Uniwersytet im. A. Mickiewicza w Poznaniu

Wydział Matematyki i Informatyki

Praca magisterska

Ekstrakcja informacji o godzinach rozpoczęcia mszy świętych

Extracting information about church services start times

Dawid Jurkiewicz

Numer albumu: 396341

Kierunek: Informatyka

Specjalność: Przetwarzanie języka naturalnego

Promotor:

prof. UAM, dr hab. Krzysztof Jassem

Poznań, 2018

Poznań, dnia 22 czerwca 2018

Oświadczenie

Ja, niżej podpisany Dawid Jurkiewicz, student Wydziału Matematyki i Informatyki Uniwersytetu im. Adama Mickiewicza w Poznaniu oświadczam, że przedkładaną pracę dyplomową pt.

„Ekstrakcja informacji o godzinach rozpoczęcia mszyświętych”,

napisałem samodzielnie. Oznacza to, że przy pisaniu pracy, poza niezbędnymi konsultacjami, nie korzystałem z pomocy innych osób, a w szczególności nie zlecałem opracowania rozprawy lub jej części innym osobom, ani nie odpisywałem tej rozprawy lub jej części od innych osób.

Oświadczam również, że egzemplarz pracy dyplomowej w wersji drukowanej jest całkowicie zgodny z egzemplarzem pracy dyplomowej w wersji elektronicznej.

Jednocześnie przyjmuję do wiadomości, że przypisanie sobie, w pracy dyplomowej, autorstwa istotnego fragmentu lub innych elementów cudzego utworu lub ustalenia naukowego stanowi podstawę stwierdzenia nieważności postępowania w sprawie nadania tytułu zawodowego.

[ ]* - wyrażam zgodę na udostępnianie mojej pracy w czytelni Archiwum UAM

[ ]* - wyrażam zgodę na udostępnianie mojej pracy w zakresie koniecznym do ochrony mojego prawa do autorstwa lub praw osób trzecich

*Należy wpisać TAK w przypadku wyrażenia zgody na udostępnianie pracy w czytelni Archiwum UAM, NIE w przypadku braku zgody. Niewypełnienie pola oznacza brak zgody na udostępnianie pracy.

....

Spis treści

Wstęp **12**

1. Ekstrakcja godzin rozpoczęcia mszy świętych **15**

1.1. Ogólny zarys systemu 15

1.2. Zbieranie informacji o parafiach 18

1.3. Wyszukiwanie adresów URL parafii 20

1.4. Indeksowanie stron parafialnych 22

1.4.1. Komponenty pająka 23

1.4.2. Przepływ danych 24

1.4.3. Sprawdzanie typu odpowiedzi 25

1.4.4. Automatyczna regulacja częstości zapytań 28

1.4.5. Indeksowanie wieloprocesorowe 30

1.4.6. Organizacja danych 31

1.5. Konwersja HTML na tekst. 32

1.6. Ekstrakcja godzin 33

1.7. Filtrowanie stron 34

1.8. Anotator danych 36

1.8.1. Ogólny zarys 37

1.8.2. Identyfikacja urządzeń 39

1.8.3. Architektura anotatora 41

1.9. Regułowa ekstrakcja godzin mszy 43

1.10. Klasyfikacja godzin 45
1.10.1. Model teoretyczny 46
1.10.2. FastText 52

2. Rezultaty **53**

3. Podsumowanie **56**

Spis rysunków **59**

Spis algorytmów **60**

Spis tabel **61**

Bibliografia **63**

Streszczenie

Praca przedstawia proces tworzenia systemu ekstrakcji informacji o godzinach rozpoczęcia mszy świętych. Opisane zostają sposoby zbierania danych o polskich parafiach, w szczególności proces tworzenia pająków. Następnie pokazane zostają dwie metody ekstrakcji godzin rozpoczęcia mszy świętych: regułowa i oparta na uczeniu maszynowym. Większa uwaga zostaje poświęcona metodzie opartej na uczeniu maszynowym, która polega na wykorzystaniu klasyfikatora tekstu.

Słowa kluczowe: ekstrakcja informacji, indeksowanie stron internetowych, klasyfikacja tekstu

Abstract

The thesis presents the process of creating a system for extracting information about opening hours of holy masses. The methods of collecting data of Polish parishes are being described, especially the process of creating spiders. Then two methods are shown for the extraction of opening hours of masses: a rule-based method and machine learning-based method. More attention is devoted to machine learning-based method that uses a text classifier.

Key words: information extraction, web spidering, text classification

Wstęp

Msza święta to najważniejsze duchowe wydarzenie w tygodniu chrześcijanina. Każdy wierzący katolik uczęszcza na niedzielną mszę świętą, a wielu również na msze święte w dni powszednie. Ze względu na rangę tego wydarzenia powstało kilka serwisów, które umożliwiają wyszukiwanie najbliższej godziny i miejsca mszy świętej. Z jednej strony powstały wyszukiwarki ogólnopolskie takie jak `kiedymsza.pl` lub `msze.info`. Wadą tych wyszukiwarek jest to, że wyświetlane godziny mszy świętych bardzo często są albo błędne, albo już nieaktualne. Z tego względu nie zdobyły one dużej popularności. Z drugiej strony powstały wyszukiwarki lokalne jak na przykład wyszukiwarka mszy świętych dla archidiecezji łódzkiej `archidiecezja.lodz.pl/wyszukiwarka-mszy-swietych/` lub aplikacja mobilna Drogowskaz[1] służąca do wyszukiwania mszy świętych w archidiecezji poznańskiej. Oczywistą wadą tych wyszukiwarek jest to, że obejmują małą liczbę polskich parafii. Oferują one za to bardzo wiarygodne informacje i są częściej aktualizowane. Zarówno ogólnopolskie, jak i lokalne wyszukiwarki mszy świętych zbierają swoje dane manualnie. Dane wpisują albo internauci, albo autorzy wyszukiwarek. W celu zasięgnięcia informacji dzwonią oni do parafii lub przepisują godziny mszy świętych ze stron parafialnych. Takie postępowanie jest bardzo czasochłonne i kosztowne.

Niniejsza praca przedstawia system, który służy do automatycznego zbierania informacji o godzinach rozpoczęcia mszy świętych. W pierwszym rozdziale

[1] `www.aplikacjadrogowskaz.pl`

dokładnie opisano system ekstrakcji godzin mszy świętych. W drugim rozdziale przedstawiono osiągnięte rezultaty. W trzecim rozdziale podsumowano pracę.

W momencie oddania pracy do druku system opisany w niniejszej pracy jest jedynym w Polsce automatycznym system ekstrakcji godzin mszy świętych. System przykuł uwagę jednego z autorów Drogowskazu. Zostały przeprowadzone wstępne rozmowy i jeśli system się sprawdzi, to być może będzie dostarczał dane dla aplikacji Drogwskaz.

Rozdział 1

Ekstrakcja godzin rozpoczęcia mszy świętych

1.1. Ogólny zarys systemu

Architektura systemu ekstrakcji godzin rozpoczęcia mszy świętych została przedstawiona na rysunku 1.1. W niniejszym podrozdziale zostanie ona krótko opisana. Szczegółowy opis poszczególnych komponentów znajduje się w podrozdziałach 1.2 - 1.10.

System zaczyna działanie od zebrania jak największej ilości danych (nazwa parafii, adres, diecezja itd.) o polskich parafiach ze strony `deon.pl`. Następnie wysyła zapytania do interfejsu API Google w celu znalezienia adresów internetowych parafii. Dla każdej parafii, dla której udało się znaleźć adres URL, pobierane są wszystkie podstrony w odległości[1] co najwyżej 3 od strony startowej.

Z dużej liczby stron parafialnych, za pomocą reguł wyodrębnione zostają te, na których z dużym prawdopodobieństwem znajdują się godziny mszy świętych. Na-

[1]Zakładamy, że sieć jest grafem, zatem odległość definiujemy tak jak w teorii grafów.

stępnie godziny zostają wydobyte ekstraktorem godzin o bardzo niskiej precyzji i bardzo wysokiej wartości pokrycia. Każda godzina wraz z kontekstem, w jakim się znajduje, trafia do anotatora. Tam jest oznaczana jako poprawna lub niepoprawna godzina mszy świętej[2]. Regułowy ekstraktor mszy świętych o bardzo wysokiej precyzji znajduje poprawne godziny mszy świętych i dołącza je do zaanotowanych danych. Co więcej, w celu wyrównania klas z nieodfiltrowanego zbioru stron parafialnych wylosowane zostają niepoprawne godziny mszy świętych. Zebrane dane zostają użyte do wytrenowania klasyfikatora godzin opartego na płytkich sieciach neuronowych.

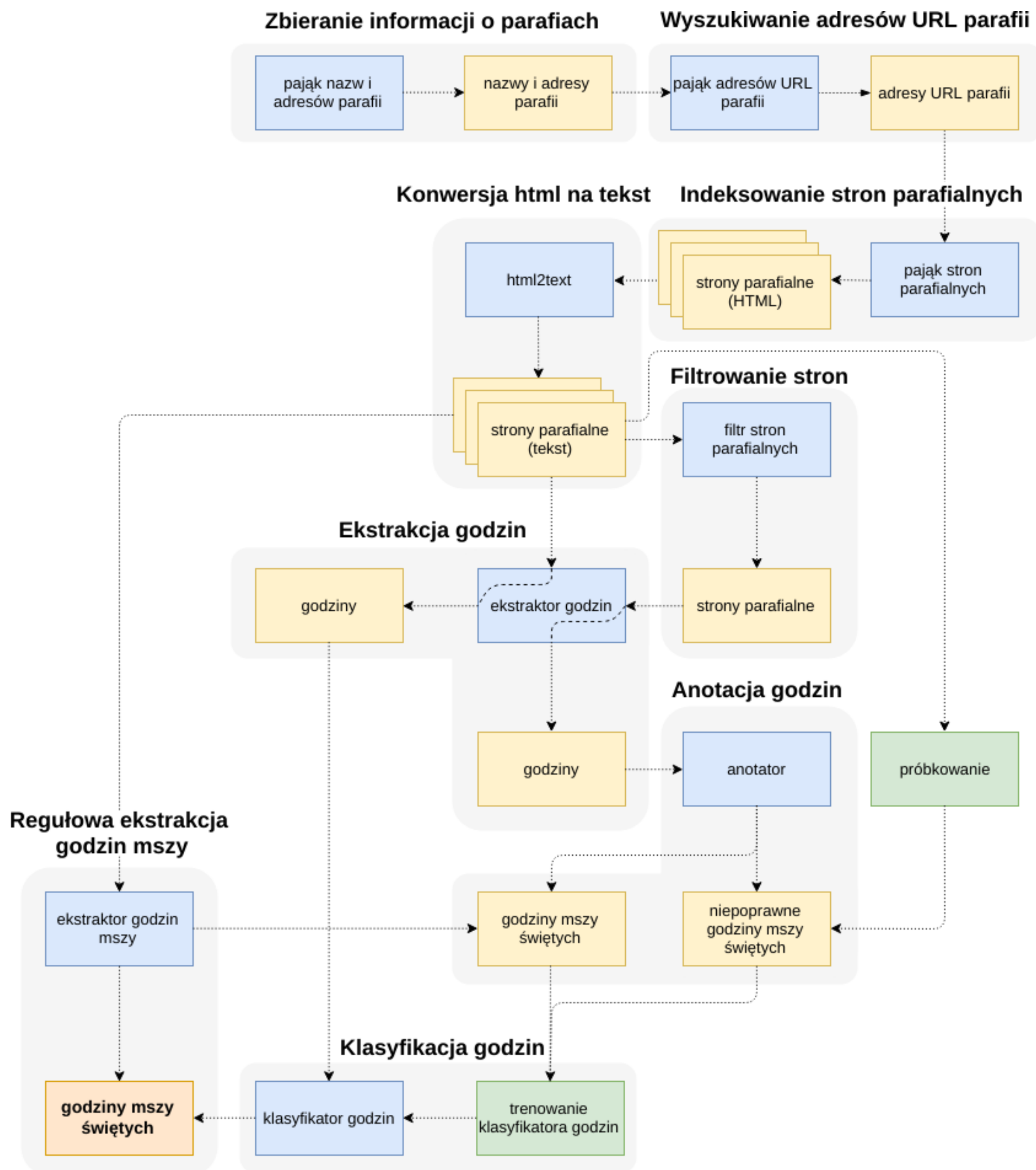
Klasyfikator używany jest do przyporządkowania godzin znalezionych przez ekstraktor godzin do następujących klas:

- poprawne godziny mszy świętych,
- niepoprawne godziny mszy świętych.

