POLITECHNIKA WROCŁAWSKA
WYDZIAŁ ELEKTRONIKI

KIERUNEK: INFORMATYKA
SPECJALNOŚĆ: INŻYNIERIA SYSTEMÓW INFORMATYCZNYCH

PRACA DYPLOMOWA MAGISTERSKA

Klasyfikacja tematyczna tekstów
w języku polskim

Subject classification of texts in Polish

AUTOR:
Jakub Pomykała

PROWADZĄCY PRACĘ:
dr inż. Tomasz Walkowiak, W4/K9

OCENA PRACY:

WROCŁAW, 2018

Opracował: Tomasz Kubik <tomasz.kubik@pwr.edu.pl>
Data: maj 2016

Streszczenie

Celem niniejszej pracy dyplomowej było opracowanie systemu automatycznego przypisywania tekstów w języku polskim do grup tematycznych. Zapoznanie się z narzędziami NLP (ang. natural language processing) dla języka polskiego oraz porównanie efektywności algorytmów klasyfikacji z nauczycielem. W pracy wykorzystano dwa różne modele: *Bag-Of-Words* wraz z normalizacją *tf-idf* oraz *fastText*, który jest rozwinięciem modelu *word2vec*. Badania zostały przeprowadzone na dwóch korpusach danych: zebranych artykułach z popularnych stron internetowych oraz porównawczym udostępnionym w repozytorium Clarin.

Głównymi wyzwaniami w pracy było:

- Zebranie korpusu dokumentów w języku polskim z informacjami o przynależności do grup tematycznych,
- opracowanie algorytmów przetwarzania tekstów z wykorzystaniem metod NLP dla języka polskiego,
- implementacja programowa zaproponowanych algorytmów,
- nauczenie i przetestowanie klasyfikatorów,
- modyfikacja metod przetwarzania dokumentów.

W pracy wykorzystano następujące technologie:

- Java 9
- Spring Boot
- Python 3.6
- SciKit Learn
- fastText

Spis treści

Spis rysunków

Spis tabel

Skróty

NLP (ang. *Natual Language Processing*)
NB (ang. *Naive Bayes*)
LDA (ang. *Latent Dirichlet allocation*)
BoW (ang. *Bag-Of-Words*)
SVM (ang. *Support Vector Machine*)
DT (ang. *Decision tree*)
JVM (ang. *Java virtual machine*)
tf-idf (ang. *term frequency–inverse document frequency*)
CCL (ang. *corpus constraint language*)
SGJP (*słownik gramatyczny języka polskiego*)
LDA (ang. *Latent Dirichlet allocation*)
REST (ang. *representational state transfer*)
API (ang. *application programming interface*)
XML (ang. *extensible markup language*)
JSON (ang. *JavaScript object notation*)
SEO (ang. *search engine optimization*)
AJAX (ang. *asynchronous JavaScript and XML*)
HTML (ang. *hypertext markup language*)
URL (ang. *uniform resource locator*)
DOM (ang. *document object model*)

Rozdział 1

Wstęp

1.1. Wprowadzenie

Zasadniczym problemem badawczym podejmowanym w niniejszej pracy dyplomowej jest przeprowadzenie analizy porównawczej różnych metod stosowanych przy kategoryzacji tekstów, ekstrakcja cech określających przynależność danego tekstu do grupy tematycznej oraz klasyfikacja zwektoryzowanych dokumentów pomocy wybranych klasyfikatorów z nauczycielem. Tekstami wejściowymi, które zostały poddane analizie, są artykuły pobrane z wybranych portali internetowych oraz, jako drugi korpus porównawczy, wpisy z serwisu Wikipedia udostępnione w repozytorium Clarin.

1.2. Przetwarzanie języka naturalnego

Przetwarzanie i analiza języka naturalnego za sprawą coraz szybszych i wydajniejszych komputerów znajduje wiele zastosowań nie tylko w przemyśle, ale i w życiu codziennym każdego użytkownika dzisiejszego Internetu. Początkowo prace nad elementami lingwistyki były stosowane jedynie do wykrywania nieprawidłowości i oczywistych błędów w komputerowych edytorach, z czasem rozwój procesorów oraz coraz niższe ceny za pamięć pozwoliły na przetwarzanie oraz analizowanie coraz większej ilości danych w relatywnie krótkim czasie. Umożliwiło to również stosowanie bardziej skomplikowanych algorytmów, choć oczywiste optymalizacje w celu skrócenia czasu przetwarzania wciąż są stosowane. [25]

1.3. Cel badań

Podstawowym celem było opracowanie systemu automatycznego przypisywania tekstów w języku polskim do grup tematycznych oraz selekcja zbioru cech deskryptywnych opisujących informacje świadczące o przynależności tekstu do grupy tematycznej. W tym celu została stworzona aplikacja w języku Python, która implementowała model *Bag-Of-Words* wraz z normalizacją *tf-idf* oraz trzy klasyfikatory z biblioteki *SciKit Learn*. Dodatkowo, zaimplementowany został wrapper dla biblioteki *fastText* aby porównać skuteczność drugiego modelu opartego o *word embedding*. Aplikacja uruchamia się w trybie konsolowym, dzięki czemu możliwy jest swobodny dobór parametrów. Wszystkie trzy klasyfikatory należą do grupy algorytmów klasyfikacji z nauczycielem, dlatego też ważnym aspektem niniejszej pracy było przygotowanie danych wejściowych. Następnie dokonana została ekstrakcja cech deskryptywnych, dobranych empirycznie w celu uzyskania jak najlepszego wyniku. W pracy również wskazane zostały parametry, które miały największy wpływ na zmianę wskaźnika jakości klasyfikacji, obliczonego

dzięki sprawdzeniu, ilu tekstom klasyfikator przypisał poprawne grupy tematyczne. Odbyło się to dzięki podziałowi sklasyfikowanych danych wejściowych na dane treningowe oraz testowe. Za dane wyjściowe przyjęto artykuły pobrane ze stron takich jak:

- historykon.pl
- pap.pl
- focus.pl
- kafeteria.pl
- niebezpiecznik.pl
- purepc.pl
- zaufanatrzeciastrona.pl

Do głównych wyzwań jakie zostały podjęte w pracy należy zaliczyć:

- probabilistyczne dobranie cech deskryptywnych,
- przygotowanie odpowiednich narzędzi,
- konstrukcja środowiska testowego,
- przygotowanie poprawnych językowo danych wejściowych wraz z poprawnymi grupami tematycznymi.

1.4. Zakres pracy

Główną częścią pracy było zbadanie różnych metod klasyfikacji tekstu, dobieranie odpowiednich parametrów oraz wyselekcjonowanie cech w celu uzyskania najlepszej jakości dopasowania każdego z klasyfikatorów. W pracy nie został zaimplementowany własny algorytm, jednak podjęto próbę modyfikacji algorytmów oraz sposobu przetwarzania danych w celu poprawy wyników. Praca skupiała się głównie na naukowym porównaniu jakości oraz wydajności wybranych metod, klasyfikatorów oraz dokumentów wejściowych. Wszystkie zastosowane klasyfikatory należą do rodziny klasyfikatorów z nauczycielem, co oznacza, że rozpatrywane klasy zostały dobrane przez autora pracy na podstawie wcześniejszej analizy zebranego korpusu.

1.5. Wykorzystane technologie

W pracy dobrano technologie, które pozwoliły na jak najłatwiejszą pracę z wybranymi klasyfikatorami oraz przetwarzanie danych tekstowych w prosty sposób. System został podzielony na dwie aplikacje. W języku Java napisana została aplikacja odpowiedzialna za pobieranie, wstępne przetwarzanie oraz zapisywanie danych w formacie XML. Część badawcza pracy została napisana w języku Python; nastąpiła w niej dokładniejsza analiza korpusu, właściwe przetworzenie tekstu, wektoryzacja, klasyfikacja oraz badanie jakości.

- **Python 3.6** - interpretowany język programowania wysokiego poziomu do programowania ogólnego przeznaczenia.[24] Stworzony przez Guido van Rossuma i po raz pierwszy wydany w 1991 roku. Został zastosowany w pracy ze względu na jego dużą popularność, która, niejednokrotnie umożliwiła wsparcie ze strony społeczności tegoż języka.
- **NumPy** - podstawowy pakiet do obliczeń naukowych w języku Python. Pozwala na manipulację danymi w prostszy sposób niż za pomocą wbudowanych funkcji w język Python. [29]
- **MatPlot** - biblioteka, która pozwala na generowanie wykresów w kodzie języka Python. [19]
- **SciKit-Learn 0.19.1** - darmowe narzędzie stosowane do komputerowego nauczania maszynowego dla języka Python. Zawiera różne algorytmy klasyfikacji, regresji i grupowania, w tym

Support Vector Machine, NaiveBayes, drzewa decyzyjne. Aby zapewnić maksymalną rzetelność przeprowadzanych testów, w szczególności tych, które dotyczyły pomiarów czasowych, wszystkie wykorzystane w pracy klasyfikatory pochodziły z tej biblioteki. [30]

- **Java 9** - uniwersalny język programowania, został zastosowany ze względu na łatwą obsługę współbieżności, obiektowość, doświadczenie autora pracy oraz mnogość dostępnych bibliotek. [15] Oryginalnie stworzona przez firmę Sun Microsystems w 1995 roku, aktualnie wspierana przez firmę Oracle. Zastosowana wersja, w momencie pisania pracy jest aktualnie obowiązującą stabilną wersją. [23]
- **Spring Boot 2.0 RC3** - framework Javowy, pozwalający na pisanie skalowalnych aplikacji webowych, workerów. Został wybrany, aby ułatwić pracę wymagającą dostępu do danych internetowych oraz dzięki swojej modularności, co pozwoli w przyszłości w prosty sposób rozwijać aplikację. [43]
- **Spring Boot Shell 2.0 M2** - jeden z projektów od twórców frameworka Spring, umożliwia stworzenie aplikacji działającej przez interfejs konsolowy (CLI), dzięki czemu wszystkie parametry można ustalić za pomocą flag oraz argumentów. [12]

Wszystkie wykorzystane narzędzia, które zostały wykorzystane w pracy, są darmowe i udostępnianie na zasadzie wolnego oprogramowania.

Rozdział 2

Klasyfikacja tematyczna

Klasyfikacja tematyczne polega na przypisywaniu jednej lub więcej kategorii/klas danemu dokumentowi. Można to zrobić ręcznie lub algorytmicznie. W tej pracy zebrane zostały dokumenty z portali internetowych, sklasyfikowane ręcznie przez ich autorów. Następnie spróbowano odnaleźć wzorce, którymi kierowano się podczas kategoryzacji oraz podjęto próbę klasyfikacji algorytmicznej. Wyłanianie kategorii na podstawie tekstu ma zastosowanie w wielu dziedzinach, np. w bibliotekoznawstwie, w urzędach lub innych miejscach, gdzie przetwarza się duże ilości danych, których ręczna klasyfikacja jest niemożliwa.[3] Teksty mogą być klasyfikowane według różnych atrybutów, np.: płci autora, roku wydania, rodzaju dokumentu. W omawianej pracy podjęto próbę wyłonienia odpowiedniej kategorii podanych tekstów.

2.1. Przegląd literatury

W pracy wykorzystano wiedzę z artykułów oraz książek, które niekoniecznie odnoszą się do przetwarzania tekstu w języku polskim. W większości zaproponowane sposoby klasyfikacji były tożsame, dopóki nie odnosiły się do zagadnień opartych o zasady, którymi kieruje się dany język. Gramatyki dla języka polskiego oraz angielskiego różnią się w znaczący sposób, konieczne zatem było zaproponowanie odpowiednich klas gramatycznych. [33]

2.2. Przyjęta terminologia

Wiele nowych technologii i odkryć dotyczy najczęściej języka angielskiego i to przede wszystkim ten język poddawany jest różnego rodzaju testom, podobnie jest także w odniesieniu do przetwarzania i automatycznej kategoryzacji tekstów. Oczywistą różnicą miedzy językiem polskim, a językiem angielskim jest rozbudowana fleksja w przypadku języka polskiego, co wymagało przekształcenia niektórych terminów. W niniejszej pracy przyjęte terminy zostały zaczerpnięte z literatury skupiającej się głównie na indeksowaniu treści [25], te z kolei są przystosowane do języka polskiego bazując na rozwiązaniach stosowanych w języku angielskim oraz wiedzy zawartej w książce *Statystyka dla językoznawców* [36]. Podstawowe terminy wykorzystane w pracy:

- **token** - (segment) najmniejsza niepodzielna jednostka tekstu,
- **słowo** - ciąg znaków pomiędzy deliminatorami tekstu,
- **słowoforma** - (forma wyrazowa) słowo o przypisanych cechach semantycznych i gramatycznych,
- **leksem** - zbiór słowoform, zawierający wszystkie poprawne formy gramatyczne,
- **hasło** - (lemat) - jedna zwyczajowo przyjęta forma gramatyczna,

- **wyraz** - graficzna postać leksemu, hasła lub słowoformy,
- **słowa nierelewantne** (ang. stop words) - wyrazy mało znaczące.

2.2.1. Analiza morfologiczna

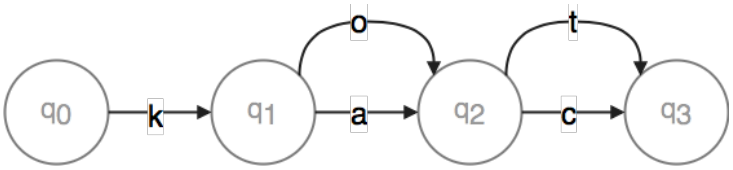
Morfologia jest nauką o budowie słów. Dziedzina ta zajmuje się dwiema odrębnymi dziedzinami takimi jak fleksja, która opisuje różne formy danego fleksemu oraz słowotwórstwo, które definiuje zasady tworzenia wyrazów pochodnych. [28] Istotnym aspektem pracy była analiza morfologiczna, która dostarcza potrzebną wiedzę o zadanym tekście. Analiza morfologiczna polega na wyznaczeniu wszystkich form podstawowych dla danego słowa (lematów) oraz wyznaczeniu wszystkich możliwych interpretacji wyznaczonego lematu. Operacją odwrotną do analizy morfologicznej jest synteza morfologiczna, czyli stworzenie wykładnika formy fleksyjnej na podstawie lematu oraz cech fleksyjnych. Kontekst w przypadku analizy morfologicznej nie jest brany pod uwagę. [38]

2.2.2. Segmentacja

Segmentacja skupia się na podziale tekstu na poszczególne segmenty. Wbrew pozornej niezależności segmentacja jest silnie związana z cechami dla danego języka. Oznacza to, że problem segmentacji nie jest łatwo rozwiązywalny przy pomocy reguły dzielenia tekstu na podstawie odstępów między segmentami. [28] Wyróżnić można kilka klas segmentów:

- ciąg cyfr,
- ciąg cyfr z wewnętrznym przecinkiem/kropką,
- ciąg małych liter,
- ciąg małych liter poprzedzonych wielką literą,
- ciąg składający się tylko z wielkich liter,
- znak interpunkcyjny.

Do rozpoznawania granic tokenów można zastosować automaty skończone, przykład został pokazany na rysunku 2.1. Jest to trudna w realizacji metoda, gdyż wymaga zbioru słowników, zawierających wszystkie formy każdego leksemu.



Rys. 2.1: Przykład automatu dla fragmentu słownika języka polskiego z formami mianownikowymi wyrazów: kot, kat, koc, kac

Drugim powszechnie znanym sposobem na segmentację tekstu jest zastosowanie algorytmu regułowego. Opiera się on na zastosowaniu kilku zasad, na podstawie których wyznaczane są zakończenia i rozpoczęcia tekstu oraz potencjalne zakończenia i potencjalne rozpoczęcia tekstu. Potencjalnym zakończeniem tekstu może być skrót, na przykład *inż.*. Mimo, iż zdania zawsze kończone są kropkami, to użycie skrótu *inż.* w zdaniu nie powinno dzielić tego zdania na dwa osobne byty. [37]

2.2.3. Lematyzacja

Celem lematyzacji jest sprowadzanie formy fleksyjnej wyrazu do postaci słownikowej. Zastosowanie form podstawowych w klasyfikacji tekstu ma bardzo dużą zaletę z uwagi na możliwość

ograniczenia zbioru, na którym wykonywana jest praca. Dodatkowo umożliwia to traktowanie wszystkich wyrazów, które stanowią odmianę danego zwrotu, jak to samo słowo. W przypadku języka angielskiego, który ma bardzo ubogą fleksję, wyznaczanie form podstawowych może się odbywać, np. na drodze usuwania końcówek wyrazów *-ing* lub *-s*. Język polski ma bardzo bogatą fleksję, dlatego takie podejście zupełnie by się nie sprawdziło. Konieczne zatem było przeprowadzenie kompletnej analizy morfologicznej z wykorzystaniem gotowych narzędzi i bibliotek.

2.3. Istniejące rozwiązania

Podczas wyszukiwania istniejących rozwiązań skupiono się głównie na narzędziach spełniających rolę analizatorów morfologicznych, ponieważ przede wszystkim one będą dostarczać potrzebną wiedzę o tekście.

2.3.1. Morfeusz SGJP

Morfeusz jest analizatorem morfologicznym dla języka polskiego, który jest najczęściej spotykany w różnego rodzaju pracach. [45] Dla zadanego ciągu słów potrafi podzielić zdanie na słowa, a następnie zinterpretować każde słowo podając jego klasę gramatyczną i formę podstawową. Każde słowo może mieć więcej niż jedną interpretację, co wynika ze złożoności języka. Tagi zastosowane w programie Morfeusz są pozycyjne. Pierwsza pozycja definiuje część mowy, kolejne oznaczają wartości kategorii gramatycznych każdej klasy. [38] Program *Morfeusz SGJP* w wersji demo jest dostępny pod adresem `http://sgjp.pl/morfeusz/demo` Analizator ten wykorzystuje metodę automatów skończonych do segmentacji tekstu.

2.3.2. TaKIPI

TaKIPI (Tager Korpusu IPI PAN) to narzędzie, które ustala opis morfo-syntaktyczny dla wyrazów w tekście. Program wykorzystuje algorytm regułowy, składający się z niewielkiego zestawu ręcznie stworzonych reguł oraz kilku tysięcy pozyskiwanych automatycznie. Według źródła, skuteczność TaKIPI wynosi około 93.4 procent. [31] TaKIPI wykorzystuje program *Morfeusz* omówiony wcześniej oraz narzędzie *Odgadywacz* do odkrywania opisów morfo-syntaktycznych nieznanych wyrazów i automatycznego budowania wcześniej wspomnianych reguł. [32] Oprogramowanie jest rozpowszechniane w postaci programu komputerowego, niestety brak kompatybilności z system *MacOS* sprawił, że nie był on brany po uwagę podczas rozważań na wyborem analizatora. [31]

2.3.3. Platforma CLARIN

CLARIN [41] jest projektem europejskim, który zapewnia zbiór narzędzi do przetwarzania języka w postaci wygodnego interfejsu webowego REST API. Dodatkowym atutem jest mnogość dostępnych konfiguracji oraz przyjmowane formaty danych (np.: pliki *.zip, pliki *.txt lub tekst w zapytaniu POST do serwera i inne). Komunikacja przez API odbywa się w sposób asynchroniczny; oznacza to, że dane najpierw muszą zostać wysłane i zapisane na serwerze, a odwołujemy się do nich przy pomocy ID, które dostajemy w odpowiedzi. Następnie, można użyć dostępnych narzędzi poprzez wysyłanie odpowiednich zapytań HTTP do serwera.

2.3.4. Tagger WCRFT2

Tagger WCRFT2 (Wrocław CRF Tagger) jest jednym z narzędzi dostępnych w Clarin; jest to tagger morfo-syntaktyczny dla języka polskiego. Składa się on z kilku mniejszych programów i technologii takich jak: Corpus2, MACA, Morfeusz SGJP [34] oraz WCCL. Użycie tego narzędzia odgrywa istotną rolę w niniejszej pracy, ponieważ pozwoliło ono na uzyskanie lematów z tekstów oraz wyznaczenie klas gramatycznych dla każdego słowa w prosty sposób. Dostęp do narzędzia można uzyskać przez skompilowanie go na swoim komputerze lub przez API, udostępnione w Clarin. Stworzenie tego narzędzia było zainspirowane przez podobny program o nazwie Wrocław Memory-Based Tagger. WCRFT znajduje się na licencji GNU LGPL v3.0.

Rozdział 3

Projekt i implementacja

3.1. Pozyskanie dokumentów do analizy

Założeniem pracy była analiza artykułów dostępnych w internecie. W celu zebrania korpusu dokumentów, napisana została aplikacja w języku Java w wersji 9. Dzięki wykorzystaniu frameworka *Spring Boot* oraz biblioteki *Spring Shell*, program został w prosty sposób przekształcony w aplikację shellową. Podczas uruchamiania użytkownik ma możliwość wyboru:

1. ile wątków zostanie użytych podczas pobierania,
2. ścieżkę zapisu dokumentów,
3. w jakim formacie zostaną zapisane pliki (XML, JSON lub TXT)
4. kodowania plików (lista dostępnych kodowań jest tożsama z listą kodowań obsługiwanych przez język Java).

