EmGiBot: the World’s first Voice Assistant that sees you!

Emanuele Alessi (1486470)
Gianmarco Forcella (1725967)

Index

1. Introduction

The last decade saw the introduction in the market of the *Intelligent Personal Assistant*, which rollout slowly began in 2010 with Apple's **Siri** and, for four years after that date, these types of *bots* were relegated only to smartphones, with no luck of using them for domotic stuff. This changed in 2014, when Amazon launched its new device, **Amazon Echo**, that was equipped with **Alexa**: this was literally a game changer for the *I.P.A.* sector, as it opened the doors also for **Google Home** and Apple's **HomePod.**
None of these companies, though, ever thought of making an *Intelligent Personal Assistant* for computers that combines the power of Semaphoric Gestures to stop a song or skip to the previous/next one.
For this reason, we thought of trying to make an experiment to see whether such a combination could be possible. And that successful experiment is called **EmGiBot**, which we are going to present in this report.

2. Theoretical overview

Before going deeper with describing our solution, let us first describe all the possible *Multimodal* Interaction problems that we will face and, also, a brief description of the tools that we will use.

2.1 Speech Recognition

Speech Recognition (or **Automatic Speech Recognition** (ASR), or **computer speech recognition**) is the process of converting a speech signal to a sequence of words, by means of an algorithm implemented as a computer program.[1]
This is the core and, naturally, the most important part of the entire project.
Building a Speech Recognizer relies, mainly, in N variables:

1. The **Environment** itself: if the place in which the user is using the IPA is too *noisy,* this will be troublesome as the program won't be able to understand very well when an **utterance actually ends or not**;
2. The **Speaker**, also, can make a huge impact on the performance of the Speech Recognition: there can be a tremendous difference whether the user uses an *external microphone* or the computer's *internal microphone;*
3. Last but not least, the way the user **speaks** also makes a huge difference: the speed of the utterance can, in fact, impact what the algorithm understands and, then, "translate" to written language.

Though there are 3 different ways in doing a possible Speech Recognition, we decided to rely on the **Artificial Neural Networks** methodology.

2.2 Speech Interaction

Speech Interaction doesn't have any peculiar definition: it's just what it is. The way with which we humans communicate to each other.
In terms of abstraction, the **Speech Interaction** has 3 layers[2], as it can be seen from the below image.

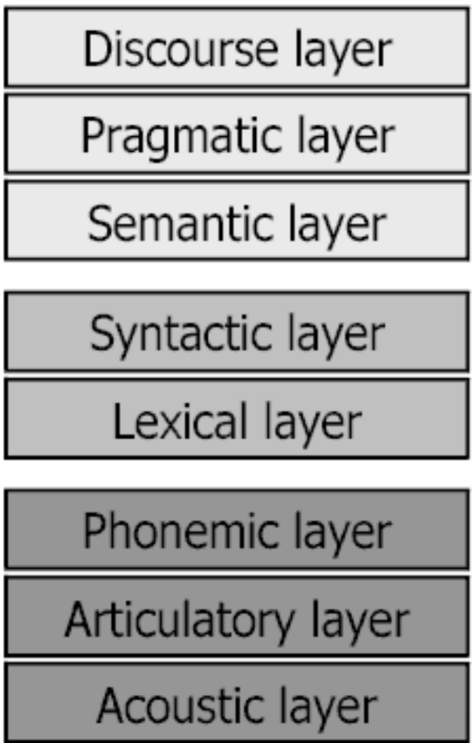


Figure 1 The Stack of abstraction of the Speech Interaction. From bottom to top: the phonetics area; the linguistic area; the meaning area

A Multimodal system that wants to try to imitate a human's speech, needs to perfectly be able to work with all of these layers, since it has been proved that **Speech Interaction** is probably the best way to interact with a human being.

2.3 OpenCV

OpenCV (Open Source Computer Vision Library) is an open source computer vision and machine learning software library. **OpenCV** was built to provide a common infrastructure for computer vision applications.
The library has more than 2500 optimized algorithms, which includes a comprehensive set of both classic and state-of-the-art computer vision and machine learning algorithms. These algorithms can be used to detect and recognize faces, identify objects, classify human actions in videos, track camera movements, track moving objects, etc.

OpenCV is a cross-platform library using which we can develop real-time computer vision applications. It mainly focuses on image processing, video capture and analysis including features like face detection and object detection and it has more than 47 thousand people of user community and estimated number of downloads exceeding 18 milion.
The library is used extensively in companies, research groups and by governmental bodies.

Figure 2 OpenCV's logo

2.4 RASA NLU

Rasa NLU is an open-source natural language processing tool for intent classification, response retrieval and entity extraction in chatbots. Probably, it's the best opensource library available in the market.
Based on the **Long Short Term Memory** (**LSTM**) **Neural Network Model**, **RASA** takes in input a dictionary on <*intent*> - <*possible sentences*> and, then, the Machine Learning model does the magic.
Once it has been trained, **RASA** is ready to perform all of its logic to a new input sentence and try to understand the intent of this phrase. This usually has an accuracy of at least **80%** even if that intent only has 3 associated sentences.
It relies on SpiCy's corporas.

Figure 3 RASA's logo

2.5 Tensorflow

TensorFlow is probably the most famous framework for working out any large-scale Machine Learning: originally created by the *Google Brain Team*, it is an open-source library which bundles mainly Deep Learning models and algorithms.
The library can train and run Deep Neural Networks for many tasks, ranging from digit classification to image recognition.

But how does it work?
TensorFlow allows the creation of so-called “dataflow graphs”; structures that describe how data moves through a graph. Here:

- A node represents a mathematical operation;
- An edge between two nodes symbolize a “Tensor” (short for multidimensional array).

The nodes, though, are not executed in Python: to ensure a higher speed of computation, in fact, the library executes these operations in C++, so that they can be worked out at low-level.
Another great advantage is that the developer can choose to execute calculations either on the CPU or the GPU, to ensure more computational power to the program.

As of 2019, TensorFlow is accredited as one of the most used libraries for Deep Learning and it keeps growing, even with a recent release for JavaScript.