Docelowe godziny rozpoczęcia mszy świętych otrzymujemy z:

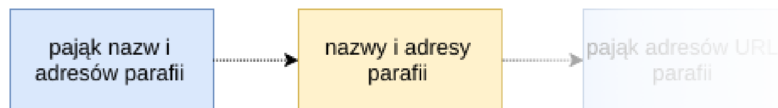
- ekstraktora godzin mszy świętych,
- klasyfikatora godzin.

[2]Przez „niepoprawne godziny mszy świętych” rozumiemy godziny, które nie są godzinami rozpoczęcia mszy świętych.



Rys. 1.1. Architektura systemu do ekstrakcji godzin mszy świętych.

1.2. Zbieranie informacji o parafiach



Rys. 1.2. Fragment architektury systemu przedstawiający komponent odpowiedzialny za zbieranie informacji o parafiach.

Na rysunku 1.2 został przedstawiony fragment architektury systemu z rysunku 1.1, który zostanie omówiony w niniejszym podrozdziale.

Dane zostały zebrane z serwisu internetowego deon.pl, który zawiera 10130 parafii. Są to prawie wszystkie polskie parafie, ponieważ według danych statystycznych GUS[17] z 2016 roku w Polsce było 10255 parafii.

Dla każdej parafii zebrano:

- nazwę parafii,
- miejscowość, w której się znajduje,
- dokładny adres,
- nazwę dekanatu, do którego należy,
- nazwę diecezji, do której przynależy,
- województwo, w którym się znajduje.

Fragment zebranych danych został przedstawiony w tabeli 1.1.

Parafia	Miejscowość	Adres	Diecezja	Dekanat	Województwo
Bożego Ciała	Hel	ul. Gdań...	gdańska	Morski	pomorskie
Ducha Św.	Śrem	ul. Prym...	poznańska	Śrem	wielkopolskie
Św. Trójcy	Paszowice	Paszowic...	legnicka	Jawor	dolnośląskie

Tab. 1.1. Fragment danych zebranych przez pająka nazw i adresów parafii.

Do wydobycia danych użyto skryptu w języku Python, który korzystał z parsera HTML z biblioteki *Beautiful Soup*[9]. Przy wysyłaniu zapytań do serwisu deon.pl zastosowano algorytm *Expotential Backoff* [7] (patrz algorytm 1).

Algorytm 1: *Expotential Backoff*

Stałe:

max_wait_time – maksymalny czas oczekiwania.
repeat_limit – limit liczby powtórnych zapytań pod rząd.

Algorytm:

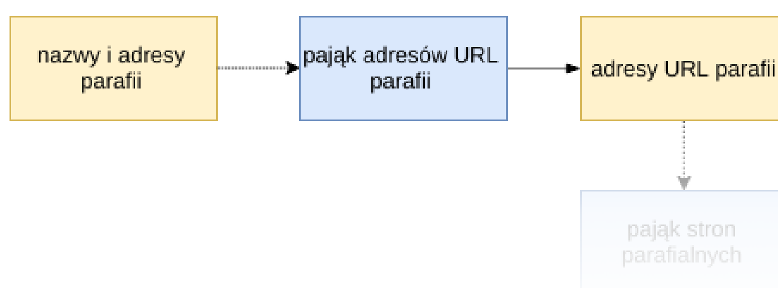
1. Wyślij zapytanie do serwisu;
2. Jeśli zapytanie się nie powiodło, poczekaj 2s i wyślij kolejne zapytanie,
3. Jeśli zapytanie się nie powiodło, poczekaj 4s i wyślij kolejne zapytanie,
4. Jeśli zapytanie się nie powiodło, poczekaj 8s i wyślij kolejne zapytanie,
5. Powtarzaj do czasu aż zapytanie się powiedzie lub liczba ponownych zapytań pod rząd wyniesie repeat_limit.

gdzie:

- Czas oczekiwania to 2^t, gdzie t to liczba nieudanych zapytań.
- Czas oczekiwania nie może być większy niż max_wait_time.

Algorytm 1 uodparnia skrypt na przejściowe problemy z połączeniem i zapobiega zbyt częstemu wysyłaniu zapytań do serwisu. Dla przykładu załóżmy, że dany serwis jest obciążony i odpowiada na zapytanie z dużym opóźnieniem. Wtedy algorytm *Expotential Backoff* przy każdym kolejnym niepowodzeniu będzie czekał coraz dłużej, zanim wyśle kolejne zapytanie. W ten sposób nie będzie niepotrzebnie obciążał serwisu.

1.3. Wyszukiwanie adresów URL parafii



Rys. 1.3. Fragment architektury systemu przedstawiający komponent odpowiedzialny za wyszukiwanie adresów URL parafii.

Na rysunku 1.3 został przedstawiony fragment architektury systemu z rysunku 1.1, który zostanie omówiony w niniejszym podrozdziale.

Pierwsze próby

Do wyszukiwania adresów URL parafii próbowano wykorzystać wyszukiwarki takie jak Google i DuckDuckGo. Automatycznie wysyłano zapytanie złożone z konkatenacji nazwy parafii, jej miejscowości i ulicy, na której się znajduje. Wyszukiwarka Google dawała zadowalające wyniki, jednak po kilkunastu zapytaniach blokowała adres IP. Ponadto, w warunkach użytkowania serwisu i w pliku *robots.txt* Google

zabrania korzystania z pająków na ich wyszukiwarce. Wyszukiwarka DuckDuckGo nie blokowała adresu IP, ale zabraniała indeksowania w pliku *robots.txt* i słabo radziła sobie z polskimi zapytaniami. W obu przypadkach powyższa metoda stwarzała kolejny problem do rozwiązania – z wielu wyników wyszukiwania trzeba było wybrać ten, który zawierał adres URL parafii.

Rozwiązanie

Po wielokrotnych próbach poszukiwań znaleziono klucz do rozwiązania problemu wyszukiwania adresów URL, jakim jest *Google Places API*[5]. Serwis *Text Search*[4] pozwala na wyszukanie miejsca danego obiektu na podstawie jego nazwy. Ponadto, mając już wyszukany dany obiekt i jego identyfikator można odpytać serwis *Place Detail*[3], aby wyciągnąć więcej szczegółów o danym miejscu. Między innymi można otrzymać adres URL danego obiektu.

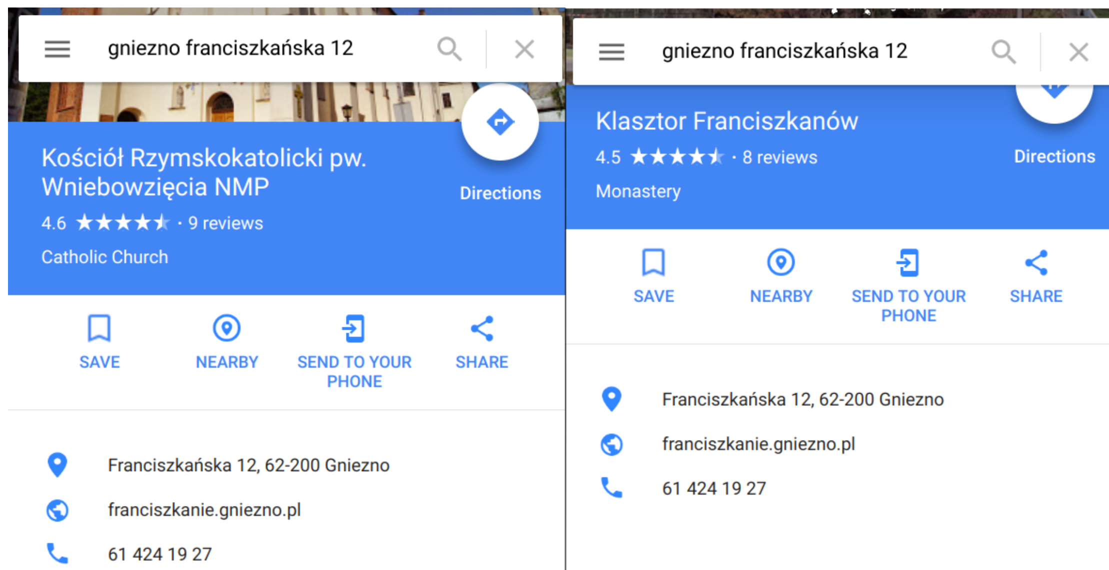
Jedynym minusem jest ograniczenie liczby zapytań do 1000 na 24 godziny. W dodatku, każde zapytanie do serwisu *Text Search* traktowane jest jak 10 zapytań. Podając swoją kartę płatniczą, można zwiększyć limit zapytań do 150 000 na 24 godziny. Karta płatnicza jest potrzebna Google do identyfikacji osoby. Żadna opłata nie jest pobierana za korzystanie z interfejsu API.

Dla każdej parafii wykonywane jest zapytanie do serwisu *Text Search*. Składa się ono z konkatenacji nazwy parafii, jej miejscowości i ulicy, na której się znajduje. Jeśli nie zostanie znaleziony żaden obiekt, wysyłane jest powtórne zapytanie, lecz tym razem składające się tylko z nazwy parafii i jej miejscowości.

Zdarza się, że serwis *Text Search* zwraca kilka obiektów. W takim przypadku brany jest adres URL pierwszego obiektu z listy wyników. Najczęściej jednak oba obiekty należą do tej samej parafii, więc mają taki sam adres internetowy. Taki przypadek przedstawia rysunek 1.4. Serwis *Text Search* zwraca dużo danych w formacie JSON, które trudno jest przedstawić w przejrzystej postaci. Dla czytel-

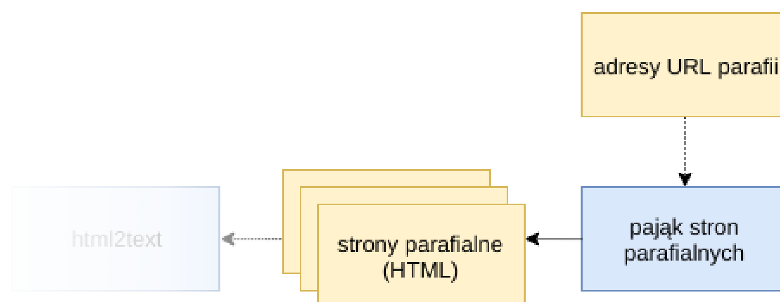
ności na rysunku 1.4 pokazano zrzuty ekranu z wyszukiwarki *Google Maps*, które odpowiadają rezultatowi, jaki otrzymano, korzystając z serwisu *Text Search*.

Powyższą metodą udało się zebrać adresy URL dla ok. 5600 parafii.



Rys. 1.4. Przykład dwóch obiektów zwróconych przez wyszukiwarkę Google, które mają ten sam adres internetowy.

1.4. Indeksowanie stron parafialnych



Rys. 1.5. Fragment architektury systemu przedstawiający komponent odpowiedzialny za indeksowanie stron parafialnych.

Na rysunku 1.5 został przedstawiony fragment architektury systemu z rysunku 1.1, który zostanie omówiony w niniejszym podrozdziale.

Pająk został napisany przy użyciu biblioteki *Scrapy*[14]. Punktem startowym jest pojedynczy adres URL parafii podawany na wejście programu. Z początkowego adresu URL wydobywana jest domena, w której obrębie porusza się pająk. Oznacza to, że jedna instancja pająka zajmuje się pobieraniem tylko jednej parafii. W ramach jednej parafii pająk jest w stanie asynchronicznie wysłać wiele zapytań do serwera i odbierać wiele odpowiedzi od serwera.

