UNIVERSITY OF
GOTHENBURG

CHALMERS
UNIVERSITY OF TECHNOLOGY

Design of AI Systems

Assignment 7: Dialogue systems

Group 2:

Tim Boleslawsky, 980524-6478, Applied Data Science Master's, gusbolesti@student.gu.se
Mikołaj Nowacki, 20010918-2014, Applied Data Science Master's, gusnowami@student.gu.se

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.

1. Reading and Summary

The paper discusses GUS (Genial Understander System). GUS is an experimental dialogue system designed to engage in goal-oriented English conversations within a restricted domain, specifically acting as a travel agent to help clients plan simple round trips to a single city in California. The system is part of a broader research program on language understanding and serves as a progress report on the authors' work in this area.
The authors define the relevancy of GUS as being a study of the phenomena of natural dialog. GUS is not the most convenient way of booking a flight, but it is interesting to study the natural dialog. GUS attempts to model these natural dialogue phenomena, including:

- Mixed Initiative: The system allows both the user and the system to take control of the conversation, making it more flexible and natural.
- Indirect Answers: GUS can interpret indirect responses, such as inferring flight preferences from user statements.
- Anaphora Resolution: The system resolves references to previously mentioned entities (e.g., flights, dates).
- Sentence Fragments: GUS can handle incomplete sentences by inferring the full meaning based on the context of the question.
- Conversational Patterns: The system follows typical conversational conventions, such as providing minimal but sufficient information.

When talking about the organization of the program, the authors mention two key concepts. First, GUS is built using a modular approach, with separate components for morphological analysis, syntactic parsing, semantic interpretation (case-frame analysis), and dialogue management. These modules communicate through well-defined interfaces, allowing for easier debugging and integration. Second, GUS employs a multiprocess control mechanism, where tasks are placed on a central agenda and executed in a cycle. This allows the system to handle multiple processes asynchronously, such as parsing, reasoning, and generating responses.

We now have a look at how GUS processes user input and generates responses. The GUS system is built around a set of knowledge structures and processes that work together to understand and generate dialogue. These processes are organized in a modular fashion, with each module handling a specific aspect of language understanding and dialogue management. The system's processes are organized in a pipeline, where each process takes input from the previous one and produces output for the next. The processes are: Dictionary Lookup (takes user input and identifies individual words and phrases), Syntactic Analysis, (parses the input sentence and build one or more canonical syntactic structures), Case-Frame Analysis, (produces a case-frame structure that represents the semantic roles of the words in the sentence), Domain-Dependent Translation (interprets the case-frame structure in the context of the travel domain and updates the system's internal representation of the trip), Frame Reasoning (updates the system's internal frames and triggers associated reasoning processes for the output), Response Generation (uses templates to produces a natural language response)

There are two main reasoning components in the described system. First, frames are used to represent collections of information at many levels within the system. Some frames describe the sequence of a normal dialog, others represent the attributes of a date, a trip plan, or a traveler. Second, procedures are attached to slots in frames to indicate how certain operations are to be performed. These procedures fall into two general classes: servants (activated on demand) and demons (activated automatically when a datum is inserted into an instance).

Frames are not only a main focus in reasoning, they also direct the dialog. GUS assumes that the conversation will be of a known pattern for making trip arrangements. The system creates an instance of the dialog frame and goes through the slots of this instance attempting to find fillers for them in accordance with the specifications given in the prototype.

Lastly, the authors make note of the important distinction between real and realistic dialogues. To investigate this difference, we are using examples from GUS's interactions with users. While GUS can conduct realistic dialogues within its restricted domain, it struggles with the complexities of real human conversations, which often involve nuances such as politeness, indirectness, and reasoning about user motivations. First, the authors represent a realistic sample dialogue conducted with GUS. Here GUS handles mixed initiative, indirect answers, anaphora resolution, and sentence fragments effectively, demonstrating its ability to model certain aspects of natural dialogue. But when confronted with a real dialogue, several challenges were revealed that GUS could not handle. These include interpreting politeness, handling pronominal references, and understanding user motivations and reasoning.

2. Implement a simple dialogue system

2.1 Implementation

For our dialogue system we decided we wanted to cover three domains: flight, restaurant, weather. The intents we cover in these domains are: book flight, cancel flight, flight information, book restaurant, cancel reservation, find restaurant, temp tomorrow, chance of rain tomorrow, air quality tomorrow, and weather today. The basic structure of the code is that we have one class $DialogueManager$ were we define all the necessary attributes and functions and one *main* function were we run the dialogue system.

