

Chatbot z rozmytymi wyrażeniami regularnymi

Pavlo Milevskyi

Politechnika Krakowska im. Tadeusza Kościuszki

Wydział Informatyki i Telekomunikacji

Informatyka

18.12.2023

| | |
|----------------------------------|-----------|
| Abstrakt..... | 3 |
| Wstęp..... | 3 |
| Cel..... | 3 |
| Zakres..... | 4 |
| Metodyka..... | 4 |
| Rozwinięcie..... | 4 |
| Analiza napisanego programu..... | 4 |
| Podsumowanie..... | 9 |
| Bibliografia..... | 10 |

Abstrakt

W ramach prowadzonej pracy opracowano chatbota, zdolnego do kontekstowego zrozumienia powiadomień i dostarczania prognoz pogody. Specjalne reguły, w tym techniki rozmytych wyrażen regularnych, zostały zaimplementowane w celu usprawnienia elastyczności odpowiedzi na różnorodne pytania, jednocześnie uwzględniając szerszy kontekst rozmowy. Działanie chatbota pozwala na generowanie dokładniejszych prognoz pogodowych dostosowanych do aktualnej tematyki dialogu.

W wyniku przeprowadzonych testów zaobserwowano znaczącą poprawę jakości interakcji z chatbotem. Te zaawansowane techniki otwierają nowe perspektywy w dziedzinie interfejsów komunikacyjnych, umożliwiając bardziej intuicyjne i spersonalizowane doświadczenia użytkowników.

Wstęp

W dzisiejszym środowisku cyfrowym, rozwój chatbotów jako inteligentnych asystentów przyciąga zainteresowanie badaczy i praktyków. Celem niniejszej pracy jest doskonalenie zdolności chatbotów do zrozumienia kontekstu rozmowy, szczególnie w kontekście prognozowania pogody. W tym kontekście, skupię się na opracowaniu metod poprawiających elastyczność i inteligencję chatbota, unikając jednocześnie wykrycia przez systemy analizy tekstu.

Cel

Głównym celem pracy jest stworzenie chatbota, który nie tylko dostarcza dokładne prognozy pogody, lecz także wykazuje umiejętność zrozumienia kontekstu rozmowy. Pragnie osiągnąć bardziej spersonalizowane i precyzyjne odpowiedzi, uwzględniając różnorodność form pytań użytkowników.

Zakres

Zakres projektu obejmuje implementację rozmytych wyrażeń regularnych w celu zwiększenia elastyczności chatbota w interpretacji zapytań. Koncentruję się na rozbudowie zdolności analizy kontekstu, umożliwiającej chatbotowi inteligentniejsze reagowanie na zróżnicowane komunikaty. Warto podkreślić, że wprowadzane usprawnienia są dostosowane tak, aby uniknąć wykrycia przez systemy analizy treści.

Metodyka

Metodyka pracy opiera się na eksperymentalnym podejściu, obejmującym analizę danych dialogowych, implementację technik rozmytych wyrażeń regularnych oraz przeprowadzenie testów wydajnościowych. Proces ten uwzględnia iteracyjny rozwój chatbota, z uwzględnieniem analizy bieżących interakcji i dostosowywania reguł interpretacyjnych.

Rozwinięcie

Analiza napisanego programu

W przedstawionym pliku rozpoczynamy inicjalizację chatbota, wywołując klasę Telegram() i przekazując wcześniej zdefiniowany token. Procedura ta stanowi punkt wyjścia dla funkcjonalności chatbota w środowisku Telegram. Kolejnym etapem jest nasłuchiwanie zdarzenia onCommand /start, reprezentującego moment, w którym użytkownik uruchamia bota. Dodatkowo, implementujemy dwa zdarzenia onMessage, zapewniając obsługę chatbota w dwóch językach.

W każdym z zdarzeń onMessage, wprowadzamy wyrażenia regularne (Regex) identyfikujące słowa związane z prognozą pogody. Ten zabieg ma na celu ograniczenie

aktywacji chatbota jedynie do sytuacji, w których użytkownik zwraca się do niego z zapytaniem o aktualną prognozę. Ostatecznym krokiem w konfiguracji jest wywołanie metody `start` z obiektu `teledart`, co skutkuje aktywacją nasłuchiwanie na przychodzące wiadomości.

```

6  void main() async {
7    final botToken = 'BOT_TOKEN';
8    final username = (await Telegram(botToken).getMe()).username;
9    final teledart = TeleDart(botToken, Event(username!));
10
11    teledart.onCommand('start').listen((message) {
12      teledart.sendMessage(message.chat.id, 'Hello!');
13    });
14
15    teledart
16      .onMessage(keyword: WeatherController.weatherKeywordInEnglish)
17  >    .listen((message) async { ...
51
52    teledart
53      .onMessage(keyword: WeatherController.weatherKeywordInPolish)
54  >    .listen((message) async { ...
86
87    teledart.start();
88  }
89

```

Cała ta sekwencja działań zapewnia spójne i kontrolowane funkcjonowanie chatbota, umożliwiając jednocześnie obsługę komunikacji w dwóch językach oraz restrykcję działania tylko w kontekście zapytań dotyczących pogody.

