Cognitive Robotics

Final project 2022/2023 - To Do List with Pepper – Group 06

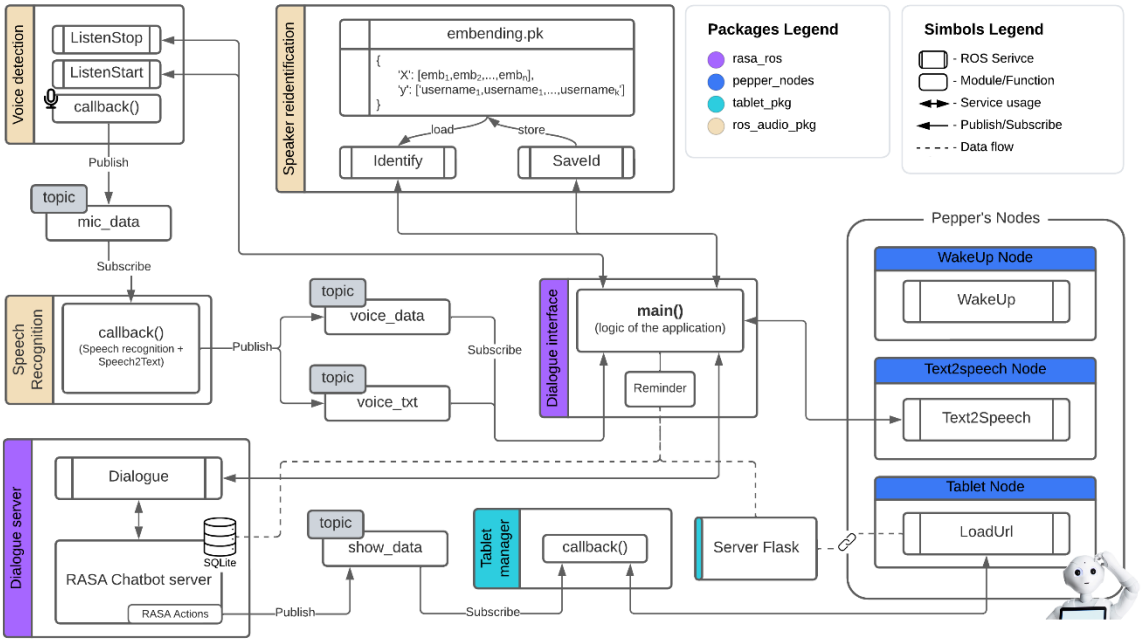
Amato Mario	0622701670	m.amato72@studenti.unisa.it
Avitabile Margherita	0622701825	m.avitabile6@studenti.unisa.it
Battipaglia Lucia	0622701758	l.battipaglia6@studenti.unisa.it
Sonnessa Francesco	0622701672	f.sonnessa@studenti.unisa.it

Summary

1. WP1 & WP2 – Architecture 2
   1.1. Rasa_ros package 2
      1.1.1. Dialogue Server Node 2
      1.1.2. Dialogue Interface Node 3
   1.2. Ros_audio_pkg package 3
      1.2.1. Voice detection node 3
      1.2.2. Speech Recognition node 3
      1.2.3. Speaker reidentification node . 3
         Identify 3
         SaveId. 4
   1.3. Table_pkg package 4
   1.4. Pepper_nodes package 4
2. WP3 – Rasa integration 5
   2.1. Working examples 5
      Action insert 5
      Action show 5
      Action delete 5
      Action update 6
   2.2. Explanation of the design choices .. 7
      Domain File. 7
      2.2.1. NLU File 7
      2.2.2. Config File 8
      Pipeline. 8
      Policies 8
      Performance. 9
      2.2.3. Actions 10
      2.2.4. Manage multiple users 11
      2.2.5. Set deadline and reminder 11
   2.3. ROS integration 11
3. WP4 – Re-Identification 12
   3.1. Identify Service 12
   3.2. SaveId Service 12
   3.3. Speaker Identification 12
   3.4. Design choices 12
   3.5. Pepper speaking issues 12
4. WP5 & WP6 – Test 13
   4.1. Unit Test 13
      4.1.1. Re-identification node 13
      4.1.2. Speech2Text 13
      4.1.3. Pepper nodes 13
   4.2. Integration test 14
      4.2.1. Ros and Rasa 14
      4.2.2. Chatting with Pepper 14
   4.3. Final test (without Pepper) 15
      TEST 1.a 15
      TEST 1.b. 15
      TEST 2. 15
      TEST 3. 15

1. WP1 & WP2 – Architecture

In this chapter we describe the architecture of our application by introducing its individual parts.