Implementacja, wykorzystanie parametrów i wartości domyślne zostały przedstawione na listingu poniżej.

```
@ShellMethod(
  value = "Uruchom parser z domyślnymi parametrami",
  key = "start-download")
public void startDownload(
    @ShellOption(help = "threads", defaultValue = "1") @Min(1)
          Integer threads, \\ 1
    @ShellOption(help = "save path", defaultValue = "/Users/jakub/Desktop")
          String path, \\ 2
    @ShellOption(help = "file format: xml, json, txt", defaultValue = "xml")
          String format, \\ 3
    @ShellOption(help = "encoding", defaultValue = "utf8")
          String encoding \\ 4
) {
  //implementacja programowa
}
```

W celu rozpoczęcia pobierania należy wydać komendę *start-download*, a następnie podać argumenty w formacie *-nazwa_argumentu wartość_argumentu*. Zalecane jest aby wartość parametru *threads* była równa lub mniejsza liczbie dostępnych wątków na uruchamianym sprzęcie. Zapobiegnie to niepotrzebnym przestojom aplikacji podczas zrównoleglania pracy i oczekiwania na dostępne zasoby.

Aby przyspieszyć proces implementacji programu, w projekcie Javowym użyto kilku dodatkowych bibliotek, które wspomagają pracę:

- **Apache Commons Lang** - biblioteka, która zawiera wiele klas typu *Utilities* wspomagających pracę z danymi tekstowymi,

- **Lombok Project** - auto generacja kodu, tworzenie *getterów/setterów*, wydajna implementacja metod *hashCode* oraz *equals*, automatyczna implementacja wzorca projektowego *budowniczy* [16],
- **Jackson** - prosta biblioteka udostępniająca wspólny interfejs do zapisu obiektów Javowych do pliku formacie XML lub JSON,
- **JUnit** - pozwala na przyspieszenie pracy tworząc testy jednostkowe poszczególnych metod, dzięki czemu nie trzeba uruchamiać aplikacji, aby sprawdzić, czy dana funkcja spełnia swoje założenia [27],
- **JSoup** - popularna biblioteka do pobierania dokumentów HTML z internetu oraz ekstrakcji danych z kodu XHTML.

Działanie aplikacji może zostać w prosty sposób zmodyfikowane bez zmiany istniejącego kodu, ponieważ kod został napisany zgodnie z zasadą *open/closed principle* [27], która mówi, że moduły powinny być otwarte na rozszerzenie, ale zamknięte na modyfikacje. Aby zapewnić czytelność kodu, program pobierający wykorzystuje również wiele wzorców projektowych, co pozwala na uniknięcie duplikacji kodu oraz ułatwia dalszą pracę. Zastosowanie wzorca projektowego *template method* [16] oraz *depencency intjection* z frameworka Spring [42] daje możliwość bardzo prostego rozszerzenia listy stron, z których pobierane będą artykuły. W tym celu należy jedynie stworzyć nową klasę, rozszerzyć ją o klasę abstrakcyjną *ParserTemplateStep.class* i zaimplementować wymagane metody (Spring sam zadba o dodanie nowej implementacji do listy dostępnych stron). Kod klasy *ParserTemplateStep* został przedstawiony na listingu poniżej.

```
public abstract class ParserTemplateStep {

  private ArticleWriter articleWriter;
  protected abstract void parse();
  protected abstract String parseBody(Document document);
  protected abstract String parseCategory(Document document);
  protected abstract String parseAuthor(Document document);
  protected abstract Set<String> getKeywords(Document document);
  protected abstract String getArticlesUrl(long page);
  protected abstract Set<String> extractArticleUrls(Document document);
  protected void parseTemplate(String articleUrl) {
    Document articleDocument = Jsoup.connect(articleUrl).get();
    Article article = tryGetArticleData(articleDocument);
    writeArticle(article);
  }

  protected Set<String> getArticleUrlsFromPage(long page) {
    String articlesUrl = getArticlesUrl(page);
    Document document = Jsoup.connect(articlesUrl).get();
    return extractArticleUrls(document);
  }

  private Article tryGetArticleData(Document doc) {
    String articleUrl = doc.location();
    String body = parseBody(doc);
    String title = parseTitle(doc);
    String author = parseAuthor(doc);
    String category = parseCategory(doc);
    Set<String> keywords = getKeywords(doc);
    String commaSeparatedKeywords = String.join(",", keywords);
    return Article.builder()
        .body(body)
        .title(title)
        .source(articleUrl)
        .author(author)
        .category(category)
        .keywords(commaSeparatedKeywords)
        .build();
  }
  protected String parseTitle(Document document) {
    String location = document.location();
        //utworzenie skrotu MD5 na podstawie adresu URL
```

```
    return DigestUtils.md5DigestAsHex(location.getBytes());
  }
  protected void writeArticle(Article article) {
    System.out.printf("Writing %s\n", article);
    articleWriter.write(article);
  }
}
```

Dla zwiększenia czytelności kodu pominięte zostały *gettery* i *settery* oraz obsługa wyjątków. W przypadku niektórych artykułów pobierane są również zbiory słów kluczowych (tagów) wyznaczanych przez autorów tekstów. W przypadku, gdy nie ma możliwości pobrania takiej informacji z artykułu, metoda *getKeywords*, zgodnie w dobrymi praktykami programowania, powinna zwracać pustą kolekcję. Przykładowa implementacja została przedstawiona na listingu poniżej.

```
@Component
@Order(10)
public class HistorykonStep extends ParserTemplateStep {
  private Logger log = LoggerFactory.getLogger(HistorykonStep.class);

  @Override
  public void parse() {
    long page = 1;
    while (page < 100) {
      log.info("Fetch page: {}", page);
      Set<String> articleUrlsFromPage = getArticleUrlsFromPage(page++);
      articleUrlsFromPage.forEach(this::parseTemplate);
    }
  }
  @Override
  protected Set<String> extractArticleUrls(Document document) {
    Elements titleContainers = document.getElementsByClass("item-list");
    return titleContainers.stream()
        .map(titleContainer -> titleContainer.getElementsByTag("a"))
        .map(Elements::first)
        .map(link -> link.attr("href"))
        .filter(url -> url.startsWith("https://historykon.pl/"))
        .collect(Collectors.toSet());
  }
  @Override
  protected String getArticlesUrl(long page) {
    return "https://historykon.pl/artykuly/page/" + page;
  }
  @Override
  protected String parseTitle(Document doc) {
    return Optional.ofNullable(doc.getElementsByClass("post-title"))
        .map(Elements::first)
        .map(Element::text)
        .orElse("");
  }
  @Override
  protected String parseBody(Document doc) {
    Element articleContent = doc.getElementsByClass("entry").first();
    return articleContent.text();
  }
  @Override
  protected String parseCategory(Document doc) {
    Element categoryContent = doc.getElementsByClass("category").first();
    return categoryContent.text();
  }
}
```

Do działania aplikacji wymagane jest udostępnienie listy artykułów ze strony źródłowej wraz z prostą paginacją, co jest wymagane przez metodę *getArticlesUrl* oraz *extractArticleUrl*, gdzie kolejne strony mogą zostać pobrane przez zmianę w adresie URL (strony doładowujące listę przez zapytania AJAX nie są obsługiwane). Kolejnym wymaganiem jest to, aby strona nie korzystała z języka JavaScript do renderowania strony HTML po stronie klienta, ponieważ biblioteka *Jsoup* nie będzie w stanie odnaleźć odpowiednich klas i tagów w strukturze DOM. W większości, strony z artykułami nie korzystają z takich rozwiązań ze względu na problemy z pozycjonowaniem SEO.

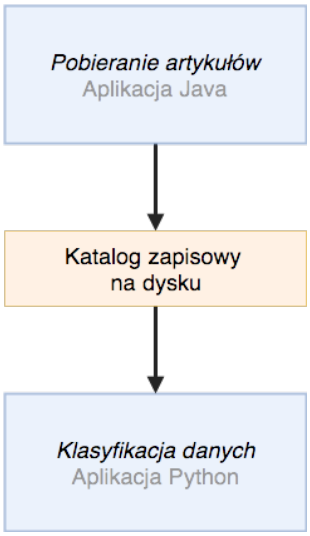
Schemat działania aplikacji jest bardzo prosty i pozwala na otrzymacie gotowych dokumentów w krótkim czasie. Program pobiera dokument z listą artykułów, następnie znajduje i odwiedza każdy znaleziony adres URL do artykułu. Artykuły zapisywane są na bieżąco, bez potrzeby pobierania wszystkich stron z artykułami, co jest zaletą, ponieważ prace programu można przerwać w momencie, kiedy liczba artykułów będzie wystarczająca. Działanie głównej pętli programu można przedstawić za pomocą pseudokodu.

Algorithm 1 Schemat pobierania artykułów

```
1: function FETCHARTICLES
2:     List<A> ← pobierz stronę o numerze N
3:     for dla każdej strony z artykułami A do
4:         List<U> ← znajdź adresy URL do artykułów
5:         for dla każdego adresu artykułu U do
6:             DOC ← pobierz dokument HTML
7:             ART ← znajdź w strukturze DOM potrzebne elementy
8:             zapisz artykuł do pliku
```

Aplikacja jest zdolna do rozdzielenia pracy na kilka wątków, każdy wątek pracuje wówczas nad jednym portalem, a kiedy skończy, zajmuje się kolejnym; w przypadku, gdy nie ma więcej stron do przetworzenia, jego praca jest kończona. Takie rozwiązanie było proste w implementacji i dawało wystarczające efekty pod kątem czasu i wydajności pracy. [9]

W pracy nie użyto żadnej bazy danych, ponieważ skomplikowałby to jedynie implementację oraz wymagałoby od użytkownika posiadania konkretnej bazy danych zainstalowanej lokalnie. Zdecydowano, że dane będą pobierane bezpośrednio na dysk twardy, co będzie bardziej uniwersalnym sposobem zapisu dokumentów. Co więcej, większość bibliotek do przetwarzania języka naturalnego oczekuje, że dane wejściowe będą zapisane na dysku w postaci osobnych dokumentów. Pozwoli to na ponowne wykorzystanie aplikacji pobierającej w innych pracach. Schemat przepływu danych został zaprezentowany na rysunku 3.1.



Rys. 3.1: Architektura dostępu do danych

3.2. Struktura dokumentów

Zapisywane dokumenty domyślnie mają strukturę XMLową, co zapewnia uniwersalność i kompatybilność między użytymi narzędziami.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<article>
    <author>Redakcja</author>
    <body>Egzekucja rozpoczęła się o godz. 20. [...]</body>
    <category>Historia</category>
    <keywords>narodowy dzień pamięci [...]</keywords>
    <source>http://www.focus.pl/artykul/1-m[...]</source>
    <title>1 marca - Narodowy Dzień [...]</title>
</article>
```

W niektórych przypadkach pobranie wszystkich danych było niemożliwe, dlatego mogą być one puste. Aplikacja, w fazie pobierania artykułów, odrzucała jednak te, które:

- nie posiadały tekstu,
- nie posiadały kategorii,
- nie posiadały źródła.

Zapisywane dokumenty przyjmowały nazwę z tytułu artykułu (w przypadku, gdy taki plik już istniał, był on nadpisywany); takie podejście pozwala na zbieranie artykułów fermentacyjnie, bez szkody dla aktualnie pobranych dokumentów. W przypadku, gdy tytuł artykułu był pusty, tworzony został skrót za pomocą funkcji haszującej *MD5* [35] z adresu źródła. Dzięki takiemu rozwiązaniu, katalog zawierał tylko unikalne artykuły pod względem treści. Pliki na dysku zapisywane były według źródła, z którego zostały pobrane; katalogi natomiast przyjmowały nazwy domen, tak jak zostało przedstawione na rysunku 3.2. W celu łatwej manipulacji i zmiany ścieżek, w których są zapisywane dokumenty, zaproponowano interfejs o nazwie **PathResolver.class**, wymagający implementacji dwóch metod: **String resolveRelativePath(Article article)** oraz **String resolveFileName(Article article)**.

```
public interface PathResolver {

  String resolveRelativePath(Article article);
  String resolveFileName(Article article);

}
```

Zmiana sposobu rozwiązywania ścieżek jest dokonywana poprzez zmianę implementacji w klasie **ArticleWriter.class**.

```
public interface ArticleWriter {

  void write(Article article) throws IOException;
  void setPathResolver(PathResolver strategy);

}
```

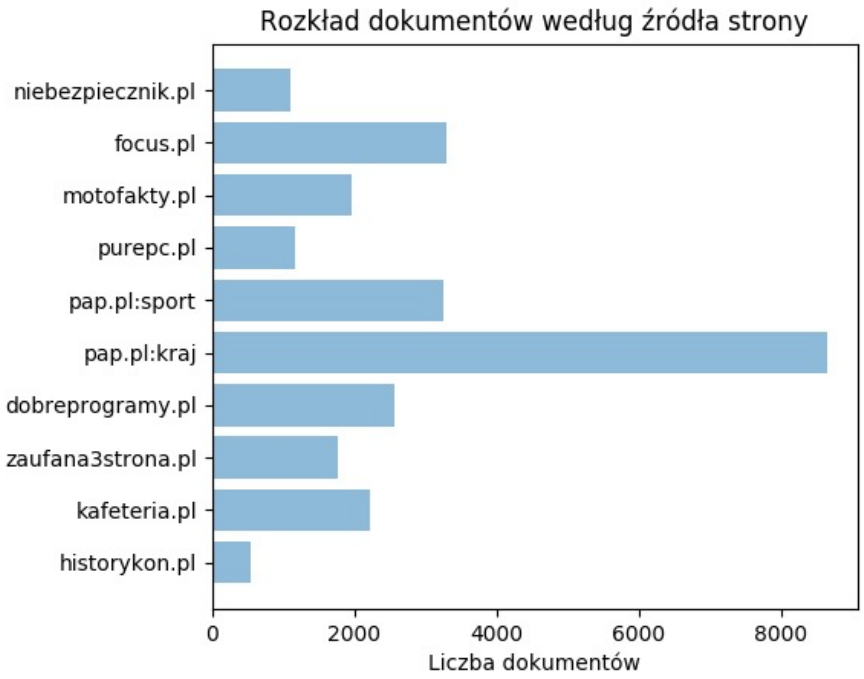
Nazwy wszystkich plików są unikalne względem katalogu, który został wyznaczony do zapisu. Mogą być bez przeszkód umieszczone w jednym katalogu, gdyby była taka potrzeba.

```
katalog docelowy
├── dobreprogramy.pl
│   ├── "8475.txt"
│   └── "8476.txt"
├── historykon.pl
│   ├── "3489.txt"
│   └── "3490.txt"
└── ..
```

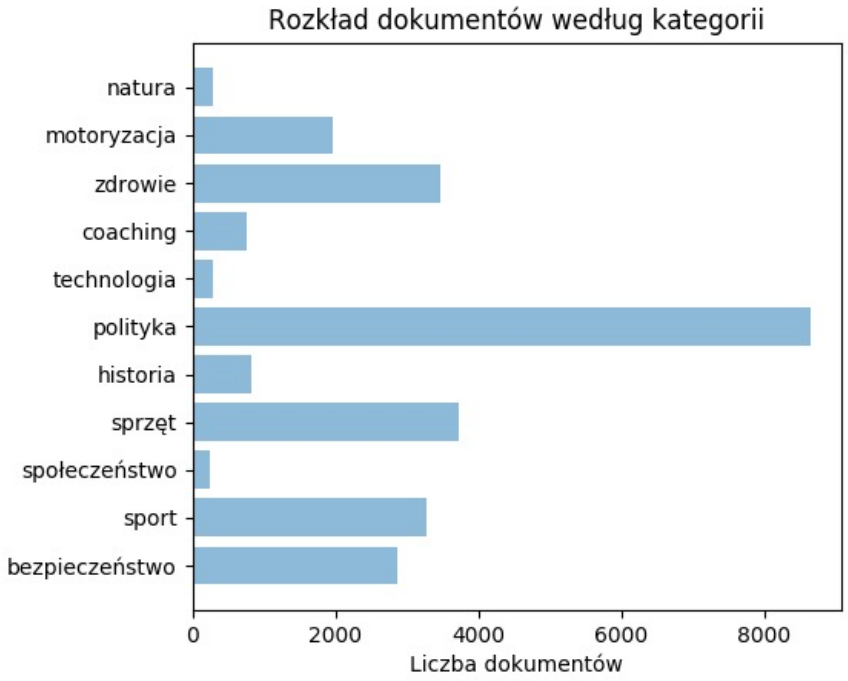
Rys. 3.2: Struktura plików pobranych dokumentów przez aplikację

3.3. Analiza zebranego korpusu

Zebrany korpus składa się z ponad 22 tysięcy dokumentów i został zebrany z 9 różnych portali internetowych. Najlepszym źródłem wysokiej jakości dokumentów była Polska Agencja Prasowa, dlatego z tego portalu pobrano najwięcej dokumentów do analizy. Rozkład według portalu (wyjątkiem jest serwis PAP, ze względu na ilość danych) przedstawiony został na wykresie 3.3.



Rys. 3.3: Rozkład pobranych artykułów według źródła



Rys. 3.4: Rozkład pobranych artykułów według kategorii

3.3.1. Selekcja rozpatrywanych klas

Na podstawie analizy rozkładu dokumentów na kategorie 3.4 wyselekcjonowano 7 grup tematycznych które będą analizowane w pracy. [39] Lista wybranych kategorii została zaprezentowana w tabeli 3.1. Kategorie z liczbą artykułów poniżej 100 zostały odrzucone.

Tab. 3.1: Rozłożenie liczby artykułów poszczególne kategorie

kategoria	**źródło**	**liczba dokumentów**
motoryzacja	motofakty.pl	1890
zdrowie	kafeteria.pl focus.pl/zdrowie	3823
polityka	pap.pl/kraj	8351
sport	pap.pl/sport focus.pl/sport	2759
bezpieczeństwo	zaufanatrzeciastrona.pl dobreprogramy.pl niebezpiecznik.pl	2311
sprzęt	dobreprogramy.pl	3942
historia	historykon.pl	1249

3.3.2. Ograniczenie zbioru dokumentów

Ograniczenie zbioru dokumentów polegało na odrzuceniu dokumentów, które posiadały mniej niż 200 znaków. Artykuły takie prawdopodobnie były reklamami, które mogłyby przeszkodzić w dalszej analizie. Następnie, do każdej kategorii wybierano 500 dokumentów, aby dane wejściowe dla każdej klasy były równe pod kątem ilościowym. Przetworzone pliki zapisane zostały do nowego katalogu z rozróżnieniem na kategorie, co zostało zaprezentowane na rysunku 3.5.

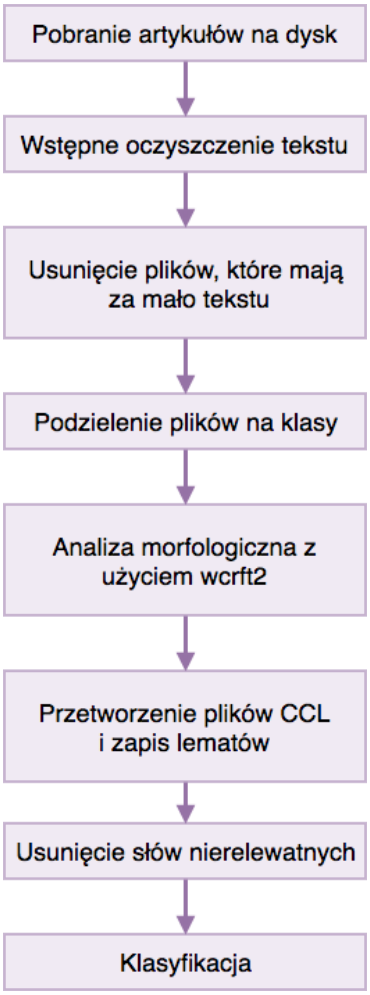
```
według_kategorii
├── sport
│   ├── "2475.txt"
│   └── "2476.txt"
├── polityka
│   ├── "6489.txt"
│   └── "6490.txt"
└── ..
```

Rys. 3.5: Struktura plików podzielona według klas

Tak skonstruowana struktura plików umożliwiała manipulację wbudowanymi narzędziami w bibliotece *SciKit Learn*.

3.4. Przetwarzanie wstępne

W fazie wstępnego przetwarzania tekstu z pobranych plików w miarę możliwości usuwane są zbędne dane, takie jak adresy e-mail i cytaty. Forma tekstów jest ujednolicana, teksty zostają poddane lematyzacji i usunięciu słów nierelewantnych. Ogólny proces przepływ danych podczas przetwarzania wstępnego został pokazany na rysunku 3.6.



Rys. 3.6: Schemat kolejnych kroków przetwarzania danych

3.4.1. Analiza morfologiczna tekstu

Istotnym etapem przetwarzania tekstu było zastosowanie analizatora morfologicznego. W pracy wykorzystany został analizator morfologiczny Morfeusz 2 ze słownikiem SGJP, udostępniony w postaci REST API. Praca z serwisem odbywa się w sposób asynchroniczny, oznacza to, że najpierw należało przesłać plik tekstowy do analizatora, a następnie odpytywać serwer co jakiś czas, aby sprawdzić czy praca się zakończyła. Przykład użycia analizatora WCRFT2 przez REST API:

- Wysłanie pliku do serwera.

