

Narzędzie do tworzenia podsumowań tekstu

Mikołaj Petecki, Jan Rejowski, Michał Riabcew

1. Abstrakt

Stworzenie narzędzia do generowania podsumowań tekstu to jak próba ułatwienia życia w gąszczu informacyjnego chaosu. W dzisiejszych czasach jesteśmy nieustannie bombardowani nowymi informacjami, kolejne artykuły lądują w naszych rękach w zasadzie co chwile. Generator podsumowań pomaga wydobyć sedno treści, usuwając zbędny balast, abyśmy mogli szybko dotrzeć do najważniejszych informacji.

2. Wstęp

2.1 Cel pracy

Celem pracy jest stworzenie aplikacji internetowej o prostym i przejrzystym interfejsie służącej do automatycznego generowania zwięzłego podsumowania tekstu wprowadzonego przez użytkownika. Aplikacja osiągać ten cel będzie poprzez eliminację zbędnych informacji oraz identyfikację kluczowych treści w danym tekście.

2.2 Zakres pracy

Stworzenie narzędzia z wykorzystaniem języka Python i jego bibliotek służącego do analizy semantycznej struktury zdania, oceniania podobieństwa między zdaniami oraz określania ważności poszczególnych fragmentów tekstu. Dodatkowo w naszej pracy stworzymy prosty interfejs za pomocą biblioteki React i języka JavaScript.

2.3 Metodyka

Praca będzie składać się z kilku ważnych elementów:

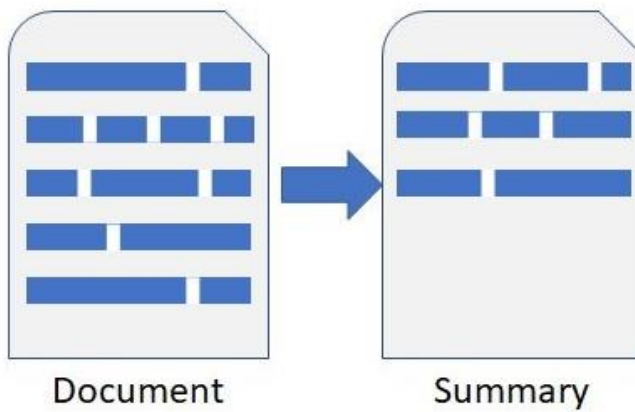
1. Analiza zdaniowa – program analizuje strukturę każdego zdania, ignorując słowa stopu oraz uwzględniając znaczenie słów
2. Ocena podobieństwa zdań – wykorzystując podobieństwo cosinusowe program określać będzie jak bardzo powiązane są ze sobą poszczególne zdania
3. Tworzenie macierzy podobieństwa – to pomoże odzwierciedlić stopień relacji między zdaniami
4. Wykorzystanie algorytmu PageRank – posłuży on do oceny ważności każdego ze zdań w kontekście całego tekstu
5. Sortowanie i wybór najlepszych fragmentów
6. Zwrócenie podsumowania

3. Teoria

Rozważając zagadnienie tworzenia narzędzi do podsumowywania tekstu możemy wyróżnić dwa główne podejścia: podsumowywanie ekstraktywne oraz abstrakcyjne

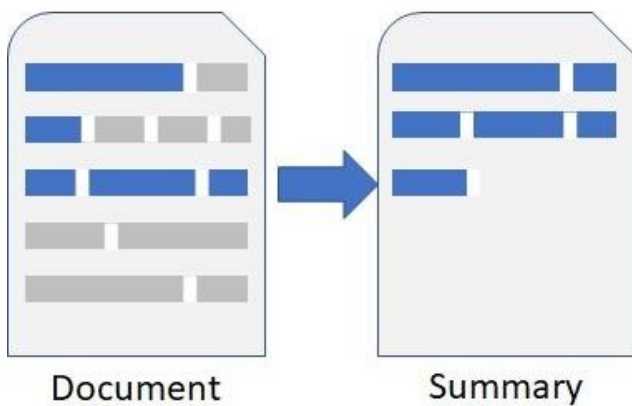
Podsumowywanie abstrakcyjne - w tej metodzie model tworzy własne frazy i zdania, w podobny sposób jak robiłby to człowiek - wyciągając sens tekstu a następnie opisując go "własnymi słowami". Ten sposób jest zdecydowanie bardziej atrakcyjny, ale jest też znacznie trudniejszy do zaimplementowania

Abstractive Summarization



Podsumowywanie ekstrakcyjne - polega na wybieraniu z tekstu takiego zbioru zdań, które zachowują najważniejsze informacje. Na podstawie konkretnego algorytmu (na przykład algorytmu pageRank) szereguje się zdania według ich ważności, a następnie najistotniejsze informacje łączy się w podsumowanie. W naszym projekcie skupimy się właśnie na tej metodzie.

Extractive Summarization



4. Praktyka

4.1 Backend

Serwer z API został wystawiony we Flasku - frameworku Pythona. Umożliwia on efektywne podsumowywanie tekstów.

