Artificial Intelligence

Natural Language Processing

Task-oriented SDS implementation for a cleaning robot

Franco Pestarini
<fpestarini {at} gmail {dot} com>

Simone Ferrari
<ferrari.1645086 {at} studenti {dot} uniroma1 {dot} it>

Kai Niu
<niukai1995 {at} gmail {dot} com>

January 22, 2019

1 Introduction

Modern task-based dialog systems use the frame-based architecture[5] in which the designer specifies a domain ontology. There are a set of frames with information that the system is designed to acquire from the user, each consisting of slots with typed fillers.

In order to get these slots, there is a control structure in the dialog system, which knows which question the user is answering. And ask a new question for more information based on the user's reply.

The control systems can be divided into several categories depending on whether they are system or user controlled regarding who takes the iniative in the dialog. There is a mixed control system called mixed initiative, in which the user can reply some irrelevant answers. The user can talk without fully following what the system asks and the system can automatically skip irrelevant answer and update the filled slots with relevant answers.

Because this domain ontology system has finite states, we can extract slots from users' input by many simple rules. That means we can simplify the language understanding procedure using the dependence tree structure and regular matching. For example, we should take compound words, conjunction words, negative words into account.

With this in mind we set up to implement such an agent using different tools as we will describe in the next section.

2 Approach

2.1 Speech-to-text and text-to-speech

One of the first things we did was to study and install the recommended tools for achieving the goals of the project. Being a dialogue system we started with the tools necessary for the interaction of the agent with the real world. Such as automatic speech recognition (ASR) and text-to-speech (SAY).

As ASR we first tested the Sphinx Recognizer[1] because according to its documentation it offers the possibility of offline use. Then we tried the Google Speech Recognition library[2] which proved to be much more efficient and accurate than the previous one. It requires an Internet connection and gives the option of translating audio into multiple outputs, each associated to a certain confidence. In the end we opted for this one.

English is the language we chose to do the project. We downloaded high quality audio files from the Internet to compare the performance of the two ASRs. The next step was to use the microphone of our laptop to generate the audio to be the input of the Google Speech Recognizer. To this end we installed PyAudio[3], a cross-platform audio input/output stream library.

For the text-to-speech part we simply installed Espeak[4], which is a software speech synthesizer for English and some other languages.

Using together Google ASR and Espeak made possible to create a test which has as a task to listen (using the microphone), convert audio into text and then reply it back (using speakers).

The approach was to first adjust the attribute *energy threshold* of the Recognizer, which represents the energy level threshold for sounds. Values below this threshold are considered silence, values above are considered speech. This tuning process can be done either by hand, trying different values, or it can be done using the method to *adjust for ambient noise*. We opted for the second one which is automatic.

After this adjustment, the Recognizer's listen method can be called. It either records audio for a limited amount of seconds or it records until you stop talking. The audio is then converted into text and, if it is recognized, the system replies it back using speakers.

2.2 CoreNLP

Stanford CoreNLP[6] is a well-known Natural Language software that provides a set of tools for language recognition. It offers wide variety of functionalities. For our project we will mainly focus on the syntactic dependencies generated by using the dependency parser. Also, we used the token extraction (tokenization) which preprocessing is extremely useful (*e.g.* it filters `cannot` as `can` and `not` separately).

The software is implemented in Java and can be used from Python using a wrapper to communicate with the program. It acts as a server where we can submit different sentences and analyzed them. The nature of this server-client communication makes the communication really slow. It is not a problem for our project but it should be used natively if fast and efficient responses are needed.

We tested the Python wrapper with a few of simple examples to experiment the different functionalities offered by the program. For example, we tried the named entity recognizer (NER) which labels the words recognized as entities in a given sentence, like names, days of the week, etc. It worked fine with simple examples but this particular tool was very slow and sometimes it made out program crash unexpectedly. We reproduced this behaviour in our computers, so we decided not to use it.

2.3 Cleaning Agent

After setting up the whole chain from speech-to-text (and text-to-speech) to CoreNLP we were ready to start the project itself.

We started by choosing the domain for our agent. We chose the **Cleaning Robot** and defined a frame, slots and values for this particular kind of robot. We decided to use only one frame but to have several slots with multiple values to fill. Our next step was to think of the dialogue graph that we expected given our choice of frame. As we saw, with this type of interaction we can have a *mixed initiative* environment, we encouraged this with our agent by starting the conversation after some time if the user did not provide any input.

