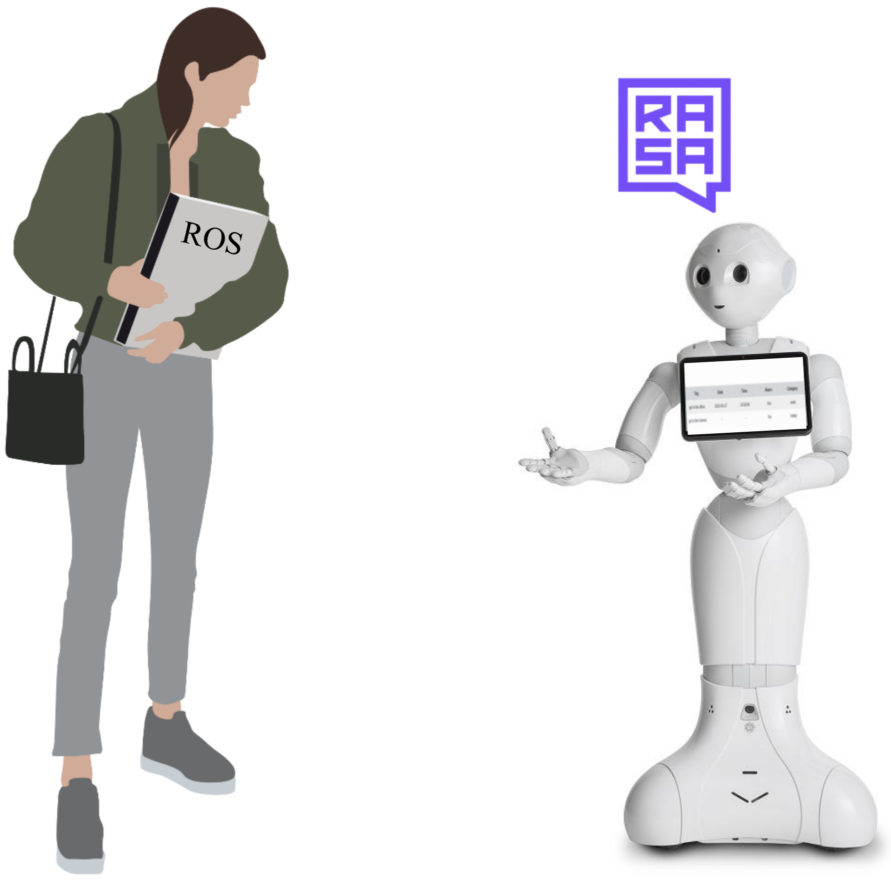


To-Do list with Pepper

Group 4

Avella Antonello	a.avella19@studenti.unisa.it	**0622701703**
Carpentieri Eugenio	e.carpentieri6@studenti.unisa.it	**0622701804**
Costantino Valerio	v.costantino2@studenti.unisa.it	**0622701675**
De Pisapia Claudio	c.depisapia1@studenti.unisa.it	**0622701712**

Course
Cognitive Robotics

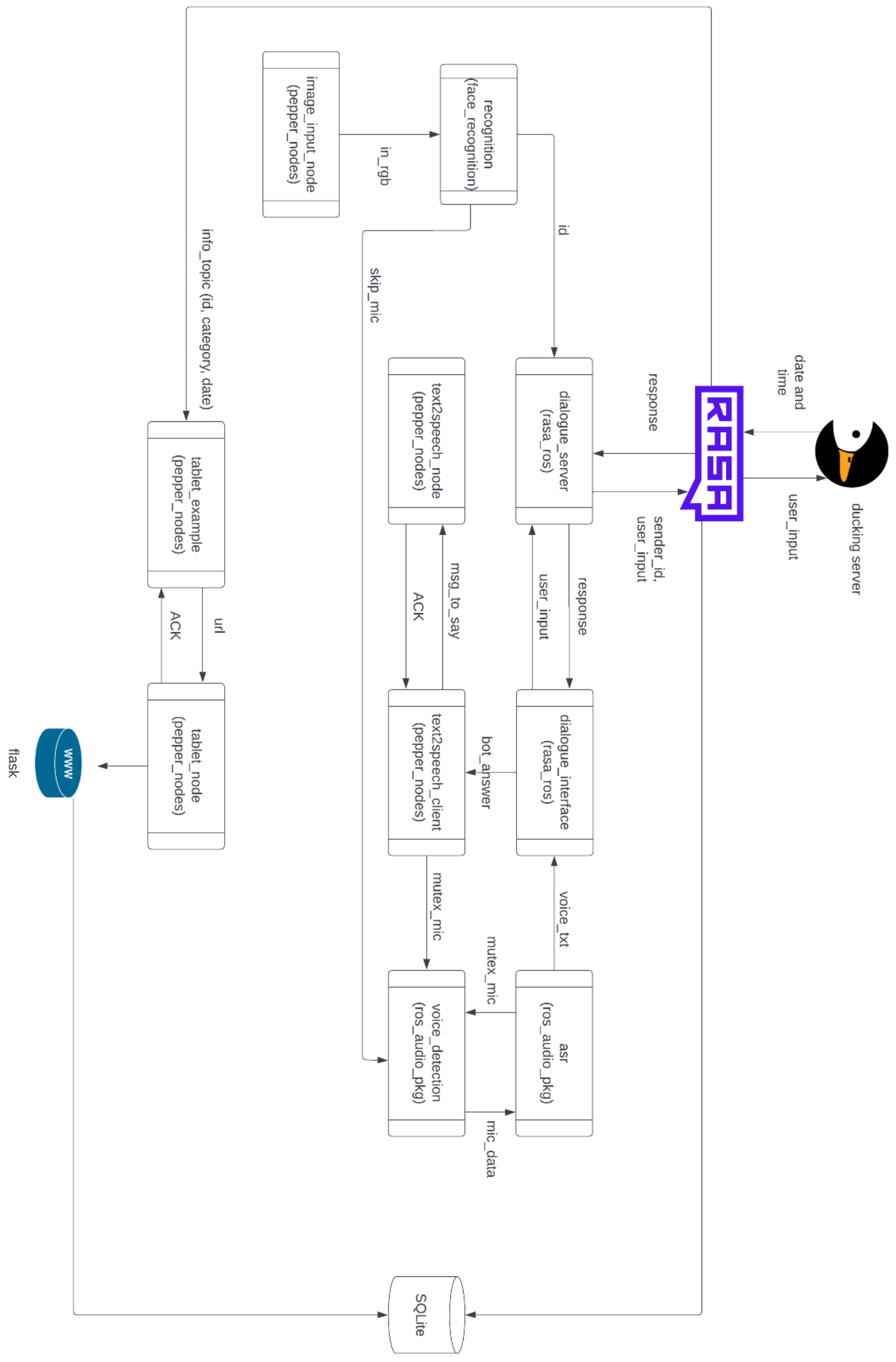
Lecturers
Alessia Saggese

Index

1 WP1 & WP2 - Architecture and Design Choices

1.1 General Architecture

Below is a general outline of the architecture, the individual modules of which will be discussed later in this report:



image_input_node (pepper_nodes)
in_rgb
recognition (face_recognition)
id
skip_mic
info_topic (id, category, date)
dialogue_server (rasa_ros)
response
sender_id, user_input
date and time
user_input
ducking server
text2speech_node (pepper_nodes)
msg_to_say
ACK
tablet_example (pepper_nodes)
url
ACK
tablet_node (pepper_nodes)
www
flask
response
user_input
dialogue_interface (rasa_ros)
bot_answer
text2speech_client (pepper_nodes)
mutex_mic
voice_txt
mutex_mic
asr (ros_audio_pkg)
voice_detection (ros_audio_pkg)
mic_data
SQLite