We decided to use Tensorflow for its *pretrained* MobileNetV1 Neural Network, which helped us in a task that will be described in a few chapters.

Figura 1 Tensorflow's logo

3. Project setup

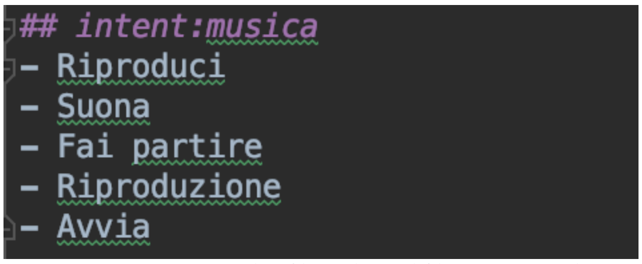
Let us know enlist the technologies that we used while working on this project.

- **Python 3.x**
- **PyCharm as our IDE**
- **NumPy**
- **OpenCV**
- **RASA**
- **Selenium + Headless Mode** for allowing songs to play from **YouTube**;
- **NewsAPI.org** for fetching the latest news;
- **Speech_Recognition** for the **Speech Recognition** part;
- **Pyttsx3** for the **Speech Interaction** part;

To check out the full project, please refer to this GitHub repository, which also contains all the papers we took inspiration from for our work.

3.1 Dataset

In order for work, *RASA* needs a dataset.
The dataset itself, actually, it's just a .md file that we will call "*NLU*" which will be structured like this:



```
## intent:musica
- Riproduci
- Suona
- Fai partire
- Riproduzione
- Avvia
```

Figure 4 An example of how the .md file is structured

The ## part represent the label of the intent that we want to predict through *RASA.*
Immediately after that part, *RASA* accepts a list of all possible sentence combinations with which it can lead back to that intent.

3.2 Typical Workflow

The following picture represents a typical workflow for **EmGiBot.**

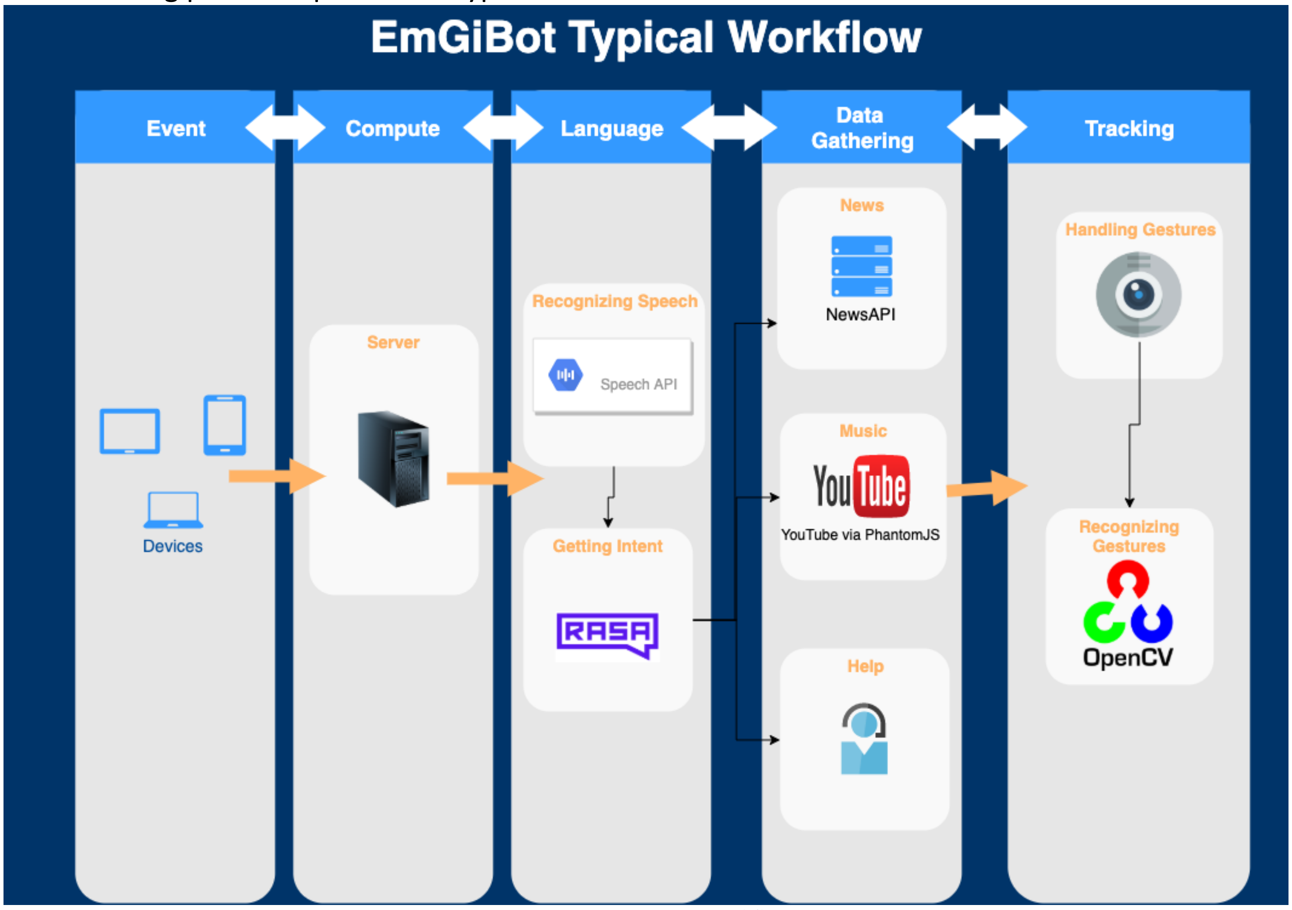


Figure 5 EmGiBot's workflow

- From a device of any kind (for now, just a desktop computer. In the future, perhaps, more!), the user performs a request through **Speech Interaction.** This is called the **Event Phase;**
- The server then receives the request. The **Compute Phase** starts;
- The **Language Phase** is all dedicated to the core part of the application. It is divided into:
  - **Speech Recognition Part**: through *Google's Cloud Speech API*, we make a call to convert from voice to text;
  - **Natural Language Understanding Part**: after converting the voice to text, we take that to *RASA* in order to understand the user's intent
- Depending on which intent was found, the **Data Gathering Phase** starts:
  - **News** will be gathered if and only if *RASA* labels the request for that;
  - **Music** will be played if and only if *RASA* labels the request for that;
    - Should music be requested, then we also enter in the **Tracking Phase,** through which we understand (thanks to *OpenCV*) whether to go to next / previous song or stop the playback;
  - **Help** on how the *IPA* works will be dispatched if and only if *RASA* labels the request for that

