* przedstawia rysunek 16.



Rysunek 16 Histogramy dla algorytmu adjacent pairing przy minimalnym poziomie podobieństwa równym 0,9

Przy tak dobranym poziomie algorytm niemal zawsze stwierdza, że wyrazy nie są wystarczająco podobne. Świadczy o tym liczebność klasy o zerowym wskaźniku pokrycia.

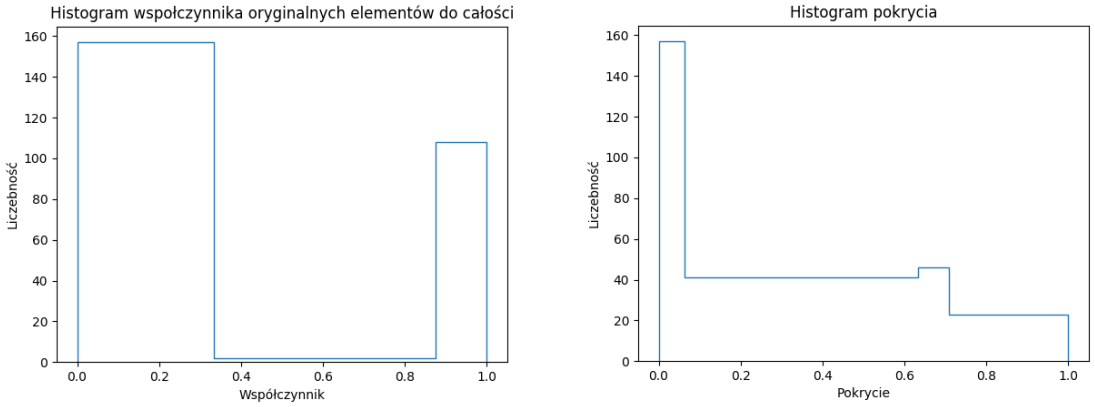
Wyniki dla algorytmu *cosine similarity* przedstawiono na rysunku 17.



Rysunek 17 Histogramy dla algorytmu cosine similarity przy minimalnym poziomie podobieństwa równym 0,9

W wynikach widać znaczną poprawę w stosunku do pozostałych wyników otrzymanych dla tego algorytmu. Wyodrębniła się znaczna klasa grup zawierających wysoki wskaźnik pokrycia oraz wysoki stosunek oryginalnych elementów do całości.

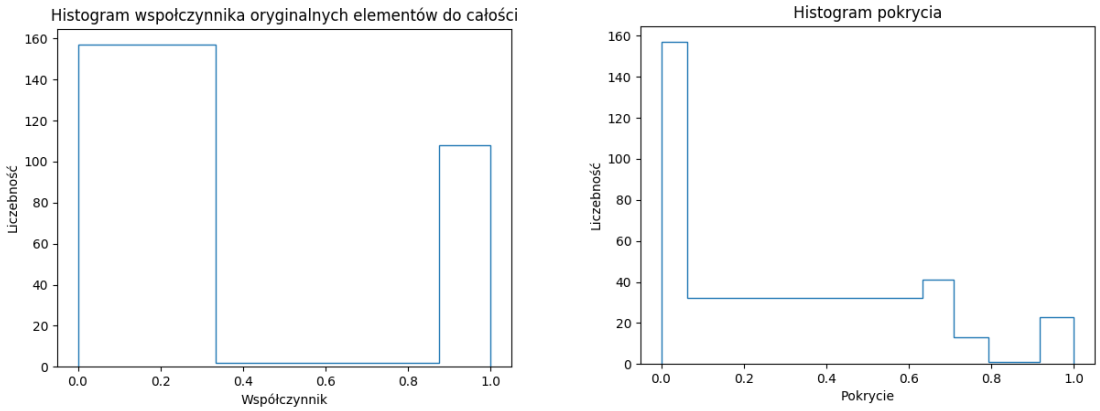
Wyniki dla algorytmu *Levenshtein* przedstawione są na rysunku 18.



Rysunek 18 Histogramy dla algorytmu Levenshtein przy minimalnym poziomie podobieństwa równym 0,9

Przy wysokim minimalnym poziomie skuteczność tego algorytmu znacznie spadła. Z obu histogramów można wyczytać, że elementy rzadko spełniały wymaganie minimalnego podobieństwa.

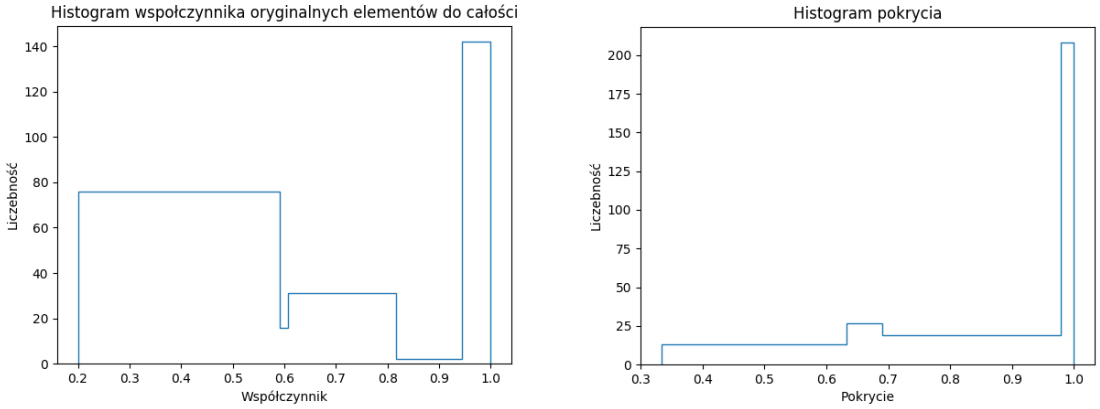
Wyniki dla algorytmu *Damerau-Levenshtein* przedstawia rysunek 19.



Rysunek 19 Histogramy dla algorytmu Damerau-Levenshtein przy minimalnym poziomie podobieństwa równym 0,9

Otrzymano wyniki niemal identyczne jak przy algorytmie Levenshtein. Rzadko był spełniany warunek minimalnego poziomu podobieństwa.

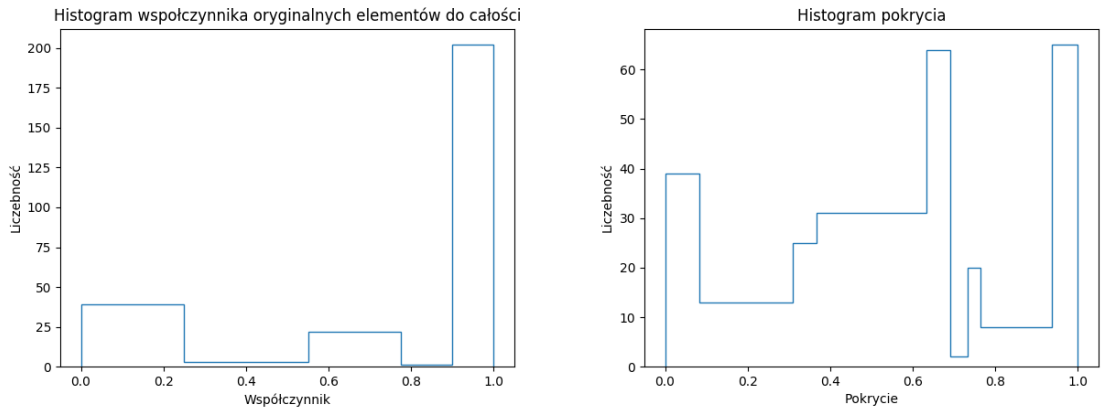
Wyniki dla algorytmu *Jaro-Winkler* przedstawione są na rysunku 20.



Rysunek 20 Histogramy dla algorytmu Jaro-Winkler przy minimalnym poziomie podobieństwa równym 0,9

Dzięki podwyższeniu minimalnego poziomu podobieństwa znacznie zmalała liczba przydzielanych nadmiarowych wyrazów, jednak dalej jest to dość spora klasa. Prawdopodobnie dalsze zwiększenie progu podobieństwa podniosłoby wartość współczynnika oryginalnych elementów do całości.

Wyniki dla algorytmu *Ratcliff/Obershelp* przedstawia rysunek 21.

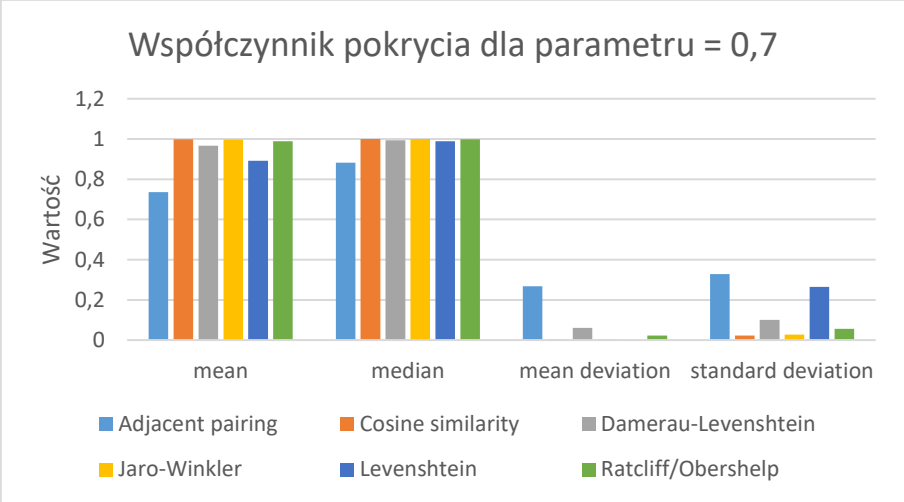


Rysunek 21 Histogramy dla algorytmu Ratcliff/Obershelp przy minimalnym poziomie podobieństwa równym 0,9

Dla takiego poziomu minimalnego podobieństwa otrzymano bardzo nieregularny histogram pokrycia. Można wyróżnić dwie duże klasy, jedna z nich to ta o wskaźniku równym 1, a druga 0,6. W większości przypadków algorytm nie przydzielał wyrazów nie znajdujących się w oryginalnych grupach.

4.3.2.4. Grupowe wyniki

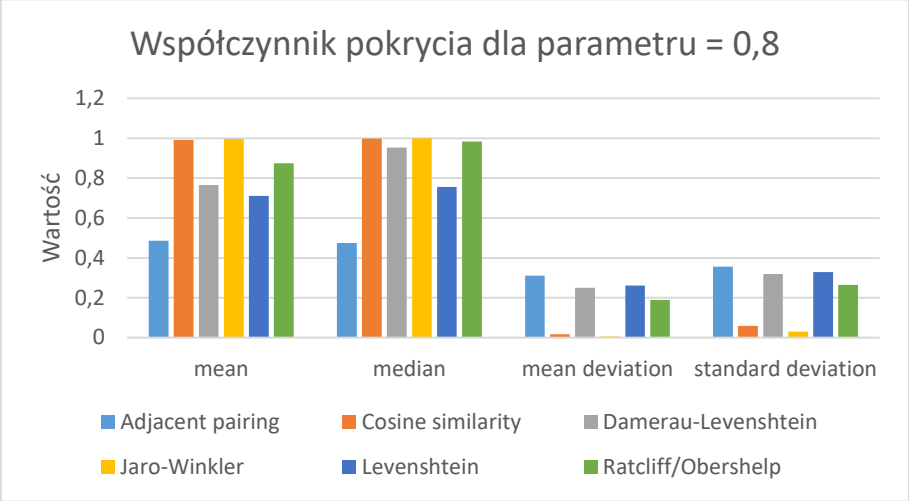
Analizę wartości miary pokrycia (*coverage*) dla różnych progów podobieństwa przedstawiają rysunki 22, 23 i 24.



Rysunek 22 Wykres kolumnowy zbiorczy miary pokrycia(coverage) dla minimalnego poziomu podobieństwa równego 0,7

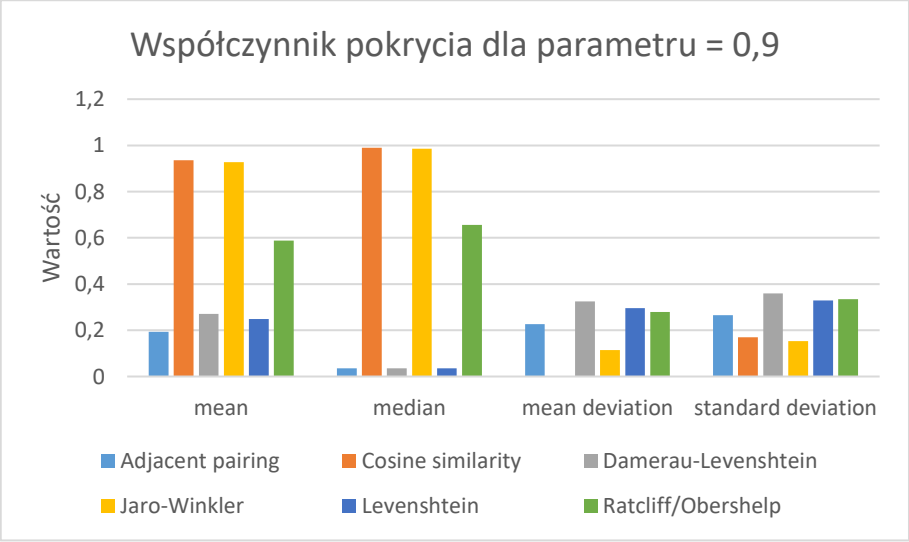
Praktycznie wszystkie rozwiązania przy takim minimalnym poziomie, osiągnęły prawie, że 100% stopień pokrycia dla grup. Jedyny algorytm, który zwrócił inne wyniki jest

adjacent pairing. Można zauważyć, że miary rozproszenia, to jest obydwa odchylenia są dla niego dosyć wysokie. Może to wskazywać na nieregularność algorytmu.



