Chalmers Tekniska Högskola

DESIGN OF AI SYSTEMS (DAT410)

Module 7: Dialogue systems and question answering

Group 38

Himanshu Sahni
9812075498
sahni@chalmers.se
MPCAS

Nina Uljanić
9609134920
uljanic@chalmers.se
MPCAS

We hereby declare that we have both actively participated in solving every exercise. All solutions are entirely our own work, without having taken part in other solutions.

Hours:
Himanshu Sahni: 20 h
Nina Uljanić: 22h

1. Reading and reflection

The paper "GUS, A Frame-Driven Dia|og System" describes the first of a series of experimental computer systems that was built as part of a program of research on language understanding. GUS (Genial Understander System) is intended to engage a sympathetic and highly cooperative human in an English dialog, directed toward a specific goal within a very restricted domain of discourse. The paper highlights and elaborates on the problems of natural dialog, involving enabling both the client and the system to initiate conversation, comprehending indirect responses to queries, resolving anaphora, comprehending sentence fragments provided as answers to questions, and interpreting the dialogue in the context of recognized conversational patterns.

GUS aims to maintain control of the conversation without disrupting its natural flow. It is capable of managing simpler versions of the problem due to its limited expectations about the subject matter and the objectives of the client. Dominating position is a desired effect in interactive computer systems but until the point where it does not affect the flow of the conversation. Treating indirect answers can be done with narrow knowledge of the subject. When referring to previous statements of the conversations keywords such as dates or relative or absolute positions should be used. Sentence fragments are not part of complete sentences, questions should be used to complete sentences, asking the user to fill in the blank space. In GUS, skeletons are used for this and the blank spaces are located at the end of the sentence. Conversational strategies are adapted to the conversational patterns and the goals of the client and modular approaches are recommended to develop dialog systems.

GUS, a system for managing knowledge, includes various processes and structures such as the morphological analyzer, syntactic analyzer, frame reasoner, and language generator. These were developed as independent processes with clear communication interfaces using specific languages or data structures. Each component was tested and debugged separately and then integrated together using an asynchronous control mechanism. GUS controls its processes in a central agenda, examines, chooses the next job, and does it. It is necessary to preserve the process state because the flow in the system is not unidirectional.

A lexical analyzer analyzes an input of strings and produces a chart of syntactic and semantic information to be used by the parser. The syntactic analyzer builds syntactic structures and identifies context. The case-frame analysis involves utilizing linguistic information associated with specific words to connect how they are used in a semantic context with their appearance in typical syntactic structures. conversational patterns define the domain by passing the ground for the user's goals. Frame reasoning is used to fill in the appropriate information based on the context. The process of producing English output is directed by a query map, which comprises a collection of templates designed to handle all possible questions that could be posed to the system. The system employs a table lookup technique to locate the relevant template and generates the English output by completing the template format. The component responsible for formulating queries for the user also generates one or more outlines, which can be utilized to integrate their responses in case they are not complete sentences by themselves.

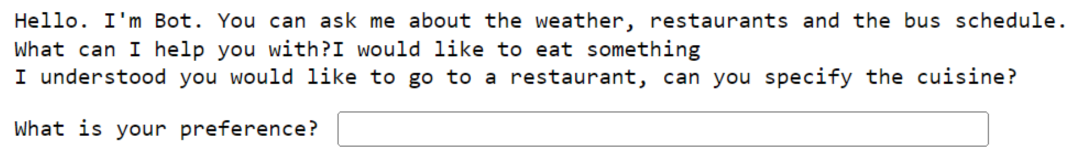
Frames serve as a means of organizing data at various levels within the system. Some frames depict the sequence of a standard dialogue, while others portray the features of a date, travel itinerary, or traveler. Essentially, a frame is a data structure that can contain a name, a pointer to a prototype frame, and a group of slots. When a frame is considered the prototype of another frame, the latter is considered an example of the former. The prototype acts as a blueprint for its examples. With the exception of the most generalized frames in the permanent database, all frames in GUS are examples of some prototypes. The majority of examples are generated during the reasoning process, although a few (such as those illustrating particular cities) are included in the initial database. The crucial components of a frame and its connections to other frames are established through its slots. A slot consists of a slot name, a value or content, and potentially a collection of associated procedures. Procedures are linked to a slot to specify the manner in which certain actions should be executed that pertain to either the slot within the provided frame or the corresponding slot in its examples. Procedures associated with slots can be classified into two broad categories: servants and demons. Demons are procedures that are triggered automatically when a piece of data is added to an example. Servants are procedures that are triggered only when requested. Frames play a crucial role in guiding the flow of conversation at various levels. At the highest level, GUS assumes that the dialogue will follow a specific pattern for arranging travel plans. It scans through the slots of this example, endeavoring to locate relevant content for them based on the requirements outlined in the prototype. Whenever a slot is filled by a new example of a frame, the slots of that example are filled in the same way. GUS follows this straightforward, recursive depth-first method, completing work on a specific slot before moving to the next. This is how GUS strives to maintain control over the conversation.

