Politechnika Wrocławska
Wydział Informatyki i Telekomunikacji

Zastosowania informatyki w gospodarce

System śledzenia rozpływu informacji

Autorzy:
♚Bartosz Powęska, 234720
Justyna Skalska, 225942
Dawid Kondziela, 249440
Paweł Fedorczak, 243751

Grupa A, wtorek, 11^{15}

Dr inż. Tomasz Walkowiak

Semestr letni 2022

Spis treści

1 Analizowane tematy

- wiadomości ze świata gier
  - GRYOnline - Twitter
  - Eurogamer - Twitter
  - CDAction Twitter
  - PPE Twitter

2 Technologia realizacji

- Pobieranie danych - pyTwitter, scrapy
- Przetwarzanie danych -
- Wyświetlanie web - java + spring boot
- Docker Compose
- Baza danych - PostgreSQL

3 Harmonogram realizacji

- I kamień milowy (12.04.22)
  - wybranie technologii,
  - schemat architektury systemu,
  - schemat bazy danych,
  - podział zadań w grupie.
- II kamień milowy (17.05.22)
  - stworzenie bazy danych,
  - wstępna wizualizacja danych,
  - implementacja pobierania danych dla wiadomości pierwszej kategorii.
- III kamień milowy (07.06.22)
  - implementacja pobierania danych dla wiadomości drugiej kategorii,
  - pełna implementacja wizualizacji danych,
  - prezentacja gotowej aplikacji.

4 Pobieranie danych

Dane wykorzystane do budowy grafu rozpływu informacji zostały zdobyte na dwa różne sposoby:

- pobrane z Twittera
- pobrane ze stron internetowych

Różnice pomiędzy tymi źródłami danych przejawiają się w wykorzystywanej do pozyskania danych bibliotece oraz w jakości i ilości danych. Wiele postów z Twittera zawiera jedynie link do artykuły ale pozostałe dostarczają wiele dodatkowych informacji takich jak: wspomniane osoby, miejsca, dokładny czas publikacji itd. W przypadku danych ze stron internetowych zdobycie danych jest trudniejsze, a nawet czasami niemożliwe, czas publikacji często ogranicza się tylko do daty ale za to jest pewne, że dane będą długim tekstem ciągłym.

4.1 Twitter

Do pobierania danych z Twittera wykorzystana została biblioteka języka python "pytwitter", która ułatwia dostęp do API Twittera. Wymagane również było konto deweloperskie, które za pomocą specjalnego tokena pozwało pobierać dane. Dane pobierane był przez REST "get_timeline", który pozwalał na pobranie przy jednym żądaniu do 100 postów. Konto deweloperskie pozwala na dostęp do ostatnich 3000 postów danego konta więc niezbędny okazał się mechanizm paginacji, zagłębiający się w coraz starsze posty użytkownika. Przykład pobierania wraz z paginacją jest widoczny w 5. Wszystkie dane zostały następnie zapisane w oddzielnych dla każdego portalu plikach.

Listing 1: Pobieranie danych z Twittera

```
while len(current_tweets_data) <= max_tweets:
    current_tweets = self.api.get_timelines(user_id, **query)
    print('Pobrano {} tweetow z {}'.format(
        current_tweets['meta']['result_count'], page))
    if 'data' in list(current_tweets.keys()):
        current_tweets_data.extend(current_tweets['data'])
    if 'next_token' not in current_tweets['meta'].keys():
        break
    query.update({'pagination_token': current_tweets['meta']['next_token']})
```

4.2 Strony internetowe

Zbieranie danych ze stron internetowych odbywało się poprzez crawler'y, które analizowały stronę przez plik HTML. Do napisania crawler'ów służyła biblioteka scrapy, dzięki której pisało się Spider'y. Spider ma wpisane na jakie elementy składni HTML powinien uważać i je zapisywać. Do poruszania się po tych plikach wykorzystywane są klasy CSS lub ścieżka XPATH. Przykładowy Spider widoczny jest w 2.

Listing 2: Spider skonfigurowany pod stronę Gry-Online.pl

```
class GryOnline(scrapy.Spider):
    name = 'gryonline'

    start_urls = [
        'https://www.gry-online.pl/newsroom/news/',
    ]

    def parse(self, response, **kwargs):
        news_page_links = response.xpath(
                              "//div[@class='lista lista-news']/div/a")
        yield from response.follow_all(news_page_links, self.parse_news)

        next_page_link = response.xpath("//div[@class='np-right']/a")
        yield from response.follow_all(next_page_link, self.parse)

    def parse_news(self, response):
        yield {
            'title': response.css('h1::text').get(),
            'date': response.css('span.a-d-data::text').get(),
            'text': response.css('article p::text').getall(),
        }
```

5 Ujednolicenie daty

W celu porównywania danych z różnych stron internetowych, niezbędnym było przeprowadzenie procesu unifikacji daty. W tym celu zostały napisane 2 skrypty w języku python, których zdaniem jest edycja

powstałych plików json w taki sposób, aby data każdego artykułu miała postać yyyy-mm-ddTHH:MM