Rysunek 23 Wykres kolumnowy zbiorczy miary pokrycia(coverage) dla minimalnego poziomu podobieństwa równego 0,8

Przy większym minimalnym poziomie wszystkie algorytmy zwracają bardziej zróżnicowane wyniki, przedstawia to rysunek 23. Świadczy o tym wzrost miar rozproszenia. Dla *cosine similarity* i *Jaro-Winkler* miary te pozostały na podobnym poziomie. Można zaobserwować spadek wartości średniej dla większości rozwiązań.

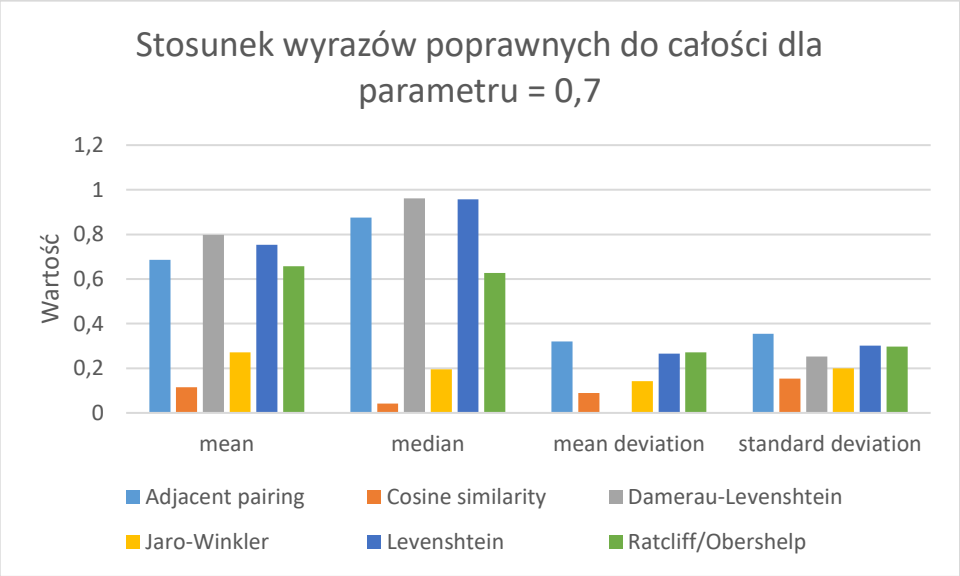


Rysunek 24 Wykres kolumnowy zbiorczy miary pokrycia(coverage) dla minimalnego poziomu podobieństwa równego 0,9

Największy przebadany poziom podobieństwa spowodował znaczny średni spadek współczynnika pokrycia, co przedstawia rysunek 24. Jedynie algorytmy *Cosine similarity* i *Jaro-Winkler* zachowały wysoką jego wartość, przy niskim poziomie rozproszenia

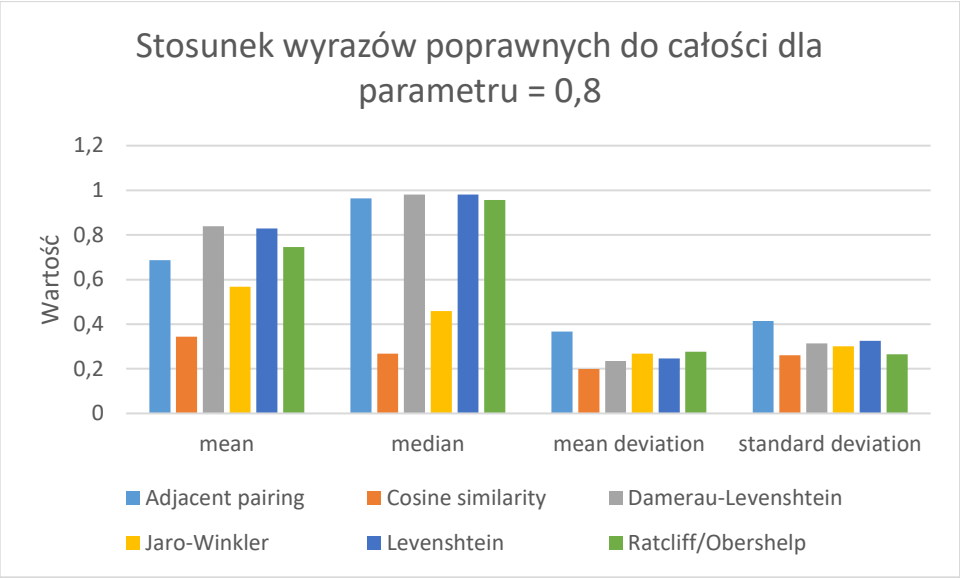
wyników. Duży wskaźnik zachował również algorytm *Ratcliff/Obershelp* jednak przy wyższych miarach rozproszenia.

Analizę wartości stosunku poprawnych wyrazów do całości dla różnych progów podobieństwa przedstawiają rysunki 25, 26 i 27.



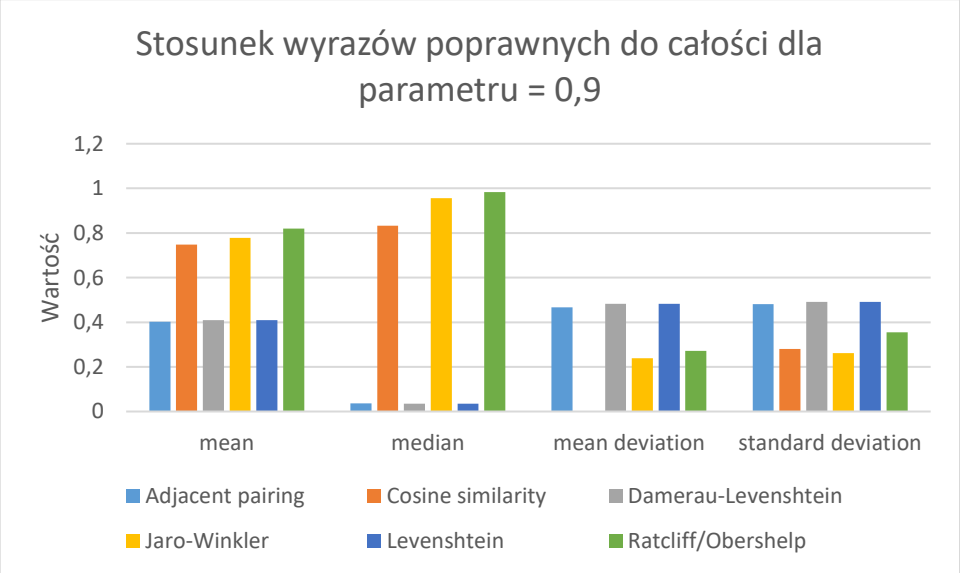
Rysunek 25 Wykres kolumnowy zbiorczy stosunku wyrazów poprawnych do całości dla minimalnego poziomu podobieństwa równego 0,7

Wysoki stosunek poprawnych wyrazów do całości posiadają algorytmy *Levenshtein*, *Damerau-Levenshtein* i *Adjacent Pairing*, można to zaobserwować na rysunku numer 25. Miary rozproszenia są podobnego poziomu dla każdego z badanych algorytmów. Algorytmy *Cosine similarity* i *Jaro-Winkler* przypisują dużą liczbę nadmiarowych wyrazów przy takim poziomie minimalnego podobieństwa.



Rysunek 26 Wykres kolumnowy zbiorczy stosunku wyrazów poprawnych do całości dla minimalnego poziomu podobieństwa równego 0,8

Po zwiększeniu minimalnego poziomu znacznie zwiększyła się mediana dla algorytmu *Ratcliff/Obershelp*, przy podobnej wartości średniej, można to wywnioskować z wykresu przedstawionego na rysunku 26. Oznacza to zdecydowanie zmniejszenie ilości wartości skrajnych w stosunku do poprzedniego eksperymentu. Pozostałe wartości pozostały na takim samym poziomie, oprócz wartości dla algorytmów *Jaro-Winkler* i *Cosine similarity*, które przydzielają nieco mniej nadmiarowych elementów do grup.



Rysunek 27 Wykres kolumnowy zbiorczy stosunku wyrazów poprawnych do całości dla minimalnego poziomu podobieństwa równego 0,9

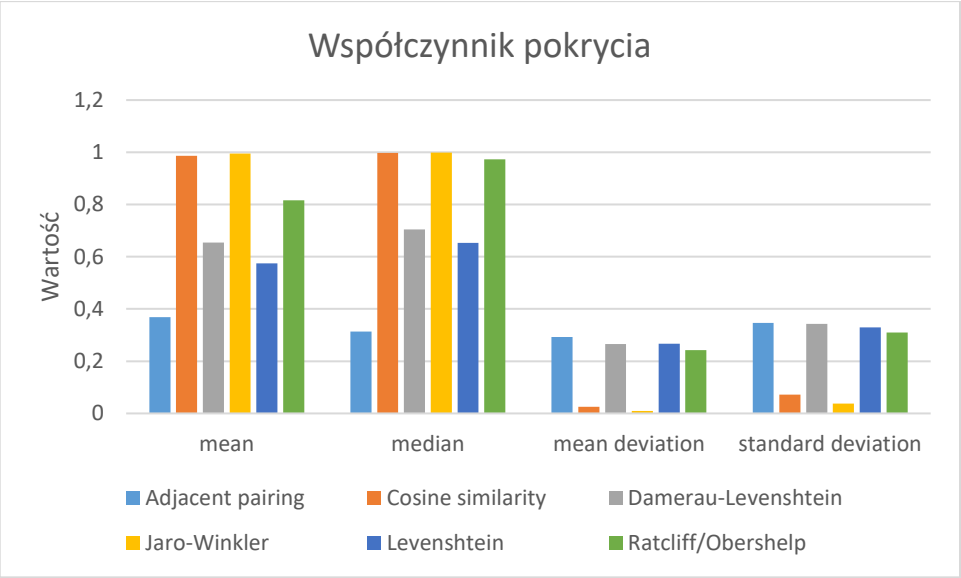
W ostatnim eksperymencie, którego wyniki przedstawia rysunek 27, można zaobserwować ogólny wzrost miar rozproszenia oraz znaczy spadek badanej wartości dla algorytmów *Adjacent Pairing*, *Levenshtein* oraz *Damerau-Levenshtein*. Jest to spowodowane bardzo wysokim wymaganym poziomem podobieństwa, przez co bardzo mało wyrazów spełnia ten warunek. Można zauważyć też dalszą poprawę badanej wartości dla algorytmów *Cosine similarity*, *Jaro-Winkler* oraz *Ratcliff/Obershelp.*

4.3.3. Porównanie ze względu na długość łańcucha znaków

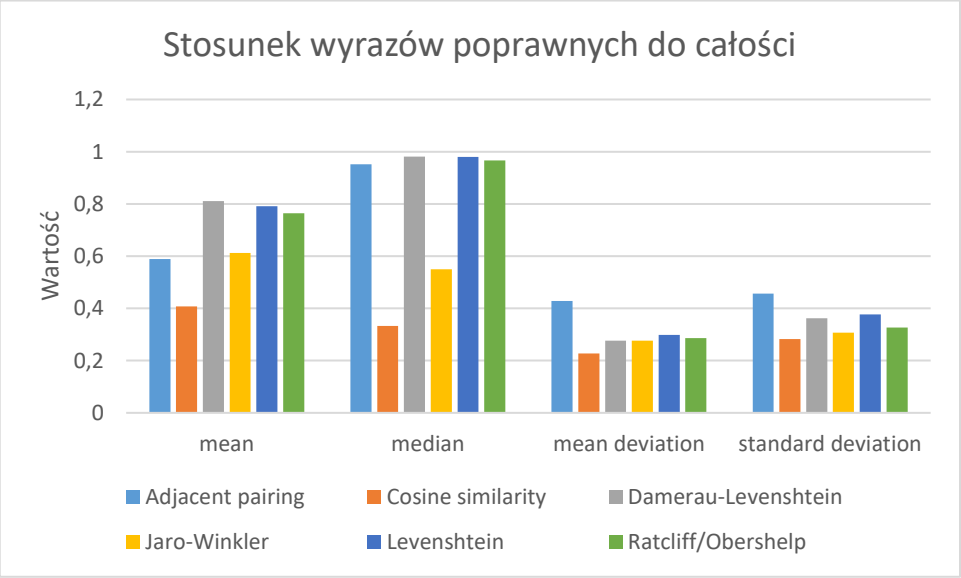
Przeprowadzono także eksperyment mający na celu zbadanie zależności pomiędzy długością słowa a wynikami grupowania przez algorytm. Do eksperymentu użyto minimalny poziom podobieństwa równy 0,8. Żeby móc przebadać taką zależność napisany został skrypt, który przyjmuje za parametry: plik wejściowy – jest to plik otrzymany w jednym z wcześniej przeprowadzonych eksperymentów, a dokładnie dane otrzymane z jednego z zadań *MapReduce*. Skrypt ten przyjmuje dwa parametry: nazwy plików wyjściowych i wartość filtra. Skrypt ten powoduje podzielenie zbioru wejściowego według wartości filtra. Wartość ta określa liczbę znaków poprawnie zapisanego słowa, według którego zostanie podzielony zestaw danych. To jest przykładowo dla wartości filtra 10 zostaną utworzone dwa nowe zestawy – jeden zawierający grupy dla poprawnie zapisanych wyrazów dłuższych niż 10 znaków oraz drugi zawierający pozostałe.