3.3 Technical Limitations

In order to develop **EmGiBot**, we wanted to build / use four different Neural Networks:

1. One that could perform the NLU part, something for which we decided from the very beginning to use *RASA;*
2. One that could be able to perform the **Speech Recognition;**
3. One that could perform **Speech Interaction;**
4. And one that could possibly be able to track and recognize the user's movement.

But since all the development was carried out in two machines with similar computational power (both equipped with a *3,1 GHz Intel Core i5,* 8GB of RAM and an NviDia GeForce 130MX graphic card), we knew from the very beginning that **EmGiBot** would have never gotten too far because of the high calculation power that it required.
For this reason:

1. The only Neural Network that we decided to keep was *RASA's*, because of its power and precision even with just a small dataset;
2. We decided to give the **Speech Recognition** part to *Google's Speech API*, because of its tremendous power on getting the sentence right;
3. Leave the **Speech Interaction** part to Python's *pyttsx* library, which seemed perfect for the task;
4. And use *OpenCV* to detect the gestures.

Doing so granted **EmGiBot** to perfectly work on the computers used, even on a *RaspBerry Pi* 4 equipped with a camera and a microphone.
In the following image, we represent the Python's custom-made classes that can be found in the project while, in the next image, we will represent the use cases of this project, which will be described in the next chapters.