1.2 Ros Audio Package

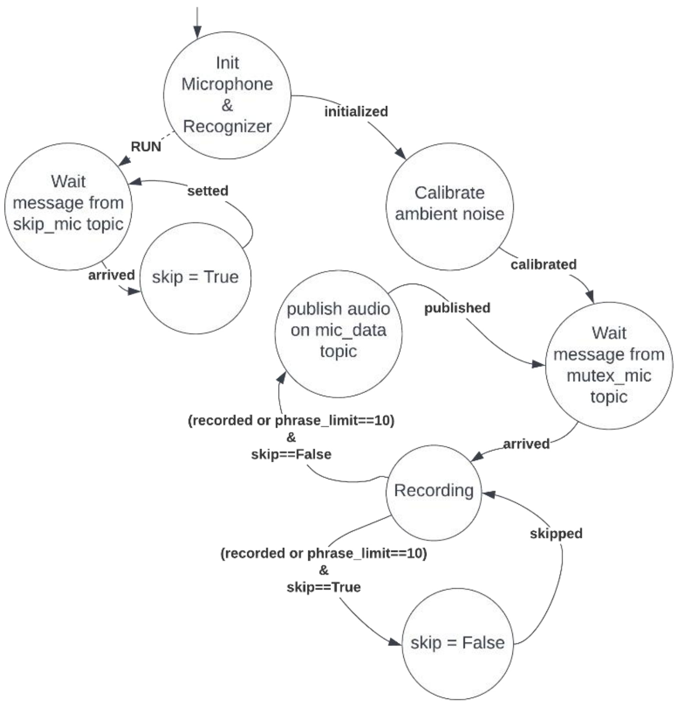
1.2.1 Audio Node

The audio node was implemented in the voice_detection.py file. This module first of all adjusts the energy threshold dynamically using audio from source to account for ambient noise, then starts the audio detection and, at the end, return the listened audio when acquired.

This module has the following requirements:

- Detect the audio in a controlled environment (with silence), by hypothesis.
- Provide a starting service who return the detected audio.

To avoid the fact that Pepper listens to itself, i.e. listens to its own response provided to the user in front of it, we used a publisher-subscriber mechanism to indicate to him whether or not to listen to what comes from the surroundings.



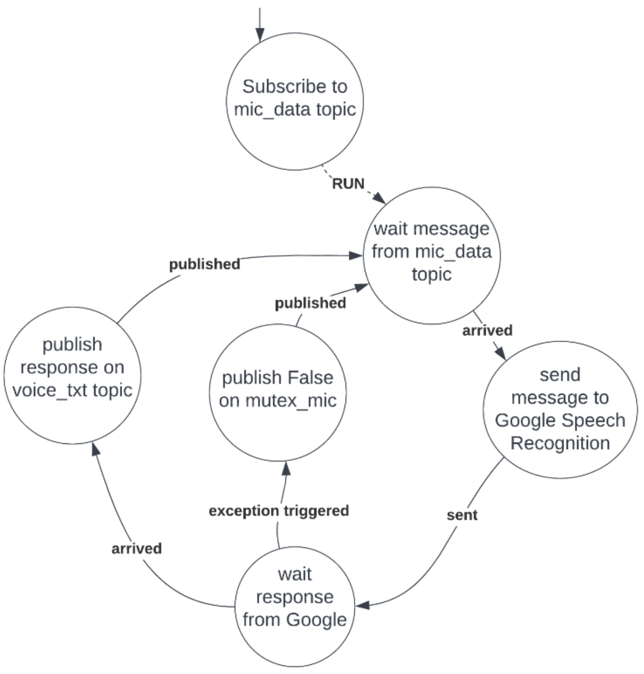
Init Microphone & Recognizer
initialized
Calibrate ambient noise
calibrated
Wait message from mutex_mic topic
arrived
Recording
(recorded or phrase_limit==10) & skip==False
publish audio on mic_data topic
published
(recorded or phrase_limit==10) & skip==True
skip = False
skipped
RUN
Wait message from skip_mic topic
setted
skip = True
arrived

1.2.1 Speech to Text Node

The speech to text node was implemented in the asr.py file. This module receives as input the raw audio wave from voice_detection.py and outputs the transcript of that audio.

This module was implemented as an interface of the google speech services to simplify the call.

When this module raises an UnknownValueError exception, if the speech is unintelligible, or a RequestError exception, if there are some errors in the connection to the service, we publish a value in the topic 'mutex_mic' to indicate to Pepper that the environment recording can start again.



Subscribe to mic_data topic
RUN
wait message from mic_data topic
arrived
send message to Google Speech Recognition
sent
wait response from Google
arrived
publish response on voice_txt topic
published
exception triggered
publish False on mutex_mic
published

1.3 Rasa Ros Package

This package will be discussed in detail in the WP3 chapter, regarding the features of the developed chatbot and its integration with ROS.

1.4 Pepper Nodes Package

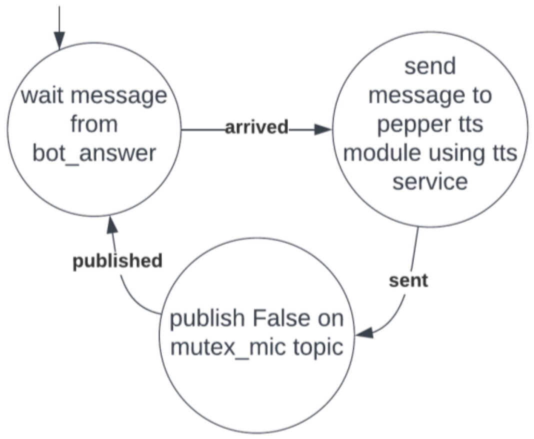
1.4.1 Text to Speech Node

We implemented this functionality in the Text2speech_node.py file, and it makes pepper speak through the ALTextToSpeech.say service.

1.4.2 Text to Speech Client

We implemented this functionality in the Text2speech_client.py file. It waits on the 'bot_answer' topic for the messages pepper has to say and calls the text2speech_node service with the string received as input.

After pepper says the messages, it will publish on the 'mutex_mic' topic that it will be possible to resume recording with the microphone.