1.4.1. Komponenty pająka

Pająk składa się z następujących komponentów:

Silnik – odpowiada za kontrolę przepływu danych i komunikację między komponentami.

Dyspozytor – otrzymuje żądania od *silnika*, kolejkuje je i na prośbę *silnika* odsyła z powrotem.

Downloader – odpowiada za pobieranie stron parafialnych i przekazywanie ich *silnikowi.*

Procesor danych – zajmuje się końcową obróbką i zapisem danych.

Spider [3] - definiuje sposób, w jaki pobierać dane, między innymi jak parsować stronę i za jakimi linkami podążać.

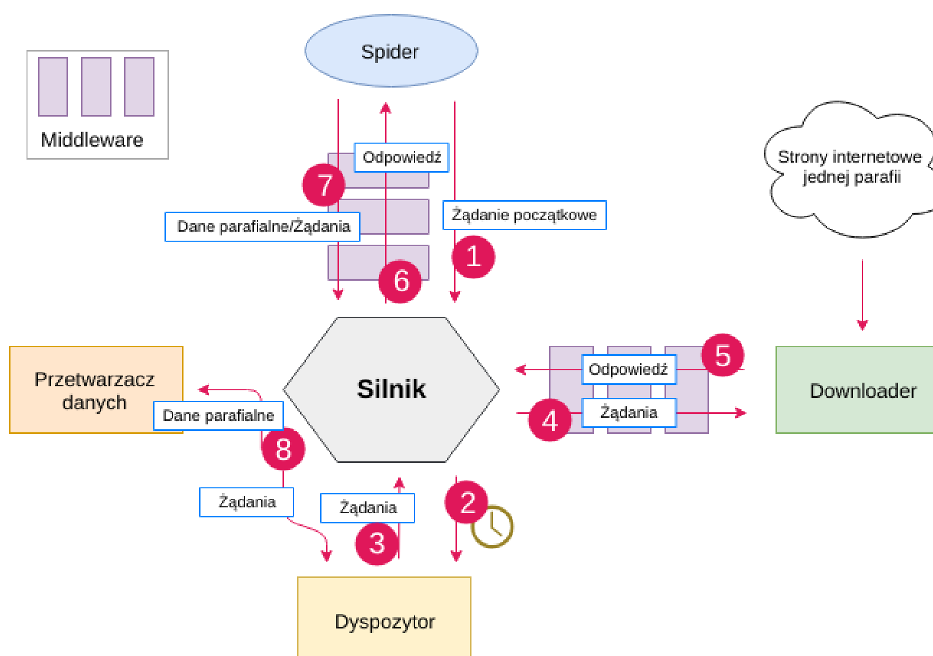
Spider middleware – programy pośredniczące między *silnikiem*, a *spider'em.* Odpowiedzialne są za dodatkowe przetwarzanie danych wyjściowych (dane parafialne i żądania) i wejściowych (odpowiedzi) *spider'a.*

[3] Użyto angielskiej nazwy, aby rozróżnić *spider'a* (komponent pająka), od pająka (cały program odpowiedzialny za indeksowanie stron parafialnych).

Downloader middleware – programy pośredniczące między silnikiem, a *downloader'em*. Zajmują się dodatkowym przetwarzaniem danych wejściowych (żądań) i wyjściowych (odpowiedzi) *downloader'a*.

1.4.2. Przepływ danych

Przepływ danych kontrolowany jest przez *silnik* i ma postać przedstawioną na rysunku 1.6[4]:



Rys. 1.6. Silnik kontrolujący przepływ danych przez komponenty pająka.

1. *Silnik* otrzymuje od *spider'a* żądanie pobrania początkowej strony danej parafii (najczęściej jest to strona główna parafii).
2. *Silnik* oddaje żądania *dyspozytorowi*, który kolejkuje je do dalszego przetwarzania oraz pyta *dyspozytor* o żądania gotowe do przekazania *downloader'owi*.

[4]Diagram i opis wzorowany jest na dokumentacji znajdującej się pod linkiem https://doc.scrapy.org/en/latest/topics/architecture.html.

3. *Dyspozytor* zwraca *silnikowi* następne żądania.

4. *Silnik* wysyła żądania do *downloader'a*. Zanim żądania dotrą do *downloader'a*, przetwarzane są przez *downloader middleware*.

5. *Downloader* pobiera stronę parafialną i umieszcza ją w odpowiedzi, którą przesyła *silnikowi*. Zanim odpowiedź dotrze do *silnika*, przetwarzana jest przez *downloader middleware*.

6. *Silnik* otrzymuje odpowiedź od *downloader'a* i przekazuje ją *spider'owi* do dalszego przetwarzania. Zanim odpowiedź trafi, do *spider'a* przetwarzana jest przez *spider middleware*.

7. *Spider* przetwarza odpowiedź i zwraca dane strony parafialnej *silnikowi*. Zanim dane trafią do *silnika*, przechodzą przez *spider middleware*. Dodatkowo *spider* wysyła żądania z nowymi stronami parafialnymi do pobrania.

8. *Silnik* wysyła zebrane dane do *procesora danych*, który zapisuje je do pliku. Następnie przekazuje nowe żądania do zakolejkowania *dyspozytorowi*.

Cały proces trwa dopóty, dopóki są nowe żądania do przetworzenia.

1.4.3. Sprawdzanie typu odpowiedzi

Podczas indeksowania ważne jest rozpoznawanie typu pobieranych danych. W przypadku indeksowania stron parafialnych przedmiotem zainteresowania są wyłącznie dane tekstowe. Należy zatem zadbać o to, aby nie pobierać danych binarnych takich jak np. video, audio i obrazy.

Biblioteka *Scrapy* obsługuje rozpoznawanie typu zawartości odpowiedzi, bazując na następujących kryteriach:

- wartości `Content-type`[11], `Content-Encoding`[10] i `Content-Disposition`[15] w nagłówku odpowiedzi;
- nazwie pliku lub adresie URL (jeśli nie udało się rozpoznać po nagłówku);
- obecności znaków kontrolnych w pierwszych 5000 bajtów odpowiedzi (jeśli nie udało się rozpoznać po nazwie pliku lub adresie URL).

Powyższy schemat postępowania jest skuteczny, jeśli serwisy internetowe zwracają poprawne odpowiedzi. Niestety, niektóre strony parafialne zwracają odpowiedzi, które nie są zgodne z rozdziałem 3.1 z dokumentu RFC7231[10][5]. Dla przykładu strona potrafi zwrócić `Content-Type: text/html` w nagłówku, a w ciele – binarną zawartość np. film. Tego rodzaju anomalie są wykrywane i eliminowane. Stosując algorytm 2, można określić typ zawartości ciała odpowiedzi.

Algorytm 2: Rozpoznawanie plików binarnych

```
   Wejście: bytes ← 1024 pierwsze bajty pliku
   Wyjście: True jeśli plik binarny, False jeśli plik tekstowy
1  if bytes puste or bytes dekodowalne jako unikod then
2  |  return False;
   else if znak null in bytes then
3  |  return True;
4  end
5  /* Za znaki kontrolne uznajemy znaki o kodach EASCII od 0 do
      7, 11, od 14 do 32 i od 127 do 159.                       */
6  control_char_count ← liczba znaków kontrolnych w bytes;
7  high_ascii_count ← liczba znaków EASCII o kodach od 160 do 255 w
    bytes;
8  control_char_ratio ← control_char_count / 1024;
9  high_ascii_ratio ← high_ascii_count / 1024;
10 if (control_char_ratio > 0.3 or high_ascii_ratio > 0.7) then
11 |  return True;
12 end
13 return False;
```

[5] RFC to zbiór technicznych dokumentów w formie memorandum opisujących protokoły związane z Internetem i sieciami komputerowymi.

Algorytm 2 analizuje zawartość ciała odpowiedzi w celu stwierdzenia, czy jest ona binarna, czy nie. W linii 6 za znaki kontrolne uznano wszystkie znaki kontrolne ze zbioru C0[6] i C1[7] z wyłączeniem następujących znaków:

- znak nowej linii (oznaczany przez `\n`),
- znak powrotu karetki (oznaczany przez `\r`),
- znak tab (oznaczany przez `\t`),
- znak backspace (oznaczany przez `\b`),
- znak nowej linii (oznaczany przez `\n`),
- znak końca strony (oznaczany przez `\f`),

Powyższe znaki zostały wykluczone, ponieważ często występują w plikach tekstowych.

Warto zwrócić uwagę na linię 10. Współczynnik `control_char_ratio` oznacza procent znaków kontrolnych w pierwszych 1024 bajtach pliku. Jeśli współczynnik `control_char_ratio` jest większy niż $0,3$, to plik jest uznawany za binarny. Wartość $0,3$ została przyjęta z rozwiązania z kodu[8] źródłowego języka Perl, który między innymi zajmuje się rozpoznawaniem plików binarnych. Natomiast współczynnik `high_ascii_ratio` oznacza procent znaków EASCII[9] o kodach od 160 do 255. Reprezentacja tych znaków zależy od rozszerzenia ASCII. Najczęściej jednak są to znaki drukowalne, które rzadko występują w tekście. Jeśli współczynnik

[6] C0 to znaki kontrolne z kodu ASCII o kodach od 0 do 32 i o kodzie 127.

[7] C1 to znaki kontrolne o kodach od 128 do 159. Zostały zdefiniowane w standardzie ISO/IEC 2022. Wiele innych systemów kodowań rezerwuje sobie kody od 128 do 159 na znaki kontrolne.

[8] Kod znajduje się pod linkiem `https://github.com/Perl/perl5/blob/v5.27.11/pp_sys.c#L3605-L3665`. Wartość 0,3 występuje w linii 3661.

[9] EASCII oznacza rozszerzone kodowanie ASCII. Przykładowe rozszerzenia to systemy kodowania ISO 8859 lub UTF-8.

`high_ascii_ratio` jest większy niż $0,7$, to plik jest uznawany za binarny. Wartość $0,7$ została dobrana na podstawie następujących obserwacji:

1. Zdarzają się pliki binarne, które mają dużo znaków `high_ascii`. Przykładem jest plik z katalogu `data.tar.gz/spec/resources/pixelstream.rgb` z archiwum `https://rubygems.org/downloads/chunky_png-1.2.8.gem`. Plik zawiera bardzo dużo znaków o kodzie 255 w początkowych bajtach.

2. Mało prawdopodobne jest, aby plik tekstowy miał w pierwszych 1024 bajtach więcej niż 70% znaków `high_ascii`. Nawet jeśli pająk natrafiłby na taki plik, to z dużym prawdopodobieństwem nie zawierałby on informacji parafialnych.

1.4.4. Automatyczna regulacja częstości zapytań

Biblioteka *Scrapy* zawiera przydatne rozszerzenie, które potrafi automatycznie regulować częstość zapytań w zależności od obciążenia pająka i serwera.

Algorytm 3 przedstawia sposób postępowania, w jaki pająk automatycznie reguluje częstość zapytań. Idea algorytmu jest następująca. Załóżmy, że serwer potrzebuje `latency`[10] sekund, aby odpowiedzieć pająkowi. Jeśli pająk chce mieć przetworzone równolegle `target_concurrency`[11] zapytań, to powinien wysyłać każde zapytanie co `latency/target_concurrency` sekund.

[10] Zmienna `latency` została zdefiniowana w algorytmie 3.
[11] Stała `target_concurrency` została zdefiniowana w algorytmie 3.

Algorytm 3: Algorytm regulacji częstości zapytań

Stałe:

init_download_delay	– początkowe opóźnienie wysłania zapytania.
min_download_delay	– minimalne opóźnienie wysłania zapytania.
max_download_delay	– maksymalne opóźnienie wysłania zapytania.
target_concurrency	– średnia wartość równoległych zapytań do wysłania.

