Assignment 7 - Dialogue Systems

March 5, 2024

1 Group 5 - Module 7: Dialogue systems and question answering

---

1.0.1 Group Members:

- **Nils Dunlop, 20010127-2359, Applied Data Science, e-mail: gusdunlni@student.gu.se (15 hours)**
- **Francisco Erazo, 19930613-9214, Applied Data Science, e-mail: guserafr@student.gu.se (15 hours)**

We hereby declare that we have both actively participated in solving every exercise. All solutions are entirely our own work, without having taken part of other solutions." (This is independent and additional to any declaration that you may encounter in the electronic submission system.)

2 Assignment 7

2.1 Problem 1: Reading and Summary

The paper describes the development and functionality of GUS (Genial Understander System), a dialogue-based system designed primarily to simulate a travel agent's role for booking return trips within California. The system's main objective is to process and understand English dialogues, showcasing its capacity to manage various elements of natural conversation. These include handling shifts in initiative, interpreting indirect responses, resolving anaphora or expressions, and understanding conversational implicates.

GUS's architecture is modular, integrating multiple components such as morphological and syntactic analyzers and a frame reasoner. This structure allows it to execute dialogues efficiently through procedural attachment and scheduling techniques. The paper highlights the system's innovative approach to dialogue management, demonstrating its potential for understanding and generating human-like conversations within its specified domain.

Despite its advancements, the paper also discusses GUS's limitations and emphasizes the complexity of achieving realistic dialogue interactions, especially at the time of its publication in 1976. It points out the necessity for more sophisticated knowledge representation and reasoning capabilities

to improve the system's performance. The development of GUS marks a significant step in the field of language understanding systems, illustrating both the potential and challenges of creating machines capable of engaging in meaningful dialogues with humans. Through its frame-driven design, GUS contributes to the ongoing research in artificial intelligence, particularly in natural language processing and understanding, setting a foundation for future improvements and innovations in the area.

2.2 Problem 2: Implement A Simple Dialogue System

2.2.1 2 a) Implement a simple text-based digital assistant that can help with at least three things

