

Przetwarzanie Języka Naturalnego

Asystent Głosowy

Autorzy:

Adamczyk Patryk

Bauman Jakub

Borowiecki Mateusz

Kraków 2024

Abstrakt

W dobie sztucznej inteligencji staramy się szukać nowych sposobów na wymianę informacji z komputerami. Mamy do czynienia z multimedialnością w prawdziwym tego słowa znaczeniu. Oprócz odbierania informacji w różnych formach, coraz częściej szukamy środków na przekazywanie ich nie tylko za pomocą tekstu. Asystenci głosowi zdają się być następnym krokiem w tej dziedzinie. W niniejszej pracy opisujemy proces tworzenia prostego asystenta głosowego, pozwalającego na wykonywanie części zadań dzięki wydawaniu poleceń za pomocą mowy.

1. Wstęp

Obserwując obecnie panujące trendy, nie sposób nie zauważyć ogromnego wzrostu popularności tematów związanych ze sztuczną inteligencją. W rozważaniach na ten temat oraz kierunku rozwoju konkretnych technologii, coraz częściej napotykamy pojęcia "multimodalny". Multimodalność oznacza, że w komunikacji z komputerem korzystamy coraz częściej z większego zakresu sposobów niż tylko tekst. Prawdą jest, że na rynku od dawna mamy dostęp do asystentów głosowych, jednakże do tej pory były one traktowane bardziej jako ciekawostka technologiczna niż przydatne na co dzień narzędzie.

Obecnie dzięki rozwojowi sztucznej inteligencji coraz więcej użytkowników zdaje się otwierać na nowe rozwiązania. Częściej można spotkać na ulicy osoby, które zamiast odpisywać na wiadomości, nagrywają krótkie wiadomości audio. Jest to jednoznaczne określenie kierunku rozwoju, które wydarzy się w nadchodzących latach. Dobrze więc zapoznać się z metodami oraz architekturą kryjącymi się w podobnych rozwiązaniach.

Celem pracy jest stworzenie asystenta głosowego, pozwalającego za pomocą głosu, wykonywać pewne zadania czy pozyskiwać informacje. Asystent działający na systemie Windows będzie mieć charakter ogólny.

Asystent głosowy zostanie zaimplementowany w języku Python, ze względu na szeroką gamę dostępnych bibliotek, pozwalających na łatwiejszą transkrypcję oraz przygotowywanie i przetwarzanie tekstu. Głównymi narzędziami wykorzystanymi w projekcie będą biblioteki `speech_recognition`, `pyttsx3`, `nlk` oraz `sklearn` o których opowiemy w kolejnych częściach pracy.

2. Projektowanie systemu

Projektując asystenta głosowego, całość należy podzielić na kilka osobnych, połączonych ze sobą modułów. W kolejności użytkownika, wszystko zaczyna się od wypowiedzenia swojego polecenia w kierunku asystenta. Za to zadanie odpowiada Automatic Speech Recognition. Kolejnym krokiem jest uporządkowanie oraz tokenizacja otrzymanego tekstu. Jest to istotny proces, pozwala on zwiększyć szansę na wybranie odpowiedniej odpowiedzi poprzez odsianie niepotrzebnych słów, odmian czy form. Tak przygotowane dane trafiają do następnego modułu, którym jest wektoryzacja za pomocą TF-IDF(Term Frequency-Inverse Document Frequency). Przeprowadza ważenie słów w dokumentach w oparciu o ich częstotliwość wystąpień oraz kontekst dokumentu. Pozwala on uczynić asystenta bardziej elastycznym narzędziem. Użytkownik nie musi martwić się o idealne dopasowanie swoich słów do szablonu. Na końcu wybrana ze słownika odpowiedź jest przetwarzana na mowę za pomocą pyttsx3. Dla systemów Windows wykorzystuje technologię SAPI5.