W ramach tego eksperymentu wartość filtra ustalono na 10 i podobnie jak we wcześniejszych eksperymentach wygenerowano dane dla każdego algorytmu z osobna. Wartość filtra 10 dobrano tak, by podzielić badany zbiór mniej więcej na połowę, jest to także wartość średniej długości słowa. Aby móc wykonać późniejszą analizę utworzono nowe zbiory walidujące, które powstały poprzez przefiltrowanie oryginalnego zbioru walidującego poprzez skrypt filtrujący. Do wygenerowania statystyk użyto tego samego skryptu, co w poprzednich eksperymentach, tym razem wykorzystując odpowiednio przefiltrowane wyniki eksperymentów oraz nowe zbiory walidujące.

Dla poprawnie zapisanych wyrazów o długości nie przekraczającej 10 znaków otrzymano wartości przedstawione na wykresach przedstawionych na rysunkach 28 i 29.



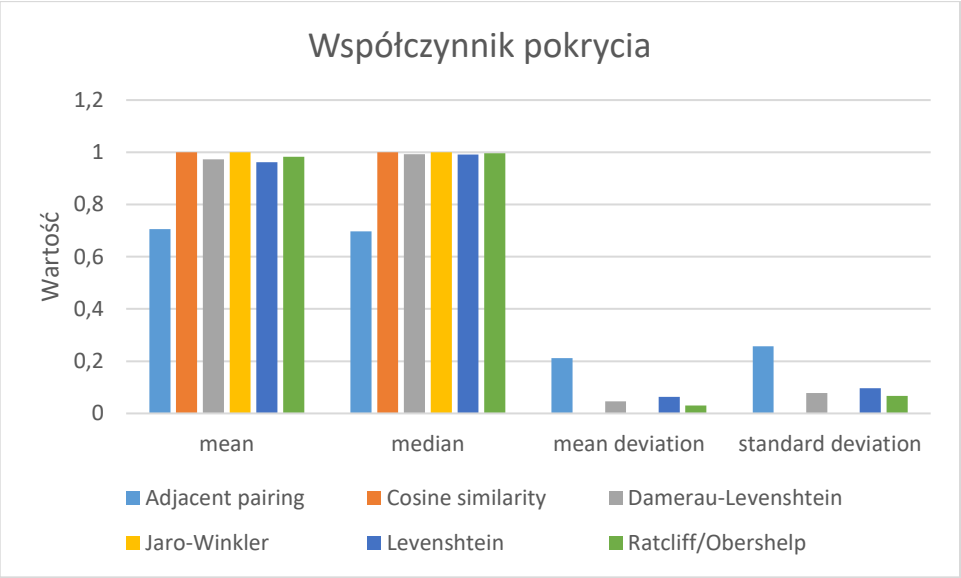
Rysunek 28 Wykres kolumnowy zbiorczy miary pokrycia(coverage) dla poprawnych słów krótszych niż 11 znaków



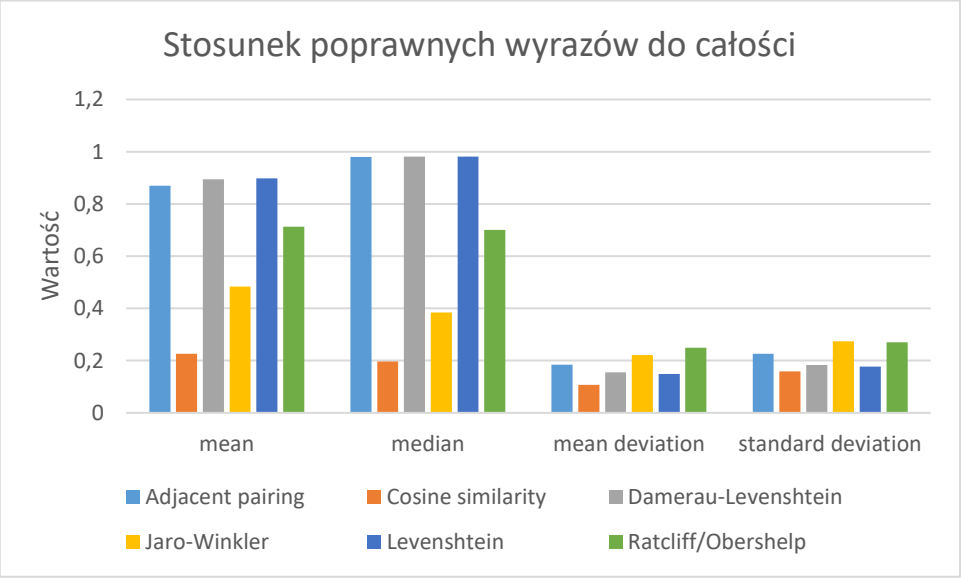
Rysunek 29 Wykres kolumnowy zbiorczy miary stosunku wyrazów poprawnych do całości dla poprawnych słów krótszych niż 11 znaków

Dla krótszych wyrazów najlepsze statystyki osiągnął algorytm *Ratcliff/Obershelp*. Algorytmy *Cosine similarity* oraz *Jaro-Winkler* mimo wysokiego pokrycia oryginalnych grup, posiadają tendencję do nadmiarowego przypisywania słów do grup.

Dla poprawnie zapisanych wyrazów o długości przynajmniej 11 znaków otrzymano wartości przedstawione na rysunkach 30 i 31.



Rysunek 30 Wykres kolumnowy zbiorczy miary pokrycia(coverage) dla poprawnych słów dłuższych niż 10 znaków



Rysunek 31 Wykres kolumnowy zbiorczy miary stosunku wyrazów poprawnych do całości dla poprawnych słów dłuższych niż 10 znaków

Przy dłuższych wyrazach widać znaczącą poprawę wartości pokrycia dla wszystkich algorytmów, przy bardzo niskich miarach rozproszenia. Natomiast stosunek poprawnych wyrazów do całości uległ zmianie negatywnie dla algorytmu *Ratcliff/Obershelp*, dopisuje on do grup wyrazy, które nie znajdowały się tam początkowo. Znacznie lepiej z dłuższymi wyrazami sprawuje się algorytm *Adjacent pairing*. Spadki w stosunku poprawnych wyrazów do całości zanotowały algorytmy *Cosine similarity* oraz *Jaro-Winkler*. Podobny stosunek

poprawnych wyrazów do całości, przy wzroście pokrycia zachowały *algorytmy Damerau-Levenshtein* i *Levenshtein*, co wskazuje na ich lepsze właściwości przy dłuższych łańcuchach znaków.

4.3.4. Metoda łączona

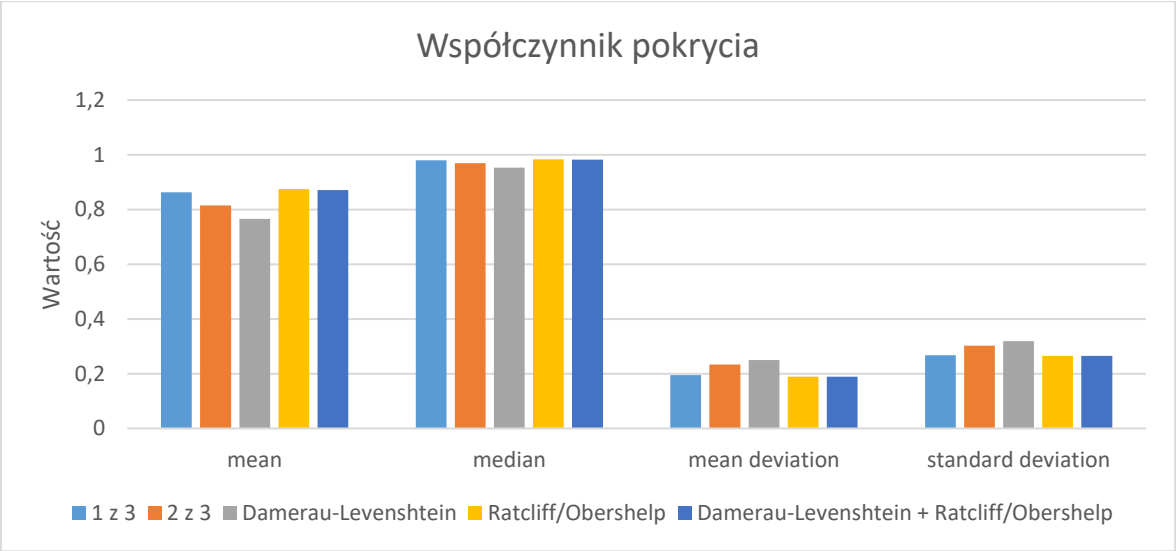
W ramach kolejnego eksperymentu przebadano trzy sposoby budowania grup podobnych wyrazów, bazujące na wynikach poprzednich eksperymentów.

Pierwsza z metod wykorzystuje algorytmy *Ratcliff/Obershelp* oraz *Damerau-Levenshtein*. Bazuje ona na eksperymencie badającym zależność skuteczności algorytmów od długości słowa. W ramach tego eksperymentu wykryto zależność, że dla wyrazów dłuższych niż 10 znaków znacznie lepiej grupuje wyrazy algorytm *Damerau-Levenshtein*, a odwrotną zależność wykazał algorytm *Ratcliff/Obershelp.* Zmodyfikowano implementację zadania tak, by poprawne wyrazy o długości mniejszej od 11 porównywano za pomocą algorytmu *Ratcliff/Obershelp,* a pozostałe za pomocą algorytmu Damerau-*Levenshtein.* W tym celu nowa implementacja zawiera 3 kroki. Pierwszy z tych kroków otrzymuje linię z pliku wejściowego, następnie sprawdza długość słowa poprawnego. Jeżeli jest dłuższe niż 11 znaków emitowana jest para klucz-wartość, o kluczu „dl" i wartości zawierającej dwa słowa do porównania. W przeciwnym przypadku wartość klucza ustalana jest na „rat". W tym kroku występuje tylko zmodyfikowana faza *map,* w fazie *reduce* nie jest wykonywana żadna operacja. Następny krok wykorzystuje wartość wyemitowanego wcześniej klucza do wybrania odpowiedniego algorytmu do badania podobieństwa. Jeżeli ta wartość to „dl" wybierany jest algorytm *Damerau-Levenshtein*, w drugim przypadku wybierany jest *Ratcliff/Obershelp.* Następnie tak jak w poprzednich zadaniach następuje emisja pary klucz-wartość, gdzie kluczem jest poprawny wyraz, a wartością para wyraz dopasowany i wynik ich podobieństwa. Następnie faza *reduce* tego kroku jest dokładnie taka sama jak w poprzednich zdaniach, a trzeci krok odpowiada krokowi drugiemu z poprzednich zadań. W ramach eksperymentu ustalono próg podobieństwa na 0,8, taki sam jak przy badaniu zachowania algorytmów w zależności od długości słowa.

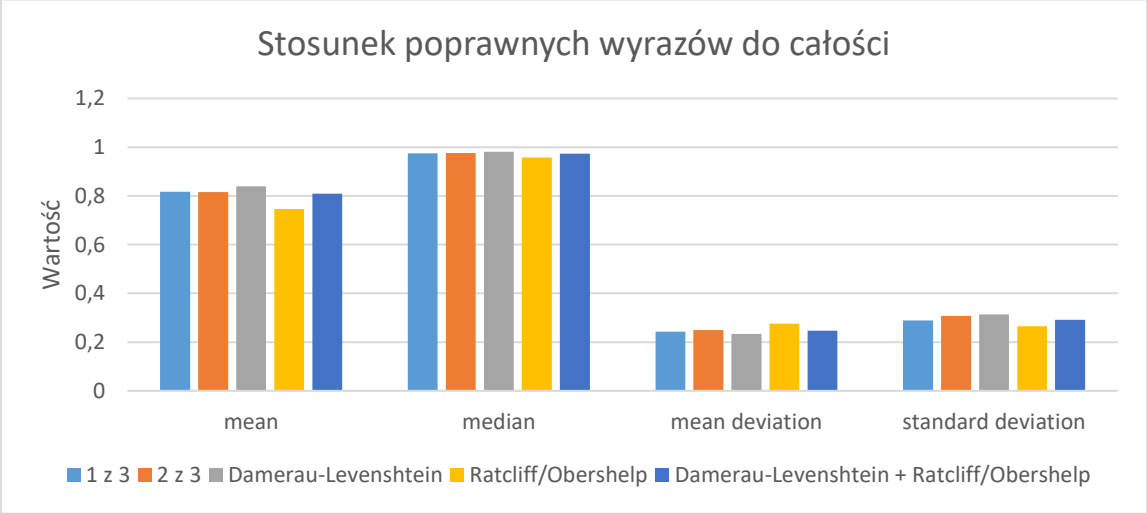
Druga i trzecia z metod wykorzystuje jako bazę pierwszą metodę. Różnią się tym, że po ustaleniu podobieństwa przez algorytm *Damerau-Levenshtein* lub *Ratcliff/Obershelp*