2. Implementation

In this task, we were asked to implement a simple text-based digital assistant that can help with finding a restaurant, providing the weather forecast, and finding the next tram/bus. We have implemented a digital assistant that has the three suggested functionalities i.e., for weather forecast, suggesting restaurants, and showing the bus schedule going from one place to another.

A digital assistant relies heavily on user input, as the input provides the necessary information for it to be able to function. In our implementation, all input is matched against predefined keywords using regular expressions (regex) in order to extract information.

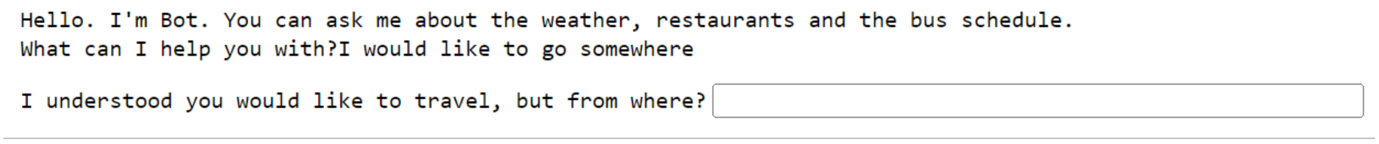
Just as with any dialogue, the digital assistant first greets the user. This serves the purpose of getting the user comfortable with the digital assistant. It then inquires of the user how it can be of assistance, notifying the user that it can assist them with either restaurant recommendations, bus schedules, or weather reports. The user inputs their request, which may contain the full information or just enough for the bot to recognize the context. The bot stores a set of keywords for each of the three contexts: 'weather' and 'temperature' for the weather reports, 'bus', 'travel' and 'go' for the bus schedule, or 'restaurant', 'food' and 'eat' for restaurant suggestions. This list could be expanded, of course, but we decided to keep it simple. If the user inputs "I would like to eat something" we assume that the user is looking for a place that serves food, i.e. a restaurant. See Fig. 1 Likewise, if the user inputs "I would like to go somewhere" we assume that the user would like to know which bus they can take to reach their (yet unspecified) destination. See Fig. 2.



```
Hello. I'm Bot. You can ask me about the weather, restaurants and the bus schedule.
What can I help you with?I would like to eat something
I understood you would like to go to a restaurant, can you specify the cuisine?

What is your preference?
```

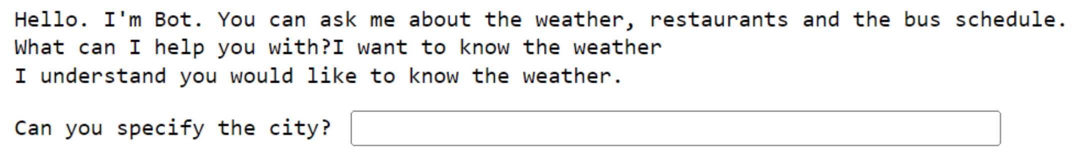
Figure 1: Identifying restaurant as context.



```
Hello. I'm Bot. You can ask me about the weather, restaurants and the bus schedule.
What can I help you with?I would like to go somewhere

I understood you would like to travel, but from where?
```

Figure 2: Identifying bus schedule as context.



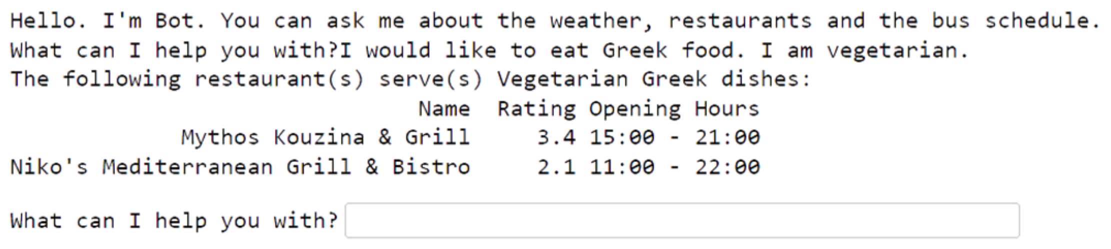
```
Hello. I'm Bot. You can ask me about the weather, restaurants and the bus schedule.
What can I help you with?I want to know the weather
I understand you would like to know the weather.

Can you specify the city?
```

Figure 3: Identifying weather as context.

