Cognitive Robotics

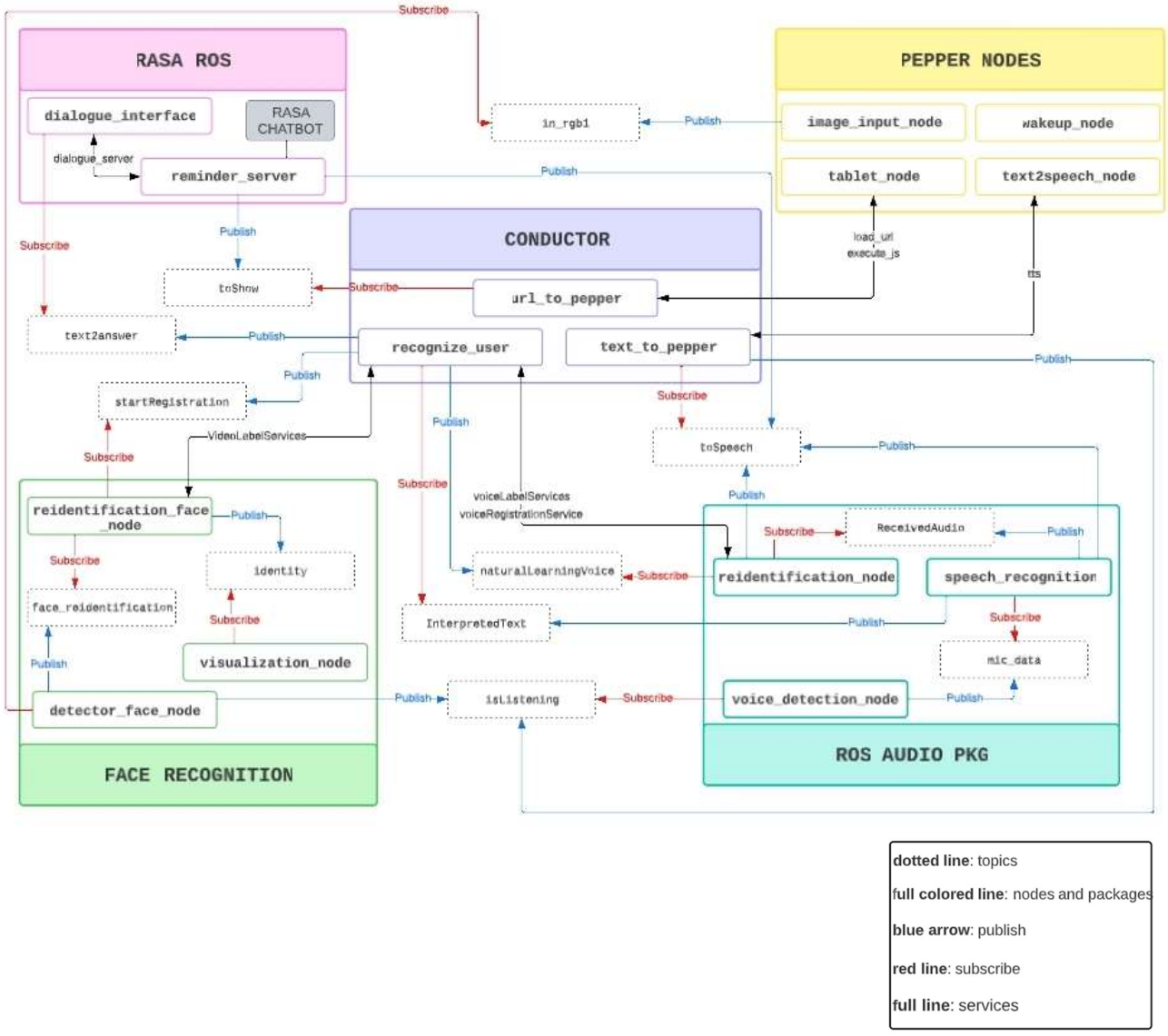
Final Project – 2022/23

Camilla Spingola	0622701698	c.spingola@studenti.unisa.it
Mattia Marseglia	0622701697	m.marseglia1@studenti.unisa.it
Vito Turi	0622701795	v.turi3@studenti.unisa.it
Sica Ferdinando	0622701794	f.sica24@studenti.unisa.it

UNIVERSITÀ DEGLI STUDI
DI SALERNO

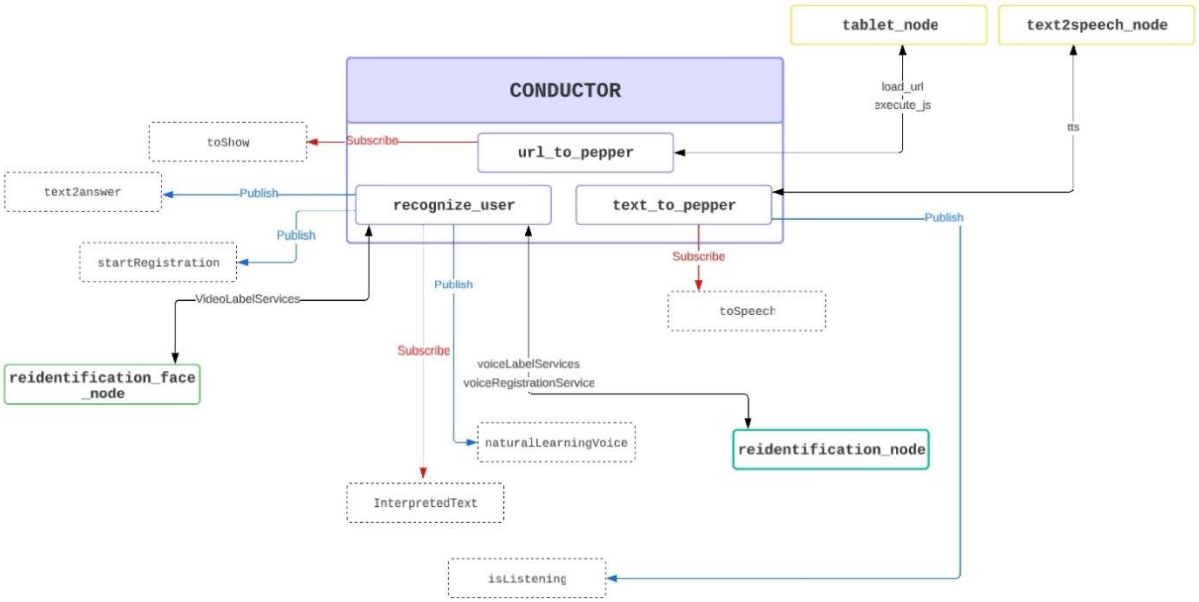
1 WP1 & WP2 – Architecture design and Implementation

This section shows the application architecture and explains all its parts.



1.1 Conductor package

The conductor package plays a central role in the functioning of our application. It contains ROS nodes that gather informations from other node in external packages, such as “face_reidentification” or “ros_audio_pkg”, are used to communicate with a RASA chatbot, and, once received responses, perform actions through nodes of “Pepper_nodes” package. In short, the conductor package acts as the hub for decisioning and coordination within the application. All the nodes in this package, as it is explained below, have a specific functionality like combining the information acquired from audio and video to recognize the user, and allow him to have a flowing conversation with Pepper, exploiting the implemented RASA chatbot as the operative mind of the answers and of the actions connected to user’s requests.



1.1.1 Text_to_Pepper

The text to pepper node is responsible for sending text to the Pepper robot for it to read out loud through the “tts” service. Additionally, it manages the activation and deactivation of the microphone during speech. Specifically, the microphone is deactivated while Pepper is speaking to prevent it from picking up its own voice and engaging in a self-conversation. It is done publishing a Bool value on “isListening” topic, to inform subscribers that Pepper is talking or not, indeed the function “say” called through the TTS service provided by Pepper is a blocking function, which at the end returns an acknowledgment which can be used to deactivate the microphone before Pepper starting to talk and reactivated it as soon as Pepper ends to speech. This allows the system to ensure that Pepper does not listen to its own speech and avoids any confusion.

1.1.2 Url_to_Pepper

The web browser node is responsible for sending URLs and JavaScript code to Pepper. Specifically, the node is subscribed to “toShow” topic, from which obtains crucial messages related to user’s requests, to displaying and elaborate information shown on Pepper’s tablet. So taken a message, it checks if the message contains the keywords "js" or "reload" and if so, it triggers the execution of two different JavaScript code, once to check for small update in the page already shown yet, the other to reload the entire page, both these actions are done by calling “execute_js” service provided by Pepper. If the message does not contain these keywords, the node knows that the message is an URL, so calls “load_url” service that loads the corresponding page into Pepper's web browser, showing it on its tablet. This node plays an important role in allowing the application to display web pages and execute JavaScript through Pepper’s tablet, providing additional functionality and interactivity to the system, above all simplifying interactions with the user in those situations in which the visual component is essential and the vocal component would not be sufficient, such as for displaying users’ activities.