We wanted to make our dialogue system versatile and tried to implement it in such a way that it does not rely on exact keyword matching. In order to do so, we employed a Multinomial Naive Bayes classifier to recognize the domain of the user and fuzzywuzzy to match the input to an intent based on the content similarity defined by some keywords. Furthermore, we implemented frame-based dialogue management to be able to manage the information the user has already provided instead of going through the same set of questions every time.

- Multinomial Naive Bayes:
  For the system to recognize the user's domain correctly, we wanted to make the most out of the content they provided. We trained an NB classifier on 100 queries (created with the help of ChatGPT) with features "flight", "restaurant" and "weather". This solution allows us to accurately recognize the domain for a large variety of sentences. After the user has input their domain, it is vectorized into numerical representation with $CountVectorizer()$, and then given to the *predict*() function of the classifier where the domain of the input is recognized.

- $FuzzyWuzzy$ Matching:
  We employed fuzzywuzzy to search through keywords of a domain in order to recognize the specific details of an action a user might want to perform. I.e. "flight" $\rightarrow$ "book" or "cancel" or "information". The match that gets the similarity score higher than 70% is chosen as the intent domain. A huge advantage to this solution is that the bot recognizes words that are more than just a 'literal' match to the keywords.

- Dialogue Management:
  To keep track of the state of the dialogue and to be able to react to changes in the sate we define status. There are 7 different status:

  1. Start: Start conversation.
  2. Domain: Domain was recognized with multinomial NB.
  3. Intent: Intent was recognized with $fuzzywuzzy$.
  4. Awaiting further input: Analyze input with *spacy* to populate frames with data already provided by the user in the first query (restaurant name, date, etc.). If enough data has been provided in the first query, proceed with the conversation. Else, request missing data.
  5. Requesting approval: Formulate response and request approval if recognized intent is correct.

6. Awaiting approval: If intent is approved, execute intent.
7. Ready for execution: Respond to user with desired information.

In the *main* function we can check this attribute of the dialogue system to ascertain what actions must be performed next. In the various functions of the dialogue system itself we keep the status of the dialogue updated.

- Frame Entities:
  Frames allow the dialogue system to accurately fill in the information that has already been provided by the user and move on with the conversation instead of going through the sequence of the same questions every time. To manage these frames we implement the two functions *setup_frame* and *update_frame*. In *setup_frame* we define a unique frame for each intent with unique entities for each frame. An entity could, for example, be city, name, or day. In *update_frame* we fill the entities of the frame. For this we use the library *spacy*. This library gave us a readily trained model to recognize dates, names or cities which greatly reduced the complexity.

- Formulate Responses and Execute Intent: Lastly we have the two functions *formulate_response* and *execute_intent*. These functions are not really complex and basically just check the state of the dialogue system in a rule-based fashion and formulate the response or execute the intent accordingly.

2.2 Example conversations

2.2.1 Weather

Figure 1 portrays the query for the weather for the day. The information in the sentence is incomplete so the bot asks for further information about the city. After making sure the intent has been understood correctly, the bot proceeds to complete the request.

```
Welcome to the dialogue system! Type 'exit' to quit.
You:  I want to know the weather for today
Detected Domain: weather
Detected Intent: weather today
System: To give you the weather for today, I am missing the following information: ['city']
You:  Warsaw
System: So you want to know the wather in Warsaw for today? Press 'Yes' if that is correct.
You:  Yes
It is 10.35°C today in Warsaw with clear sky.
```

Figure 1: Provided partial information

```
Welcome to the dialogue system! Type 'exit' to quit.
You:  Whats the weather in Gdansk today
Detected Domain: weather
Detected Intent: weather today
System: So you want to know the wather in Gdansk for today? Press 'Yes' if that is correct.
You:  Yes
It is 4.67°C today in Gdansk with light intensity drizzle.
```

Figure 2: Provided full information

In figure 2 we can see an example conversation with data already provided in the sentence. City in the first sentence has been correctly recognized as "Gdansk".