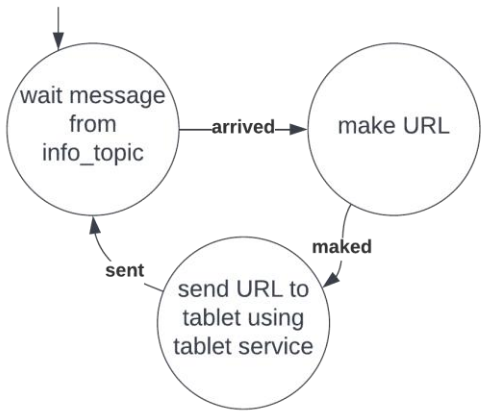

wait message from bot_answer
arrived
send message to pepper tts module using tts service
sent
publish False on mutex_mic topic
published

1.4.3 Tablet Node

We implemented this functionality in the Tablet_node.py file, and it show a website on Pepper's tablet through the ALTabletService service.

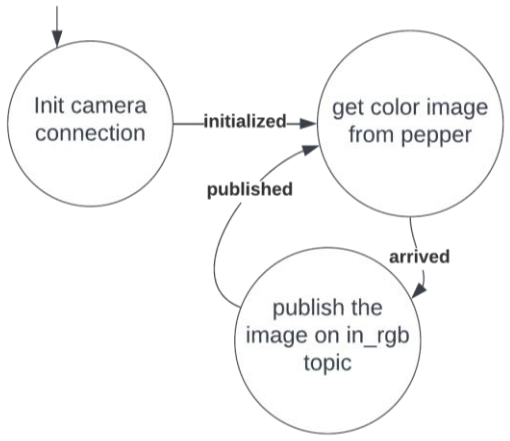
1.4.4 Tablet Client

This client has been implemented in the TabletExample.py file. This module is responsible for sending to the Tablet node the information received about the topic 'info_topic', i.e., the id of the user you are interacting with and the category or date, if any, to customize the to-do visualization via a URL that contains the above information.


wait message from info_topic
arrived
make URL
maked
send URL to tablet using tablet service
sent

1.4.5 Image Input Node

This module configures a ROS node able to read the video stream from the robot's camera. It creates a session to Pepper and initializes the services, then opens a video stream with the Pepper camera that is published on a specific topic. The resolution used is 480p with 20fps.


Init camera connection
initialized
get color image from pepper
arrived
publish the image on in_rgb topic
published

1.4.6 Face Tracker

We implemented this functionality in the Face_Tracker.py file. This module allows pepper to follow the user with its head should it move around the environment using the ALTracker service.

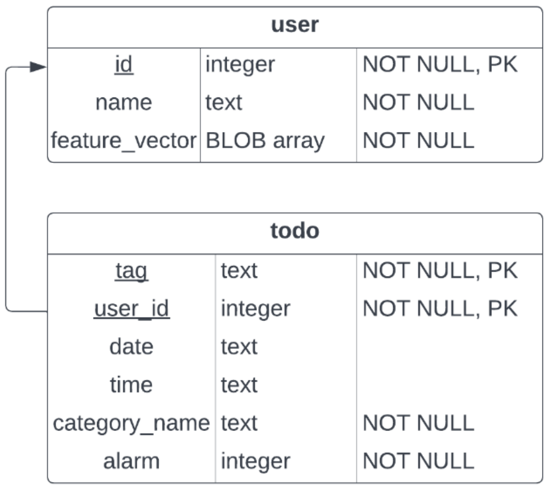
1.5 Face Recognition Package

This package will be discussed in detail in the WP4 chapter, regarding that specific part of our architecture.

1.6 Database

To store the application data, we chose to use a SQLite database consisting of only two tables.

- user is the table that keeps the data relative to a user. This is identified by an AUTOINCREMENT id.
- todo is the table that keeps the todo for each user. These are identified by the user_id and tag pair.



user		
id	integer	NOT NULL, PK
name	text	NOT NULL
feature_vector	BLOB array	NOT NULL

todo		
tag	text	NOT NULL, PK
user_id	integer	NOT NULL, PK
date	text	
time	text	
category_name	text	NOT NULL
alarm	integer	NOT NULL

1.7 Web Server

As mentioned earlier, to show a user's to-do's and alarm, a web page showing this is displayed on the robot's tablet. To accomplish this, Flask was used, which is a Web micro-framework written in Python. The web page made available for our app is view.html and can take the following parameters using the GET method:

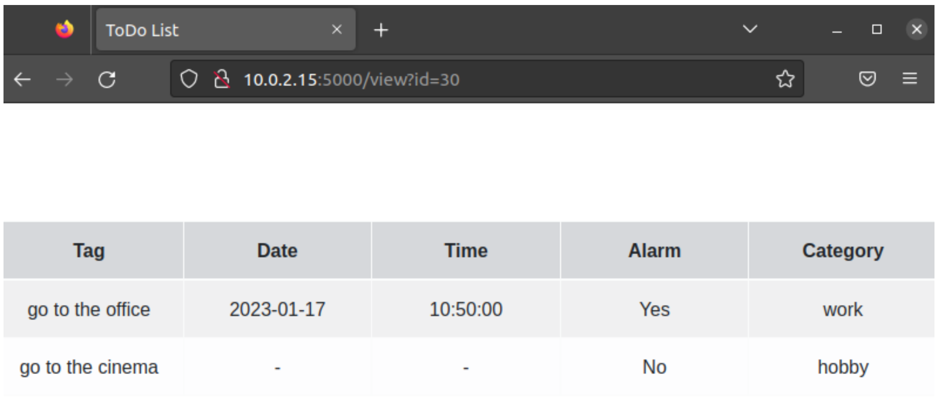
- user id
- date (optional)
- category (optional)

An interface like the one shown in the figure opposite will be displayed showing all the to-does that satisfy the passed parameters.

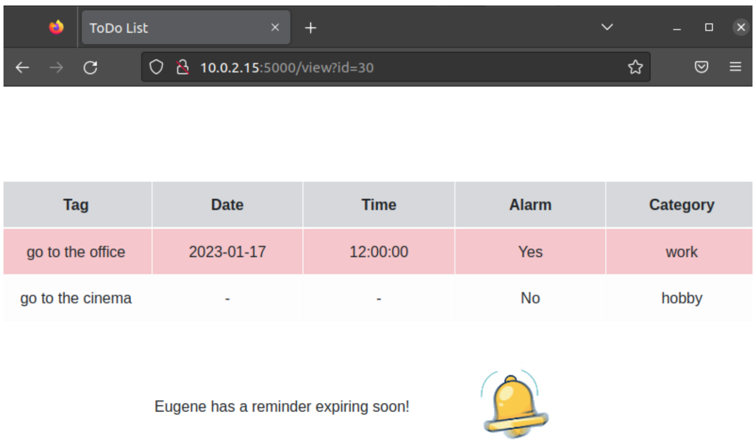
The behaviour of this web app provides the following functionality:

- Expiring to-do will be shown in red while expired to-do will be shown in yellow.
- When an alarmed to-do expires a small bell will be displayed to indicate this.
- When a to-do of a user other than the one currently in front of the robot expires, a bell will be displayed with the name of the user whose to-do is expiring but without showing the details of the to-do for privacy reasons.