To illustrate the dialogue that we expect to have we can see *Figure 1*. We drew a simple finite-state automaton architecture for our frame-based dialog. The flow of questions is, of course, related with the slots we defined. Each question has as a goal to fill at least a particular slot. But we are capable of filling many slots at a time if the user submits them.

Even as we decided to use only one frame, our implementation allows the addition of any new frames in the future. We paid a lot of attention to this kind of things and tried to implement our agent in a modular way. This was a challenge as we divided the work to do and wrote different parts of the code separately.

The slots that we need to fill are the following: `room`, `mode`, `day`, `time` and `permanent`. The first one is obvious, it refers to a room of the house to clean. It is possible to mention more than one room at a

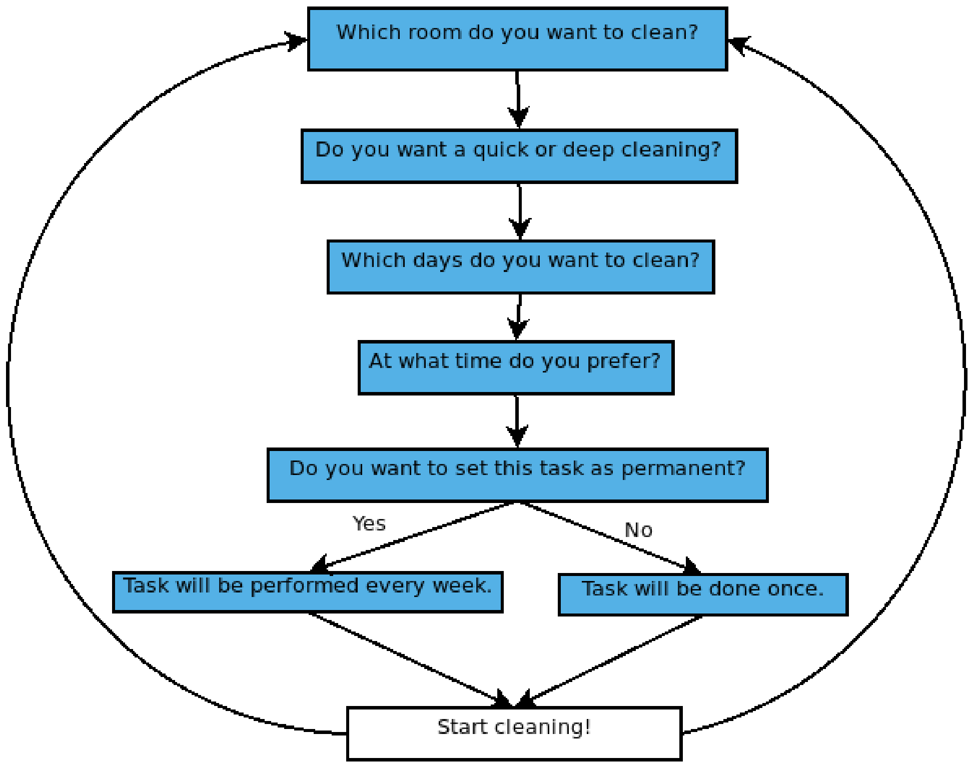

Which room do you want to clean?
Do you want a quick or deep cleaning?
Which days do you want to clean?
At what time do you prefer?
Do you want to set this task as permanent?
Yes
No
Task will be performed every week.
Task will be done once.
Start cleaning!

Figure 1: Dialogue graph scheme.

time. The second one depends on the type of cleaning the user wants, a *fast* or a *in-depth* clean. The user can use different words to say this, for example *quick* or *deep cleaning*. The day and time are also obvious. Several options can be submitted, like mentioning two days or just saying *every day* to do it daily. The last slot contains a *yes/no* value which indicates the agent to save the task. For example, if the user specifies *every Monday*, then we fill the `day` slot as well as the `permanent` slot as *yes*, meaning that it is a task to execute each Monday.

After getting the frames and slots defined we moved on to set up the whole structure of our agent. In *Figure 2* we can see a full description of the control flow we implemented. We take the user's voice as input and transfer the voice into text by the Speech Recognition API. Because of the finite slots, we only use the dependence relationship produced by Stanford NLP tools. Then we check which slot is still empty and ask for more information from the user and continue with the same loop.