Once the digital assistant knows about the context it begins to ask questions relevant to the context as defined in its functionality. It checks whether any additional information can be extracted from the user, such as the type of cuisine in the case of a restaurant, the source station in the case of a bus schedule, or the city in the case of the weather prediction, to name a few fields that those context forms hold. That is, it tries to fill the form that is specific to the context identified in the previous step. To this end, it stores a list of keywords against which the input is matched. In the case of the bus schedule, the list contains the bus stops, for the weather it stores names of cities and dates, and for restaurants, it stores the cuisine types as well as dietary options.

If the user inputs "I would like to eat Greek food. I am vegetarian." the system identifies that the cuisine is Greek and that the dietary option is vegetarian. See Fig. 4.

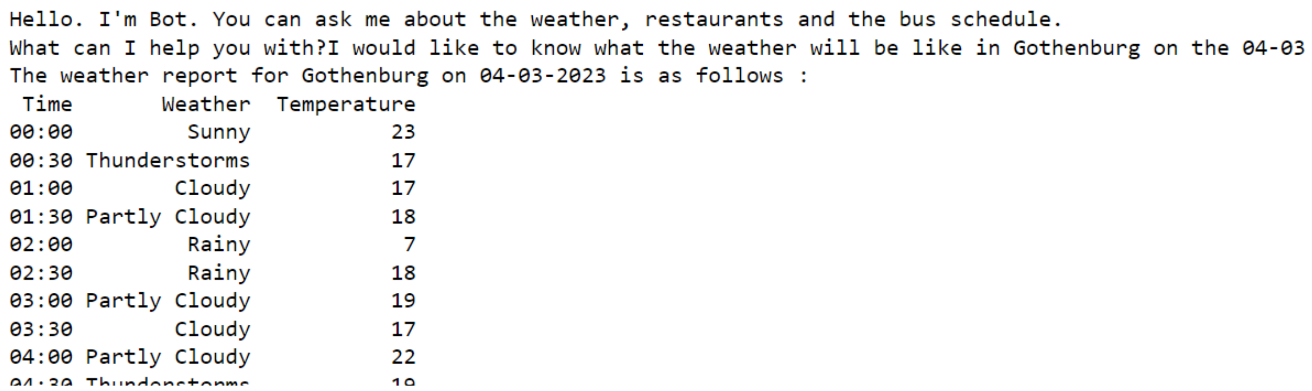


```
Hello. I'm Bot. You can ask me about the weather, restaurants and the bus schedule.
What can I help you with?I would like to eat Greek food. I am vegetarian.
The following restaurant(s) serve(s) Vegetarian Greek dishes:
                                Name  Rating Opening Hours
             Mythos Kouzina & Grill     3.4 15:00 - 21:00
Niko's Mediterranean Grill & Bistro     2.1 11:00 - 22:00

What can I help you with?
```

Figure 4: Extracting all form data from input.

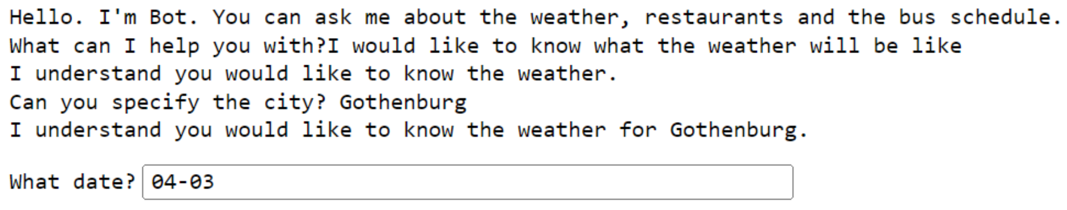
Likewise, if the user inputs "I would like to know what the weather will be like in Gothenburg on the 04-03" the system identifies that the city is Gothenburg and that the date is 04-03-2023. See Fig. 5.



```
Hello. I'm Bot. You can ask me about the weather, restaurants and the bus schedule.
What can I help you with?I would like to know what the weather will be like in Gothenburg on the 04-03
The weather report for Gothenburg on 04-03-2023 is as follows :
 Time       Weather  Temperature
00:00         Sunny           23
00:30 Thunderstorms           17
01:00        Cloudy           17
01:30 Partly Cloudy           18
02:00         Rainy            7
02:30         Rainy           18
03:00 Partly Cloudy           19
03:30        Cloudy           17
04:00 Partly Cloudy           22
04:30 Thunderstorms           19
```

Figure 5: Displaying data matches.

In case of empty slots in the form, the system continues posing questions to the user so that the remaining slots can be filled. See Fig. 6. No result will be returned if the user does not provide enough information. If all form slots have been filled, the system proceeds to the next step.