Wykorzystane biblioteki:

- nltk
- networkx
- Flask
- math
- re
- numpy

4.1.1 Endpoint POST /text-to-summarize

Żądanie to przyjmuje jako parametr obiekt JSON z następującymi polami:

- *text*: treść artykułu bądź innego tekstu, który chcemy streścić
- *percent*: procent o jaki chcemy skrócić tekst

Odpowiedź jest również zwracana jako obiekt JSON, zawierający pola:

- *summarize*: streszczony tekst
- *words*: liczba słów w streszczonym tekście
- *sentences*: liczba zdań w streszczonym tekście
- *readingTime*: szacowany czas czytania w minutach

4.1.2 Klasa TextSummarizer

Klasa ta zawiera metody niezbędne do generowania streszczeń tekstów

- `__init__(self)` - metoda inicjalizująca obiekt TextSummarizer, pobiera listę angielskich stopwords z biblioteki nltk

```
def __init__(self):  
    nltk.download("stopwords")
```

- `read_article(self, article)` - metoda ta rozdziela tekst na zdania, korzystając z wyrażeń regularnych, aby prawidłowo zidentyfikować zdania w tekście.

```
def read_article(self, article):  
    article_text = re.split(r'(?![\w\.\w.])(<![A-Z][a-z]\.)(?<=\.|\?|\s)', article)  
    sentences = []  
  
    for sentence in article_text:  
        sentences.append(sentence.replace("[^a-zA-Z]", " ").split(" "))  
    sentences.pop()  
  
    return sentences
```

- `sentence_similarity(self, sent1, sent2, stopwords=None)` - oblicza podobieństwo między dwoma zdaniami. Konwertuje słowa na małe litery, a następnie tworzy dla nich wektory, reprezentujące ich zawartość. Następnie oblicza odległość kosinusową między nimi, co daje miarę podobieństwa zdań.

```
def sentence_similarity(self, sent1, sent2, stopwords=None):  
    if stopwords is None:  
        stopwords = []  
  
    sent1 = [w.lower() for w in sent1]  
    sent2 = [w.lower() for w in sent2]  
  
    all_words = list(set(sent1 + sent2))  
  
    vector1 = [0] * len(all_words)  
    vector2 = [0] * len(all_words)  
  
    # build the vector for the first sentence  
    for w in sent1:  
        if w in stopwords:  
            continue  
        vector1[all_words.index(w)] += 1  
  
    # build the vector for the second sentence  
    for w in sent2:  
        if w in stopwords:  
            continue  
        vector2[all_words.index(w)] += 1  
  
    return 1 - cosine_distance(vector1, vector2)
```

- `build_similarity_matrix(self, sentences, stop_words)` - metoda, która tworzy macierz podobieństwa dla listy zdań.

```
def build_similarity_matrix(self, sentences, stop_words):  
    # Create an empty similarity matrix  
    similarity_matrix = np.zeros((len(sentences), len(sentences)))  
  
    for idx1 in range(len(sentences)):  
        for idx2 in range(len(sentences)):  
            if idx1 == idx2: #ignore if both are same sentences  
                continue  
            similarity_matrix[idx1][idx2] = self.sentence_similarity(sentences[idx1], sentences[idx2], stop_words)  
  
    return similarity_matrix
```