emitowane są trzy pary klucz wartość. Emitowane pary zawierają taką samą wartość – dwa wyrazy do porównania, różnią się tylko kluczami, które przybierają odpowiednio wartości „ap", „cos" i „jw". Wartości te wskazują jaki algorytm ma zostać użyty w kolejnym kroku. Do konstrukcji tej metody wybrano te algorytmy ze względu na ich specjalne właściwości. *Adjacent Pairing* – klucz „ap", wybrano ze względu na to, że szczególnie na jego wartość wpływa ułożenie znaków w badanym słowie, dzieje się tak, gdyż algorytm buduje pary ze sąsiednich znaków. Algorytm *Cosine Similarity* - klucz „cos", całkowicie pomija znaczenie kolejności znaków. Ostatni algorytm *Jaro-Winkler* - klucz „jw", mocno premiuje zgodność początkowych znaków w badanym ciągu znaków. Progi minimalnego podobieństwa określono na podstawie poprzednich eksperymentów z różnymi progami. I tak dla algorytmu *Adjacent Pairing* wybrano próg 0,8, a dla *Cosine Simalarity* i *Jaro-Winkler* próg równy 0,9. W tym kroku, jeżeli dane słowo spełnia próg emitowana jest para klucz-wartość, o kluczu, który jest złączeniem dwóch badanych wyrazów oraz wartość zawierająca dwa badane słowa i wartość „True", jeżeli zaś nie spełnia minimalnego progu podobieństwa jest to wartość „False". Kolejną fazą jest *reduce,* dzięki takiej budowie klucza do każdego *reducer* trafiają trzy wartości dla każdych badanych dwóch słów. Są to rezultaty otrzymane dla tych trzech wybranych algorytmów. Dokonywane jest w tej fazie głosowanie. W pierwszej z metod do zakwalifikowania wyrazów jako podobne potrzebna jest jedna wartość „True", a w drugiej wymagane są dwie wartości „True". Następnie w obydwóch metodach jeżeli wyrazy uznano za podobne, emitowana jest para klucz-wartość, gdzie kluczem jest poprawny wyraz, a wartością wyraz dopasowany. W kolejnej fazie map nie jest wykonywana żadna operacja, emitowana jest wartość otrzymywana na wejściu. Następna faza *reduce* jest dokładnie taka sama jak w poprzednich zdaniach, a ostatni krok odpowiada ostatniemu krokowi z poprzednich zadań.

Otrzymane wyniki przedstawiono na wykresach na rysunkach 32 i 33.



Rysunek 32 Wykres kolumnowy zbiorczy miary pokrycia(coverage) dla metod łączonych



Rysunek 33 Wykres kolumnowy zbiorczy miary stosunku wyrazów poprawnych do całości dla metod łączonych

Dzięki zastosowaniu metody pierwszej, czyli połączenia algorytmów *Ratcliff/Obershelp* oraz *Damerau-Levenshtein*, udało się osiągnąć lepszy współczynnik pokrycia niż dla dwóch z tych algorytmów osobno. Stosunek poprawnych wyrazów do całości uległ nieznacznej poprawie w stosunku do algorytmu *Ratcliff/Obershelp* i nieznacznemu pogorszeniu w stosunku do algorytmu *Damerau-Levenshtein.* Metody wykorzystujące głosowanie, głownie kosztem wskaźnika pokrycia nieznacznie podnoszą stosunek poprawnych wyrazów do całości. Przy głosowaniu „1 z 3” strata ta nie jest zbyt duża, przy jednoczesnym niewielkim wzroście stosunku wyrazów poprawnych do całości. Metoda ta również generuje najbardziej skupione wyniki, świadczą o tym najniższe miary rozproszenia.

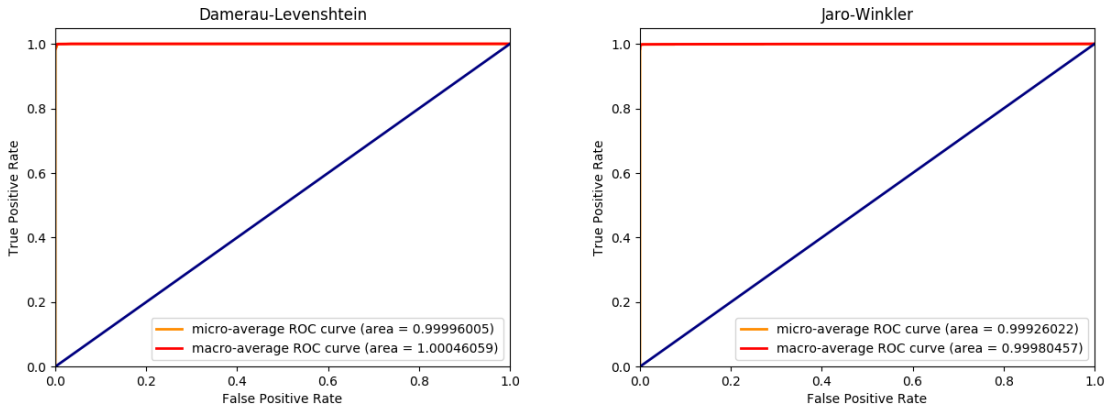
Metoda głosowania „2 z 3” powoduje już dużo większy spadek czynnika pokrycia przy jednoczesnym bardzo niewielkim wzroście stosunku wyrazów poprawnych do całości.

4.4. Drugi zestaw danych

Zestaw ten zbudowany zarówno z dłuższych łańcuchów znaków jak i pojedynczych słów. Średnia długość wynosi około 20 znaków. Wyrażenia te są bardzo unikalne, świadczy o tym wysoka średnia odległość edycyjna, dzięki czemu łatwo wyodrębnić grupy. Elementy błędne zostały wygenerowane przez skrypt, powoduje to, że odległość edycyjna wewnątrz grupy powinna wynosić od 1 do 3, co także ułatwia wyznaczenie grup.

4.4.1. Krzywa ROC

Tak samo jak w przypadku pierwszego zestawu danych przeprowadzono operacje wykreślenia krzywych ROC i wyznaczenia wartości AUC. Użyto takiej samej metodyki oraz przeprowadzono eksperyment z parametrem `SIMILARITY_THRESHOLD` równym -1.0. Do wykreślenia użyto również tego samego skryptu.



Rysunek 34 Wybrane krzywe ROC

Rysunek 34 przedstawia wybrane otrzymane krzywe ROC. Pozostałe krzywe mają taki sam kształt. Wynika z tego, że wybrane algorytmy są niemal idealnymi klasyfikatorami dla danego zestawu danych.

	AUC
Adjacent pairing	0,99797092

Cosine similarity	0,99933177
Damerau-Levenshtein	0,99996005
Jaro-Winkler	0,99926022
Levenshtein	0,99995026
Ratcliff/Obershelp	0,99998771

Tabela 3 Wartość Micro-Average AUC

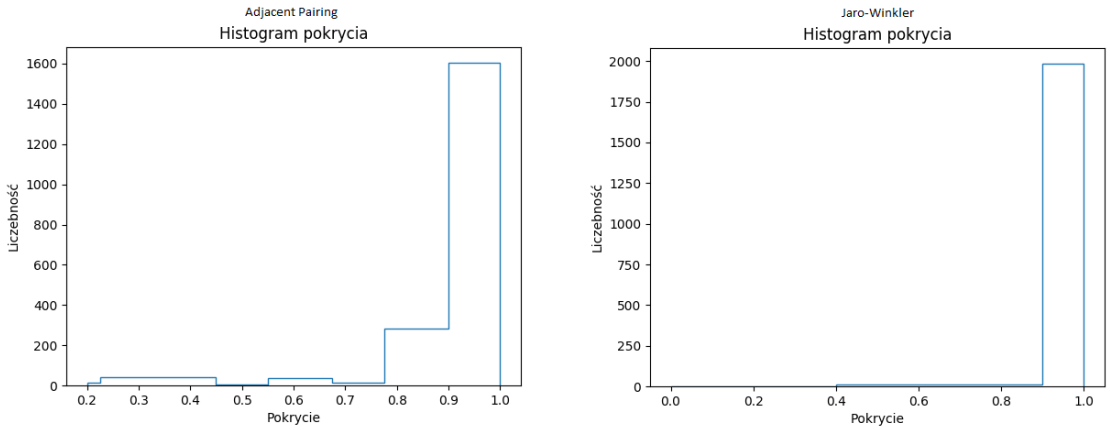
Tabela 2 przedstawia otrzymane wartości AUC dla metodyki *micro-average*. Otrzymane wartości świadczą o wysokiej jakości algorytmów jako klasyfikatory. W porównaniu do poprzedniego zestawu danych wszystkie wartości uległy poprawie, można dzięki temu stwierdzić, że algorytmy są lepiej dostosowane do klasyfikacji drugiego zestawu danych.

4.4.2. Porównanie ze względu na poziom podobieństwa

Dla drugiego zestawu danych również przeprowadzono trzy eksperymenty, w których ustalono różne poziomy minimalnego podobieństwa. Badania przeprowadzono dla takich samych wartości parametru `SIMILARITY_THRESHOLD` jak dla pierwszego zestawu.

4.4.2.1. Eksperyment przy minimalnym poziomie podobieństwa równym 0,7

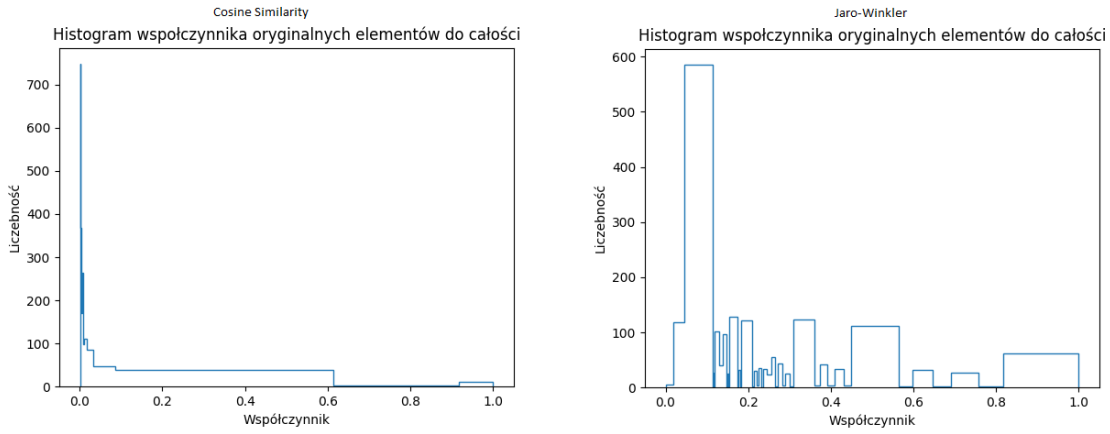
Histogramy wskaźnika pokrycia dla algorytmów *Adjacent Pairing* i *Jaro-Winkler* przedstawiono na rysunku 35.



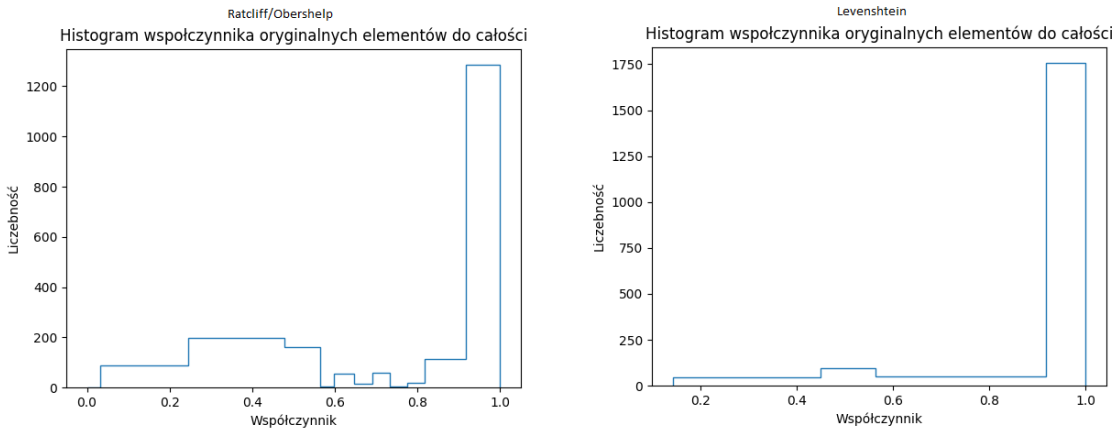
Rysunek 35 Wybrane histogramy wskaźnika pokrycia

Histogram wartości wskaźnika pokrycia przy tym minimalnym poziomie w większości przypadków przypomina ten przedstawiony dla algorytmu *Jaro-Winkler*. W takim typie histogramu największą klasą jest ta której wartość jest bliska 1. Liczebność pozostałych klas jest niemal równa 0. Jedynym wyjątkiem Jest algorytm *Adjacent Pairing*, gdzie istnieje klasa o wartości 0,8 z liczebnością około 300. W porównaniu do poprzedniego zestawu danych można zauważyć znaczne zwiększenie liczebności klasy o współczynniku pokrycia równym 1 oraz zmniejszenie liczebności pozostałych klas. Zanikły też różnice w histogramach pomiędzy różnymi algorytmami.