2.1 Rozpoznawanie oraz synteza mowy.

Rozpoznawanie mowy jest technologią pozwalającą komputerowi na interpretację mowy ludzkiej, przykładowo w celach transkrypcji lub interakcji. Jej historia rozpoczęła się już w latach 50-tych ubiegłego wieku, wraz z fonetografem Drayfusa-Grafa oraz maszyną rozpoznającą zbiór 10 izolowanych wyrazów. W 2011 roku słownik Google osiągnął kamień milowy, zawierając około miliona różnych słów. Warto wspomnieć również, że próg komercyjnej akceptowalności systemów rozpoznawania mowy przyjmuje się zazwyczaj jako 95% poprawności rozpoznania.

W systemach operacyjnych Windows 10 oraz Windows 11, zaimplementowana została funkcja rozpoznawania mowy. Dzięki temu rozwiązaniu możemy uzyskać dostęp głosowy do sterowania komputerem i tworzenia tekstu za pomocą głosu.

O procesie syntezy mowy możemy po części mówić jako o odwrotności rozpoznawania mowy. Pozwala ona na przetwarzanie tekstu na wypowiedź w postaci dźwiękowej. Pomędzy oczywistymi zastosowaniami takimi jak alarmowanie użytkownika czy tłumaczeniu, systemy syntezy mowy mogą także odczytywać dokumenty pisane alfabetem Braila.

2.2 Tokenizacja tekstu

Tokenizacja tekstu to w najprostszym ujęciu dzielenie go na mniejsze części. Najprostszym przykładem będzie oddzielenie słów w tekście za pomocą spacji. Nie jest to najlepsza metoda ze względów językowych. Często np. w języku angielskim słowa łączone są za pomocą apostrofów, lub nabierają sensu jako połączenia dwóch słów np. "Sri Lanka". Dlatego też warto aby tokenizer potrafił określić w wystarczającym zakresie podobne przypadki.

W ramach tokenizacji warto również wspomnieć o technice lematyzacji. Jest to proces redukowania różnych form słowa do jednej formy, na przykład redukowanie „budowanie”, „budowania” lub „zbudowania” do lematu „budowa”. Jest to bardzo istotny krok, jeżeli dodatkowo odpowiednio przygotowujemy docelowe zapytania do asystenta. Dzięki temu użycie dłuższych zawiłych zdań, będzie mogło zostać rozpoznane jako odpowiednia komenda.

2.3 TF-IDF

TF-IDF jest jedną z metod obliczania wagi słów za pomocą częstotliwości ich występowania. Algorytm analizuje wagę dokumentu jako wektora na podstawie występowania konkretnych słów z jednoczesnym uwzględnieniem że niektóre słowa ogólnie występują częściej niż inne. TF IDF składa się z dwóch komponentów. Pierwszy z nich TF to względna częstotliwość występowania terminu T w dokumencie D. Korzystanie tylko z TF prowadziłoby do sytuacji w których wysoki wynik uzyskiwały by słowa i zwroty ogólne takie jak "tak" lub "nie", dlatego TF należy pomnożyć przez IDF. IDF to odwrotna częstość w dokumentach. Odpowiada ona za wyznaczenie istotności danego słowa. Dzięki tej metodzie możemy z dużą dokładnością określić co jest głównym kontekstem dokumentu.

3. Implementacja

Aplikacja składa się z kilku modułów zaimplementowanych jako obiektowy kod Python. Wybór ten zdaje się być naturalny ze względu na rozległą bazę bibliotek oraz rozwiązań open-source dotyczących podobnych tematów. Python stanowi fundament projektów dotyczących przetwarzania języka naturalnego oraz uczenia maszynowego.

Głównym problem związanym z przetworzeniem z danymi tekstowymi jest to, że są one w formie ciągów znaków. Algorytm potrzebuje numerycznego wektora cech, dlatego zaimplementowano przetwarzanie tekstu za pomocą biblioteki NLP.