Below are pictures illustrating the web page in different situations:

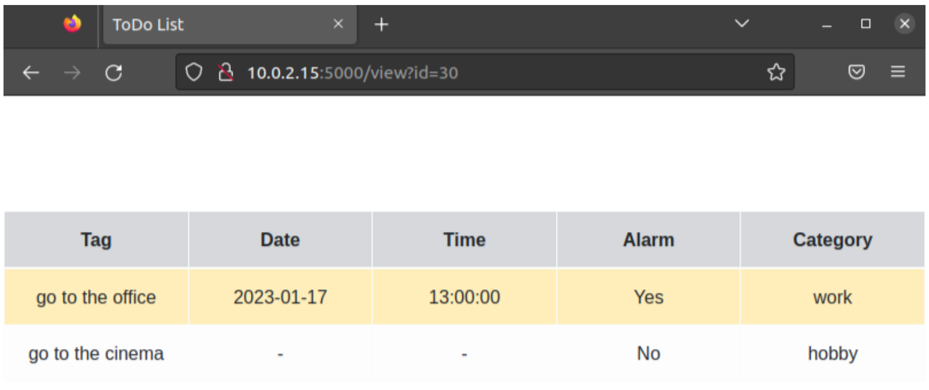


Tag	Date	Time	Alarm	Category
go to the office	2023-01-17	10:50:00	Yes	work
go to the cinema	-	-	No	hobby



Tag	Date	Time	Alarm	Category
go to the office	2023-01-17	12:00:00	Yes	work
go to the cinema	-	-	No	hobby

Eugene has a reminder expiring soon!



Tag	Date	Time	Alarm	Category
go to the office	2023-01-17	13:00:00	Yes	work
go to the cinema	-	-	No	hobby

2 WP3 – RASA

2.1 Chatbot

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.
- **affirm:** handles an affirmative response from the user.
- **deny:** handles a negative response from the user.
- **stop:** handles the situation where the user wants to reset or stop the flow of the conversation.
- **only_information:** handles situations where the user wants to provide values for individual entities separately.
- **identification:** handles user identity when they present themselves to the system.
- **insert_todo:** handles situations where the user wants to enter a new reminder.
- **todo_remove**: handles situations where the user wants to remove a reminder.
- **todo_visualize:** handles situations where the user wants to visualize the reminders.
- **todo_delete_all:** handles situations where the user wants to delete all the reminders.
- **todo_update:** handles situations where the user wants to update a reminder.
- **out_of_scope:** handles situations where it is not clear what the user wants to do.

The identified entites are:

- **alarm:** boolean value representing if the todo should have an alarm or not.
- **category:** represents which category the todo belongs to.
- **name:** represents the name of the user.
- **tag:** represents the todo.
- **time:** represents date and time of the todo.

For each intent, in the *nlu.yml* file, there are examples of possible phrases and questions that a user might ask.

Instead, below are the forms used to write the rules:

Insert form

This form is filled in when a todo list entry is requested. The required slots are:

- Todo_tag: filled from entity tag.
- Todo_date: filled from custom action.
- Todo_time: filled from custom action.
- Todo_category: filled from entity category.
- Todo_alarm: filled from the affirm intent with true value or filled from the deny intent with false value, and filled from entity alarm.

Delete form

This form is filled in when a user wants to delete a todo. The required slots are:

- Todo_tag: filled from entity tag.

Update form

This form is filled in when a user wants to update a todo. The required slots are:

- Todo_tag: filled from entity tag.
- Todo_date: filled from custom action.
- Todo_time: filled from custom action.
- Todo_category: filled from custom action.
- Todo_alarm: filled from custom action.

Rules

In our project we decided to use rules to handle every behaviour of our chatbot, since that we have point-in-time, deterministic, behaviour in all the case scenarios in which the robot would find itself. For example, if you want to make an entry, once you enter the form you have to necessarily fill in all the slots or require the abortion of the aforementioned process to continue the normal control flow. Using only rules, furthermore, can make it easier to test the chatbot, as the set of rules can be easily evaluated to ensure that the chatbot behaves as expected, whereas for stories this is not guaranteed because these are handled by a neural network, which may behave differently depending on the situation. In addition, mixing rules and stories can make it more difficult to understand how the chatbot makes decisions and can make it more difficult to update and modify the chatbot's behaviour.

Below we report the rules written for the previous forms:

1. **To-do <insert, delete, update> activate:** rule is executed at the time the user indicates that they want to start the specific operation. For example, to activate the insert form the user have to say a phrase that activate the insert_todo.
2. **To-do <insert, delete, update> deactivate:** rule is executed at the time all the slots are filled. Then the action corresponding to the operation the user wants to perform is executed and all filled slots are reset.
3. **Stop <insert, delete, update> form:** rule is executed at the time the user indicates that they want to stop the specific operation. To stop a specific operation the stop intent must be detected. Then the action corresponding to the operation the user wants to perform is aborted and all filled slots are reset.
4. **Only information <insert, delete> info:** this rule is performed to make the missing slots that are provided individually fill in so that you do not get out of the loop.

The other rules are:

1. **To-do visualization:** when the todo visualize intent is activated the action is triggered, after which the filled slots are reset. This is useful for allowing a user to visualize their todo.
2. **Identify user:** when identification intent is detected, the action to perform user identification is triggered.
3. **To-do delete all:** when the todo delete all intent is detected, the action to delete all todoes is triggered and then all filled slots are reset.
4. **Fallback rule:** This rule handles the situation where no intent is recognized and ask to the user to repeat the sentence.

Actions

In addition to the simple response actions, the following custom actions were implemented:

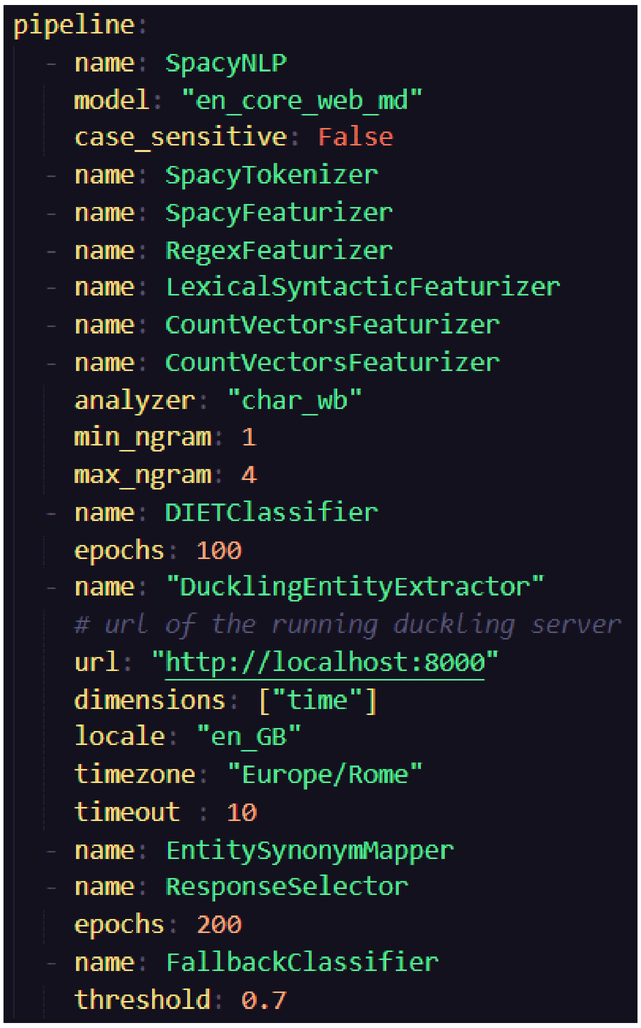
- **action_delete_all_db:** action executed when the user wants to delete all the to-do in their list; this action simply performs a delete query to the database specifying the id of the user it is interacting with.

