UNIVERSITY OF SALERNO

DEPARTMENT OF INFORMATION ENGINEERING, ELECTRICAL ENGINEERING AND APPLIED MATHEMATICS

PROJECT REPORT

Cognitive Robotics

FINALE PROJECT 2025

Group 06

Students	Id number
Alberti Andrea	0622702370
Attianese Carmine	0622702355
Capaldo Vincenzo	0622702347
Esposito Paolo	0622702292

Prof.ssa Saggese Alessia

Academic Year 2024 – 2025

CONTENTS

ABSTRACT

This project presents the development of a ROS-based architecture for reactive communication between the humanoid robot Pepper, acting as a robotic guardian in a shopping mall, and a user. Pepper is designed to answer user queries regarding the shopping mall, such as the number of people present, identifying individuals with specific attributes (specifically the gender, the presence of a hat and a bag), or tracking their movements. Moreover, Pepper has knowledge about the Artificial Vision Contest, including information such as group rankings, scores and members.

The ROS architecture includes modules for speech recognition, dialogue management, speech synthesis with robot animations and people perception. Additionally, user face emotion recognition is implemented and the RASA framework is adopted for natural language understanding and conversation management.

The testing phase is divided into individual component verification and integration tests, ensuring the system's reliability.

CHAPTER 1

W1 & W2: ROS BASED ARCHITECTURE

This chapter describes the ROS-based architecture designed to enable reactive communication between the robot and users. It presents the main software modules of the system, including speech recognition, dialogue management, speech synthesis with robot animations and people perception.

1.1 ROS based architecture design

The system architecture has been designed to enable reactive communication between the Pepper robot and users, using the ROS (Robot Operating System) framework. This system is based on several software modules that interact with each other to perform specific tasks such as speech recognition, dialogue management, speech synthesis with robot animation and people perception. Each module is implemented as a ROS node, which communicates with others through topics and services. The design aims to minimize the number of ROS nodes while ensuring system modularity and facilitating extensibility. Figure 1.1 illustrates the overall system architecture.

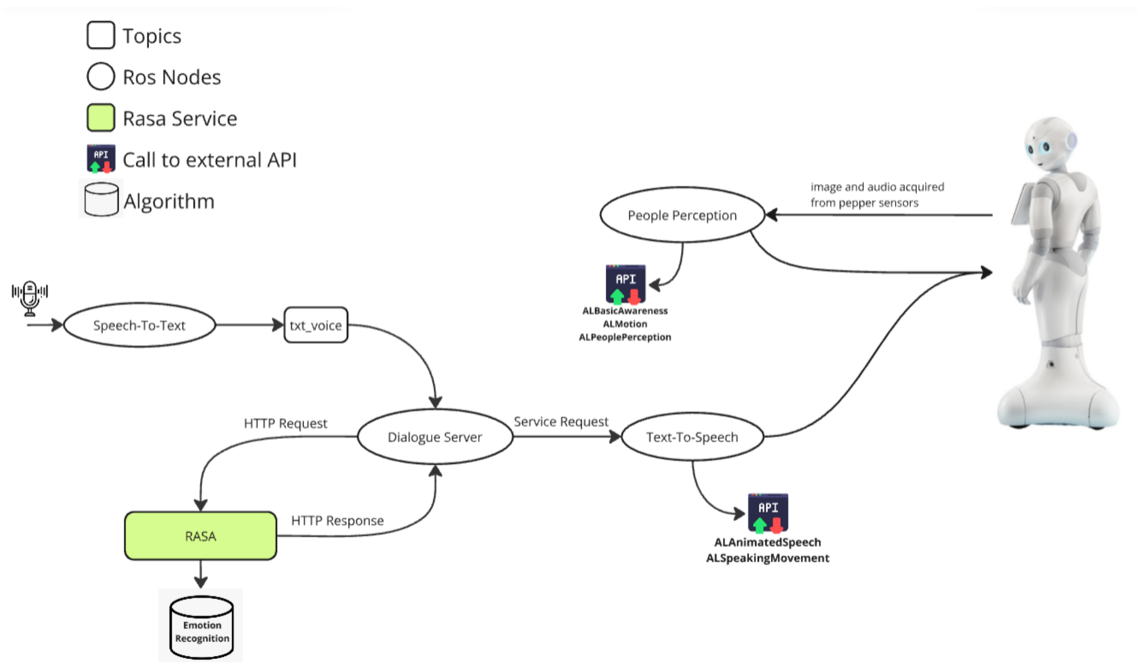


Figure 1.1: System Architecture Overview.

The ROS nodes of the architecture are structured as follows:

- **Speech-to-Text**: module that captures audio from the environment and converts it into text.
- **Dialogue Server**: management ROS node, which communicates with the RASA service and acts as an intermediary between the input modules and the output modules.
- **Text-to-Speech**: module that converts the generated text response into a spoken response for Pepper and play the relative gestures animation to communicate with the user.
- **People Perception**: module that uses Peppers sensors, such as microphones, RGB camera and 3D sensor, to detect and track people in the surrounding environment.

1.2 ROS based architecture implementation

This section outlines the implementation of the ROS-based architecture, detailing how each software module is developed and integrated to achieve seamless interaction between the robot and users. It covers the core components such as the Speech-to-Text, Dialogue Server, Text-to-Speech, and People Perception modules, explaining their functions, intercommunication, and how they contribute to the overall system's operation.

1.2.1 Speech-to-Text

The **Speech-to-Text** module, implemented in the *voice_detection.py* file, is the first step in the system's voice interaction process. This module uses a ROS node to capture the user's audio from the microphone and convert it into text using *Google Speech-to-Text*, a speech recognition service. The default language is English. The resulting text is then published to the *txt_voice* topic by the Speech-to-Text module, which acts as the publisher, while the Dialogue Server node subscribes to this topic to access the text for the analysis. This allows the dialogue system to process the user's input and generate appropriate responses. Voice recognition is executed in the background, allowing the node to process audio continuously without blocking the main process. This is particularly useful to ensure a responsive system. Once started, the node remains in listening mode until explicitly terminated.
Below is the code of the *voice_detection.py* file:

```python
#!/usr/bin/python3
import rospy
import numpy as np
import time
import speech_recognition as sr
from speech_recognition import AudioData
from std_msgs.msg import String

rospy.init_node('voice_detection_node', anonymous=False)
pub = rospy.Publisher('voice_txt', String, queue_size=10)

# this is called from the background thread
def callback(recognizer, audio):
    data = np.frombuffer(audio.get_raw_data(), dtype=np.int16)
    audio = np.array(data,dtype=np.int16)
    audio_data = AudioData(audio.tobytes(), 16000, 2)
    try:
        spoken_text= r.recognize_google(audio_data, language='en-US')
        print("Google Speech Recognition pensa tu abbia detto: " + spoken_text)
        pub.publish(spoken_text) # Publish audio only if it contains words
    except sr.UnknownValueError:
        print("Google Speech Recognition non riesce a capire da questo file audio")
    except sr.RequestError as e:
        print("Could not request results from Google Speech Recognition service;
    {0}".format(e))

# Initialize a Recognizer
r = sr.Recognizer()
r.dynamic_energy_threshold = False
```

```
r.energy_threshold = 150 # Modify here to set threshold. Reference: https://github.com/Uberi/speech_recognition/blob/1b737c5ceb3da6ad59ac573c1c3afe9da45c23bc/speech_recognition/__init__.py#L332
m = sr.Microphone(device_index=None, sample_rate=16000, chunk_size=1024)

# Calibration within the environment
# we only need to calibrate once, before we start listening
#print("Calibrating...")
#with m as source:
#    r.adjust_for_ambient_noise(source,duration=3)
#print("Calibration finished")

# start listening in the background
# `stop_listening` is now a function that, when called, stops background listening
print("Recording...")
stop_listening = r.listen_in_background(m, callback)

rospy.spin()
```

1.2.2 Dialogue Server

The **Dialogue Server** node, implemented in the *dialogue_server.py* file, is responsible for managing the communication flow between the user and the robot. In particular, it receives the input from the user (from the Speech-to-Text module), sends it to RASA for processing, receives the response from RASA and sends it (to the Text-to-Speech module):

1. **from the Speech-to-Text module**: the Speech-to-Text module captures the audio and publishes the resulting text on the *voice_txt* topic. The Dialogue Server listens to this topic and whenever it receives new text, the *callback* function is triggered to process the input.

2. **to RASA**: once the Dialogue Server receives the text from the user, it needs to send this input to RASA for natural language processing. In particular, in the *callback* function, the received text is packaged into a JSON object and sent to the RASA server (running locally) via an HTTP POST request.

3. **from RASA**: RASA processes the input text and returns to the Dialogue Server a response that contains the bot's answer in a text format.

4. **to the Text-to-Speech module**: the Dialogue Server triggers the Text-to-Speech service to convert the response text into speech and start contextual animations. This is handled by the *Handler* class, which acts as a ROS service client to call the */tts* service.