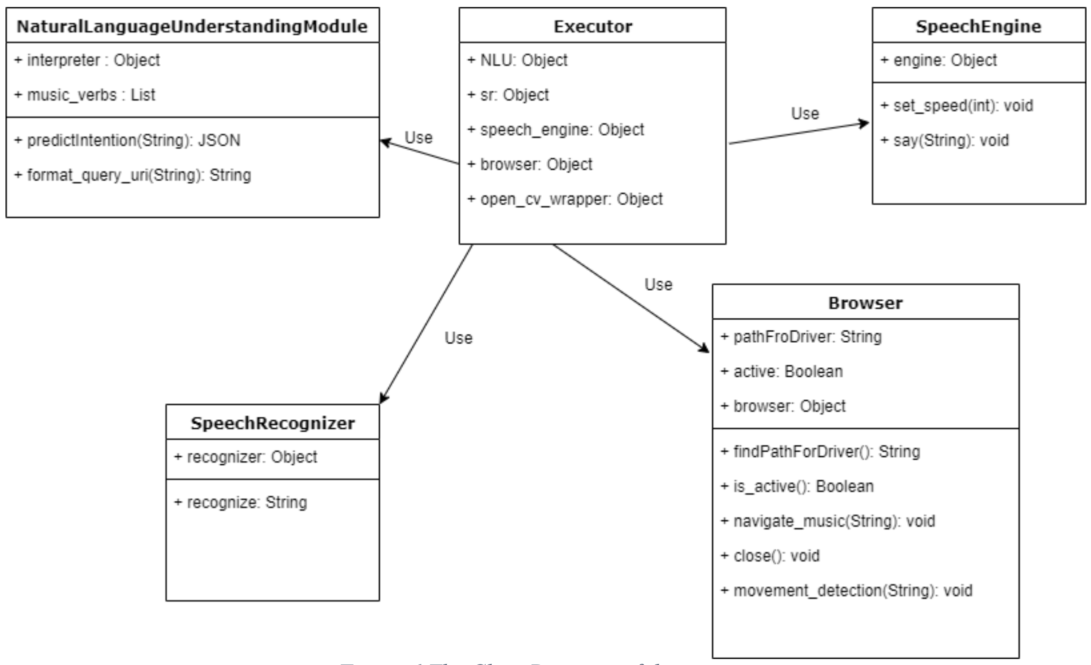


Figure 6 The Class Diagram of this project

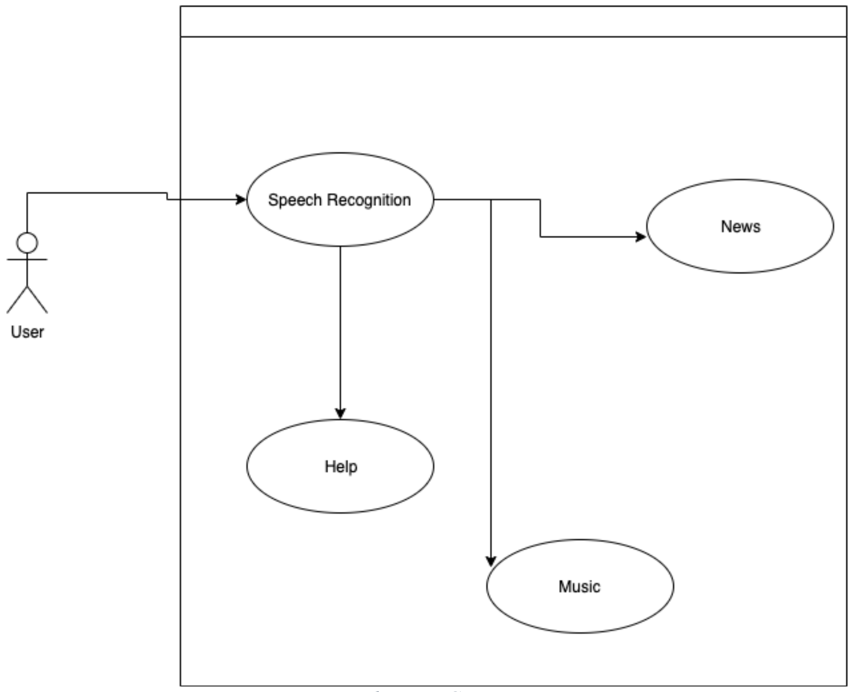


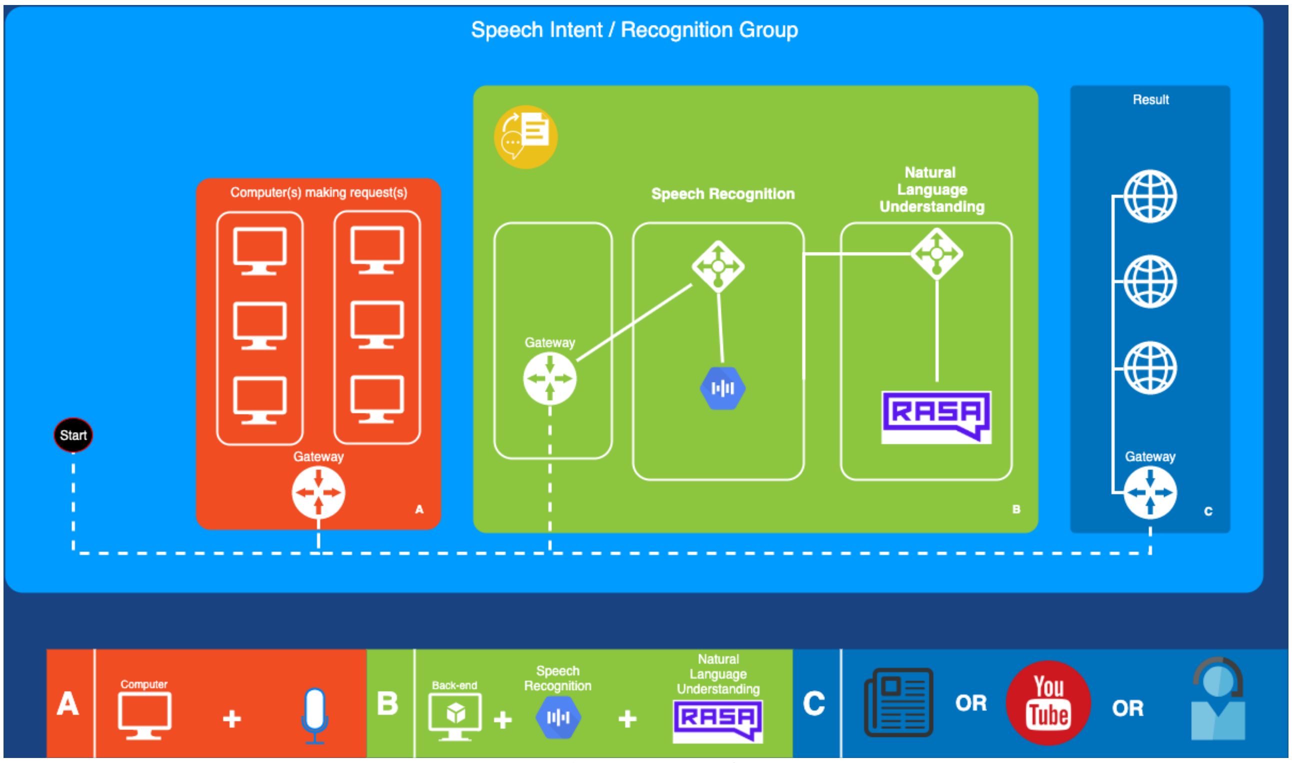
Figure 7 The Use Case Diagram

3.4 Report Organization

Before proceeding, just a quick note on how this report will be organized.
The code will not be reported: for full reference, please check the link to the Github's repository that was previously mentioned.
We will divide the report in 6 more sections:

1. The first section will talk about the entry point for using **EmGiBot**: one's voice;
2. Then, we will examine the possibilities that **EmGiBot** offers, which are explained if asking for Help;
3. Then, we will examine the part in which the latest news are retrieved;
4. We will then examine the way **EmGiBot** handles music requests and gesture recognition;
5. A presentation of a real test case made with the help of 20 testers;
6. Lastly, we will discuss our conclusions.

4. Level One – Speech Interaction & Recognition


Speech Intent / Recognition Group
Computer(s) making request(s)
Gateway
A
Start
Speech Recognition
Natural Language Understanding
Gateway
RASA
B
Result
Gateway
C
A
Computer
+
B
Back-end
Speech Recognition
+
Natural Language Understanding
RASA
C
OR
You Tube
OR
Figure 8 The Entry Point of EmGiBot

This is probably the most important point of the whole application, the Gateway that dictates everything.
The first thing, right after *RASA* has loaded its model, is to start *pyttsx* in order to have the **Speech Interaction** part ready.
The process that now follows is to be considered a *while loop*.
The microphone gets activated via the *Speech Recognition* library and starts to "listen". As soon as it believes that an utterance is finished, the recorded audio is sent to *Google's Speech Recognition* tool for the **Speech Recognition part**. Since only one language could be set, we decided to set the Italian one.
After that Google returns its output, which can either be an empty string or the audio "stringified":

- If an empty string is returned, the process **starts** again;
- If a string is actually returned, instead, *RASA* will try to understand the intent of the utterance and, then, redirect to the correct level of the application.