Zmienne:

target_download_delay	– docelowe opóźnienie wysyłania zapytania.
download_delay	– opóźnienie wysłania zapytania.
latency	– czas od ustanowienia połączenia do otrzymania nagłówków odpowiedzi.

Algorytm:

1. Wyślij zapytanie do serwisu;
2. Ustaw download_delay $\leftarrow$ init_download_delay.
3. Gdy odebrano odpowiedź, ustaw target_download_delay $\leftarrow \frac{\text{latency}}{\text{target_concurrency}}$.
4. Ustaw download_delay $\leftarrow \frac{\text{download_delay} + \text{target_download_delay}}{2}$
5. Czekaj download_delay sekund.
6. Wyślij kolejne zapytanie do serwisu;
7. Wróć do kroku nr 3.

gdzie:

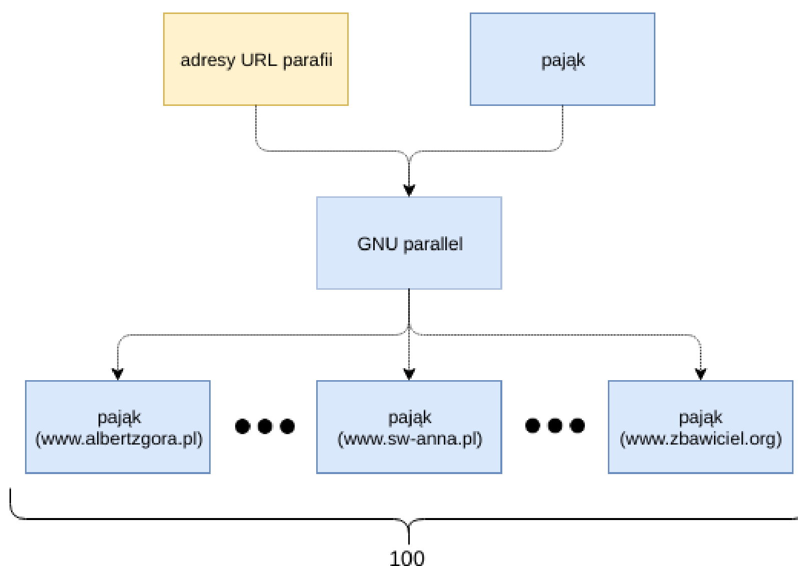
- Opóźnienia liczone są w sekundach.
- download_delay nie może być mniejszy niż min_download_delay i większy niż max_download_delay.
- Czasy oczekiwania na odpowiedzi z kodem http różnym od 2xx nie są brane pod uwagę.
- Algorytm kończy się, gdy nie ma więcej zapytań do wysłania.

W pająku stron parafialnych stałe z algorytmu 3 ustawiono następująco:

- `min_download_delay = 0`
- `max_download_delay = 300`
- `init_download_delay = 5`
- `target_concurrency = 1`

Stałe `min_download_delay` i `max_download_delay` zostały ustawione w taki sposób, aby nie ograniczać zbyt mocno pająka co do doboru wartości `download_delay`. Celem jest przecież automatyczna regulacja wartości `download_delay`. Niska wartość stałej `target_concurrency` umotywowana jest dużą liczbą równolegle pracujących pająków (patrz podrozdział 1.4.5).

1.4.5. Indeksowanie wieloprocesorowe



Rys. 1.7. 100 pająków pracujących jednocześnie.

Pająk został zaprojektowany w ten sposób, aby bardzo łatwo można było urównoleglić pobieranie stron parafialnych. Z pomocą programu *GNU parallel*[?] indeksowane jest jednocześnie 100 parafii (patrz rys. 1.7). Gdy jedna ze stu parafii zostanie pobrana, zastępuje ją kolejna parafia. Tym sposobem przez prawie cały czas równolegle pracuje 100 pająków. Takie podejście pozwoliło maksymalnie wykorzystać łącze internetowe, które było wąskim gardłem w procesie indeksowania stron parafialnych.

1.4.6. Organizacja danych

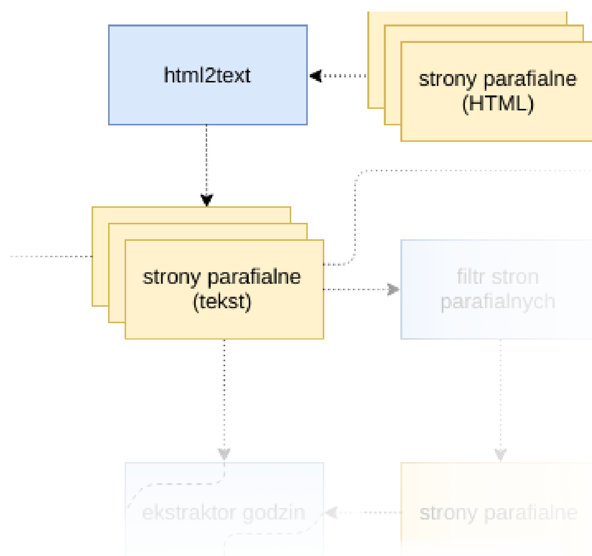
Dane zbierane przez pająka zapisywane są do pliku w formacie JSONL[2]. Format JSONL charakteryzuje się tym, że w każdej linii pliku znajduje się poprawny obiekt JSON. Dla każdej parafii pobrane dane zapisywane są w oddzielnym pliku. W każdej linii pliku znajduje się strona parafialna zapisana w formacie JSON. Taki sposób organizacji danych przynosi szereg korzyści takich jak:

1. wygodne przetwarzanie równoległe,
2. łatwa obróbka danych za pomocą narzędzi Uniksowych,
3. mniejszy rozmiar pliku w porównaniu do zwykłego formatu JSON.

Dla każdej strony parafialnej zapisywane są następujące informacje:

1. adres URL strony,
2. adres URL poprzedniej strony,
3. adres URL strony początkowej,
4. domena parafii,
5. strona parafii w formacie HTML,
6. tekst z odnośnika (linku), który doprowadził do bieżącej strony.

1.5. Konwersja HTML na tekst.



Rys. 1.8. Fragment architektury systemu przedstawiający komponent odpowiedzialny za konwersje HTML na tekst.

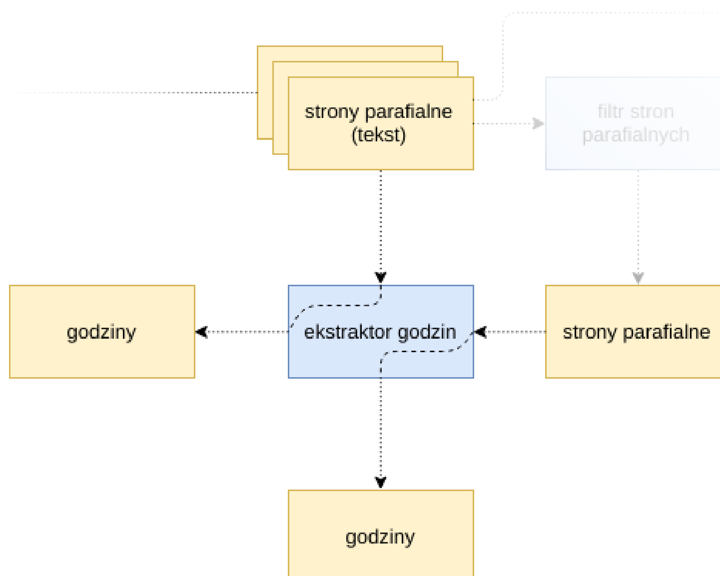
Na rysunku 1.8 został przedstawiony fragment architektury systemu z rysunku 1.1, który zostanie omówiony w niniejszym podrozdziale.

Do konwersji z formatu HTML na format tekstowy wykorzystano bibliotekę `html2text`[6] pierwotnie rozwijaną przez Aarona Schwartza. `html2text` konwertuje HTML na czysty, czytelny tekst w formacie *Markdown*[13]. Biblioteka oferuje wiele opcji do kontroli konwersji i jest bardzo łatwa w modyfikacji.

Zastosowano następujące opcje i modyfikacje przy konwersji:

- ignorowanie linków, tabel i obrazków,
- usuwanie znaków odpowiedzialne za pogrubienie i kursywę tekstu,
- usuwanie znaków odpowiedzialne za tworzenie list.

1.6. Ekstrakcja godzin



Rys. 1.9. Fragment architektury systemu przedstawiający komponent odpowiedzialny za ekstrakcję godzin ze stron parafialnych.

Na rysunku 1.8 został przedstawiony fragment architektury systemu z rysunku 1.1, który zostanie omówiony w niniejszym podrozdziale.

Ekstraktor godzin służy do znajdowania bardzo ogólnych ciągów znaków mogących być godzinami rozpoczęcia mszy świętych. Został napisany z myślą, aby miał bardzo wysoką wartość pokrycia, ale już niekoniecznie wysoką precyzję. Celem jest, aby w zbiorze wyekstrahowanych godzin znalazło się jak najwięcej godzin rozpoczęcia mszy, bez względu na to, jak duży jest ten zbiór.

Do osiągnięcia tego celu zastosowano następujące reguły. Ciąg znaków oznaczony jako `hour` zostanie wyekstrahowany, jeśli zajdzie każdy z poniższych warunków:

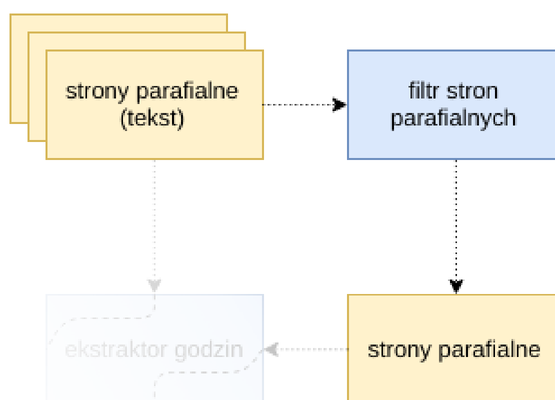
1. `hour` pasuje do wyrażenia regularnego
   `(0?[6-9]|1\d|2[0-2])[:.](oo|[0-5]\d)|6|7|8|9|1\d|2[0-2];`
2. Znak przed `hour` zawiera się w `{',', '('}`;

3. Znak po `hour` zawiera się w {`','`, `')'`, `';'`};

4. Jeśli znak przed `hour` równa się `'('`, to znak po `hour` jest różny od `')'`.

Ekstraktor wraz ze znalezioną godziną zapisuje kontekst, w jakim ta godzina się znalazła.

1.7. Filtrowanie stron



Rys. 1.10. Fragment architektury systemu przedstawiający komponent odpowiedzialny za filtrowanie stron parafialnych.

Na rysunku 1.10 został przedstawiony fragment architektury systemu z rysunku 1.1, który zostanie omówiony w niniejszym podrozdziale.

Filtr stron parafialnych ma za zadanie odnaleźć strony parafialne, na których z dużym prawdopodobieństwem znajdują się godziny mszy świętych. Taki zabieg jest potrzebny, aby ograniczyć liczbę godzin, które trafią do anotatora. Gdyby nie zastosowano filtru stron parafialnych, ekstraktor godzin wśród wszystkich stron parafialnych znalazłby ponad 3 miliony godzin. Po zaaplikowaniu filtru stron i przetworzeniu ich przez ekstraktor godzin otrzymano 10920 godzin. Później godziny wraz z kontekstami poddawane są anotacji. Etap ten będzie dokładniej opisany w podrozdziale 1.8.

Reguły zastosowane do zadecydowania czy dana strona zawiera godziny mszy świętych, zostały przedstawione w algorytmie 4.

Algorytm 4: Rozpoznawanie stron z godzinami mszy świętych.

Wejście:

url ← adres internetowy analizowanej strony,

btn_text ← tekst na przycisku, który doprowadził do analizowanej strony.