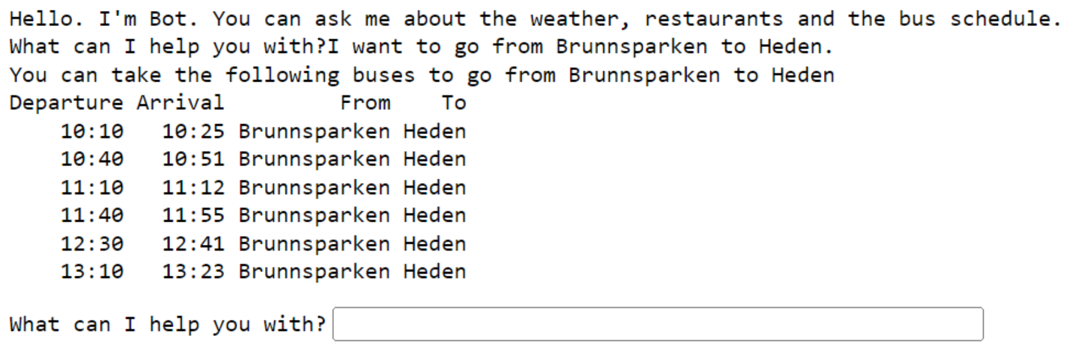


```
Hello. I'm Bot. You can ask me about the weather, restaurants and the bus schedule.
What can I help you with?I would like to know what the weather will be like
I understand you would like to know the weather.
Can you specify the city? Gothenburg
I understand you would like to know the weather for Gothenburg.

What date? 04-03
```

Figure 6: Inquiring the user for more data.

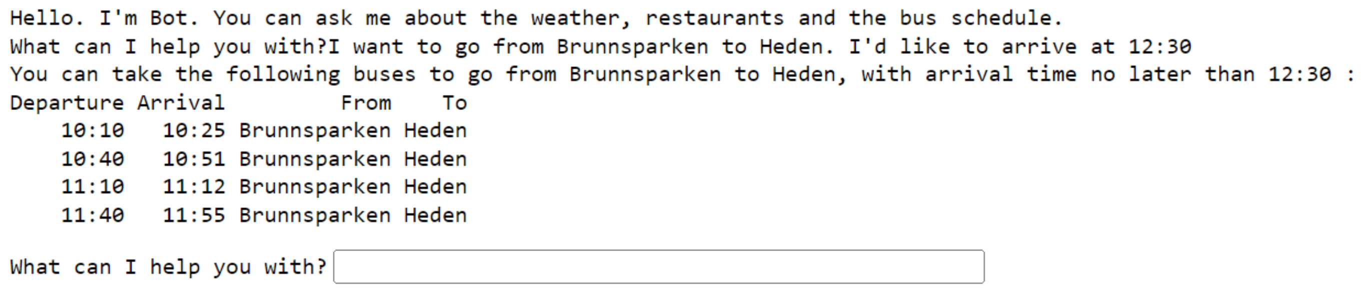
As a side note, we would like to add that, while our system has only the most essential fields in the corresponding context forms, we put a bit extra effort into the bus context in order to show the possibilities that digital assistants bring. We have made it so that the user can also specify when they would like to leave at the latest, or alternatively when they would like to arrive. See Fig. 7 and 8. By having the ability to extract more information, the conversation between the user and the digital assistant is more efficient, and the user consequently gets more concrete answers.



```
Hello. I'm Bot. You can ask me about the weather, restaurants and the bus schedule.
What can I help you with?I want to go from Brunnsparken to Heden.
You can take the following buses to go from Brunnsparken to Heden
Departure Arrival        From     To
    10:10   10:25 Brunnsparken Heden
    10:40   10:51 Brunnsparken Heden
    11:10   11:12 Brunnsparken Heden
    11:40   11:55 Brunnsparken Heden
    12:30   12:41 Brunnsparken Heden
    13:10   13:23 Brunnsparken Heden

What can I help you with?
```

Figure 7: The user places a simple request.

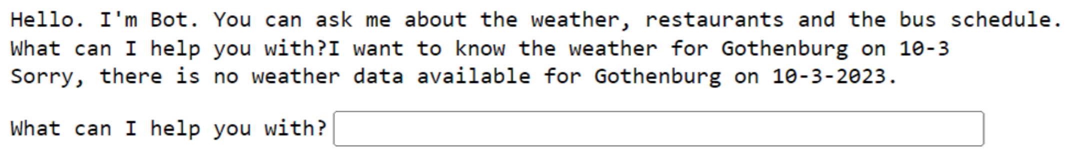


```
Hello. I'm Bot. You can ask me about the weather, restaurants and the bus schedule.
What can I help you with?I want to go from Brunnsparken to Heden. I'd like to arrive at 12:30
You can take the following buses to go from Brunnsparken to Heden, with arrival time no later than 12:30 :
Departure Arrival        From     To
    10:10   10:25 Brunnsparken Heden
    10:40   10:51 Brunnsparken Heden
    11:10   11:12 Brunnsparken Heden
    11:40   11:55 Brunnsparken Heden

What can I help you with?
```

Figure 8: The user places a complex request.