3 Results

On the final pages we displayed a few examples to illustrate the behaviour of our agent. The first example shows how we fill a particular slot using the dependency tree of the sentence. And the last two examples show a full run of the program, first filling all the slots with one sentence and then filling them one by one.

Due to the fact that CoreNLP takes a lot time to run we came up with a few test files to be able to test our slot-filling rules quickly. We created a file with a lot of test sentences and then generated the tokens and trees for all of them. Once we had this we use that static information to test our new rules on them. Having this simple tool enabled us to move faster to complete the project.

We also added an option to disable the microphone and select the keyboard as input. This was another way to test the dialogue graph that made it easier and more efficient to test.

We also thought about possible ways to continue our work. One natural extension is to add a second frame for updating and deleting permanent tasks. For example, if the have a task to perform every

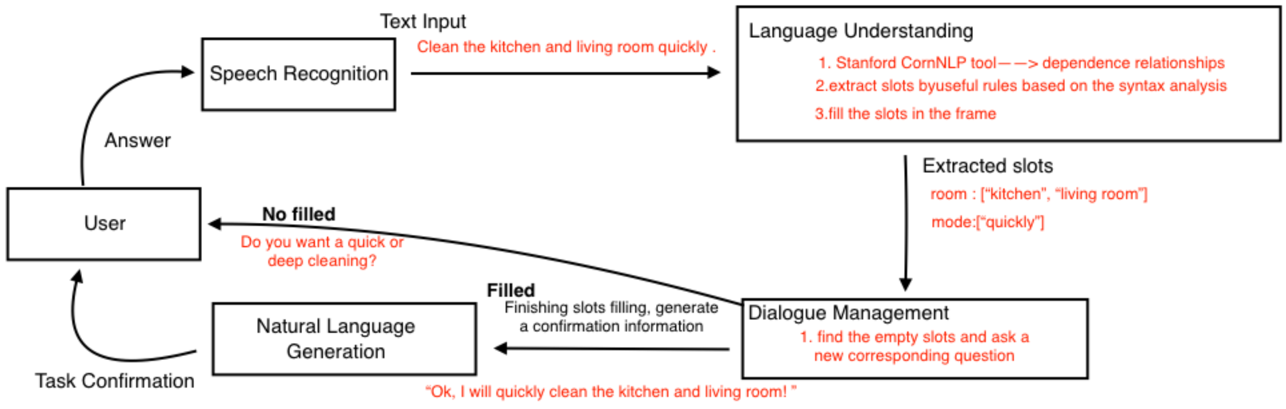

Text Input
Clean the kitchen and living room quickly .
Speech Recognition
Language Understanding
1. Stanford CornNLP tool——> dependence relationships
2.extract slots byuseful rules based on the syntax analysis
3.fill the slots in the frame
Answer
Extracted slots
room : ["kitchen", "living room"]
mode:["quickly"]
User
No filled
Do you want a quick or deep cleaning?
Filled
Finishing slots filling, generate a confirmation information
Natural Language Generation
Dialogue Management
1. find the empty slots and ask a new corresponding question
Task Confirmation
"Ok, I will quickly clean the kitchen and living room! "

Figure 2: Different components of our agent.

Monday, then the user can interact with the agent to modify or remove it. Or just to ask the agent to list all the permanent tasks.

Adding more rules will of course be useful. As to add more values to our slots.

Regarding the speech recognition, there are other tools to use. For example, we only used the *listen* method and we could also make use of the *listen in background* functionality in order to recognize audio only after saying a keyword.

It was a bit tricky to install all the libraries to get everything working. But it is really rewarding to see it finally working, an be able to interact with the agent.

References

[1] https://cmusphinx.github.io/wiki/.

[2] https://pypi.org/project/SpeechRecognition/.

[3] https://pypi.org/project/PyAudio/.

[4] http://manpages.ubuntu.com/manpages/trusty/man1/espeak.1.html.

[5] Daniel Jurafsky and James H. Martin. *Speech and Language Processing (2Nd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2009.

[6] Christopher D. Manning, Mihai Surdeanu, John Bauer, Jenny Finkel, Steven J. Bethard, and David McClosky. The Stanford CoreNLP natural language processing toolkit. In *Association for Computational Linguistics (ACL) System Demonstrations*, pages 55–60, 2014.