Wyjście:

- True jeśli jest wysokie prawdopodbieństwo, że strona zawiera godziny mszy
- False jeśli jest niskie prawdopodbieństwo, że strona zawiera godziny mszy

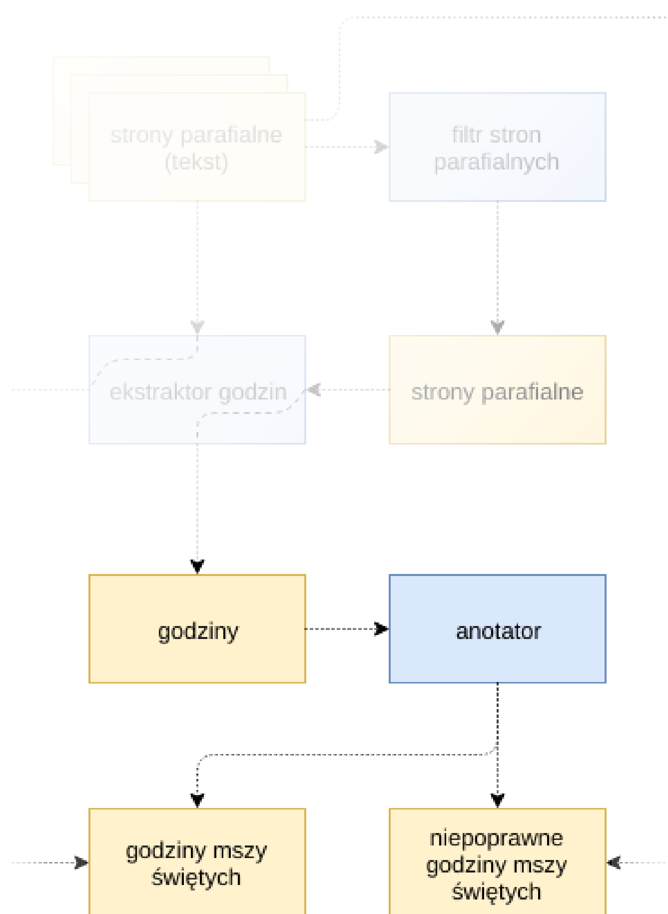
```
1  /* Wyrażenia regularne ok_regex i bad_regex ignorują wielkość
      liter.                                                   */
2  ok_regex ← 'msze|nabo[żz]e[ńn]stw(a|(?=\W\d)|$)|
     porz[ąa]dek($|\.htm)|porz[aą]dek.(liturgi|mszy)|
     (rozk[lł]ad|plan|godziny|uk[lł]ad|harmonogram|grafik|rozpiska).mszy'
3  bad_regex ← 'nabo[zż]e[nń]stwa.(majowe|wielk|czerwcowe
     |maryjne|pasyjne|pokutne|fatimskie|do|ro[żz]a|czterdzie|w.wielk)'
4  url_suf ← ciąg znaków w url po ostatnim wystąpieniu '/';
5  if (
6      ( url_suf pasuje do ok_regex and url_suf nie pasuje do bad_regex ) or
7      ( btn_text pasuje do ok_regex and btn_text nie pasuje do bad_regex )
8  ) then
9  |   return True;
10 else
   |   return False;
11 end
```

W algorytmie 4 warto zwrócić uwagę na wyrażenia regularne `ok_regex` i `bad_regex`. Wyrażenie regularne `ok_regex` ma za zadanie dopasować się do słów, które są powiązane z porządkiem mszy świętych. Stąd też pojawiają się tam wyrażenia takie

jak „rozkład mszy” lub „porządek liturgii”.

Wyrażenie regularne `bad_regex` ma za zadanie dopasować się do słów, które są powiązane z innymi nabożeństwami niż msze święte. Stąd pojawiają się tam wyrażenia takie jak „nabożeństwa czerwcowe” czy „nabożeństwa maryjne”.

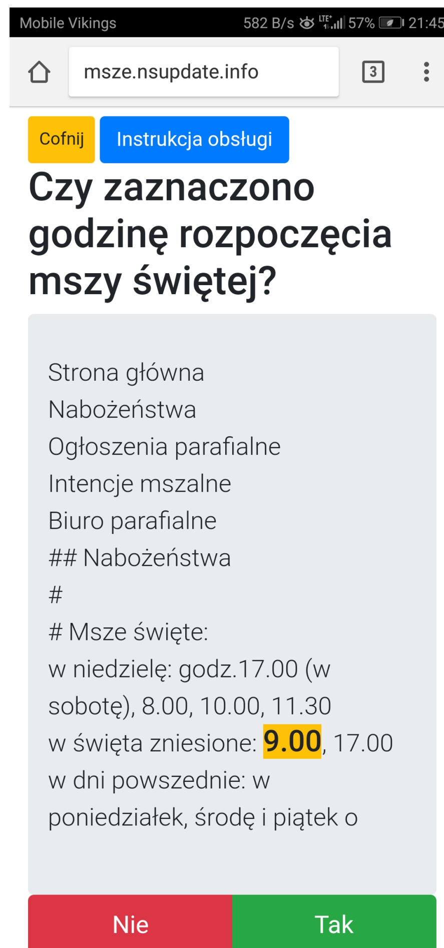
1.8. Anotator danych



Rys. 1.11. Fragment architektury systemu przedstawiający komponent odpowiedzialny za anotację danych.

Na rysunku 1.11 został przedstawiony fragment architektury systemu z rysunku 1.1, który zostanie omówiony w niniejszym podrozdziale.

1.8.1. Ogólny zarys



Rys. 1.12. Zrzut ekranu pokazujący interfejs anotatora na urządzeniu mobilnym.

Anotator danych został stworzony w celu zebrania jak największej ilości danych dla klasyfikatora przewidującego czy zaznaczony fragment jest godziną rozpoczęcia mszy świętej, czy nie. Żeby osiągnąć zamierzony cel, anotator został zaprojektowany w ten sposób, aby:

- był szybki,
- był dostępny na urządzeniach mobilnych i stacjonarnych,

- był prosty i wygodny w użyciu,
- umożliwiał wykrywanie oszustów (osób intencjonalnie źle anotujących).

Anotator jest dostępny jako aplikacja internetowa pod adresem `msze.nsupdate.info`. Aplikacja jest responsywna, więc można z niej wygodnie korzystać na każdym urządzeniu wyposażonym w co najmniej 4-calowy wyświetlacz. Interfejs jest przejrzysty i został pokazany na rysunku 1.12. Jedyne akcje, jakie może wykonać użytkownik to:

- kliknąć „Tak”, jeśli zaznaczono godzinę rozpoczęcia mszy,
- kliknąć „Nie”, jeśli zaznaczono inną godzinę,
- cofnąć się do poprzedniej anotacji,
- wyświetlić instrukcję obsługi.

Po naciśnięciu przycisku „Tak” lub „Nie” ekran jest automatycznie przewijany na sam dół. Taka operacja zapewnia łatwy dostęp do przycisków odpowiedzialnych za anotację. Dzięki temu znajdują się one również zawsze w tym samym miejscu, co ułatwia szybką anotację. Po naciśnięciu przycisku „Cofnij” ekran nie jest już przewijany na sam dół. W ten sposób zapewniono wygodny dostęp do przycisku „Cofnij”. Jest to szczególnie istotne w przypadku gdy użytkownik zamierza cofać się wiele razy.

Aby zapewnić odpowiednią jakość anotacji, przy pierwszym uruchomieniu wyświetlana jest instrukcja obsługi. Opisuje ona sposób, w jaki należy anotować godziny oraz przedstawia przykłady poprawnie zaanotowanych godzin. Instrukcję można zamknąć dopiero po przewinięciu jej na sam dół.

Aplikacja nie wymaga logowania. Taka decyzja została podjęta ze względu na fakt, że anotatorami są wolontariusze. Wymóg rejestracji i logowania spowodowałby zmniejszenie liczby osób chętnych do anotacji. Takie podejście wiąże się jednak

z problemem identyfikacji użytkowników. Identyfikacja jest niezbędna do prawidłowego funkcjonowania antotora. Chcielibyśmy wiedzieć, które godziny zostały zaanotowane przez danego użytkownika, aby między innymi nie dać mu tych samych godzin do anotacji.

1.8.2. Identyfikacja urządzeń

Skuteczną identyfikację można przeprowadzić, używając ciasteczek oraz pobierając różne informacje o urządzeniu. Za pomocą biblioteki `fingerprintjs2` można między innymi zebrać następujące dane o kliencie[1]:

1. wersję przeglądarki,
2. język,
3. głębię koloru,
4. rozdzielczość ekranu,
5. strefę czasową,
6. obsługę *localStorage*[12],
7. obsługę *sessionStorage*[13],
8. wspieranie *indexed DB*[14],
9. klasę CPU,
10. system operacyjny,
11. listę zainstalowanych czcionek,
12. listę zainstalowanych wtyczek,

[12] Obiekt przechowujący dane w przeglądarce bez daty ważności.
[13] Obiekt przechowujący dane w przeglądarce tylko na czas sesji (po zamknięciu przeglądarki dane są usuwane).
[14] Niskopoziomy interfejs API dla transakcyjnej bazy danych do przechowywania ustrukturyzowanych danych.

13. *Canvas fingerprint*[15],

14. *WebGL fingerprint*[16],

15. *Audio fingerprint*[17],

16. *Pixel ratio*[18],

17. liczbę procesorów logicznych,

18. pojemność pamięci RAM.

Nadawanie identyfikatora nowemu urządzeniu, a potem jego identyfikacja przedstawia się następująco:

1. Klient wysyła żądanie GET w celu załadowania anotatora.

2. Anotator ładuje się w przeglądarce i oblicza *odcisk palca*[19] przeglądarki za pomocą biblioteki *fingerprintjs2*. Obliczony *odcisk palca* będzie dołączany do każdego kolejnego żądania.

3. Klient wysyła serwerowi nowe żądanie z prośbą wyświetlenia fragmentu z godziną do anotacji. Wraz z żądaniem przesyłany jest *odcisk palca* przeglądarki.

4. Serwer odbiera żądanie od klienta i oblicza unikalny identyfikator dla klienta. Serwer zapisuje w bazie danych, że *odciskowi palca* wysłanemu przez klienta odpowiada wygenerowany identyfikator.

[15] Mechanizm do tworzenia odcisków palca przeglądarki na podstawie HTML5 Canvas. HTML5 Canvas to element języka HTML służący do renderowania obrazów bitmapowych.

[16] Mechanizm do tworzenia odcisków palca przeglądarki na podstawie obrazków generowanych przez silnik do rysowania grafiki *WebGL*.

[17] Mechanizm do tworzenia odcisków palca przeglądarki za pomocą analizy sygnału audio.

[18] Jest to stosunek rozdzielczości logicznej do rozdzielczości fizycznej.

[19] Odcisk palca przeglądarki to informacje zebrane w celu jej identyfikacji.

5. Serwer wysyła klientowi fragment do anotacji wraz z wcześniej wygenerowanym identyfikatorem, który zostaje zapisany w nagłówku *Set-Cookie*.

6. Klient odbiera fragment do anotacji. Zostaje on wyświetlony na ekranie urządzenia. Identyfikator zawarty w nagłówku *Set-Cookie* zostaje zapisany w przeglądarce klienta. W tym momencie kończy się ładowanie anotatora.

7. Przy każdym kolejnym żądaniu ciasteczko z identyfikatorem jest wysyłane do serwera. Na jego podstawie serwer identyfikuje dane urządzenie. Jeśli użytkownik usunie ciasteczka z urządzenia i wyśle kolejne zapytanie, to serwer zidentyfikuje użytkownika po *odcisku palca* przeglądarki i w nagłówku *Set-Cookie* prześle pierwotny identyfikator przydzielony danemu urządzeniu.