Once all form fields have been filled, the system extracts the matching data from the files that store the relevant information. In our case, the data is generated using scripts that we created ourselves. If no matches are found, it notifies the user; else it displays the found matches. See Fig. 9 for the former case.



```
Hello. I'm Bot. You can ask me about the weather, restaurants and the bus schedule.
What can I help you with?I want to know the weather for Gothenburg on 10-3
Sorry, there is no weather data available for Gothenburg on 10-3-2023.

What can I help you with?
```

Figure 9: Notifying the user that no matches have been found.

Future Work

These are some of the functionalities we would like to add if we had more time:

- Using real-time data with the digital assistant. For example, we wanted to use an API (Yr) for accessing real-time weather data for the location as asked by the user. Another good extension would be allowing the user to specify the time of the day for which they would like to know the weather.
- In the current implementation of the restaurant functionality we suggest options irrelevant of the location of the user and the restaurants. However, if we had a database with information about different restaurants in various cities, we could add the feature of suggesting the options for the desired city as specified by the user. We also wanted to add features such as booking a table for the user and suggesting the best-rated dishes of the restaurant.
- In the case of bus functionality, we wanted to use real-time bus data through an API (Västtrafik) in order to fetch the relevant data. We would also have liked to add the feature of suggesting the available buses as per the current time i.e, only the future options available based on the current time at which the user is asking the digital assistant. Due to the limitation of data and time, we could not implement these features at the present moment.

We would like to mention that the digital assistant can be built to handle more functionalities with more complex features provided access to more data is possible. It can be made to have longer conversations and handle different scenarios, ask more specific questions, etc. However, we can say that our current implementation is robust and the functionalities work properly without any bugs as discussed in the previous section above. In our opinion, if we had this assignment in the previous weeks, then we would have wanted to extend the same as our project and make it more robust with more features.

Individual Summary and Reflection

Lecture 7: Dialogue systems and question answering (2023-02-28)

Lecture summary by Himanshu Sahni

Modern systems use a combination of rules and ML-based techniques. We looked into the properties of human conversations and noted that it happens in the form of dialogues. A dialogue is a sequence of turns and the dialogue structure is usually an answer which follows the question.
The speech acts:

- Constantives - answering, claiming, confirming, etc.
- Directives - advising, forbidding, inviting, etc.
- Commisives - promising, planning, vowing, etc.
- Acknowledgements - apologizing, greeting, thanking, etc.

Chatbots: These are based on the keywords, and usually prefer responses based on the most specific keyword.

- Corpus-based chatbots - Based on very large datasets of real conversations and the response is based on the user's last turn (or two)
- IR-based chatbots - Return response to most similar user turn based on word vector or word embedding
- ML-based encoder decoder chatbot

Dialogue systems (pre-programmed to do some texts)

- Speech Recognition
- NLU (extract meaning from what it is said)
- Dialogue Manager (rule Based)
- Text to Speech Synthesis
- Natural Language Generation (learn from data that already exists)

Natural Language Understanding (NLU)

- Classify domain
- determine intent
- extract relevant information

There is a need for Dialogue Management so as to keep the conversation going and get all the information from the user. It can either be finite state-based or frame-based.

Module 6 Reflection by Himanshu Sahni

In the previous module, we implemented a tree search algorithm in order to learn the best actions to win a tic-tac-toe game i.e., the Monte Carlo tree search. It deals with random sampling and is based on experiments with randomly selected actions. It repeats the four-step process of selection, expansion, simulation, and backpropagation. We can formalize the best paths using search trees in order to be able to select the best action. An important note to consider is that there exists a tradeoff between exploration and exploitation when choosing the best-so-far action or finding a better one. Upper confidence trees (UCT) are a solution to this problem. In conclusion, a strategic approach is needed to decide future actions, and search trees along with algorithms like minimax can be used to achieve this goal.

Lecture summary by Nina Uljanić

Dialogues are a fundamental component of human communication (dia logos meaning "through language"), consisting of a sequence of turns in which individuals communicate interchangeably. Dialogues have a structure, for example, one provides an answer after being answered a question. Grounding is a way to confirm you stand on "commong round," i.e. that you understand each other.

Dialogues might follow certain patterns, which might be interrupted in order to provide additional information. The patterns can be interrupted with sub-dialogues which commonly serve as clarifications. Speech acts are used to convey meaning in a dialogue: constantives which commit the speaker to something (answering, claiming, denying, etc.), directives which attempt to make the addressee do something (advising, forbidding, inviting, etc.), comissives which commit the speak to some future action (promising, planning, etc.), and acknowledgements which express the speaker's attitude regarding the hearer (greeting, thanking, etc.).