Final Architetture

1.1. Rasa_ros package

In the *rasa_ros* package the *Dialogue Server* and *Dialogue Interface* nodes are implemented, respectively for the integration of the RASA chatbot with the ROS environment and for the management of the entire application logic.

1.1.1. Dialogue Server Node

Below is a detailed explanation of the *Dialog* service and the *RASA Chatbot Server*.

Dialogue service. Implement a ROS service to interact with the RASA chatbot. This service takes in input a string, such as a question or a simple sentence, to be submitted to the chatbot and returns the chatbot's response as a string.

RASA Chatbot server. Runs the chatbot implemented for the midterm. The parts highlighted in the architecture image are the SQLite database, where activities and categories are saved for each user, and RASA Actions.

The required features for the chatbot are:

1. Manage multiple users
2. Insert a new activity in a category
3. Remove an activity in a category
4. Remove a category and all activities in it
5. Show all categories
6. Show all activities within a category
7. Update an activity within a category
8. Set a deadline and a *reminder* for an activity

1.1.2. Dialogue Interface Node

Contains the *main()* that is the core of the system. It integrates all the modules described here and implemented the entire logic of the application. It scans the following phases:

1. **Re-identification –** the system waits for the user to start the conversation. Once the interlocutor's voice has been identified, the re-identification module is asked to recognize the user by invoking the *Identify* service.

   If the re-identification module fails to recognize the interlocutor (response *None* from *Identify*) the system asks the user to introduce himself by providing his name. Once the name is obtained, it is passed to the re-identification module so that it can save the association $(embedding_i, username_i)$ via the *SaveId* service. We proceed to step 2.

   If instead the re-identification module recognizes the user, the *Identify* service will return the identifier and we can proceed to step 2.

2. **Conversation session –** after identifying the user, the actual conversation can begin. In this phase, the system acquires the user's voice, translates it into text and sends it to the chatbot. The response string from the chatbot is reproduced by Pepper via the *Text2Speech* service, if it is present, otherwise from the computer.

   The conversation session ends after the user greets (e.g. "bye"), then the system resets himself to welcome a new user and returns to step 1.

Reminder. Implement the allert functionality. After user identification and during the conversation session, checks if there are any tasks due for the current user. If present, it returns only the activities for which the reminder is set, in order to be able to notify the user. Deciding as soon before you want to receive the reminder of the deadline is a system parameter.

1.2. Ros_audio_pkg package

This package contains all the nodes and modules needed to process the audio captured by the microphone.

1.2.1. Voice detection node

Enables audio capture via microphone. A *callback* function publishes the audio track on the *mic_data* topic once it has been acquired.

The node also provides the *ListenStop* and *ListenStart* services for muting and unmuting the microphone, respectively.

1.2.2. Speech Recognition node

It implements a *Speech Recognition* function by publishing only the audio tracks in which speech is present on the *voice_data* topic. Once an audio track containing speech has been acquired, it publishes its transposition into text (*Speech2Text*) on *voice_txt*.

1.2.3. Speaker reidentification node

Allows you to identify a speacker by voice. This functionality is managed through two services.

Identify. Acquired an audio track in which speech has been identified, it returns the identity of a user (if known) with its embedding, *None* otherwise.

SaveId. Allows you to associate an identity (label) to an embedding. Each association $(embedding, username)$ is saved in a pickle file to maintain the state of the embedding space. For each user, it guarantees at least `MAX_EMBEDDING` samples to maximize their identification.