Rysunki 36 i 37 przedstawiają histogramy stosunku poprawnych wyrażeń do całości odpowiednio dla algorytmu *Cosine similarity, Jaro-Winkler, Ratcliff/Obershelp* oraz *Levenshtein.*



Rysunek 36 Histogramy dla Cosine Similarity i Jaro-Winkler

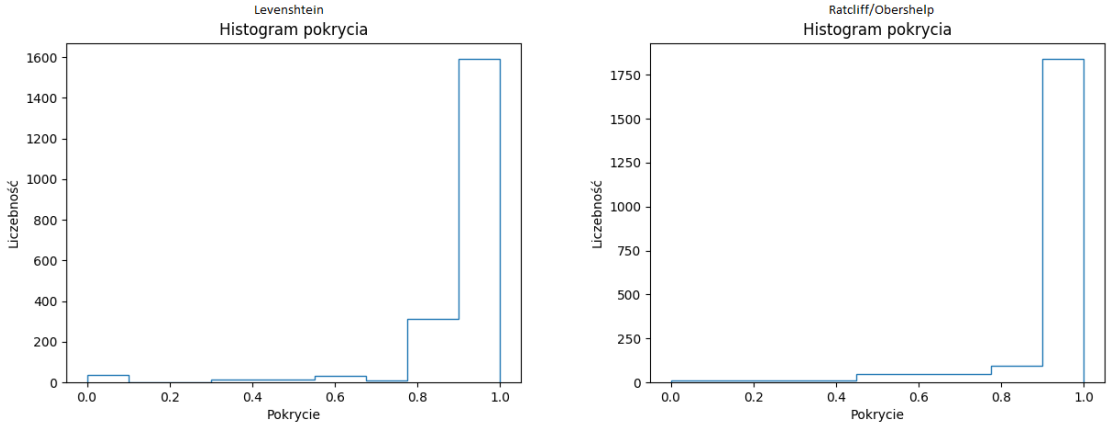


Rysunek 37 Histogramy dla Ratcliff/Obershelp i Levenshtein

Przy takim minimalnym poziomie podobieństwa algorytmy *Adjacent Pairing, Damerau-Levenshtein, Levenshtein* oraz *Ratcliff/Obershelp* mają średni stosunek oryginalnych elementów do całości równy niemal 1. Odzwierciedlają to histogramy, w których najliczniejszą jest klasa o wartości współczynnika równego 1, liczebność pozostałych klas jest bliska 0. Wyjątkami są algorytmy *Jaro-Winkler* oraz *Cosine Similarity,* gdzie najliczniejsza jest klasa o wartości współczynnika równa 0.

4.4.2.2. Eksperyment przy minimalnym poziomie podobieństwa równym 0,8

Wybrane histogramy wskaźnika pokrycia przedstawione są na rysunku 38.

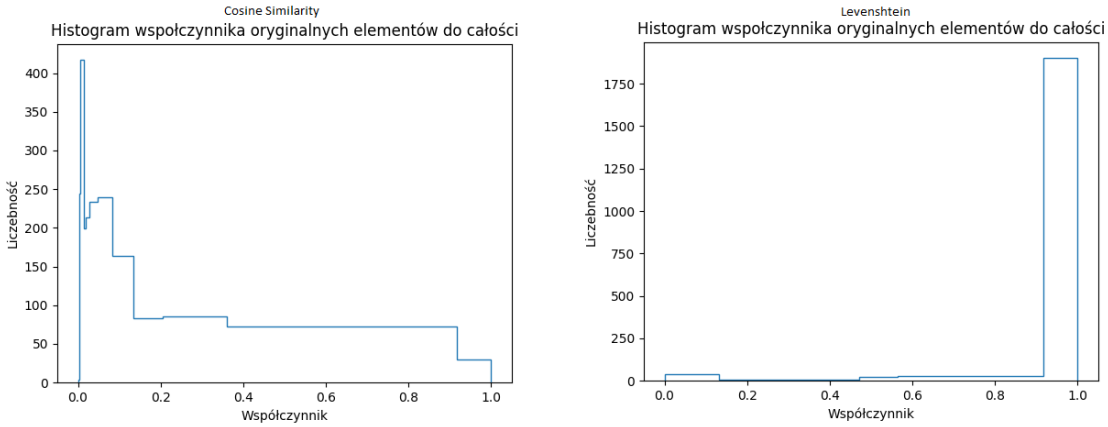


Rysunek 38 Wybrane histogramy wskaźnika pokrycia

Przy takim poziomie minimalnego podobieństwa wszystkie histogramy pokrycia zawierają najbardziej liczebną klasę o współczynniku równym 1. Liczebność pozostałych klas jest

niska. Oznacza to, że niemal wszystkie błędne wyrażenia zostały zakwalifikowane do odpowiednich grup.

Histogramy współczynnika elementów oryginalnych do całości przedstawione są na rysunku 39.

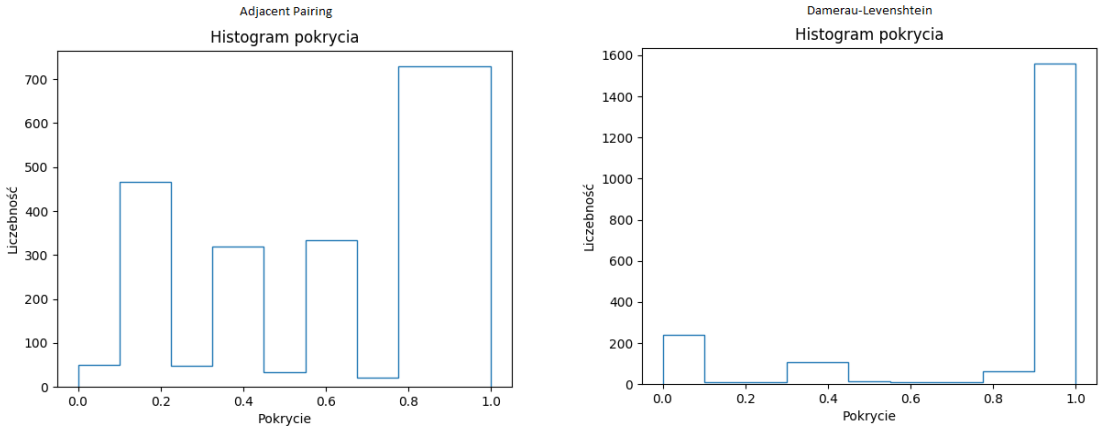


Rysunek 39 Wybrane histogramy współczynnika oryginalnych elementów do całości

Histogramy współczynnika oryginalnych elementów do całości w większości przypadków jest bardzo podobny do tego przedstawiającego wyniki dla algorytmu *Levenshtein*, gdzie najbardziej liczna jest klasa o współczynniku 1. Jedynym wyjątkiem ponownie jest histogram dla algorytmu *Cosine Similarity*, gdzie największa jest liczebność dla klasy o współczynniku bliskim 0.

4.4.2.3. Eksperyment przy minimalnym poziomie podobieństwa równym 0,9

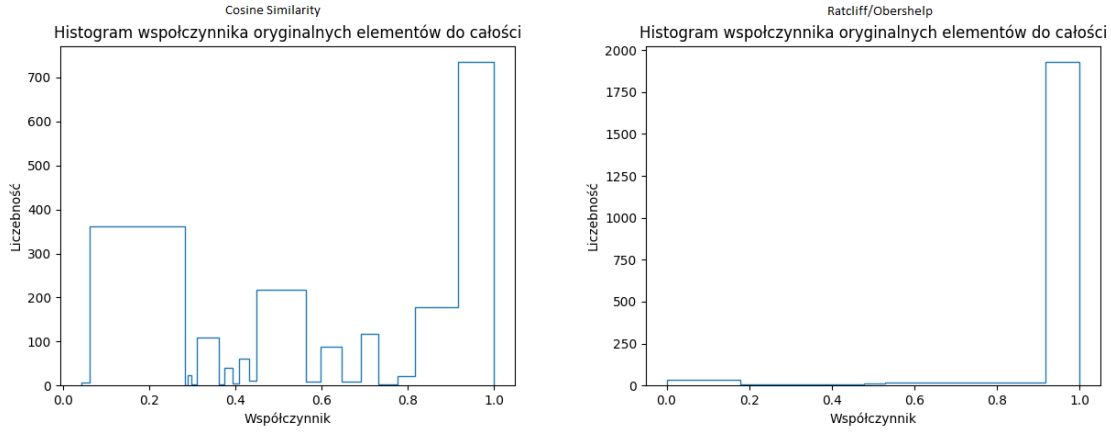
Wybrane histogramy wskaźnika pokrycia przedstawione są na rysunku 40.



Rysunek 40 Wybrane histogramy wskaźnika pokrycia

Histogramy wskaźnika pokrycia dla tak ustalonego minimalnego poziomu podobieństwa dalej posiadają najbardziej liczną klasę o współczynniku równym blisko 1, pojawiają się jednak inne klasy o znaczącej już liczebności, co powoduje obniżenie średniej wartości wskaźnika.

Histogramy współczynnika elementów oryginalnych do całości przedstawione są na rysunku 41.

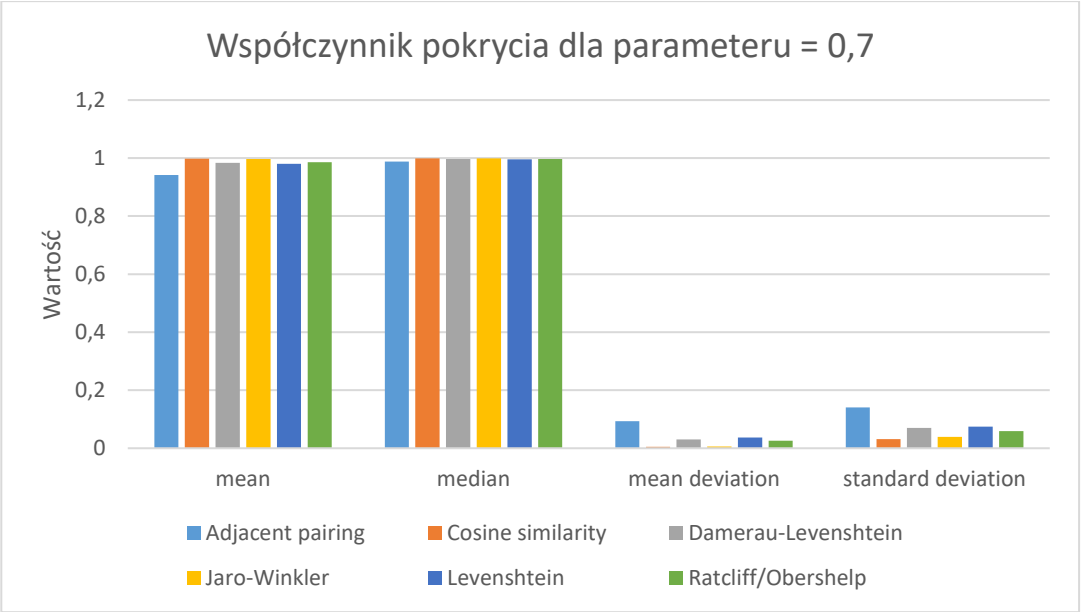


Rysunek 41 Wybrane histogramy współczynnika oryginalnych elementów do całości

Dla wszystkich algorytmów wykreślone histogramy posiadają najbardziej liczną klasę o wartości bliskiej 1. Dla algorytmu *Cosine Similarity* pojawiają się inne klasy o znacznej liczebności.

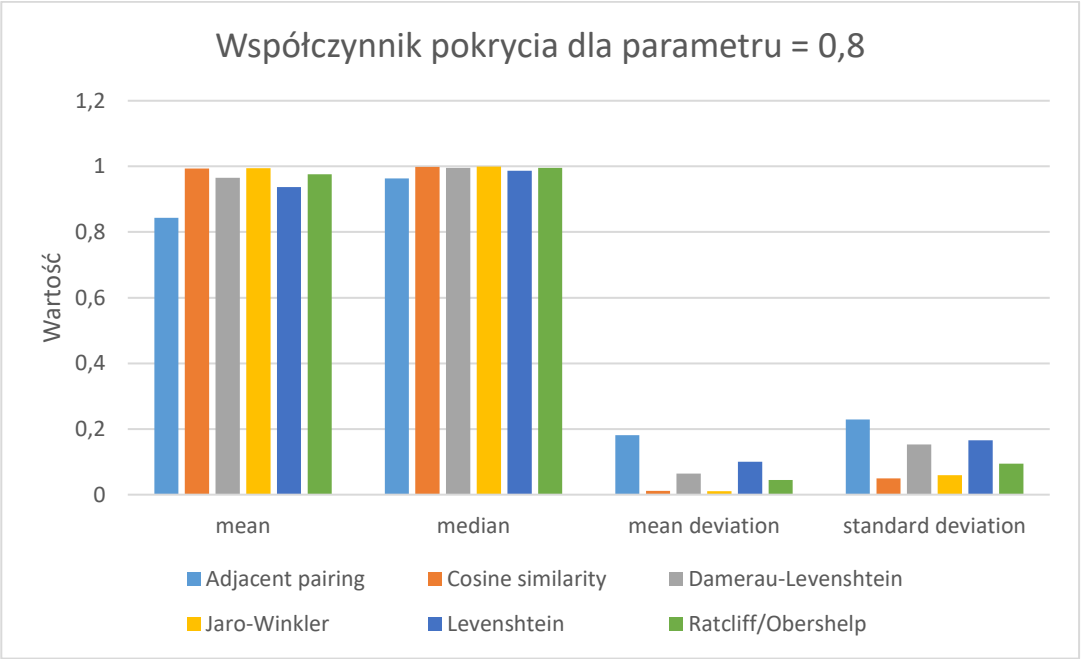
4.4.2.4. Grupowe wyniki

Analizę wartości miary pokrycia (*coverage*) dla różnych progów podobieństwa przedstawiają rysunki 42, 43 i 44.