5. Level Two – Help

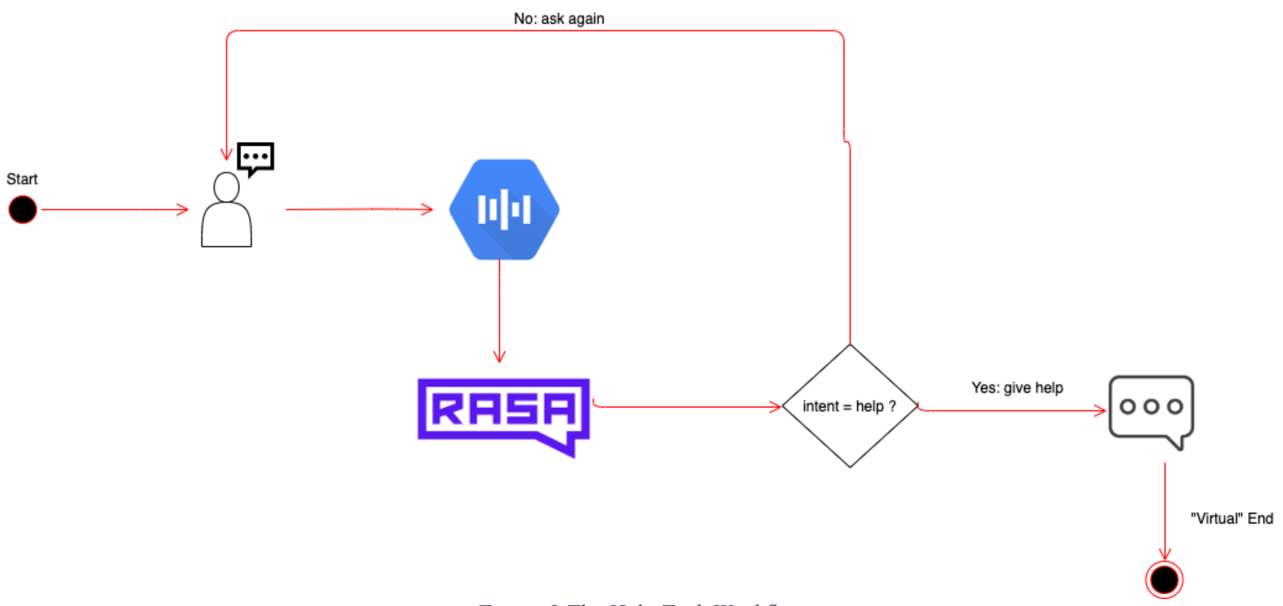


Figure 9 The Help Task Workflow

An *Intelligent Personal Assistant* is a special type of an User Interface. It still allows the user to do stuff, but via his/her voice. Without touching anything. Because of that, some might have troubles in understanding how it may work.
For this reason, every time that **EmGiBot** is turned on, we make it say, shortly, how it works.
But what if the user forgets this?
By just asking to the *IPA* for help, the program is able to understand the request via *Google's Speech Recognition* and *RASA's* NLU module, and perform the desired **Speech Interaction**. With this done, technically the task is done and that would be it. But this just doesn't really apply to an *Intelligent Personal Assistant*, since it never quits unless the user shut it down! For this reason, we've labeled the "End" as a "Virtual End", since it actually doesn't finish.

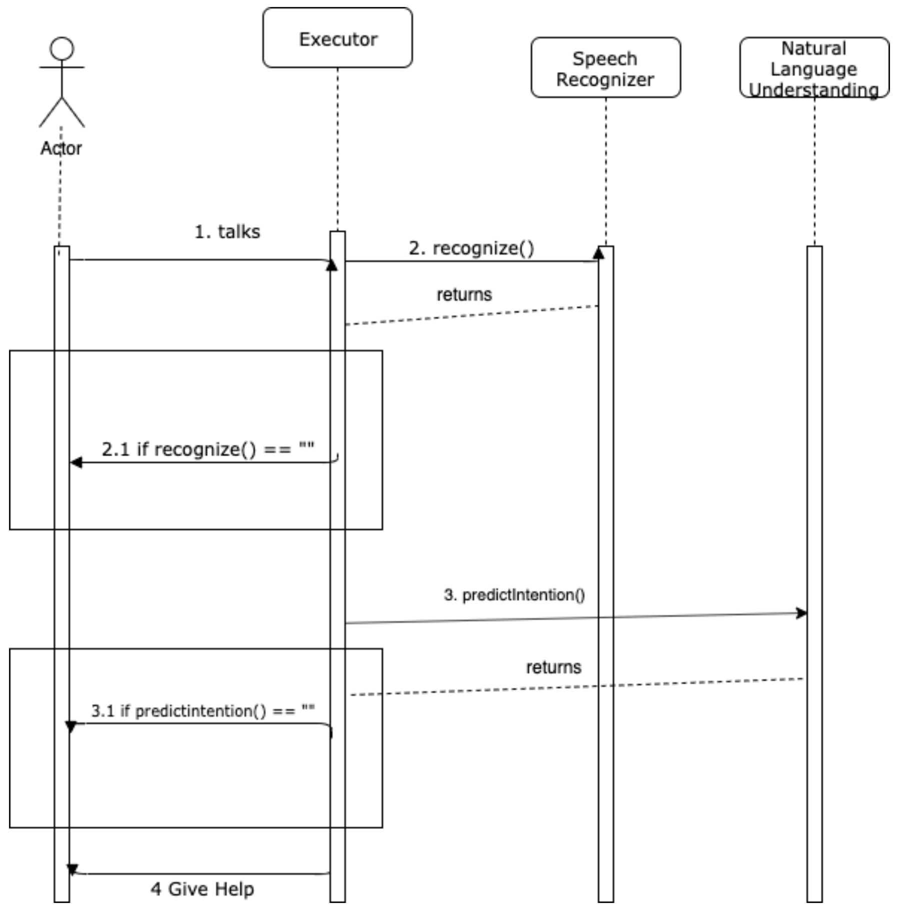


Figure 10 The Help's Sequence Diagram

The above image describes the same with a Sequence Diagram.