1.1.3 Recognize_user_node

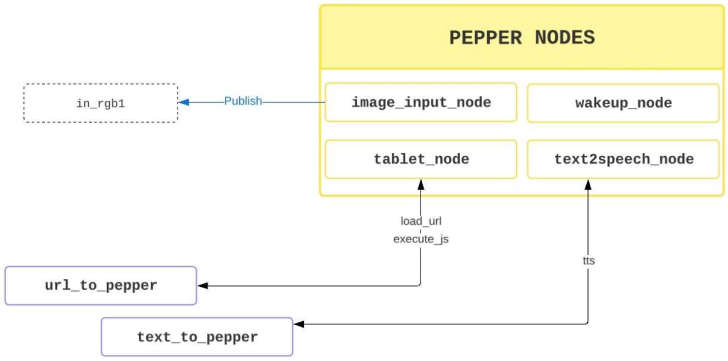
The identity verification node is a crucial component in the application's security system, it's subscriber of "InterpretedText" topic where are broadcasted all sentences extracted from ros_audio_package. When this node read a sentence on this topic, first requests information on person's identity to facial recognition and voice recognition packages through a Client-Server mechanism, using "VideoLabelServices" and "voiceLabelServices" services, and obtaining lists of probabilities of the person being one of the already registered users on the platform.
Then the node combines face's and voice's probabilities, giving more weight to the probabilities obtained from the facial recognition package, creating a new list of probabilities. The user ID associated with the highest probability is then determined. If the probability exceeds a certain threshold, the person is considered to be correctly recognized. Otherwise, the user registration process is initiated, which is managed by the corresponding packages, using the "StartRegistration" topic to demand the acquisition of new video sample and the "voiceRegistrationService" to demand the acquisition of new audio samples, in order to register the identity of the new user. Additionally, for the audio part, has been implemented a Natural Learning procedure that, in cases of uncertain recognition (under a fixed threshold), saves other audio samples in order to make the system more robust in recognizing that speaker identity. This procedure is managed by the corresponding audio package through "NaturalLearningVoice" topic.

1.1.4 Test_camera_node and Test_video_saver

These two nodes are two "utility" ones; the first one read from the camera and publish frames to "test_in_rgb1" topic, the second one read from "test_in_rgb1" through waits for the messages.It takes images, resizes them to a specific resolution (640,480) and saves them. Those images have been used to create a sort of database, used by the facial recognition package to allows to our model to recognize our 4 faces. It is not more used, infact in real time, when a new user starts the registration procedures, face_reidentification_node implements the "append" of new user features in the db.

1.2 Pepper nodes package

The Pepper package is essential in our project, it allows us to use Pepper robot's skills and features probrabilities. Is composed by a group of nodes with a client-server service architecture so whoever want to use a certain functionality of the robot call the specific service.



These methods are implemented using NaoQi as a proxy to the Pepper function. Most of these nodes can leave Pepper in an incoherent state, so they have a shutdown callback to avoid this issue (for example, the Start and stop node has a shutdown callback which puts Pepper in a rest position).

1.2.1 Image_input_node

The image input node is used to read the video stream from Pepper's camera. It opens a video stream with "ALVideoDevice" service to register the Pepper camera, then it publishes the stream on a "in_rgb1" topic. The parameters for resolution and fps are the default ones. The subscriber of this topic is the detector node in the face recognition package, in order to start identifying the aspects of interest within the image.

1.2.2 Tablet_node

The tablet node is used to control the Pepper tablet, using the “ALTabletService” of the robot. The two services instantiated in this node are the “load_url” and “execute_js”, the first allows to load a web page on tablet’s screen, the second to inject a js script on a loaded page. The node satisfies client requests, in our case those coming from the “url_to_Pepper” node in the conductor package. These services let us to display the web Page for to do lists on the tablet in terms of categories and activities saved for an user with all the related information about deadline, status activity and deadline. The service “execute_js”, is important to modify the table shown in a responsive way, so that, while the user is talking, if he is viewing his categories or his activities and requests a change on these, would view the changes live.

1.2.3 Text2speech_node

This node is used to animate the robots in terms of talking an animation. it uses the “ALAnimatedSpeech” and “ALTextToSpeech” services using Italian as language of the robot. The node satisfies client requests, in our case those coming from the “text_to_Pepper” node in conductor package.

1.2.4 Utils

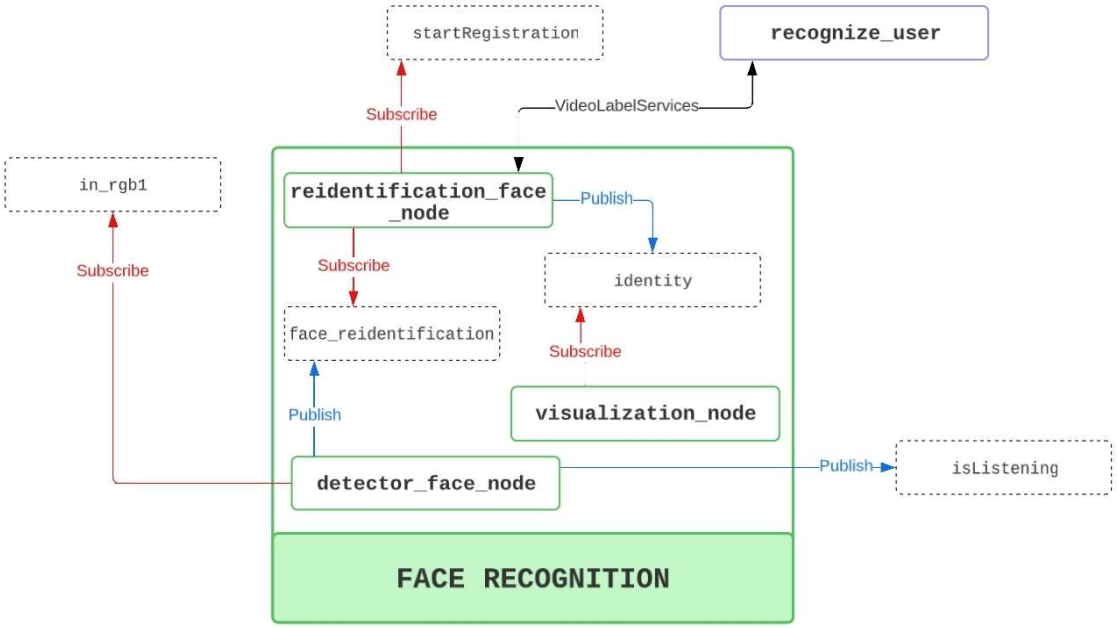
This file contains a class that creates a qi Session using ip and port parameters provided to its constructor. This class is used by every node in Pepper package to use the AL services and connect to Pepper OS.

1.2.5 Wakeup_node

This node is used to control the Pepper posture; it uses the “ALMotion” and “ALRobotPosture” services. We use it to “wake up” Pepper, that sets motors on and sets the robot posture to the initial position, and to set the rest position.

1.3 Face Recognition package

The Face Recognition package is responsible for identifying users based on their facial features. The package comprises of several ROS nodes that work together to perform the facial recognition task. The package starts by extracting the face from the image, then it passes the face through the face recognition node, which compares it against a pre-trained dataset of faces and returns the identity of the person with the highest probability of match. This package is crucial for the system's security and



personalization features, allowing it to recognize and interact with specific users.

1.3.1 Detector Node

The face detection node is responsible for identifying faces within an image. It uses a deep neural network provided by the cv2 library, which outputs a bounding box, called FaceBox, around the detected

face. Reading from “in_rgb1” topic, if a face is detected, the node publishes the box's coordinates and the original image on “face_reidentification” topic for identification task. If no face is detected, message and image are not transmitted. This node also publishes to “isListening” topic, this notification allows to indicate to audio package whether a person is present or not, allowing the system to automatically activate or deactivate the microphone accordingly. This node plays a vital role in the facial recognition process, enabling the system to accurately locate and identify faces within an image.

1.3.2 Face Reidentification Node

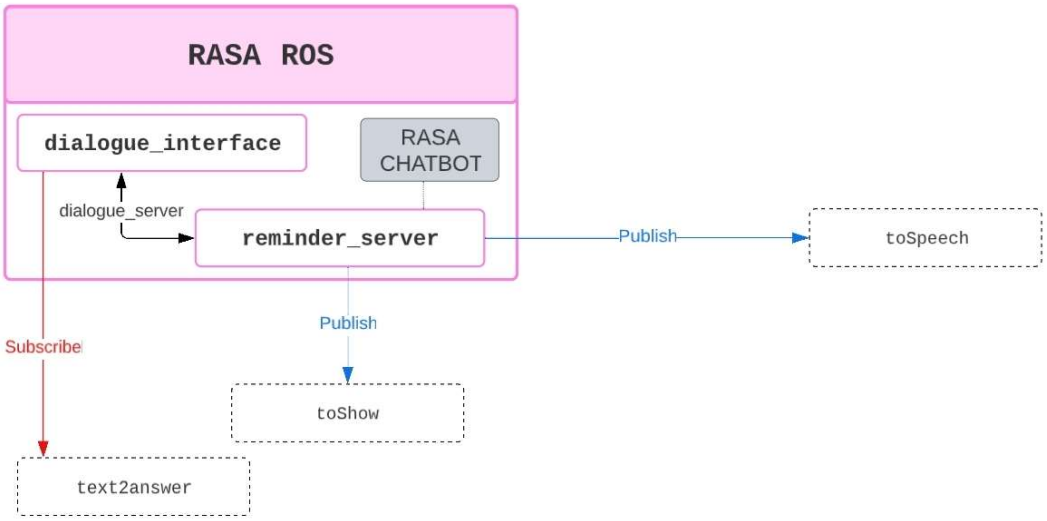
The face re-identification node is responsible to recognize a person as a registered user on not and identify his identity. This is done by giving probabilites the face detected in the scene belongs to an already saved identity. In particular, from “face recognition” topic, it takes each image containing faces previously detected by the Detector node, along with the bounding box information, then extracts the faces from this image, pass these to a deep neural network that is trained to recognize and identify the individuals based on their facial features, and in this way obtain the extracted features. Finally, through “batch_cosine_similarity” evaluates the distances between the point in the features space corresponding to the face detected and saved point corresponding to database identities’ images in terms of cosine distance; then by “dist2id” which construct probabilities associated to each registered user, achieve the proposed intent. The node has an independent flow, it continuously read end evaluate probability associated with each image obtained from the detector, at the same time it offers a server service called “VideoLabelServices” for whoever wants to know these values. It also publishes on “Identity” topic these values. This node plays a crucial role in the facial recognition process, enabling the system to accurately identify and verify the identity of an individual based on their face, in order to manage different users and to establish a more confidential conversation with the user.