There has been an ongoing effort to replicate this human interaction with computers, dating back to the 1960s. Chatbots are a type of dialogue system used for free conversation, which can be rule-based or data-based. Data-based chatbots use neural networks to learn from real data and provide responses.

Digital assistants use task-based dialogue systems, which consist of several components: speech recognition, natural language understanding, dialogue management, task management, natural language generation, and possibly text-to-speech analysis. Speech recognition may be statistical or neural network based. Natural language understanding is in charge of extracting meaning from the user's input, classifying the domain and determining the intent. It may be ML based. Dialogue managers that exist in practice are mostly rule-based. Natural language generation is mostly template-based, but it might be learned from delexicalized data.

Dialogue is an important part of chatbots/digital assistants because they simplify the context and meaning extraction. The user can relay necessary information to the system in many small bits instead of presenting all of it at once. For example, when one is looking for flights, one needs to input departure and arrival time, source and destination location, number of travellers, personal information, etc. A digital assistant inquires about the specific fields, taking the burden of making sure all required fields have been filled in off the user.

Dialogue systems can be finite-state or frame-based. The former has no flexibility and the system takes initiative in the conversation (resembling an interrogation) and it is useful in simple cases. The latter is more flexible because the system knows which questions it must pose in order to fill the forms. It can also be constrained to certain values.

The design of dialogue systems requires an understanding of the problem and the user. Simulations can be used to learn, and prototypes can be used for testing and verification. ML-based dialogue systems perform well for speech recognition and natural language understanding, while rule-based systems are useful for dialogue management and natural language generation.

Module 6 Reflection by Nina Uljanić

In module 6, we have been given the taks to implement a tree search in order to create a system that will be able to play the game tic-tac-toe. We have decided to use Monte Carlo Tree Search for this assignment. While I have heard of MC before, the tree search version was new to me. I have enjoyed learning about it and implementing it, although it proved to be tricky at times. The concept of tradeoff between exploration and exploitation was not unfamiliar to me and I was able to make a connection to particle swarm optimization, a stochastic optimization algorithm, which also considers these concepts. I now have a decent understanding of the algorithm and might like to extend this module's assignment to a project of my own.

Module 7 - Dialogue systems and question answering

March 7, 2023

```
[ ]: import pandas as pd
     import numpy as np
     import urllib
     import random
     import datetime
     import re
```

```
[ ]: # restaurants dataset
     df_restaurants = pd.read_csv("C:
      ↪\\Users\\Nina\\Desktop\\STUDIES\\Uni\\MPCAS22\\DAT410_DAIS\\Assignment␣
      ↪7\\New folder\\restaurants.csv")
     df_weather = pd.read_csv("C:
      ↪\\Users\\Nina\\Desktop\\STUDIES\\Uni\\MPCAS22\\DAT410_DAIS\\Assignment␣
      ↪7\\New folder\\weather.csv")
     df_bus = pd.read_csv("C:
      ↪\\Users\\Nina\\Desktop\\STUDIES\\Uni\\MPCAS22\\DAT410_DAIS\\Assignment␣
      ↪7\\New folder\\bus.csv")
     df_restaurants = df_restaurants.sort_values(by=['Cuisine', 'Rating'],␣
      ↪ascending=[True, False])

     #df_weather
     #df_bus
     #df_restaurants
```

```
[ ]: class Restaurant:
         def __init__(self):
             self.form = [['cuisine',''],['dietary options','']]
             self.cuisine_options = ['chinese', 'french', 'greek', 'indian',␣
      ↪'italian', 'japenese', 'mexican', 'thai']
             self.dietary_options = ['meat', 'vegetarian', 'vegan']
             self.cuisine_pattern = re.compile(r'\b(' + '|'.join(self.
      ↪cuisine_options) + r')\b')
             self.dietary_pattern = re.compile(r'\b(' + '|'.join(self.
      ↪dietary_options) + r')\b')

         def fill_form(self, user_input):
```

```
        # Extract cuisine
        cuisine_match = self.cuisine_pattern.search(user_input.lower())
        if cuisine_match:
            self.form[0][1] = cuisine_match.group(1).capitalize() # update␣
↪value for cuisine

        # Extract dietary options
        dietary_match = self.dietary_pattern.search(user_input.lower())
        if dietary_match:
            self.form[1][1] = dietary_match.group(1).capitalize() # update␣
↪value for dietary_options

    def check_form(self):

        # Check if cuisine is filled, else ask relevant questions
        if not self.form[0][1]:
            print("I understood you would like to go to a restaurant, can you␣
↪specify the cuisine?")
            input_string = input("What is your preference? ").lower()
            cuisine_match = self.cuisine_pattern.search(input_string)
            if cuisine_match:
                self.form[0][1] = cuisine_match.group(1).capitalize()
            else:
                while self.form[0][1] == '':
                    print("I did not quite understand, I can help you with the␣
↪following options")
                    print(', '.join(self.cuisine_options))
                    input_string = input("What is your preference? ").lower()
                    cuisine_match = self.cuisine_pattern.search(input_string)
                    if cuisine_match:
                        self.form[0][1] = cuisine_match.group(1).capitalize()

        # Check if dietary options is filled, else ask relevant questions
        if self.form[1][1] == '':
            input_string = input("I understood you would like to find a␣
↪restaurant which serves " + self.form[0][1] + " cuisine. Could you tell␣
↪about your dietary options? ").lower()
            dietary_match = self.dietary_pattern.search(input_string)
            if dietary_match:
                self.form[1][1] = dietary_match.group(1).capitalize()
            else:
                while self.form[1][1] == '':
                    print("I did not quite understand, Would you prefer␣
↪Vegetarian, Vegan or Meat")
                    input_string = input("What is your preference? ").lower()
                    dietary_match = self.dietary_pattern.search(input_string)
                    if dietary_match:
                        self.form[1][1] = dietary_match.group(1).capitalize()

            self.recommend_restaurants(df_restaurants)

        def recommend_restaurants(self, df):
            if self.form[0][1] == '' or self.form[1][1] == '': # handle as no␣
↪preference -> return all
                print("I need more information before I can recommend a restaurant.
↪")
                return


            result = df.loc[(df['Cuisine'] == self.form[0][1]) & (df['Dietary␣
↪options'] == self.form[1][1].capitalize())]
            result = result[['Name','Rating','Opening Hours']].
↪sort_values('Rating', ascending=False)
            if result.size == 0:
                print(f"Sorry, there exist no restaurant that serves {self.
↪form[1][1]} {self.form[0][1]} dishes.")
            else:
                print(f"The following restaurant(s) serve(s) {self.form[1][1]}␣
↪{self.form[0][1]} dishes:")
                print(result.to_string(index=False))
```