Below is the code of the *dialogue_server.py* file:

```
#!/usr/bin/env python3
from rasa_ros.srv import Dialogue, DialogueResponse
from pepper_nodes.srv import Text2Speech, Text2SpeechRequest, Text2SpeechResponse
from std_msgs.msg import String
import requests
import rospy
import json

class Handler:
    '''
    The constructor creates the service proxy object, which is able to make the robot speak
    '''
    def __init__(self):
        # Initializes a ROS service proxy to interact with the Text-to-Speech service
        self.tts = rospy.ServiceProxy("/tts", Text2Speech)

    '''
    This method calls the Text-to-Speech service and sends it the desired text to be played.
    '''
    def call(self, text: str):
        msg = Text2SpeechRequest() # Create a Text2SpeechRequest message
        msg.speech = text  # Set the speech message
```

```python
        resp = self.tts(text) # Call the service with the provided text

def callback(data):
    '''
    This callback function is called when a message is received on the "voice_txt" topic.
    It sends the input text to Rasa, receives the response and triggers robot animation and speech.
    '''
    # Extract the received text from the message
    input_text = data.data

    # Send the request to the Rasa webhook
    get_answer_url = 'http://localhost:5002/webhooks/rest/webhook'  # Rasa webhook URL
    message = {
        "sender": "bot",  # The sender of the message
        "message": input_text  # The message text to be sent to Rasa
    }
    r = requests.post(get_answer_url, json=message)

    # Create an object for the response
    response = DialogueResponse()
    response.answer = ""
    for i in r.json():
        response.answer += i['text'] + ' ' if 'text' in i else ''

    print(response.answer)

    # Create a Handler object to convert the answer text to speech
    handler = Handler()
    handler.call(response.answer)  # Send the answer text to the Text-to-Speech service

def main():
    # Initialize the ROS node
    rospy.init_node('dialogue_server', anonymous=True)

    # Subscribe to the "voice_txt" topic and call the callback function when data is received
    rospy.Subscriber("voice_txt", String, callback)

    rospy.loginfo("Ready to answer")

    # Keep the node running
    rospy.spin()

if __name__ == '__main__':
    try:
        main()
    except rospy.ROSInterruptException:
        pass
```

1.2.3 Text-to-Speech

The **Text-to-Speech** module, implemented in the *text2speech_node.py* file, allows Pepper to communicate with users using synthesized speech. The Dialogue Server node generates text responses using RASA based on the user's input. Once the response is ready, the Dialogue Server sends the text to the Text-to-Speech module by calling the */tts* service. Upon receiving the text through the service call, the Text-to-Speech node uses Pepper's built-in *ALAnimatedSpeech* API to convert the text into synthesized speech, which is played through the robot's speakers, allowing Pepper to communicate verbally with the user. In addition to the text, the *ALAnimatedSpeech* API plays an animation matching the spoken sentence. To achieve this, the configuration is set up on *contextual*, allowing specific animations to be executed based on the spoken words, associating the animation with the context. To further refine the animations, the *ALSpeakingMovement* API is used, enabling the mapping of specific animation tags to specific key words. For example, if the text to be spoken contains the word "hi" then an animation corresponding to the *hello* tag is activated. This allows for further customization of animations based on the spoken words.
Below is the code of the *text2speech_node.py* file:

```python
#!/usr/bin/python3
from utils import Session
from pepper_nodes.srv import Text2Speech
from optparse import OptionParser
import rospy
import random

class Text2SpeechNode:
    '''
    This class defines a ROS node responsible for speaking and animation of the robot.
    '''

    def __init__(self, ip, port):
        '''
        The costructor creates a session to Pepper and inizializes the services
        '''
        self.ip = ip      # Pepper ip
        self.port = port # Pepper port
        self.session = Session(ip, port) # Initialize the session

        # Requires the following services:
        # ALAnimatedSpeech needed to move Pepper while speaking.
        # ALSpeakingMovement to personalize Pepper's movements.
        self.tts = self.session.get_service("ALAnimatedSpeech")
        self.speak_move_service = self.session.get_service("ALSpeakingMovement")

        # Set the configuration to "contextual": Pepper catch key words to play the right animation.
        self.configuration = {"bodyLanguageMode":"contextual"}

        # Customization Pepper's movements: mapping animation tags to key words.
        self.textToTag = {"hello": ["hi", "hello", "greetings", "hey", "goodbye"],
                          "affirmative": ["yes", "yeah", "ok", "okay"],
                          "enthusiastic": ["happy"],
                          "reason": ["think", "repeat","rephrase"],
                          "no": ["no", "negative"],
                          "I": ["me", "i", "pepper", "i'm", "am", "myself", "robot", "bot"]
                         }
        self.speak_move_service.addTagsToWords(self.textToTag) # add mapping to tags

    def say(self, msg):
        '''
        Rececives a text message and call the ALAnimatedSpeech service.
        The robot will play the text of the message and the contextual movement.
        '''
        msg = msg.speech.lower()
        self.tts.say(msg, self.configuration)
        return "ACK"

    def start(self):
        '''
        Starts the node and create the tts service.
        '''
        rospy.init_node("text2speech_node")
        rospy.Service('tts', Text2Speech, self.say)
        rospy.spin()

if __name__ == "__main__":
    parser = OptionParser()
    parser.add_option("--ip", dest="ip", default="10.0.1.207")
    parser.add_option("--port", dest="port", default=9559)
    (options, args) = parser.parse_args()

    try:
        ttsnode = Text2SpeechNode(options.ip, int(options.port))
        ttsnode.start()
    except rospy.ROSInterruptException:
        pass
```

1.2.4 People Perception

The **People Perception** module, implemented in the *people_perception_node.py* file, is responsible for detecting and tracking people using the robot's sensors. This module works under the assumption that only one person can interact with the robot. Once the robot detects a person, using its RGB camera and 3D sensor, it moves its head to follow the movements of the engaged person. Furthermore, the robot is able to move its head in the direction of the perceived sound, but the camera has priority over the microphones, meaning that if the user is within Pepper's field of view, the robot will maintain visual contact. Otherwise, it will use the sound's direction of origin to rotate its head accordingly. This module uses the following APIs:

- *ALMotion*: is used to put Pepper into *wakeup* mode, when the node starts, and into *rest* mode, when the node stops.
- *ALBasicAwareness*: is used tracking people capturing external stimulis. The engagement mode is set up on *FullyEngaged*, in such a way that once Pepper engage a person doesn't search for stimuli anymore. The tracking stimuli are set up to detect only people and sounds, in order to reduce the complexity and the computational time needed to process the stimuli.
- *ALPeoplePerception*: is used implicitly from *ALBasicAwareness* to retrieve the informations about the tracked person and explicitly to set the maximum detection range to 3 meters.

Below is the code of the *people_perception_node.py* file:

```python
#!/usr/bin/python3
from utils import Session
from optparse import OptionParser
import rospy

class PeoplePerceptionNode:
    '''
    This class defines a ROS node responsible for perception people using the robot
    's sensors.
    '''

    def __init__(self, ip, port):
        self.ip = ip      # Pepper ip
        self.port = port # Pepper port
        self.session = Session(ip, port) # Initialize the session

        # Requires the following services:
        # ALMotion to set Pepper into wakeup or rest mode.
        # ALBasicAwareness to tracking people capturing external stimulis.
        # ALPeoplePerception is used implicitly by ALBasicAwareness and for set the
 maximum detection range.
        self.motion_proxy = self.session.get_service("ALMotion")
        self.basic_awareness = self.session.get_service("ALBasicAwareness")
        self.people_perception = self.session.get_service("ALPeoplePerception")

        # Set the engagement mode to "FullyEngaged": once Pepper engage a person
    doesn't search for stimuli anymore.
        self.basic_awareness.setEngagementMode("FullyEngaged")

        # Set tracking stimuli to detect people and sounds: it helps to reduce the
    complexity and the computational time needed to process the stimuli.
        self.basic_awareness.setStimulusDetectionEnabled("People", True)
        self.basic_awareness.setStimulusDetectionEnabled("Sound", True)

        # Set the maximum detection range to 3 meters.
        self.people_perception.setMaximumDetectionRange(3.0)

    def start_people_perception(self):
        '''Start the people perception and puts the robot to wakeup.'''
        self.motion_proxy.wakeUp()
        self.basic_awareness.setEnabled(True)

    def stop_people_perception(self):
        '''Stop the people perception and puts the robot to rest.'''
        self.basic_awareness.setEnabled(False)
        self.motion_proxy.rest()

```

```python
if __name__ == "__main__":
    parser = OptionParser()
    parser.add_option("--ip", dest="ip", default="10.0.1.207", help="IP address of the robot")
    parser.add_option("--port", dest="port", default=9559, type="int", help="Port number of the robot")
    (options, args) = parser.parse_args()

    node = PeoplePerceptionNode(options.ip, options.port)

    rospy.init_node("people_perception_node")

    try:
        node.start_people_perception()
        rospy.loginfo("People Perception started.")
        rospy.spin()
    except rospy.ROSInterruptException:
        rospy.loginfo("ROS node interrupted.")
    finally:
        node.stop_people_perception()
        rospy.loginfo("People Perception ended.")
```

CHAPTER 2

WP3: VIDEO ANALYTICS MODULE AND SPEECH-TO-TEXT MODULE

2.1 Video Analytics module

2.1.1 People Perception

The People Perception module is responsible for detecting and tracking people using the robot's sensors. It operates under the assumption that only one person interacts with the robot at a time. Once Pepper detects a person using its RGB camera and 3D sensor, it can move its head to follow the movements of the engaged individual. Additionally, Pepper can orient its head toward the direction of perceived sound, although visual tracking has priority. This module leverages the NaoQi framework to process data from Pepper's audio and video sensors and control its movements.