6. Level Three – News

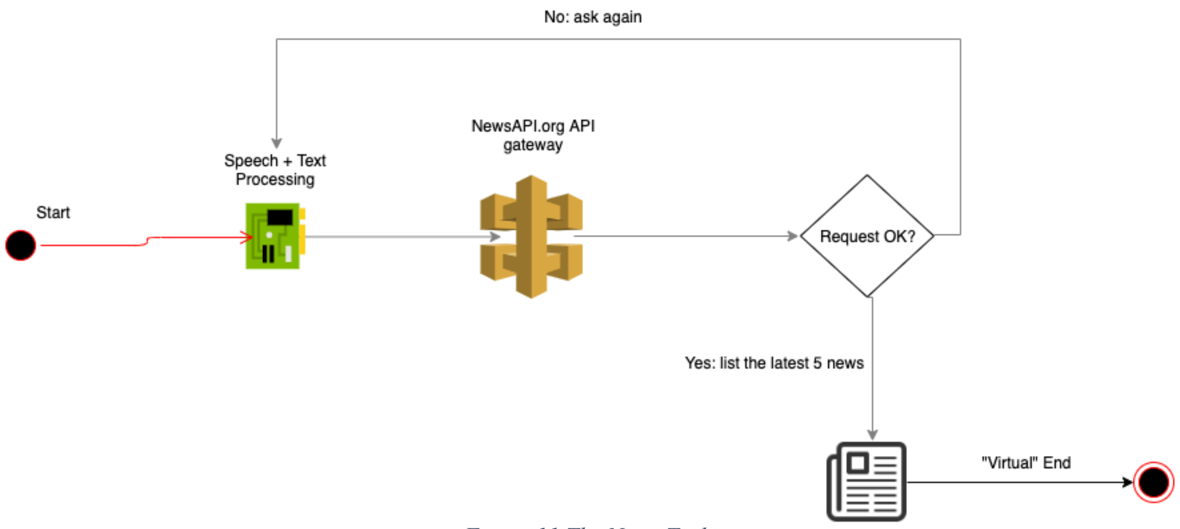


Figure 11 The News Task

When the user queries for having the latest news, the system does not use any kind of external Python library / wrapper etc. Instead, it entirely relies on *NewsAPI.org*, a website which exposes several REST endpoints to query, depending on what we want to know.
In our case:

- We need the *top headlines*, which the service interprets as the latest news, so we will query the **top-headlines** endpoint;
- Since **EmGiBot** only speaks Italian, we query the news for the *Italian Country*;
- And, even though it's a free service, we pass the *API Key* which will grant us the access to the service.

If the request goes smoothly, by default a JSON Array of 10 elements is returned, containing all kind of news already ordered by the newest one; **EmGiBot** will then extrapolate the first 5 news and perform the **Speech Intent** action to tell them to the user. Should, instead, the request fail for some kind of reason, the *IPA* will query again NewsAPI.org and, should it fail again, it will tell to the user that it couldn't manage to fetch the news.
Once again, we have a "Virtual End" endpoint instead of an actual endpoint as the system will still be in listen.
The below image describes the same with a Sequence Diagram.