1.8.3. Architektura anotatora

Architektura anotatora została przedstawiona na rysunku 1.13. Komunikacja między serwerem a klientem przedstawia się następująco:

1. Klient wysyła żądanie do serwera.

2. Serwer *nginx* odbiera żądanie od klienta i przekazuje je serwerowi *Gunicorn*.

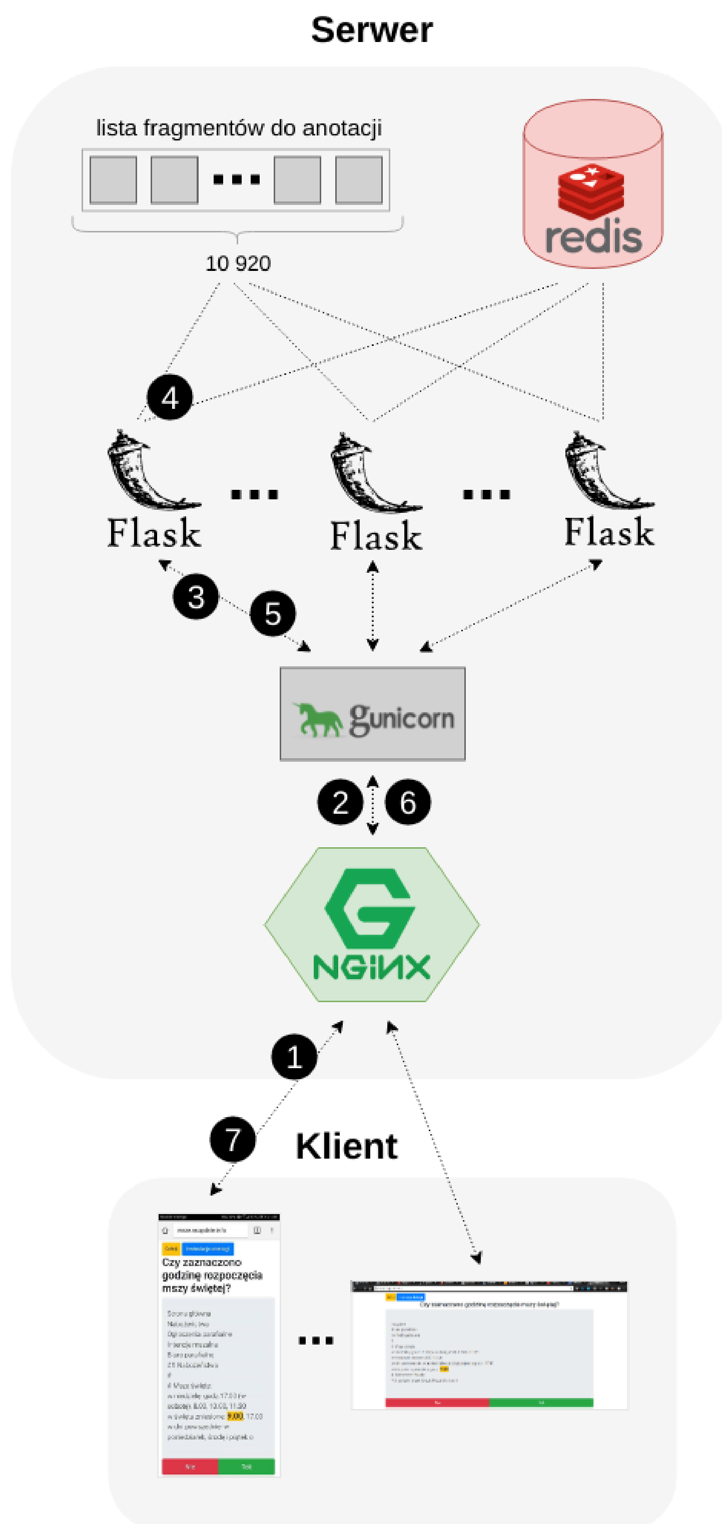
3. Serwer *Gunicorn* przekazuje żądanie wolnemu wątkowi roboczemu. Jeśli każdy wątek roboczy jest zajęty, żądanie zostanie przekazane pierwszemu wolnemu wątkowi roboczemu.

4. Aplikacja webowa *Flask* przetwarza żądanie. W zależności od żądania odpowiednio modyfikuje dane w bazie *Redis*.

5. Aplikacja webowa *Flask* zwraca kolejną godzinę do anotacji z *listy fragmentów do anotacji* i przekazuje ją w postaci odpowiedzi do serwera *Gunicorn*.

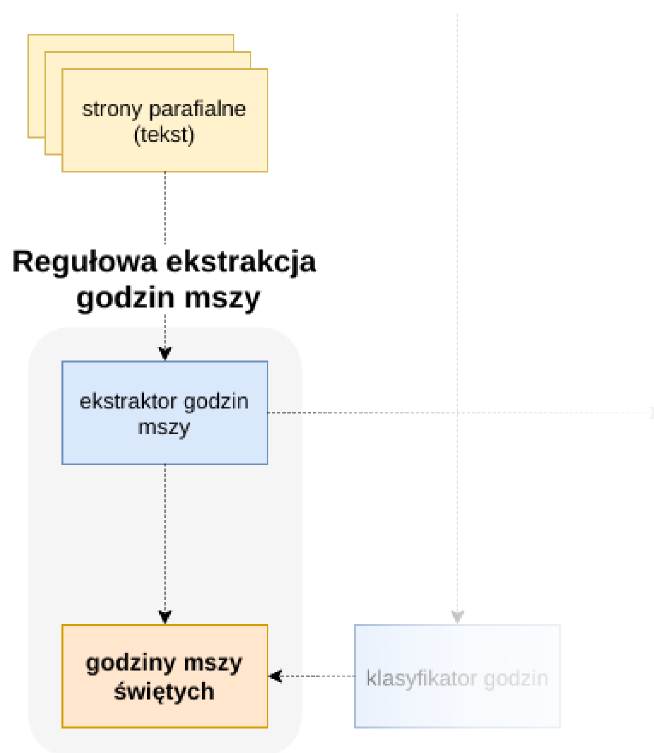
6. Serwer *Gunicorn* odbiera odpowiedź i przekazuje ją serwerowi *nginx*.

7. Serwer *nginx* wysyła odpowiedź klientowi.



Rys. 1.13. Architektura anotatora.

1.9. Regułowa ekstrakcja godzin mszy



Rys. 1.14. Fragment architektury systemu przedstawiający komponent odpowiedzialny za ekstrakcję godzin rozpoczęcia mszy świętych.

Na rysunku 1.14 został przedstawiony fragment architektury systemu z rysunku 1.1, który zostanie omówiony w niniejszym podrozdziale.

Ekstraktor regułowy oparty jest na wyrażeniach regularnych. Wyróżniamy pięć kluczowych fragmentów wyszukiwanych przez ekstraktor godzin. Są to:

główny nagłówek – nagłówek rozpoczynający spis godzin mszy świętych (np. „porządek mszy”) lub „msze święte”. Jest on ekstrahowany przez wyrażenie regularne[20]

```
'porządek mszy (świętych|św|św\.)|
msz[ea][\n]+([śs]wi[eę]t[ea]|św|św\.)'
```

[20]Zakładamy , że wyrażenia regularne ignorują wielkość liter.

nagłówek niedzielny – nagłówek, po którym występują niedzielne i świąteczne godziny mszy (np. „porządek świąteczny”). Jest on ekstrahowany przez wyrażenie regularne `'niedziel[a|e][ \n]+i[ \n]+(dni[ \n]+(świąteczne|św|św\.)|święta)|niedziel[ea]|porządek świąteczny'`

niedzielne godziny – niedzielne i świąteczne godziny mszy świętych. Są one ekstrahowane przez wyrażenie regularne `'.*[^\d]\d{1,2}[^\d].*?''`

nagłówek powszedni – nagłówek, po którym występują godziny mszy świętych dla dni powszednich (np. „w tygodniu”). Jest on ekstrahowany przez wyrażenie regularne `'dzień powszedni|dni powszednie|w tygodniu|porządek zwykły|od poniedziałku do soboty'`

powszednie godziny – powszednie godziny mszy świętych. Są one ekstrahowane przez wyrażenie regularne `'(.*?[^\d\n]?\d{1,2}[^\d\n]?.*?\n)+'`.

W dużym uproszczeniu ekstrakcja godzin przedstawia się następująco:

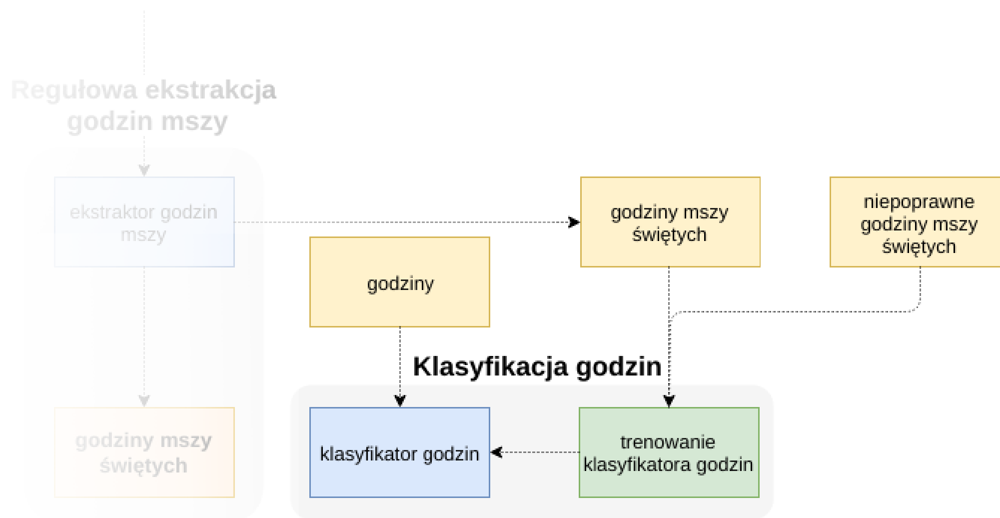
1. Wyszukaj *nagłówek główny*, *nagłówek niedzielny* i *nagłówek powszedni* w taki sposób, aby:
    - *nagłówek niedzielny* był za *nagłówkiem głównym*, ale przed *nagłówkiem powszednim*.
    - między *nagłówkiem głównym* a *nagłówkiem powszednim* nie znajdował się żaden inny *nagłówek* niż *nagłówek niedzielny*

2. Jeśli wyszukiwanie w kroku 2. się nie powiodło, szukaj na kolejnej stronie parafialnej.

3. Niedzielnymi i świątecznymi godzinami mszy świętych będą godziny między *nagłówkiem niedzielnym* a *nagłówkiem powszednim* pasujące do wyrażenia regularnego *niedzielnych godzin*. Jeśli wyszukiwanie się nie powiedzie, to rozpocznij szukanie od początku na kolejnej stronie parafialnej.

4. Godzinami powszednimi, będą godziny po *nagłówku powszednim* pasujące do wyrażenia regularnego *powszednich godzin*. Jeśli wyszukiwanie się nie powiedzie, to rozpocznij szukanie od początku na kolejnej stronie parafialnej.

1.10. Klasyfikacja godzin



Rys. 1.15. Fragment architektury systemu przedstawiający komponent odpowiedzialny za klasyfikację godzin.

Na rysunku 1.15 został przedstawiony fragment architektury systemu z rysunku 1.1, który zostanie omówiony w niniejszym podrozdziale.

Klasyfikator opary jest na płytkich sieciach neuronowych optymalizowanych metodą stochastycznego spadku wzdłuż gradientu z liniowo malejącym współczynnikiem uczenia. Został on wykorzystany do klasyfikacji godzin na te będące godzinami mszy świętych i na te nimi niebędące.

1.10.1. Model teoretyczny

Podrozdział opiera się na artykule *word2vec Parameter Learning Explained*[16] i artykule *Bag of Tricks for Efficient Text Classification*[12].

Notacja

W podrozdziale 1.10.1 przyjęto poniższą notację.

$\boldsymbol{X_{V \times N}}$ – macierz $\boldsymbol{X}$ o $\boldsymbol{V}$ wierszach i $\boldsymbol{N}$ kolumnach.

$\boldsymbol{X_{i,j}}$ – wartość w macierzy $\boldsymbol{X}$ w wierszu $\boldsymbol{i}$ i kolumnie $\boldsymbol{j}$.