1.3.3 Visualization node

The face visualization node is a development tool that was used during the testing phase of the facial recognition package. It receives the original image, the bounding box information, and the identified person's identity from “identity” topic. This node is not intended for use in the official release of the product and is only used for testing and display purposes. It displays “recognized user” if the person currently present in front of the camera is associated with one of the persons registered in the database (four team members in our case), otherwise it display “unrecognized user”. This node provides developers with a quick and easy way to visualize the performance of the facial recognition system during the testing but also development phases, making it easier to identify and troubleshoot issues.

1.4 Rasa Ros package

In this section, there is an explanation of the ROS nodes that are used to communicate with the RASA chatbot. From the conductor package arrives structured messages to these nodes, which contain the phrase to be analyzed and answered by the chatbot, together with the label of the user who is speaking. Through a node the phrase to be analyzed and the corresponding user’s label is then forwarded to a RASA server for analysis. Subsequently another ROS node receives and manages the asynchronous response given by the chatbot that the user is waiting for. This set of nodes plays a vital role in allowing the system to understand and respond to user input through the RASA chatbot, allowing the user to have a flowing conversation.



1.4.1 Dialogue interface

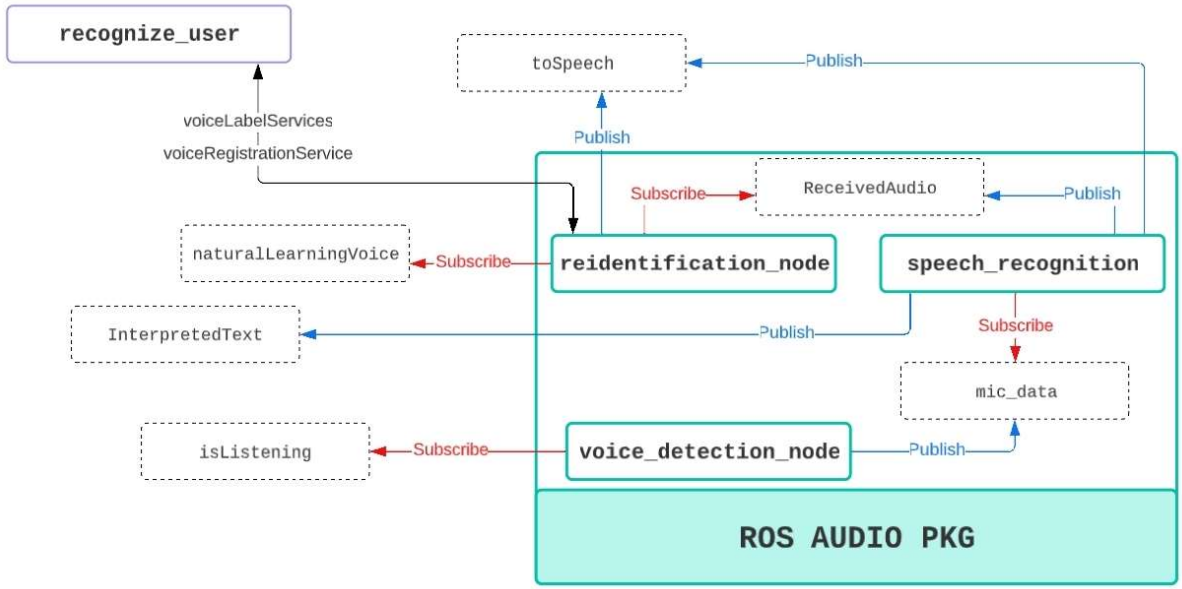
This node is responsible for forwarding user input and label information to the server who talks with RASA chatbot for analysis. It stands as a proxy, when the node is started, it initiates a communication session with RASA server by sending "/session_start" messages through “dialogue_server” service, managed from “reminder_server” node. This notifies RASA that a new conversation is beginning and allows it to retrieve any reminders that are close to expiring from the database. Subsequently this node waits for messages on "text2answer" topic, it forwards these messages, always through the “dialogue_server” service, to the “reminder_server” node that talks with RASA server. If the communication does not work, it returns an error message. This node facilitates the exchange of information and ensuring smooth communication between the two systems.

1.4.2 Reminder server

This node is responsible for sending input requests to the RASA chatbot and manage eventually response. The input to send to RASA coming from client requests from the “dialogue_interface” node, and it has an app that listens on a specific local port (5034) for messages sent by the chatbot and, upon receiving a message, it decides whether to post the response on the "toSpeech" topic and/or post on the "toShow" respectively to take advantage of Pepper's voice or tablet. The node also manages the presence or absence of JSON-formatted information in the message. As previously explained, in some situations, RASA sends additional information in addition to the simple answer, which is necessary for the correct display of the To-Do list. This node plays an important role in the communication between the application and RASA, enabling the system to process and act on the chatbot's responses in real-time.

1.5 Audio Recognition package

The ROS AUDIO package is responsible for recognizing individuals based on their voices and converting speech to text for laters processing and analysis. The package comprises of several ROS nodes that work together to perform the speech recognition and conversion tasks. The package is designed to work independently for testing purposes.
This package is able to run with different type of microphones managing the audio source differently depending on the instrument used. This package plays an important role in the system's personalization features, allowing it to recognize and interact with specific users.



1.5.1 Voice Detection

The audio capture node is the starting point of the ROS AUDIO package. It is responsible for identifying the audio capture device in use, and it makes a distinction between the use of a simple one-way microphone and the directional multi-array microphone supplied for the project.
During the testing phase, it was found that the incoming streams from the multi-array microphone was not synchronized, resulting in an echo that made it difficult to understand the spoken words. To solve this issue, the node sends the library a sampling rate multiplied by the number of microphones available, and then performs a down-sampling of the acquired audio sample. This allows the system to separate the audio streams of the individual microphones. The audio stream that is then propagated in the package is chosen based on the sum of the values stored within the flow, thus choosing the "lauder" one. This choice is correct because all the microphones hear the same thing, with the difference that the one closest to the source will have higher values. The chosen audio stream is then published on “mic_data” topic for further processing.

1.5.2 Asr

The speech-to-text (STT) node read audios from "mic_data" on which publishes the voice_detection node, and converts it into text. In fact, obtained audio is passed to the STT API of Google, which analyzes it and returns the corresponding text if it is recognized. If the audio is not recognized, the API throws an exception in which is published “scusa non ho capito” on “toSpeech” topic. If the audio is recognized correctly, the corresponding text is published on “InterpretedText” topic while audio on “RecivedAudio” ones. This node plays a vital role in the system's speech recognition functionality, enabling the system to understand user spoken user input and also to respond to him in case of difficult in understanding.

1.5.3 Speaker identification

The speaker recognition node receives the audio from "RecivedAudio" topic, and its task is to identify the person wo is talking based on their voice.
Like the Face Re-identification node, the audio is analyzed by a deep neural network, obtaining extracted features of the sample. Through “batch_cosine_similarity” are evaluated the distances between the point in the features space corresponding to the current audio sample and saved point corresponding to database identities’ voices in terms of cosine distance; then by “dist2id” is constructed probabilities the new audio sample belongs to an already registered user. On these probabilities we don’t apply the threshold value to predict the identity, because we use a combination of audio and video to make prediction, as explained above, and so on it we use the threshold. The probabiliteies obtained during this elaboration are obtained after the client requests through “VoiceLabelServices”. The service “voiceRegistrationService” is delegated to start the voice registration procedure during which predefined phrases that the user must repeat are published on the “toSpeech” topic. Then the obtained audio associated to these phrases are analyzed, the features are extracted and are saved into the voices

database with the label intended for the new user. Finally, this node is subscribed on “naturalLearningVoice” topic, that is conceived to be able to add new samples also for already registered user whose voice prediction is not confident over a threshold value.
This node plays a crucial role in the system's personalization features, allowing it to recognize and interact with specific users based on their voice.

1.6 VisionMsgs package

In summary, this package establishes a standardized communication system for computer vision and object detection within the Robot Operating System (ROS) framework. Specifically, the messages defined in this package have been utilized in the face recognition package. The "detection2D" and "detection2DArray" messages have been employed to facilitate the exchange of information related to face detection between nodes. This package is a useful tool for coordinating and integrating various computer vision and object or face detection efforts within the ROS ecosystem.

1.7 No Pepper nodes package

1.7.1 Test_camera_node

This node is used for webcam simulation, it’s useful to test the face detection and recognition capabilities of the system without
Pepper robot. It captures frames from a PC webcam and sends them to the appropriate topic “in_rgb1” for processing by the face recognition package. This allows developers to test and debug the facial recognition functionality without the need for the physical robot, improving the development and testing process.

1.7.2 Test_tts_node

This node is used for text to speech simulator, it’s useful to test the system without Pepper robot. It easily prints the phrase that should be said.

1.7.3 Test_url_node

This node is used for text to speech simulator, it’s useful to test the system without Pepper robot. It easily prints the requests sent to tablet.

1.8 Services

1.8.1 tts

This service, when called, allows communication with the NAOqi “ ALAnimatedSpeech” service. This service receives an input text, which is then read by Pepper, along with animations during the reading process. This service enables the system to interact with the user through speech, providing a more natural and engaging experience.

1.8.2 load_url

The tablet service, when called, allows communication with the NAOqi “ALTabletService” service. This service specifically calls the function that sends the URL to be loaded on the tablet. This service allows the system to interact with the user through the tablet, providing a more interactive and rich experience. The system can use the tablet to display information and media, and to receive input from the user.

1.8.3 execute_js

The web service, when called, allows communication with the NAOqi “ALTabletService” service. Specifically, this service calls the function that performs a js injection on the current web page. This service web page a lot more responsive. The system can in this way execute JavaScript and make dynamic changes, allowing for an interactive and responsive user experience.