- `generate_summary(self, article, percent)` - główna funkcja, przyjmuje tekst do streszczenia, następnie wykorzystując poprzednie metody, dzieli tekst na zdania i tworzy z nich macierz podobieństwa. Wykorzystuje algorytm PageRank do oceny ważności każdego zdania w kontekście całego tekstu, a następnie wybiera najlepsze. Na końcu zwraca streszczony tekst o podany procent, liczbę słów, zdań i szacowany czas czytania.

```
def generate_summary(self, article, percent):
    stop_words = stopwords.words('english')
    summarize_text = []

    # Step 1 - Read text and split it
    sentences = self.read_article(article)
    top_n = int(floor(len(sentences) * int(100 - percent) * 0.01))
    if top_n == 0:
        top_n = 1

    # Step 2 - Generate Similarity Matrix across sentences
    sentence_similarity_matrix = self.build_similarity_matrix(sentences, stop_words)

    # Step 3 - Rank sentences in similarity matrix
    sentence_similarity_graph = nx.from_numpy_array(sentence_similarity_matrix)
    scores = nx.pagerank(sentence_similarity_graph)

    # Step 4 - Sort the rank and pick top sentences
    ranked_sentence = sorted(((scores[i],s) for i,s in enumerate(sentences)), reverse=True)

    print(len(ranked_sentence), top_n)

    for i in range(top_n):
        summarize_text.append(" ".join(ranked_sentence[i][1]))

    # Step 5 - Offcourse, output the summarize text
    print("Summarize Text: \n", " ".join(summarize_text))

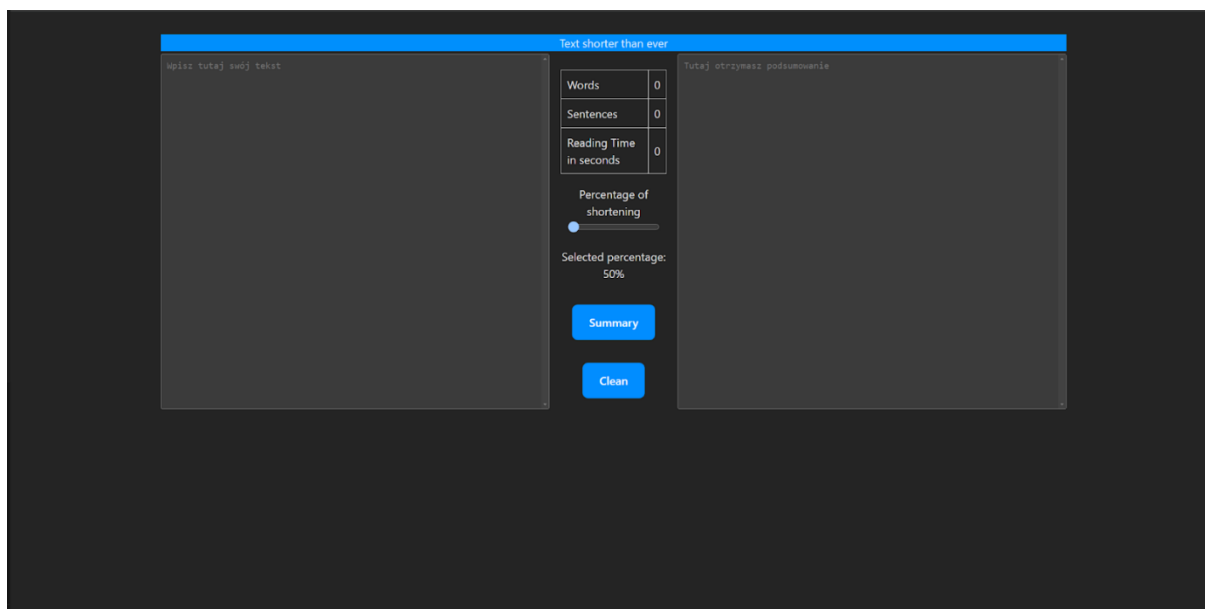
    summarized_text = " ".join(summarize_text)
    words = len(summarized_text.split())

    data = {
        'summarize': summarized_text,
        'words': words,
        'sentences': len(summarize_text),
        'readingTime': words // 5,
    }
    return data
```

4.2 Frontend

Frontend został utworzony react - frameworku javascript. Udostępnia on prosty oraz przejrzysty interfejs użytkownika otwierany w oknie przeglądarki.

Aplikacja frontendowa została napisana aby była responsywna oraz nie obciążała komputera użytkownika



4.2.1. Hanldery

- handleSliderChange - ustawia zmienną shorteningPercentage która jest wysyłana do API w jakim stopniu ma zostać skrócony tekst źródłowy

```
const handleSliderChange = (event) => {  
  const value = event.target.value;  
  setShorteningPercentage(value);  
};
```

- handleSummarize - wysyła zapytanie do api za pomocą akcji fetch, metodą post. Warto zwrócić uwagę na body zapytania do api - wysyłany jest tekst oraz procent skrócenia go

```
const handleSummarize = async () => {  
  console.log({ text: inputText, percent: shorteningPercentage });  
  try {  
    const response = await fetch('http://127.0.0.1:5000/text-to-summarize', {  
      method: 'POST',  
      headers: {  
        'Content-Type': 'application/json',  
      },  
      body: JSON.stringify({ text: inputText, percent: shorteningPercentage }),  
    });  
  
    if (!response.ok) {  
      throw new Error('Nie udało się uzyskać odpowiedzi od serwera.');    }  
  
    const data = await response.json();  
  
    setTableData({  
      words: data.words,  
      sentences: data.sentences,  
      readingTime: data.readingTime,  
    });  
  
    setSummaryText(data.summarize);  
  } catch (error) {  
    console.error('Błąd podczas przetwarzania danych:', error.message);  
  }  
};
```

- handleInputChange - gdy zmienia się wartość inputu zmiana zapisywana jest do zmiennej

```
const handleInputChange = (event) => {  
  setInputText(event.target.value);  
};
```


5. Podsumowanie

Podsumowując cel został osiągnięty. Udało się stworzyć aplikację internetową z przejrzystym interfejsem, która umożliwia streszczanie tekstów.

Wnioski/problemy - nie dla wszystkich tekstów generują się dobre streszczenia, np. Dla tekstów składających się z niewielu długich zdań, streszczenie nie będzie zbyt efektywne. Jest to na pewno otwarta możliwość rozwoju. Inną możliwością jest dodanie użytkowników i przechowywanie streszczonych tekstów w bazie danych.

6. Bibliografia

<https://heartbeat.comet.ml/text-summarization-using-python-and-nltk-d1022ac347eb>

<https://stackoverflow.com/questions/25735644/python-regex-for-splitting-text-into-sentences-sentence-tokenizing>

<https://flask.palletsprojects.com/en/3.0.x/>