The complete functioning of the re-identification procedure is described in chapter 3 (WP4).

1.3. Table_pkg package

Composed of the following modules.

Tablet manager node. When the user requests to view activities or categories, the intent propagates from RASA actions through the *show_data* topic to the Tablet manager node. This acts as a bridge to allow viewing on Pepper's tablet through the appropriate APIs. Specifically, it allows you to view a web page hosted on a Flask server on the local network.

Server Flask. A simple Web Server running on the local network used to display the activities and categories for the current user on an HTML page.

1.4. Pepper_nodes package

Inside this package there are several nodes already implemented to use some of Pepper's features. The original package was implemented by DIEM, supplied us for the realization of the project and available on GitHub.

Of the features implemented, those used in our application are:

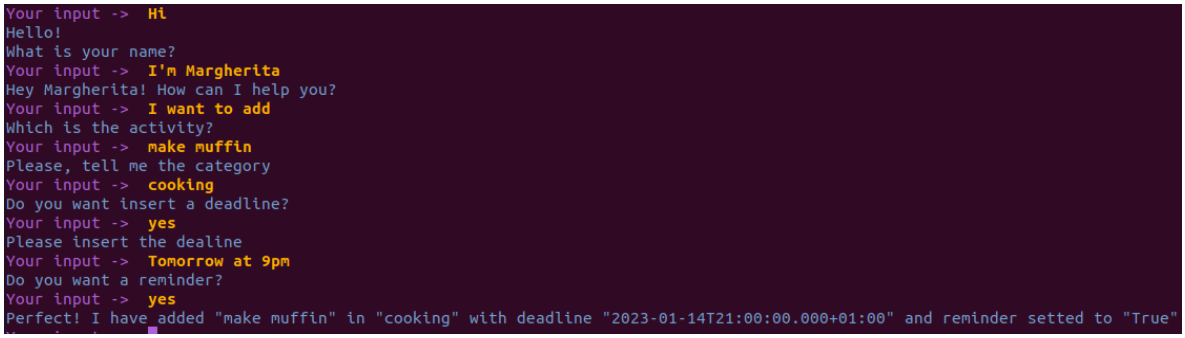
1. **WakeUp –** for the awakening and setting up of the robot;
2. **Text2Speech –** for the use of Pepper's synthetic voice;
3. **TabletNode –** to display web pages on the tablet browser.

2. WP3 – Rasa integration

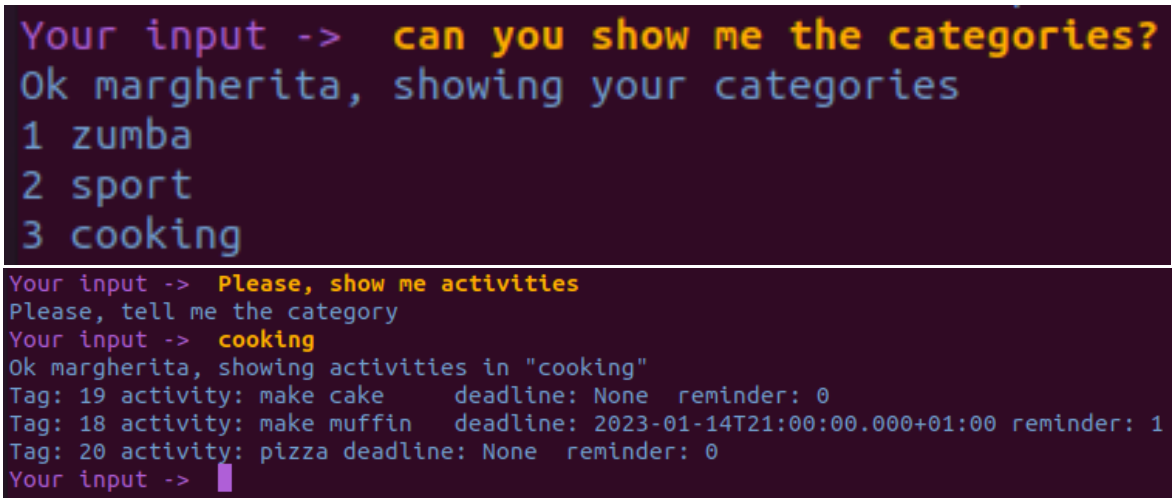
2.1. Working examples

Simple examples of chatbot operation are given in this section.