Zaimplementowano moduł lematyzacji przetwarzanego tekstu, oznacza to, że słowa są sprowadzane do ich podstawowej formy leksykalnej, zwanej lematem. Przed wywołaniem metody lematyzacji, słowa konwertowane są na tokeny. Dzięki użyciu metody **WordNetLemmatizer**, grupuje ona słowa w zestawy synonimów co pozwala na lepsze wychwycenie pożądanego przez nas dopasowania.

Przykłady konwersji danych przez metodę **WordNetLemmatizer**:

- rocks -> rock
- corpora -> corpus
- better -> good

```
1 def preprocess_text(self, text):
2     if text:
3         tokens = word_tokenize(text)
4         words = [self.lemmatizer.lemmatize(word) for word in tokens if
5 word.isalpha() and word not in self.stop_words]
6         return ' '.join(words)
7     return ""
```

Metoda **get_response** odpowiada za przetwarzanie zapytań zadanych przez użytkownika i przetworzenie go tak aby zwrócona została odpowiedź na podstawie podobieństwa tekstowego. Inicjalizowany jest wektor TF IDF z wbudowaną listą wyrazów stopu. TfidfVectorizer tworzy słownik wszystkich unikalnych tokenów występujących w korpusie. Do każdego tokenu przypisywany jest unikalny indeks. Obliczane TF, mierzy ile razy dane słowo pojawia się w ciągu tekstowym.

$$TF(t, d) = \frac{\text{Liczba wystąpień } t \text{ w } d}{\text{Całkowita liczba słów w } d}$$

Obliczany jest IDF - to miara tego jak ważne jest dane słowo w całym korpusie.

$$IDF(t, D) = \log \left(\frac{\text{Całkowita liczba dokumentów}}{\text{Liczba dokumentów zawierających } t} \right)$$

Całkowite obliczenie wektora TFIDF jest realizowane za pomocą poniższego wzoru.

$$TF-IDF(t, d, D) = TF(t, d) \times IDF(t, D)$$

TFIDF jest iloczynem TF i IDF dla danego słowa w dokumencie. Współczynnik ten jest miarą służącą do oceny znaczenia słowa w konkretnym dokumencie.

Następnie potencjalne odpowiedzi są przekształcane na macierz TFIDF. Mając macierz możemy dokonać konwersji zapytania użytkownika na wektor zapytania. Mając macierz i wektor możemy obliczyć podobieństwo cosinusa, mierzy ono podobieństwo pomiędzy dwoma dokumentami.

$$\text{Cosine Similarity}(A, B) = \frac{A \cdot B}{\|A\| \times \|B\|}$$

Mając procentową wartość podobieństwa następuje obliczenie indeksu podobieństwa, a następnie wybranie tego indeksu ze słownika jako odpowiedź

Przykład użycia podobieństwa cosinusa:

- Zapytanie asystenta to: "name"
- W słowniku odpowiedzi jest tylko jeden element pasujący do danego słowa a jego klucz to: "what is your name"

```
Query vector: (0, 13) 1.0  
Cosine similarity: (0, 1) 0.5898245452468465  
Most similar idx: 1
```

- Funkcja ta obliczyła, że dla indeksu pierwszego(pierwszy klucz w słowniku licząc od zera) jest najbardziej prawdopodobny jako zapytanie.
- Zwracana jest wartość klucza "what is your name" czyli "My name is Steve."

```

1  def get_response(self, query, source=responses):
2      if query:
3          try:
4              # Preprocess the query
5              preprocessed_query = self.preprocess_text(query)
6              tfidf_vectorizer = TfidfVectorizer()
7              answers = list(source.keys())
8              tfidf_matrix = tfidf_vectorizer.fit_transform(answers)
9              query_vec = tfidf_vectorizer.transform([preprocessed_query])
10             print(f"Query vector: {query_vec}")
11             # Calculate cosine similarity between query and answers
12             cosine_similarities = np.dot(query_vec, tfidf_matrix.T)
13             most_similar_idx = np.argmax(cosine_similarities)
14             print(f"Cosine similarity: {cosine_similarities}\nMost similar idx:
{most_similar_idx}")
15             response_key = answers[most_similar_idx]
16             response = source[response_key]
17             while callable(response):
18                 callable_response = response(self)
19
20                 if callable_response == False:
21                     self.speak("Sorry, I couldn't process this information.")
22                 else:
23                     return callable_response
24             return response
25         except AttributeError:
26             return "I'm sorry, I didn't understand that."