```
# [POST] http://ws.clarin-pl.eu/nlprest2/base/upload
# Content-Type: 'binary/octet-stream'

def upload(file_path):
    with open(file_path, "rb") as file:
        file_bytes = file.read()
    req = urllib.request.Request(url + '/upload/', file_bytes,
    {'Content-Type': 'binary/octet-stream'})
    return urllib.request.urlopen(req).read().decode("utf-8")
```

  Serwer w odpowiedzi wyśle ID zapisanego pliku (id_pliku).

- Rozpoczęcie przetwarzania tekstu z użyciem analizatora WCRFT2.

```
# [POST] http://ws.clarin-pl.eu/nlprest2/base/startTask
# data = {
# 'lpmn': 'any2txt|wcrft2',
# 'user': '209897@student.pwr.edu.pl',
# 'file': id_pliku
# }

def process(data):
    json_data = json.dumps(data).encode('utf-8')
    start_task_req = urllib.request.Request(url=url + '/startTask/')
    start_task_req.add_header('Content-Type', 'application/json')
    start_task_req.add_header('Content-Length', len(json_data))
    task_id = urllib.request
        .urlopen(start_task_req, json_data)
        .read()
        .decode("utf-8")
    time.sleep(0.1)
```

  Serwer w odpowiedzi wyśle ID rozpoczętego zadania (id_zadania).

- Sprawdzenie czy analiza się zakończyła.

```
# [GET] http://ws.clarin-pl.eu/nlprest2/base/getStatus/{id_zadania}
    reqUrl = url + '/getStatus/' + task_id
    get_status_req = urllib.request.Request(reqUrl)
    status_response = urllib.request.urlopen(get_status_req)
    data = json.load(status_response)
    while data["status"] == "QUEUE" or data["status"] == "PROCESSING":
        time.sleep(0.1)
        get_status_req = urllib.request.Request(reqUrl)
```

```
        status_response = urllib.request.urlopen(get_status_req)
        data = json.load(status_response)
    if data["status"] == "ERROR":
        print("Error " + data["value"])
        return None
    return data["value"]
```

Serwer w odpowiedzi poinformuje o błędzie ("*ERROR*"), przetwarzaniu ("*PROCESSING*") lub oczekiwaniu na przetworzenie ("*QUEUE*").

- Pobranie pliku wynikowego.

```
# [GET] http://ws.clarin-pl.eu/nlprest2/base/download/{id_pliku}
def process_file(source_file, output_file):
    file_id = upload(source_file)
    print("[F] Processing: " + source_file)
    data = {'lpmn': lpmn, 'user': user, 'file': file_id}
    data = process(data)
    data = data[0]["fileID"]
    download_req = urllib.request.Request(url + '/download' + data)
    content = urllib.request
        .urlopen(download_req)
        .read()
        .decode("utf-8")

    output_file = output_file + ".ccl.xml"
    save_text(output_file, content)
```

W projekcie wzorowano się na udostępnionym kodzie dla języka Python 2.6 [6], który został odpowiednio dostosowany do nowszej wersji języka.

3.4.2. Ustalenie podstawowej postaci wyrazów

Formy podstawowe wyrazów (haseł) zostały ustalone dzięki analizatorowi WCRFT2, który zwracał XML (CCL) jako plik wynikowy. [4]

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE chunkList SYSTEM "ccl.dtd">
<chunkList>
 <chunk id="ch1" type="p">
  <sentence id="s1">
   <tok>
    <orth>Niemalże</orth>
    <lex disamb="1">
        <base>niemalże</base><ctag>qub</ctag>
    </lex>
   </tok>
   <tok>
    <orth>od</orth>
    <lex disamb="1">
        <base>od</base><ctag>prep:gen:nwok</ctag>
    </lex>
   </tok>
```

```
    <tok>
     <orth>początku</orth>
     <lex disamb="1">
          <base>początek</base><ctag>subst:sg:gen:m3</ctag>
    </lex>
    </tok>
    </sentence>
    </chunk>
</chunkList>
```

Do odczytania pliku wykorzystano domyślny parser w języku Python o nazwie *ElementTree*. Napisana funkcja *ccl_to_lemma* przyjmowała plik w formacie XML (CCL) oraz zapisywała plik tekstowy z oddzielonymi przecinkami formami podstawowymi.

```
def ccl_to_lemma(input_file, output_file):
    tree = ElementTree.parse(input_file)
    root = tree.getroot()
    base_words = []
    for token in root.getiterator('tok'):
        base = token.find('lex').find('base').text
        base_words.append(base)

    output_file = output_file + ".lemma"
    content = " ".join(base_words)
    save_text(output_file, content)
```

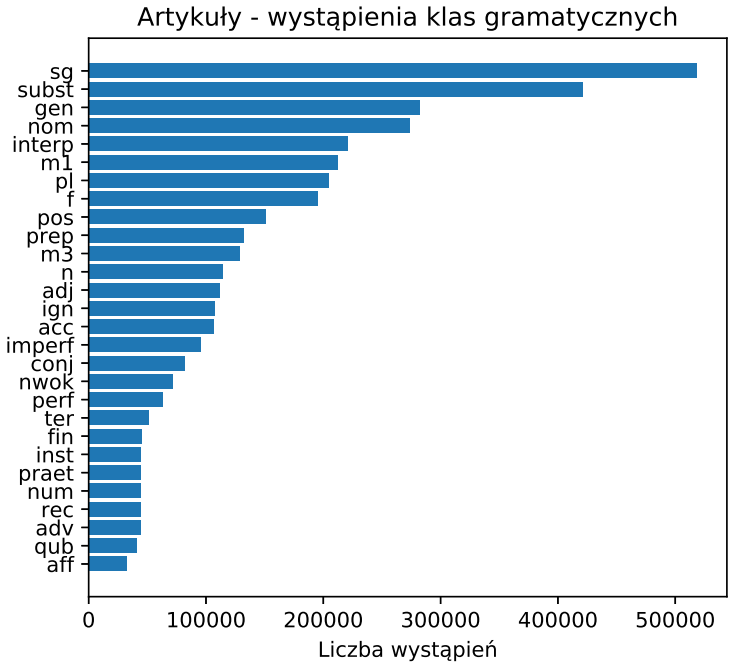
Z uwagi na wykorzystanie gotowego analizatora morfologicznego, w pracy przyjęto zbiór tagów gramatycznych autorstwa Adama Przepiórkowskiego [33], zaprezentowanych w tabeli 3.2, który jest wykorzystywany przez ten analizator. Pełny spis klas można znaleźć na stronie `http://nkjp.pl/poliqarp/help/ense2.html`.

Tab. 3.2: Klasy gramatyczne

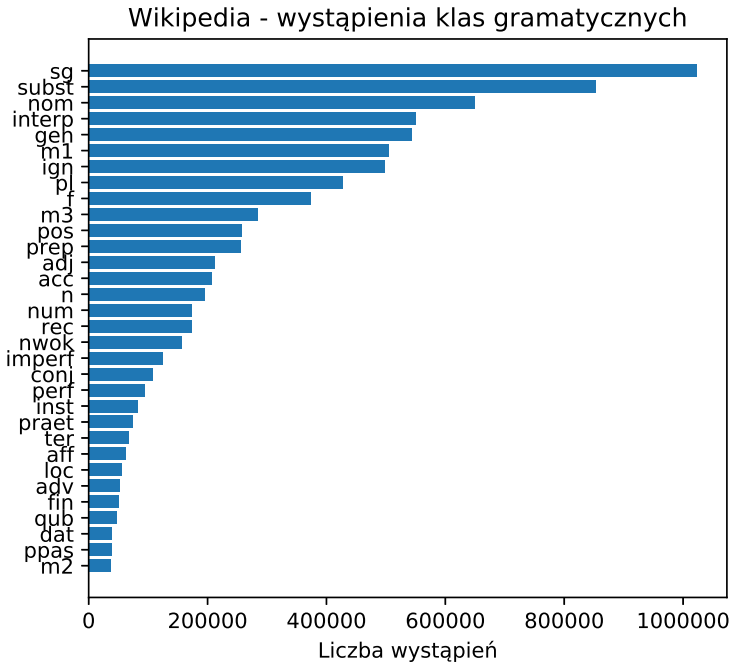
fleksem	**skrót**
rzeczownik	subst
rzeczownik deprecjatywny	depr
liczebnik główny	num
liczebnik zbiorowy	numcol
przymiotnik	adj
przymiotnik przyprzym.	adja
przymiotnik poprzyimkowy	adjp
przymiotnik predykatywny	adjc
przysłówek	adv
zaimek nietrzecioosobowy	ppron12
zaimek trzecioosobowy	ppron3
zaimek siebie	siebie
forma nieprzeszła	fin
forma przyszła być	bedzie
aglutynant być	aglt
pseudoimiesłów	praet
rozkaźnik	impt
bezosobnik	imps
bezokolicznik	inf
im. przys. współczesny	pcon
im. przys. uprzedni	pant
odsłownik	ger
im. przym. czynny	pact
im. przym. bierny	ppas
winien	winien
predykatyw	pred
przyimek	prep
spójnik współrzędny	conj
spójnik podrzędny	comp
kublik	qub
skrót	brev
burkinostka	burk
wykrzyknik	interj
interpunkcja	interp
ciało obce	xxx
forma nierozpoznana	ign

3.4.3. Analiza słów nierozpoznanych

Po przeprowadzeniu analizy morfologicznej dokonano analizy klas gramatycznych w obu korpusach 3.7. Na przedstawionych rysunkach widać, że dużą część korpusu stanowią wyrazy, które nie zostały rozpoznane przez analizator WCRFT2 (klasa *ign*)



(a) Artykuły

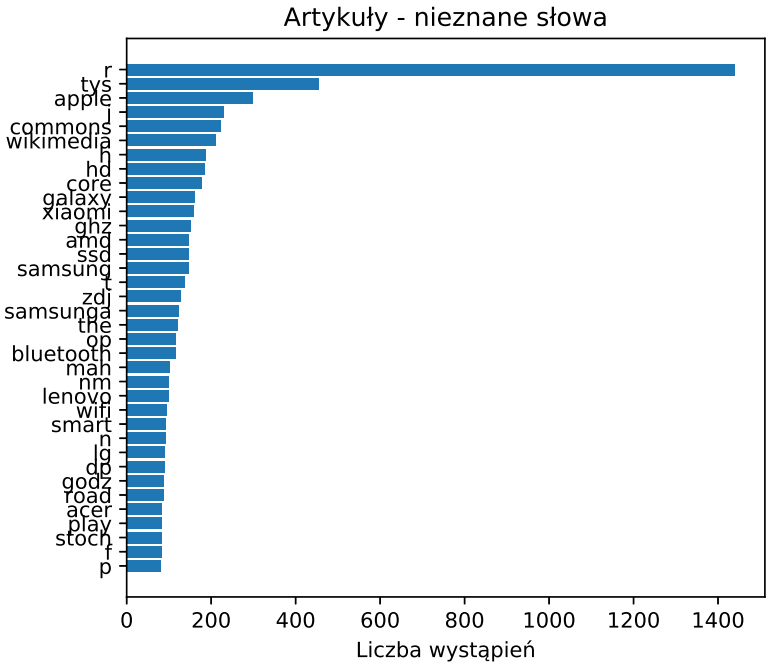


(b) Wikipedia

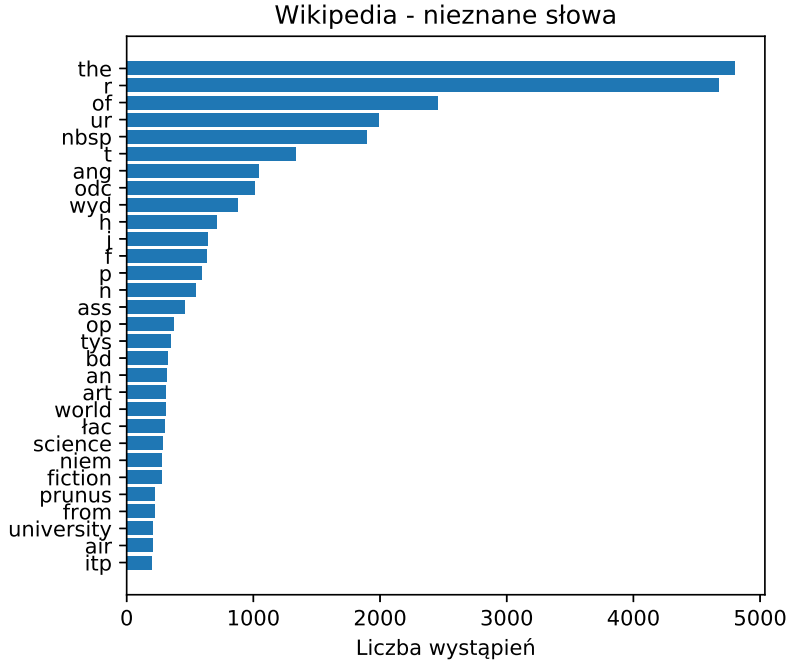
Rys. 3.7: Porównanie wystąpień klas gramatycznych w obu korpusach

W przypadku korpusu *Artykuły* nie rozpoznano około 5.19% wyrazów w stosunku do całego korpusu. W korpusie *Wikipedia* wartość ta wynosi tylko 1.12%. Rozbieżności między tymi

wartościami wynikają najprawdopodobniej z literówek, jakie mogły zaistnieć podczas pisania artykułów. W artykułach z platformy Wikipedia często są one poprawiane i redagowane przez więcej niż jedną osobę. W pełni automatyczna poprawa tekstu byłaby zadaniem trudnym; takie działanie mogłoby skutkować pogorszeniem wyników, jeśli słowo zostałoby nieprawidłowo zinterpretowane. W celu poprawy jakości zebranych danych, wyznaczone zostały najczęściej występujące nierozpoznane słowa zaprezentowane na rysunku 3.8.



(a) Artykuły



(b) Wikipedia

Rys. 3.8: Porównanie wystąpień nierozpoznanych form w obu korpusach

Formami nierozpoznanymi okazały się w głównej mierze nazwy własne. W obu korpusach są one częścią istotnych cech, które sugerują daną grupę tematyczną, dlatego pozostały w korpusach bez zmian. *r* - prawdopodobnie jest to pozostałość po podziale wartości liczbowej roku oraz skrótu *r.*. Usunięto jedynie kilka wyrazów

- *nbsp* (ang. non breaking space) - kod HTML dla spacji niełamliwej używanej w kodzie [1]
- *commons* oraz *wikipedia* - od nazwy własnej Wikipedia Commons, prawdopodobnie błąd podczas pobierania

Należy jednak pamiętać, że klasyfikowane artykuły zawsze zawierać będą pewien odsetek tekstów, które są błędnie napisane. Dlatego zdecydowano się na usunięcie jedynie oczywistych błędów popełnionych podczas implementacji programów zbierających korpusy.

3.4.4. Usunięcie wyrazów nierelewantnych

Podczas analizy tekstu istotnym aspektem jest jakość analizowanych danych i dokumentów. W pracach dotyczących klasyfikacji tematycznej częstą praktyką jest usuwanie słów nierelewantnych w fazie przetwarzania [26][25][28]. Do słów nierelewantnych zalicza się wyrazy, które najczęściej występują w danym języku (ang. stop words). Dzięki temu współczynnikowi, jakość klasyfikacji ulega poprawie poprzez pomijanie mało istotnych wyrazów. Dodatkową zaletą jest mniejsze zapotrzebowanie na pamięć oraz moc obliczeniową, ponieważ przetwarzanych danych jest mniej. W pracy wykorzystano listę wyrazów, która została udostępniona na stronie *https://www.ranks.nl/stopwords/polish* [2] w postaci tabeli.

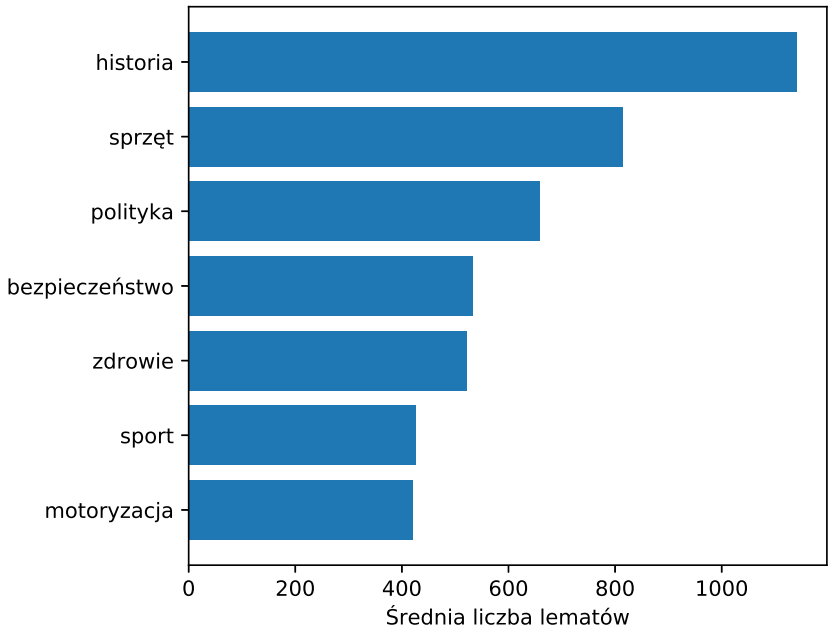
3.4.5. Analiza zawartości korpusu

Kluczowym etapem okazała się ręczna analiza zawartości korpusu z wykorzystaniem takich metod jak modelowanie tematów (ang. topic modeling). Modelowanie tematów jest techniką analizy dużych zbiorów dokumentów; modele automatycznie wyszukują tematy w tekście z wykorzystaniem ukrytych zmiennych losowych. Technika ta często jest wykorzystywana do rekomendacji tagów, wyszukiwania słów kluczowych, kategoryzacji tekstów. [46] W pracy wykorzystano gotową implementację *LDA* (Latent Allocation Dirichlet) z biblioteki *SciKitLearn* w celu analizy zawartości zebranego korpusu. Wyniki analizy 3.3 wykazały, że zawartość korpusu, mimo starannych prób usunięcia wszystkich wyrazów mogących powodować błędne wyniki, nie była idealna.

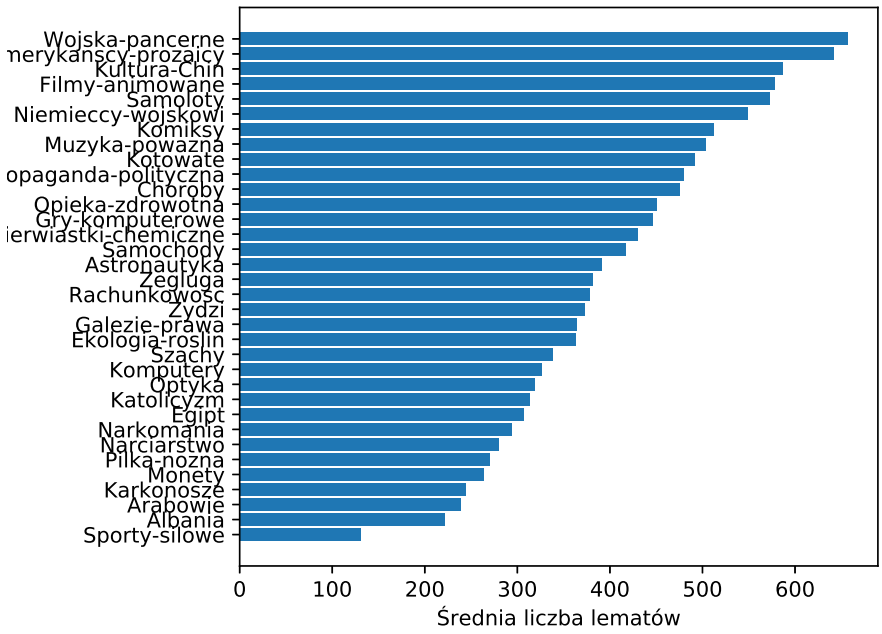
Tab. 3.3: Wynik modelowania tematycznego przy pomocy LDA

Topic 0	mecz rok polska 00 minuta pap pierwsza druga miejsce raz
Topic 1	urządzenie model smartfon móc sprzęt nowa ekran producent można wersja
Topic 2	skóra kobieta móc należeć choroba woda często czas zabieg można
Topic 3	samochód pojazd prezydent kierowca powiedzieć zostać ustawa złoty projekt auto
Topic 4	metr pkt polska konkurs niemcy pap stefan rok miejsce już pozycja seria
Topic 5	niebezpiecznik dana serwer firma atak serwis móc sieć informacja błąd
Topic 6	rok polski polska lato czas były państwo pap była wojna więc

W zawartości korpusu znalazły się słowa, które mogły dać klasyfikatorom jednoznaczne podpowiedzi, do której grupy przynależy tekst, na przykład *pap* oraz *niebezpiecznik*. W celu usunięcia niechcianych wyrazów dodano je do listy słów nierelewantnych. Analiza była przeprowadzana tak długo, dopóki wszystkie niechciane słowa nie zostały usunięte z wyników pracy *LDA*. Ostatecznie zliczono wszystkie formy podstawowe wyrazów dla każdej kategorii i przedstawiono je na rysunku 3.9.