For sound localization, it uses the *ALSoundLocalization* API, which relies on Pepper's microphones and the Time Differences of Arrival (TDOA) technique. This technique determines the direction of sound by analyzing the time delays with which the sound wave reaches each microphone, achieving an accuracy of 10 degrees.

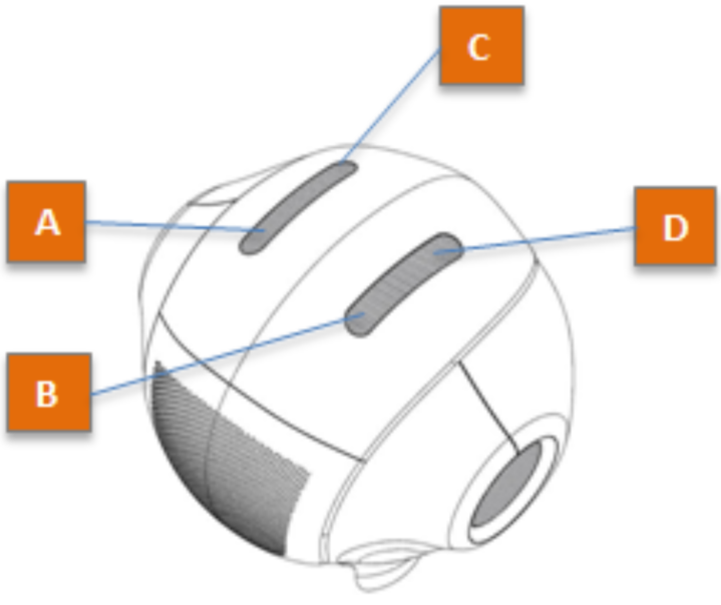


Figure 2.1: Pepper Microhpones Scheme.

For visual perception, Pepper uses the *ALPeoplePerception* API, which tracks individuals around the robot and provides basic information such as their position. This API maintains a list of visible people, providing various information, including their position (in degrees) relative to Pepper's torso. This list is updated over time, allowing people to be tracked across consecutive frames. The *ALPeoplePerception* API gathers visual data from two RGB cameras (one above its eyes and another on its chin) and a 3D sensor located in Pepper's left eye. While the RGB cameras detect people, the 3D sensor measures their distance, limiting interactions to a maximum range of 3 meters.

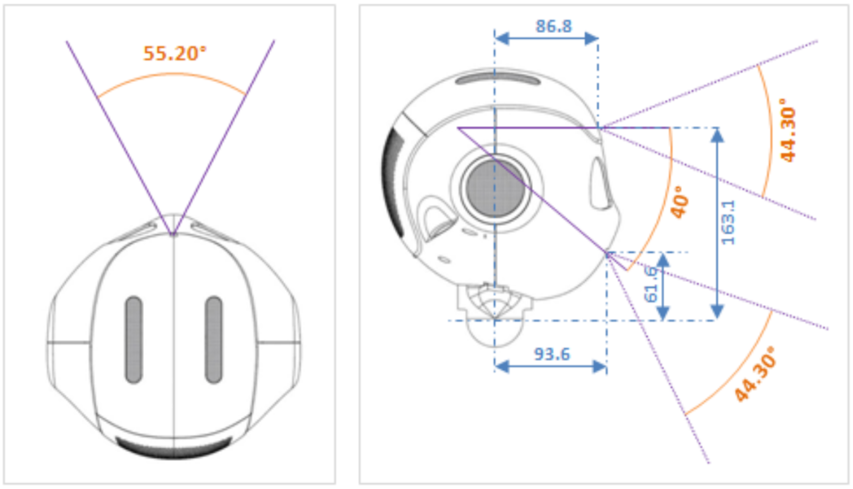


Figure 2.2: Pepper has two RGB cameras, one at the top, above the eyes, and one at the bottom, on the chin.

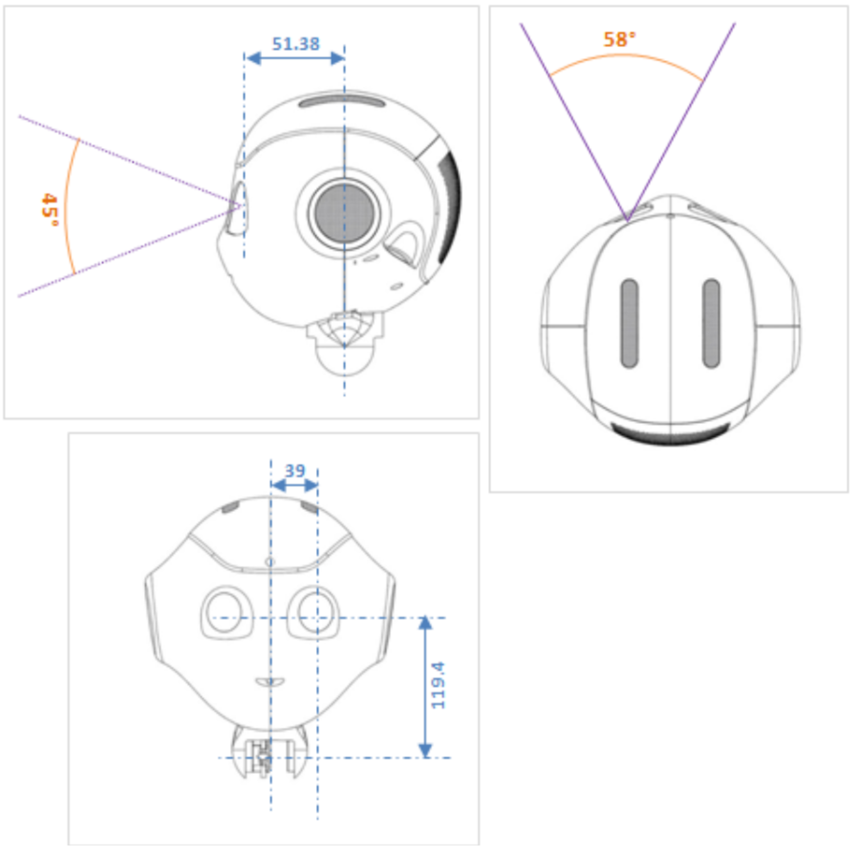


Figure 2.3: The 3D sensor is located in Pepper's left eye and measures the distance between the robot and the target.

The overall behavior is managed by the *ALBasicAwareness* API, which orchestrates responses to stimuli and prioritizes them. In the proposed solution, visual stimuli (*People*) and auditory stimuli (*Sound*) are enabled. The process works as follows:

1. Pepper waits for a stimulus.
2. Upon detecting a stimulus, the robot turns its head in the corresponding direction and processes the event.
3. If no person is detected, it returns to waiting. If a person is identified, Pepper begins tracking them.
4. When the person is no longer detected, the robot resumes waiting for new stimuli.

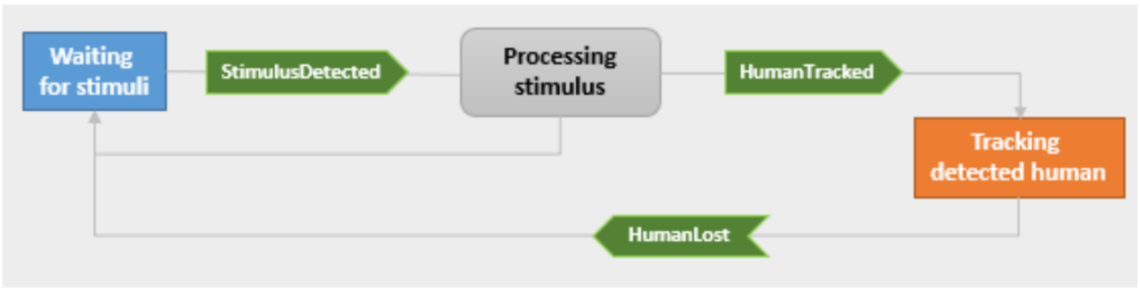


Figure 2.4: State diagram of the process.

More specifically, there can be multiple types of stimuli, each with an assigned priority. Figure 2.5 illustrates how these priorities are managed.