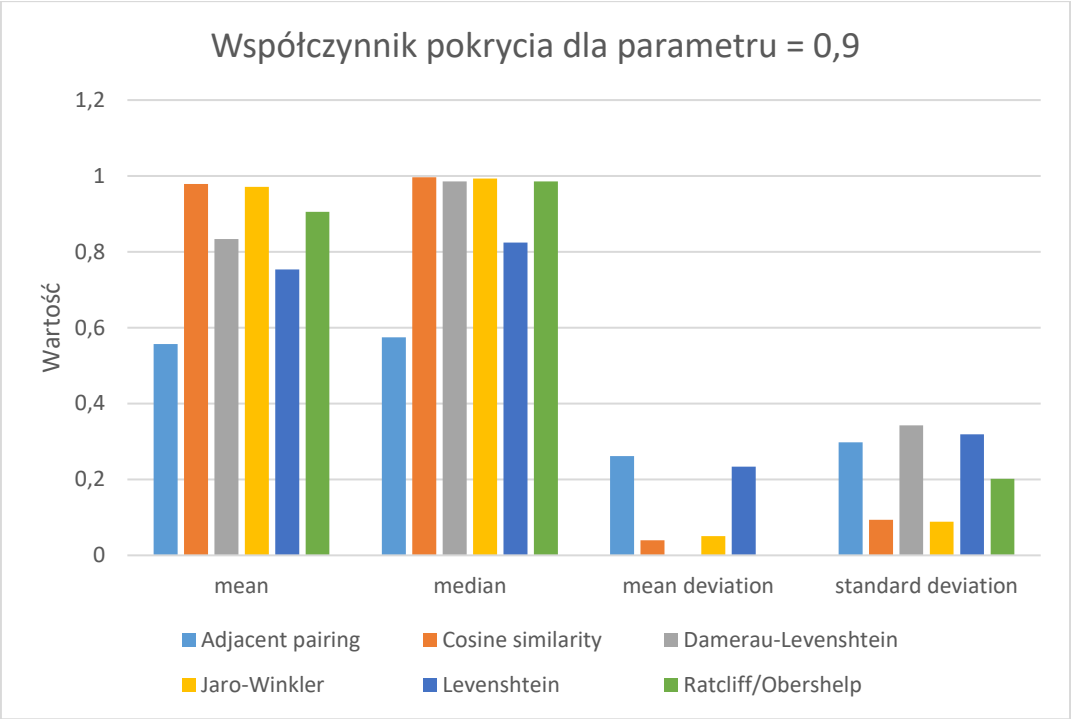
Rysunek 42 Wykres kolumnowy zbiorczy miary pokrycia(coverage) dla minimalnego poziomu podobieństwa równego 0,7

Wszystkie rozwiązania osiągnęły niemal 100% pokrycia przy określonym minimalnym poziomie podobieństwa o wartości 0,7. Wszędzie też występują niskie miary rozproszenia.



Rysunek 43 Wykres kolumnowy zbiorczy miary pokrycia(coverage) dla minimalnego poziomu podobieństwa równego 0,8

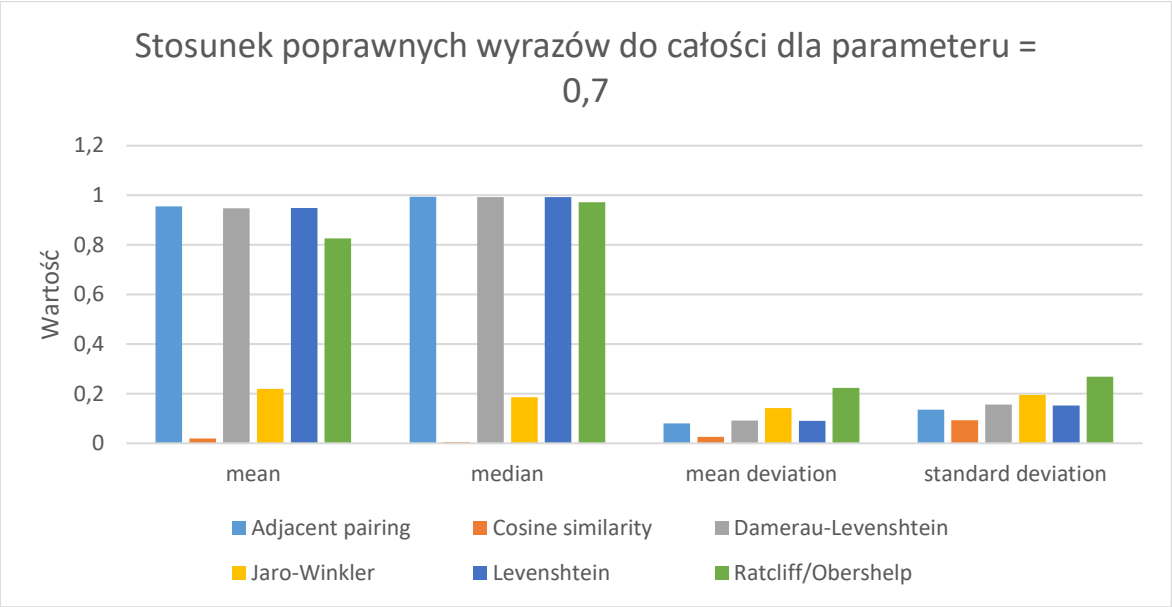
Po zwiększeniu progu minimalnego podobieństwa do 0,8 średnie wartości pokrycia nieznacznie zmalały, nadal są to wartości bliskie 1. Wzrosły nieznacznie miary rozproszenia.



Rysunek 44 Wykres kolumnowy zbiorczy miary pokrycia(coverage) dla minimalnego poziomu podobieństwa równego 0,9

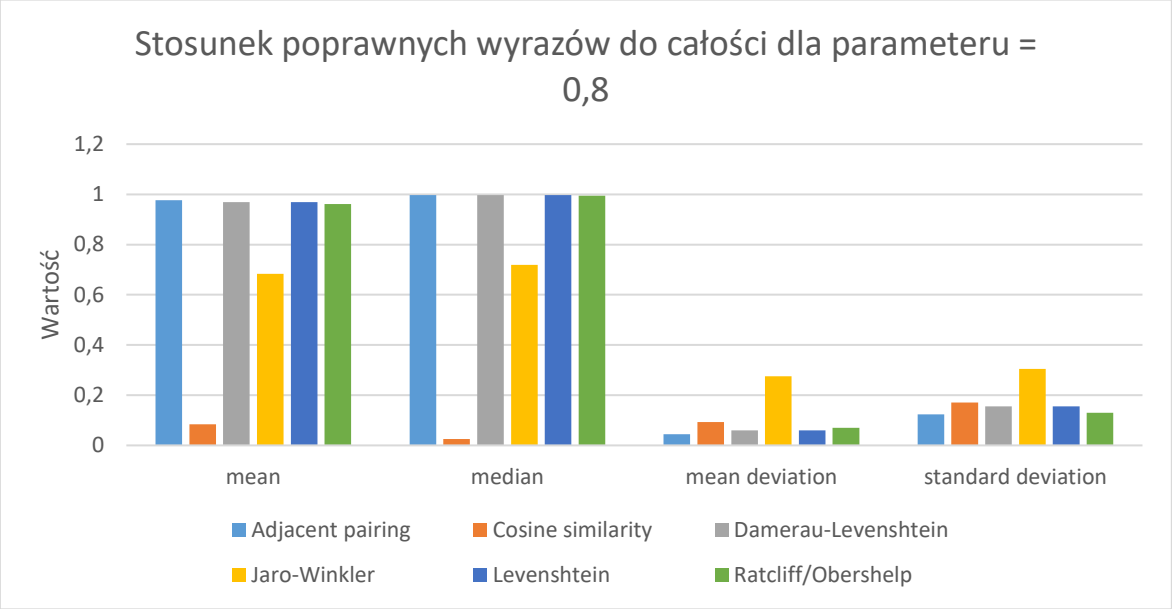
Dalszemu zwiększeniu progu minimalnego podobieństwa do wartości 0,9, towarzyszy dalszy spadek średniej wartości pokrycia. Duży spadek zanotował algorytm *Adjacent Pairing*, podobnie algorytmy *Levenshtein* i *Damerau-Levenshtein.* Pozostałe algorytmy również zanotowały spadek, ale nie tak znaczny, można także zaobserwować dalszy wzrost miar rozproszenia.

Analizę wartości stosunku poprawnych wyrazów do całości dla różnych progów podobieństwa przedstawiają rysunki 45, 46 i 47.



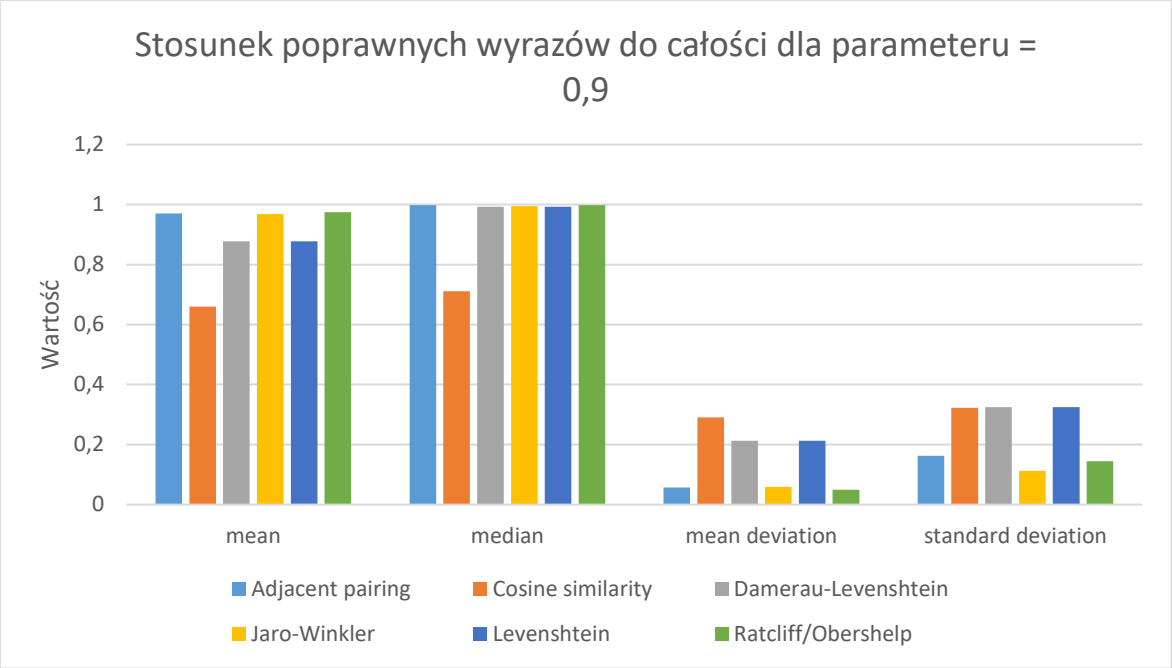
Rysunek 45 Wykres kolumnowy zbiorczy stosunku wyrazów poprawnych do całości dla minimalnego poziomu podobieństwa równego 0,7

Dla ustanowionego progu podobieństwa 0,7 stosunek dla algorytmów *Adjacent Pairing, Levenshtein, Damerau-Levenshtein* oraz *Ratcliff/Obershelp* jest bliski 1, natomiast pozostałe algorytmy posiadają tą wartość bardzo niską, co oznacza bardzo dużą liczbę nadmiarowych wyrazów. Dla tych algorytmów konieczne jest zwiększenie wartości progu, gdyż uznają one zbyt wiele wyrazów za podobne.



Rysunek 46 Wykres kolumnowy zbiorczy stosunku wyrazów poprawnych do całości dla minimalnego poziomu podobieństwa równego 0,8

Po zwiększeniu minimalnego progu podobieństwa do 0,8 można zauważyć wzrost dla algorytmów *Jaro-Winkler* oraz *Ratcliff/Obershelp,* nieznaczny wzrost zanotował również algorytm *Cosine Similarity*. Pozostałe wartości pozostały na podobnym poziomie.