Examples

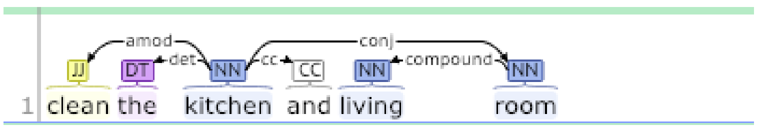
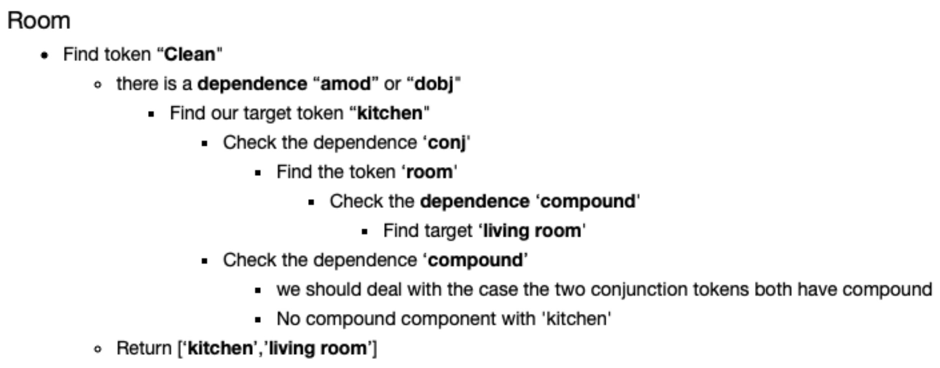


Figure 3: An example of the dependecy tree. This was produced by the *corenlp.run* site, which proved to be really useful while developing our project.



Room

- Find token **“Clean**”
  - there is a **dependence “amod”** or **“dobj**”
    - Find our target token **“kitchen**”
      - Check the dependence **‘conj**’
        - Find the token **‘room**’
          - Check the **dependence ‘compound**’
            - Find target **‘living room**’
      - Check the dependence **‘compound’**
        - we should deal with the case the two conjunction tokens both have compound
        - No compound component with 'kitchen'
  - Return [**‘kitchen**’,’**living room**’]

Figure 4: An example of how to fill the **room** slot for the sentence displayed on *Figure 3*.

```
I’m listening
I’m not listening anymore
I think you said: clean the kitchen quickly and the living room at 17 every Monday

Frame Cleaning Frame:
{      ’day’: [’monday’],
     ’mode’: [’quickly’],
’permanent’: [’yes’],
      ’room’: [’kitchen’, ’living room’],
     ’time’: [17]}

Ok, I will quickly clean the kitchen and the living room at 17 every monday
```

Figure 5: A run of our program filling all slots at once.

```
I’m listening
I’m not listening anymore
I think you said: clean the kitchen and the living room

Frame Cleaning Frame:
{      ’day’: None,
      ’mode’: None,
 ’permanent’: None,
      ’room’: [’kitchen’, ’living room’],
      ’time’: None}

Do you want a quick or deep cleaning?
I’m listening
I’m not listening anymore
I think you said: do it quickly

Frame Cleaning Frame:
{      ’day’: None,
      ’mode’: [’quickly’],
 ’permanent’: None,
      ’room’: [’kitchen’, ’living room’],
      ’time’: None}

Which days do you want to clean?
I’m listening
I’m not listening anymore
I think you said: on Friday and the today

Frame Cleaning Frame:
{      ’day’: [’friday’, ’monday’],
      ’mode’: [’quickly’],
 ’permanent’: None,
      ’room’: [’kitchen’, ’living room’],
      ’time’: None}

At what time do you prefer?
I’m listening
I’m not listening anymore
I think you said: after 12

Frame Cleaning Frame:
{      ’day’: [’friday’, ’monday’],
      ’mode’: [’quickly’],
 ’permanent’: None,
      ’room’: [’kitchen’, ’living room’],
      ’time’: [12]}

Do you want to set this task as permanent?
I’m listening
I’m not listening anymore
I think you said: yes

Frame Cleaning Frame:
{      ’day’: [’friday’, ’monday’],
      ’mode’: [’quickly’],
 ’permanent’: [’yes’],
      ’room’: [’kitchen’, ’living room’],
      ’time’: [12]}

Ok, I will quickly clean the kitchen and the living room at 12 every friday and monday
```

Figure 6: A run of our program filling all slots one by one.