- Skrypt odpowiadający za artykuły ze strony GryOnline

```
import json
import os
import time
import datetime
with open('gryonline.json', 'r+', encoding="utf8") as a_file:
    json_object = json.load(a_file)
miesiace = ['stycznia', 'lutego', 'marca', 'kwietnia', 'maja',
    'czerwca', 'lipca', 'sierpnia', 'wrze Żnia', 'pa dziernika',
    'listopada', 'grudnia']
lista = []
for element in json_object:
    if element['date'] != None:
        data = element['date'].split("␣")
        index = miesiace.index(data[1])+1
        godzina = data[-1].split(":")
        a=datetime.datetime(int(data[2][:-1]),
            int(index), int(data[0]), int(godzina[0]), int(godzina[1]))
        print(a)
        element['date'] = a.strftime("%Y-%m-%dT%H:%M")
        lista.append(element)
with open('fileGO.json', 'w', encoding='utf8') as f:
    f.write(json.dumps(lista, ensure_ascii=False))
```

- Skrypt odpowiadający za artykuły ze strony PPE

```
import json
import os
import time
import datetime
with open('PPE.json', 'r+', encoding="utf8") as a_file:
    json_object = json.load(a_file)

lista = []
for element in json_object:
    if element['date'] != None:
        data = element['date'].split("␣")
        if data[1] != 'Dzisiaj,' and data[1] !='Wczoraj,':
            godzina = data[-1].split(":")
            dzien = data[1].split(".")
            a=datetime.datetime(2022, int(dzien[1][:-1]),
                int(dzien[0]), int(godzina[0]), int(godzina[1]))
            print(a)
            element['date'] = a.strftime("%Y-%m-%dT%H:%M")
            lista.append(element)
with open('filePPE.json', 'w', encoding='utf8') as f:
   f.write(json.dumps(lista, ensure_ascii=False))
```

6 Clarin API

Do ustalenia podobieństwa między artykułami wykorzystujemy zapytanie wzorownae na usłudze websty clarina.

Listing 3: Pobieranie danych z Twittera

```
any2txt
|wcrft2
|fextor2({
    "features":"base␣noun_count␣bigrams",
    "base_modification":"startlist",
    "orth_modification":"startlist",
    "lang":"pl",
    "filters":{"base":[{"type":"lemma_stoplist",
        "args":{"stoplist":"@resources/fextor/ml/polish_base_startlist.txt"}}]}})
|dir
|featfilt2({
    "weighting":"all:sm-mi_simple",
    "filter":"min_tf-2␣min_df-2",
    "similarity":"cosine"})
|cluto({"no_clusters":2,"analysis_type":"plottree"})
```

następnie w dalszych krokach wykorzystujemy zwrócony plik similarity.json zawierający macierz podobieństwa plików.

7 Wyświetlanie danych

Wyświetlanie i przetwarzanie danych zostało zaimplementowane przy wykorzystaniu języka Java oraz frameworka Spring Boot. Jako silnik szablonów użyliśmy Thymeleaf, a do generowania grafów posłużyła nam JavaScriptowa biblioteka vis.js.

Użytkownik serwisu jest w stanie wrzucić wygenerowany we wcześniejszych krokach plik ZIP zawierający dane zwrócone przez system Clarin. Przesłany plik jest następnie otwierany i wyciągane są z niego dane potrzebne do narysowaniu grafu. Klasa przedstawiona na listingu numer 4 obrazuje strukturę wykorzystywaną do wygenerowania drzewa.

Listing 4: Klasa DataSet przechowująca dane grafu

```
@AllArgsConstructor
@Getter
@Setter
public class DataSet {
    private List<Node> nodes;
    private List<Edge> edges;

    @AllArgsConstructor
    @Getter
    @Setter
    public static class Node {
        private Integer id;
        private String label;
        private String title;
    }

    @AllArgsConstructor
    @Getter
    @Setter
    public static class Edge {
        private Integer from;
        private Integer to;
    }
}
```

Listing numer 5 opisuje przetwarzanie danych znajdujących się w pliku similarity.json. Na bazie zwróconych przez nie danych budowany jest graf. Podczas tego procesu filtrowane są wyniki ze zbyt małym "podobieństwem", ponieważ generowałyby one zbyt dużą liczbę połączeń w drzewie. Współczynnik został przez nas ustawiony metodą prób i błędów.