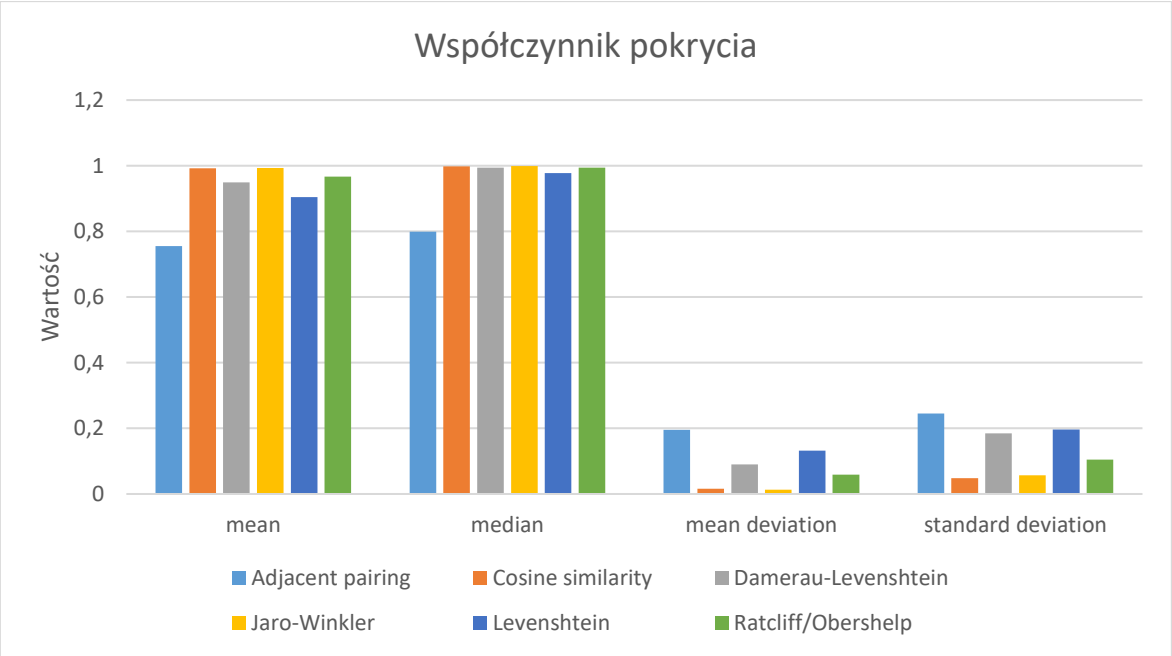
Rysunek 47 Wykres kolumnowy zbiorczy stosunku wyrazów poprawnych do całości dla minimalnego poziomu podobieństwa równego 0,9

Przy wartości parametru równej 0,9 można zauważyć pierwsze spadki średniej wartości dla algorytmów *Damerau-Levenshtein* i *Levenshtein*. Znaczny wzrost stosunku poprawnych wyrazów do całości zanotowały algorytmy *Cosine Similarity* oraz *Jaro-Winkler*. Pozostałe algorytmy nie zanotowały żadnej poważnej zmiany średniej wartości.

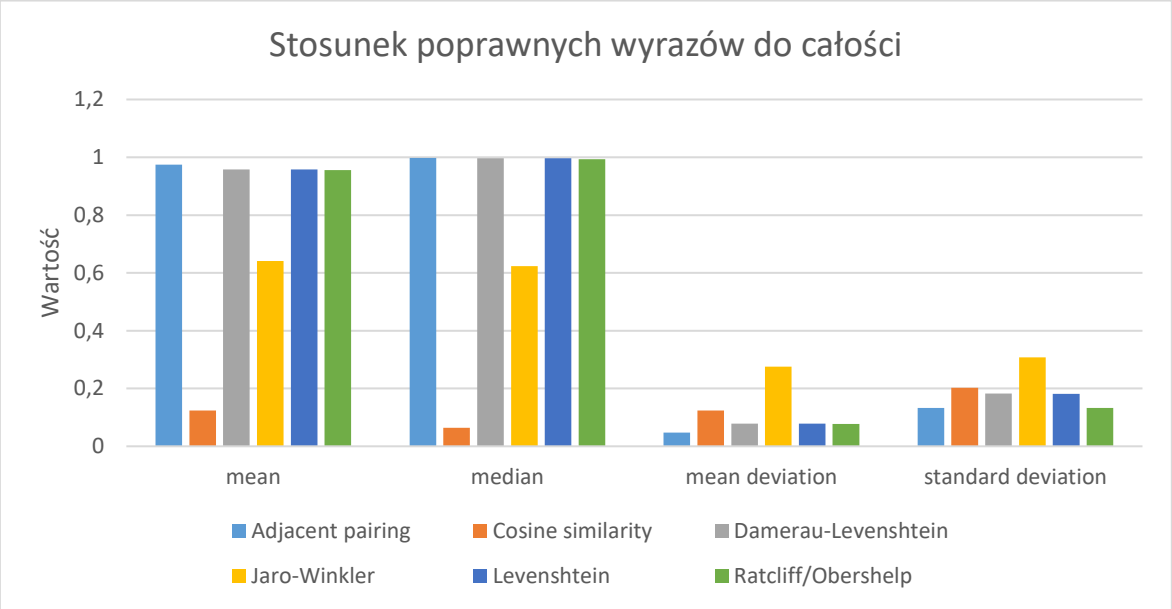
4.4.3. Porównanie ze względu na długość łańcucha znaków

Eksperyment wykonano analogiczne do tego dla pierwszego zbioru danych. Wykorzystano te same narzędzia i wcześniej przygotowane skrypty. Jedyną zmianą w stosunku do wcześniej przeprowadzonego eksperymentu jest zmiana wartości filtra, którą ustalono na 20. Wynika to z średniej długości rekordu znajdującej się w tym zbiorze danych.

Otrzymane wyniki dla ciągów krótszych niż 21 znaków przedstawiają rysunki 48 i 49.



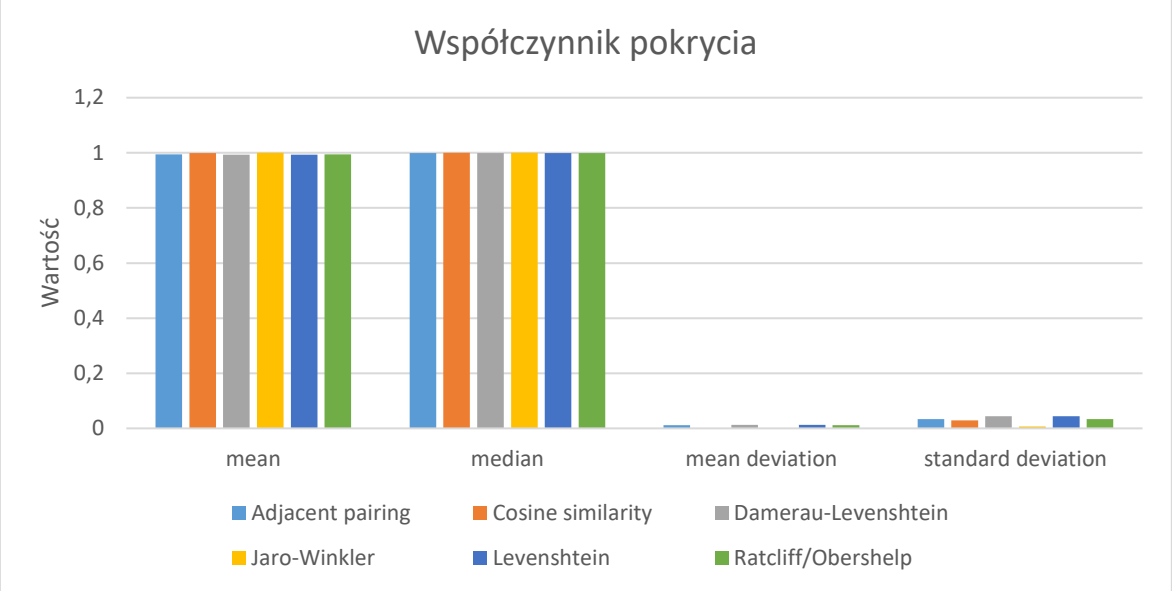
Rysunek 48 Wykres kolumnowy zbiorczy miary pokrycia(coverage) dla poprawnych łańcuchów znaków krótszych niż 21 znaków



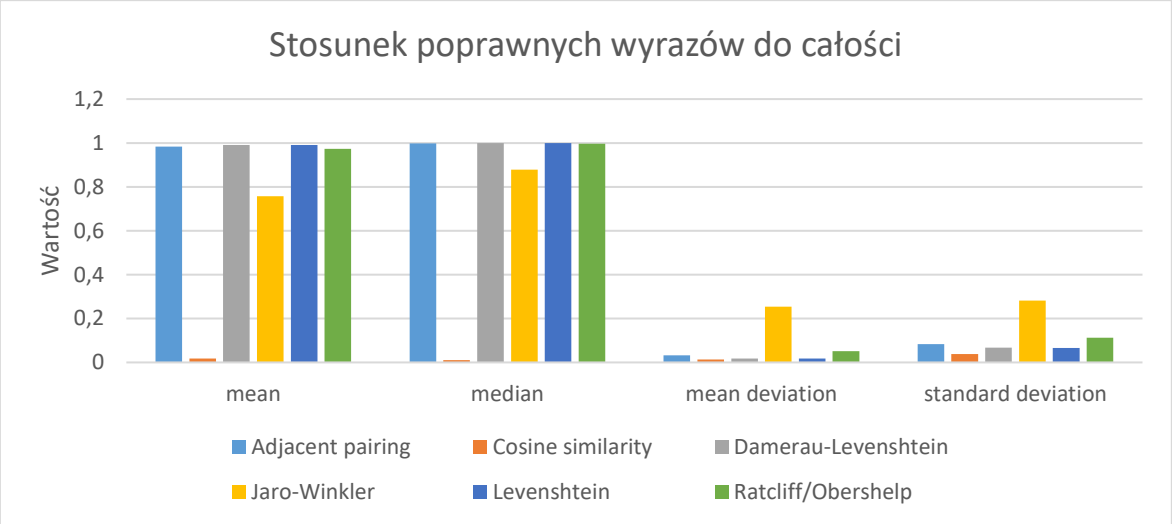
Rysunek 49 Wykres kolumnowy zbiorczy miary stosunku wyrazów poprawnych do całości dla poprawnych łańcuchów znaków krótszych niż 21 znaków

Dla krótszych wyrażeń najlepsze wyniki osiągnęły algorytmy *Damerau-Levenshtein, Levenshtein* oraz *Ratcliff/Obershelp*. Podobnie jak w eksperymencie z użyciem pierwszego zbioru danych algorytmy *Jaro-Winkler* i *Cosine Similarity* wpisują do wyznaczonych grup wiele nadmiarowych wyrażeń.

Otrzymane wyniki dla ciągów dłuższych niż 20 znaków przedstawiają rysunki 50 i 51.



Rysunek 50 Wykres kolumnowy zbiorczy miary pokrycia(coverage) dla poprawnych łańcuchów znaków dłuższych niż 20 znaków

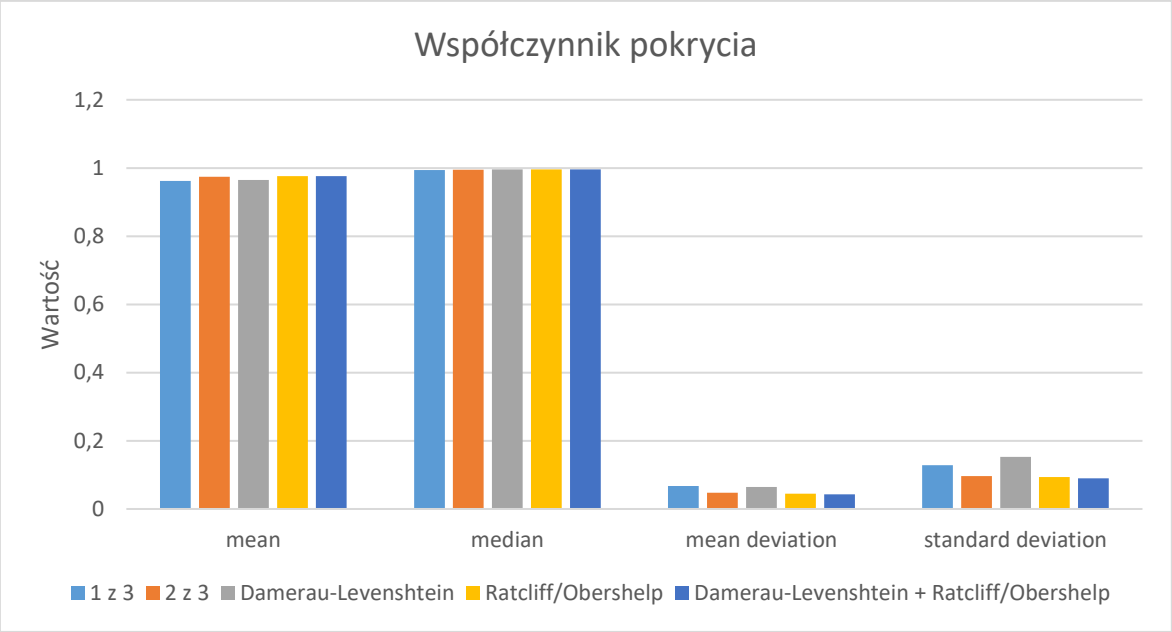


Rysunek 51 Wykres kolumnowy zbiorczy miary stosunku wyrazów poprawnych do całości dla poprawnych łańcuchów znaków dłuższych niż 20 znaków