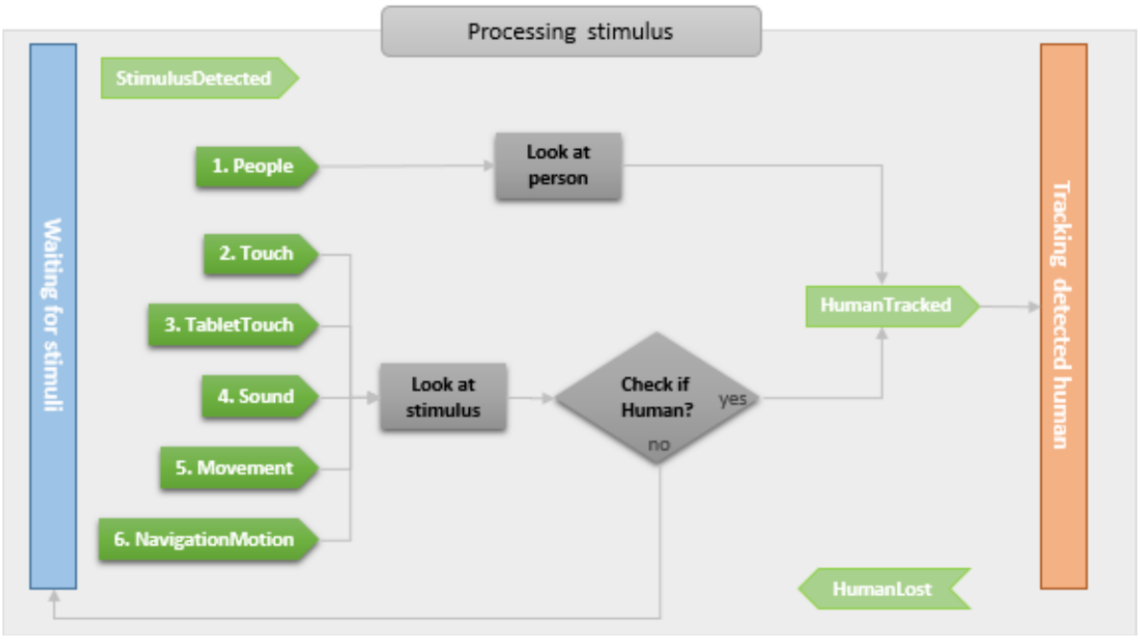


Figure 2.5: Stimuli priorities.

To enable the robot to maintain visual contact with the engaged person, the *ALMotion* API is used to facilitate programming Pepper's movements. Specifically, the information from the sound or vision APIs is used to set the target angles (i.e., the current position of the person). Figure 2.6 shows the reference system according to which Pepper rotates its head, while Figure 2.7 shows the angles used and their range of motion.

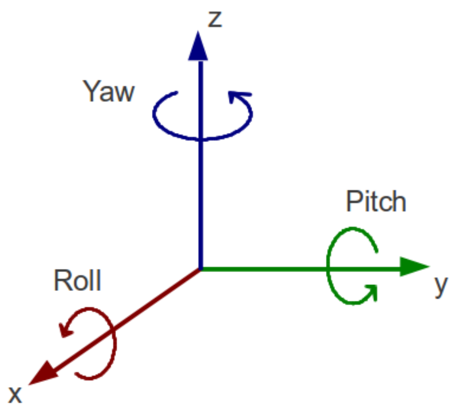


Figure 2.6: Reference system for head rotation.

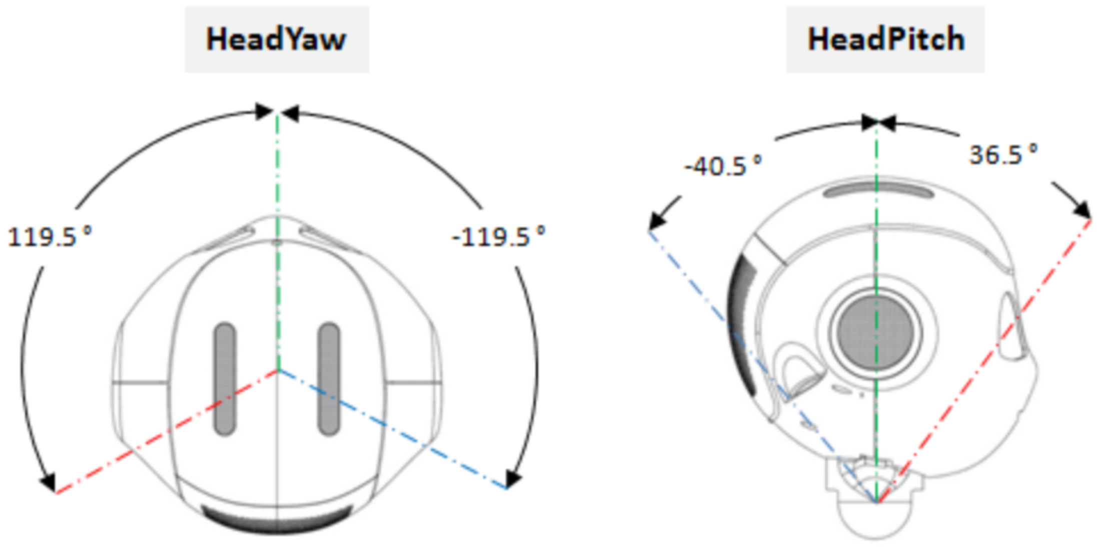


Figure 2.7: Range of rotation angles for yaw and pitch.

2.1.2 Emotion Recognition

The **Emotion Recognition** module, implemented in the *emotion_recognition.py* file, enables the identification and classification of user's facial emotions using Pepper's front-facing camera. The module acquires a frame from the robot's camera, detects the face using an *OpenCV*-based model and performs cropping and normalization. The preprocessed image is then processed by a pre-trained classifier (*emotionNet*), which assigns one of the following emotional labels:

- **surprise**
- **fear**
- **disgust**
- **happiness**
- **sadness**
- **anger**
- **neutral**

If no face is detected, the user is prompted to correctly reposition themselves in front of the camera to allow for more accurate analysis.
Below is the code of the *people_perception_node.py* file:

```python
#!/usr/bin/python3
import os
import numpy as np
import cv2
import tensorflow as tf
from tensorflow.keras.models import load_model
from emotion_recognition.utils import Session

class EmotionRecognition:
    '''
    This class can be used to perform facial emotion analysis.
    '''

    def __init__(self, ip = "10.0.1.207", port = 9559):
        """
        Initializes the EmotionRecognition class with the given IP and port.
        This method also sets up the face detector, emotion classifier and the
    video service for capturing frames.
        """
        self.ip = ip      # Pepper ip
        self.port = port  # Pepper port

        # Initialize face detector
        self.faceProto = "./emotion_recognition/opencv_face_detector.pbtxt"
        self.faceModel = "./emotion_recognition/opencv_face_detector_uint8.pb"
        self.faceNet = cv2.dnn.readNet(self.faceModel, self.faceProto)

        # Initialize emotion classifier
        self.emotionModel = "./emotion_recognition/emotion.hdf5"
        self.emotionNet = load_model(self.emotionModel)

      # List of emotions predicted by the classifier
        self.emotionList = ['surprise', 'fear', 'disgust', 'happiness', 'sadness',
    'anger', 'neutral']

        self.padding = 0.2 # Padding factor to expand the face detection area when
    cropping the image
        self.MEANS = np.array([131.0912, 103.8827, 91.4953]) # Mean color values
    for normalization of images (RGB channels)
        self.INPUT_SIZE = (224, 224) # Size to which images are resized for the
    classifier

        # Create a session
        self.session = Session(self.ip, self.port)

        # Requires the following service:
```

```python
        # ALVideoDevice needed to connect to the robot's camera.
        self.video_service = self.session.get_service("ALVideoDevice")

        # Subscribe to the front camera
        self.camera_id = 0  # Front camera
        self.resolution = 2  # Resolution (2 = 640x480)
        self.color_space = 13  # RGB
        self.fps = 20  # Frames per second
        self.video_stream = self.video_service.subscribeCamera("camera", self.camera_id, self.resolution, self.color_space, self.fps)

    def getFaceBox(self, frame, conf_threshold=0.8):
        """
        Detects faces in the given image frame and returns the bounding boxes
        around the detected faces along with the image with drawn rectangles.
        """
        frameOpencvDnn = frame.copy()
        frameHeight = frameOpencvDnn.shape[0]
        frameWidth = frameOpencvDnn.shape[1]
        blob = cv2.dnn.blobFromImage(frameOpencvDnn, 1.0, (300, 300), [104, 117, 123], True, False)
        self.faceNet.setInput(blob)
        detections = self.faceNet.forward()
        bboxes = []
        for i in range(detections.shape[2]):
            confidence = detections[0, 0, i, 2]
            if confidence > conf_threshold:
                x1 = int(detections[0, 0, i, 3] * frameWidth)
                y1 = int(detections[0, 0, i, 4] * frameHeight)
                x2 = int(detections[0, 0, i, 5] * frameWidth)
                y2 = int(detections[0, 0, i, 6] * frameHeight)
                bboxes.append([x1, y1, x2, y2])
                cv2.rectangle(frameOpencvDnn, (x1, y1), (x2, y2), (0, 255, 0), int(round(frameHeight / 300)), 8)
        return frameOpencvDnn, bboxes

    def process_emotion(self):
        """
        Captures an image from the front camera, detects faces, processes the detected face and
        predicts the emotion based on the face image using the pre-trained emotion classifier.
        Returns the predicted emotion as a string.
        """
        # Acquire a frame
        raw_image = self.video_service.getImageRemote(self.video_stream)
        if raw_image is None:
            return "Sorry, I am unable to acquire image from the camera."

        # Decode the image
        width = raw_image[0]
        height = raw_image[1]
        array = np.frombuffer(raw_image[6], dtype=np.uint8)
        frame = array.reshape((height, width, 3))

        # Face detection
        frameFace, bboxes = self.getFaceBox(frame)

        # If there are bounding boxes, select the one with the largest area (closest face)
        if bboxes:
            # Calculate the area of each bounding box (width * height)
            areas = [(bbox[2] - bbox[0]) * (bbox[3] - bbox[1]) for bbox in bboxes]

            # Find the index of the bounding box with the largest area
            max_area_idx = np.argmax(areas)

            # Select the bounding box with the largest area
            bbox = bboxes[max_area_idx]

            # Adjust crop
            w = bbox[2] - bbox[0]
            h = bbox[3] - bbox[1]
            padding_px = int(self.padding * max(h, w))
            face = frame[max(0, bbox[1] - padding_px):min(bbox[3] + padding_px, frame.shape[0] - 1),
                         max(0, bbox[0] - padding_px):min(bbox[2] + padding_px, frame.shape[1] - 1)]
            face = face[face.shape[0] // 2 - face.shape[1] // 2: face.shape[0] // 2 + face.shape[1] // 2, :, :]

            # Preprocess image
            resized_face = cv2.resize(face, self.INPUT_SIZE)
            blob = np.array([resized_face.astype(float) - self.MEANS])

            # Predict emotion
            emotionPreds = self.emotionNet.predict(blob)
            emotion = self.emotionList[emotionPreds[0].argmax()]

            return (f"Your current emotion is: {emotion}")
        else:
            return "Sorry, I couldn't detect your face, try asking the question by coming closer to me."
```

2.2 Speech-to-Text module

The **Speech-to-Text** module, implemented in the *voice_detection.py* file, is the first step in the system's voice interaction process. This module uses a ROS node to capture the user's audio from the microphone and convert it into text using *Google Speech-to-Text*, a speech recognition service. The default language is English.