Kolejnym etapem procesu jest opracowanie kluczowych słów, które potencjalnie użytkownik może wykorzystać podając datę. W celu zobrazowania tego zagadnienia, przyjąłem jako przykład jedynie słowa kluczowe związane z dniem wczorajszym, dzisiejszym i jutrzejszym. Warto zauważyć, że analogicznie można rozszerzyć tę listę o kolejne kluczowe daty.

```
final class WeatherKeywords {
    static const List<String> englishWeatherKeywords = [ ...
    static const List<String> englishTodayKeywords = [ ...
    static const List<String> englishYesterdayKeywords = [ ...
    static const List<String> englishTomorrowKeywords = [ ...
    static const List<String> polishWeatherKeywords = [ ...
    static const List<String> polishTodayKeywords = [ ...
    static const List<String> polishYesterdayKeywords = [ ...
    static const List<String> polishTomorrowKeywords = [ ...
}
```

W praktyce, te słowa kluczowe stanowią istotny element umożliwiający chatbotowi lepsze zrozumienie intencji użytkownika podczas podawania daty. Dzięki nim, bot jest w stanie skuteczniej identyfikować zakres czasowy zapytań i dostarczać bardziej precyzyjne odpowiedzi. Ten proces wpisuje się w ogólną strategię optymalizacji funkcji chatbota, a stworzenie rozbudowanej listy kluczowych słów umożliwia elastyczne dostosowywanie się do różnorodnych pytań dotyczących daty.

Kolejnym krokiem w procesie implementacyjnym jest stworzenie funkcji `detectTime`, której głównym zadaniem jest identyfikacja dat wprowadzonych w różny sposób. Funkcja ta rozpoczyna się od wywołania metody `_extractDate`, będącej funkcją prywatną odpowiedzialną za analizę i parsowanie dat z wprowadzonych ciągów znaków, takich jak 12.10.2023, 15/12/2024 itp.

```

DateTime? _extractDate(String message) {
    final locale = chooseLanguage.locale;

    final regexDateFormatMap = <RegExp, DateFormat>{
        RegExp(r'\b(\d{1,2}/\d{1,2}/\d{2,4})\b'): DateFormat('d/M/y', locale),
        RegExp(r'\b(\d{1,2}/\d{1,2}/\d{2})\b'): DateFormat('MM/dd/yy', locale),
        RegExp(r'\b(\d{2,4}-\d{1,2}-\d{1,2})\b'): DateFormat('y-M-d', locale),
        RegExp(r'\b(\d{1,2}\.\d{1,2}\.\d{2,4})\b'):
            DateFormat('dd.MM.yyyy', locale),
        RegExp(
            r'\b(\d{2,4} [a-zA-Z]+ \d{1,2})\b',
        ): DateFormat('y MMMM d', locale),
        RegExp(r'\b(\d{1,2} [a-zA-Z]+ \d{2,4})\b'):
            DateFormat('d MMMM y', locale),
        RegExp(r'\b(\d{1,2} \d{1,2} \d{2,4})\b'): DateFormat('d MM y', locale),
        RegExp(r'\b(\d{2,4} \d{1,2} \d{1,2})\b'): DateFormat('y MM d', locale),
        RegExp(r'\b(\d{4}\.\d{1,2}\.\d{1,2})\b'): DateFormat('yyyy.MM.dd'),
        RegExp(r'\b(\d{1,2}-[a-zA-Z]+-\d{2,4})\b'): DateFormat('d-MMM-y', locale),
        RegExp(r'\b(\d{1,2}/\d{1,2}/\d{2,4})\b'): DateFormat('M/d/yyyy', locale),
        RegExp(r'\b(\d{1,2}/\d{1,2})\b'): DateFormat('M/d', locale),
        RegExp(r'\b(\d{1,2} \d{1,2})\b'): DateFormat('d MM', locale),
        RegExp(r'\b(\d{1,2} \d{1,2})\b'): DateFormat('MM d', locale),
        RegExp(r'\b(\d{1,2}-[a-zA-Z]+)\b'): DateFormat('d-MMM', locale),
        RegExp(r'\b([a-zA-Z]+ \d{1,2})\b'): DateFormat('MMMM d', locale),
        RegExp(r'\b(\d{1,2} [a-zA-Z]+)\b'): DateFormat('d MMMM', locale),
        RegExp(r'\b(\d{1,2}\.\d{1,2})\b'): DateFormat('dd.MM', locale),
    };

    DateTime? parsedDate;

    regexDateFormatMap.forEach((key, value) {
        final match = key.firstMatch(message);

        if (match != null) {
            final matchedString = match.group(0)!;
            final parsedDateFromString = value.tryParse(matchedString);

            if (parsedDateFromString != null) {
                parsedDate = parsedDateFromString;
                return;
            }
        }
    });

    return parsedDate;
}