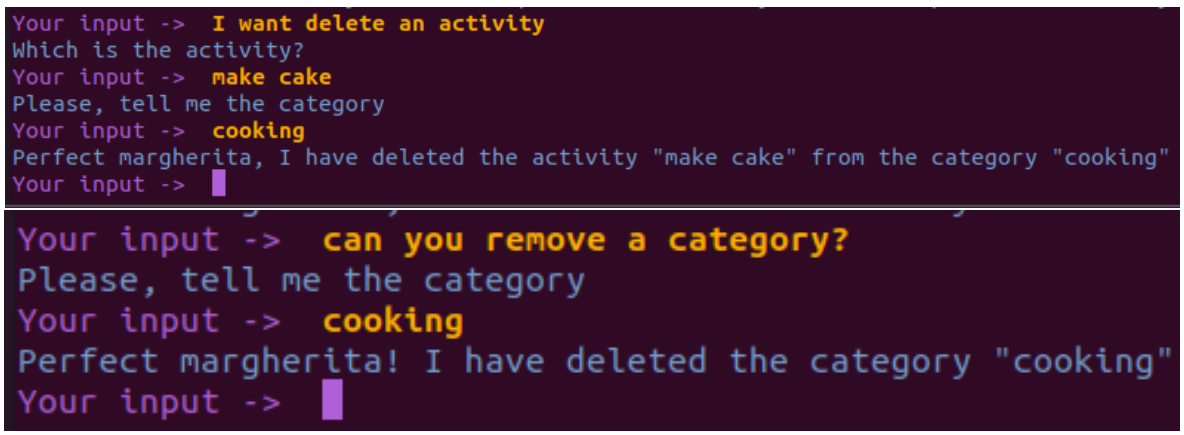
Action insert



Action show



Action delete



Action update

```
Your input ->  change an activity
Please, tell me the activity
Your input ->  make muffin
Please, tell me the category
Your input ->  cooking
To replace with?
Your input ->  make plumcake
The "make muffin" activity has been replaced successfully with "make plumcake" activity
Your input ->
```

An important feature of the bot is that, if it doesn't get enough confidence in detecting intent, it asks the user to rephrase the sentence. This avoids errors, false interpretations and the execution of unwanted actions.

```
Your input ->  by
I'm sorry, I didn't quite understand that. Could you rephrase?
Your input ->
```

This is a prime example, but it can also be realized in more complex use cases.

2.2. Explanation of the design choices

In this chapter we go deeper into dialogue management, in particular we analyze the most significant files for implementing the chatbot, discuss in detail the NLU pipeline, custom actions, and how we implemented some optional features for the bot, such as multi-user management.

One design choice was to use a database that would save users' todo lists in a single table. In this way, through simple queries, we can access the fields of interest and perform functionality. We use SQLite for database management.

tag	user	category	activity	deadline	reminder
10	roberto	shopping list	buy banana	2022-11-12T00:00:00.000+0...	1
11	francesco	shopping list	by pizza	NULL	0
12	lucia	reminder	solve exercise	2022-12-16T00:00:00.000+0...	0

From the image, we can see the fields of the table that contains the todo lists of the various users.