$\boldsymbol{X_{i,*}}$ – wiersz $\boldsymbol{i}$ w macierzy $\boldsymbol{X}$.

$\boldsymbol{X_{*,j}}$ – kolumna $\boldsymbol{j}$ w macierzy $\boldsymbol{X}$.

$\boldsymbol{x_i}$ – $\boldsymbol{x_i}$ wartość pod indeksem $\boldsymbol{i}$ w wektorze $\boldsymbol{x}$.

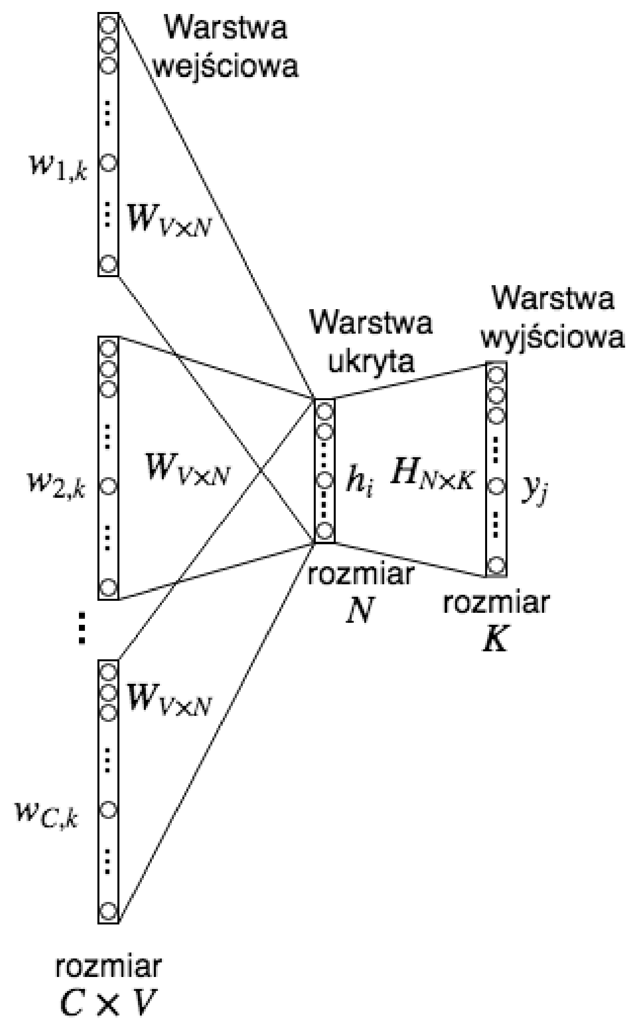
$\boldsymbol{w_i}$ – wektor słowa o indeksie $\boldsymbol{i}$.

$\boldsymbol{w_{i,j}}$ – wartość pod indeksem $\boldsymbol{j}$ w $\boldsymbol{i}$-tym słowie.

Architektura sieci

Rysunek 1.16[21] przedstawia sieć neuronową, która służy do klasyfikacji tekstu. Wejście do sieci składa się z wektorów słów $\boldsymbol{w_1, \ldots, w_C}$, gdzie $\boldsymbol{w_1, \ldots, w_C}$ to

[21] Rysunek wzorowany jest na rysunku z artykułu *word2vec Parameter Learning Explained*[16]



Rys. 1.16. Architektura sieci neuronowej.

słowa z klasyfikowanego zdania. Wektory słów zakodowane zostały za pomocą kodowania *one-hot encoding*[22]. Na wejściu mamy $\boldsymbol{C}$ wektorów słów o rozmiarze $\boldsymbol{V}$, gdzie $\boldsymbol{V}$ to rozmiar słownika, a $\boldsymbol{C}$ to liczba słów w zdaniu. Każdy wektor słów połączony jest z warstwą ukrytą $\boldsymbol{h}$ macierzą wag $\boldsymbol{W_{V \times N}}$ (współdzielą tę samą macierz). Warstwa ukryta $\boldsymbol{h}$ jest rozmiaru $\boldsymbol{N}$. Warstwa ukryta $\boldsymbol{h}$ łączy się z warstwą wyjściową macierzy $\boldsymbol{H_{N \times K}}$. Warstwa wyjściowa $\boldsymbol{y}$ jest rozmiaru $\boldsymbol{K}$, gdzie

[22]Sposób kodowania zwany jako „kod 1 z n”.

$\boldsymbol{K}$ to liczba klas. Sieć klasyfikuje zdania do klas $\boldsymbol{k_1, \ldots, k_K}$.

Propagacja w przód

Warstwa ukryta jest wynikiem średniej z wektorów słów $\boldsymbol{w_1, \ldots, w_C}$[23] pomnożonych przez macierz $\boldsymbol{W_{V \times N}}$ (patrz równanie 1.1).

$$\boldsymbol{h = \frac{1}{c} \sum_{i=1}^{C} w_i W = \frac{1}{c} (\sum_{i}^{C} w_i) \cdot W} \tag{1.1}$$

W warstwie wyjściowej jako funkcja aktywacji została zastosowana funkcja *softmax*[8]. Wejście o indeksie $\boldsymbol{j}$ do funkcji *softmax* obliczane jest za pomocą równania 1.2.

$$\boldsymbol{u_j = h \cdot H_{*,j}} \tag{1.2}$$

$\boldsymbol{y_j}$ w warstwie wyjściowej otrzymujemy używając funkcji *softmax* zgodnie z równaniem 1.3

$$\boldsymbol{y_j = \frac{exp(u_j)}{\sum_{i=1}^{K} exp(u_i)}} \tag{1.3}$$

Funkcja *softmax* normalizuje nam wektor wyjściowy. Dzięki niej na wyjściu z sieci otrzymujemy rozkład prawdopobieństwa klas. Możemy zatem zapisać, że

$$\boldsymbol{p(k_j | w_1, w_2, \ldots, w_C) = y_j,} \tag{1.4}$$

gdzie $\boldsymbol{k_j}$ to klasa o indeksie $\boldsymbol{j}$.

Proces uczenia sieci neuronowej

Proces uczenia zaczynamy od zainicjalizowania macierzy $\boldsymbol{W_{V \times N}}$ i $\boldsymbol{H_{N \times K}}$ losowymi wartościami. Następnie przeprowadzamy propagację w przód i obliczamy błąd

[23]Zakładamy, że wektory są wektorami wierszowymi.

między aktualnym a oczekiwanym wyjściem. Następnie obliczamy gradient funkcji kosztu i poprawiamy wartości macierzy $\boldsymbol{W}$ i $\boldsymbol{H}$ w kierunku gradientu.

Celem jest maksymalizacja prawdopobieństwa $\boldsymbol{p(k_{j^*}|w_1, \ldots w_C)}$ (patrz równania 1.5 - 1.7), gdzie $\boldsymbol{k_{j^*}}$ to wyjściowa klasa o indeksie $\boldsymbol{j}^*$.

$$\boldsymbol{max(p(k_{j^*}|w_1, \ldots w_C)) = max(\frac{exp(u_{j^*})}{\sum_{i=1}^{K} exp(u_i)})} \quad \text{z 1.4 i 1.3} \tag{1.5}$$

$$\boldsymbol{= max(log(\frac{exp(u_{j^*})}{\sum_{i=1}^{K} exp(u_i)}))} \tag{1.6}$$

$$\boldsymbol{= max(u_{j^*} - log(\sum_{i=1}^{K} exp(u_i)))} \tag{1.7}$$

Zatem funkcja kosztu (celem jest jej minimalizacja) będzie przedstawiała się tak jak na równaniu 1.9.

$$\boldsymbol{E = -(u_{j^*} - log(\sum_{i=1}^{K} exp(u_i)))} \quad \text{z 1.7} \tag{1.8}$$

$$\boldsymbol{= log(\sum_{i=1}^{K} exp(u_i)) - u_{j^*}} \tag{1.9}$$

Aktualizowanie wag macierzy $\boldsymbol{H_{N \times K}}$

Najpierw należy policzyć pochodną cząstkową funkcji kosztu $\boldsymbol{E}$ względem wektora $\boldsymbol{u}$ (patrz równania 1.10 - 1.15).

$$\boldsymbol{\frac{\partial E}{\partial u_j} = \frac{\partial(log(\sum_{i=1}^{K} exp(u_i)) - u_{j^*})}{\partial u_j}} \tag{1.10}$$

$$\boldsymbol{= \frac{\partial(log(\sum_{i=1}^{K} exp(u_i)))}{\partial u_j} - \frac{\partial u_{j^*}}{\partial u_j}} \tag{1.11}$$

Z reguły łańcuchowej otrzymujemy:

$$= \frac{\partial(\sum_{i=1}^{K} exp(u_i))}{\partial u_j} \cdot \frac{\partial(log(\sum_{i=1}^{K} exp(u_i)))}{\partial(\sum_{i=1}^{K} exp(u_i))} - \frac{\partial u_{j^*}}{\partial u_j} \tag{1.12}$$

$$= exp(u_j) \cdot \frac{1}{\sum_{i=1}^{K} exp(u_i)} - \frac{\partial u_{j^*}}{\partial u_j} \tag{1.13}$$

$$= y_j - \frac{\partial u_{j^*}}{\partial u_j} \quad \text{z 1.3} \tag{1.14}$$

Za $\frac{\partial u_{j^*}}{\partial u_j}$ podstawiamy t_j:

$$= y_j - t_j := e_j \tag{1.15}$$

gdzie $t_j = 1$, gdy $j = j^*$, w przeciwnym wypadku $t_j = 0$.

Pochodna e_j to błąd predykcji warstwy wyjściowej. Gradient dla wag z macierzy H otrzymamy, licząc pochodną cząstkową funkcji kosztu E względem wag macierzy H (patrz równania 1.16 - 1.19).

$$\frac{\partial E}{\partial H_{i,j}} = \frac{\partial E}{\partial u_j} \cdot \frac{\partial u_j}{\partial H_{i,j}} \quad \text{z reguły łańcuchowej} \tag{1.16}$$

$$= e_j \cdot \frac{\partial(h \cdot H_{*,j})}{\partial H_{i,j}} \quad \text{z 1.2} \tag{1.17}$$

$$= e_j \cdot \frac{\partial(\sum_{i'=1}^{N} h_{i'} \cdot H_{i',j})}{\partial H_{i,j}} \quad \text{z definicji mnożenia macierzy} \tag{1.18}$$

$$= e_j \cdot h_i \tag{1.19}$$

Aktualizowanie wag macierzy H zostało przedstawione w równaniu 1.20.

$$H_{i,j}^{nowe} = H_{i,j}^{stare} - \eta \cdot e_j \cdot h_i \tag{1.20}$$

gdzie η to liniowo malejący współczynnik uczenia.

Aktualizowanie wag macierzy $\boldsymbol{W_{V\times N}}$

Najpierw należy policzyć pochodną cząstkową funkcji kosztu $\boldsymbol{E}$ względem warstwy ukrytej $\boldsymbol{h}$ (patrz równania 1.21 - 1.24).

$$\frac{\partial E}{\partial h_i} = \sum_{j=1}^{K} \frac{\partial E}{\partial u_j} \cdot \frac{\partial u_j}{\partial h_i} \quad \text{z reguły łańcuchowej} \tag{1.21}$$

$$= \sum_{j=1}^{K} e_j \cdot \frac{\partial (h \cdot H_{*,j})}{\partial h_i} \quad \text{z 1.2 i z 1.10 - 1.15} \tag{1.22}$$

$$= \sum_{j=1}^{K} e_j \cdot \frac{\partial (\sum_{i'=1}^{N} h_i' \cdot H_{i',j})}{\partial h_i} \quad \text{z definicji mnożenia macierzy} \tag{1.23}$$

$$= \sum_{j=1}^{K} e_j \cdot H_{i,j} \tag{1.24}$$

W równaniu 1.21 mamy do czynienia z sumą ze względu na fakt, że neuron $\boldsymbol{h_i}$ połączony jest z $\boldsymbol{K}$ neuronami warstwy wyjściowej. Każdy błąd predykcji powinien być uwzględniony.