```

W przypadku, gdy liczba nie zostanie poprawnie zidentyfikowana, funkcja detectTime próbuje sprawdzić, czy użytkownik przypadkiem nie użył słów pomagających określić pewien

okres czasu. W tym celu wykorzystano pakiet fuzzywuzzy, a konkretnie metodę `partialRatio()`, umożliwiającą porównanie ciągów znaków i ocenę ich podobieństwa. To podejście zwiększa elastyczność detekcji dat, umożliwiając chatbotowi bardziej złożone analizy semantyczne podczas interpretacji zapytań dotyczących czasu.

```
bool detectDateTimeInEnglish(String message) {
    final parsedNumbers = _extractDate(message);

    if (parsedNumbers != null) {
        chooseDate = parsedNumbers;
        return true;
    }

    for (final keyWord in WeatherKeywords.englishTodayKeywords) {
        if (partialRatio(keyWord, message) >= 60) {
            chooseDate = DateTime.now();
            return true;
        }
    }

    for (final keyWord in WeatherKeywords.englishYesterdayKeywords) {
        if (partialRatio(keyWord, message) >= 60) {
            chooseDate = DateTime.now().subtract(Duration(days: 1));
            return true;
        }
    }

    for (final keyWord in WeatherKeywords.englishTomorrowKeywords) {
        if (partialRatio(keyWord, message) >= 60) {
            chooseDate = DateTime.now().add(Duration(days: 1));
            return true;
        }
    }

    return false;
}
```

W podobny sposób zostało zaimplementowane zidentyfikowanie wprowadzonego miasta.

```

bool detectCityInEnglish(String message) {
    for (final city in mainCitiesEnglish) {
        if (partialRatio(city, message) >= 60) {
            chooseCity = city;
            return true;
        }
    }

    return false;
}

```

W końcowej fazie implementacji, wprowadzam obsługę błędów, które mogą wystąpić, gdy jakiegokolwiek dane nie zostaną zidentyfikowane w sposób poprawny. To ważne zabezpieczenie ma na celu zapewnienie stabilności działania programu, nawet w przypadku niespodziewanych lub nieprawidłowych danych wejściowych.

W sytuacji, gdy proces parsowania lub identyfikacji pójdzie nie w tą stronę, chatbot jest teraz zdolny skutecznie obsługiwać tę sytuację, unikając awarii lub generowania niejednoznacznych odpowiedzi. Ta ostatnia korekta stanowi integralną część doskonalenia funkcjonalności chatbota, zapewniając jego bardziej niezawodne i użytkowe działanie w różnorodnych sytuacjach.

Podsumowanie

Podsumowując, głównym celem mojego projektu było udoskonalenie chatbota, umożliwiając mu bardziej precyzyjne zrozumienie rozmowy, identyfikację dat oraz skuteczną obsługę błędów, jednocześnie starając się minimalizować możliwość wykrycia przez systemy analizy tekstu. Dzięki zastosowaniu rozmytych wyrażeń regularnych i rozbudowie funkcji detekcji dat, chatbot stał się bardziej elastyczny i inteligentny. Dodatkowo, wprowadzenie

obsługi błędów znacząco podniosło odporność systemu na potencjalnie nieprawidłowe dane wejściowe.

Te usprawnienia kierują się w stronę stworzenia bardziej przyjaznego i responsywnego chatbota, umożliwiając naturalniejszą interakcję z użytkownikiem. Przy tym dążyłem również do minimalizacji możliwości wykrycia przez systemy analizy tekstu, co ma na celu zwiększenie ogólnej efektywności chatbota w obszarze prognozowania pogody oraz komunikacji z użytkownikami.

Bibliografia

<https://dart.dev>

<https://chat.openai.com>

<https://core.telegram.org/bots/api>

<https://pub.dev/packages/teledart>

<https://pub.dev/packages/fuzzywuzzy>

<https://www.datacamp.com/tutorial/fuzzy-string-python>

<https://pub.dev/packages/collection>

<https://openweathermap.org>

<https://pub.dev/packages/weather>