1.8.4 dialogue_server

This service, when called, allows communication with the RASA chatbot. This service receives as input the text to be processed by the chatbot and the user ID, which will be used to customize the chatbot's

responses. This service enables the system to interact with the user through the chatbot, providing a more natural and engaging experience. The system can use the user ID to provide personalized responses based on the user's previous interactions with the chatbot.

2 WP3 – RASA Integration

2.1 The Chatbot

A ROS dialogue system that allows the user to manage a to-do list has been developed. The user can insert and remove activities from the list, activate a reminder when the deadline is approaching. Each element in the list is identified by a tag (identifying the activity), a deadline, and a category (e.g. CR course, sport, personal, etc.).
The dialogue system allows the user to:

- View the activities in the to-do list
- Insert a new activity in the to-do list
- Remove an activity from the to-do list

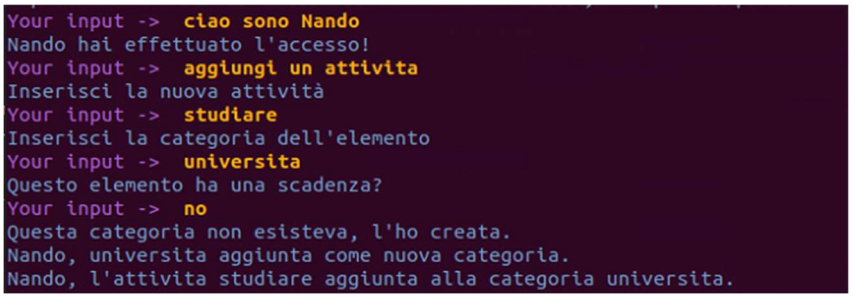
Additional features that have been implemented include:

- Managing multiple users
- Managing multiple categories of to-do lists
- Updating an activity in the to-do list

The system is designed to be user-friendly and interactive, allowing the user to easily manage their tasks and deadlines. The use of ROS allows the integration of this system with other modules of the project, making it a versatile and flexible solution for managing tasks and reminders.
The implementation of the chatbot, has been made through RASA, that is an open-source chatbot framework that allows developers to build conversational AI assistants using natural language understanding (NLU) and dialogue management. Rasa is built on cutting-edge machine learning techniques and can be integrated with various messaging platforms. The framework is designed to be highly customizable, allowing developers to define their own conversational flow, NLU models, and responses. The framework is particularly useful for creating chatbots that can handle multiple intents and tasks, making it ideal for creating multi-purpose chatbots such as the one we develop.
Through RASA framework our team was able to create a chatbot that is capable of managing a multi-class and multi-user ToDo list. The chatbot is designed to understand and respond to various requests related to tasks' management, such as creating, deletion, modifying, and completing tasks. Additionally, we implemented a real database to store all the user's data, ensuring that the chatbot can provide personalized responses and keep track of each user's tasks and lists. The chatbot designed to be compliant with request required also implementing certain modification's action which are inherently complex.
In fact, about the activity updating different possibilities have been conceived, for example the change of the name, category and deadline of an activity, including also the modify of the status activity (completed or not), and also the addition of a reminder for an already inserted activities. Also, the different to-do-lists (categories) can be modified, removed or added. The show operation could be characterized, for example asking to see all the activities that respect a particular constraint (completed or not).
About the reminder was managed the possibility for a user to receive the notification even if the chatbot is restarted and even if he is engaged in another conversation flow even I with another user.
Despite these challenges, the RASA framework provided solutions to manage these issues, thus enabling the development team to resolve these critical problems in a coherent and consistent manner with the entire project.

2.1.1 Working example

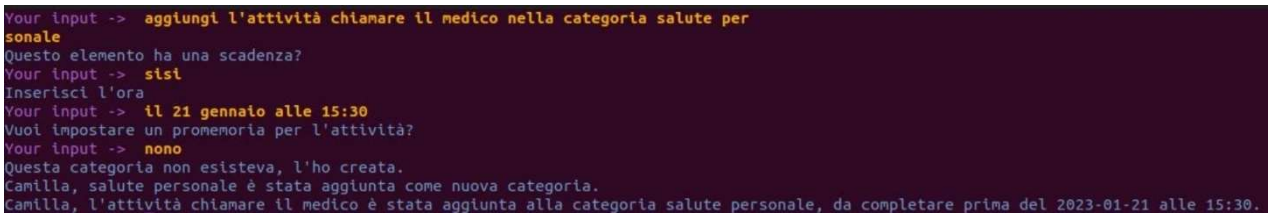
In this paragraph, we present some simple examples of conversation with the chatbot, obtained through rasa shell. The bot can handle more complex cases, as explained above, but these examples are meat to



show the ability of the chatbot ability to accurately recognize the user's intent and fulfill their requests, having a flowing conversation.

The bot can correctly recognize the user's intent and the different entities within the text, even in complex contexts such as modification requests.

The management of dates has been delegated to Duckling, which is run in a Docker environment. Duckling comes with modules that can parse temporal expressions in English, Spanish, French, Italian, and Chinese, which makes it a powerful tool for handling date-related tasks. Overall, the bot demonstrates an elevated level of understanding and capability in fulfilling user requests, making it a valuable addition to the system. Because of this reason we don’t need to training our model on the date/time extraction.



The conversation is guided by the various forms, in which the bot tries to gather all the necessary information for the operation by asking the user the necessary information if these are not provided immediately. In the example shown below, the bot understands the modify intent, so it may ask all the parameters necessary to identify the action to be modified like the deadline and also ask the new name of the activity.

Following is reported an example of update, specifically to set as completed an activity.

For all the removal operations is also asked if the user is sure to complete this action.

```
Your input ->  mostra attivita
Nando, ecco le tue attività:
                                Attivita          Categoria           DeadLine          Completata
                1           camminare            palestra    2023-01-15 00:00                   1
                2            studiare          universita               None                   0
.
Your input ->  rimuovi camminare in palestra per domani
Sei sicuro di completare questa azione!?
Your input ->  si
Nando, l'attività camminare è stata rimossa dalla categoria palestra .
Your input ->  mostra attivita
Nando, ecco le tue attività:
                                Attivita          Categoria           DeadLine          Completata
                1            studiare          universita               None                   0
```

As explained above also the reminder has been managed, but the real notification of the reminder could become visible to user only after the integration with ROS.

```
Your input ->  ricordami di chiamare mamma
Inserisci la categoria dell'elemento
Your input ->  personale
Inserisci l'ora
Your input ->  tra 10 minuti
Nando, l'attività chiamare mamma è stata aggiunta alla categoria personale, da completare prima del 2023-01-14 alle 12:10.Te lo ricorderò
```

When the bot is not able to understand the user's requests then it asks them to repeat it to avoid errors. This feature is particularly useful in cases where the user's input is unclear or ambiguous. This feature can be implemented in more complex use cases as well, providing a more robust and user-friendly experience.

```
Your input ->  regressione o classificazione?
Mi dispiace, non posso aiutarti, puoi ripetere?.
```

Additionally, a unique intent has been implemented to allow the user to restart the conversation from the beginning if it has deviated from the expected logical flow. This feature enhances the overall user experience by providing a way to correct any misunderstandings or missteps that may have occurred.
For the activity or category update it has been necessary to use the roles “old” and “new” of the different entities. In particular it has been necessary for category, activity and deadline, so that during an update operation different slot will be filled for the parameter to be modified and for the new parameter so that there cannot be risks of overlapping.

2.2 Solution Description

2.2.1 NLU

First, the intent and entities were identified. The intents used are:

- *greet:* handles starting conversation greetings from user.
- *goodbye:* handles ending conversation greetings from user.
- *add_item*: the user wants to add a new activity in a ToDo list.
- *add_category*: the user wants to create a new category of ToDo list.
- *set_status_activity*: set the activity status as completed or not.
- *presentation*: the user wants to login or create a new account
- *view_activities*: the user wants to see his activites in the ToDo lists.
- *inform*: this intent indicates that information of interest has been provided during a form.
- *affirm*: handles intent of affirmation by users
- *deny*: handles intent of not affirmation by users
- *modify_activity_deadline*: this intent indicates that the user wants to modify the deadline of an already existing activity.
- *modify_activity_name:* this intent indicates that the user wants to modify the name of an already existing activity.
- *modify_activity_category*: this intent indicates that the user wants to modify the category of an already existing activity.
- *remove_item*: this intent indicates that the user wants to remove an activity from the ToDo

lists.
- *remind_me_of*: this intent indicates that the user wants to create a new activity with a reminder.
- mood_great: this intent indicates that the user is happy with the work of the chatbot
- *mood_unhappy*: this intent indicates that the user is not happy with the work of the chatbot
- *bot_challenge*: handles intent of gather information about the bot by users
- *remove_category*: this intent indicates that the user wants to remove a category of ToDo list.
- *view_activities*: the user wants to see his category ToDo lists.
- *help*: handles intent of gather help from the bot by users
- *modify_category*: this intent indicates that the user wants to modify the name of an already existing category.
- *clean_activity*: this intent indicates that the user wants to remove all his complete actions from the database
- *nlu_fallback*: this intent handles situations where the bot doesn't understand it
- *EXTERNAL_reminder*: intent necessary to manage the expiration of a reminder
- *ask_name*: this intent indicates that the user wants to verify that the bot recognized him correctly

The identified entites are:

- *activity*;
- *activity_status;*
- *category;*
- *name;*
- *time;*

For each intent, in the *nlu.yml* file, there are examples of phrases and questions that a user might ask. Also, for each intent, if are present, all the entities have been identified, with respective roles. It was important to provide different configuration of phrases in order to make the system able to generalize. For the updating intents, have been provided phrases in which the same value of the entities that will be changed are once as “old” and once as “new”, for generalization and robustness reasons.

