Artificial Intelligence 2
Spoken Human-Robot Interaction

Simone Tedeschi, Sveva Pepe, Marco Pennese

Master's Degree in Artificial Intelligence and Robotics

Department of Computer, Control and Management Engineering

Sapienza University of Rome

January 2020

Contents

1 Introduction

The goal of the project is to build a task oriented Spoken Dialogue System (SDS). In order to do this there are different steps to follow:

- Install and use the tools for speech processing, dependency parsing and text-to-speech. For the speech processing we have used Google Speech Recognition, then we used StanfordNLP as dependency parser and the python library pyttsx3 for the text-to-speech process.
- Implement a program that acquires a spoken sentence, calls the ASR (Automatic Speech Recognition) to get the corresponding text, then calls the Dependency Parser and prints the graph. Therefore, test the correct operation of the above mentioned pipeline.
- In the end, implement the task oriented dialogue in a specific scenario, by identifying a set of frames and implementing a dialogue finalized to slot filling. In our application the scenario is the space, in particular planetary systems, in which the agent (that we have called **AstroBot**) is able to learn new things about them and also answer to queries. Further on more details about frames structure will be given.

2 Implementation

The structure of the project is divided in three main modules: the Listener, the Agent and the Speaker. In this chapter we will see in detail how they work and we'll understand the general behaviour of the program.

2.1 Listener

The first thing that the AstroBot says when the program starts is *"Hi, How can I help you?"* and then, obviously, the Listener module is invoked.

The aim of this class is only to acquire commands from the microphone and store these commands in a string in such a way that then, other modules, could use it. To acquire informations from the microphone we have used the above mentioned Automatic Speech Recognition provided by Google.

Moreover, to incentivize the user to speak, and for making the software more interactive, the program shows the message *"Say something..."*, listening until some command is received.

The following fragment of code shows what just described:

```
def _listen(self):
  r = sr.Recognizer()

  with sr.Microphone() as source:
    print('Say something...')
    r.pause_threshold = 1
    r.adjust_for_ambient_noise(source, duration=1)
    audio = r.listen(source)

    command=""
    try:
      command = r.recognize_google(audio).lower()
      ret = self.OK

    except sr.UnknownValueError:
      ret = self.UNKNOWN_VALUE

  return command, ret
```

Listing 1: The function that actually listens to the user and returns a string of text

This function is called inside a "while" loop, that allows the agent to listen for consecutive messages when something goes wrong in the acquisition of the command.

2.2 Agent

When the command is acquired, it is passed to the Agent module that will process the information through two main steps that will be analyzed in this chapter.

2.2.1 Check

The first thing that the agent do is to check whether the received message matches with his capabilities, namely if the message matches with the templates that agent knows.

Templates are composed by regular expressions that allow the agent to "classify" the messages in different categories. For example when the message is of the type *"What is ..."*, he understands that is a query. On the other hand, when the sentence has a declarative form such as *"The temperature of Mars is ..."*, he becomes immediately ready to acquire new informations and asks you to confirm what you said before the overwrite of the database.

Now there are two possibilities: either the command matches with one of his templates or the command his unsolvable for the agent. In the first case the process goes on with the next step of the pipeline, in the second one he answers *"Probably I didn't understand what you said"* and he starts to listen for a new command.

Now what the agent has to do is to understand at which class the message belongs to and behave consequently. Suppose that the message is *"Solar is a system"*. He searches between his templates and sees that this sentence belongs to category "tell_system", as shown below:

```
templates = {
  ...
  "tell_system":["(?P<name>[a-z]+) is a system"],
  ...
}
```

Each category is composed by a list of several templates, so that the user has a set of possibilities to ask for the same command to the bot.

2.2.2 Act

If we are here it means that the message said by the user matches with one of the templates of the agent.

Once he recognized the category, he calls the *"act"* function in which he acts differently depending on the category of the message. Continuing the previous example he will create a frame in his knowledge base, associated to the solar system with an empty list of planets and an unknown main star. These fields of the system can be filled with subsequent commands of the type *"Earth is a planet of solar system"* or *"Sun is the star of solar system"* and so on.