(a) Artykuły



(b) Wikipedia

Rys. 3.9: Rozkład średniej liczby lematów według kategorii

3.5. Metody klasyfikacji tekstu

W pracy zdecydowano się na przetestowanie dwóch różnych metod klasyfikacji, *Bag-Of-Words* oraz *fastText*, który jest pochodną *word2vec*.

3.5.1. Bag-Of-Words

Aby móc poddać tekst klasyfikacji, konieczne jest wstępne przetworzenie tekstu wejściowego i przedstawienie go w postaci macierzy lub, w przypadku prostszych metod, wektora wierszowego dla danego tekstu. Istnieje wiele różnych sposobów wektoryzacji dokumentów; najprostszym i najczęściej stosowanym [13] jest model *Bag-Of-Words*. Każdy dokument jest przedstawiany w postaci wektora wierszowego, który zawiera liczbę wystąpień słowa w danym dokumencie.[22]

Przykład Bag-of-Words

Utworzenie *Bag-of-Words* na przykładzie dwóch dokumentów [47] wyglądałoby następująco:

```
dokument 1: "The Sun is a star. Sun is beautiful."
dokument 2: "The Moon is a satellite."
```

Na podstawie zadanych dwóch dokumentów utworzony zostanie słownik z unikalnymi słowami. Liczba wszystkich unikalnych słów będzie również liczbą kolumn w każdym wektorze wierszowym dla całego korpusu dokumentów.

Przykładowe dokumenty składają się z 8 unikalnych słów, dlatego każdy dokument jest reprezentowany jako 8-elementowy wektor wierszowy i został przedstawiony na rysunku 3.4.

Tab. 3.4: Tabela wektorów Bag-Of-Words

słowo	**dokument 1**	**dokument 2**
The	1	1
Sun	2	0
is	2	1
a	1	1
star	1	0
beautiful	1	0
Moon	0	1
satellite	0	1

Cyfra pierwsza ’1’ w obu dokumentach oznacz, że w dokumencie tylko raz wystąpiło słowo ’The’. Druga cyfra (’2’) w dokumencie pierwszym oznacza, że słowo ’Sun’ wystąpiło dwa razy, natomiast w dokumencie drugim (’0’) ani razu nie pojawiło się słowo ’Sun’.

Listing 3.1: Utworzone wektory

```
dokument 1: [1, 2, 2, 1, 1, 1, 0, 0]
dokument 2:  [1, 0, 1, 1, 0, 0, 1, 1]
```

Ustalenie wag - *tf-idf*

Algorytm *tf-idf* służy do określania wag słów względem całego korpusu. Jeśli słowo pojawia się często we wszystkich dokumentach, jego znaczenie maleje, ponieważ prawdopodobnie nie wnosi ono żadnej praktycznej wiedzy. *tf-idf* mierzy znaczenie, a nie częstotliwość i jego wartość jest obliczana według wzoru[17]:

$$Wt, d = TFt, dlog(N/DFt) \tag{3.1}$$

Gdzie:

- TFt, d - oznacza liczbę wystąpień słów t w dokumentach d,
- DFt - oznacza liczbę dokumentów zawierających słowo t,
- N - oznacza liczbę wszystkich dokumentów w korpusie.

Implementacja Bag-Of-Words

W pracy użyto narzędzia o nazwie *CountVectorizer* z biblioteki *SciKit Learn*, która służy do wyodrębniania cech numerycznych z treści tekstowej, a mianowicie[30]:

- tokenizuje ciągi znaków,
- zlicza wystąpienia tokenów w każdym dokumencie,
- usuwa *stop words* z zadanej listy,
- normalizuje i ustala wagi, liczbę próbek / liczbę dokumentów.

Tekstem wejściowym dla zastosowanego narzędzia *CountVectorizer* były zlematyzowane teksty (hasła) uzyskane w fazie przetwarzania wstępnego. Kapitalizacja wszystkich wyrazów została usunięta i wszystkie litery zostały zamienione na małe. Efektem działania *CountVectorizera* jest macierz terminów *dokument* (ang. document-term), która opisuje częstotliwość występowania tokenów w zbiorze dokumentów. Kolumny odpowiadają kolejnym wyrazom, a wiersze dokumentom, tak jak przedstawiono w tabeli 3.5.

Tab. 3.5: Macierz dokument-wyrażenie

	The	Sun	is	a	star	beautiful	Moon	satellite
D1	1	2	2	1	1	1	0	0
D2	1	0	1	1	0	0	1	1

Końcową optymalizacją jest zastosowanie normalizacji *tf-idf* w celu ustalenia wag lematów w macierzy. Tak przygotowana macierz może już być wprowadzona do klasyfikatora w celu nauki oraz późniejszych testów. Implementacja została zaprezentowana na listingu poniżej.

```
#funkcja uruchamiajaca test BagOfWords
def invoke(
      data_train, data_test, #dane uczace
      target_train, target_test, #dane testowe
      clf, iterations): #klasyfikator, ilosc iteracji

      #zmienne tymczasowe do wyznaczenia wartosci srednich
    fit_time_acc = 0
    predict_time_acc = 0
    accuracy_acc = 0
    f1_acc = 0

  #petla iteracyjna
    for iter_index in range(0, iterations):
      #wykonanie obliczen dla jednego kroku
        predicted, fit_time, predict_time = iter_step(
          data_train, data_test, target_train, clf)

        #sumowanie obliocznych wynikow
        fit_time_acc += fit_time
        predict_time_acc += predict_time
        accuracy_acc += accuracy_score(target_test, predicted)
        f1_acc += f1_score(target_test, predicted, average='micro')

  #wyznaczenie wartosci srednich
    mean_fit_time = fit_time_acc / iterations
    mean_predict_time = predict_time_acc / iterations
    mean_accuracy = accuracy_acc / iterations
    mean_f1_acc = f1_acc / iterations

  #zwrocenie wynikow
    return mean_f1_acc, mean_accuracy, mean_fit_time, mean_predict_time


def iter_step(data_train, data_test, target_train, classifier):
  #pobranie listy slow nierelewatnych
    stop_words_list = read_stop_words_list("../data/stop_words/list.txt")
    countVectorizer = CountVectorizer(stop_words=stop_words_list)
    tfidf_vectorizer = TfidfVectorizer()

      #stworzenie procesu
    pipeline = Pipeline([
      ('cnt', countVectorizer),
        ('vect', tfidf_vectorizer),
        ('clf', classifier)])

  #nauczenie wszystkich elementów w 'pipeline'
    start = time.time()
    pipeline = pipeline.fit(data_train, target_train)
    end = time.time()
```

```
    #wyznaczenie czasu nauki
  fit_time = (end - start)

#klasyfikacja danych testowych
  start = time.time()
  predicted = pipeline.predict(data_test)
  end = time.time()
    #wyznaczenie czasu predykcji
  predict_time = (end - start)

#zwrocenie wyników
  return predicted, fit_time, predict_time
```

Badane klasyfikatory

W pracy zdecydowano się na wykorzystanie trzech klasyfikatorów; do tego celu wybrano 3 najczęściej wykorzystywane klasyfikatory w pracach dotyczących klasyfikacji tematycznej. [40] [3][8]

- **Wielomianowy naiwny klasyfikator bayesowski** (ang. Multinomial Naive Bayes) - metoda, która określa przynależność danego obiektu do klasy na podstawie prawdopodobieństwa. [5] Klasyfikacja odbywa się przy wykorzystaniu cech z wartościami dyskretnymi (np. liczba słów dla klasyfikacji tekstu). Rozkład wielomianowy zwykle wymaga liczby elementów całkowitych. Jednak w praktyce mogą być to również liczby ułamkowe, występujące w przypadku wektoryzacji *TF-IDF*. [30]

- **Liniowa maszyna wektorów nośnych** (ang. Linear Support Vector Machine) - SVM to zestaw nadzorowanych metod uczenia używanych do klasyfikacji. Metoda polega na wyznaczaniu granic hiperpłaszczyzny oddzielającej próbki każdej klasy. W pracy wykorzystano SVM z liniową funkcją separującą; oznacza to, że przestrzenie są tak dzielone, aby odległość pomiędzy każdym jej punktem, a najbliższym punktem innej płaszczyzny był jak największy. [5]

- **Drzewo decyzyjne** (ang. Decision Tree) - to nadzorowana metoda uczenia stosowana do klasyfikacji i regresji. Tworzony jest model, który jest w stanie klasyfikować dane poprzez naukę prostych reguł decyzyjnych wywnioskowanych z danych uczących. Im głębsze drzewo, tym bardziej złożone reguły decyzyjne. [30]

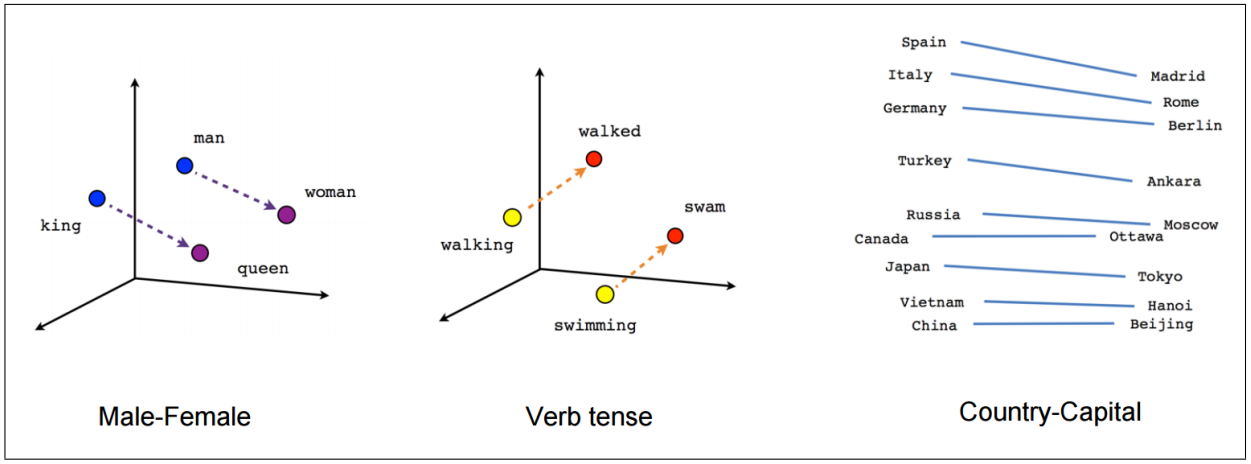
3.5.2. fastText

fastText to metoda klasyfikacji, stworzona w laboratorium AI Research (FAIR) w Facebooku jako projekt open-source, aby sprostać potrzebie zrozumienia znaczenia dużych zbiorów tekstów. Biblioteka została zaprojektowaną tak, aby ułatwić tworzenie skalowalnych rozwiązań do reprezentacji i klasyfikacji tekstu. Jest to stosunkowo nowa metoda wykorzystywana do kategoryzacji tekstu. W założeniu twórców ma ona umożliwiać klasyfikację bardzo dużych ilości danych w krótkim czasie przy niskich wymaganiach sprzętowych. Według autorów, *fastText* potrafi klasyfikować 500 tysięcy zdań na 300 tysięcy kategorii w mniej niż 5 minut na zwykłych komputerach domowych. [10]

FastText wykorzystuje technikę osadzonych słów (ang. word embedding) i domyślnie używa klasyfikatora Softmax [40]. Wykorzystuje także tablicę przeglądową (ang. lookup table) z haszowaniem dla n-gramów, składających się ze słów lub znaków. Każdy dokument reprezentowany jest jako średnia z osadzania słów. Zaletą tej metody jest zdolność do tworzenia wektorów dla dowolnych słów, nawet tych, które są niepoprawnie napisane (ponieważ w rzeczywistości wektory te zbudowane są z podłańcuchów znaków w nim zawartych). Dzięki temu, wektory mogą być budowane na podstawie złączonych ze sobą tylko dwóch słów. Istotną zaletą jest również zdecydowanie krótszy czas klasyfikacji oraz nauki w porównaniu do innych metod. Jest to rozszerzenie metody *word2vec*, w której każde słowo w korpusie jest traktowane jako atomowa jednostka i generowany jest dla niej oddzielny wektor. [22] *fastText* domyślnie ignoruje kolejność słów, podobnie jak metoda *BoW*. Aby uwzględnić lokalną kolejność słów, *fastText* pozwala na użycie n-gramów, jednak funkcja ta ostatecznie nie była używana w eksperymentach.[21]

Działanie word2vec

Modele *word2vec* oraz *fastText* zostały stworzone przez zespół prowadzony przez Tomasa Mikolova z Google. Algorytm przyjmuje duży korpus tekstu jako wejście i tworzy przestrzeń wektorową, [14] zazwyczaj o kilkuset wymiarach z każdym unikalnym wyrazem w korpusie, któremu przypisuje odpowiedni wektor w przestrzeni. Wektory słów są umieszczane w przestrzeni wektorowej w taki sposób, że słowa, które mają wspólne konteksty w korpusie, znajdują się blisko siebie w przestrzeni. Przykład takich relacji został zaprezentowany na rysunku 3.10. [7]



Rys. 3.10: Liniowa zależność word2vec

Implementacja Python

Biblioteka *fastText* została oryginalnie napisana w języku C++ i nie ma dostępnych żadnych oficjalnych *wrapperów* do języka Python. Jednak w sieci istnieje wiele rozwiązań, które pozwalają na wykorzystanie *fastText* w języku Python. W pracy wykorzystano wrapper o nazwie *ShallowLearn* (więcej informacji `https://github.com/giacbrd/ShallowLearn`), który jest zbiorem modeli uczenia opartym na płytkich podejściach do sieci neuronowych (np. *word2vec* i *fastText*). Do wykorzystanego wrappera należało również stworzyć odpowiedni adapter, który realizował funkcję konwertera danych wejściowych do oczekiwanego przez wrapper format. Dodatkowo otrzymywane przez *ShallowLearn* wyniki musiały zostać poddane odpowiedniemu przetworzeniu tak, aby dane wyjściowe były reprezentowane w takiej samej postaci, jak ma to miejsce w przypadku implementacji *Bag-Of-Words*. W tym celu stworzona została klasa *calc_adaper.py*, która poprawiała kompatybilność z biblioteką *SciKit-Learn*. Możliwe stało się wykorzystanie metod dostarczanych przez interfejs biblioteki, *fit* oraz *predict*.

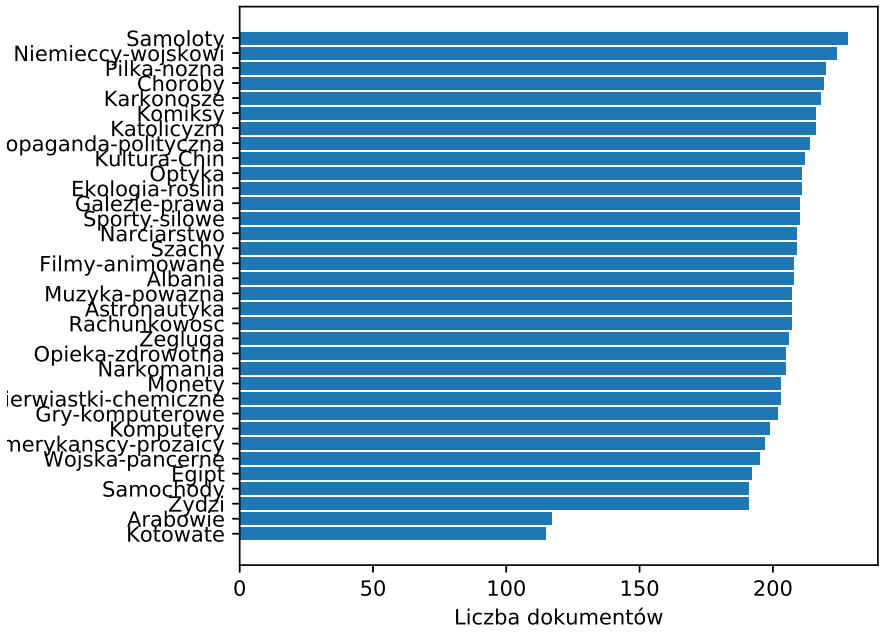
Rozdział 4

Badanie skuteczności metod

Wszystkie testy zostały przeprowadzone 10 razy z tymi samymi parametrami lecz za każdym razem z nowo wylosowanymi dokumentami, poddanymi wcześniej fazom przetwarzania, lematyzacji, wektoryzacji, a następnie klasyfikacji. Wyniki z iteracji tych samych testów zostały uśrednione i przedstawione na wykresach.

4.1. Korpus porównawczy

Do celów porównawczych wykorzystano dodatkowo korpus o nazwie *Wiki train - 34 categories*, dostępny w repozytorium Clarin [44]. Składa się on z 6800 dokumentów podzielonych na 34 klasy. Został on poddany tym samym procesom przetwarzania, co zebrany korpus z artykułami. Poszczególne kroki przetwarzania zostały pokazane na rysunku 3.6.



Rys. 4.1: Porównanie ilości dokumentów dla każdej klasy dla korpusu Wikipedii

4.2. Badane parametry

Do określenia jakości wykorzystano cztery różne powszechnie stosowane parametry miar jakościowych [40] [8]. Dodatkowo zmierzony został czas pracy każdego z klasyfikatorów nad różnymi etapami pracy.

- **czułość** (ang. recall) - jest liczbą poprawnych pozytywnych wyników podzieloną przez liczbę wszystkich odpowiednich próbek (wszystkie próbki, które powinny zostać zidentyfikowane jako pozytywne). [30] Czułość określa się wzorem:

$$R = \frac{T_p}{T_p + F_n} \tag{4.1}$$

  Gdzie:
  - T_p - oznacza liczbę poprawnie rozpoznanych dokumentów
  - F_n - oznacza liczbę niepoprawnie nierozpoznanych dokumentów

- **precyzja** (ang. precision) - jest liczbą poprawnych pozytywnych wyników podzieloną przez liczbę wszystkich pozytywnych wyników zwróconych przez klasyfikator. [30] Precyzję określa się wzorem:

$$P = \frac{T_p}{T_p + F_p} \tag{4.2}$$

  Gdzie:
  - T_p - oznacza liczbę poprawnie rozpoznanych dokumentów
  - F_p - oznacza liczbę niepoprawnie rozpoznanych dokumentów

- **miara f1** (ang. f1-score) - w statystycznej analizie klasyfikacji jest to miara dokładności testu. Uwzględnia ona dwa wcześniejsze parametry: precyzję oraz czułość. Jest średnią harmoniczną, gdzie wynik F1 osiąga najwyższą wartość 1 (doskonała precyzja i czułość), a najgorszy - 0. [20]

$$F1 = 2 * \frac{P * R}{P + R} \tag{4.3}$$

  Gdzie:
  - P - oznacza precyzję opisywaną we wzorze 4.2
  - R - oznacza czułość opisywaną we wzorze 4.1

- **dokładność** (ang. accuracy) - stosunek między poprawnie sklasyfikowanymi dokumentami do wszystkich dokumentów. [30]

$$A = \frac{T_p + T_n}{T_p + T_n + F_p + F_n} \tag{4.4}$$

  Gdzie:
  - T_p - oznacza liczbę poprawnie rozpoznanych dokumentów
  - T_n - oznacza liczbę poprawnie nierozpoznanych dokumentów
  - F_p - oznacza liczbę niepoprawnie rozpoznanych dokumentów
  - F_n - oznacza liczbę niepoprawnie nierozpoznanych dokumentów

Dodatkowo w testach uwzględniono inne miary niż jakościowe, a mianowicie:

- **czas nauki klasyfikatora** - czas wyrażony w sekundach, potrzebny systemowi na naukę danego klasyfikatora danymi wejściowymi. Ściślej mówiąc, jest to czas potrzebny na wykonanie się metod; był odpowiedzialny za nauczenie modelu dla każdego użytego klasyfikatora.
- **czas klasyfikacji** - czas wyrażony w sekundach, potrzebny systemowi na klasyfikację zbioru dokumentów.
- **całkowity czas pracy** - czas wyrażony w sekundach, jest to suma czasu potrzebnego na naukę oraz klasyfikację zbioru dokumentów.

4.3. Wyniki testów

Całość badanego korpusu z artykułami składała się z 2600 artykułów, podzielonych na 7 klas, po 380 dokumentów dla każdej kategorii. Szczegółowe informacje dotyczące wykorzystanych korpusów przedstawiono w tabeli 4.1. Z tego zbioru część artykułów przeznaczono do trenowania oraz część do testowania. W przypadku krzywej nauczania, wszystkie artykuły brały udział w teście. Ilość danych testowych była uzależniona od liczby danych uczących i była liczbą, która pozostała po odjęciu liczby dokumentów niebiorących udziału w fazie nauczania. Badania rozpoczęto od próby dopasowania optymalnych parametrów dla danego zbioru danych. Z obu pełnych korpusów zostały wyznaczone dwa dodatkowe, zawierające jedynie formy podstawowe rzeczowników, sklasyfikowane przy pomocy analizatora WCRTF2 podczas przetwarzania wstępnego. Taki korpus został podpisany hasłem *rzeczowniki*.