- **action_delete_db:** action executed when the user wants to delete a todo; this action simply performs a delete query to the database specifying the id of the user it is interacting with and the tag of the reminder he wants to delete.
- **action_insert_db:** action executed when inserting a reminder into the database; this action performs an insert query to the database specifying the tag, the date and the time as well as the presence or absence of the alarm, the id of the user and the category of the new reminder.
- **action_reset_slot:** action executed at the end of every operation, i.e. insert, delete, update or visualize, or if the user asks to abort the current operation to empty the slots of the various forms.
- **action_update_db:** action executed when the user wants to update a reminder; specifically, based on the information provided by the user (the tag of the reminder in question and one among category, date, time and alarm) it performs an update query to the database.
- **action_user_db:** action executed every time the user wants to authenticate himself to the chatbot; this action checks that the name declared by the user is the same as the name within the database related to the id associated with the user it is interacting with.
- **action_visualize_db:** action executed whenever the user wants to see his saved reminders; provides the information needed to make the reminder display on Pepper's tablet.
- **validate_<insert, delete, update>_form:** action required to extract the information provided as input, i.e., tag, date, time, category, and alarm; specifically, it allows parsing of date and time from the timestamp returned by duckling; for the validate_update_form action, since we needed to have the tag information necessarily and one among date, time, category and alarm, our proposed solution is to set the specified information to the specified value, while the remaining unspecified information is set to a dummy value 'null'.

Configuration

In designing the chatbot, we opted to use a pre-trained pipeline. Specifically, we used the spacy library to load pre-trained language models, which are used to represent each word in the user's input by taking advantage of the word embeddings it has. this was done to increase the accuracy of our model, even with very little training data, and because this way training does not start from scratch, which makes it fast and prone to improvements. Unfortunately, complete and accurate word integrations are not available for all languages, so we opted to develop the chatbot in English.

In more detail, the components used within the pipeline are specified below:

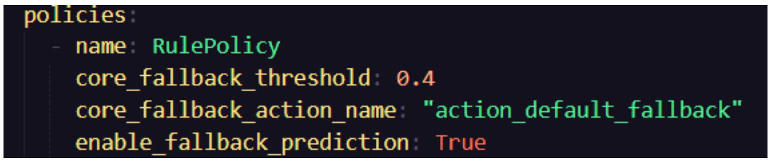


```
pipeline:
  - name: SpacyNLP
    model: "en_core_web_md"
    case_sensitive: False
  - name: SpacyTokenizer
  - name: SpacyFeaturizer
  - name: RegexFeaturizer
  - name: LexicalSyntacticFeaturizer
  - name: CountVectorsFeaturizer
  - name: CountVectorsFeaturizer
    analyzer: "char_wb"
    min_ngram: 1
    max_ngram: 4
  - name: DIETClassifier
    epochs: 100
  - name: "DucklingEntityExtractor"
    # url of the running duckling server
    url: "http://localhost:8000"
    dimensions: ["time"]
    locale: "en_GB"
    timezone: "Europe/Rome"
    timeout : 10
  - name: EntitySynonymMapper
  - name: ResponseSelector
    epochs: 200
  - name: FallbackClassifier
    threshold: 0.7
```

- **RegexFeaturizer:** creates a vector representation of the user's message using regular expressions.
- **LessicalSyntacticFeaturizer:** creates lexical and syntactic features for a user message to support entity extraction.
- **CountVectorsFeaturizer:** uses sklearn's CountVectorizer to help recognize words even in the presence of typos; by setting ngram parameters, instead of getting a unique feature to map a single word, we can use the many parts of which this word is composed as features.
- **DIETClassifier:** DIET classifier of Intent and Entity.
- **EntitySynonymMapper:** if the training data contains defined synonyms, this component will ensure that the detected entity values are mapped to the same value.
- **ResponseSelector:** selectors predict a response from a set of candidate responses.
- **SpacyNLP:** initializes spaCy structures.
- **SpacyTokenizer:** tokenizer that uses spacy.
- **SpacyFeaturizer:** featurizer that uses spacy.
- **FallbackClassifier:** classifies a message with the intent nlu_fallback if the NLU intent classification scores are ambiguous.
- **DucklingEntityExtractor:** library used to extract date and time from text provided as input by the user.

Policies

The policy used in the development of our chatbot is RulePolicy, since, as specified above, we make use of rules only.



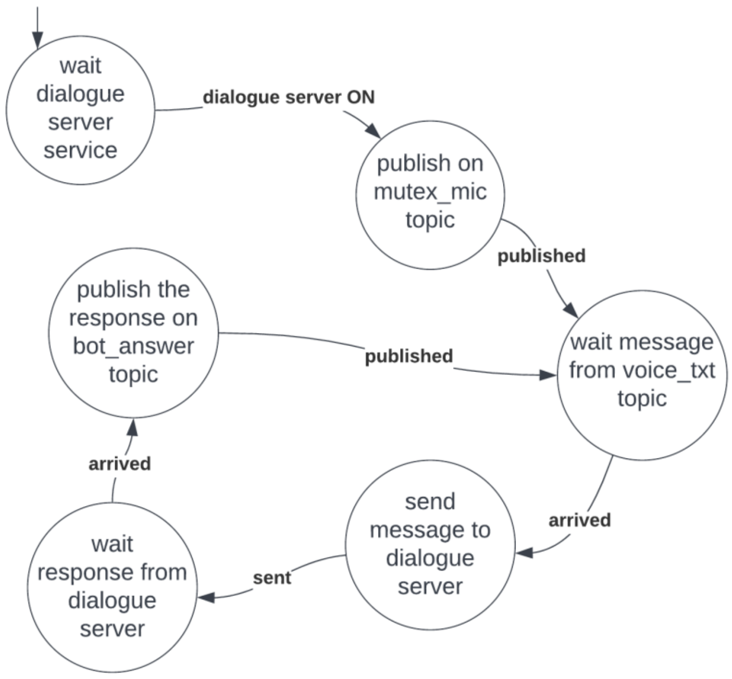
```
policies:
  - name: RulePolicy
    core_fallback_threshold: 0.4
    core_fallback_action_name: "action_default_fallback"
    enable_fallback_prediction: True
```

2.2 Integration with ROS

To integrate the RASA server with the ROS project, we basically needed to exchange three pieces of information, namely the id of the user who is interacting with the chatbot, the input provided by the user and the response selected by the chatbot. To do this, we implemented a Client-Server mechanism through two modules, respectively dialogue_interface.py and dialogue_server.py.

dialogue_interface.py

The client module is responsible for interacting with the speech to text and the text to speech ROS nodes, previously described, and with the server module. More specifically, it receives as input the transcript of the speech input provided by the user and forwards it to the server module. After that, it waits for the chatbot's response, passed by the server module, and publishes on a topic the message that will be spoken by Pepper.