Następnie należy policzyć pochodną cząstkową funkcji $\boldsymbol{E}$ względem wag w macierzy $\boldsymbol{W}$ (patrz 1.25 - 1.30).

$$\tag{1.25}$$

Z reguły łańcuchowej:

$$\frac{\partial E}{\partial W_{k,i}} = \frac{\partial E}{\partial h_i} \cdot \frac{\partial h_i}{\partial W_{k,i}} \tag{1.26}$$

Z 1.21 - 1.24 i z 1.1:

$$= (\sum_{j=1}^{K} e_j \cdot H_{i,j}) \cdot \frac{\partial (\frac{1}{C}(\sum_{i'=1}^{C} w_{i'}) W_{*,i})}{\partial W_{k,i}} \tag{1.27}$$

Z definicji mnożenia macierzy:

$$= (\sum_{j=1}^{K} \boldsymbol{e_j} \cdot \boldsymbol{H_{i,j}}) \cdot \frac{\partial(\frac{1}{C} \sum_{l=1}^{V}((\sum_{i'=1}^{C} \boldsymbol{w_{i',l}})\boldsymbol{W_{l,i}}))}{\partial \boldsymbol{W_{k,i}}} \tag{1.28}$$

$$= (\sum_{j=1}^{K} \boldsymbol{e_j} \cdot \boldsymbol{H_{i,j}}) \cdot \frac{1}{C} \sum_{i'=1}^{C} \boldsymbol{w_{i',k}} \tag{1.29}$$

Zauważmy, że $\sum_{i'=1}^{C} \boldsymbol{w_{i',k}} = \mathbf{1}$, bo wektory słów zakodowane są kodowaniem *one-hot encoding*. Innymi słowy jest tylko jeden wektor słowa, który ma wartość 1 pod indeksem $\boldsymbol{k}$. Reszta wektorów słów ma pod indeksem $\boldsymbol{k}$ wartość 0. Zatem otrzymujemy:

$$= \frac{1}{C} \sum_{j=1}^{K} \boldsymbol{e_j} \cdot \boldsymbol{H_{i,j}} \tag{1.30}$$

Aktualizowanie wag macierzy $\boldsymbol{W}$ zostało przedstawione w równaniu 1.31.

$$\boldsymbol{W_{i,j}^{nowe}} = \boldsymbol{W_{i,j}^{stare}} - \boldsymbol{\eta} \frac{1}{C} \sum_{j=1}^{K} \boldsymbol{e_j} \cdot \boldsymbol{H_{i,j}} \tag{1.31}$$

1.10.2. FastText

Autorzy biblioteki *fastText* wskazują, że podstawowy model opisany w podrozdziale 1.10.1 można usprawnić. Do modelu można dodać dodatkowe cechy w postaci *n-gramów*(w modelu z podrozdziału 1.10.1 zostały zastosowane unigramy). Artykuł *Bag of Tricks for Efficient Text Classification*[12] pokazuje, że dodanie bigramów poprawia wyniki klasyfikatora. Ponadto, aby przyspieszyć trenowanie, można zastosować funkcję *hierarchical softmax*[12], zamiast funkcji *softmax*. Taka operacja jest korzystna w przypadku dużej liczby klas [12]. W przypadku klasyfikacji godzin użyto funkcji *softmax* ze względu na małą liczbę klas. Do klasyfikacji użyto biblioteki *fastText* z domyślnymi parametrami. Rezultaty zostały przedstawione w rozdziale 2.

Rozdział 2

Rezultaty

W niniejszym rozdziale zebrane zostały wyniki, jakie osiągnął system ekstrakcji informacji o godzinach rozpoczęcia mszy świętych.

Dane parafii i ich adresy URL

Udało się zebrać:

- **10130** nazw i adresów parafii,
- **5600** adresów internetowych stron parafialnych,

Ręcznie na małej próbce adresów URL stwierdzono, że w ponad 90% prowadzą one do poprawnych parafii.

Warto zaznaczyć, że w momencie oddania pracy do druku nie znaleziono obszerniejszego zbioru adresów URL parafii. Serwisy internetowe zawierające adresy URL parafii posiadały nie więcej niż 2500 tysiąca adresów URL parafii. Większość z nich to były błędne adresy internetowe.

Pająk

Z **5600** adresów internetowych parafii pająk zaindeksował **5177** stron parafialnych. W kilka dni pobrał około **3 000 000** stron HTML o łącznym rozmiarze **152G**. Po konwersji stron parafialnych z formatu HTML do formatu tekstowego otrzymano **22G** tekstu.

Anotator

W dwa tygodnie od udostępnienia anotatora udało się:

- zebrać **10260** anotacji godzin,
- zaanotować **9313**[1] z **10920** godzin.

Dane anotowano z **177** unikalnych urządzeń. Średni czas anotacji na urządzenie wyniósł **2,5** sekundy.

Liczba urządzeń, z których anotowano godziny mszy świętych świadczy o zainteresowaniu społeczności katolickiej projektem automatycznego ekstraktora godzin mszy świętych. Entuzjazm udzielał się również w komentarzach na Facebook'u (komentarze pod postem z prośbą o anotowanie godzin mszy świętych). Średni czas anotacji, jak i liczba zaangażowanych użytkowników potwierdzają cechy anotatora wymienione w podrozdziale 1.8.1 (szybkość, dostępność i wygoda użytkowania).

Regułowy ekstraktor danych

Regułowy ekstraktor danych był w stanie znaleźć godziny mszy świętych dla około **2600** parafii z **5177** parafii. Na małej próbce parafii ręcznie stwierdzono bardzo wysoką precyzję ekstrahowanych danych.

[1] Dopuszczane jest by jedna godzina była anotowana przez wielu unikalnych użytkowników. Stąd rozbieżność między liczbą anotacji a zaanotowanymi godzinami.

Klasyfikator godzin

Do ewaluacji ekstraktora godzin wykorzystano dane zebrane przez anotator. Z 9313 zaanotowanych godzin 80% użyto do trenowania klasyfikatora, a 20% do ewaluacji. Nie przeprowadzono optymalizacji parametrów.

Otrzymano następujące wyniki:

- wartość pokrycia $= \mathbf{0,884}$
- precyzja $= \mathbf{0,850}$
- F1[2] $= \mathbf{0,884}$
- dokładność $= \mathbf{0,858}$

Wyniki są obiecujące i z pewnością można je w przyszłości znacznie poprawić.

Rozdział 3

Podsumowanie

W pracy opisany został zrobiony przeze mnie system do ekstrakcji informacji godzin rozpoczęcia mszy świętych. Na początku przedstawiłem komponent odpowiedzialny za zbieranie informacji o parafiach. Następnie przybliżyłem metodę wyszukiwania adresów internetowych parafii. Potem dokładnie omówiłem architekturę pająka do pobierania stron parafialnych. W kolejnych rozdziałach krótko opisałem bibliotekę do konwersji tekstu, ekstraktor godzin oraz wyszukiwanie stron, na których z dużym prawdopodobieństwem znajdują się godziny mszy. Następnie omówiłem anotator służący do zbierania danych do uczenia maszynowego. Anotator dostępny jest pod adresem `msze.nsupdate.info`. Potem krótko opowiedziałem o regułowym ekstraktorze danych. W kolejnym rozdziale przedstawiłem model teoretyczny klasyfikatora godzin mszy. W szczególności wyprowadziłem równania na aktualizacje wag sieci neuronowej. Na końcu przedstawiłem informacje o wszystkich zebranych danych oraz wynikach anotatora, ekstraktora godzin mszy świętych i klasyfikatora godzin. Otrzymane rezultaty świadczą o sensowności projektu i zachęcają do jego dalszego rozwoju.

Spis rysunków

1.1. Architektura systemu do ekstrakcji godzin mszy świętych. 17
1.2. Fragment architektury systemu przedstawiający komponent odpowiedzialny za zbieranie informacji o parafiach. 18
1.3. Fragment architektury systemu przedstawiający komponent odpowiedzialny za wyszukiwanie adresów URL parafii. 20
1.4. Przykład dwóch obiektów zwróconych przez wyszukiwarkę Google, które mają ten sam adres internetowy. 22
1.5. Fragment architektury systemu przedstawiający komponent odpowiedzialny za indeksowanie stron parafialnych. 22
1.6. Silnik kontrolujący przepływ danych przez komponenty pająka. . . 24
1.7. 100 pająków pracujących jednocześnie. 30
1.8. Fragment architektury systemu przedstawiający komponent odpowiedzialny za konwersje HTML na tekst. 32
1.9. Fragment architektury systemu przedstawiający komponent odpowiedzialny za ekstrakcję godzin ze stron parafialnych. 33
1.10. Fragment architektury systemu przedstawiający komponent odpowiedzialny za filtrowanie stron parafialnych. 34
1.11. Fragment architektury systemu przedstawiający komponent odpowiedzialny za anotację danych. 36
1.12. Zrzut ekranu pokazujący interfejs anotatora na urządzeniu mobilnym. 37
1.13. Architektura anotatora. 42

1.14. Fragment architektury systemu przedstawiający komponent odpowiedzialny za ekstrakcję godzin rozpoczęcia mszy świętych. 43
1.15. Fragment architektury systemu przedstawiający komponent odpowiedzialny za klasyfikację godzin. 45
1.16. Architektura sieci neuronowej. 47

List of Algorithms

1. *Expotential Backoff* 19
2. Rozpoznawanie plików binarnych 26
3. Algorytm regulacji częstości zapytań 29
4. Rozpoznawanie stron z godzinami mszy świętych. 35

Spis tabel

1.1. Fragment danych zebranych przez pająka nazw i adresów parafii. . 19

Bibliografia

[1] Dokumentacja fingerprintjs2. https://github.com/Valve/fingerprintjs2/blob/master/README.md. [Online; dostęp 22.06.2018].

[2] Dokumentacja formatu JSON Lines. jsonlines.org. [Online; dostęp 22.06.2018].

[3] Google Place Details. https://developers.google.com/places/web-service/details. [Online; dostęp 22.06.2018].

[4] Google Place Search. https://developers.google.com/places/web-service/search. [Online; dostęp 22.06.2018].

[5] Google Places API. https://developers.google.com/places/web-service/intro. [Online; dostęp 22.06.2018].

[6] Repozytorium html2text. https://github.com/Alir3z4/html2text. [Online; dostęp 22.06.2018].

[7] Exponential backoff — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Exponential_backoff&oldid=830252246, 2018. [Online; dostęp 22.06.2018].

[8] Softmax function — Wikipedia, the free encyclopedia. https://en.wikipedia.org/w/index.php?title=Softmax_function&oldid=843761006, 2018. [Online; dostęp 22.06.2018].

[9] Strona internetowa BeautifulSoup. https://www.crummy.com/software/BeautifulSoup/.

[10] Fielding, R., and Reschke, J. Hypertext transfer protocol (http/1.1): Semantics and content. RFC 7231, RFC Editor, June 2014. http://www.rfc-editor.org/rfc/rfc7231.txt.

[11] Freed, N., and Borenstein, N. S. Multipurpose internet mail extensions (mime) part one: Format of internet message bodies. RFC 2045, RFC Editor, November 1996. http://www.rfc-editor.org/rfc/rfc2045.txt.

[12] Joulin, A., Grave, E., Bojanowski, P., and Mikolov, T. Bag of tricks for efficient text classification, 2016.

[13] Leonard, S. The text/markdown media type. RFC 7763, RFC Editor, March 2016.

[14] Myers, D., and McGuffee, J. W. Choosing scrapy. *J. Comput. Sci. Coll. 31*, 1 (Oct. 2015), 83–89.

[15] Reschke, J. Use of the content-disposition header field in the hypertext transfer protocol (http). RFC 6266, RFC Editor, June 2011.

[16] Rong, X. word2vec parameter learning explained. *CoRR abs/1411.2738* (2014).

[17] Rozkrut, D. Rocznik statystyczny rzeczypospolitej polskiej 2017. *GUS T. LXXVII* (2017), 194–195.