Tab. 4.1: Porównanie badanych korpusów danych

nazwa korpusu	l. dokumentów	l. klas	l. dokumentów na klasę	rozmiar na dysku
Wikipedia	∼6800	34	∼200	34MB
Artykuły	∼2600	7	∼380	12MB
Wikipedia (rzeczowniki)	∼6800	34	∼200	28MB
Artykuły (rzeczowniki)	∼2600	7	∼380	8.6MB

4.3.1. Bag-Of-Words

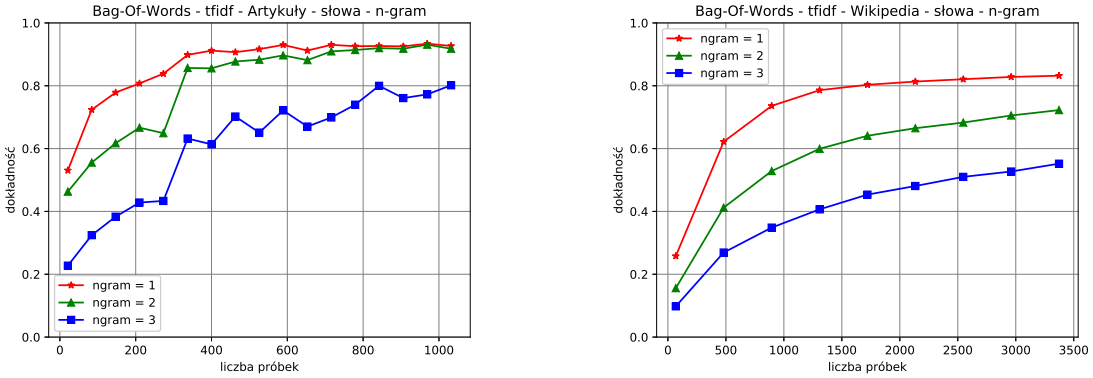
W modelu *Bag-Of-Words* w każdym teście została domyślnie użyta normalizacja *tf-idf* z listą słów nierelewantnych. Zmieniany był parametr n-gram oraz rodzaj zawartości n-gramów. Przetestowano dwa różne scenariusze:

- Scenariusz 1: analiza słów n-gramowych dla wartości parametru *n-gram*: 1, 2, 3
- Scenariusz 2: analiza znaków n-gramowych dla wartości parametru *n-gram*: 5, 6, 7

Testy zostały przeprowadzone z wykorzystaniem trzech klasyfikatorów (NaiveBayes, SVM, drzewo decyzyjne) a wyniki zostały uśrednione w celu zwiększenia czytelność wykresów oraz opisane pod wspólną nazwą *BoW* na wykresach. Scenariusze zostały przetestowane dla obydwu wykorzystywanych w pracy korpusów.

W przypadku analizy znaków, lista *stop words* nie była wykorzystywana i żadne słowa nie zostały odrzucone.

W pierwszym testowanym scenariuszu 4.2 dokładność klasyfikacji była niższa dla większych wartości *n-gram*, co jest zgodne z intuicją. Większa ilość słów wchodzących w skład n-gramu, w przypadku niewielkiej ilości danych, zwraca niższe rezultaty; jest to spowodowane większą trudnością odnalezienia i dopasowania takiego samego n-gramu. Wykres 4.2 pokazuje, że bigramy oraz unigramy uzyskały ostatecznie zbliżone wyniki. Wynika to prawdopodobnie z faktu,



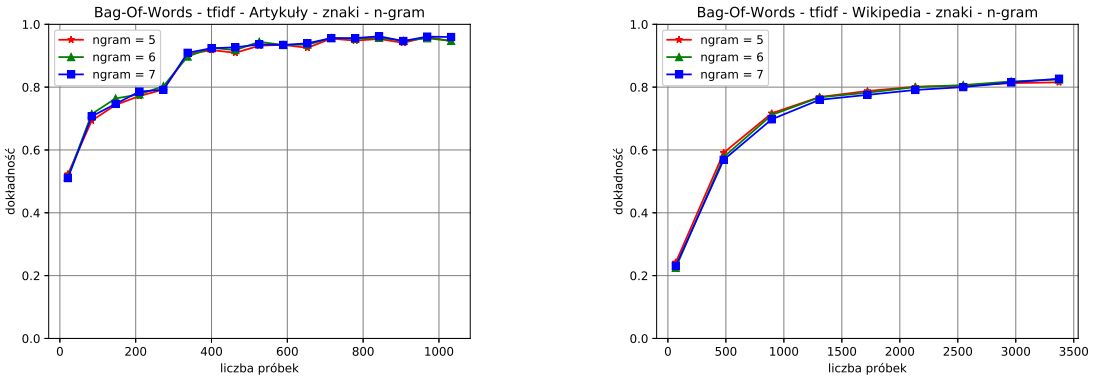
(a) Artykuły (b) Wikipedia

Rys. 4.2: Porównanie dokładności klasyfikacji *Bag-Of-Word* dla n-gramów składających się ze słów - Scenariusz 1

Tab. 4.2: Najlepsze wyniki testów *Bag-Of-Words* dla różnych wartości n-gram i rodzaju analizowanych n-gramów

scenariusz	nazwa korpusu	n-gram	dokładność
Scenariusz 1	Artykuły	1 (słowa)	0.92
Scenariusz 1	Wikipedia	1 (słowa)	0.82
Scenariusz 2	Artykuły	7 (znaki)	0.96
Scenariusz 2	Wikipedia	5 (znaki)	0.82

iż liczba klas była mniejsza, przez co pojawiało się zdecydowanie mniej unikalnych słów w korpusie.



(a) Artykuły (b) Wikipedia

Rys. 4.3: Porównanie dokładności klasyfikacji *Bag-Of-Word* dla n-gramów składających się ze znaków - Scenariusz 2

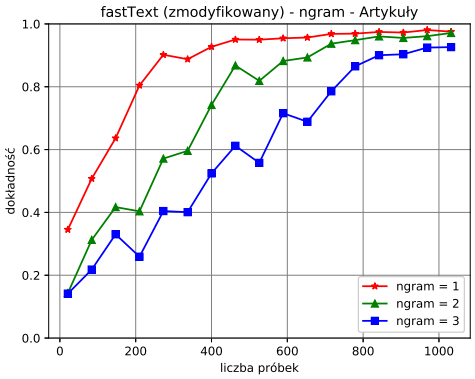
W drugim testowanym scenariuszu nie stwierdzono większych różnic pomiędzy użytymi wartościami parametrów. Wyniki końcowe wykazały, że użycie n-gramów składających się ze słów, w tym przypadku zwraca lepsze wyniki niż użycie n-gramów składających się ze znaków. Wyniki końcowe z najlepszymi uzyskanymi wartościami dokładności i odpowiadającym im parametrom przedstawiono w tabeli 4.2.

Użycie modelu *n-gramowego* dla znaków mogłoby zwracać lepsze rezultaty w przypadku dużej liczby słów, których analizatory morfologiczne nie były w stanie poprawnie rozpoznać i sprowadzić do ich form podstawowych. Przykładem mogłaby być sytuacja, kiedy w tekście znajduje się duża liczba literówek - uniemożliwiłoby to poprawne porównanie dwóch takich samych słów jednak napisanych z błędami.

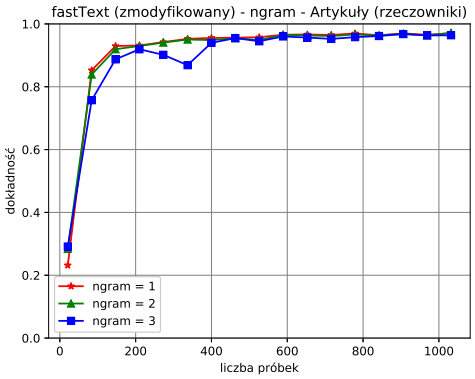
4.3.2. fastText

Biblioteka *fastText* według założeń przyjmuje zbiory dokumentów, bez żadnych modyfikacji. [21] W pracy podjęto próbę optymalizacji tej metody, mającą na celu zmniejszenie czasu pracy bez wpływu na jakość wyników. W celu ograniczenia zbioru dokumentów wejściowych dla *fastText*, zostały one poddane takiemu samemu procesowi jak w metodzie *BoW*; proces ten został przedstawiony na rysunku 3.6. W celu wyboru optymalnych parametrów dla zmodyfikowanych korpusów wejściowych, przetestowano zmianę każdego z parametrów przy pozostawieniu domyślnych parametrów dla pozostałych wartości. Za najlepszą wartość parametru przyjmowano tą, przy użyciu której ostatecznie zwracane były najwyższe wyniki dla miary dokładności. Rezultaty przy użyciu zmienionych parametrów zostały zaznaczone na wykresach jako *fastText (zmodyfikowany)*[1]. Wyniki zostały uśrednione po 10 przebiegach dla każdego testu.

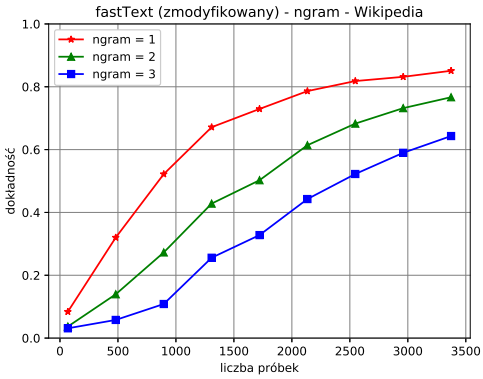
Zmiana parametru *ngram*



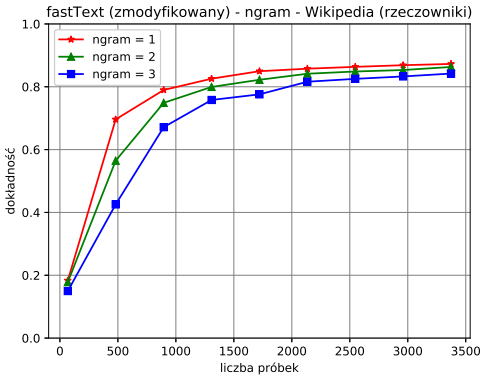
(a) Artykuły



(b) Artykuły (rzeczowniki)



(c) Wikipedia



(d) Wikipedia (rzeczowniki)

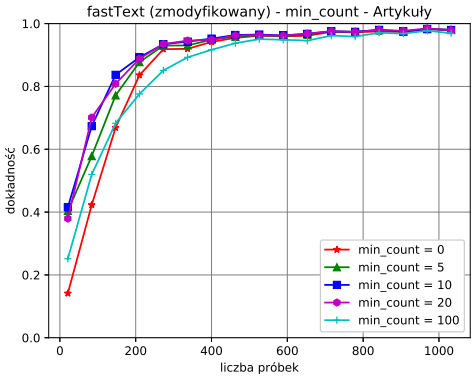
Rys. 4.4: Porównanie wyników zmiany parametru *n-gram*

[1] W dalszej części pracy *fastText (zmodyfikowany)* jest również zamiennie określany jako *fastText*.

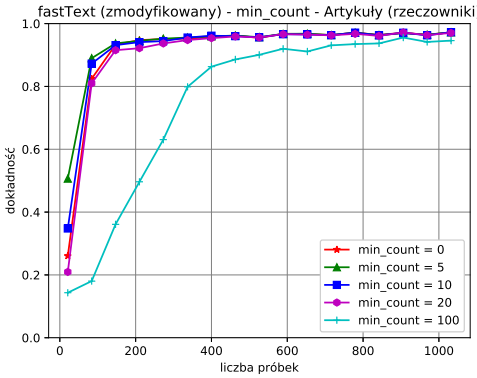
Parametr *ngram* określa maksymalną długość na jaką zostanie podzielony zbiór wyrazów. Wszystkie trzy instancje w początkowej fazie osiągnęły podobny wynik dokładności, kolejne iteracje wskazały że instancja z parametrem *ngram=1* uczyła się najszybciej i to ona ostatecznie osiągnęła też najlepsze wyniki, dla wszystkich czterech przypadków. Jest to skutek odwrotny do oczekiwanego, w żadnym momencie *fastText* z użyciem bigramów nie osiągnął lepszego wyniku niż unigramy [22]. Powodem takich rezultatów jest prawdopodobnie fakt, iż w przypadku korpusu z artykułami bardziej informacyjne były unigramy niż formy zawierające więcej n-gramów. Wyniki zostały przedstawione na rysunku 4.4.

Zmiana parametru *min_count*

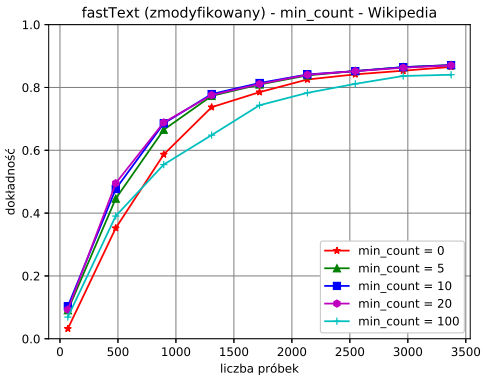
Parametr *min_count* odrzuca wyrazy, które występują częściej niż zadana wartość, wyniki zmiany parametru zaprezentowano na rysunku 4.5.



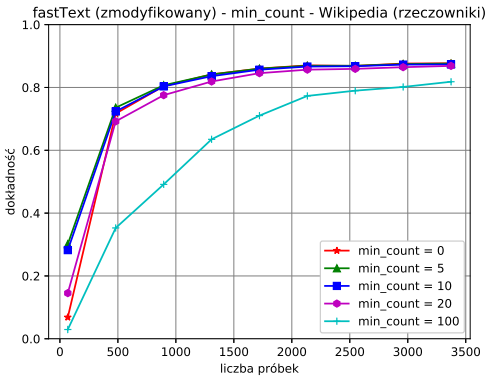
(a) Artykuły



(b) Artykuły (rzeczowniki)



(c) Wikipedia



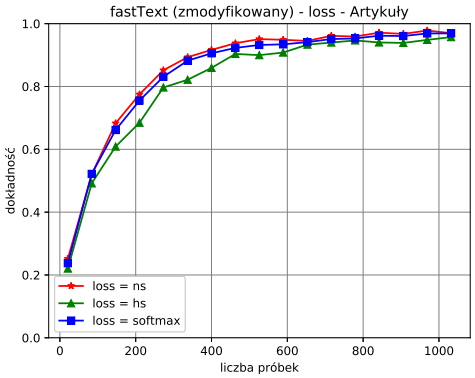
(d) Wikipedia (rzeczowniki)

Rys. 4.5: Porównanie wyników zmiany parametru *min_count*

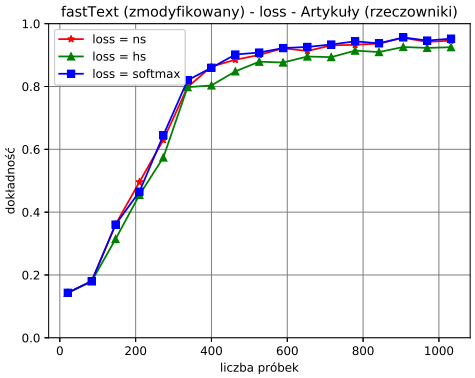
Wyniki dla 5 testowanych wartości zbiegają do tego samego poziomu. Najwięcej próbek potrzebował algorytm w przypadku parametru *min_count=100*. Początkowe wyniki na przedstawionych rysunkach 4.5 wyraźnie pokazują, że najlepszą wartością w tym przypadku jest wartość 20. Dla korpusów z rzeczownikami można zaobserwować zmniejszenie się różnic w wynikach jakościowych pomiędzy wartościami: 0, 5, 10 oraz 20.

Zmiana parametru *loss*

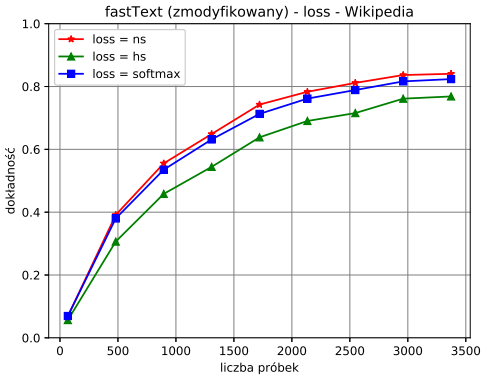
Do wyboru są trzy różne wartości, które definiują który klasyfikator zostanie użyty podczas pracy. Domyślnym klasyfikatorem dla *fastText* jest *softmax* [40].



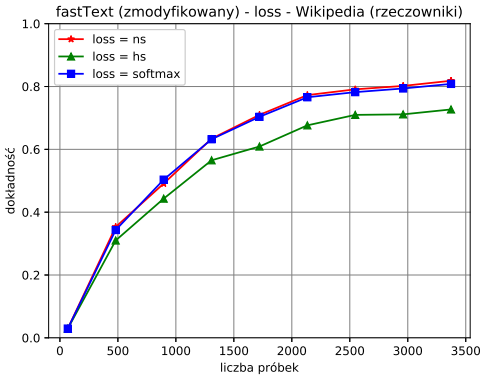
(a) Artykuły



(b) Artykuły (rzeczowniki)



(c) Wikipedia



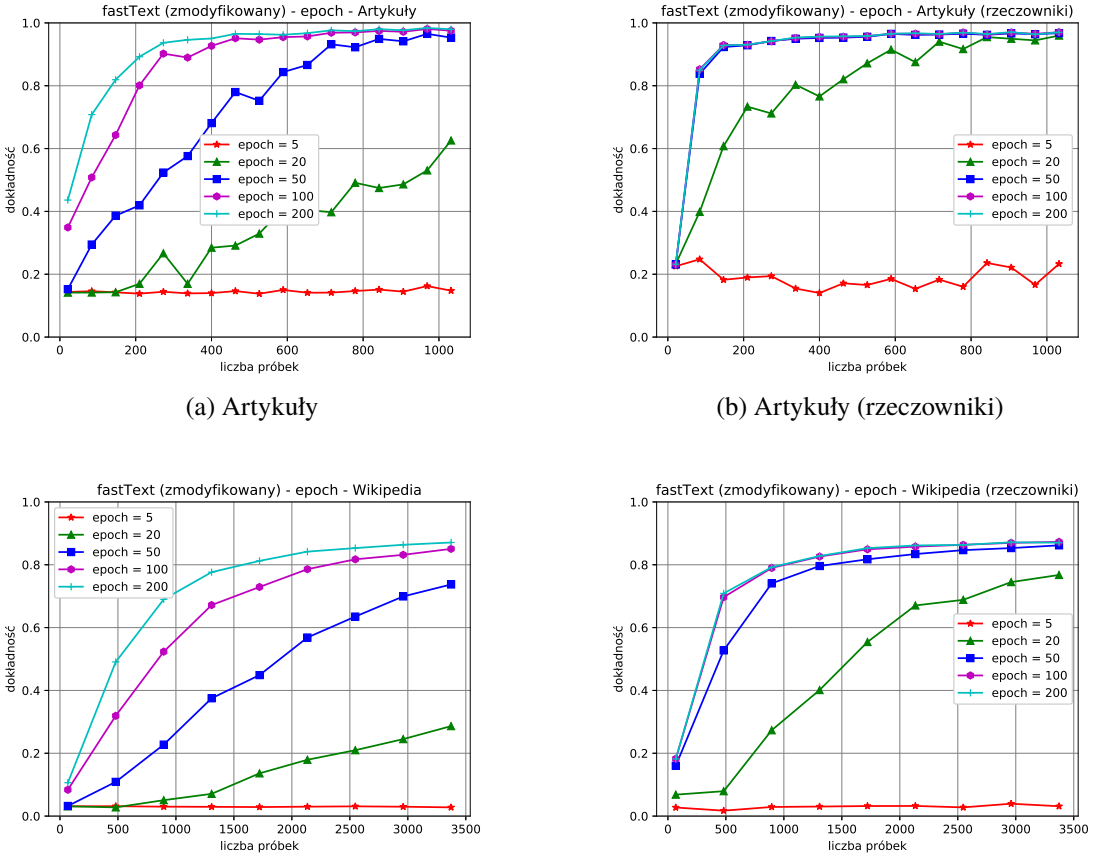
(d) Wikipedia (rzeczowniki)

Rys. 4.6: Porównanie wyników zmiany parametru *loss*

Zmiana parametru *loss* nie powodowała tak dużych zmian w wynikach jak poprzednie parametry. Jednoznacznie jednak można stwierdzić, że przy mniejszej ilości próbek algorytm *softmax* zwraca lepsze wyniki niż pozostałe dwie metody. Ostatecznie najlepsze wyniki w przypadku użytego korpusu osiągnięto przy pomocy algorytmu *ns (negative samples)* i to on zostanie użyty w końcowych testach porównawczych. Wyniki na wykresach zostały przedstawione na rysunku 4.6.