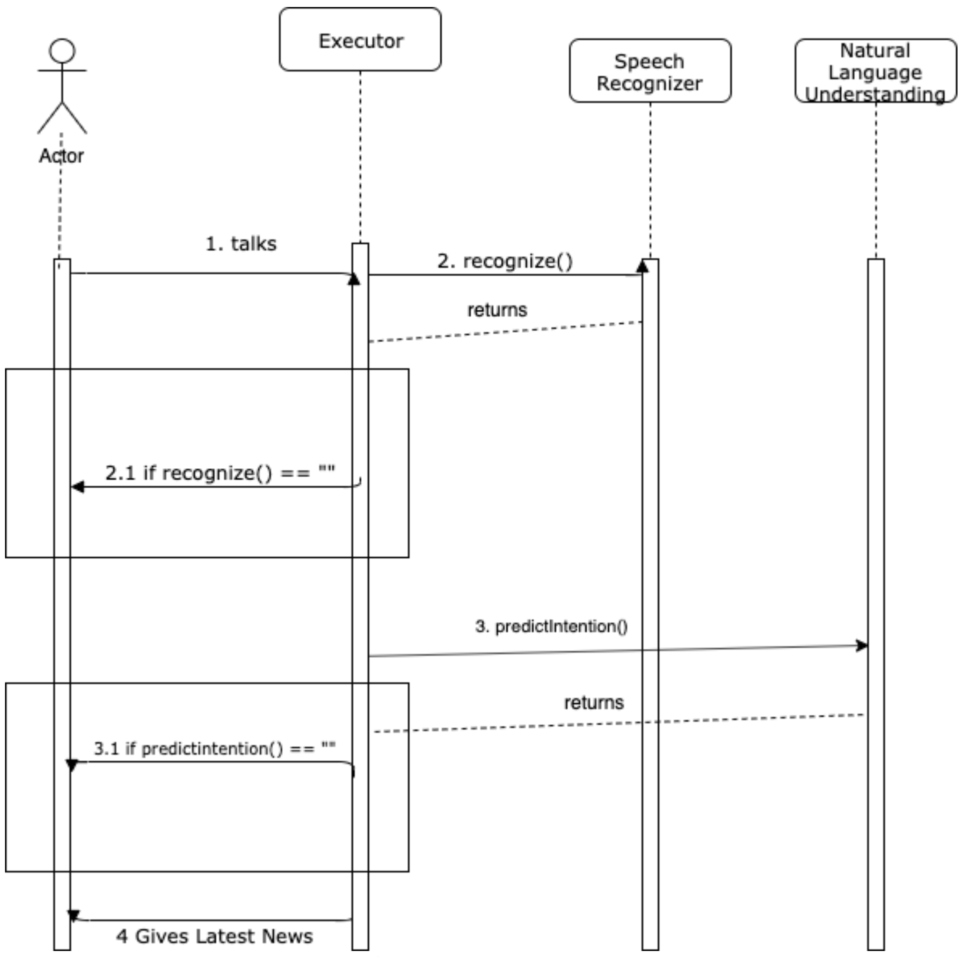


Figure 12 The Latest News Sequence Diagram

7. Level Four – Music

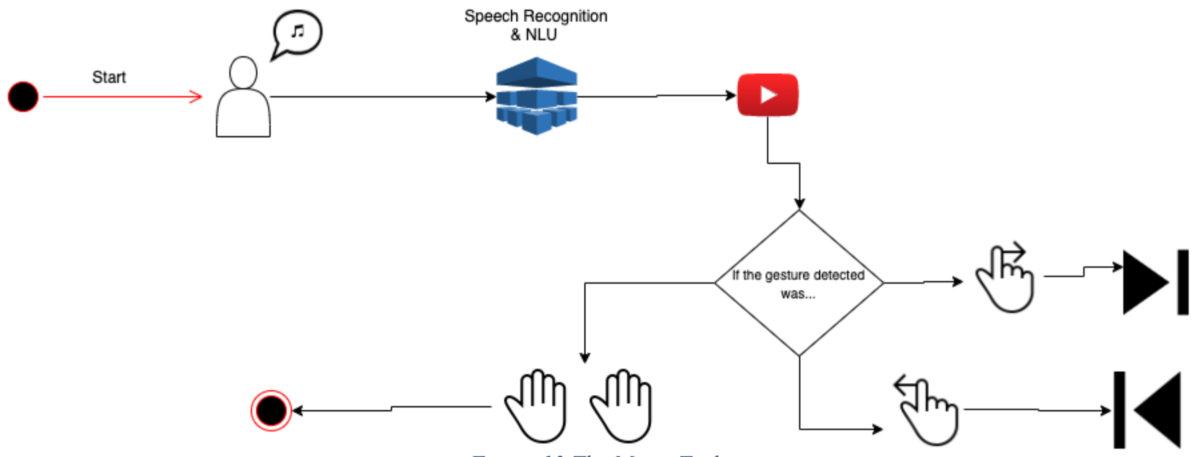


Figure 13 The Music Task

We now present the final task, which is probably the most articulated after the **Speech Intent + Speech Recognition** one.
Everything starts with the user asking to play a song: the **Speech Recognition** + **Natural Language Understanding** gets the request and then opens an instance of *Selenium's Chrome* to query *YouTube* for the specific song requested. The reasons behind the mentioned choices were the following:

- First of all, every music streaming service (ie, *Spotify*, *Tidal*, etc…) all offered just a 30 seconds preview, while *YouTube* didn't. Also, reach the full webpage in which YouTube exposes its search results is quite easy (the URI is something like */results?search_query=<name of song>*. The only thing needed to do is just to use a bit of manipulation on the string in order to add the + sign at the place of every whitespace occurence;
- Secondly, the only way to emulate a music streaming service-like feature was to use an *Automation Browser Tool* (such as Selenium) combined with *Selenium's Chrome,* which is a headless browser (meaning that the user will never get to see a window).

So, when *Selenium's Chrome* reaches the query results page, it automatically selects and load the first result. This is done because, statistically speaking, we are able to do a perfect query that matches the user's request thanks to the power of *Google's Speech API*, which grants us always a really high success rate (about 98% most of the times) in understanding what the user is asking, even if he's asking for a non-italian song. The singer can also be specified: this will only grant more specificness to the query and, hence, to the song.
After that, the selected song is loaded: at this point, the track should either play or load, instead, some ads. Since there is no way to determine whether there's an advertisement going on or not, we decided to not take any precautions about this.
By reaching the track's page, *OpenCV* finally gets triggered and will start its **Semaphoric Gestures Detection**. We cabled this part of the program for recognizing 3 gestures:

- **Swipe Right:** the application will load YouTube's next song;
- **Swipe Left:** the application will load the previous page;
- **Double Palms**: the application will close the music task and will return back to listen for new input. The choice of using a *Double Palm* gesture instead of just a closing fist was dictated from the fact that the only *OpenCV's model* that we found for this gesture was entirely made with an *Haar Cascade* approach, which meant that it deeply suffered of light conditions. By doing so, though, we resolved the illumination issues and also avoided false positives.

Below we also attach a sequence diagram of the just described use case.

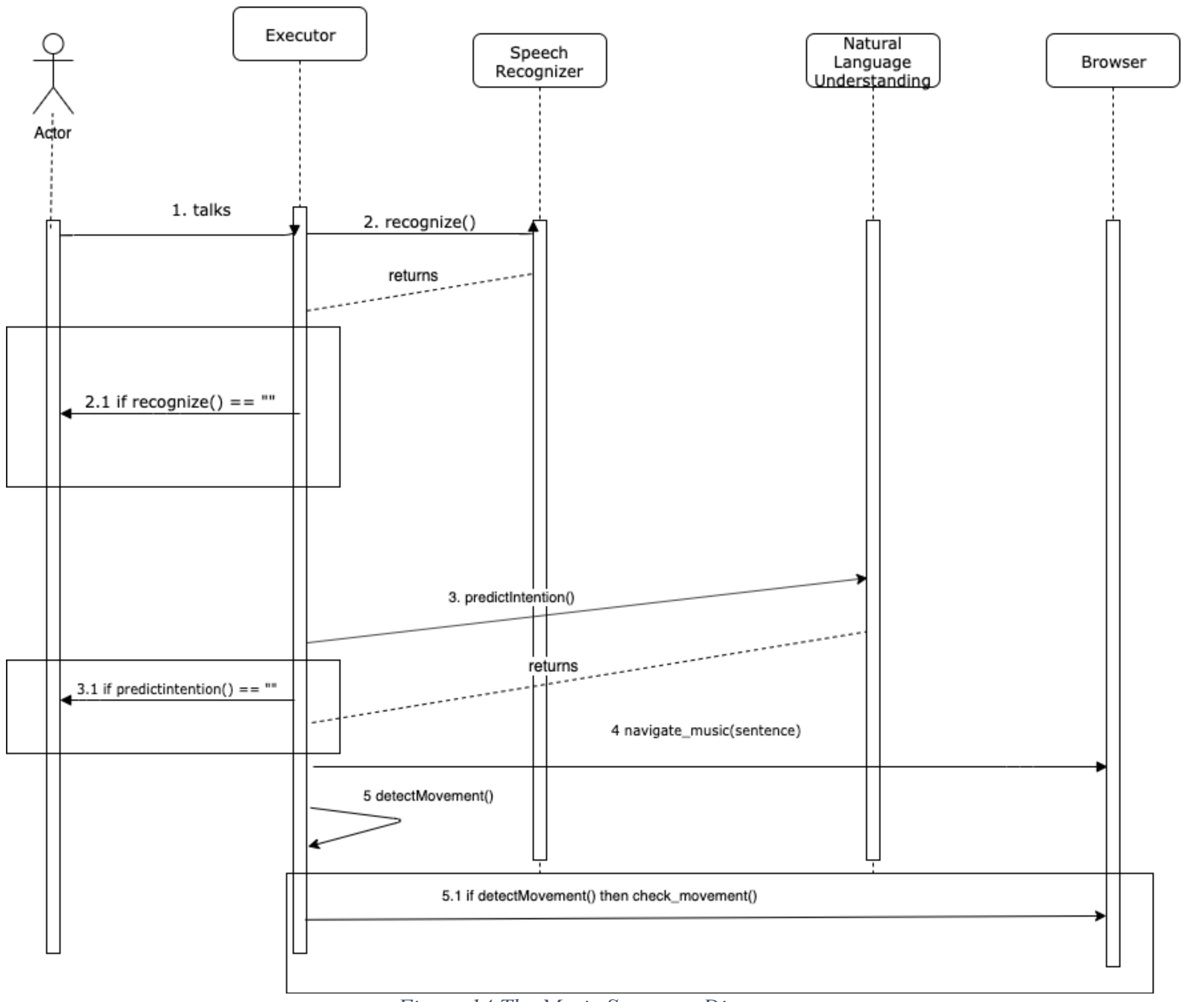
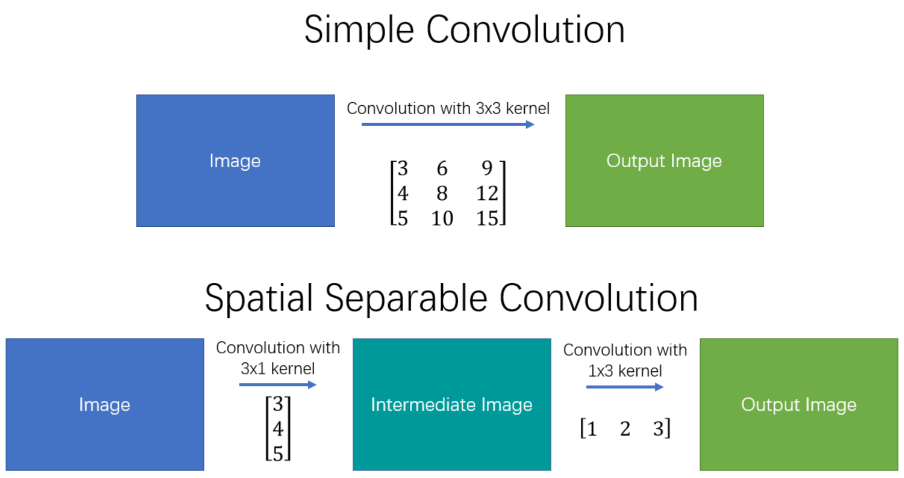