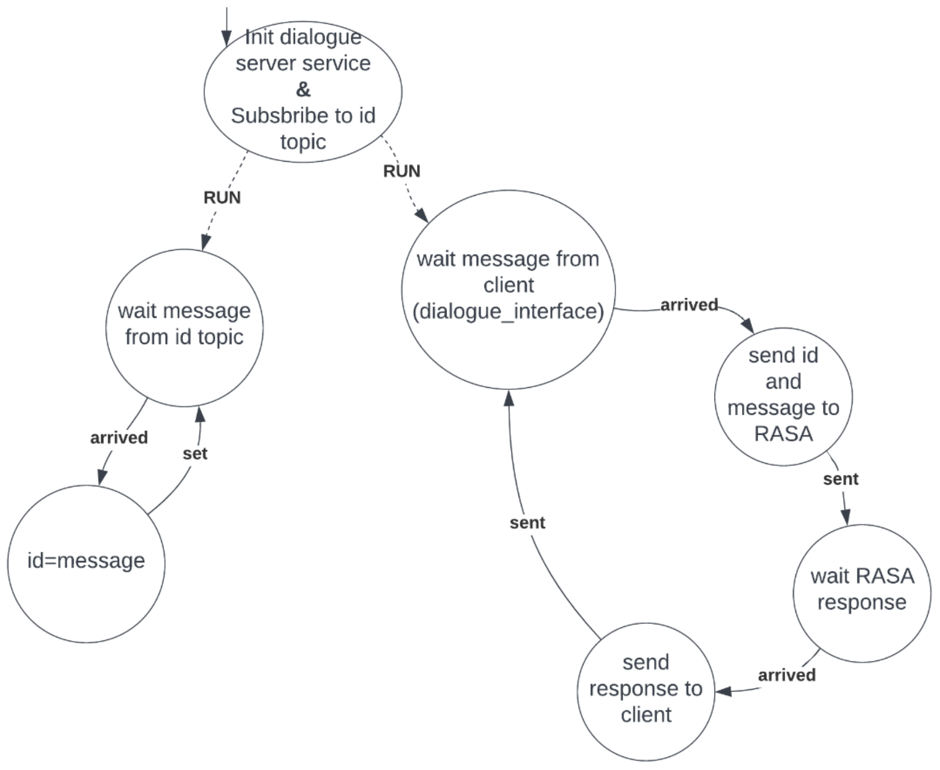

wait dialogue server service
dialogue server ON
publish on mutex_mic topic
published
wait message from voice_txt topic
arrived
send message to dialogue server
sent
wait response from dialogue server
arrived
publish the response on bot_answer topic
published

dialogue_server.py

The server module acts as a conduit between the chatbot and the previously described client module. Essentially, it takes as input:

- the user's identifier, published by the face recognition module, can arrive at any time and will always be updated based on the user who is interacting with Pepper; by hypothesis, let us assume that there is always one and only one person in front of the robot and that this person is framed before the interaction begins.
- the user's input, via the client module.

These two pieces of information are sent to the chatbot, which returns the selected response. This communication takes place via a POST request made to the RASA server.


Init dialogue server service & Subsbribe to id topic
RUN
RUN
wait message from id topic
wait message from client (dialogue_interface)
arrived
set
arrived
id=message
send id and message to RASA
sent
sent
wait RASA response
arrived
send response to client

Rosbridge

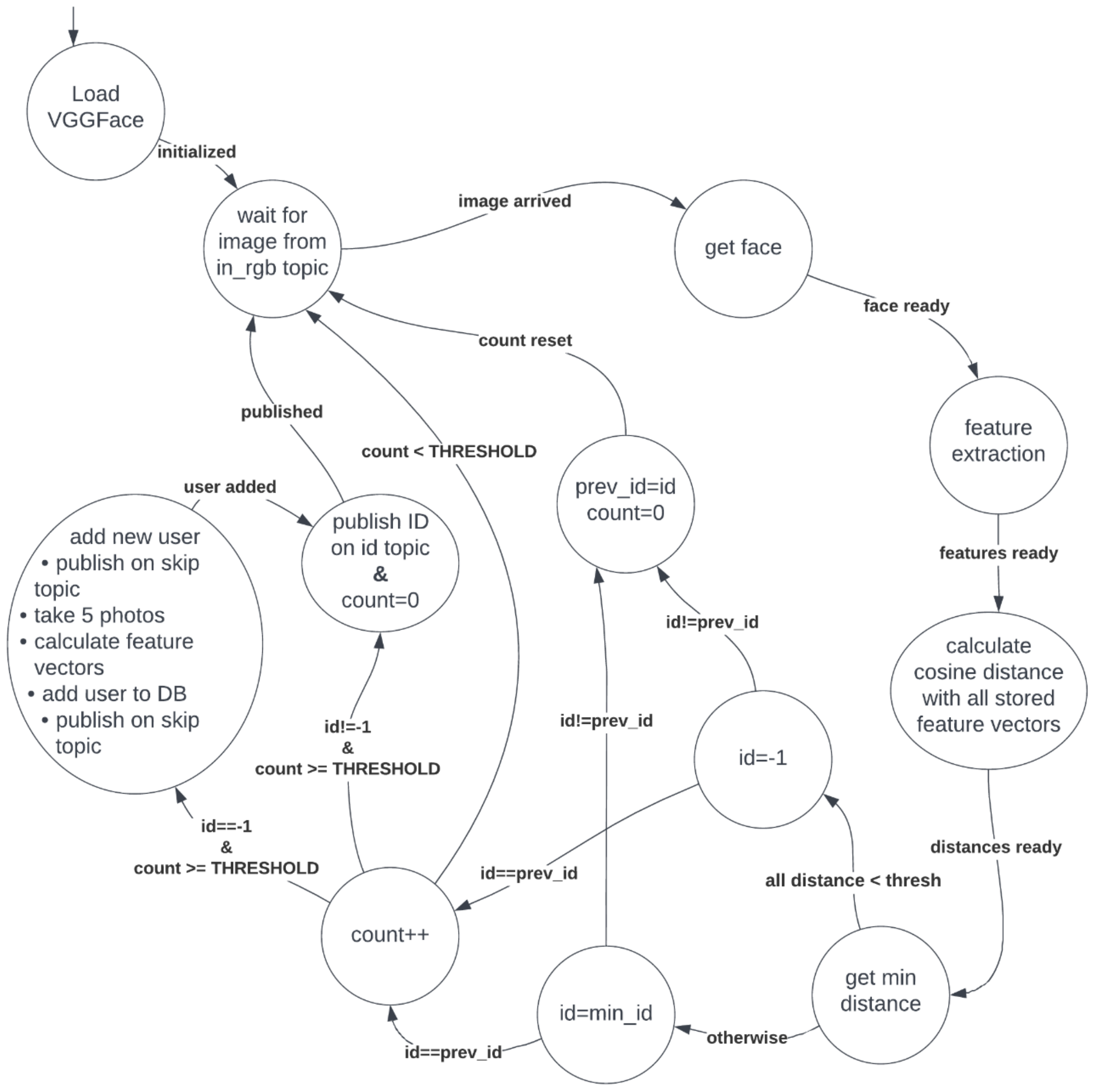
Another detail of the integration between RASA and ROS is the use of Rosbridge to allow RASA's action server to send the necessary information for the to-do display on Pepper's tablet. Rosbridge provides a JSON API to ROS functionality for non-ROS programs. Then, the above information is posted on the 'info_topic' topic. In particular, a new type was defined, called InfoView, consisting of three strings:

- id
- category
- date

3 WP4 – Face Identification

To identify the user with whom Pepper is interacting, we opted for a face identification mechanism that consists of recognizing the user by face image (face recognition task) and verifying that the identity claimed by the user matches the identity recognized by the robot (face verification task). We have chosen to approach this problem as an open set problem, which means that we do not restrict the set of users who can interact with Pepper. Consequently, when a new person shows up in front of Pepper, that person is added to the database after following a procedure of capturing several images to get a good representation of his or her face.