Zmiana parametru *epoch*

Parametr *epoch* oznacza liczbę iteracji, które musi wykonać algorytm, aby nauczyć się korpusu [11]. Liczba iteracji jest tożsama z liczbą możliwości zobaczenia przez *fastText* konkretnego dokumentu.



(a) Artykuły

(b) Artykuły (rzeczowniki)

(c) Wikipedia

(d) Wikipedia (rzeczowniki)

Rys. 4.7: Porównanie wyników zmiany parametru *epoch*

Domyślną wartością parametru *epoch* w *fastText* jest wartość 5 i w przypadku wykorzystanych korpusów zwracane wyniki są bardzo niskie, niezależnie od ilości próbek. Najlepsze wyniki osiągnięto przy użyciu wyższych wartości, kosztem czego jest dłuższy czas nauki algorytmu, co jest oczekiwanym efektem. Wartości 50, 100 oraz 200 uzyskują zbliżone do siebie wyniki w przypadku korpusu z rzeczownikami, z widoczną korzyścią dla wartości 200 w pełnym korpusie. Wartości pośrednie - 50 oraz 20, wraz ze zwiększającą ilością danych testowych dąży do wyników osiąganych przez najlepsze wartości parametrów.

Zmodyfikowane parametry

Ostatecznie dobrano optymalne wartości dla każdego parametru i przedstawiono je w tabeli 4.3. Kolejne testy wykonano przy ich użyciu, a metodę określano jako *fastText (zmodyfikowany)*.

Tab. 4.3: Optymalne parametry biblioteki *fastText* dla testowanych danych

ngram	min_count	loss	epoch
1	20	ns (negative samples)	200

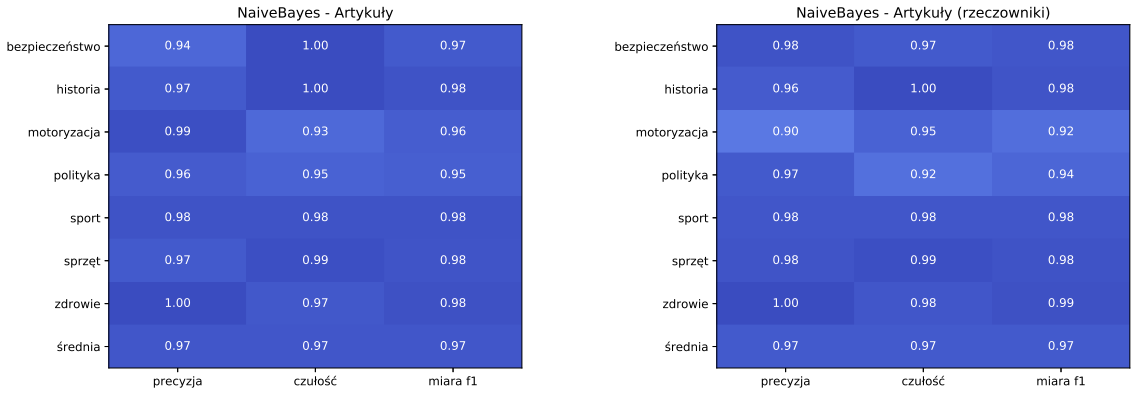
4.4. Porównanie wyników

Podczas wyznaczania raportów jakości, dane treningowe stanowiły stosunek 70 do 30 dla obu testowanych korpusów. Składały się z 1820 dokumentów uczących oraz 780 dokumentów testujących, co stanowiło stosunek 70 do 30.

4.4.1. Analiza zbadanych miar

Dla każdego klasyfikatora wygenerowana została tabela z wynikami miar dla każdej klasy w zbiorze dokumentów: *precyzji (ang. precision)*, *czułości (ang. recall)*, *miary f1 (ang. f1-score)*. W ostatnim wierszu znajdują się wartości uśrednione oznaczone jako *średnia*.

Naive Bayes

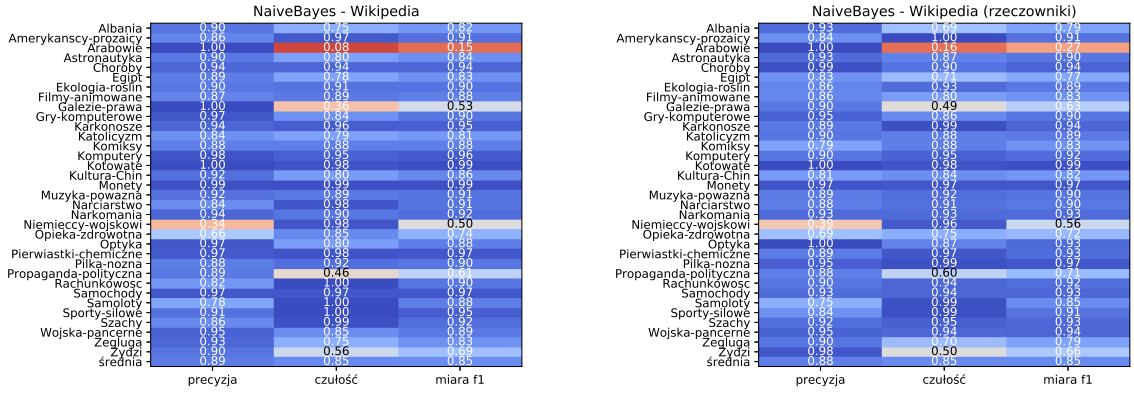


(a) Artykuły

(b) Artykuły (rzeczowniki)

Rys. 4.8: Miary jakościowe - NaiveBayes - Artykuły

Rysunek 4.8 przedstawia porównanie zbadanych miar dla pełnego korpusu oraz korpusu z samymi rzeczownikami. Według analizy skuteczność klasyfikatora *NaiveBayes* w obu przypadkach jest zbliżona do siebie. Pojawiają się minimalne różnice na korzyść pełnych korpusów, co można zaobserwować w przypadku klasy *zdrowie* oraz *sprzęt*. Średnie wartości były takie same.



(a) Wikipedia

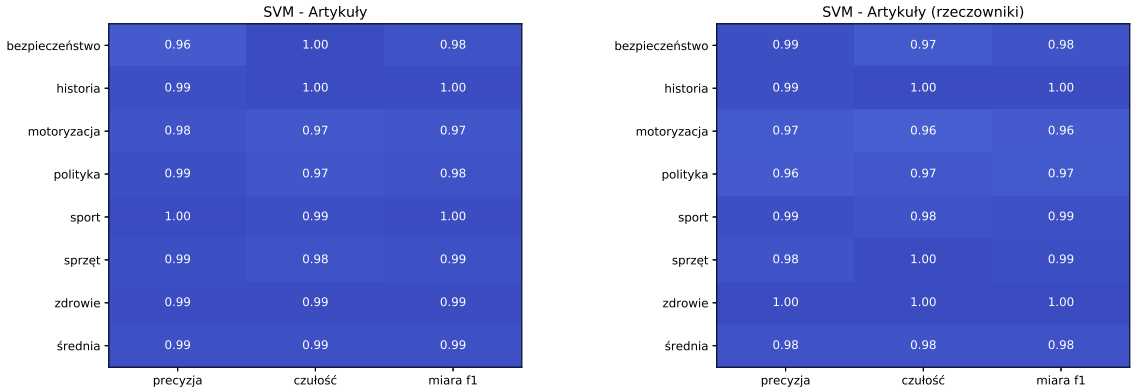
(b) Wikipedia (rzeczowniki)

Rys. 4.9: Miary jakościowe - NaiveBayes - Wikipedia

Rysunek 4.9 przedstawia analogiczne porównanie zbiorów dla 34 klasowego korpusu Wikipedii. Średnie wyniki wskazują na to, że różnice pogłębiły się lecz wciąż były one bardzo małe. Wyniki miary f1 zostały bez zmian. Warto również zauważyć, że dla niektórych klas uzyskano lepsze wyniki; dobrym przykładem jest klasa *propaganda-polityczna*, której wynik miary f1 poprawił się o 10 punktów procentowych w stosunku do pełnego korpusu. Mimo braku zmiany średniej wartości miary f1 widać poprawę w większości klas, które były słabo rozpoznawane. Wyjątkiem od reguły jest klasa *kultura-chin*, której wynik czułości spadł o 4 punkty procentowe.

SVM

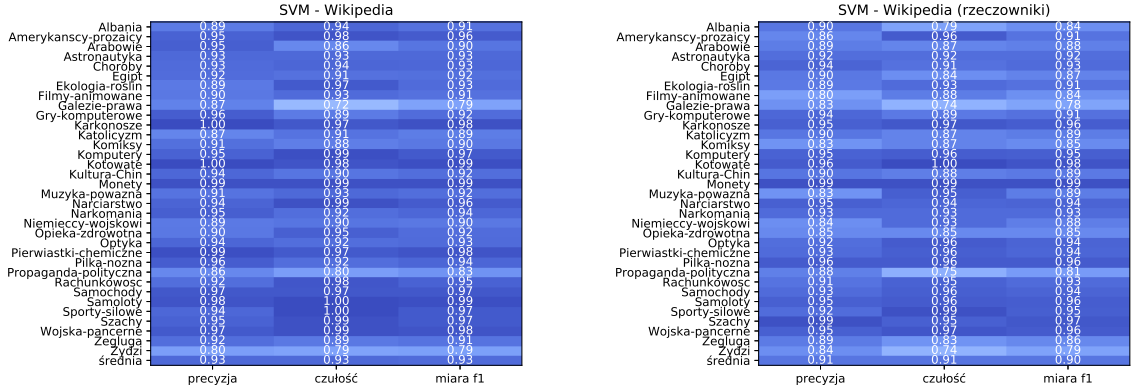
SVM uzyskał bardzo dobre wyniki we wszystkich przypadkach. Średnie wartości były najwyższe dla każdego typu testowanych danych dla badanych klasyfikatorów.



(a) Artykuły (b) Artykuły (rzeczowniki)

Rys. 4.10: Miary jakościowe - SVM - Artykuły

Wykresy na rysunku 4.10 jednoznacznie wskazują na bardzo dobre dopasowania każdej z klas. Najniższa zanotowana wartość miary była nie niższa niż 0.96; klasyfikator *SVM* niemal bezbłędnie rozpoznawał wszystkie klasy. Dla tych dwóch zbiorów ocena, czy użycie samych rzeczowników pozytywnie wpływa na jakość klasyfikacji, byłaby nadinterpretacją lecz z całą pewnością można stwierdzić, że nie ma negatywnego wpływu na przypisywanie kategorii zadanym tekstom.



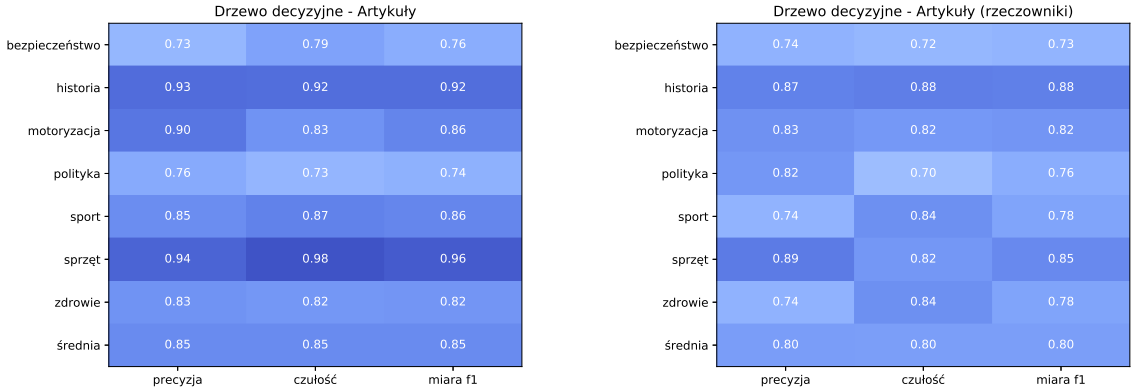
(a) Wikipedia (b) Wikipedia (rzeczowniki)

Rys. 4.11: Miary jakościowe - SVM - Wikipedia

Bardziej precyzyjne wnioski można otrzymać dopiero po analizie wyników na rysunku 4.11. Większa liczba klas wystarczająco utrudniła dopasowywanie kategorii, można zaobserwować, że średnie wyniki wszystkich rozpatrywanych miar pogorszyły się o 1/2 punkty procentowe. Słabo rozpoznawane kategorie z przedziału od 0.7 do 0.8 nie były lepiej dopasowywane w zbiorze dokumentów z samymi rzeczownikami.

Drzewo decyzyjne

Drzewo decyzyjne osiągało najgorsze wyniki, ze wszystkich testowanych klasyfikatorów dla badanych zbiorów danych.

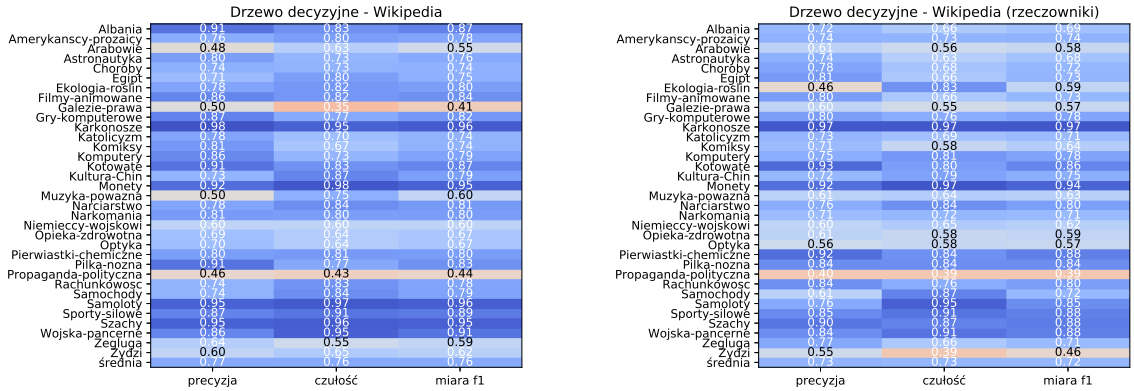


(a) Artykuły

(b) Artykuły (rzeczowniki)

Rys. 4.12: Miary jakościowe - Drzewo decyzyjne - Artykuły

Wszystkie kategorie uzyskały gorsze wyniki dla miary f1 po odrzuceniu innych części mowy niż rzeczowniki, o czym również świadczą niższe wartości średnie. Wysokie wskaźniki dla klasy *sprzęt* są wynikiem bardzo charakterystycznych zbiorów słów występujących w tej kategorii, bowiem zawiera ona bardzo dużo nazw własnych, co zostało zobrazowane na rysunku 3.8.



(a) Wikipedia

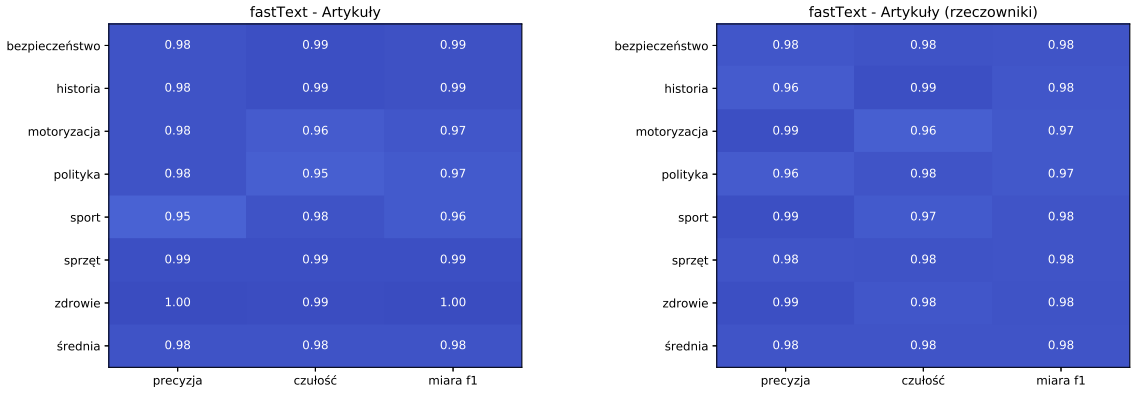
(b) Wikipedia (rzeczowniki)

Rys. 4.13: Miary jakościowe - Drzewo decyzyjne - Wikipedia

Ważnym wnioskiem po analizie rysunku 4.13 jest fakt iż, klasyfikacja dla korpusu z rzeczownikami w większości pogłębiła różnice między korpusami. Słabe wyniki, poniżej wartości 0.7 miary f1, obniżyły się, a wysokie wyniki, powyżej 0.8, polepszyły. Przykładami takiego zachowania są klasy *karkonosze* (polepszenie), *żydzi* (pogorszenie), *propaganda-polityczna* (pogorszenie), *gałęzie-prawa* (pogorszenie)

fastText

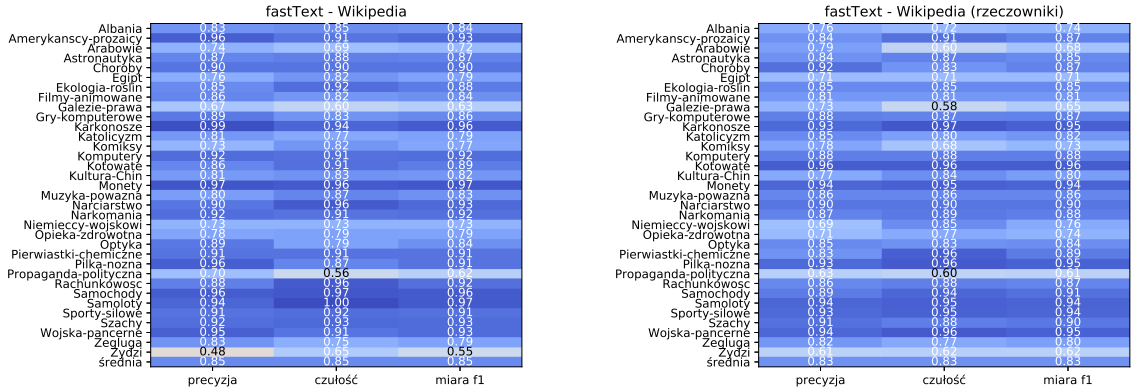
fastText osiągał wyniki zbliżone do tych, które zwracały klasyfikatory *NaiveBayes* oraz *SVM*, co plasuje go na drugim miejscu pod względem jakości dopasowywanych klas na podstawie otrzymanych średnich wyników dla testowanych miar na badanych korpusach.



(a) Artykuły (b) Artykuły (rzeczowniki)

Rys. 4.14: Miary jakościowe - fastText - Artykuły

fastText bardzo dobrze klasyfikował artykuły w obu przypadkach, co pokazuje rysunek 4.14. Nie utracił on znacznie skuteczności w jakości działania, mimo ograniczania zbioru dokumentów do samych rzeczowników.



(a) Wikipedia (b) Wikipedia (rzeczowniki)

Rys. 4.15: Miary jakościowe - fastText - Wikipedia

Pełniejszy obraz dało przebadanie klasyfikatora na zbiorze danych z większą liczbą klas. Podobnie jak w pozostałych przypadkach, końcowe wartości średnie nieznacznie obniżyły się, co można uznać za regułę niezależnie od ilości danych i rodzaju użytego klasyfikatora badanego w pracy.

Wnioski

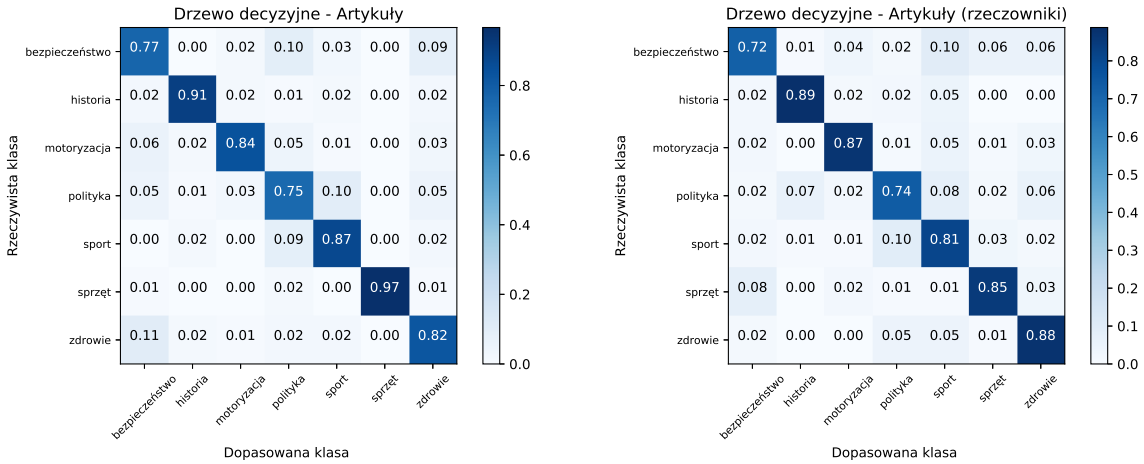
Na podstawie przebadanych klasyfikatorów i analizie badanych miar, podstawowym wnioskiem jaki należy wysnuć jest fakt, iż w każdym przypadku jakość klasyfikacji nieznacznie obniża się o około kilka punktów procentowych dla wszystkich miar lub pozostaje bez zmian. Jest to oczekiwany wynik [40] obniżenia się jakości wyników. Bardzo niskie wartości miar dla

drzewa decyzyjnego w stosunku do innych klasyfikatorów pozwoliły na stwierdzenie pewnej zależności między pełnym korpusem, a korpusem z rzeczownikami. W wielu przypadkach, gdzie dopasowanie klasy w pełnym korpusie jest niskie, widać pogorszenie się tego wyniku dla samych rzeczowników. Natomiast jeśli wartość miary f1 jest wysoka dla pełnego korpusu, wynik będzie wyższy dla samych rzeczowników. Reguła ta najbardziej widoczna jest w klasyfikatorach *NaiveBayes* oraz *drzewo decyzyjne*. Trudno jednak przyjąć taki wniosek za regułę, gdyż istnieją również odstępstwa; na przykład klasa *propaganda-polityczna* w *drzewie decyzyjnym*, w której po wykonaniu testów na zbiorze z samymi rzeczownikami wynik wzrósł w przeciwieństwie do bardzo niskiego wyniku miary f1 0.45 w pełnym zbiorze.