2.2.2 Stories

The nlu.yml file in a RASA chatbot is used to define the natural language understanding (NLU) components of the chatbot. These components are responsible for understanding the intent and entities in user inputs and mapping them to the appropriate actions in the chatbot.
The intents in this NLU file have been carefully crafted to cover all possible user intentions. An inform intent has been included to handle situations where the user needs to provide a single entity. All examples must be written in this file to inform the bot of the intents and entities contained within them. Special attention should be given to the modification intents, which require the same entity to be obtained twice, representing two different things (in this case, the old and the new values). Thankfully, this version of RASA is equipped to handle this through the use of "roles." Roles have been defined for the old and new values of the specific entity being modified.
In RASA, synonyms are used to identify different ways in which a user might express the same intent. By defining synonyms in the NLU configuration file, we can increase the chances of correctly identifying the user's intent, even if the user phrases their request differently. Synonyms can be defined for both intents and entities, and can include single words or phrases. Synonyms can also be used for entities, such as recognizing different variations of a location name like "New York City" and "NYC" as the same entity. Indeed, synonyms have been utilized to account for different ways a user may express completing or not completing a task, while still resulting in the same value in the Slot.
Overall, the nlu.yml file serves as the backbone for the chatbot's ability to understand and respond to user inputs.

2.2.3 Form

In RASA chatbot, a form is a specific type of dialogue management component that allows the bot to collect and validate specific information from the user. Forms help the bot to understand the user's intent and extract the necessary information to complete a task or fulfill a request. The forms are defined in the domain file and can be triggered by specific intents or actions. The user's responses are stored in a tracker and can be used to fill slots or trigger actions. The bot can also prompt the user for missing information and validate their input before proceeding with the conversation. The use of forms allows for a more structured and efficient conversation flow and helps the bot to understand the user's needs better. In general, the forms used are very basic, customized utterances have been defined according to the form that was running at that moment, in this way it is easier for the user to understand what is required at that specific moment. The only "complex" forms are those related to the modify, specifically the ask have been replaced with real custom actions that allow us to manage and ensure the correct filling of the slots.

2.2.4 Pipeline

We used the following pipeline for text processing.

```
pipeline:
  - name: SpacyNLP
  - model: 'it_core_news_lg'
  - name: SpacyTokenizer
      intent_tokenization_flag: True
  - name: SpacyFeaturizer
  - name: RegexFeaturizer
  - name: LexicalSyntacticFeaturizer
  - name: CountVectorsFeaturizer
  - name: CountVectorsFeaturizer
    analyzer: char_wb
    min_ngram: 1
    max_ngram: 4
  - name: DIETClassifier
    epochs: 150
    constrain_similarities: true
  - name: EntitySynonymMapper
  - name: ResponseSelector
    epochs: 150
    constrain_similarities: true
  - name: FallbackClassifier
    threshold: 0.7
    ambiguity_threshold: 0.1
  - name: "DucklingEntityExtractor"
    url: http://localhost:8000
    timezone: "Europe/Berlin"
    locale: it_IT
    dimensions: ["time"]
    timeout : 3
```

The pipeline used for the project is composed by:

- A pre-trained model for Italian language, present in spacyNLP, named ‘it_core_news_lg’.
- From spacy, we use also SpacyTokenizer and SpacyFeaturizer; the first one is used with intent tokenization flag on true that means that intent labels are also tokenized. RegexFeaturizer creates a vectorial representation of the user message using the regular expressions.
- LexicalSyntacticFeaturizer, that creates lexical and syntactic features for a user message to support entity extraction.
- CountVectorsFeaturizer, that creates bag-of-words representation of user messages, intents, and responses; we set analyzer on char_wb so that character n-grams are used as features. Min_ngam and max_ngram are the lower and upper boundaries of the n-grams.
- The DIETClassifier is used to classify intents and extract entities. We set the parameter constraint similarities to true: this applies a sigmoid cross entropy loss over the similarity terms; this helps to keep similarities between input and negative labels to smaller values, in a way to achieve better generalization of the model.

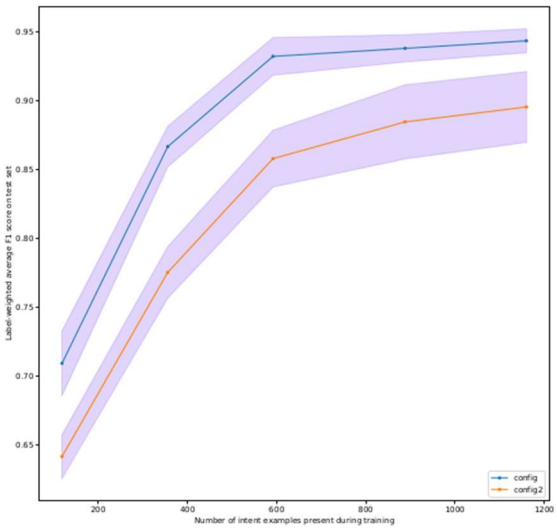
- The EntitySynonymMapper, that maps synonymous entity values to the same value.
- The response selector, that has to predict the bot response from the candidate responses; it builds a response retrieval model, and the prediction of this model is used by the dialogue manager to utter predicted responses. The configuration of the selector is with 150 epochs, so the algorithm will see 150 times the training data. Initially we have set the epochs equal to 100 for DIETClassifier and Response Selector, but the performances of the chatbot wasn't enough good, so we tried to increase this number arriving to 150, that gives us the expected performance.
- The fallback classifier, that is used to handle incoming messages with low NLU confidence; in this way the intent nlu_fallback will be predicted when all the other intent predictions are below the threshold. After speaking tests with the chatbot, we have chosen 0.7 as confidence threshold.
- The duckling entity extractor, that is used to extract time entities, as dates or hours.

Before choosing the pipeline, we compared two of them, thanks to the rasa test command that allows to train and evaluate the model on different pipelines and different amounts of training data.

```
pipeline:
  - name: WhitespaceTokenizer
  - name: LexicalSyntacticFeaturizer
  - name: CountVectorsFeaturizer
  - name: CountVectorsFeaturizer
    analyzer: char_wb
    min_ngram: 1
    max_ngram: 4
  - name: DIETClassifier
    epochs: 150
    constrain_similarities: true
  - name: EntitySynonymMapper
  - name: ResponseSelector
    epochs: 150
    constrain_similarities: true
  - name: FallbackClassifier
    threshold: 0.7
    ambiguity_threshold: 0.1
  - name: "DucklingEntityExtractor"
    url: http://localhost:8000
    timezone: "Europe/Berlin"
    locale: it_IT
    dimensions: ["time"]
    timeout : 3
```

This is the second pipeline that has been compared with the first. We use a WhiteSpaceTokenizer that uses whitespaces as a separator; a LexicalSyntacticFeaturizer that uses syntactic information,and then is used CountVectorsFeaturizer. In the end, we use a DIETClassifier to classify the intent of the phrase and extract the entities at the same time.

The comparison output graph is:



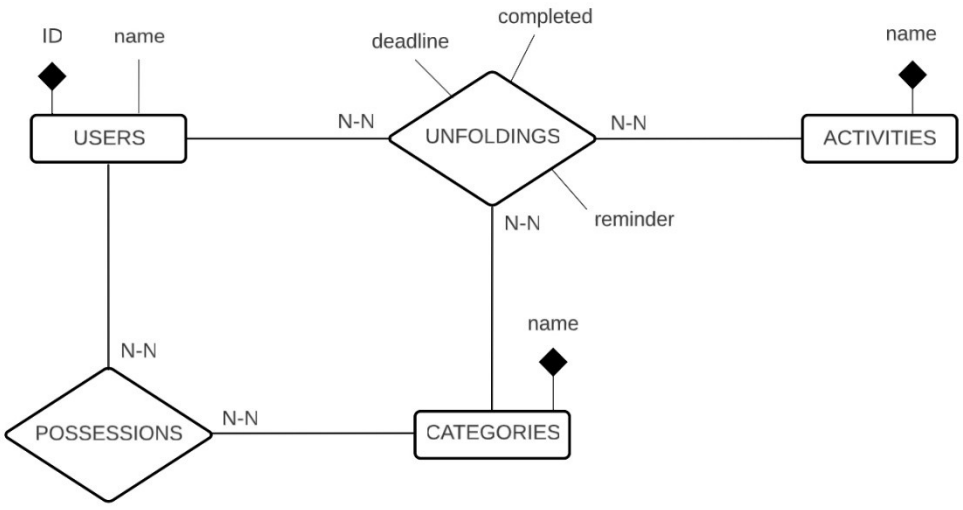
This graph indicates the mean and standard deviations of f1-scores across all runs. We have done 4 runs with different percentages of excluded data from the global train split (0, 25, 50, 70, 90). As we can see from the graph, the blue line (that indicates the first pipeline, the one chosen for our project) has good improvements with the increase of the training data; we can see also that there is no improvement on the last increases of training data, so adding more data will not help to have better performance.

2.2.5 Database Design

The chatbot to be created required the storage of data, that should allow the user to maintain various lists of activities with associated deadlines and possibly receive alerts for each of them. Even if data to be stored were not excessively complex, it was our intent to carry out a database design that complied with all good design standards towards maximum correctness, avoiding redundancies or inconsistencies.
Starting from the description of the problem, the requirements were identified and analysed. Therefore, data and operation specifications were taken into account.
First of all, starting from the requirements, a conceptual design was carried out, following the conceptual rules of the E-R model, identifying design patterns. Starting from the specifications, the conceptual schema was created following a bottom up strategy. So, the initial specifications were divided into increasingly smaller components, until each of them described an elementary fragment of the reality of interest. All the various components were represented by simple conceptual schemes which were then merged to arrived at the final conceptual scheme. Subsequently it has been proceeded with the restructuring of the E-R scheme and with the following translation towards the logical model.