What is worth noting is that the bot has access to an online weather API which provides real-time data for most of the cities around the world making the bot versatile with the input. (Important Note: the API key will be deactivated on the 21st of March)

In the example for the restaurant reservation shown in figure 3 we see that the input is flexible. The order in which the user inputs the information does not matter and past information is saved in frames.

```
Welcome to the dialogue system! Type 'exit' to quit.
Hi, I want to make a reservation.
Detected Domain: restaurant
Detected Intent: book reservation
System: To book a reservation, I am missing the following information: ['restaurant', 'day', 'time', 'name']
Mmh maybe on the 24th of May and at 8pm. My name is Tim but for the restaurant I am not sure.
System: To book a reservation, I am missing the following information: ['restaurant']
Let's say at Burger Barn.
System: So you want to make a reservation at the restaurant Burger Barn on the day the 24th of May at 8pm? Press 'Yes' if that is correct.
Yes
Reservation booked at Burger Barn on the day the 24th of May at the time 8pm under the name Tim.
```

Figure 3: Example Restaurant Reservation

```
Welcome to the dialogue system! Type 'exit' to quit.
You:  I want to make a reservation at Golden Dragon on 1st of April at 6 pm. My name is Mik
Detected Domain: restaurant
Detected Intent: book reservation
System: So you want to make a reservation at the restaurant Golden Dragon on the day 1st of April at 6 pm? Press 'Yes' if that is correct.
You:  Yes
Reservation booked at Golden Dragon on the day 1st of April at the time 6 pm under the name Mik.
```

Figure 4: Example Restaurant Reservation with full data

Figure 4 shows how the bot understands the input and extracts all the keywords necessary to make a reservation.

2.3 Struggles

While working on our implementation, we encountered a few particularly challenging problems. In this section, we want to outline them and describe our approach to solving them. Recognizing the domain and intent proved to be fairly straightforward. Our struggles began when trying to come up with an overall structure of the dialogue system and frames within.

For the overall structure or dialogue management, we needed to figure out a way to assert, manage, and communicate the current state of the dialogue system. What proved especially difficult was to keep this structure flexible and allow for real interaction to be mimicked. We decided on defining a ”status” attribute of the dialogue system class and track the state of the dialogue system through that. The defined status allow us to at all times recognize what the state of the conversation is and respond accordingly. But, because the flow of a real conversation is much more flexible and not easily described in fixed states, this is only a compromising solution. For example, our conversations can only be in one direction. What we mean by that is that once we state a domain, we cannot change this domain except by starting over.

As described, decided to work with frame-based dialogue management. This allowed us to be very flexible in the information handling and disregarding, for example, the order in which the user feeds the system information. But it also came with its own challenges. For example, we defined multiple entities for each frame. Now we needed to figure out a way to fill these entities for each intent. This proved to be very difficult. The main problem was that we did want to pre-specify all the cities, names, etc. the system should be able to recognize. The library *spacy* gave us a workable solution that automatically recognizes dates, names, and cities, for example. Now the idea was to make the code as concise as possible by using the same functionality for the entity city, regardless of intent. But the challenge we faced here was that the logic to retrieve such an entity might vary from intent to intent. For example, one intent could have multiple cities. How do we handle that? These issues led to very cumbersome and error-prone coding.

Another issue related to frame-based dialogue management was the question of when we check if the input is actually valid. Valid meaning for example that the user requests weather data, for which the system has information. Because we wanted reusable functions to update the entities regardless of intent, if we were to implement the validation of these entities in the same place as the updating of the entities we would need very complex code structures. Ultimately we decided to separate the "input" part of the conversation from the "execution" part to easier handle these issues. In the dialogue system this is apparent as a confirmation question followed by the actual execution of the intent and validation of the entities.

2.4 Improvements

When thinking about improvements to our dialogue system, we differentiate between improvements while keeping our overall structure and a complete overhaul. We also don't talk about obvious improvements like improving the modularization of the code and expanding the data. So if we want to keep our basic structure of machine learning methods like classification combined with rule-based methods and frame-based dialogue management, we could, for example, implement context awareness. If one were to implement a memory module, scenarios like, for example, if a user asks, "What is the temperature tomorrow?" and then follows up with "And on Wednesday?" could be handled. Another improvement we would see as very beneficial is to make the conversation more flexible. As discussed, right now we only have conversations in one direction. Through the introduction of, for example, graph-based or transformer-based dialogue state tracking, one could improve upon our rigid status-solution. Also, for named entity recognition we currently use the predefined models of the *spacy* library. A custom model that is trained to better perform on our tasks would surely be an improvement. Lastly, we would suggest the introduction of sentiment analysis or inference as a big improvement. Inference is one of the most complex aspects of human interaction, but a system that could introduce that would result in a much more human-like interaction.

Another more drastic way to improve upon our solution for the dialogue system, would be to disregard the rule-based approach and introduce deep learning methods. Instead of using handcrafted rules, one could train a sequence-to-sequence or Transformer-based model that can generate responses directly based on user input and context.