SHRI Project Report

Andrea Caciolai
1762906

February 12, 2020

Contents

1 Introduction

This is the report of the work I have done for the project of the Spoken Human Robot Interaction module of the course in Artificial Intelligence (A.Y. 2019/2020).

The goal of the project is the implementation of a *task-oriented Spoken Dialouge System* (*SDS*).

This system relies on a pipeline consisting in Automatic Speech Recognition (ASR), Syntax Analysis to obtain a dependency parse of the transcribed sentences, finally Dialogue Management through slot filling of a set of frames, based on the parsed sentences from the user.

Among the proposed scenarios for the task of the bot, I chose to implement the "Waiter Robot", which is capable of representing a menu, adding entries to it, answering questions about it and handling an order, all via spoken interaction.

2 Implementation

In this section I will describe the implementation of the project.

2.1 Resources

I chose *Python3* for the implementation language, as suggested by Professor Nardi.

I chose *Google Speech Recognition* library for handling the speech-to-text and ASR processes, the first component of the pipeline.

I chose *spaCy* Natural Language Processing library for handling the dependency parsing, the second component of the pipeline.

The third component of the pipeline (dialogue management through slot filling of frames) has been handled all by myself since I asked for a FrameNet license to use their software to handle the slot filling, but I received an answer when I had almost completed the project, end even if their solution is (of course) much more powerful, flexible and elegant than mine, I decided to keep my own implemented solution for practical reasons.

Finally, I chose *pyttsx3* for handling the text-to-speech process.

2.2 Project Structure

The project comprises several files for modularization and readability, but it revolves around two main modules: *bot.py* and *nlp.py*, and the handling of

the bot *knowledge base*, which for convenience I will refer to as just "menu".

In the following, I will briefly describe all the files of the project.

2.2.1 Main

It is the main file of the project. Here the bot is initialized, and the interaction takes place in the form of a *while loop* in which the user speaks, the audio is recorded, transcribed and passed on to the bot, that processes the resulting command.

2.2.2 Exceptions

Here I defined a few custom exceptions that help me in enforcing the consistency of the menu.

2.2.3 Utils

Here I put some utility functions, mainly used to pretty print information about the output of the parsing process (the dependency tree).

2.2.4 Speaker

Here I defined a class that handles the text-to-speech process, using the *pyttsx3* library.

2.2.5 Listener

Here I defined a class that handles the speech-to-text process, using the *SpeechRecognition* library. The *listen* method records audio from the microphone, then tries to recognize a sentence from it, and returns an object with a success flag, an error (*None* if successful), the transcribed sentence (*None* if not successful).

2.2.6 Frames

Here I defined the *Frame* class, which gets extended by the different frames I used to encode the different user intentions the bot would need to be able to handle. In particular, those are:

- AddInfoFrame to handle the intention to add information about the menu.

- AskInfoFrame to handle the intention to ask information about the menu.
- OrderFrame to handle the intention to order.
- EndFrame to handle the intention to stop the interaction.

Each frame has its own type of slots storing information useful to satisfy the appropriate interaction required by the respective user intention being handled.

For each frame, I also define a method to recognize if a parsed sentence triggers that frame, a mechanism that I use to determine user intention in the first place, which I explain better in Section 2.2.8.

2.2.7 Nlp

Here I defined all the methods performing the NLP processing needed to parse, explore and retrieve words and lemmas from a given command.

Listing 1: *syntax_analysis* method

```
1 def syntax_analysis(sentence):
2     nlp = spacy.load(model_en)
3     doc = nlp(sentence)
4     parsed = list(doc.sents)[-1]
5     return parsed
```

The *parsed* object is a tree-like object in which each node is a token from the tokenized sentence, and represents the dependency tree of the sentence. Each node carries information about its dependency relation, its original word (or more generally, text), the corresponding lemma and *POS* tag and so on.

Starting from this object, the bot (Section 2.2.8) does all the reasoning needed to detect user intention and perform slot filling.

2.2.8 Bot

This is the core of the project, as one can imagine.

The Bot class makes use of all the classes and methods described earlier, to go from raw audio from the microphone to an "intelligent" reply provided through the speakers.

The bot main attribute is its current *frame* that is handling and the *stack* of frames that it has not finished handling yet. This way it can interrupt taking an order without losing progress, to answer a question about the menu, and then go back to taking the order.

Listing 2: *process* method

```
1 def process(self, command):
2     # ...
3     parsed = syntax_analysis(command)
4     # ...
5     frame = self._determine_frame(parsed)
6     # ...
7     # change current frame if necessary, storing old
          one
8     # obtain reply by handling parsed command based on
          the current frame
9     # ...
```

The *process* method outlined in Listing 2 is the core of the bot, in which all the other methods are triggered. It receives a command, resulting from the ASR processing previously done on the raw audio from the user, then it determines the user intention, and therefore the appropriate frame, finally elaborating an appropriate reply.

The user intention detection is based on a fairly simple but effective (for the very simple task the bot has to carry out) mechanism. For each frame the needs to handle, I defined some *triggers*, i.e. particular words that when found in the dependency tree of the parsed command in a particular dependency relation, trigger the frame. For each command received, the frame which gets the largest amount of triggers is determined to be the appropriate one.

If this frame is of the same type of the one currently handled by the bot, then no change happens, otherwise, the current frame is stored in the frame stack and the new one replaces it for the time being. As soon as the current frame gets handled completely, the old one is popped out of the stack and restored to be completed as well.

The bot stores information in its *menu*, which is a knowledge base consisting in a dictionary with a list of entries each carrying information about the course it belongs to, among the available ones (starter, main course, side dish, dessert, drink).

Listing 3: *menu* structure

```
{"entries": [
    ...
    {"name": "pizza", "course": "main course"},
    {"name": "pasta", "course": "main course"},
    ...
    {"name": "coke", "course": "drink"}
]}
```

The menu follows the *JSON* format, so that it can saved to disk and loaded aftwerards with ease.

3 Conclusions

This project uses a very simple approach in order to implement a task-oriented SDS. In fact, there is room for some improvements, for example:

- Have more frames for handling more situations and have them more complex and flexible to handle those situations better and more comprehensively.
- Devise a better user intention detection mechanism to match a command to the appropriate frame. I think an approach based on machine learning would be a much more viable solution: for instance a classification problem in which a dependency tree of some command needs to be assigned to the correct class, which is one of the possible frames. In fact, I had to think beforehand of "all" the possible ways I could phrase a certain command, and then make sure that their resulting dependency trees were elaborated correctly by the pipeline. This is certainly not a scalable solution.

Even if the project is very simple and could be improved a lot, it shows the entire pipeline of a common Spoken Dialogue System, and I have learned much by working on it.