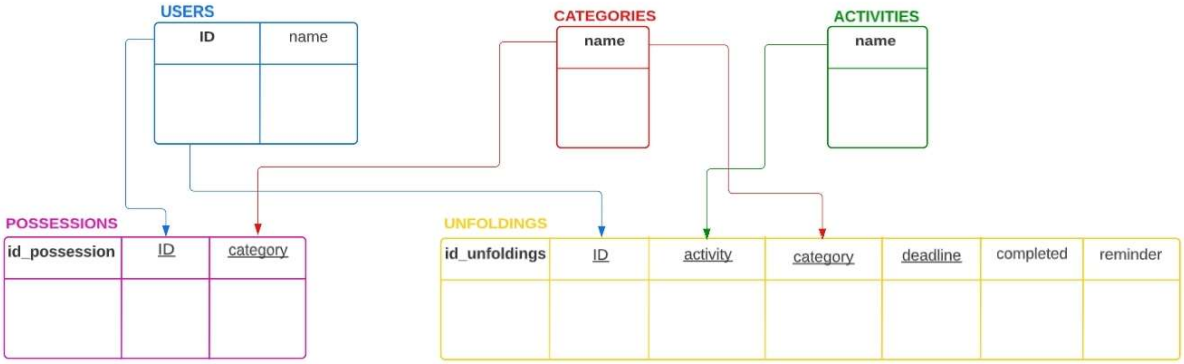
2.2.5.1 Conceptual Design: E-R scheme

The conceptual schema obtained during this phase consists of a many-to-many relationship and a ternary relationship, as shown below. For the different entities all the attributes and primary identifiers have been identified as well as relationships' between them. The ternary relationship exploits the fact that each user can have different activities, each associated with different categories and vice versa, that each category can have different activities, each associated with different users and vice versa and also that , that each activity can have different categories, each associated with different users and vice versa. The relationship many-to-many is necessary as each user could own categories that temporarily have no associated activities. In this way therefore, the presence of several users was also managed, making sure that everyone could manage their own to-do list.
As it is possible to see below, the obtained scheme respects all the requirements that an high quality E-R schema must have, in particular it is correct both syntactically and semantically as it correctly uses the constructs made available by the reference conceptual model; it is completed as all the data of interest are represented and all the necessary operations can be performed; it is readable as all requirements are represented in an understandable way and finally it is minimal as all data specifications are represented only one.



2.2.5.2 Logical Design: relational model

The E-R schema was restructured and translated into the logical model. So, the relationships identified have been translated to the following model.



Here it is possible to see the final logical schema. Each table has its own primary keys, indicated in bold. Through the arrows it is possible to identify the foreign key relationships. The underlined terms identify the unique constraints in the various tables. For “unfoldings” we choose as primary key an identifier called *“id_unfoldings”* to avoid too long keys, this value comes out of the SHA256 of concatenations of ID, activity, category and deadline attributes that are the parameters that univocally identifies an item. This is the reason why on these attributes the unique constraint was not make explicit in the table creation code, but the constraint is however respected. The attributes that participate in the SHA256 have a not null constraint except for “deadline” attribute. Also, the attributes *completed* and *reminder* can assume 0 or 1 value, but null values are not accepted.

2.2.6 Actions

To carry out all the operations requests to the chatbot, an “actions.py” file was created which contains all the necessary custom actions. Specifically, 22 custom actions were created to handle all the intents mentioned above, is important to say that this file has an high level view on the database management in order to heal better each operation. In fact indeed regarding application data, that are stored and managed in a database with sqlite3, we create another python file called “database_connectivity.py”. It contains all the logic to manage data stored in the database, such as creating tables, adding, removing, and modifying rows for each of them.

This separation of concerns allows for better readability and maintainability of the code. Furthermore, the code has been designed to be modular and consistent, which means that it can work seamlessly both with the integration of ROS and in standalone mode through the command line interface (CLI) or "shell". This allows for flexibility in the deployment of the system and facilitates testing and debugging. Particularly since the project is centered on the use of ROS and RASA, the developed code was designed precisely considering the arrival of the message from outside together with the unique ID of the person, but this does not happen when simply using the shell. The main difference between the two modes is the "sender_id" attribute within the tracker. When ROS is used, the “sender_id” is a unique integer, but when connecting from the shell, it is a random value that cannot be converted into an integer. During the “session_start”, the code attempts to convert the “sender_id” into an integer, if successful, it indicates that the connection was started from the ROS system, otherwise from the shell. Further problem arises from this management, when connecting from the shell since the value is random, it cannot be used to identify the user, so we used in this case the SHA of the user's name as the unique ID.

Another challenge faced in the project was related to the management of reminders. Unlike other actions, reminders have an asynchronous and time-based response. The typical conversation flow consists of a user asking a question and receiving a response from the bot. However, with reminders, the response is not immediate and may be triggered at a later time. To handle this, the official RASA guide was followed, which involves setting up a callback server to handle these asynchronous responses. This required modifying the logic implementation for the integration between RASA and ROS. To create a functioning reminder, the reminder object must be returned in the custom action that creates it, and a rule must be created to react to the reminder's expiration. Additionally, a custom event must be created to notify that the reminder has been triggered. All reminders stored in the database are loaded during the session start, in order to notify the user if any of them are due or have expired in the past. Once notified, the reminder for that specific action is updated in the database and will not be notified again the next time the system is turned on.

2.2.7 Chatbot new feature

One of the main differences from the midterm delivery was the change in language for the chatbot from English to Italian. This decision was made due to issues with the Speech to Text system, which had difficulty to correctly recognizing English sentences. This led to the chatbot receiving input that was very different from what was said, which was mainly caused by the difficulties in speaking with an actually English accent however very far from our way of speaking. To avoid these issues and focus more on the implementation of features, it was decided to switch to Italian as the language for the chatbot. The concrete functioning of the reminder has been inserted so no longer as a simple parameter set within the database. Some other intents have been added to expand the functionality of the system and make its use more user-friendly, like the reset of the conversation if the user realizes that the flow of the conversation has not been well understood. We have therefore provided for the mitigation of all critical issues encountered, also in the light of a greater understanding of the use for which the chatbot was intended.

2.3 Integration with ROS

To integrate the RASA server with the re-identification module must be: a ROS node with an interface provides a service that inputs a question string and an identity (label) to the RASA chatbot, and that wait an answer in an asynchronous way. The chatbot's responses are sent to a specific URL, which tells RASA to forward the response to the callback server. This server is always in a ROS node that posts the response to the appropriate to-do list.

A problem relative to this integration was that we needed to pass an "identity" parameter from outside the chatbot and read it inside the same. As said before, ROS client sends together with the message, the ID of the person whose are speaking, so to resolve this problem, in each action, we set ID with value obtained from *"tracker.current_state()["sender_id*"]" (if the systems is not working in a only shall way). This operation, allows not only that the action is performed for that ID, but also that if a certain ID are talking with Pepper and another person ask something to the same, the action relative is performed for the new speaker. This design also allows us to:

- manage people with the same name.

- Only sent messages to the chatbot if the user is correctly recognized; otherwise, initiated a registration procedure at the end of which a message is sent to trigger the creation of a new user in RASA.

- To implement reminders and deadline.

2.3.1 dialogue-server Service

As previously discussed, the integration between the chatbot and ROS was implemented using a ROS service called "dialogue_server". This service is used to send messages to RASA chatbot and takes in input two parameters - "text" and "id" - and outputs an "ACK" from the RASA server as a confirmation.

2.3.2 reminder_server Service

This other service is responsible for publishing all the answers from the chatbot. As a ROS node, it is able to publish the answers on specific topics so that they can be accessed by other ROS nodes as needed. This allows for a seamless integration of the chatbot with the rest of the system.

3 WP4 – Re-Identification

People's voices can change due to factors such as illness or stress. Therefore, we decided to use both the sound wave and the subject's face for re-identification. This allows us to accurately identify the subject using multiple cues, rather than relying on just one method. By using both the sound wave and the subject's face, we can improve the accuracy and reliability of the re-identification process.

Additionally, we have made several modifications to improve the quality of the microphone, which is now decent. However, the camera has lower quality. To compensate for this, we use a resnet50 model trained on VGGFace to recognize faces. This allows us to increase the security of the system by using a more accurate method for identifying people. We carefully evaluate the results of both the face and sound wave classifiers to ensure the accuracy of the person's identity. If the person is not recognized, a registration process is initiated to store the features of their face and voice for future identification.

To facilitate communication between the different recognition packages, some ROS services have been implemented. These services allow the system to request a person's identity synchronously from the various recognition packages. The functionality and usage of these services will be explained in the following sections. These services provide a convenient way for the system to access the information provided by the different recognition packages, enabling the system to utilize the information provided by multiple recognition methods in real-time.

3.1 Face-reidentification model

For the facial reidentification we use a resnet50 model trained on VGGFace to perform face recognition on a video stream. We extract facial features using this model in face recognition package; the choice to use this model is because the network chosen has good performance in terms of accuracy and our system, in terms of reidentification, is strongly imprinted to the facial features; in fact, in the combination rule between facial and voice probabilities, we give more importance to the second for the user recognition. On the other hand, we noted that also in terms of performance, with the refresh rate of the image of Pepper camera, this satisfies our goals; it’s also to take into consideration that the model used will work on a little database, so the different identities it has to recognize will not be that many. Precisely, we give the opportunity to register new users, so our database is not closed to 4 identities, but the expectations are that the number of users can’t be very high.

3.2 Audio-reidentification model

For the audio reidentification model, we use a Keras model defined in “conv_model.py” where we load weights, it is a good choice for us in terms of accuracy-complexity ratio, it can be seen when we use effectively the application. In terms of accuracy this model actually aren’t really enough but through natural learning the database size increase its dimension during the application life. The fact that we load a pretrained network assures us that if the accuracy are not so good are for the few number of samples in the database, and not for the extracted feature from the model.