This functionality has been implemented in the recognition.py file. Let's look at its operation in more detail.



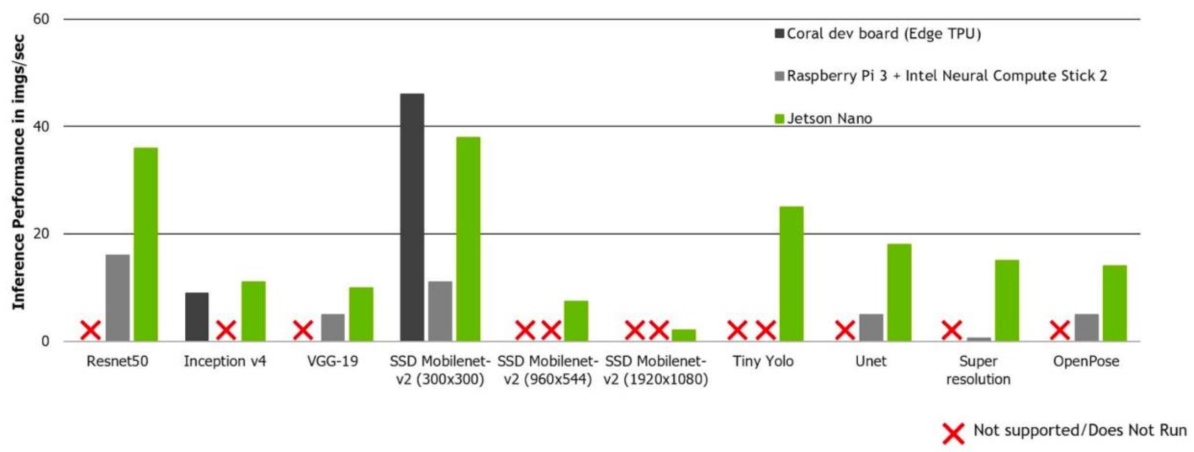
After receiving the image on the 'in_rgb' topic, the bounding boxes of the face within the acquired image are extracted and via a DCNN (in our case VGGFace) we extract the features of the detected face. At this point, the cosine distance between the feature vector just computed and the feature vectors stored within the database is computed and the identity associated with the feature closest to the one just computed is returned (id=min_id). If all calculated distances are less than a certain threshold, the user in front of the robot is classified as an unknown user (id=-1). To prevent occasional pose variations by the user from leading the system to stop recognizing the user and thus abruptly terminating the interaction, we opted for a solution whereby only after THRESHOLD=5 consecutive equal identifications the user's identity value is determined.

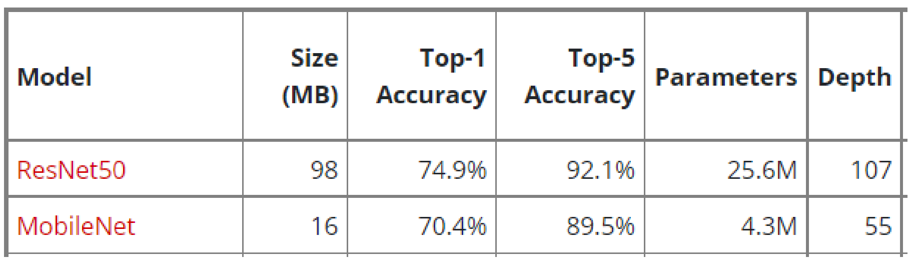
After this step, if the user id is equal to -1 it means that the user is a new user and therefore the procedure of insertion within the database should be followed. The latter consists of acquiring 5 images of this user, calculating the feature vectors of the acquired images and saving them in the database as a blob by associating them with a new id.

At this point, either after the procedure just described or directly in the case where the identifier is other than -1, we proceed to publish the identifier determined on the topic 'id'.

Model selection

Comparing various models both at the level of FPS on various embedded architectures and at the level of accuracy, it appears that the candidates are Mobilenet and Resnet50. We preferred to choose Resnet50 because, as can be seen following from the graphs and the table, it has a slightly higher accuracy than Mobilenet and on an embedded system such as the Jetson Nano it already achieves the minimum value of FPS for our specific application, which is 20 (FPS of pepper camera acquisition).





Model	Size (MB)	Top-1 Accuracy	Top-5 Accuracy	Parameters	Depth
ResNet50	98	74.9%	92.1%	25.6M	107
MobileNet	16	70.4%	89.5%	4.3M	55

4 WP5 & WP6 – Tests

4.1 Unit tests

For testing our system we have used a bottom-up strategy: we started by testing individual components and then testing all components together.

4.1.1 Test for Face Identification Package

For testing this package we have developed 5 test cases that differ in the number of train and test samples afferent to each identity:

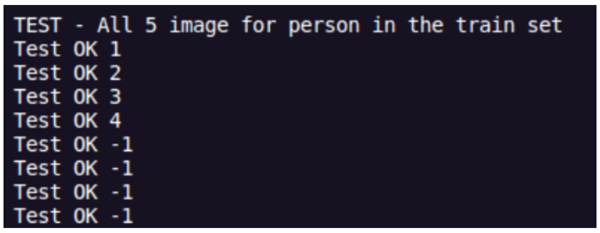
- Test case 1: for each known identity we have only one sample in the training set and only one sample in the test set; in addition, there are 4 samples of the unknown identities in the test set.
- Test case 2: for each known identity we have two samples in the training set and only one sample in the test set; in addition, there are 4 samples of the unknown identities in the test set.
- Test case 3: for each known identity we have three samples in the training set and only one sample in the test set; in addition, there are 4 samples of the unknown identities in the test set.
- Test case 4: for each known identity we have four samples in the training set and only one sample in the test set; in addition, there are 4 samples of the unknown identities in the test set.
- Test case 5: for each known identity we have five samples in the training set and only one sample in the test set; in addition, there are 4 samples of the unknown identities in the test set.

Of these, only test case 1 fails, while the others pass correctly. For this reason, we opted to capture five images for each new user (meaning that the extracted features cannot be associated with any feature vector stored in the database) who interacts with Pepper.