Domain File. Below we list the intents defined for the chatbot:

- **Greet -** handles initial greetings from the user's conversation
- **Goodbye** - handles end-of-conversation greetings from the user
- **Thankyou** - handles thank you conversations
- **Identification** - handles user identification
- **Update** - the user wants to perform the change action
- **Affirm -** the user's intention is to affirm
- **Deny** - the user's intention is to deny
- **Insert** - the user wants to perform the insert operation
- **show_categories** - the user's intention is to display his categories
- **show_activities - the** user wants to display his activities in a category
- **remove_category** – the user wants to remove a category
- **remove_activity** – the user wants to remove an active in a category
- **inform** - this intent identifies the information of interest to be provided on the RASA *form*
- **bot_challenge** - handles chatter from the user with the bot

Finally, we list the entities defined for the chatbot:

- **activity**
- **category**
- **time**
- **username**

2.2.1. NLU File

The training examples are grouped by intent in the *nlu.yml* file. The structuring of the example sentences has been modified, along with their respective intents, from the delivery of the midterm test.

In fact, before integrating the chatbot with the Speech2Text module, the system used training examples that included long sentences, within which multiple entities were specified. For example, for the intent of insert, the wording was as follows: "*I want insert [go out with friends] (activity) in [reminder](category) for [tomorrow](time)*".

In the light of the tests made with the Speech2Text module, it was decided to restructure the conversation into shorter sentences to improve their interpretability.

2.2.2. Config File

Pipeline. Below we report the chosen pipeline.

```yaml
pipeline:
  - name: SpacyNLP
    model: "en_core_web_md"
  - name: SpacyTokenizer
  - name: SpacyFeaturizer
  - name: RegexFeaturizer
  - name: CRFEntityExtractor
  - name: EntitySynonymMapper
  - name: SklearnIntentClassifier
    kernels: ["linear","poly","rbf"]
  - name: ResponseSelector
    epochs: 100
    model_confidence: "linear_norm"
    constrain_similarities: true
  - name: "DucklingEntityExtractor"
    url: "http://localhost:8000"
    dimensions: "time"
    timezone: "Europe/Berlin"
    - name: FallbackClassifier
    threshold: 0.3
```

It is a pre-trained pipeline based on the pre-trained SpacyNLP template for the English language (*en_core_web_md*) and completed by the following components:

1. **SpacyTokenizer** - to split user input text into tokens
2. **SpacyFeaturizer** - to create features from conversation input text, which can be used to identify user intents and entities
3. **RegexFeaturizer** - which uses regular expressions to extract information from input text
4. **CRFEntityExtractor** - which uses a recognition algorithm to identify specific entities in user input text
5. **EntitySynonymMapper** - used to map synonyms of a specific entity to a standard entity
6. **SklearnIntentClassifier** – additional module for classifying intentions
7. **ResponseSelector** - used to select the most appropriate response for a given user intent
8. **DucklingEntityExtractor** - entity extractor that uses Facebook's Duckling service to extract temporal and numeric information from input text
9. **FallbackClassifier** - for handling cases where the model is unable to identify the intention or entity with certainty, due to too low confidence.
   With the threshold parameter we specify the minimum confidence threshold for which the model must consider an answer as a fallback.

These components execute in sequential order, with the output of one component becoming the input for the next. In this way, the pipeline allows user input to be processed efficiently and accurately, providing usable information for the conversation model.

Policies. The evolution of the conversation is managed by various policy mechanisms. The three policies used are:

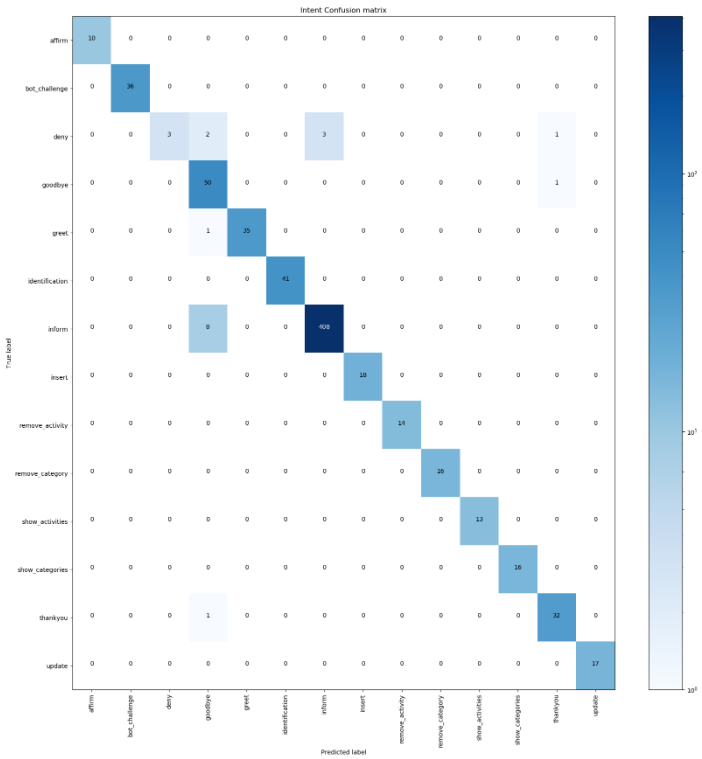
1. **RulePolicy –** the conversation evolves according to the *rules*, defined in the *rules.yml* file