3.3 Services

3.3.1 voiceLabelService

This service, when called, returns the probabilities related to the recognition of the voice. It uses lock and acquire mechanisms to ensure that the service waits for the analysis of the audio stream to be completed before returning the probabilities. The service returns the results as a Float32MultiArray, which contains a list of probabilities of different individuals. This service allows the system to access the probabilities provided by the speaker recognition package in a synchronous manner, enabling the system to use this information in real-time.

3.3.2 videoLabelService

This service, when called, returns the probabilities related to the recognition of a face. It uses lock and acquire mechanisms to ensure that the service waits for the analysis of the image to be completed before returning the probabilities. The service returns the results as a Float32MultiArray, which contains a list of probabilities of different individuals. This service allows the system to access the probabilities provided by the face recognition package in a synchronous manner, enabling the system to use this information in real-time.

3.3.3 voiceRegistrationService

This service, when called, tells the speaker recognition package to add the previously extracted features of a person's voice to the database. This occurs when the probabilities given from face identification and speaker identification is not high enough. By adding the features to the database, the system can recognize this usure in future interactions. This process is done in speaker identification node, by the registration function.

4 WP5 & WP6 - Tests

We have used a bottom-up strategy to develop tests: we started from unit tests and then developed integration tests.

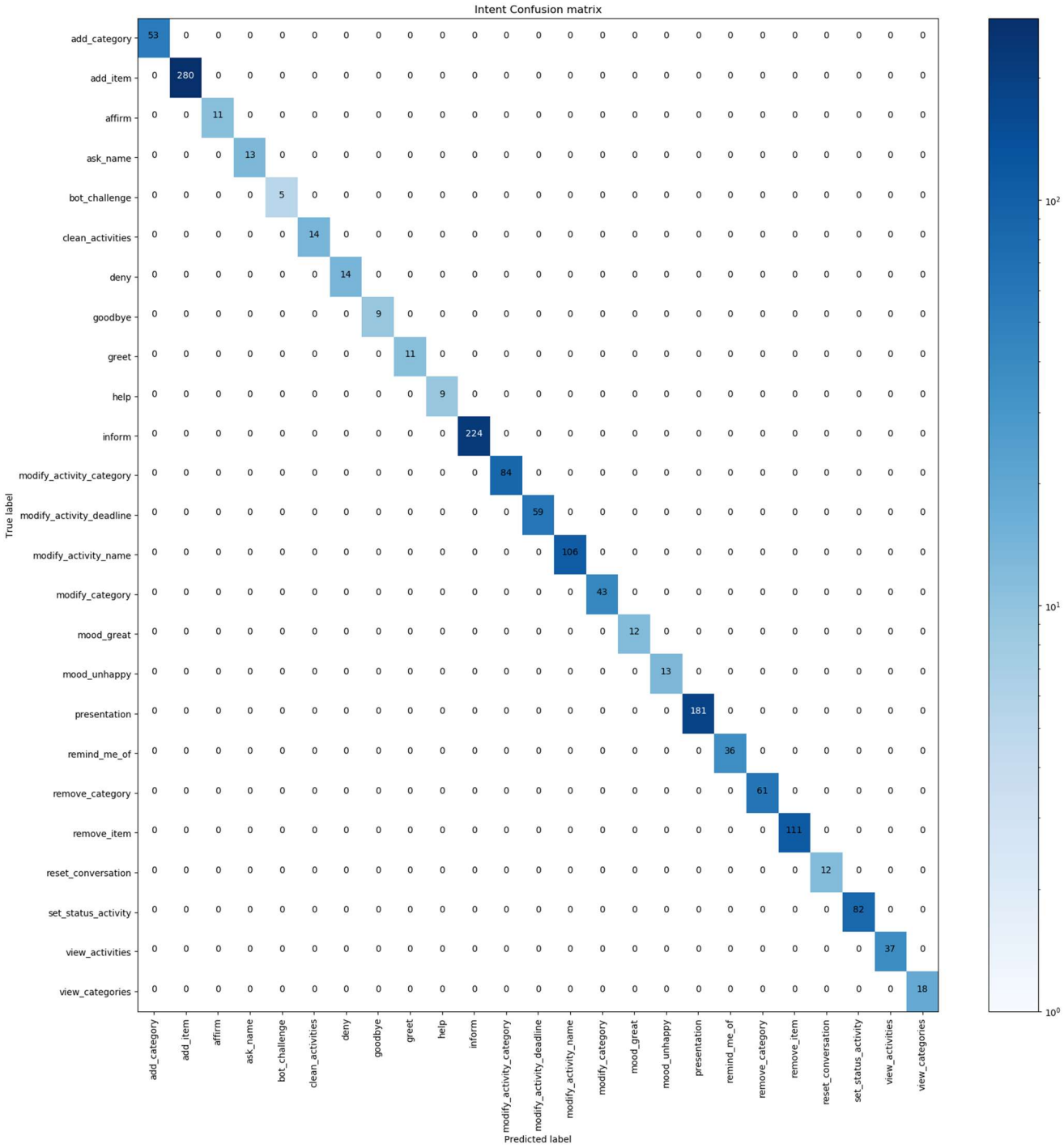
4.1 Unit tests

4.1.1 Rasa Package

To ensure that all functionality of RASA all works better we test all the step throug which a user messages passes, so the individuation of the intent, the extraction of the entities and then the logical flow of the conversation, dictated by the forms and stories.

4.1.1.1 Test Intent

Through the command rasa test we evaluate the performances of our model in recognize intents.



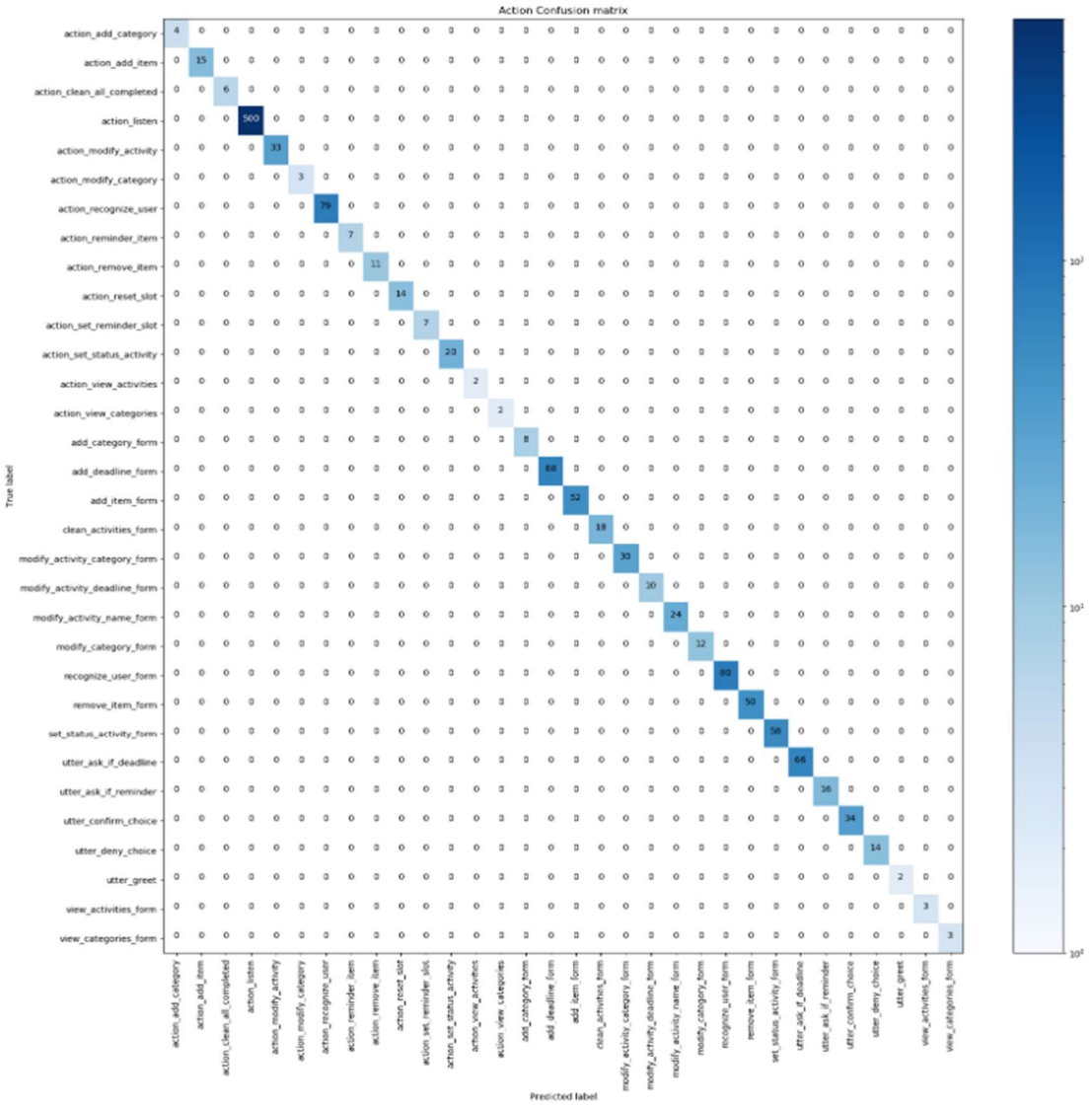
In addition to testing phrases more complex it has been used the option rasa interactive, in order to see step by step the intent predicted by the bot for the different user's messages.

4.1.1.2 Test Entities

To evaluate the robustness in recognize entities we use the command rasa interactive to ensure that for each intent all the entities will be correct recognize and that all the slot are filled in the correct way, also in relations to the forms flow.