```

3.1 Odczyt audio

Rozpoczęcie zapisu audio odbywa się na podstawie różnicy dźwięku głosu względem stanu początkowego, którym są dźwięki otoczenia lub cisza. Po pobraniu próbki dźwięku, następuje próba odczytu wypowiedzianej sentencji. Aplikacja korzysta z dwóch różnych systemów rozpoznawania mowy. Pierwszym z nich jest rozwiązanie dostarczane przez Google. W wersji angielskiej do jego użycia nie jest potrzebny klucz. Podczas testów okazało się, że Google lepiej rozpoznaje zdania niż otwartoźródłowy PocketSphinx. Mimo to, Google częściej zgłasza całkowitą porażkę w rozpoznawaniu. Z kolei PocketSphinx częściej popełnia błędy, ale rzadziej zwraca komunikat o całkowitym niepowodzeniu.

Dlatego w przypadku niepowodzenia w rozpoznawaniu przez algorytm dostarczony przez Google, asystent spróbuje ponownie dokonać transkrypcji z udziałem Sphinx, aby zmaksymalizować szanse na podanie użytkownikowi odpowiedzi. Kod odpowiedzialny za odczyt audio możemy zobaczyć poniżej.

```
def listen(self):
    with self.microphone as source:
        print("Say something!")
        self.recognizer.adjust_for_ambient_noise(source)
        audio = self.recognizer.listen(source)
    try:
        text = self.recognizer.recognize_google(audio, language='en-US')
        print("Google Assistant: thinks you said: " + text)
        if text == None:
            text = self.recognizer.recognize_sphinx(audio, language='en-US')
            print("Sphinx Assistant: thinks you said: " + text)
        assistant.speak("I heard: " + text)
        return text
    except sr.UnknownValueError:
        print("Assistant could not understand audio")
    except sr.RequestError as e:
        print("Assistant error; {0}".format(e))
```

4. Podsumowanie

Stworzony asystent głosowy to nie tylko udany projekt, ale również solidna baza do dalszego rozwoju i tworzenia bardziej zaawansowanych rozwiązań. Aplikacja spełnia założone cele, oferując funkcjonalność, która może znacząco zwiększyć produktywność użytkowników podczas korzystania z komputera.

Sama implementacja była bardzo pouczającym doświadczeniem. Pozwoliła nam zajrzeć za kulisy obecnie bardzo popularnych i rozwijających się gałęzi informatyki. To z pewnością nie koniec naszej przygody z przetwarzaniem języka naturalnego. Już teraz planujemy dalsze prace nad rozszerzeniem funkcjonalności aplikacji i udoskonaleniem jej działania. Jesteśmy przekonani, że tego typu rozwiązania odegrają kluczową rolę w przyszłości, stając się nieodłącznym elementem naszego życia i pracy.

Bibliografia:

1. Voice Assistant using python

<https://www.geeksforgeeks.org/voice-assistant-using-python/> [ostatni dostęp 19.05.2024]

2. Sirisha Rella, *Essential Guide to Automatic Speech Recognition Technology*, 2022,
<https://developer.nvidia.com/blog/essential-guide-to-automatic-speech-recognition-technology/>
3. Bird, Steven, Edward Loper and Ewan Klein (2009), *Natural Language Processing with Python*. O'Reilly Media Inc.
4. Cambridge Dictionary, *lemmatization*
<https://dictionary.cambridge.org/dictionary/english/lemmatization> [dostęp 21.05.2024]