2. **MemoizationPolicy** – uses *stories* to predict the evolution of the conversation if there are no matches with the defined *rules*.
3. **TEDPolicy** – uses machine learning to predict the next best action, if there is no match with what is written in the *rules* and *stories*.

These three policies, at each turn, predict the next action with a certain level of confidence, according to a hierarchy based on priorities.

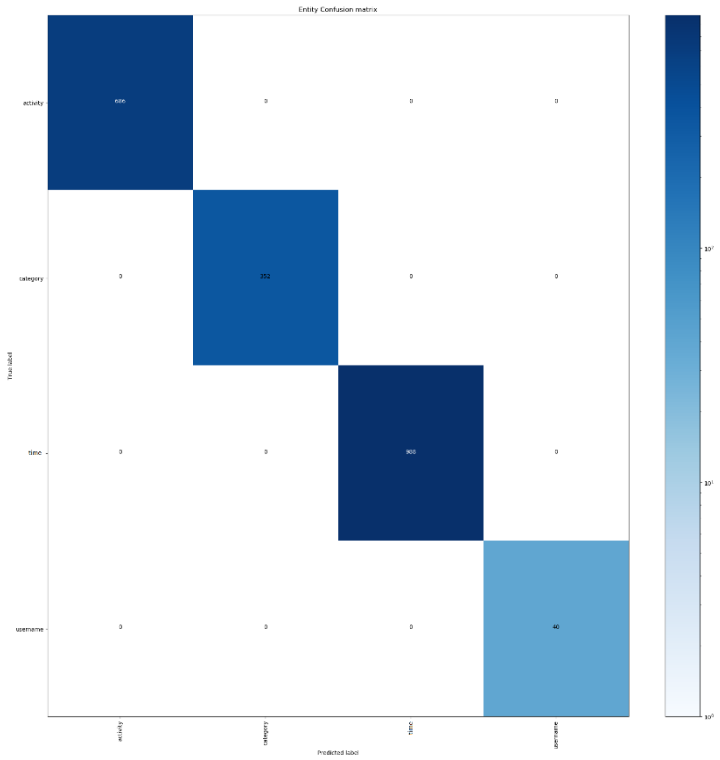
Performance. The matrices below are intended to highlight the performance achieved by the system with the chosen pipeline.

The intent confusion matrix follows.



The intents on which the system has the most difficulties are *deny*, *inform* and *thank you*.

Finally, the entity confusion matrix.



We note that since more *activity* examples have been provided, these are better classified than *category* and *username*. The use of the *duckling* extractor allows to obtain a high accuracy on the *time* entity.

2.2.3. Actions

The dialog system shall allow the user to add, remove, update and show tasks in the user's todo list. For this reason, the following six custom actions have been defined in the *actions.py* file:

1. **ActionInsert** – for inserting an activity and all its details into the database;
2. **ActionRemove** – for removing an activity or a category from the database;
3. **ActionShow** – to show activities in a category (or categories) for the current user;
4. **ActionUpdate** – to change the name of an activity in the database;
5. **ActionCustomReset** – to reset all slots, except "username" (useful for Dialog Management purposes);
6. **ActionStoreActivity** – to save the identified activity in a temporary slot (useful for Dialog Management purposes).