4.1.1.3 Test Stories

To test the NLU model we wrote some test stories, that is the best way to know how the bot will act in certain situations. We have written different stories for every case that could happen in every story present in the system. This is the confusion matrix of the stories obtained with those tests:

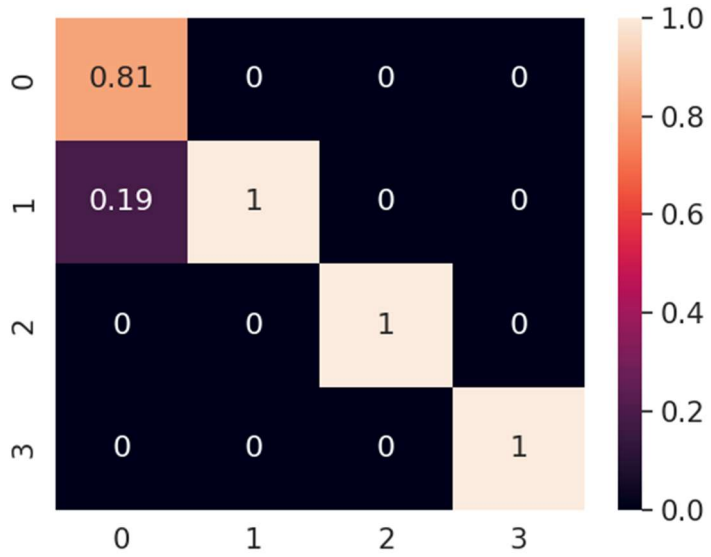


4.1.2 Face Recognition Package

The video stream test was carried out using the "visualization node" which takes the original image and information relative to the facebox from the previous node (face recognition). We also used this node to store the video stream in order to show the images of Pepper's camera offline and how the identified person is then shown on the screen. These images are stored inside the "cameraAcquisition" folder. It is therefore visible how correctly the face is detected and the facebox is tracked.

4.1.3 Face Re-identification tests

A "test" folder has been added to the package, which contains several photos of the four subjects present in the database during the development phase of the product, as well as a python script that can be used to test the accuracy of the reidentification. Specifically, the analysis will compare different photos of the subjects and will then display the confusion matrix. The result of this operation is shown below.



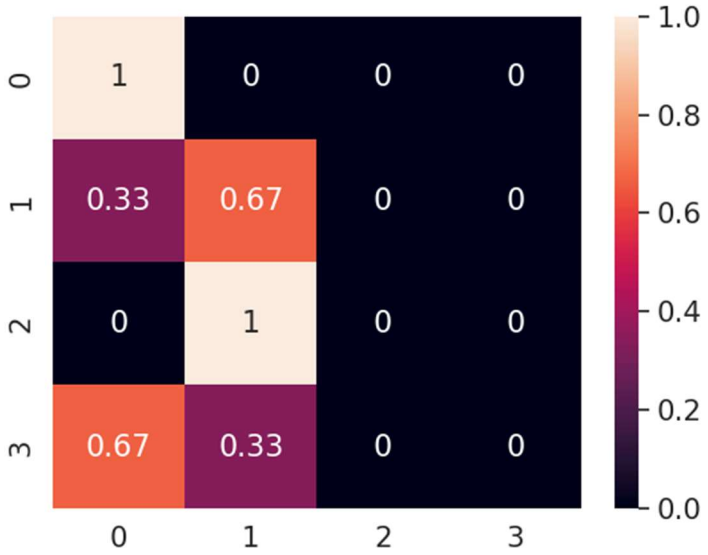
The functionalities, instead of the individual nodes, were tested during product development. The sense of the test folder in the face recognition package was to test the face recognition algorithm with images also different from the ones on which Pepper will often work on; in fact, the test images of the four subjects are with different lighting conditions, poses, angles and various facial expressions. The results of this test were satisfactory, with a high accuracy rate in identifying faces in the images.

4.1.4 Face-reidentification module

To test the video functionality, we utilized a package called "no_Pepper_nodes", which contains several nodes that simulate Pepper's services and publishers. In this specific case, we used a test node that publishes on the same topic used by Pepper, but using the available camera. This allows us to simulate the interaction with Pepper even when it is not physically present. Additionally, we used the "visualization node" to verify the acquired images. The test for the "face recognition" package focuses on the detection and re-identification of a person. Specifically, the flow of images (captured using Pepper's internal camera) will first show an empty shot and then a registered user will arrive. It is possible to see the exact moment when the person's face is detected and then when it is recognized. Furthermore, the detect can also concern several people at the same time. For test reasons, the "visualization node" was left among the launchers used to run the entire project, allowing the user on the PC side to verify that the detect works and that the person is recognized or not. In case of a recognized person, "recognized user" will be shown, otherwise "unrecognized user". To launch the test of the video reidentification we have provided a launcher in face_recognition package named faceRecognition.launch.

4.1.5 Speaker identification tests

The test of this part of the project are made “on field” for most of the time. However, it was decided, as for the face reidentification package, to insert a specific test folder to evaluate the accuracy of the reidentification of the voice. This test has been made on samples coming from the four subject that Pepper has to recognize. Every one of the subjects registered some samples with a different microphone from the one used with Pepper. The script will then calculate the accuracy of the voice recognition and display a confusion matrix to show the results.



As we can see from the confusion matrix, the results are not good, mostly for the identities 2 and 3. This is because the comparison with the features of the database for the speaker identification are very different from the ones in the test set; this was a challenge for this module to recognize user also in different conditions, with different background noises and different microphones. The identities 2 and 3 probably have a poor-quality microphone, respect to the multi-directional microphone on Pepper, or a bigger background noise that makes their voice so different from the one listened on Pepper. Also, the test is made with really few samples to have a better indication of the accuracy. Nevertheless, the combination with face identification and the combination rule used to identify a person allowed us to reach good performance, so the poor accuracy of the audio identification with those samples didn't surprise us and didn't make us change the speaker identification algorithm.

4.1.6 Speaker identification module

To test the speaker identification functionality singularly, in addition to tests made, we use the ros_audio_pkg using the voice detection module, audio speech recognition module (asr) and speaker identification module. We provided a launcher in this package, named voiceRecognition.launch that tests the speaker identification module. Launching this file, the user can speak to test the speech to text functionality and the system will show the probabilities of belonging to the different identities saved in the database, which in this case are the 4 members of the group.

4.1.7 Pepper modules

We have launched the "Pepper.launch" file in the Pepper_nodes package to try these modules. After that, we called the provided services with the command *rosservices call [service_name] [parameters]*.

4.2 Integration test

4.2.1 Ros and Rasa

To make an integration test and try the Ros interface for Rasa, we used an interface from the command line, which takes in input ID, name and the text question and answers with the new ID, the new name, and the answer. This interface uses the service we used in the main application logic and can be tried using the launch file dialogue.xml inside the ros_chatbot package.

4.2.2 Logic without ChatBot

The "test_recognition_wPepper.launch" and "test_recognition_woutPepper.launch" launchers in the "conductor" package are used to test the main logic of the project, including the recognition of a person's voice and face, as well as text recognition. This can be tested with or without the use of Pepper, depending on the launcher used. The flow of the test is the same as for the complete application, but without receiving responses from the chatbot. Throughout the development process, this launcher was heavily used to confirm its functionality. Additionally, this launcher allows for testing of the user registration process,

excluding the chatbot and potentially Pepper.

4.2.3 Logic without Pepper

Exactly as previously described, the launcher “test_application_woutPepper.launch” can be used to test the integration with the chatbot. This will allow you to run all the functionalities without the presence of Pepper, to test the entire product. Specifically, all the recognition logic will come into play, adding however the forwarding of the necessary information to the chatbot (message and ID). Also, in this case this run setting has been widely exploited in the test phase and we can therefore guarantee its functioning.

4.2.4 Final product

To make the final test, the launcher is “finalProduct.launch”. With this launcher, the application works and all the functionalities can be used. The video of the final functioning of the system will be part of the PowerPoint presentation. We represented different operating cases:

1.1 A person ask for her name to Pepper

1.2 A person ask some basic command for managing the To Do List.

1.3 Display of the updated to-do list on the tablet screen

1.4 A new users come into the scene and register his account.

1.5 The user just registered make some operations.

1.6 A person register ask to manage him category

1.7 A person ask for reminder and, then, the reminder will expired.

The video shows the user interacting with the chatbot, adding tasks and managing them, as well as the system correctly recognizing the user's face and voice. This demonstrates the complete functionality of the product and confirms that all components have been successfully integrated and are working properly.

5 Conclusion

In conclusion, the project has successfully implemented all of the necessary and optional features as outlined in the initial specifications. The modifications made during the development process have improved the overall usability and testing of the system and reidentification. The chatbot performs as expected under ideal conditions and has demonstrated accurate text comprehension. The integration of libraries such as "speech_recognition" has simplified the use of Google APIs and effectively manages the audio acquisition process, adjusting to background noise and converting speech to text. The modifications made to the audio package utilizing a multi-array microphone were crucial for proper audio reception. Additionally, the use of ROS allowed for the modularization of the project, making it easy to test and debug individual components separately before integrating them into the final system. This greatly streamlined the development process and made it more efficient. Overall, the utilization of ROS in this project greatly facilitated the implementation of the different functionalities and the communication between them. The use of RASA, along with ROS and other technologies, has made it possible to develop a robust and efficient chatbot that can handle multi-class and multi-user ToDo lists in a user-friendly way. The RASA framework provided a simple and intuitive way to implement the chatbot's logic, even for complex actions such as modification. The use of pre-trained models and libraries like Duckling allowed us to focus on creating the best possible product, without having to spend a lot of time on managing these situations. Additionally, the use of Docker made it easy to integrate Duckling into the project, with minimal setup and configuration required. Overall, the use of these technologies has greatly streamlined the development process, allowing us to create a high-quality chatbot. So, finally, the project has met its goals and is ready for deployment.