This test can be launched with the following command:

```
cd Group4
export PROJECT_HOME=$(pwd)
python3 ./catkin_ws/src/face_recognition/src/test_recognition.py
```

As a demonstration example, the test output for test case 5 is shown:



```
TEST - All 5 image for person in the train set
Test OK 1
Test OK 2
Test OK 3
Test OK 4
Test OK -1
Test OK -1
Test OK -1
Test OK -1
```

4.1.2 Test for ROS Audio Package

Since this module is a sort of "adapter" for wrapping the google speech recognition model, we have chosen not to develop specific tests for this unit. However, this part of our system was tested individually and gave us good results in transcribing speech, even in the presence of background noise.

4.1.3 Test for Image Input Node

Since we have various types of webcams (computer webcam, Pepper's camera and RealSense D435i), we developed various python modules for communication with these types of cameras, that are image_input_node.py, image_webcam.py and image_realsense.py. For the final project we chose to use Pepper's camera, but in general to test that images are captured correctly and are published correctly on the 'in_rgb' topic the pepper_bringup.launch file was started and by using rostopic echo it was possible to see that it was publishing correctly the images read by the selected camera.

4.1.4 Test for Text to Speech

To test this functionality, the pepper_bringup.launch file was run and through the use of the rostopic pub command strings were posted to the 'bot_answer' topic to make the robot speak the strings passed. For a higher level of detail, in the following we give an example of the rostopic pub command used:

```
rostopic pub /bot_answer std_msgs/String "Hello there"
```

4.1.5 Test for Tablet

To test this functionality, the pepper_bringup.launch file and the webserver.py file were run and through the use of the rostopic pub command information were posted to the 'info_topic' topic to make the robot show the reminders saved in the database. For a higher level of detail, in the following we give an example of the rostopic pub command used:

```
rostopic pub /info_topic pepper_nodes/InfoView "{id: '1', category: 'hobby', date: '2022-12-10'}
```

4.1.6 Test for Web Server

By starting the web server using the following command:

```
export PROJECT_HOME=$(pwd); cd webapp; python3 webserver.py
```

and opening a browser it was possible to access the web page, available at the following URL:

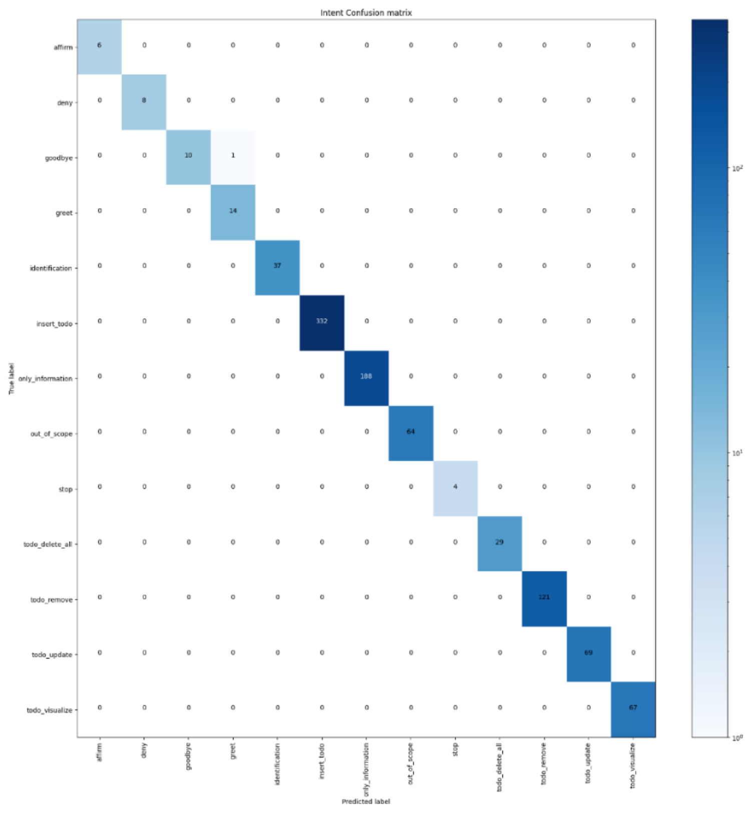
localhost:5000/view

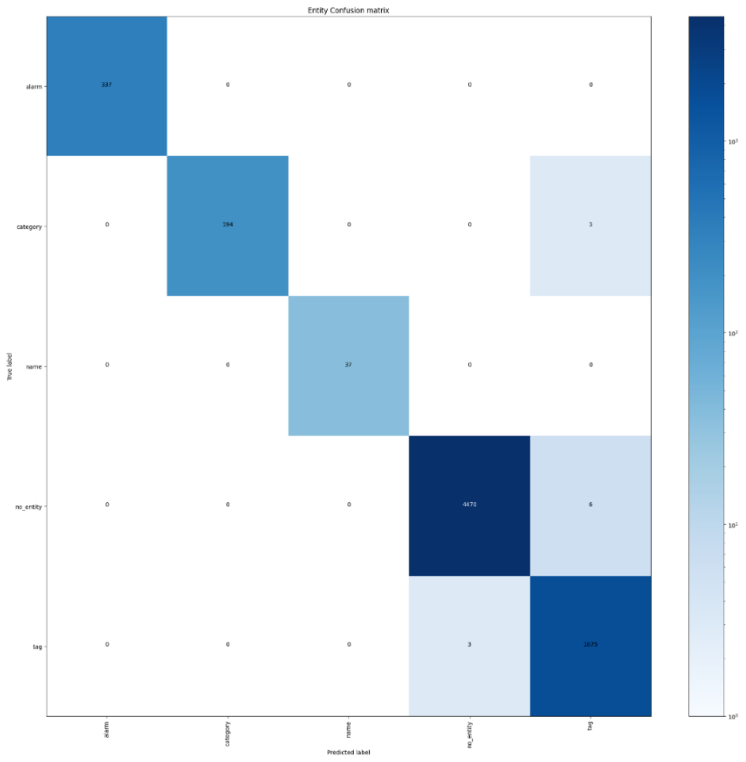
An example of the web page URL with specific parameters might be the following:

localhost:5000/view?id=1&category='hobby'

4.1.7 Test for RASA Chatbot

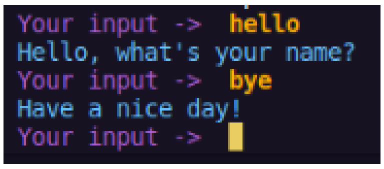
To test the performance of the developed chatbot, we plotted the confusion matrix related to intent and entity which we report below:





From these matrices we can see very good performance on intent recognition and still good performance on entity prediction.

In addition, we used the rasa interactive and rasa shell commands to test the chatbot independently of ROS:



```
Your input ->  hello
Hello, what's your name?
Your input ->  bye
Have a nice day!
Your input ->
```

4.2 Integration Test

To test the system in its entirety along with the robot, the following command must be executed:

```
cd Group4; ./start.sh
```

Examples of using our system in some real cases are given in the delivered video. This test aims to try the face identification, the face tracking, the audio part, the video acquisition part, the tablet visualization part and the behaviour of the chatbot live to test their interaction and not their working because these results are already provided.