4.4.2. Analiza macierzy błędów

Macierze błędów pozwalają na graficzne przedstawienie klas, w których najczęściej występowały błędy. Dla każdego klasyfikatora została wygenerowano macierz pomyłek. Podczas analizy pominięto analizę macierzy, które nie zawierały znacznych błędów. Macierz na osi x zawiera klasy, które powinny zostać dopasowane oznaczone (*dopasowana klasa*), natomiast na osi y znajdują się klasy, które przyporządkował klasyfikator (*rzeczywista klasa*). Macierz, która zawiera mało pomyłek, charakteryzuje się wartościami wysokimi i ciemnymi polami po przekątnej, oznacza to, że klasyfikator poprawnie dopasował kategorię do zadanego tekstu.

Drzewo decyzyjne

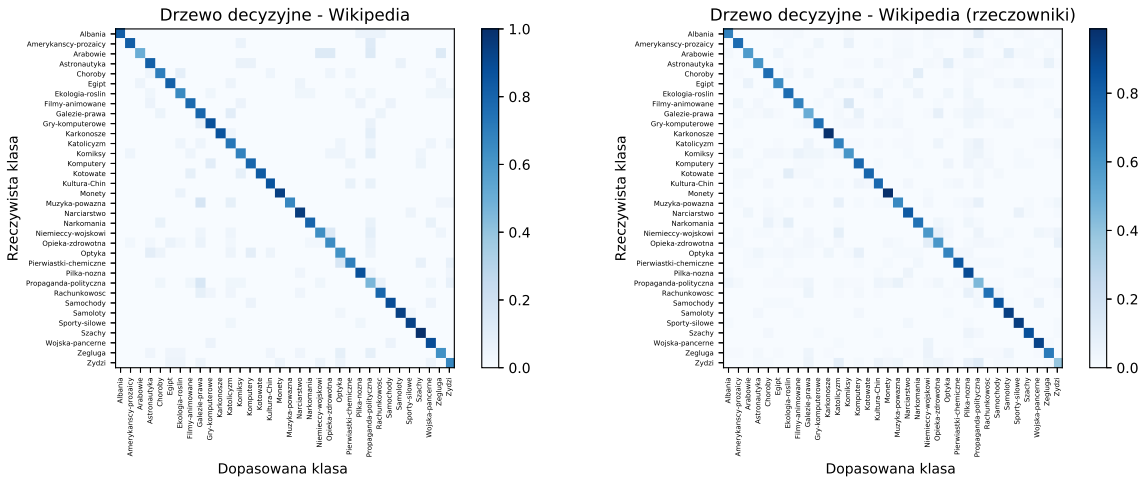


(a) Artykuły (b) Artykuły (rzeczowniki)

Rys. 4.16: Tablica pomyłek - Drzewo decyzyjne - Artykuły

Drzewo decyzyjne osiągało najgorsze wyniki, więc analiza jego tablicy pomyłek jest bardzo istotna. Można zauważyć pewną anomalię dla korpusu z rzeczownikami: klasyfikator próbował dopasować kategorię *polityka* do klas *bezpieczeństwo*, *zdrowie*, *sport*, *historia*. Jedną z częściej mylonych klas była kategoria *polityka* i klasyfikator często mylił ją z kategorią *bezpieczeństwo* (10% przypadków) oraz *zdrowie* (9% przypadków) w pełnym korpusie. Wysoką wartość dla klasy *bezpieczeństwo* można tłumaczyć faktem, że obie kategorie poruszają tematy istotne dla bezpieczeństwa i społeczeństwa. 6-procentową wartość błędu dla klasy *sprzęt* w przypadku korpusu z rzeczownikami można tłumaczyć tym, że artykuły z klasy *bezpieczeństwo* pochodziły głównie ze stron o bezpieczeństwie IT, występowanie podobnego słownictwa w obu tych kategoriach może być źródłem pomyłek. Na uwagę również zasługuje fakt, iż w korpusie ze

wszystkimi klasami gramatycznymi taka anomalia nie występuje, więc dodatkowe klasy gramatyczne skutecznie stłumiły niepasujące cechy. W przypadku bardziej zróżnicowanego korpusu



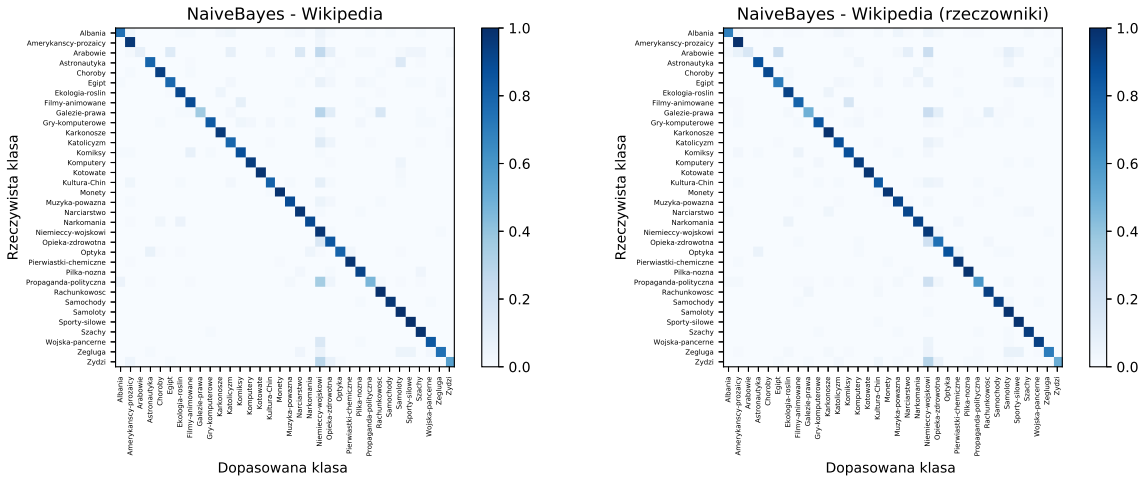
(a) Wikipedia (b) Wikipedia (rzeczowniki)

Rys. 4.17: Tablica pomyłek - Drzewo decyzyjne - Wikipedia

można dostrzec podobną sytuację dla klasy *propaganda-polityczna*. Klasyfikator częściej niż inne próbował dopasować tę kategorię do wszystkich pozostałych kategorii.

Naive Bayes

NaiveBayes mimo osiągnięcia najlepszych wyników, nie wszystkie klasy były dopasowywane z wysokim prawdopodobieństwem.

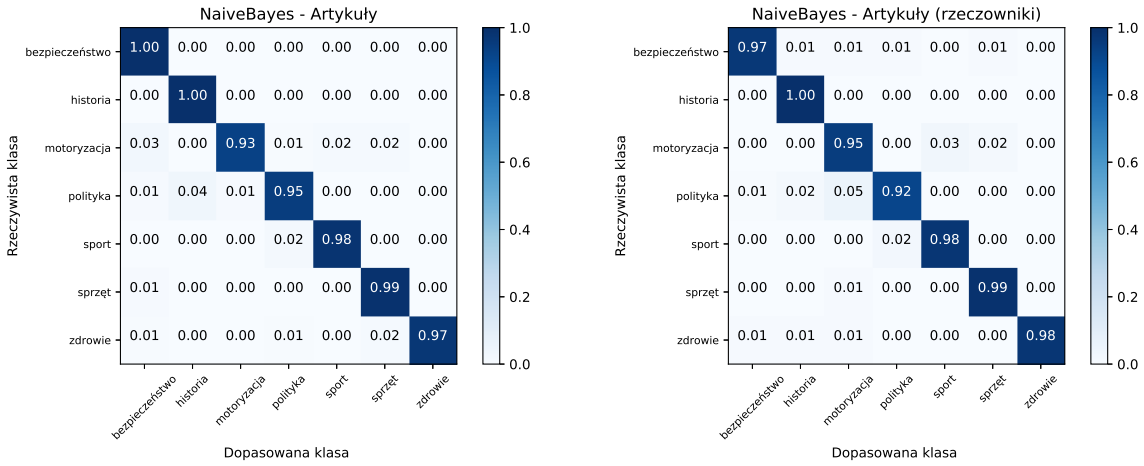


(a) Wikipedia (b) Wikipedia (rzeczowniki)

Rys. 4.18: Tablica pomyłek - Naive Bayes - Wikipedia

Dla korpusu z rzeczownikami liczba błędnych dopasowań zmniejszyła się lecz mimo to, pewna charakterystyka w obu przypadkach została zachowana. Klasa *niemieccy-wojskowi* często była dopasowywana zamiast kategorii *żydzi*, *propaganda-polityczna*, *gałęzie-prawa*, *arabowie*. Sama klasa *arabowie* jest bardzo słabo rozpoznawana i wykres na rysunku 4.18 pokazuje,

że często była mylona z kategorią *egipt*, *narciarstwo*, *niemieccy-wojskowi*, *samoloty*. Klasyfikacja dla *gałęzi-prawa* poprawiła się po odrzuceniu wyrazów innych niż rzeczowniki, głównie za sprawą lepszego rozpoznawania kategorii *niemieccy-wojskowi*.



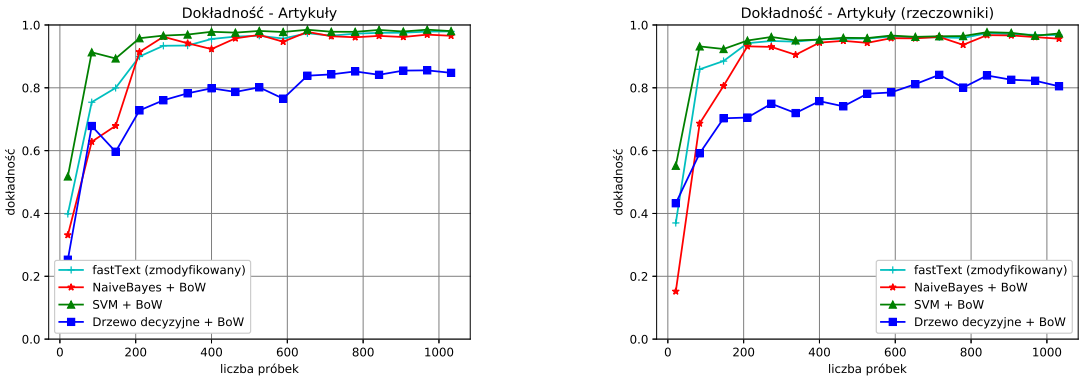
(a) Artykuły (b) Artykuły (rzeczowniki)

Rys. 4.19: Tablica pomyłek - Drzewo decyzyjne - Artykuły

W przypadku zbioru z mniejszą liczbą kategorii zaprezentowanych na rysunku 4.19 nie zaobserwowano podobnych tendencji. Macierz błędu prezentowała się wzorcowo.

4.4.3. Dokładność klasyfikacji

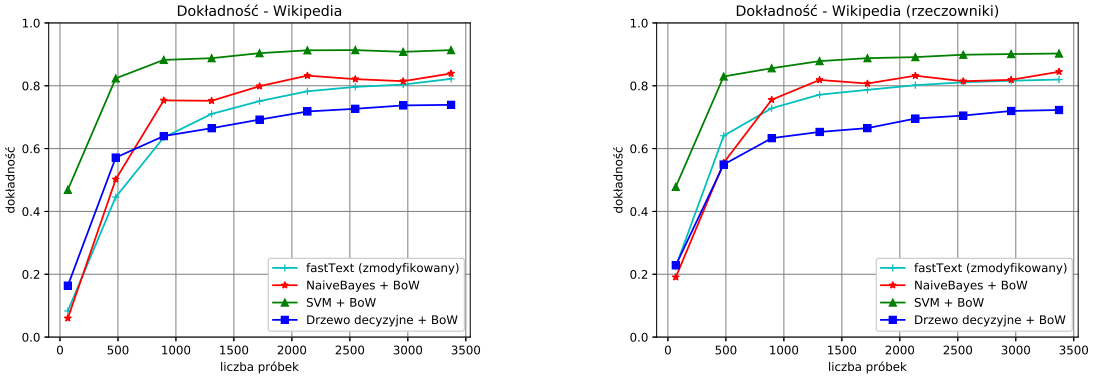
Z analizy wynika, że w każdym z czterech przetestowanych przypadków najszybciej uczącym się klasyfikatorem był *SVM*, natomiast najwolniej uczyło się drzewo decyzyjne; wyniki dla zmodyfikowanej wersji *fastText* są zbliżone do wyników modelu *Bag-Of-Words*. [18] [40] Dla zbioru



(a) Artykuły (b) Artykuły (rzeczowniki)

Rys. 4.20: Krzywe nauczania Bag-Of-Words oraz fastText - Artykuły

z 7 klasami z rysunku 4.20 można wywnioskować, iż klasyfikator *SVM* potrzebował zaledwie 14 artykułów z każdej kategorii, aby móc im przyporządkować grupę tematyczną z prawdopodobieństwem na poziomie 93% dla pełnego korpusu (a) oraz 84% dla ograniczonego korpusu (b). Odrzucenie słów z innych klas gramatycznych niż rzeczowniki poprawiło szybkość nauczania się klasyfikatora *fastText*, wynik był zbliżony do tego, który był osiągany przez *SVM* w obu przypadkach.



(a) Wikipedia

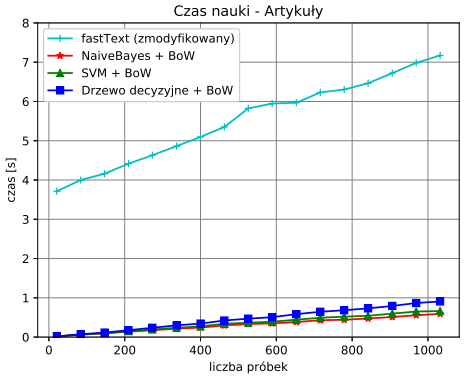
(b) Wikipedia (rzeczowniki)

Rys. 4.21: Krzywe nauczania Bag-Of-Words oraz fastText - Wikipedia

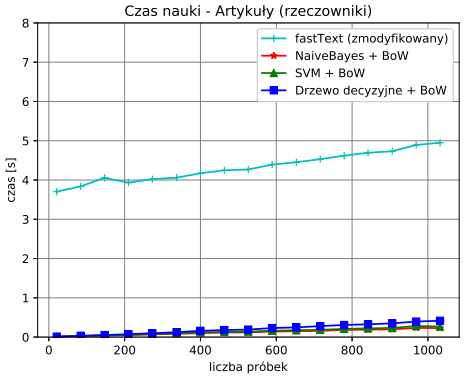
Klasyfikatory w przypadku większego zbioru, potrzebowały więcej próbek testowych, aby klasyfikować teksty z wysoką skutecznością, co zostało zobrazowane na wykresach na rysunku 4.21. Wnioski wyciągnięte na podstawie mniejszego zbioru z rysunku 4.20 są prawdziwe również w tym przypadku. Ponownie *SVM* uczył się najszybciej, wyniki są odpowiednio niższe, ale tendencja wzrostu jakości została zachowana. Po odrzuceniu innych słów niż rzeczowniki swoje wyniki ponownie polepszył *fastText*, który w pierwszym przypadku dla pełnego zbioru osiągał rezultaty niższe niż *NaiveBayes*, natomiast w drugim przypadku, dla małej liczby próbek zwracał lepsze wyniki. Modyfikacja metody *fastText* postaci odrzucenia innych części mowy niż rzeczowniki nie wpłynęła znacząco na jakość zwracanych wyników.

4.4.4. Czas

Miary jakościowe to nie jedyne istotne wskaźniki; jako że w pracy wykorzystano dwa różne modele, wzięto pod uwagę również czas. Testy czasowe zostały wykonane za każdym razem na tym samym komputerze, wszystkie działające procesy w tle zostały wyłączone na czas testów. Wykorzystany komputer miał specyfikację: 2.2 GHz Intel Core i7, 16 GB 1600 MHz DDR3. Podczas testu stale monitorowano użycie pamięci RAM, w żadnym momencie nie odnotowano jej przekroczenia ani też skokowego spadku, co może wskazywać na to, że aplikacja testująca nie musiała przerywać pracy, aby zapewnić więcej miejsca w pamięci. Pomiędzy testami komputer był uruchamiany ponownie, aby mieć pewność, że warunki testowe za każdym razem będą identyczne. Czas został uśredniony po 10 przebiegach, aby zapobiec przepełnieniu się pamięci podczas badań, co mogłoby skutkować zakłamanymi wynikami.

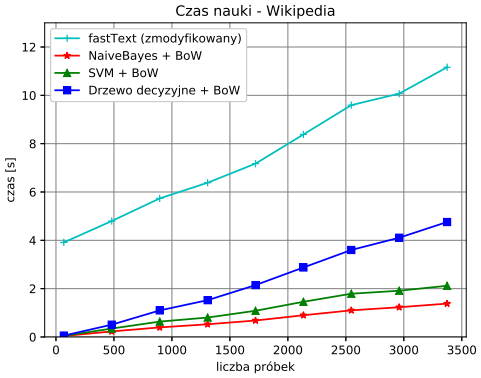


(a) Artykuły

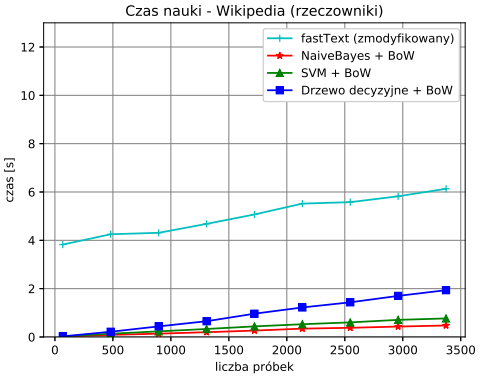


(b) Artykuły (rzeczowniki)

Rys. 4.22: Porównanie czasu nauki - Artykuły



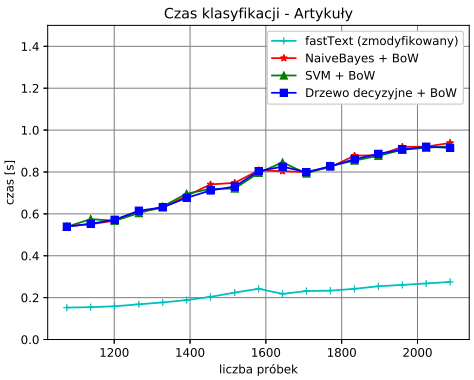
(a) Wikipedia



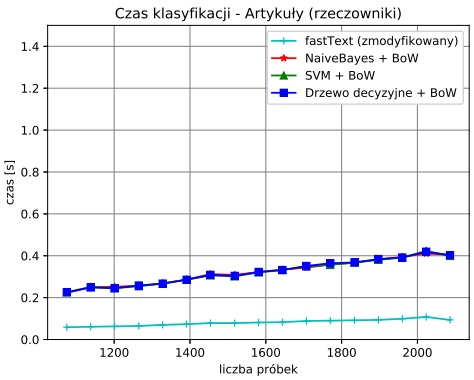
(b) Wikipedia (rzeczowniki)

Rys. 4.23: Porównanie czasu nauki - Wikipedia

Z rysunków 4.22 oraz 4.23 wynika, że czas nauki modelu był ponad 4 razy większy w przypadku narzędzia *fastText* niż *Bag-Of-Words*. Dodatkowo przewaga modelu *BoW* rosła wraz z zwiększającą się liczbą próbek treningowych.

