# Bartender

Lorenzo Michieletto e Emilian Postolache

February 25, 2019

## 1 Introduction

The aim of this project is to program a speech assistant running on an hypothetical robot bartender, which dialogues with a client in order to get orders of available drinks and to collect the final payment. The speech bot can treat a certain range of possibilities during the dialogue such as taking multiple orders, making a recap of what has been ordered, deleting something from the ordered list, giving suggestions on a certain drink category and some others that will be made clear further on.

## 2 Pipeline

The NLP problem is divided in three main phases:

1. automated speech recognition
2. answer elaboration
3. text to speech.

The first part is done by calling Google Cloud API using `Speech Recognition`, a Python library. This library has proven to be very efficient in acquiring text, even though some care must be taken when ordering not English beverages. After text is acquired, the elaboration of the answer takes place. This part relies on dependency parsing, implemented using `Spacy`, a natural language processing library that employs neural networks. This part is the most critical: the response is based on the state of the bartender, which depends on the previous states of the conversation. With this dynamic approach, the chatbot expects certain intentions from the speaker and rejects some others. Last but not least, text to speech elaboration is done with `gtts` in Linux and with `pyttsx3` in Windows.

### 2.1 Dependency parsing

The dependency graph, built with `Spacy` on a general sentence for a client order is the following:

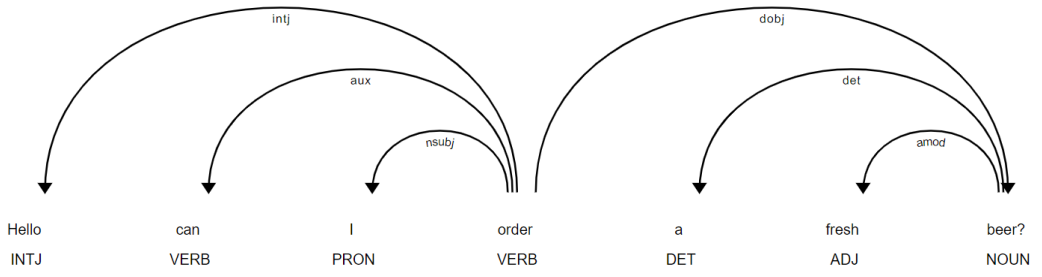


Figure 1: Dependency parsing graph. The dependency relations are labeled on the edges, while the capital words under the lemmas are the POS tags.

Dependency parsing as seen in the previous example is an generic and efficient method that can be used to analyze sentences. The main verb of a phrase is the root of the graph and the most important nouns can be extracted using dependencies with respect to the root. Other information such as numerals related to the nouns (example: "I order **four** Peroni.") can be extracted in a similar way. This technique is used in the program, especially in the order acquirement phase.

The previous analysis is supported by the use of noun chunks, which are flat phrases that have a noun as their heads (a noun plus the other words describing it). Noun chunks have been exploited in order to recognize orders in which the client requests multiple drinks and to recognize drink names having multiple words. As an example, in Figure 1, the sentence is divided into two noun chunks, which are " I " and " a fresh beer".

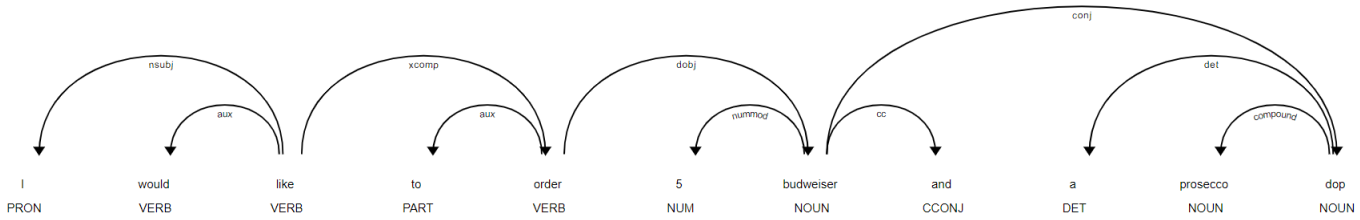


Figure 2: An example of a multiple order with a compound word.

Figure 2 provides three noun chunks: " I ", " 5 budweiser", "a prosecco dop". It is easy to get a multiple order since the two items belong to different sections. For what concerns the compound words, the logic used to assemble them requires to look for names in a noun chunk that are children of the noun chunk's root and whose dependency is `compound`.

# 3 Program structure

The chatbot has six states and in each of these some functions are applicable. Each function elaborates an answer based on the state of the chatbot and on the user's query, allowing a transition from a state to another. Without an explicit representation of these functions, the possible states of the bartender and theirs transition are depicted in Figure 3.

The core function that has been implemented is the one that gets the orders and adds them to the list of ordered items. It can also do more: if something that is not in the bar has been ordered, the function controls whether the ordered item is an actual drink by checking if its name is contained in a list of the most famous beverages. If the name is not recognized, the bartender admits that it did not understood that part. On the other case the type of the drink is understood and a suggestion is offered for that category. Combinations of correct orders, unrecognized orders and not present orders are considered when answering. As said before, one can specify a numerical quantity for each of the desired drink, and the bartender takes it into account.

Another important function finds out whether a generic order has been made, for example when asking for a "beer" or a "glass of wine". By recognizing the category of the ordered drink, it makes a suggestion or it illustrates the menu list, inherent to the ordered category. The choice between these options is made with a random probability while choosing a recommendation is done with a probability proportional to its price.

When a suggestion is asked or when one is proposed, the state changes into one where either the advice is refused or it's accepted. In the latter case, if the number is not specified, there is another transition to a state where the answer expected is a number.

There is also the possibility to remove items while ordering. Edge cases like the case in which the number of removed items is not specified or in which one removes more items than it has ordered are handled.