The resulting text is then published to the *txt_voice* topic by the Speech-to-Text module, which acts as the publisher, while the Dialogue Server node subscribes to this topic to access the text for the analysis. This allows the dialogue system to process the user's input and generate appropriate responses.

Voice recognition is executed in the background, allowing the node to process audio continuously without blocking the main process. This is particularly useful to ensure a responsive system. Once started, the node remains in listening mode until explicitly terminated.

CHAPTER 3

WP4: DIALOGUE MANAGEMENT MODULE

3.1 Introduction to RASA

Rasa is an open-source framework designed for developing chatbots based on supervised machine learning. Compared to other well-known frameworks, such as those by Google, Amazon, and OpenAI, which offer rapid and intuitive development through graphical interfaces, Rasa stands out for its fully open-source nature and extensive customization possibilities.

The choice to adopt Rasa for this project is primarily driven by the framework's ability to directly modify the code in case of bugs or specific customization needs, as well as its flexibility and scalability, which make it ideal for creating highly customized chatbots.

RASA

Figure 3.1: RASA

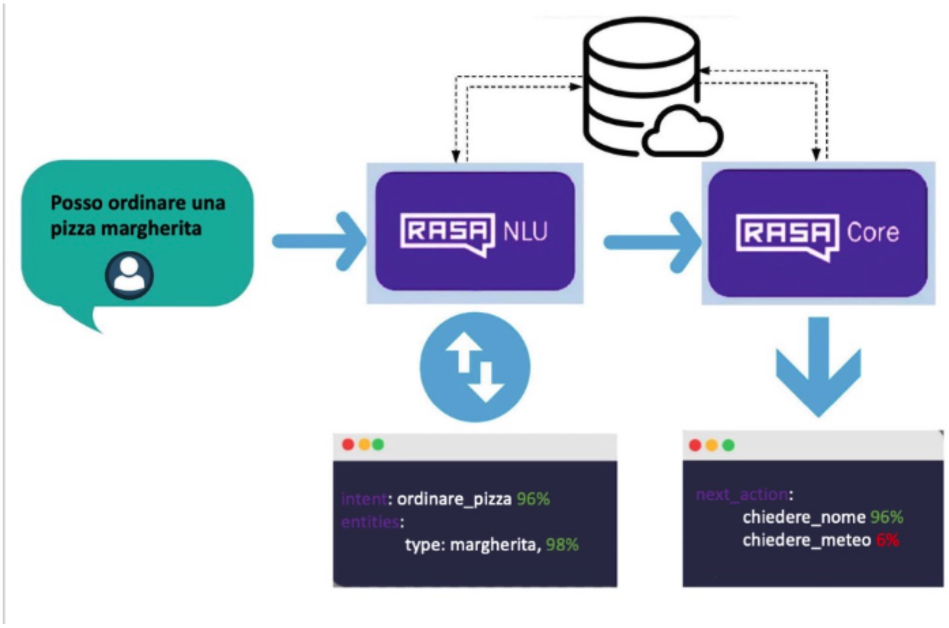
The framework adopts a modular architecture that allows the integration of various components, such as natural language understanding (NLU), dialogue management, external API integration, and other advanced functionalities.

Another significant feature of Rasa is its capability for continuous learning. Through the use of reinforcement learning, chatbots developed with this framework can progressively improve their responses by learning from user interactions and received feedback. This process enables the bot to become increasingly intelligent and to provide smoother and more natural conversational experiences.

3.1.1 How RASA Works

In general, RASA can be seen as a framework composed of two main parts, as shown in Figure 3.1:

- **Rasa NLU** (*Natural Language Understanding*), responsible for interpreting the user's text by identifying *intents* and *entities*.
- **Rasa Core**, which handles the conversational logic and flow (i.e., the *Dialogue Management*), deciding how the assistant should respond based on the current context and previous interactions.



```
intent: ordinare_pizza 96%
entities:
        type: margherita, 98%
```

```
next_action:
        chiedere_nome 96%
        chiedere_meteo 6%
```

Figure 3.2: RASA Architecture

3.2 NLU

The functioning of *RASA NLU* can be divided into two main phases: *natural language processing* (NLP) and *information extraction.*

1. The natural language processing phase consists of four fundamental steps:
   - *Tokenization*: the input text is divided into smaller units, called *tokens*, which can be words or characters.
   - *Stopword removal*: common words that do not significantly contribute to understanding the text, such as "the," "and," or "of," are removed.
   - *Lemmatization or stemming*: each word is reduced to its root form, e.g., "running" becomes "run."
   - *Part-of-Speech (POS) tagging*: each word is assigned a grammatical category, such as noun, verb, adjective, etc.
2. The information extraction phase is divided into:
   - *Intents*: identify the user's goal or request;
   - *Entities*: extract specific pieces of information from the message;

Extracting information from the user's message enables the chatbot to subsequently trigger specific actions.

3.2.1 Domain

The **domain** is a configuration file in YAML format that describes the chatbot's domain characteristics. The domain file can define the following elements:

- *intent*: user intents to be handled;
- *entity*: specific information extracted from user messages;
- *action*: actions that the chatbot can perform in response to user inputs;
- *slot*: state information that can be stored and used during the conversation or to influence the conversation flow.

3.2.2 Intent

Intents are used to identify and understand what the user is trying to communicate. Specifically, they are labels assigned to messages received from a user to classify them based on their intention.

Below is the complete definition of the *intents* within the `nlu.yml` file. In this file, in addition to example phrases for each intent, *synonyms* and *lookup tables* are also specified to optimize language recognition.

For better organization, the *intents* are divided into three macro-categories based on the type of information requested:

1. general information;
2. information related to the shopping mall;
3. information about the Mivia Contest 2025 in artificial vision.

Category 1: General Information

- **greet**: greet or start the conversation (e.g., "hi Pepper").
- **goodbye**: end the conversation (e.g., "bye," "goodbye").
- **affirm**: confirm something, express agreement (e.g., "yes," "sure").
- **deny**: deny or express disagreement (e.g., "no," "never").
- **mood_great**: communicate a very positive mood (e.g., "fantastic," "great").
- **thank_response**: thank the assistant (e.g., "thanks," "thank you").
- **bot_challenge**: ask or challenge the assistant about its bot nature (e.g., "are you a robot?").
- **ask_bot_identity**: ask the bot's name or identity (e.g., "What's your name?").
- **ask_how_are_you**: ask the assistant's state or mood (e.g., "How are you?").
- **ask_time**: ask for the time (e.g., "What time is it?").
- **ask_date**: ask for the date or day (e.g., "What's the date today?").
- **analyze_emotions**: ask to analyze/detect the emotions of the user speaking to the robot (e.g., "Analyze my face").

Category 2: Information in the Shopping Mall

- **mall_hours**: ask about opening/closing hours (e.g., "What are the opening hours of the mall?").
- **ask_directions**: ask for directions to reach a specific line or location (e.g., "How do I get to line 3?").
- **count_people**: ask how many people are present or meet certain characteristics (e.g., "How many people are in the mall with bags and hats?").

- **count_people_more_info**: ask for additional details on the people count (e.g., “How many of them are wearing a hat?”).
- **compare_gender**: compare the number of men and women (e.g., “Are there more boys or girls?”).
- **compare_attributes**: compare attributes of people (e.g., “Are there more males with hats or bags?”).
- **search_person**: search/identify a specific person within the shopping mall (e.g., “I lost my son. Can you find him?”).
- **more_info**: provide additional information about a person to facilitate the search (e.g., “He has a hat and a bag”).
- **ask_person_trajectory**: ask about the trajectory of the found person within the shopping mall (e.g., “Where else did you see him?”).
- **ask_individual_info**: ask for details about an individual (e.g., “Tell me everything you know about individual with id 11”).