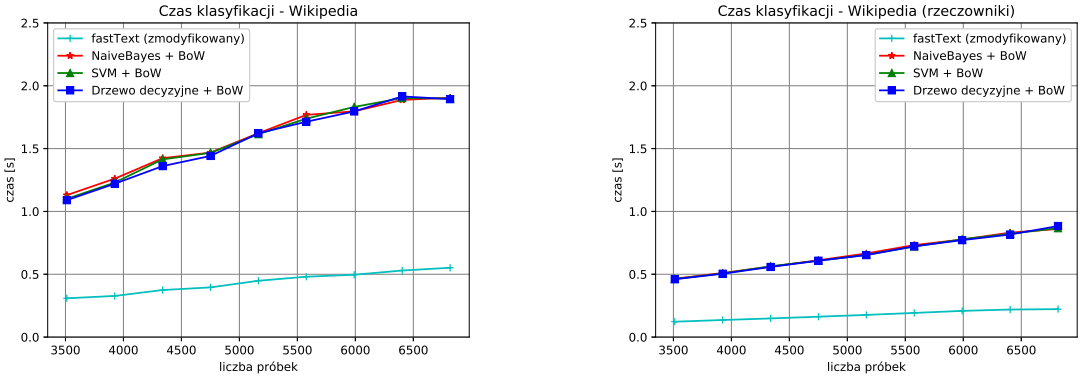


(a) Artykuły



(b) Artykuły (rzeczowniki)

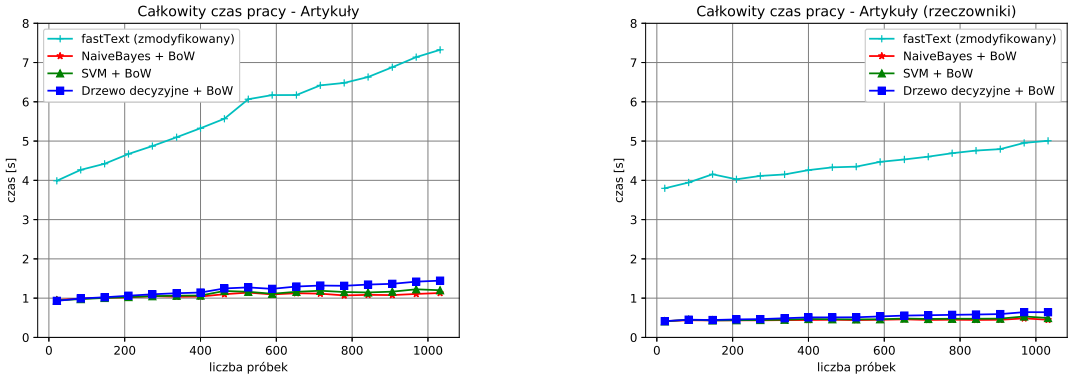
Rys. 4.24: Porównanie czasu klasyfikacji - Artykuły



(a) Wikipedia (b) Wikipedia (rzeczowniki)

Rys. 4.25: Porównanie czasu klasyfikacji - Wikipedia

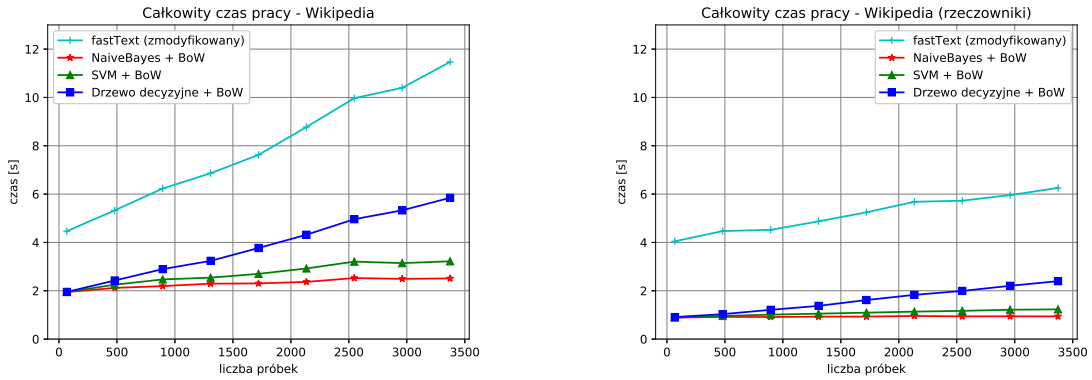
Krótki czas potrzebny na sklasyfikowanie dużej ilości dokumentów, zobrazowany na rysunkach 4.24 oraz 4.25, jest niewątpliwą zaletą biblioteki *fastText*. W dużych systemach klasyfikacja jest procesem który ma kluczowe znaczenie z biznesowego punktu widzenia, ponieważ dane muszą być przetwarzane w czasie rzeczywistym [10].



(a) Artykuły (b) Artykuły (rzeczowniki)

Rys. 4.26: Porównanie czasu całkowitej pracy - Artykuły

Dla porównania, całkowity czas pracy obu modeli również wypada na niekorzyść biblioteki *fastText* mimo jej krótkiego czasu klasyfikacji i próby modyfikacji. Wyniki zaprezentowane na rysunkach 4.26 oraz 4.27 są częściowo sprzeczne z oczekiwanymi wynikami [22]. Jedynie czas klasyfikacji *fastText* spełnił oczekiwania i był kilka rzędów mniejszy niż w przypadku drugiego modelu. Długi czas nauki klasyfikatora można wyjaśnić nieoptymalną implementacją wrappera z biblioteki *ShallowLearn*. Dodatkowa implementacja konwertująca dane wejściowe i wyjściowe nie jest przyczyną wyższego czasu nauki, ponieważ liczony jest czas wywoływania jedynie metody rozpoczynającej naukę *fastText*. Drugą możliwą przyczyną, bardziej prawdopodobną, jest zbyt wysoka wartość parametru *epoch*, jednak zmniejszenie tej wartości skutkowałoby zmniejszeniem skuteczności. W przypadku modyfikacji *fastText* zaobserwowano oczekiwany spadek potrzebnego czasu na klasyfikację oraz naukę.



(a) Wikipedia

(b) Wikipedia (rzeczowniki)

Rys. 4.27: Porównanie czasu całkowitej pracy - Wikipedia

4.5. Wnioski

Wyniki przeprowadzonych badań i wyciągnięte wnioski dla zastosowanych metod na dwóch skrajnie różnych korpusach danych, nie stwierdzają wyższości jednej metody czy klasyfikatora ponad resztę. Nie ma oczywistego wyboru jeśli chodzi o najlepszą metodę klasyfikacji tekstu, ponieważ dla każdego systemu istnieją inne istotne współczynnikami, które powinny być brane pod uwagę. W przypadku wyznaczania kategorii dla bardzo dużych zbiorów danych w których czas ma kluczowe znaczenie, np. Facebook lub inne systemy informatyczne, oczywistym wyborem będzie *fastText* ze względu na bardzo krótki czas wykonania i odpowiedzi na dane wejściowe w przypadku klasyfikacji. Czas nauki w tym przypadku odgrywa znacznie mniejszą rolę, ponieważ raz nauczony model będzie w stanie klasyfikować dokumenty na wystarczająco dobrym poziomie. Nauka klasyfikatora w przypadku dużych rozproszonych systemów odbywać się będzie asynchronicznie, co dodatkowo umniejsza istotność czasu nauczania. W przypadku, gdy system nastawiony jest na zwracanie jak najbardziej poprawnych wyników, gdzie różnica czasowa rzędu kilku godzin nie ma aż tak dużego wpływu, zdecydowanie lepszym wyborem będzie zastosowanie klasycznego modelu *Bag-Of-Words*. Warto również pamiętać, że olbrzymi wpływ na wybór metody lub klasyfikatora powinna mieć perspektywa ilości danych, które zostaną użyte w systemie. To właśnie liczebność zbiorów jest główną sfer, która powinna być brana pod uwagę przy wyborze. W przypadku modelu *Bag-Of-Words* klasyfikator *SVM* radził sobie najlepiej, należy jednak pamiętać, że nie jest to reguła która będzie się powtarzać dla każdego zbioru danych. Na uwagę zasługuje również fakt, że ograniczenie zbioru do samych rzeczowników wyraźnie zmniejszyło czas potrzebny na predykcję oraz naukę zastosowanych klasyfikatorów. Korpusy zawierające same rzeczowniki zmniejszyły swoją objętość wagową od 20 do 30% co stanowi prawie 1/3 całości danych; mimo tak dużego ubytku danych jakość średniej klasyfikacji pogorszyła się minimalnie, a na podstawie przebadanych miar różnica wynosiła nie więcej niż 2% dla miary f1. Co więcej, w niektórych przypadkach usunięcie słów innych niż rzeczowniki pozytywnie wpłynęło na jakość znajdowanych klas; dotyczy to głównie słabo rozpoznawanych kategorii, choć nie jest to regułą, ale można zauważyć pewną powtarzalność.

Rozdział 5

Podsumowanie

W niniejszej pracy cel został osiągnięty - przebadane zostały dwa modele klasyfikacji: *Bag-Of-Words* i *fastText* oraz różne klasyfikatory, tak jak zakładał temat pracy. Pomyślnie udało się zaimplementować system automatycznego przypisywania tekstów w języku polskim do grup tematycznych z wykorzystaniem języka Python oraz interfejsów dostępnych w bibliotece *SciKitLearn* i *fastText*. Największym wyzwaniem w pracy okazało się poprawne dobranie tekstów artykułów tak, aby maksymalnie zwiększyć gamę klasyfikowanych kategorii. Zebranie korpusu wysokiej jakości składającego się z artykułów, było zadaniem nietrywialnym lecz możliwym do zrealizowania. Największą przeszkodą podczas zbierania danych były niepoprawnie dobierane tagi przez autorów tekstów, co skutkowało tym, że wiele kategorii było unikalnych względem całego korpusu. W efekcie, bardzo duża liczba artykułów musiała zostać odrzucona ze względu na brak szans na pomyślną klasyfikację, spowodowany niewystarczającą ilością danych do nauki. Praca jednoznacznie określa wady i zalety obu zastosowanych modeli i jednocześnie wskazuje, jakie usprawnienia powinny zostać rozważone w celu minimalizacji negatywnych efektów wybranych metod. System może z powodzeniem zostać zaimplementowany w większych systemach informatycznych w celu klasyfikacji tekstów na wybranych kategoriach.

Podczas testów wykazano, że najskuteczniejszą metodą klasyfikacji była metoda *Bag-Of-Words* wraz z klasyfikatorem *SVM*. Był to jednocześnie najszybciej uczący się klasyfikator. Kosztem tej dokładności w stosunku do metody *fastText* okazał się czas potrzebny na klasyfikację.

W pracy zaproponowano również optymalizację w postaci klasyfikacji samych rzeczowników. Takie działanie obniżyło zasoby wymagane podczas przetwarzania oraz znacznie przyspieszyło naukę oraz klasyfikację dokumentów. Co więcej, mimo oczywistej zalety (ograniczenie przetwarzanych danych o 20-30%), to skuteczność klasyfikacji pozostała praktycznie bez zmian. Takie działanie z pewnością powinno być brane pod uwagę podczas tworzenia systemów przetwarzających i klasyfikujących duże zbiory dokumentów, gdyż jest to relatywnie prosty sposób na optymalizację procesu przetwarzania.

Podczas badań wykazano, że *fastText* jest często równie dokładny co model *Bag-Of-Words*, a jednocześnie wymaga kilku rzędów wielkości krótszego czasu, aby klasyfikować tekst na zwykłych procesorach. Dodatkowo jego parametry pozwalają na zmniejszenie czasu potrzebnego na naukę kosztem niższej skuteczności. Może być to szczególnie pożądany efekt w systemach, które działają w czasie rzeczywistym, gdzie odpowiedzi na wprowadzone dane powinny być natychmiastowe na nowo wprowadzane dane. Praca z całą pewnością może posłużyć jako punkt odniesienia dla społeczności badawczej, w której rośnie zainteresowanie porównywaniem skuteczności różnych metod klasyfikacji tekstu.

5.1. Możliwe kierunku rozwoju

Rozszerzenie o inne języki naturalne

Według założeń, klasyfikacja tekstu odbywała się przy wykorzystaniu tekstów w języku polskim. Zaimplementowany system z powodzeniem mógłby zostać użyty wraz z innymi językami naturalnymi. Konieczne wtedy byłyby minimalne zmiany w zakresie wykorzystanych słowników słów nierelewantnych oraz implementacja odpowiedniego analizatora morfologicznego. Rozszerzenie systemu o kolejne języki wymagałoby również dodatkowych parametrów, które informowałyby system o tym, który analizator powinien zostać użyty.

Metody klasyfikacji

Chociaż w pracy przetestowano dwie metody klasyfikacji tekstu, badania mogą być rozszerzone o dodatkową metodę klasyfikacji opartą o LDA (ang. Latent Dirichlet allocation), należącą do metod uczących się bez nauczyciela. W takim przypadku podejście do klasyfikacji musiałoby zostać zmodyfikowane ze względu na sposób działania LDA; należałoby zwrócić też uwagę na to, ile tematów powinno być wygenerowanych przez algorytm, a następnie dopasować jego wyniki do zaimplementowanych już metod.

Usprawnienie rozpoznawania kategorii

Bardzo duża część pobranego korpusu musiała zostać odrzucona ze względu na unikalne kategorie przypisywane do artykułów przez autorów tekstów. Usprawnienie polegałoby na próbie wyszukania bardziej ogólnych tagów/klas/kategorii dla danego artykułu. W pracy brano pod uwagę jedynie kategorie ustalane przez autorów.

Rozbudowa korpusu

Dodatkową ścieżką rozwoju pracy może być również próba rozbudowania istniejącego korpusu z artykułami o dodatkowe klasy. Może to zostać zrealizowane dzięki implementacji klasy *ParserTemplateStep.class* w aplikacji Javowej. Dodanie kolejnych kategorii wzbogaciłoby otrzymywane wyniki.

Literatura

[1] In html, what is a non-breaking space? https://kb.iu.edu/d/agjn. [dostęp dnia 02 maja 2018].

[2] Lista polskich słów nierelewantnych. https://www.ranks.nl/stopwords/polish. [dostęp dnia 26 stycznia 2018].

[3] Text classification and Naive Bayes. https://nlp.stanford.edu/IR-book/html/htmledition/text-classification-and-naive-bayes-1.html, 2009. [dostęp dnia 08 kwietnia 2018].

[4] Dokumentacja formatu CCL. http://nlp.pwr.wroc.pl/redmine/projects/corpus2/wiki/CCL_format, 2011. [dostęp dnia 20 stycznia 2018].

[5] Internetowy podręcznik statystyki. http://www.statsoft.com.pl/textbook/stathome_stat.html, 2011. [dostęp dnia 11 stycznia 2018].

[6] Dokumentacja NLP Rest API. http://nlp.pwr.wroc.pl/redmine/projects/nlprest2/wiki/Python, 2013. [dostęp dnia 08 lutego 2018].

[7] Vector Representations of Words. https://www.tensorflow.org/tutorials/word2vec, 2018. [dostęp dnia 20 maja 2018].

[8] M. Baj. Metody komputerowej analizy stylometrycznej tekstów w języku polskim. Praca magisterska, Politechnika Wrocławska, 2016.

[9] J. Bloch. *Effective Java: Third Edition*. Addison-Wesley Professional, 2017.

[10] P. Bojanowski, E. Grave, A. Joulin, T. Mikolov. Faster, better text classification! https://research.fb.com/fasttext/, 2016. [dostęp dnia 27 maja 2018].

[11] P. Bojanowski, E. Grave, A. Joulin, T. Mikolov. Library for fast text representation and classification. https://github.com/facebookresearch/fastText, 2018.

[12] E. Bottard. Spring Shell Reference Documentation. https://docs.spring.io/spring-shell/docs/current-SNAPSHOT/reference/htmlsingle/, 2017.

[13] C. Boulis, M. Ostendorf. Text Classification by Augmenting the Bag-of-Words Representation with Redundancy-Compensated Bigrams. *Proceedings of the SIAM International Conference on Data Mining at the Workshop on Feature Selection in Data Mining (SIAM-FSDM 2005)*, 2005.

[14] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, P. Kuksa. Natural Language Processing (almost) from Scratch. https://arxiv.org/pdf/1103.0398.pdf, 2011. [dostęp dnia 30 stycznia 2018].

[15] B. Eckel. *Thinking in Java (4th Edition)*. Prentice Hall, 2006.

[16] E. Gamma, R. Helm. *Design Patterns. Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[17] B. Góralewicz. The TF*IDF Algorithm Explained. `https://www.elephate.com/blog/what-is-tf-idf/`, 2018. [dostęp dnia 22 marca 2018].

[18] S. Hassan, M. Rafi, M. S. Shaikh. Comparing SVM and naïve Bayes classifiers for text categorization with Wikitology as knowledge enrichment. *Multitopic Conference (INMIC), 2011 IEEE 14th International*, 2011.

[19] J. D. Hunter. *Matplotlib: A 2D graphics environment*, wolumen 9. IEEE COMPUTER SOC, 2007.

[20] W. Jaworski. *Miary jakości*. Instytut Informatyki Uniwersytety Warszawskiego, 2008.

[21] A. Joulin, E. Grave, P. Bojanowski, M. Douze, H. Jégou, T. Mikolov. Fasttext.zip: Compressing text classification models. *CoRR*, abs/1612.03651, 2016.

[22] A. Joulin, E. Grave, P. Bojanowski, T. Mikolov. Bag of tricks for efficient text classification. *CoRR*, abs/1607.01759, 2016.

[23] E. Lavieri. *Mastering Java 9: Write Reactive, Modular, Concurrent, and Secure*. Packt Publishing, 2017.

[24] M. Lutz. *Learning Python*. O'Reilly Media, 2013.

[25] P. Malak. *Indeksowanie treści*. Wydawnictwo SBP, 2008.

[26] C. Manning, P. Raghavan, H. Schütze. *Introduction to Information Retrieval*. Cambridge University Press, 2008.

[27] R. C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008.

[28] A. Mykowiecka. *Inżynieria lingwistyczna, Komputerowe przetwarzanie tekstów w języku naturalnym*. Wydawnictwo PJWSTK, 2007.

[29] T. Oliphant. *A guide to NumPy*. CreateSpace Independent Publishing Platform, 2006.

[30] F. Pedregosa, G. Varoquaux, A. Gramfort. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research (2011)*, 2011. [dostęp dnia 21 lutego 2018].

[31] M. Piasecki. Polish Tagger TaKIPI: Rule Based Construction and Optimisation. *Task Quarterly*, 11(1–2):151–167, 2007.

[32] M. Piasecki, A. Radziszewski. Polish Morphological Guesser Based on a Statistical a Tergo Index. *Proceedings of the International Multiconference on Computer Science and Information Technology — 2nd International Symposium Advances in Artificial Intelligence and Applications (AAIA'07)*, strony 247–256, 2007.

[33] A. Przepiórkowski. Zestaw znaczników morfosyntaktycznych. `http://nkjp.pl/poliqarp/help/plse2.html`, 2011. [dostęp dnia 20 stycznia 2018].

[34] A. Radziszewski. A tiered CRF tagger for Polish. *Intelligent Tools for Building a Scientific Information Platform: Advanced Architectures and Solutions*. 2013.

[35] R. Rivest. The md5 message-digest algorithm. `https://www.ietf.org/rfc/rfc1321.txt`, 1992.

[36] H. Rolf. *Statystyka dla językoznawców*. Wydawnictwo UW, 1990.

[37] M. Rudolf, M. Świdziński. Automatic utterance boundaries recognition in large Polish text corpora. M. A. Kłopotek, S. T. Wierzchoń, K. Trojanowski, redaktorzy, *Intelligent Information Processing and Web Mining*, strony 247–256, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[38] Z. Saloni, W. Gruszczyński, M. Woliński, R. Wołosz, D. Skowrońska. Morfeusz 2 ze słownikiem SGJP i Polimorf. `http://sgjp.pl/morfeusz/morfeusz.html`, 2018. [dostęp dnia 27 marca 2018].

[39] P. Semberecki, H. Maciejewski. Deep learning methods for subject text classification of articles. strony 357–360, 09 2017. [dostęp dnia 02 maja 2018].

[40] T. Walkowiak, S. Datko, H. Maciejewski. Feature extraction in subject classification of text documents in polish. L. Rutkowski, R. Scherer, M. Korytkowski, W. Pedrycz, R. Tadeusiewicz, J. M. Zurada, redaktorzy, *Artificial Intelligence and Soft Computing*, strony 445–452, Cham, 2018. Springer International Publishing.

[41] T. Walkowiak, M. Kaliński. Analizator morfo-syntaktyczny. `http://ws.clarin-pl.eu/tager.shtml#`, 2011. [dostęp dnia 14 stycznia 2018].

[42] C. Walls. *Spring w Akcji*. Wydawnictwo Helion, 2015.

[43] P. Webb, D. Syer, J. Long, S. Nicoll, A. Wilkinson. Spring Boot Reference Guide. `https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/`, 2018.

[44] Wikipedia. Wiki train - 34 categories, 2015. CLARIN-PL digital repository.

[45] M. Woliński. Morfeusz - a Practical Tool for the Morphological Analysis of Polish. M. A. Kłopotek, S. T. Wierzchoń, K. Trojanowski, redaktorzy, *Intelligent Information Processing and Web Mining*, strony 511–520, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[46] B. Yuan, Y. Zhou. The improvement of LDA: Considering avoiding repetition. 2008.

[47] Y. Zhang, R. Jin, Z. Zhou. Understanding Bag-of-Words Model: A Statistical Framework. *International Journal of Machine Learning and Cybernetics*, strony 1–32, 2009.

Dodatek A

Dodatek A

- Płyta CD z programem
- Dokumentacja