Special rules and stories, combined with forms, guide the user in completing the slots necessary for the correct functioning of the actions.

2.2.4. Manage multiple users

At the basis of multi-user management there is the choice of using a SQLite database for storing the todo lists associated with each user.

To guarantee access to the personal todo list, we have structured the conversation so that in the initial phase the user must necessarily identify himself. Following identification, the chatbot will keep the user ID in a special slot with which to access your todo list. At the end of the conversation, the slot will be reset preparing the system to welcome a new user.

The user ID is passed directly to the chatbot if this is correctly re-identified by the re-identification module, otherwise it is provided by the user upon request from the system.

2.2.5. Set deadline and reminder

When entering a new activity, the user is asked if he wants to add a deadline and a reminder that notifies him of the expiry of that deadline. This information is kept in a special slot and saved in the database.

The logic of the remainder is the following: the Reminder module accesses the information saved in the database and through simple queries checks which activities are due for the specified user allowing notification by the Dialogue Interface.

2.3. ROS integration

As specified in the section 1.1.1, for the Rasa-Ros integration a ROS service has been implemented which takes in input a string to be submitted to the chatbot and returns the chatbot's response in the form of a string. This mechanism is accomplished via HTTP requests by the ROS service with the RASA server.

3. WP4 – Re-Identification

In this chapter we discuss the re-identification module in detail. It mainly exploits two services named as Identity Service and SaveId Service.

3.1. Identify Service

All re-identification processes work using audio embedding. This embedding is obtained by calling the Identify service, which, taking a waveform as input, returns the embedding vector and the ID of the associated person as output.

3.2. SaveId Service

This service takes as input the id of the person who is speaking, the audio signal of his voice or his embedding vector. These parameters will be saved in the current embedding space and in a file, to make the data persistent.

3.3. Speaker Identification

For the re-identification has been used an embedding space, built by saving the identities of people who have interacted with the system and their respective embedding vectors.

The re-identification procedure is divided into the following phases:

- **Feature extraction:** the audio input signal is pre-processed to extract the features;
- **Prediction phase:** the features extracted in the previous step are passed to the *DeepSpeaker* DNN to create a new embedding which will be compared with all the embeddings saved using the *cosine similarity* metric.
  If the probability of belonging to the identity with the least cosine distance is greater than a pre-set threshold, the identity will be returned. Otherwise, *None* will be returned.

3.4. Design choices

To guarantee a correct identification process, it was assumed that the chatbot interlocutor is always the same and that a possible change takes place only after receiving the goodbye intent, indicating that the conversation is over.

Furthermore, in order to ensure greater efficiency, the interlocutor's embedding space is enriched only at the beginning of the conversation. This choice was made as it was noted that by continuing to update the embedding during the conversation, the workload for the chatbot became excessive, with a consequent drop in performance and an increase in errors.

Finally, we decided to insert a maximum threshold of seven samples per user as, from the experiments conducted, it proved to be a good compromise to obtain good identity discrimination. Once this threshold is reached, no more samples will be saved for that user.

3.5. Pepper speaking issues

A quiet environment must be ensured for the correct functioning of the device. Furthermore, since verbal interaction with Pepper is foreseen, if the microphone remains active it will also acquire the audio produced by Pepper, bringing the conversation, in the worst case, into a loop or to a system block.

To overcome this problem, two services have been included in the Voice Detection node (ListenStart and ListenStop) which deactivate the microphone when the robot is speaking and reactivate it when the user needs to interact.

4. WP5 & WP6 – Test

The first tests that we conducted were focused on verifying the correct functioning of individual project features. Only later, during the integration phase, we test the entire system.

4.1. Unit Test

4.1.1. Re-identification node

The tests conducted on the re-identification node are of two types:

1. Test on the implementation of the Predict and SaveId services (to verify their correct functioning).
2. Specific tests for user identification

The tests of the second type, apart from the implementation tests, took place in the following way:

1. **Setup -** Collection of `MAX_EMBEDDING` samples for two different users;
2. **Predict -** in rounds of three attempts, each user submitted an audio sample to the module marking the correct predictions obtained.