Category 3: Information about the Mivia Contest 2025 in Artificial Vision

- **ask_contest_informations**: ask for general information about the competition (e.g., “Give me info about the artificial vision contest”).
- **ask_competition_date**: ask for the competition date (e.g., “When was the competition held?”).
- **ask_organizers**: ask who organized the contest (e.g., “Who organized the competition?”).
- **ask_group_members**: ask for the members of a specific group (e.g., “Who are the members of group 1?”).
- **ask_group_score**: ask for the score or performance of a group (e.g., “What is the score of group 1?”).
- **ask_position_contest**: ask for the ranking or find out who won (e.g., “Which group came second?”).
- **ask_participants_count**: ask for the total number of participants (e.g., “How many people participated in the contest?”).
- **ask_groups_count**: ask how many groups joined the contest (e.g., “How many groups joined the contest?”).
- **ask_lowest_score_group**: ask which group has the lowest score (e.g., “Which group has the lowest score?”).
- **ask_highest_score_group**: ask which group has the highest score (e.g., “Which group has the highest score?”).
- **query_high_scoring_groups**: ask for information about groups with scores above or below a threshold (e.g., “How many groups achieved a score higher/lower than 0.7?”).

Synonyms and Lookup Tables
In addition to intents, the configuration file includes *synonyms* and *lookup tables*. **Synonyms** allow mapping different variants of a word (e.g., “woman,” “girl,” “lady”) to the same entity value (e.g., ”female”). **Lookup tables**, on the other hand, are lists of words or phrases useful for improving the recognition of specific entities (e.g., all terms referring to “hat”). This approach simplifies the management of synonyms and recurring vocabulary, increasing the accuracy of entity extraction.

3.2.3 Entity

Entities are specific pieces of information extracted from the user's input during a conversation, representing parts of the sentence that identify a relevant value or concept. During the design phase, entities are defined in the domain and associated with patterns or examples of sentences containing entity values. Rasa uses machine learning models to identify and extract these entities from user input, enriching the understanding of the message's context.

For extracting *entities*, techniques such as *lookup tables* and regular expressions (*regex*) can be used. In the chatbot developed for this project, only *lookup tables* are employed. These allow for quickly mapping common terms or phrases to entity values without relying on machine learning models. This technique is particularly useful when entities have a known set of values.

Below is a summary of the main entities used in the chatbot:

- **has_hat**: indicates whether the person is wearing a hat. This information helps the system distinguish individuals wearing hats from those who are not.
- **has_bag**: indicates whether the person is carrying a bag (e.g., a backpack or handbag). It serves to distinguish those with a bag from those without.
- **gender**: specifies the gender of the person (male or female).
- **f_score**: represents the score (F1-score) achieved by a group in the Mivia Contest. It allows queries about the performance levels of the competing groups.
- **individual_id**: uniquely identifies a person by associating a specific number or code to each individual whose information is being tracked.
- **position**: indicates a group's ranking position (e.g., "first place," "second place," or "last place") in the Mivia Contest.
- **group_number**: identifies the number of a group participating in the Mivia Contest, allowing specific queries (such as the score or members) about that group.
- **line_id**: refers to a specific line in the shopping mall, useful for indicating where a person was detected or for providing directions to a particular location.

3.2.4 Actions

Actions are not strictly part of semantic interpretation but represent the tasks the assistant must execute in response to a given intent. Actions can be of two types:

- Custom actions, which are specifically implemented as Python classes extending the *Action* class provided by Rasa.
- System (predefined) actions, which are already implemented within the Rasa framework and can be used directly without additional customization.

Defining a *custom action* enables the creation of interactive and dynamic chatbots. To use them, a separate *action server* must be configured to execute the custom actions. For better organization, the actions have been grouped according to their final objective.

Custom Actions

- **General Utility Actions:**
  - *action_show_time*: provides the current time to the user.
  - *action_ask_date*: provides the current date, including the day of the week, the day of the month, and the month.
  - *action_reset_slots*: resets all chatbot slots to the value $None$. Useful for resetting the conversation state.
  - *action_out_of_context*: responds to out-of-context requests with a predefined message, indicating that the assistant cannot handle the request.

- **People Analysis Actions:**
  - *action_compare_gender*: compares the number of people of different genders in the provided data and returns a comparison message.
  - *action_compare_attributes*: compares the number of people with specific attributes (bag and hat).
  - *action_get_individual_info*: retrieves detailed information about a specific person based on their ID or other attributes such as gender, presence of a bag, or hat.
  - *action_count_people*: counts the number of people based on specific attributes (gender, bag, hat, crossed line).
  - *action_count_people_more_info*: provides a detailed count of people based on the same criteria as *action_count_people*, but includes additional details progressively.
  - *action_search_person*: searches for a person in the data based on provided attributes and returns the available information, including their last tracked path.
  - *action_show_trajectory*: shows the paths followed by people, indicating the lines crossed in chronological order.
  - *action_more_info*: provides additional information about people matching specific criteria.
- **Contest-Related Actions:**
  - *action_get_group_members*: retrieves and displays the members of a specific group identified by its group number.
  - *action_ask_group_score*: displays the score of a specific group identified by its group number.
  - *action_get_position*: provides information about the group occupying a specific ranking position.
  - *action_get_participants_count*: counts and returns the total number of participants in the contest.
  - *action_get_groups_count*: counts and returns the total number of groups participating in the contest.
  - *action_highest_score_group*: Displays the group with the highest score (F1-score) in the contest.
  - *action_lowest_score_group*: displays the group with the lowest score in the contest.
  - *action_query_high_scoring_groups*: Retrieves groups that scored higher or lower than a specific threshold provided by the user.
- **Emotion Analysis Actions:**
  - *action_analyze_face*: this action is linked to an emotion recognition module and is designed to analyze the user's facial emotions.

With this structure, the actions are divided into categories based on their main objectives, making the script more readable and organized.

System Actions

Similarly, system actions defined within the *domain* are grouped into categories.

- **Greetings and General Interactions:**
  - *utter_greet*: greets the user in a friendly manner and introduces the chatbot's capabilities.
  - *utter_ask_identity*: introduces itself as Pepper and briefly describes its purpose.
  - *utter_ask_how_are_you*: responds to general questions about the bot's state.
  - *utter_goodbye*: says goodbye to the user and concludes the conversation.

- **Predefined and Out-of-Context Responses:**
  - *utter_out_of_context*: indicates that the question is irrelevant to the chatbot's capabilities.
- **Contest Information:**
  - *utter_competition_date*: provides the date the contest took place (January 8, 2025).
  - *utter_contest_info*: describes the main features of the Artificial Vision 2025 contest.
  - *utter_organizers*: provides information about the contest organizers, such as professors and the MIVIA Lab.
- **Shopping Mall Information:**
  - *utter_mall_hours*: indicates the shopping mall's opening and closing hours.
  - *utter_directions*: suggests consulting the shopping mall map for directions.
- **User-Based Interactions:**
  - *utter_affirm*: positively confirms what the user said.
  - *utter_deny*: responds neutrally or negatively, offering an alternative or suggesting further questions.
  - *utter_thank_response*: responds to a thank-you message in a friendly manner.
- **Bot Challenges:**
  - *utter_bot_challenge*: responds to questions about the bot's identity and purpose.

3.2.5 Stories

Stories in Rasa are a fundamental tool for defining and modeling the conversational flow between the chatbot and the user. Each story describes an interaction scenario, specifying a sequence of actions the chatbot must perform in response to the inputs it receives. Thanks to this structure, it is possible to configure multiple conversation paths, adapting them to different user needs and ensuring a smooth and personalized interaction.

The definition of *Stories* is essential for training the dialogue model in Rasa, as it enables the chatbot to learn from potential interaction scenarios and respond consistently.

In this project, *Stories* are created using the **Rasa Interactive** feature, an integrated tool that allows designing conversational flows interactively. A significant aspect is that the interaction with *Rasa Interactive* is carried out by individuals with no prior knowledge of the developed chatbot. This ensured the creation of unbiased and realistic dialogue scenarios, based solely on the system's intuitiveness and the chatbot's responses, which improves the model's generalization ability.

During the interaction, *Rasa Interactive* suggested actions to be performed in response to user messages, providing the developer with the ability to confirm, correct, or modify these actions. In this way, it is possible to ensure consistency and realism in the conversational flows. The defined conversations are automatically saved in a YAML file (*stories.yaml*), making them ready for use during the model training phase.

This method significantly simplified the process of creating *Stories*, reducing the time required to manually write dialogue scenarios. Furthermore, it contributed to improving the overall quality of the chatbot by ensuring that the conversational flows are based on realistic interactions and consistent with user requests.

3.2.6 Slots

Slots are variables that allow the chatbot to store and reuse information acquired during the conversation, maintaining context across dialogue turns. This enables the system to provide more precise and appropriate responses without repeatedly asking for the same data.

A concrete example of the usefulness of *slots* occurs when a user wants to search for a person. They might start with: *"I'm looking for a person"*, then provide additional details such as *"He's male"* and *"He's wearing a hat"*. The system must remember these pieces of information (gender, presence of a hat) without asking redundant questions.

Similarly, when a user refers to a specific group from the *Mivia Contest*, they might begin with: *"Who came second in the contest?"*, then follow up with *"Who are the members of this group?"* or *"What is this group's F1-score?"*. In such scenarios, the chatbot must retain the reference to the mentioned group to respond coherently, avoiding the need to ask for the group number again.