Last but not least, once the order is done, the state is switched to a payment phase where a recap of the ordered drinks and the payment amount is made. Before proceeding with it a client could decide to add or delete something.

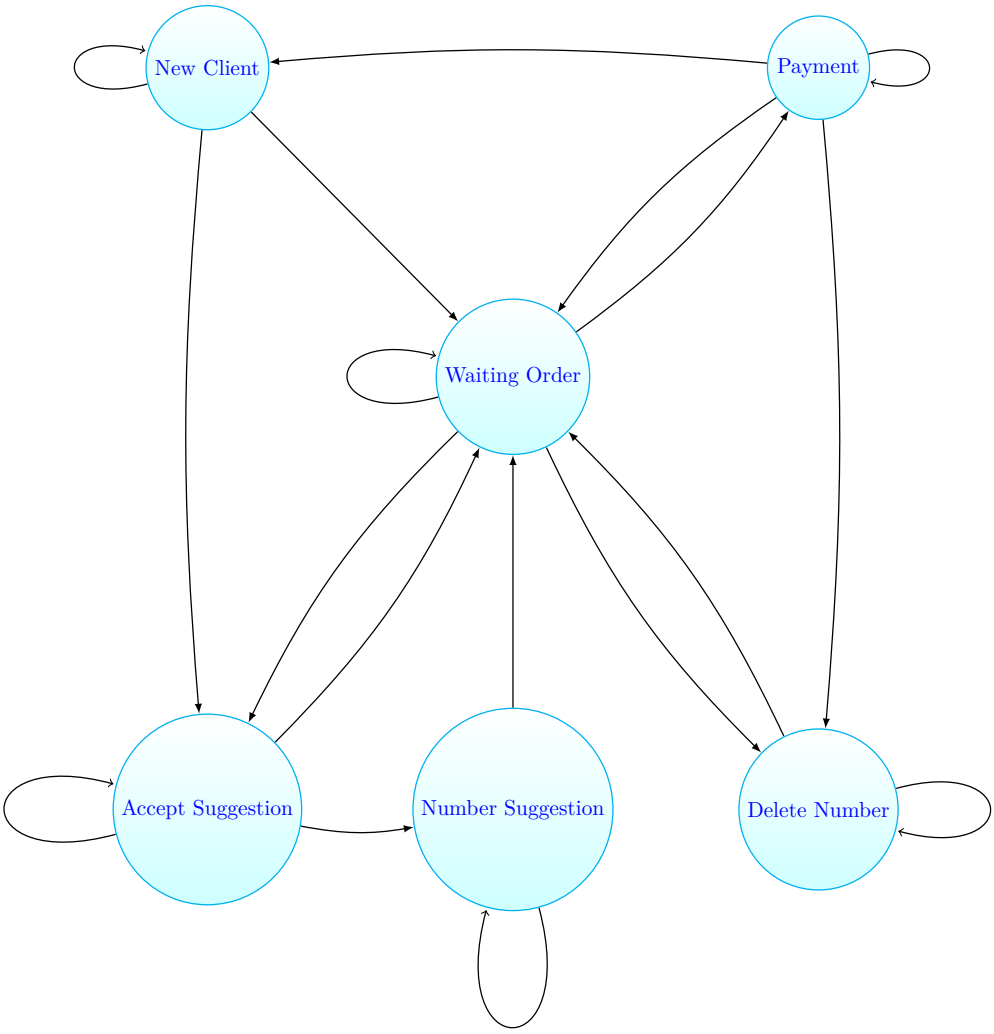


Figure 3: State diagram for the bartender chatbot. The bartender starts in **New Client**, where it can be greeted, it can receive a first order or a request for a suggestion. **Waiting Order** is the main state where an user can give a new order, can ask for a suggestion, delete some items or end by proceeding with the payment. While in **Payment** an user can pay or delete some items. **Accept Suggestion** is triggered by a suggestion of a drink to the user, so the user can accept it or refuse it. **Number Suggestion** and **Delete Number** are two support states which both expect a number when the user doesn't specifies it after accepting a suggestion or deleting an item. The bartender remains in the same state when it doesn't understand a query and resets to **New Client** if intentions to leave are express.

# 4 Notes

- The simulated bar in the application contains two category of drinks: beers and wines. The following beers can be ordered:
  - IPA (5 €)
  - Blanche (5 €)
  - Heineken (3 €)
  - Moretti (3 €)

  - Peroni (2.5 €)
  - Budweiser (3 €)
  - Tuborg (2.5 €)
  - Bavaria (1 €)
  - Franziskaner (3.5 €)
  - Leffe (4 €)
  - Ceres (5 €)

  The following wines can be ordered:

  - Prosecco DOP (20 €)
  - Don Perignon (100 €)
  - Chianti (15 €)
  - Cristal (100 €)
  - Cartizze (50 €)

  It is easy to add new drink categories and new drinks since the approach is object oriented.

- When taking an order, items must be separated with an `and` conjunction. This is because `Speech Recognizer` doesn't identifies commas in the spoken language, so `Spacy` analyzes the phrases in a wrong way.
- A big limit of the application is due to the free Google speech to text service that `Speech Recognizer` uses: sometimes correct pronunciations are not identified. For example it has difficulty to recognize drink names from a non English country. Attempts to use the payed Google Cloud service have not proven to improve the quality of the speech-to-text part.
- When there is a check that finds out if an item is a real drink, precompiled lists are used. The application could be improved with a semantic search over a knowledge base like Babelnet (some experiments with this approach have been proven by us to be successful but there was a performance downgrade).
- The application sometimes suffers low speed performances since it depends on online services. There are usage quotas that limit how many queries can be sent to the web server.