[ ]:
```
class Weather:
    def __init__(self):
        self.form = [['city',''],['date','']]
        self.cities = ['stockholm', 'gothenburg', 'malmo', 'uppsala', 'lund',␣
↪'umea', 'boras', 'halmstad']
        self.days = [str(i) for i in range(1, 32)]
        self.months = [str(i) for i in range(1, 13)]
        self.city_pattern = re.compile(r'\b(' + '|'.join(self.cities) + r')\b')
        self.date_pattern = re.compile(r'(\d{1,2})-(\d{1,2})(-\d{2,4})?$')#␣
↪DD-MM or DD-MM-YY or DD-MM-YYYY

    # if possible, fill form with user's first input
    def fill_form(self, user_input):

        # Extract city
        city_match = self.city_pattern.search(user_input.lower()) # check match␣
↪in whole string
        if city_match:
            self.form[0][1] = city_match.group(1).capitalize()

        # Extract date
        self.match_date(user_input)

    # however, in case the user did not provide enough info.. inquire!
    def check_form(self):
        if not self.form[0][1]:
            print("I understand you would like to know the weather.")
            while not self.form[0][1]:
                city = input("Can you specify the city? ").lower()
                if city == 'exit':
                    break
                if city in self.cities:
                    self.form[0][1] = city.capitalize()
                else:
                    print("I did not quite understand. Here are the cities I␣
↪can help you with:")
                    print(", ".join(self.cities))

        if not self.form[1][1]:
            print(f"I understand you would like to know the weather for {self.
↪form[0][1]}.")
            while not self.form[1][1]:
                date = input("What date?")
                if date == 'exit':
                    break
                res = self.match_date(date)
                if res:
                    break
                else:
                    print("I did not quite understand.")

        # now that the form has been filled, collect the data and present to␣
↪the user
        self.report_weather(df_weather)

    def report_weather(self, df):
        result = df.loc[(df['City'] == self.form[0][1]) & (df['Date'] == self.
↪form[1][1])]
        result = result[['Time','Weather','Temperature']]
        if result.size == 0:
            print(f"Sorry, there is no weather data available for {self.
↪form[0][1]} on {self.form[1][1]}.")
        else:
            print(f"The weather report for {self.form[0][1]} on {self.
↪form[1][1]} is as follows :")
            print(result.to_string(index=False))
    def match_date(self, input):
        date_match = self.date_pattern.search(input.lower())
        if date_match:
            date = ""
            day = date_match.group(1)
            month = date_match.group(2)
            year = date_match.group(3)
            if year is None:
                date = f"{day}-{month}-2023"
            elif len(year) == 2:
                date = f"{day}-{month}-20{year}"
            elif year is None and month is not None and day is not None:
                date = date_match.group()
            else:
                print("Invalid date.")
                return True # invalid
            self.form[1][1] = date
            return False # valid
```

```
[ ]: class Bus:
    def __init__(self):
        self.form = [['from','',''],['to','','']]
        self.stops =['brunnsparken', 'saltholmen', 'heden', 'lindholmen',␣
↪'kallebäck']
        self.from_stop_regex = re.compile(r'(?<=from\s)(?:{})'.format('|'.
↪join(self.stops)))
        self.to_stop_regex = re.compile(r'(?<=to\s)(?:{})'.format('|'.join(self.
↪stops)))
        self.from_time_regex = re.compile(r'(?<=leave\sat\s)(\d{1,2}:\d{2})')
        self.to_time_regex = re.compile(r'(?<=arrive\sat\s)(\d{1,2}:\d{2})')

    # Extract stops
    def fill_form(self, user_input):
        from_stop_match = self.from_stop_regex.search(user_input.lower())