In both cases, *slots* enable the chatbot to preserve prior information, enhancing the conversational experience and ensuring context-aware responses.

3.3 Dialogue Management

The *Dialogue Management* module, often referred to as *Rasa Core*, is responsible for managing the conversational flow within the chatbot. This module uses the information provided by the NLU (Natural Language Understanding) model, context data, and the learned *stories* to determine the most appropriate action to take at each point in the conversation.

The dialogue management process is divided into three main phases:

1. **Intent Prediction:** the user's message is interpreted by the NLU model, which identifies the main intent and associated entities.

2. **Context Data Consultation:** the values of the *slots* and the previous steps in the conversation are used to maintain the state and context of the conversation.

3. **Action or Response Selection:** based on the *stories*, defined rules, and the current conversation state, *Rasa Core* selects the next *action* or response message to send to the user.

The correct integration between *Rasa NLU* and *Dialogue Management* allows for the creation of a **proactive** and **context-aware** conversational assistant capable of handling complex conversations and providing relevant responses in real time.

A key aspect is the `config.yml` configuration file, which defines the NLU pipeline components and dialogue management policies. The configuration used in this project is as follows:

```yaml
language: en

pipeline:
- name: WhitespaceTokenizer
- name: RegexFeaturizer
- name: LexicalSyntacticFeaturizer
- name: CountVectorsFeaturizer
- name: DIETClassifier
  epochs: 100
  constrain_similarities: true
- name: EntitySynonymMapper
- name: ResponseSelector
  epochs: 100
- name: FallbackClassifier
  threshold: 0.6

policies:
- name: MemoizationPolicy
  max_history: 5
- name: TEDPolicy
  max_history: 10
  epochs: 100
- name: RulePolicy
  core_fallback_threshold: 0.3
  core_fallback_action_name: "action_default_fallback"
  enable_fallback_prediction: True
```

Listing 3.1: File di configurazione `config.yml`

The *NLU pipeline* consists of several components responsible for processing user messages to extract intents and entities. These include:

- **WhitespaceTokenizer:** splits the text into words (*tokens*) based on whitespace.
- **RegexFeaturizer:** Identifies specific patterns in the text, such as numbers or particular formats.
- **LexicalSyntacticFeaturizer:** extracts lexical and syntactic features from messages to enhance understanding.
- **CountVectorsFeaturizer:** converts text into numerical vectors using word counts.
- **DIETClassifier:** an advanced deep learning-based model for intent classification and entity extraction, configured with 100 epochs to ensure an accurate training phase.
- **EntitySynonymMapper:** maps entity synonyms to canonical values, such as ”laptop” and ”notebook.”
- **ResponseSelector:** Selects the most appropriate response from the available options.
- **FallbackClassifier:** handles unrecognized messages by triggering a predefined response when confidence falls below the threshold of 0.6.

The defined *policies* control the behavior of the *Dialogue Management*:

- **MemoizationPolicy:** stores simple conversation sequences using a maximum history of 5 steps. It is useful for predefined responses or repetitive scenarios.
- **TEDPolicy:** a deep learning-based policy designed to handle complex conversational flows. Configured with a maximum history of 10 steps and 100 epochs to account for context and improve accuracy.
- **RulePolicy:** manages strict dialogue rules, such as fallback responses or specific scenarios. Configured with a fallback threshold of 0.3 and a default action (`action_default_fallback`) for uncertain situations.

This configuration allows the chatbot to interpret inputs, maintain the conversation context, and respond dynamically and appropriately, ensuring flexibility and adaptability even in complex scenarios.

CHAPTER 4

WP5 & WP6: TESTING

To validate the system, it was decided to perform the testing in two phases: testing individual components and integration testing.

4.1 Testing Individual Components

For the individual component testing, the modules speech-to-text, dialogue server, text-to-speech, and people perception were empirically tested by observing their correct functioning during execution.

For the RASA testing, a more methodological approach was possible. This module was tested using the command:

```
rasa test nlu --cross-validation
```

which performs cross-validation on the NLU. Table 4.1 shows the results of the *accuracy*, *F1-Score*, and *precision* metrics for the evaluation of intents and entities on the test set.

Metrics	**Intent Evaluation**	**Entity Evaluation**
Accuracy	0.865	0.995
F1-Score	0.859	0.971
Precision	0.884	0.987

Table 4.1: Performance metrics for intent and entity evaluation on test set.

The table shows the excellent results obtained across all metrics, both for intents and entities. Additionally, the following results are presented:

- **Figure 4.1** represents the confusion matrix for entity extraction. Overall, the results are excellent, as most entities are correctly recognized (cells along the main diagonal). The main challenges concern the entities `position` and `individual_id`, which sometimes experience classification errors (false positives or negatives). This issue seems to arise from the fact that both entities refer to numerical values (groups or individuals) and include training examples that do not always clearly highlight their lexical distinction, leading to overlaps.
- **Figure 4.2** shows the confusion matrix for intent classification. Here too, the overall performance is high, with most intents correctly predicted. The most frequent confusions occur between `ask_highest_score_group` and `ask_lowest_score_group`, as well as between `ask_contest_informations` and `ask_individual_info`, likely due to strong lexical similarity in the training examples for these intents.
- **Figure 4.3** illustrates the confidence level distribution for correct and incorrect entity predictions. Generally, correct predictions exhibit very high confidence values, demonstrating the model's effectiveness in recognition. Errors, although occasionally associated with high confidence levels, remain relatively few compared to the total.

- **Figure 4.4** shows the confidence level distribution for correct and incorrect intent predictions. In general, correct predictions record significantly high confidence values, indicating strong certainty from the model. At the same time, there is a relatively small number of errors associated with high confidence, while most incorrect predictions are linked to medium to low confidence levels. This phenomenon could pose a challenge in cases of unclear user formulations. However, the considerable prevalence of correct predictions with high confidence demonstrates the model's strong ability to correctly interpret user requests in most cases.

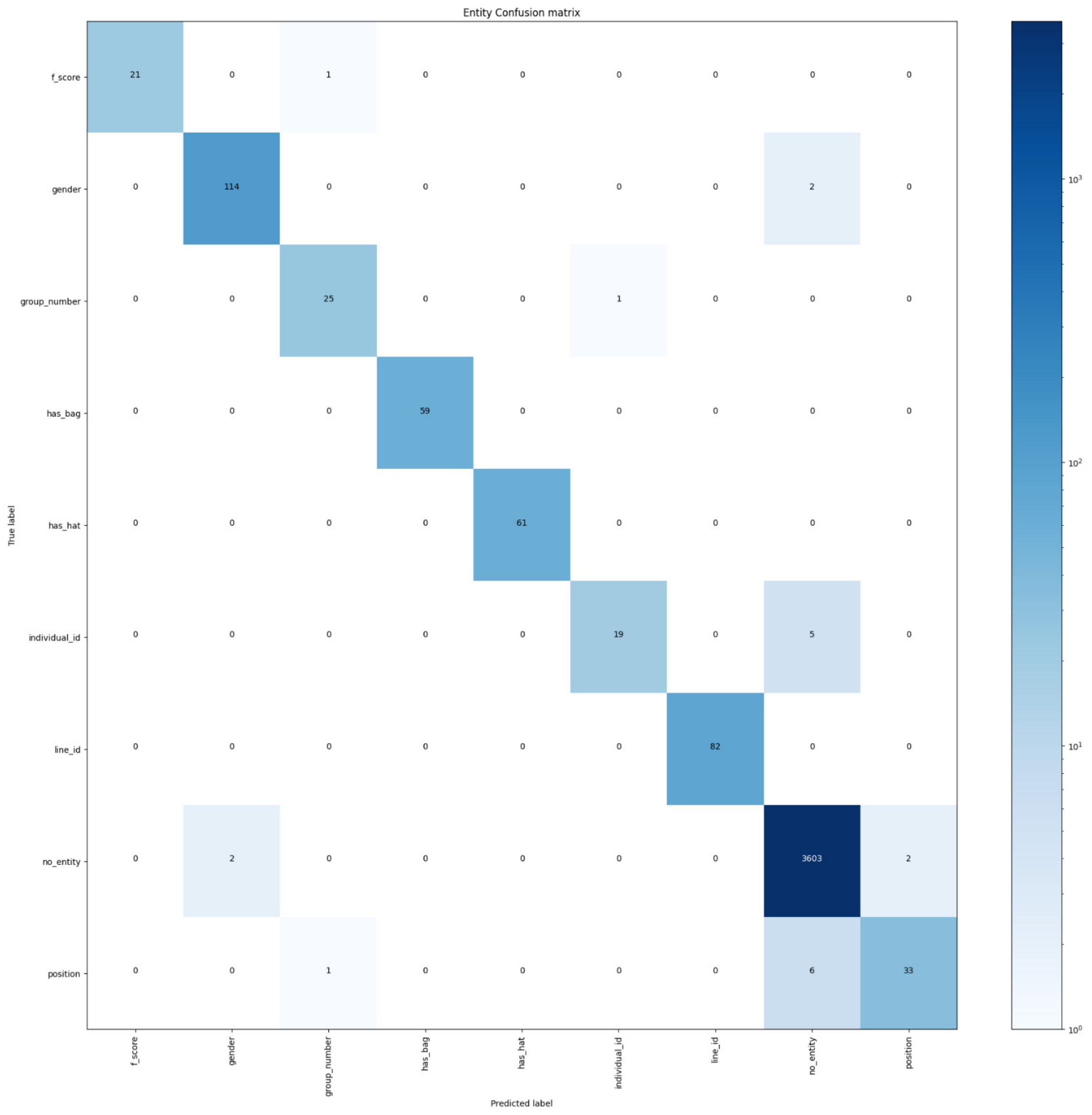


Figure 4.1: Entity Prediction Confusion Matrix.