Figure 14 The Music Sequence Diagram

7.1 MobileNet

After several searches on the web regarding the papers dealing with the topic of hand detection and localization, we have concluded that all the papers that we found encourage to use Deep Learning neural networks in order to perform the task as good as possible.
Our decision was to use a neural network known as **MobileNet v1**.
MobileNet is a lightweight convolutional neural network that is frequently used for object detection. The main reason for why it is widely used and made available by Microsoft, Amazon and Google services, due to its accuracy in results, is that it uses depthwise separable convolutions.
These convolutions are composed of two different layers: *depth-wise convolutions and point-wise convolutions*. Depth-wise separable convolutions reduce the computation with respect to the standard convolutions by reducing the number of parameters.
The following image, instead, represents the entire architecture of **MobileNet v1.**


Simple Convolution
Image
Convolution with 3x3 kernel
$\begin{bmatrix}3 & 6 & 9\\4 & 8 & 12\\5 & 10 & 15\end{bmatrix}$
Output Image
Spatial Separable Convolution
Image
Convolution with 3x1 kernel
$\begin{bmatrix}3\\4\\5\end{bmatrix}$
Intermediate Image
Convolution with 1x3 kernel
$\begin{bmatrix}1 & 2 & 3\end{bmatrix}$
Output Image
Figura 2 The architecture of the MobileNet

8. Real Field Testing

After exiting the alpha phase, we decided to do some beta testing with 20 persons, in different conditions:

- About 16 people tested **EmGiBot** in a house, so normal and optimal conditions for the program's execution;
- The remaining 4 people were invited to test our *IPA* in the outside world (ie, bars, etc) or in crowded houses, just to understand if problems such as noise and illumination could affect **EmGiBot** somehow.

The only input that we gave to the testers was just to explain them how an *Intelligent Personal Assistant* works, bringing up examples like Amazon's **Alexa**, or Apple's **Siri.** Then we left complete freedom to the user, without telling him/her anything, just to see where this could lead.
The results were actually a lot satisfying: 100% of the users were confident with using **EmGiBot**. Just 7 wanted to be sure on what to do with our *IPA* and asked for Help.
In most cases, **EmGiBot** was able to respond correctly to every formulated request. The only problem raised with the 4 people that tried the application in "the wild": because of the too much noise, in fact, Python's external *Speech* Recognition library was never able to terminate the "listen phase", since it could never find an actual end to the utterance. This is still an unresolved problem, if the user decides to go with the computer's internal microphone while, of course, it won't be a problem if he/she will just use an external one.
As for the gesture tracking part, the illumination conditions did not cause any kind of problem.

9. Conclusions

When we first started developing the idea of **EmGiBot**, everything started as a game: we honestly weren't sure whether we could actually achieve the idea of having an *Intelligent Personal Assistant* mixed with the ability of handling gestures. In fact, we don't hide the fact that the Semaphoric Gestures were probably the hardest part to implement, because of all the problems connected to the light conditions.
In the end, we ended in succeeding our initial goal (even though the mentioned problems regarding the gestures) and creating an *Intelligent Personal Assistant* which satisfaction rate was very appreciated by our 20 testers. We've demonstrated that such a state-of-the-art project can be possible.
Surely, *RASA*'s NLU module is not able to understand everything if we don't include it in the dataset, so we need to update it every time we want to cable new features, but we can't deny that we are extremely happy with what it accomplishes with just a really few lines of code.
A next step for future work, will surely be to bring **EmGiBot** not only to computers, but also to smartphones and tablets.

References

[1] Course’s Slide #06, Speech Recognition

[2] Course’s Slide #07, Speech Interaction