        to_stop_match = self.to_stop_regex.search(user_input.lower())
        from_time_match = self.from_time_regex.search(user_input.lower())
        to_time_match = self.to_time_regex.search(user_input.lower())

        if from_stop_match:
            self.form[0][1] = from_stop_match.group().capitalize()
        if from_time_match:
            self.form[0][2] = from_time_match.group()
        if to_stop_match:
            self.form[1][1] = to_stop_match.group().capitalize()
        if to_time_match:
            self.form[1][2] = to_time_match.group()
    # Check if form is filled, otherwise, ask for more information
    def check_form(self):
        # Check if "from" is filled
        if self.form[0][1] == '':
            word = input("I understood you would like to travel, but from where?
↪").lower()
            if word in self.stops:
                self.form[0][1] = word.capitalize()
            else:
                while self.form[0][1] == '':
                    word = input("I did not quite understand, could you specify␣
↪the location that you want to travel from?").lower()
                    if word in self.stops:
                        self.form[0][1] = word.capitalize()

        # Check if "to" is filled
        if self.form[1][1] == '':
            word = input("I understood you would like to travel from " + self.
↪form[0][1] + " but to where?").lower()
            if word in self.stops:
                self.form[1][1] = word.capitalize()
            else:
                while self.form[1][1] == '':
                    word = input("I did not quite understand, could you specify␣
↪the destination?")
                    if word in self.stops:
                        self.form[1][1] = word.capitalize()

        self.plan_trip(df_bus)

    def plan_trip(self, df):
        result = df.loc[(df['From'] == self.form[0][1]) & (df['To'] == self.
↪form[1][1])].sort_values('Departure')
        if result.size == 0:
            print(f"Sorry, there are no available connections leaving from␣
↪{self.form[0][1]} and going to {self.form[1][1]}.")
        else:
            answer = f"You can take the following buses to go from {self.
↪form[0][1]} to {self.form[1][1]}"
            if self.form[0][2] != '':
                result = result[result['Departure'] < self.form[0][2]]
                answer += f", with departure time {self.form[0][2]} at latest :"
            elif self.form[1][2] != '':
                result = result[result['Arrival'] < self.form[1][2]]
                answer += f", with arrival time no later than {self.form[1][2]}␣
↪:"
```

```
            print(answer)
            print(result.to_string(index=False))
```

```
[ ]: #Method to extract context from intitial question
def extract_context(sentence):
    weather_pattern = re.compile(r"(weather|temperature)") # key_words =␣
 ↪['weather','sunny','rain','cloudy','windy','hot','cold','temperature']
    bus_pattern = re.compile(r"(bus|travel|go)") # key_words =␣
 ↪['bus','tram','bus stop','travel','train', 'transport']
    restaurant_pattern = re.compile(r"(restaurant|food|eat)") # key_words =␣
 ↪['restaurant','restaurants','food','hungry', 'dish', 'sushi', 'hamburger',␣
 ↪'pizza','cuisine']

    context = ''
    if weather_pattern.search(sentence):
        context = Weather()
    elif bus_pattern.search(sentence):
        context = Bus()
    elif restaurant_pattern.search(sentence):
        context = Restaurant()
    return context
```

```
[ ]: #Main chatbot method
def get_help():
    print("Hello. I'm Bot. You can ask me about the weather, restaurants and␣
 ↪the bus schedule.")
    end = False

    inputString = None
    while inputString != 'exit':
        inputString = input("What can I help you with?").lower()
        if inputString == 'exit':
            break
        context = extract_context(inputString)
        context.fill_form(inputString)
        context.check_form()
    print("Goodbye. Have a nice day!")

get_help()
```