[77]:
```
# Import libraries
import spacy
import requests
import base64
from datetime import datetime
from spacy.matcher import Matcher

# Load the spaCy model
nlp = spacy.load("en_core_web_md")

# Define the digital assistant class
class DigitalAssistant:
    def __init__(self, intents):
        self.nlp = spacy.load("en_core_web_md")
        self.intents = intents
        self.matcher = Matcher(self.nlp.vocab)
        self.setup_matcher()
        self.intent_handlers = {
            "WEATHER": self.handle_weather,
            "RESTAURANT": self.handle_restaurant,
            "TRAM": self.handle_tram
        }
        self.intent_docs = {
            intent: [self.nlp(phrase) for phrase in phrases]
            for intent, phrases in self.intents.items()
        }

    def setup_matcher(self):
        # Add match patterns for entity recognition
        weather_patterns = [[{"LOWER": "weather"}], [{"LOWER": "forecast"}],␣
↪[{"LOWER": "temperature"}]]
        restaurant_patterns = [[{"LOWER": "eat"}], [{"LOWER": "restaurant"}],␣
↪[{"LOWER": "food"}]]
```

```
        tram_patterns = [[{"LOWER": "tram"}], [{"LOWER": "bus"}], [{"LOWER":␣
↪"schedule"}], [{"LOWER": "next"}]]
        for pattern in weather_patterns:
            self.matcher.add("WEATHER", [pattern])
        for pattern in restaurant_patterns:
            self.matcher.add("RESTAURANT", [pattern])
        for pattern in tram_patterns:
            self.matcher.add("TRAM", [pattern])

    def determine_context(self, user_input):
        doc = self.nlp(user_input)
        matches = self.matcher(doc)

        if matches:
            for match_id, start, end in matches:
                span = doc[start:end]
                intent = self.nlp.vocab.strings[match_id]
                handler = self.intent_handlers.get(intent)

                if handler:
                    response = handler(user_input)
                    return response
        else:
            clarification_message = (
                "I'm not sure what you're asking for. "
                "I specialize in weather forecasts, restaurant recommendations,␣
↪and tram schedules. "
                "Could you please specify what you need help with?"
            )
            return clarification_message

    def identify_intent(self, user_doc):
        max_similarity = 0.0
        best_intent = None
        for intent, docs in self.intent_docs.items():
            for doc in docs:
                similarity = user_doc.similarity(doc)
                if similarity > max_similarity:
                    max_similarity = similarity
                    best_intent = intent
        return best_intent, max_similarity

    def fetch_weather_forecast(self):
        # Gothenburg latitude and longitude
        latitude = "57.7089"
        longitude = "11.9746"
```

```
        url = f"https://api.open-meteo.com/v1/forecast?
↪latitude={latitude}&longitude={longitude}&current_weather=true"

        response = requests.get(url)
        if response.status_code == 200:
            data = response.json()
            weather_description = data['current_weather']['weathercode']
            temperature = data['current_weather']['temperature']
            # You may need to map the weather code to a description
            weather_description_text = self.
↪map_weather_code_to_description(weather_description)
            return f"The current weather in Gothenburg is␣
↪{weather_description_text} with a temperature of {temperature}°C."
        else:
            print(f"Failed to fetch weather data: Status Code {response.
↪status_code}, Response: {response.text}")
            return "Sorry, I couldn't fetch the weather information right now."


    def map_weather_code_to_description(self, code):
        # Mapping based on WMO Weather interpretation codes
        weather_code_mapping = {
            0: "clear sky",
            1: "mainly clear",
            2: "partly cloudy",
            3: "overcast",
            45: "fog",
            48: "depositing rime fog",
            51: "drizzle: light intensity",
            53: "drizzle: moderate intensity",
            55: "drizzle: dense intensity",
            56: "freezing drizzle: light intensity",
            57: "freezing drizzle: dense intensity",
            61: "rain: slight intensity",
            63: "rain: moderate intensity",
            65: "rain: heavy intensity",
            66: "freezing rain: light intensity",
            67: "freezing rain: heavy intensity",
            71: "snowfall: slight intensity",
            73: "snowfall: moderate intensity",
            75: "snowfall: heavy intensity",
            77: "snow grains",
            80: "rain showers: slight intensity",
            81: "rain showers: moderate intensity",
            82: "rain showers: violent intensity",
            85: "snow showers: slight intensity",
            86: "snow showers: heavy intensity",
```

```
            95: "thunderstorm: slight or moderate",
            96: "thunderstorm with slight hail",
            99: "thunderstorm with heavy hail",
        }
        # Handle compound codes by checking the presence of the code in a range
        if code in weather_code_mapping:
            return weather_code_mapping[code]
        elif any(code in range_ for range_ in [(45, 48), (51, 55), (56, 57),␣
↪(61, 65), (66, 67), (71, 75), (80, 82), (85, 86)]):
            return weather_code_mapping[min(filter(lambda x: x >= code,␣
↪weather_code_mapping))]
        else:
            return "Unknown weather condition"

    def fetch_restaurant_recommendations(self, location="Gothenburg",␣
↪term="restaurants", sort_by="rating"):
        api_key =␣
↪"k6g36lOU262KATfdkdn8eq1CxynKaupS4_pbvRLudUAZegBiUXVF6jy7gOOOHZmcb_gszKmK2-fK4Be5ggG4Ius1JV
        headers = {
            "Authorization": f"Bearer {api_key}"
        }
        params = {
            "location": location,
            "term": term,
            "sort_by": sort_by,
            "limit": 5
        }

        # Yelp search endpoint
        url = "https://api.yelp.com/v3/businesses/search"

        response = requests.get(url, headers=headers, params=params)

        if response.status_code == 200:
            # Parse the JSON response
            data = response.json()
            businesses = data["businesses"]

            # Create a list of restaurant names and their rating
            restaurant_recommendations = [
                f"{business['name']} ({business['rating']} stars)" for business␣
↪in businesses
            ]
            return restaurant_recommendations
        else:
            print(f"Failed to fetch restaurant data: Status Code {response.
↪status_code}, Response: {response.text}")
```

```
            return "Sorry, I couldn't fetch the restaurant information right␣
↪now."

    def get_vasttrafik_access_token(self, client_id, client_secret):
        # Endpoint for the access token
        token_url = "https://ext-api.vasttrafik.se/token"

        # Encode client ID and secret to base64 for the Authorization header
        client_credentials = f"{client_id}:{client_secret}"
        client_credentials_base64 = base64.b64encode(client_credentials.
↪encode()).decode()

        # Headers
        headers = {
            "Content-Type": "application/x-www-form-urlencoded",
            "Authorization": f"Basic {client_credentials_base64}"
        }

        # Body parameters
        body = {
            "grant_type": "client_credentials"
        }

        # POST request to get the token
        try:
            response = requests.post(token_url, headers=headers, data=body)
            response.raise_for_status()

            # If the request is successful, return the JSON response
            return response.json()['access_token']
        except requests.RequestException as e:
            # Return a dictionary with an error message
            return {"error": str(e)}

    def get_stop_id(self, stop_name, access_token):
        url = f"https://ext-api.vasttrafik.se/pr/v4/locations/by-text?
↪q={stop_name}&types=stoparea&limit=10&offset=0"
        headers = {"Authorization": f"Bearer {access_token}"}
        response = requests.get(url, headers=headers)

        if response.status_code == 200:
            data = response.json()
            stop_id = data['results'][0]['gid']
            return stop_id
        else:
            print(f"Failed to fetch stop ID: {response.status_code}, {response.
↪text}")
```

```
            return None

    def fetch_tram_schedule(self, stop_name):
        # Obtain the access token
        access_token = self.
↪get_vasttrafik_access_token(client_id='t6k4WYN9W93dzrH7ffXo7ZQqruwa',␣
↪client_secret='0J3e7bwhBO1lW33ffdZnanODTVYa')
        if not access_token:
            return "Could not obtain access token."

        # Get the stop ID for the given stop name
        stop_id = self.get_stop_id(stop_name, access_token)
        if not stop_id:
            return f"Could not find stop ID for {stop_name}."

        # Fetch the tram schedule using the stop ID and access token
        url = f"https://ext-api.vasttrafik.se/pr/v4/stop-areas/{stop_id}/
↪departures?
↪timeSpanInMinutes=60&maxDeparturesPerLineAndDirection=2&limit=3&offset=0&includeOccupancy=f
        headers = {"Authorization": f"Bearer {access_token}"}
        response = requests.get(url, headers=headers)
        if response.status_code == 200:
            data = response.json()
            departures = data['results']
            top_departures = departures[:3]

            schedule = []
            for d in top_departures:
                estimated_time_str = d['estimatedTime']
                if '.' in estimated_time_str:
                    # Truncate the milliseconds
                    estimated_time_str = estimated_time_str[:estimated_time_str.
↪rindex('.')+7]
                estimated_time = datetime.fromisoformat(estimated_time_str)
                readable_time = estimated_time.strftime('%H:%M:%S')

                # Add the departure to the schedule list
                line_info = f"{d['serviceJourney']['line']['name']} to␣
↪{d['serviceJourney']['direction']}"
                schedule_info = f"{line_info} departs at {readable_time}"
                schedule.append(schedule_info)

            return schedule
        else:
            print(f"Failed to fetch tram schedule: {response.status_code},␣
↪{response.text}")
            return "Sorry, I couldn't fetch the tram schedule right now."
```

```
    def handle_weather(self, user_input):
        doc = self.nlp(user_input)
        matches = self.matcher(doc)
        for match_id, start, end in matches:
            span = doc[start:end]
            if self.nlp.vocab.strings[match_id] == "WEATHER":
                forecast = self.fetch_weather_forecast()
                if forecast:
                    return f"You asked about the {span.text}. {forecast}"
        return "I'm sorry, I didn't understand your weather question. Can you␣
↪ask differently?"

    def handle_restaurant(self, user_input):
        doc = self.nlp(user_input)
        matches = self.matcher(doc)
        for match_id, start, end in matches:
            span = doc[start:end]
            if self.nlp.vocab.strings[match_id] == "RESTAURANT":
                recommendations = self.fetch_restaurant_recommendations()
                if recommendations:
                    return f"Looking for a place to {span.text}? Here are some␣
↪suggestions: {', '.join(recommendations)}"
        return "I'm sorry, I didn't catch what type of restaurant you're␣
↪looking for."

    def handle_tram(self, user_input):
        doc = self.nlp(user_input)
        matches = self.matcher(doc)
        stop_name = None
        # Extract stop name if mentioned
        # Otherwise default to 'Brunnsparken'
        for ent in doc.ents:
            if ent.label_ == "GPE" or ent.label_ == "LOC":
                stop_name = ent.text
                break
        stop_name = stop_name if stop_name else 'Brunnsparken'
        schedule = self.fetch_tram_schedule(stop_name)
        if schedule:
            return f"The next trams from {stop_name} are: {'; '.join(schedule)}"
        else:
            return "I'm sorry, I couldn't find the tram schedule right now."

    def get_response(self, user_input):
        # Calls determine_context to process the input
        return self.determine_context(user_input)
```