	Round 1	Round 2	Round 3
User 1	3/3	2/3	2/3
User 2	3/3	2/3	2/3

From the tests led we deduce that:

- a number of seven samples per user seems to be sufficient to discriminate between different identities;
- there are differences in performance (in terms of correct identification) from where user samples are acquired and where the system is working. Therefore, we recommend a re-acquisition of the samples used for re-identification if the environment of the implementation of the system and that of the acquisition of the samples should not coincide.

The tests were conducted by running a ROS node implemented by the *reidentification_test.py f*ile in *ros_audio_pkg*.

4.1.2. Speech2Text

To test this module, we launched the *speech2text.launch* file inside the *ros_audio_pkg* package which make sure that:

- a correct calibration of the environment is performed
- a correct start of the recording is done
- the functioning of the Google voice recognition model takes place correctly.

4.1.3. Pepper nodes

After starting the Pepper nodes, in the *pepper_nodes* package, via *pepper_bringup.launch*, to test the correct functioning of the implemented services, we used the command:

```
rosservice call [nome servizio] [parametri]
```

4.2. Integration test

4.2.1. Ros and Rasa

Upon delivery of the midterm, we were provided with the implementation of a ROS node with which to be able to send from the terminal the messages we wanted to submit to the chatbot running on the RASA server.

To complete the integration, we have created a *.launch* file with which to start all the services necessary for the chatbot to function (rasa server, rasa actions, duckling server) including the integration nodes provided.

With this setup we were able to verify the correct integration between RASA and ROS simply by conversing with the chatbot via the ROS node.

4.2.2. Chatting with Pepper

Below are the tests we were able to carry out on Pepper.

We managed to successfully integrate Pepper's Text2Speech module with the Dialog Interface. We then started a conversation by giving the chatbot a voice through Pepper and submitting commands to it via an external microphone. The conversation touched on the insert, remove and update features required for the chatbot. However, we were unable to test a conversation that incorporated more than two or three commands due to issues with the external microphone.

We were also able to successfully use the *LoadUrl* service to view the web pages we created. Then, following the view request, the tablet successfully showed a table containing activities or categories for a user.

The tests carried out on the re-identification module with the external microphone supplied to us were subjected to excessive "noise" and not sufficient in number to be able to declare correct functioning.

4.3. Final test (without Pepper)

With these tests we tested the entire logic of the application by simulating the Speech2Text and tablet services provided by Pepper with a speech synthesizer and a computer browser, respectively.

Running these tests requires setting the `PEPPER` variable to `False` in the *config.py* file. The tests conducted are:

TEST 1.a. In this test we verify that the re-identification of a user A is successful.

TEST 1.b. In this test we verify that the re-identification of two different users. In particular, we re-identify with user A and start a new conversation by re-identifying another user B.

TEST 2. This test aims to verify that the integration of all the modules has taken place correctly and that the system respects the required logic.

The test is structured as follows:

1. Re-identification of a user A
2. Adding an activity in a category C1
3. Addition of another activity in category C1
4. Addition of an activity in a category C2
5. Showing categories for user A
6. Showing activities in category C1
7. Updating of an activity in category C1
8. Delete of an activity in category C1
9. Delete of category C2
10. Goodbay

At the end of this test we were able to verify:

The correct functioning of the chatbot and its features (insertion, removal, display and update)

- Correct integration between the chatbot and the ROS environment
- Correct integration between the Voice Detection, Speech2Text and Re-identification modules with the logic implemented in Dialog Interface.

TEST 3. With this test we have verified the *reminder* functionality and that its integration with the system logic has taken place correctly.

Recall that the functionality under consideration allows you to notify the user when you are close to a deadline set for a task.

In the test, user A inserts a new activity by setting a deadline and the notification function (*reminder*) in a category of his choice. The test ends successfully when, approaching the deadline, the system correctly notifies the expiring activity.