Below is shown the handling of the command taken as example:

```
def act(self, key, pattern, command):
  ...
  elif key=="tell_system":
    m = re.match(pattern, command)
    name = m.group("name")
    self.kb["systems"].append({"system":name,"star":None,
                               "planets":[]})
    self.say_ok()
  ...
```

For completeness we say that once a planet is added to a system it has, in turn, many other attributes that can be filled that are: radius, mass, orbit_time and temperature. For example to add the temperature of Mars, you have to say *"The temperature of Mars is 210 degrees"* and similarly for other attributes.

There are two more types of commands, in addition to those used for asking or telling informations, that are:

- **save**: when this command is pronounced the agent stores all the notions acquired in the current session in his knowledge base;

- **exit**: when this command is pronounced the agent says *"Bye bye"* and quits the application.

Moreover, when the AstroBot is enough "cultured", you can have fun asking him questions about the things that he learnt. For example, more advanced things that you can ask (w.r.t. what we have seen in the examples) are:

- *What is the largest/hottest/fastest* (respectively *smallest/coldest/slower*) *planet of Solar System?*
- *What do you know about Solar System?* (or even a single planet)
- *What do you know?*

In the last case he will answer saying whatever he knows about systems, planets of those systems and informations about planets of those systems.

2.3 Speaker

The last module, that is invoked when the agent has something to say to the user, is the speaker. Returning to the example seen in section 2.2.2, in the act function, the agent calls another function called *say_ok()*. In the agent module we defined several functions like this, that are *say()*, *say_not_known()*, and others, that do nothing more than invoking the speaker module passing him a specific string to say (using the *speaker.speak()* method).

The speaker module is able to convert the written sentence selected by the agent into a vocal sentence. It has been implemented using the TextToSpeech python library (shown in the introduction 1).

Below is displayed the code of what we have just said:

```
1  import pyttsx3
2
3  class Speaker:
4    def __init__(self):
5      self.engine = pyttsx3.init()
6      self.engine.setProperty('volume', 10.0)
7      self.engine.setProperty('rate', 125)
8
9    def speak(self, sentence):
10     self.engine.say(sentence)
11     self.engine.runAndWait()
```

The init function simply set the volume and the rate of the voice.

3 Knowledge base

Another core part of the project, which allows a great part of the operations described in Chapter 2, is the Knowledge Base or KB. We modeled the KB as a non-relational database (in the specific case as a json file) in order to be fast in reading and writing the KB and also because it is very easy to manage. In fact, inside the Python code the KB can be easily represented as a dictionary with couples *key-value*.

Here are defined methods that we use to create, load and save knowledge bases. It is possible that inside the folder there are more versions of the knowledge base. By default, when the program starts, it loads the most recent one. This is possible because whenever the user pronounce the command *save* (explained in section 2.2.2) the bot saves the new KB with a timestamp.

Of course it is possible to load a custom knowledge base specifying its name form the command line (with the argument **- -kb_file**).

The structure of the knowledge base is the following:

- **systems**: they are the outermost boxes, so the most general objects that can be added to the KB, and they indicate planetary systems like *solar system*. Every system contains a list of planets and the main star of the system;
- **planets**: they are sub-objects of systems, but they have their own attributes: name, radius, mass, orbit_time and temperature;
- **star**: it is the main star of a system (for example Sun for Solar System).

Initially the knowledge base has only an empty list of systems and then it is populated by teaching new things to the AstroBot.

Below is represented an example of KB where the agent knows only that Solar System is a system and that the main star is the Sun:

```
1 {"systems": [{"system": "solar", "planets": [], "star": "sun"}]}
```

4 Dependency Graph

It is possible to print and see the dependency tree of every pronounced sentence. By default it is disabled, but it can be enabled with the argument **- -dep_tree True**.

We use the **stanfordnlp** model in order to create the dependency tree. It is based on a neural network trained on english sentences and it is able to find connections between words of the sentence.

5 Conclusions

This project uses a very simple approach in order to implement a system based on vocal interaction. Despite this, it is possible to make some improvements:

- The speech-to-text part is left to the Google Web Speech API. This service is very good in the testing phase but generally it is not a good idea to use it in production because you are limited to only 50 requests per day and there is also the chance that Google may revoke it at any time.
- In order to match the correct class of a query, an approach based on machine learning can easily improve the agent in terms of the number of possible sentences that the agent can handle.

Even if the project is very simple it shows the entire pipeline of a common Spoken Dialogue System.