[78]:
```
# Initialize the assistant with intents
intents = {
    "weather": ["What's the weather like?", "Tell me the weather forecast", "Is␣
 ↪it going to rain today?"],
    "restaurant": ["Find me a place to eat", "I want to order food", "Recommend␣
 ↪a restaurant"],
    "tram": ["When is the next tram?", "Tram schedule", "How often does the bus␣
 ↪run?"],
}
assistant = DigitalAssistant(intents)

# Sample user input processing
print(assistant.get_response("What's the weather like today?"))
print(assistant.get_response("I'm looking for a place to eat."))
print(assistant.get_response("When is the next tram from Brunnsparken?"))
```

```
You asked about the weather. The current weather in Gothenburg is clear sky with
a temperature of 4.2°C.
Looking for a place to eat? Here are some suggestions: Samui Thai Express (5.0
stars), Trattoria La Strega (5.0 stars), Tre Små Rum (4.9 stars), Swedish Taste
(4.8 stars), Fiskekrogen (4.8 stars)
The next trams from Brunnsparken are: Spårvagn 1 to Östra Sjukhuset departs at
19:38:00; Spårvagn 10 to Brämaregården departs at 19:38:00; Buss 16 to
Högsbohöjd departs at 19:39:00
```

As per the assignment description, we designed a simple text-based digital assistant capable of performing three primary functions (as seen above): 1. Providing Gothenburg weather forecasts 2. Finding restaurants in Gothenburg 3. Finding Gothenburg tram/bus schedules

2.2.2 Approach and Design

The assistant is built using Python with the Spacy library for natural language processing (NLP) and the Requests library for handling API requests. The core functionalities are achieved through the integration of various APIs, including Open-Meteo for weather forecasts and Yelp Fusion for restaurant recommendations. The Västtrafik public transport API is used for tram and bus schedules in Gothenburg. We decided to focus on Gothenburg to reduce the complexity of the system and to make it easier to test and verify the results.

Key Components

- **Spacy NLP**: Utilized for processing user input and identifying the context of queries based on keyword matching.
- **Matcher**: A Spacy Matcher object is used to define patterns for recognizing keywords related to weather, restaurants, and tram/bus schedules.
- **API Integration**: Requests are made to external APIs to fetch real-time data for weather, restaurant, and tram/bus schedule information.

Implementation Details

1. **Intent Recognition**: The digital assistant uses Spacy's Matcher to identify the user's intent based on predefined patterns. This allows for a flexible dialogue system that can understand variations in how users may request information.
2. **Dynamic Response Handling**: Based on the identified intent, the assistant dynamically selects the appropriate handler function to process the request and fetch the relevant information.
3. **Extensibility**: The architecture of the assistant is designed to be easily extensible. New intents and handler functions can be added to the `intent_handlers` dictionary without modifying the core logic of the assistant.

2.2.3 Sample Outputs

The assistant can process queries like:

- "What's the weather like today?"
- "I'm looking for a place to eat."
- "When is the next tram from Brunnsparken?"

For each query, it identifies the intent, calls the corresponding handler, and returns information fetched from the respective API.

2.2.4 2 b) Suggest how - if more time were available - you could make your dialogue system more advanced

---

2.2.5 Limitations

- **Keyword Dependency**: The current implementation relies heavily on keyword matching for intent recognition, which may not capture the full nuance of user queries.
- **Static Patterns**: The patterns used for matching are predefined and static, which may limit the assistant's ability to understand varied or complex queries.
- **Error Handling**: The system's response to unrecognized queries or API failures could be improved to provide more helpful feedback to the user.

2.2.6 Future Enhancements

- **Machine Learning Models**: Incorporating machine learning models for intent recognition could significantly improve the assistant's understanding of user queries.
- **Dynamic Pattern Matching**: Implementing a system for dynamically generating or updating patterns based on user interactions could enhance the assistant's flexibility.
- **User Feedback Loop**: Integrating a feedback mechanism to learn from user interactions and refine responses over time.

2.2.7 Conclusion

This digital assistant demonstrates the basic principles of designing an AI-powered dialogue system. While the current implementation provides a foundation, there's ample scope for incorporating advanced NLP techniques and machine learning models to create a more sophisticated and intuitive

assistant. With more time and resources, the system could be enhanced to handle a wider range of user queries and provide more accurate and context-aware responses.

2.3 References

- Hyperskill. (2024). Rule-based dialogue systems | Question Answering | Main NLP tasks | NLP | Data science | Computer science | Hyperskill. [online] Available at: https://hyperskill.org/learn/step/30059 [Accessed 4 Mar. 2024].
- Uni-bamberg.de. (2024). Advanced Dialogue Systems and Conversational AI - Lehrstuhl für Sprachgenerierung und Dialogsysteme. [online] Available at: https://www.uni-bamberg.de/ds/lehre/advanced-dialogue-systems-and-conversational-ai/ [Accessed 5 Mar. 2024].
- Mondal, A. (2021). How to Build Your AI Chatbot with NLP in Python? [online] Analytics Vidhya. Available at: https://www.analyticsvidhya.com/blog/2021/10/complete-guide-to-build-your-ai-chatbot-with-nlp-in-python/ [Accessed 4 Mar. 2024].

2.4 Self Check

- Have you answered all questions to the best of your ability? Yes, we have.
- Is all the required information on the front page, is the file name correct etc.? Indeed, all the required information on the front page has been included.
- Anything else you can easily check? (details, terminology, arguments, clearly stated answers etc.?) We have checked, and everything looks good.