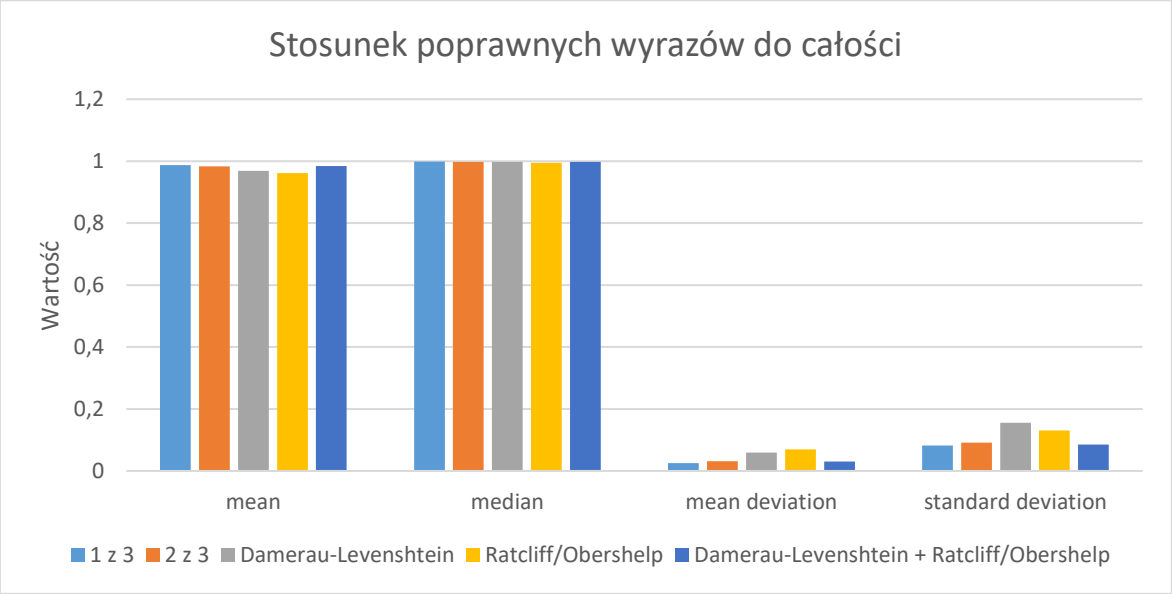
Dla wszystkich algorytmów wskaźnik pokrycia wynosi 1. Algorytm *Cosine Similarity* przypisuje jeszcze więcej nadmiarowych wyrażeń do grup niż w eksperymencie dla krótszych wyrażeń. Algorytm *Jaro-Winkler* zanotował wzrost stosunku poprawnych wyrazów do całości, podobnie jak algorytm *Adjacent Pairing*. Ogólnie algorytmy lepiej spisują się dla dłuższych wyrażeń niż krótszych.

4.4.4. Metoda łączona

W ramach tego eksperymentu wykorzystano dokładnie takie same skrypty i implementacje jak w badaniu dla pierwszego zestawu danych. Jedyną różnicą jest zmiana wartości długości wyrażenia powodującej przydzielenie do odpowiedniego algorytmu. Zgodnie z wcześniejszymi badaniami i statystykami opisującymi zbiór danych wartość tą ustalono na 20. Rysunki 52 i 53 przedstawiają otrzymane rezultaty.



Rysunek 52 Wykres kolumnowy zbiorczy miary pokrycia(coverage) dla metod łączonych



Rysunek 53 Wykres kolumnowy zbiorczy miary stosunku wyrazów poprawnych do całości dla metod łączonych

Podobnie jak w badaniu dla pierwszego zestawu danych dzięki wykorzystaniu metod łączonych udało się osiągnąć lepsze wyniki niż dla każdego algorytmu z osobna. Dla tego zestawu różnice są jednak bardzo małe, niemal wszystkie algorytmy idealnie klasyfikują rekordy.

4.5. Podsumowanie eksperymentów

W ramach pracy przeprowadzono szereg eksperymentów pozwalających przenalizować wpływ minimalnego progu podobieństwa na tworzone grupy, wpływ długości danych wejściowych. Dzięki tym eksperymentom ustalono i zbadano zachowanie metod łączonych. Zbadano także wpływ zbioru wejściowego na otrzymane wyniki.

Z przeprowadzonych eksperymentów ze zmiennym minimalnym progiem podobieństwa wynika, że nie można ustalić jednego progu dla wszystkich algorytmów. Zbyt duża wartość tego progu powoduje odrzucanie poprawnych rekordów, zbyt mała natomiast powoduje przypisywanie nadmiernej liczby rekordów. Wartość ta nie może być zbyt wysoka dla algorytmów *Levenshtein* i *Damerau-Levenshtein*, a powinna być dosyć wysoka dla algorytmów *Jaro-Winkler* i *Cosine Similaririty*.

Generalnie wraz z wzrostem długości łańcucha znaków badane algorytmy miały większą skuteczność. Jedynym wyjątkiem okazał się algorytm *Cosine Similarity,* który w badanych dla drugiego zestawu danych generalnie sprawował się gorzej niż dla pierwszego zestawu. Niektóre algorytmy nie wykazały zbyt dużej zależności od długości tak jak *Ratcliff/Obershelp.*

Dzięki wykorzystaniu metod łączonych możliwe jest eliminowanie niepoprawnych wyrazów w grupie kosztem wyrazów, które miały się w niej znaleźć w inny sposób niż manipulowanie minimalnym progiem podobieństwa.

Przebadano także wpływ zbioru danych na otrzymywane miary. W tym celu użyto znacznie większego zbioru danych z chorobami i wygenerowanymi błędnymi zapisami. Zbiór ten jest też dużo bardziej unikalny, zawiera ciągi znaków o mniejszym stopniu podobieństwa niż pierwszy zbiór. Średnia długość rekordu jest także znacznie większa w porównaniu do pierwszego zbioru. W ramach eksperymentu wykorzystano algorytmy

wcześniej używane oraz metody łączone. Wartość współczynnika pokrycia w porównaniu do poprzednio przeprowadzonych eksperymentów z innym zbiorem dla takiego samego parametru minimalnego podobieństwa znacznie wzrosła dla wszystkich algorytmów, poza *Cosinie similarity i Jaro-Winkler*, których wartość była już bliska 1. Może to wynikać z dużo większej średniej długości wyrażeń znajdujących się w tym zbiorze. Generalnie otrzymane wartości sugerują, że niemal wszystkie wygenerowane literówki zostały zaklasyfikowane do powstałych grup. Stosunek poprawnych wyrazów do całości także uległ znacznej poprawie niemal dla każdego z algorytmów. Jedynym wyjątkiem jest algorytm *Cosinie similarity,* który otrzymał gorszy wynik dla analogicznie przeprowadzonego eksperymentu dla innego zestawu danych. Algorytmowi ten wykazuję dużą zależność od długości porównywanych ciągów. Dla pozostałych algorytmów kluczowym elementem dla poprawy wyników może być unikalność rekordów, rozumiana poprzez podobieństwo poprawnych wyrażeń do innych poprawnych wyrażeń. Zbiór ten zawiera znacznie mniej podobnych poprawnych wyrażeń, co powoduje znacznie lepszą skuteczność działania algorytmów. Większość algorytmów działała regularnie, świadczą o tym miary rozproszenia, jedynym wyjątkiem jest *Jaro-Winkler,* który ze względu na swoje premiowanie zgodnych początkowych znaków jest wrażliwy na położenie popełnionego błędu.

4.6. Czas wykonywania

W ramach tego eksperymentu przeprowadzono test czasu wykonywania zadania na przedstawionym wcześniej klastrze oraz czas wykonywania z wykorzystaniem lokalnego trybu uruchomienia z pomocą biblioteki *mrjob*. Do testu lokalnego wykorzystano komputer z procesorem i7-4790k o taktowaniu 4GHz, 16 GB RAM oraz dyskiem SSD. Program lokalnie wykorzystywał jeden rdzeń procesora. W badaniu wykorzystano drugi zestaw danych oraz zadanie wykorzystujące algorytm *Ratcliff/Obershelp.*

Czas trwania zadania na komputerze lokalnym wynosił 27 minut. Z wykorzystaniem klastra 12 minut i 24 sekundy. Inne wyniki z obliczeń przeprowadzonych na klastrze przedstawia listing 8.

```
Job Counters
        Killed map tasks=1
        Killed reduce tasks=1
```

```
Launched map tasks=70
Launched reduce tasks=29
Rack-local map tasks=70
Total time spent by all maps in occupied slots (ms)=28292844
Total time spent by all reduces in occupied slots (ms)=21930276
Total time spent by all map tasks (ms)=2357737
Total time spent by all reduce tasks (ms)=1827523
Total vcore-milliseconds taken by all map tasks=2357737
Total vcore-milliseconds taken by all reduce tasks=1827523
Total megabyte-milliseconds taken by all map tasks=7242968064
Total megabyte-milliseconds taken by all reduce tasks=5614150656
```

Listing 8 Wynik działania zadania - klaster

Dzięki wykorzystaniu klastra obliczeniowego udało się skrócić czas wykonywania obliczeń dla tego zadania o połowę. Wraz z wzrostem liczby węzłów roboczych klastra czas wykonywania będzie spadać, aż do osiągnięcia wartości granicznej, przy której dalsze rozproszenie zadań nie będzie już możliwe. Innym sposobem na przyspieszenie czasu wykonywania, może być zmiana używanego narzędzia na Apache Spark. Główną cechą odróżniającą tą bibliotekę od Apache Hadoop jest przetwarzanie danych w pamięci RAM, podczas gdy w Apache Hadoop po każdej operacji informacja zostaje zapisana na dysku twardym, co znacznie wpływa na wydajność rozwiązania.

5. Podsumowanie i wnioski

Celem pracy było zaprojektowanie, implementacja i przebadanie systemu rozmytego grupowania danych bazując na narzędziu Apache Hadoop. Takie zadanie wymagało użycia przeróżnych bibliotek oraz narzędzi. Poznania nieznanych wcześniej algorytmów i sposobów prowadzenia obliczeń. Badania przeprowadzono na dwóch zbiorach danych. Dla każdego z nich przeprowadzono eksperymenty z zmiennym minimalnym progiem podobieństwa, wpływu długości łańcucha znaków na otrzymywane wyniki. Przeprowadzono także z użyciem tych dwóch wzorów operację wykreślenia krzywych ROC dla badanych algorytmów oraz przeanalizowano wpływ wykorzystania metod łączonych na otrzymywane wyniki.

W wyniku prowadzonych prac poznano sposoby porównywania łańcuchów znaków bazujących na logice rozmytej. Jest to ciekawa i przydatna umiejętność. Dzięki niej możliwie jest integrowanie danych pochodzących z różnych źródeł, wyszukiwanie zdań o podobnym znaczeniu oraz jest to metoda często wykorzystywana w eksploracji danych.

Wykorzystanie nieznanej wcześniej platformy Hadoop i modelu programowania *MapReduce* pozwala dostrzec zalety i wady przetwarzania danych w taki sposób. Dodatkową motywacją była chęć poznania tak popularnej platformy.

W pracy w zdecydowanej większości wykorzystano język Python wraz z jego różnymi obszernymi bibliotekami. Poprawiono jego wcześniejszą znajomość i poznano nowe obszary, w których jest on używany, takie jak przetwarzanie dużej ilości danych w modelu *MapReduce*, jak i analiza i interpretacja danych w postaci statystyk.

Głównym celem projektu było zapoznanie się z nowymi nieznanymi jeszcze technologiami, a także zapoznanie się z dziedzinami informatyki, które nie były wcześniej tak dobrze poznane. Wszystkie założenia pracy udało się zrealizować.

Bibliografia

1. **[Online] [Zacytowano: 1 Wrzesień 2017.] https://en.wikipedia.org/wiki/Soundex.**

2. [Online] [Zacytowano: 1 Wrzesień 2017.] https://en.wikipedia.org/wiki/Wikipedia.

3. [Online] [Zacytowano: 1 Wrzesień 2017.] http://www.cs.cmu.edu/~wcohen/postscript/ijcai-ws-2003.pdf.

4. [Online] [Zacytowano: 28 Sierpień 2017.] http://www.catalysoft.com/articles/StrikeAMatch.html.

5. [Online] [Zacytowano: 22 Sierpień 2017.] https://en.wikipedia.org/wiki/Cosine_similarity.

6. [Online] [Zacytowano: 22 Sierpień 2017.] https://blog.nishtahir.com/2015/09/19/fuzzy-string-matching-using-cosine-similarity/#fn:3.

7. [Online] [Zacytowano: 1 Wrzesień 2017.] https://en.wikipedia.org/wiki/Levenshtein_distance.

8. [Online] [Zacytowano: 1 Wrzesień 2017.] https://en.wikipedia.org/wiki/Damerau%E2%80%93Levenshtein_distance.

9. Ilyankou Ilya. [Online] [Zacytowano: 27 Sierpień 2017.] https://ilyankou.files.wordpress.com/2015/06/ib-extended-essay.pdf.

10. [Online] [Zacytowano: 30 Sierpień 2017.] https://en.wikipedia.org/wiki/Jaro%E2%80%93Winkler_distance.

11. [Online] [Zacytowano: 2 Wrzesień 2017.] https://en.wikipedia.org/wiki/Apache_Hadoop.

12. [Online] [Zacytowano: 2 Wrzesień 2017.] http://hadoop.apache.org/.

13. Radtka Zachary i Miner Donald. *Hadoop with Python.* Sebastopol : O'Reilly, 2015.

14. [Online] [Zacytowano: 2 Wrzesień 2017.] http://mrjob.readthedocs.io/en/latest/index.html.

15. [Online] [Zacytowano: 3 Wrzesień 2017.] http://jellyfish.readthedocs.io/en/latest/.

16. [Online] [Zacytowano: 13 Wrzesień 2017.] http://scikit-learn.org/stable/auto_examples/model_selection/plot_roc.html.

17. [Online] [Zacytowano: 15 Wrzesień 2017.] https://en.wikipedia.org/wiki/Multi-label_classification.

18. [Online] https://github.com/Yelp/mrjob.

Załączniki