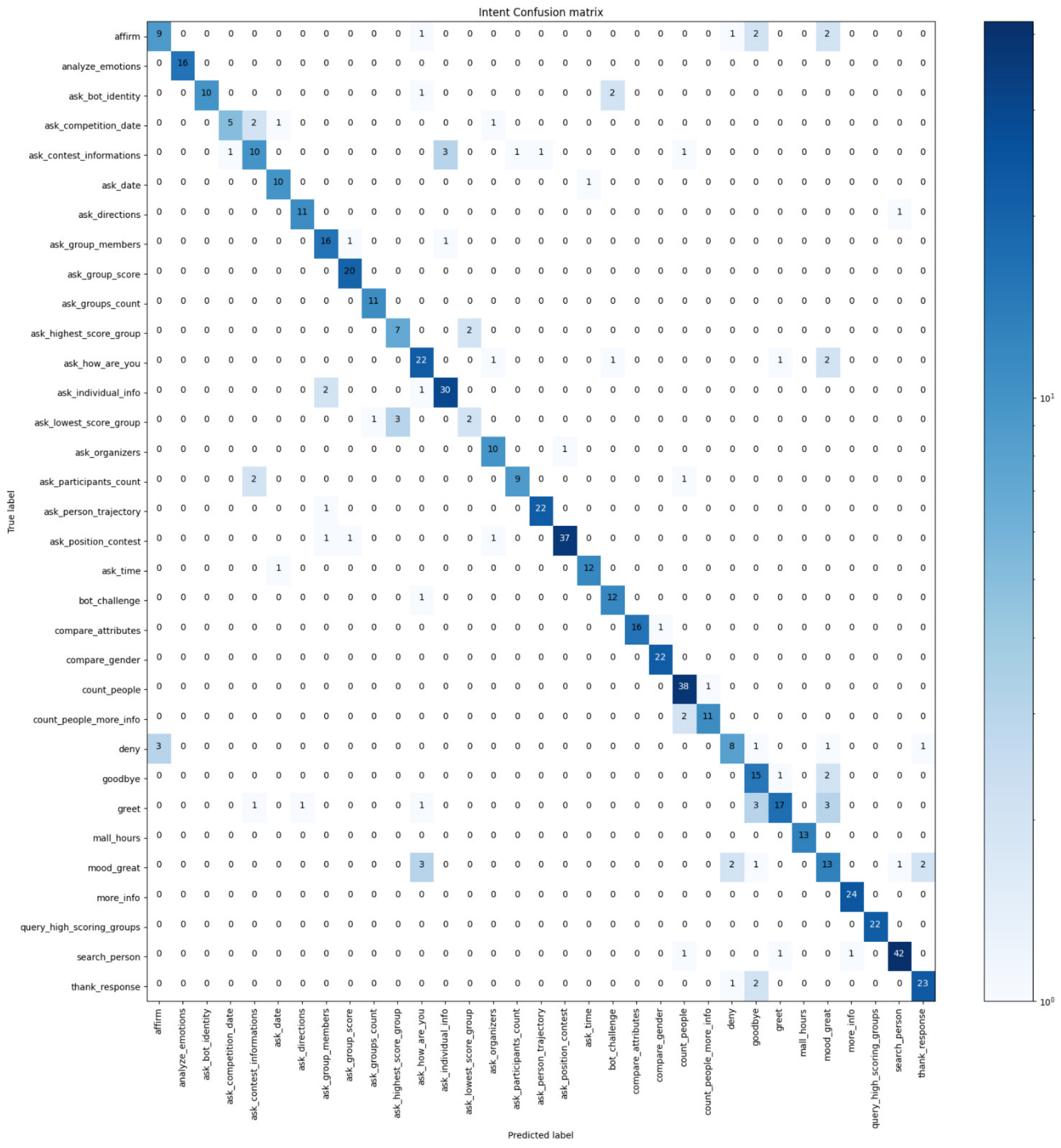


Figure 4.2: Intent Prediction Confusion Matrix.

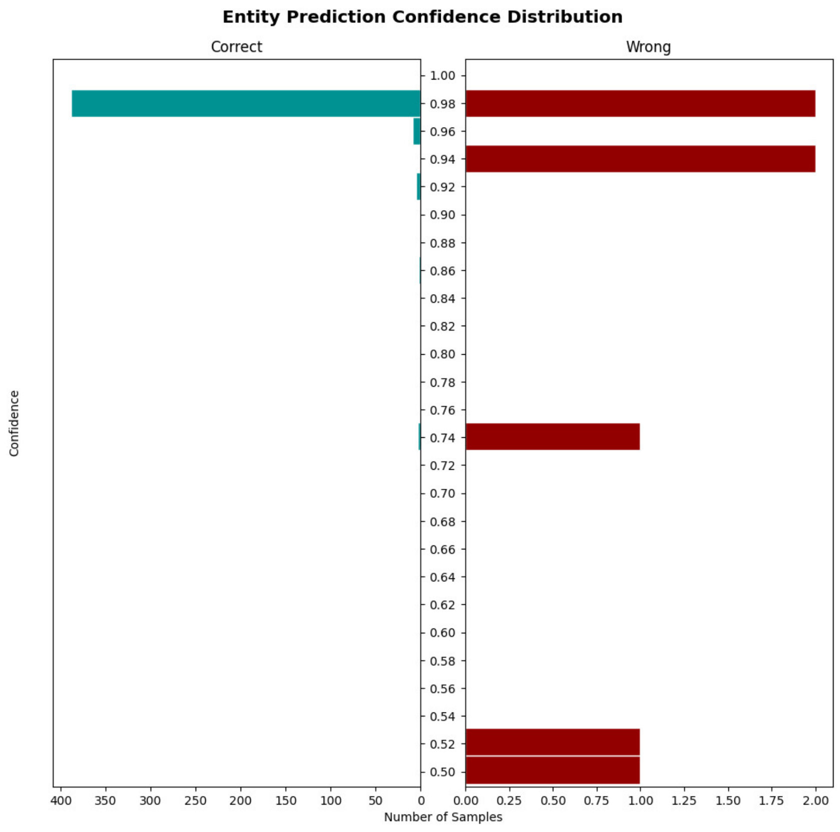


Figure 4.3: Entity Prediction Confidence Distribution.

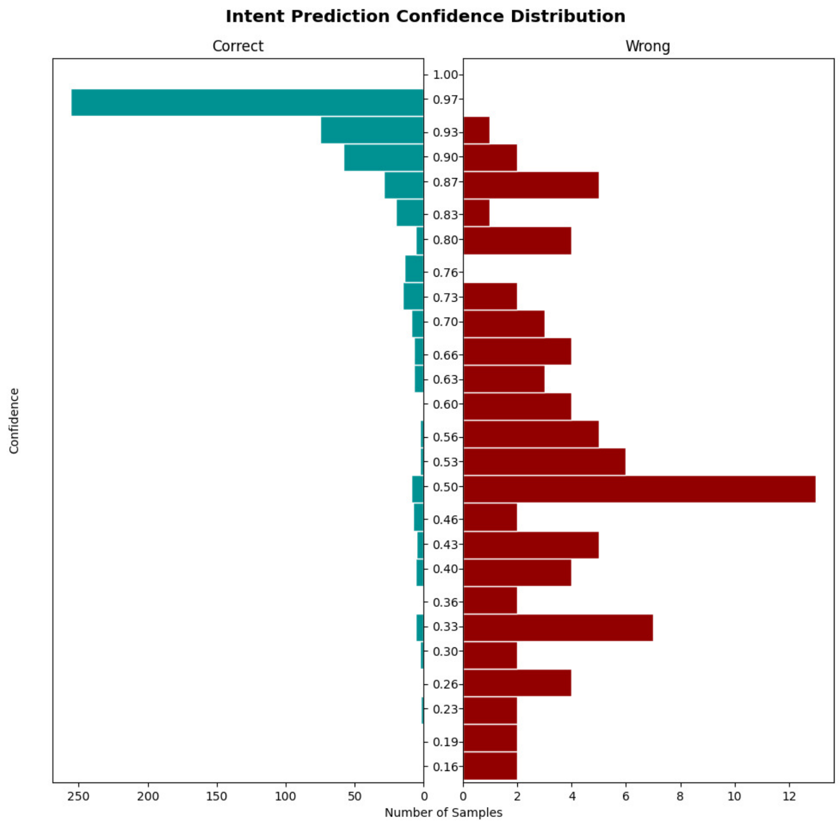


Figure 4.4: Intent Prediction Confidence Distribution.

4.2 Integration Testing

Once the individual components were validated, it was necessary to perform integration testing to ensure that the modules communicated correctly while maintaining their functionalities. To achieve this, an empirical test was conducted by directly interacting with Pepper. The results can be observed in the videos available in the project's folder.

LIST OF FIGURES