Listing 5: Przetwarzanie danych z pliku similarity.json

```
    private DataSet similarityJsonToDataSet(final SimilarityJson similarity) {
        final List<DataSet.Edge> edges = new ArrayList<>();
        final List<DataSet.Node> nodes = new ArrayList<>();

        final ArrayList<String> rowlabels = similarity.getRowlabels();
        for (int i = 0; i < rowlabels.size(); i++) {
            final String label = rowlabels.get(i);
            nodes.add(new DataSet.Node(i, "", label));
        }

        final ArrayList<ArrayList<Float>> arrays = similarity.getArr();
        for (int i = 0; i < arrays.size(); i++) {
            for (int j = i + 1; j < arrays.get(i).size(); j++) {
                if (arrays.get(i).get(j) > 0.33) {
                    edges.add(new DataSet.Edge(i, j));
                }
            }
        }

        return new DataSet(nodes, edges);
    }
```

8 Docker

Do ułatwienia odpalania projektów Pythonowych użyliśmy Dockera oraz Docker Compose. Dzięki temu wszelkie zależności instalowane są automatycznie. Aplikacja Springowa odpala się na porcie 8080 i współdzieli przestrzeń dyskową wraz z każdym z projektów, co pozwala na szybki dostęp do wygenerowanych w poszczególnych krokach plików.

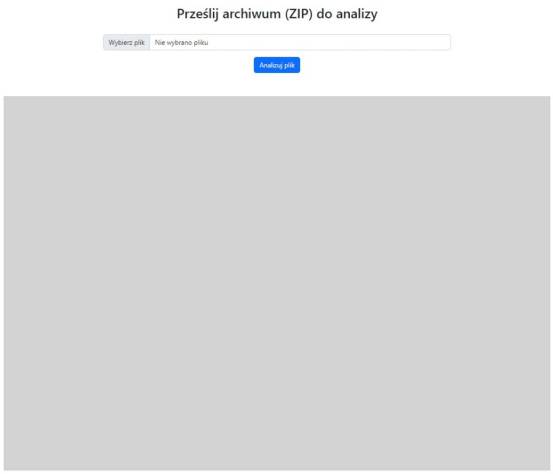
Listing 6: Docker Compose

```
version: "3.9"

services:
  dataflow:
    build: ./DataGathering/Dataflow
    volumes:
      - static-content:/usr/src/app/data
  twitterdata:
    build: ./DataGathering/TwitterData
    volumes:
      - static-content:/usr/src/app/data
  zipanator:
    build: ./DataGathering/ZipWithData
    volumes:
      - static-content:/usr/src/app/data
  lpmnclient:
    build: ./LpmnClient
    volumes:
```

```
      - static-content:/usr/src/app/data
  displaydata:
    build: ./DisplayData
    ports:
      - "8080:8080"
    volumes:
      - static-content:/usr/src/app/data

volumes:
  static-content:
    driver: local
    driver_opts:
      type: 'none'
      o: 'bind'
      device: '/mnt/c/Users/jskalska/Projects/personal/PWr/ZIwG/data'
```

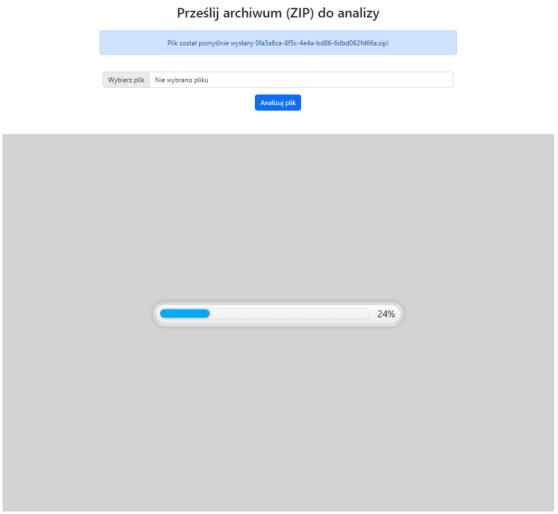
9 Strona prezentacyjna

Użytkownika przywita ekran pozwalający na przesłanie pliku zawierającego przetworzone dane output z api clarina) widoczny w 1.



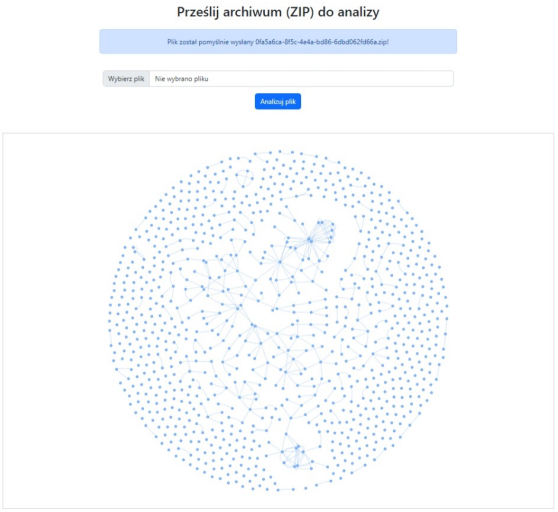
Rysunek 1: Web page w stanie przed załadowaniem danych.

Po załadowaniu pliku pojawi się pasek ładowania widoczny na grafice 2.



Rysunek 2: PWeb page podczas rysowania grafu.

Po przeanalizowaniu danych narysowany zostanie graf zależności widoczny na grafice 3, który można przybliżać i oddalać wedle potrzeb. Węzły reprezentują poszczególne artykułu, a krawędzie poprowadzone między nimi ich podobieństwo do siebie.



Rysunek 3: Web page po załadowaniu grafu.