Project Report

Project 1

Authors	Ceglia Salvatore Ferrara Luigina Gargiulo Anna Maruotto Ida	0622900145 0622900144 0622900139 0622900135
Group Number	2	

Summary

Goal of the Project

The aim of the project is to control Nao so that it pronounces sounds associated to several objects, and then the patient imitates the vocal sounds after the demonstration.

Realization

The robot is placed on the ground, in front of the patient in a predefined starting position. The objects of interest, known a priori and provided for the project (cow, sheep, car, train, dog), are placed on the ground in front of the robot in predefined positions.

The robot knows a priori the position of each object with respect to the initial position and the sound that must be reproduced "verbally" in correspondence with each object.

The robot points at the object of interest, and, as required by the procedure, pronounces the name of the object and its sound.

Then, the robot waits for 5 seconds for the patient's reaction:

- If the patient reproduces the sound correctly, the robot moves on to the next object in the sequence;
- Otherwise, it plays the sound again and waits for 5 seconds.

After 3 unsuccessful attempts, the activity is unsuccessful. Unanswered attempts add up across different objects.

At the end of the activity, if the result is positive, the robot pronounces "Very well".

Solution

Introduction

The solution was designed in different packages:

1. ***nao_nodes*** allows to control Nao, and includes:
   - the *wakeup_node* module, that allows us to wake up the robot;
   - *the arm_motion_node and the head_motion_node* modules, designed to make the arm and the head of the robot move towards the object of interest;
   - the *microphone_node* and *text2speech_node* modules, used to make the robot record sounds and speak;
   - the *audio_player_node* module, which allows to reproduce the already recorded object sounds, previously uploaded on the memory of the robot, that the patients should repeat;
   - the *led_node* module, that include the mechanism to change the color of Nao's eyes to give to the patient visual feedback during the exercise.

2. ***sound_recognition*** allows to record, listen and recognize the sounds, and includes:
   - the *classifier* module, used to recognize the sounds, pre–trained with the sounds of the five objects;
   - the *voice_activity_detection* and *ros_vad* modules, used to initialize the microphone and therefore record the sounds;
   - the *sound_recognition_node* module, used to classify the sound heard with respect to the already available sounds, with the possibility to store it assigning it to the correct class;
   - the *audio_detection* module, that allows to record, during each step of the exercise, the patient's response in the five second time lapse after the robot's request;
   - the *audio_classification_node* module, designed to load the *classifier*, to gather information about the pronunciation of the patient and to classify the sound.

3. ***project*** allows to manage the application logic, and includes:
   - the *main* module, that starts the application and manages the application flow;
   - the *nao_motion* module, that allows to move Nao in order to point to the objects, whose positions are known a priori;
   - the *text2speech_pyttsx3_node* module, that allows to use the Pyttsx3 engine to speak through the speakers of the portable computer.

To accomplish the goal of the project without working directly with Nao all the time, we decided to test most of the implemented features on a portable computer. To do so, the user can select two different modes: the first mode allows to speak with the patient and check their pronunciation without the help of the robot (*test mode*); in the second mode, the robot is part of the loop and directly interacts with the patient (*robot mode*).

Motion

The movements designed for Nao are the arm and head motions.

For each object, a specific movement for the arm and the head of the robot are implemented, with a specific set of angles that identifies each position. The first three objects are pointed at with the left arm, then switching to the right arm to point at the last two objects – as this feels more human.

```
yelbow = [-68.6, -88.7, -94.7, 88.7, 68.6]
relbow = [-0.7, -23.5, -17.8, 23.5, 0.7]
pshoulder = [53, 53, 53, 53, 53]
rshoulder = [42.5, 18.1, 2.6, -18.1, -42.5]
left_arm = [True, True, True, False, False]
phead = [5, 5, 5.4, 5, 5]
yhead = [50, 30, 0, -30, -50]
speed = [0.2, 0.15, 0.15, 0.15, 0.2]
```

For each position, the angle to choose corresponds to the index of the arrays. They represent elbow yaw, elbow roll, shoulder pitch, shoulder roll, if the movement is performed by the left arm, head pitch, head yaw and the speed. The further the hand is, the faster the arm will move - again, because it is more natural.

The first position (index 0) is the left arm fully extended to the left of the robot with head pointing in the same direction. Then, in the following positions the arm and the head start moving towards its right, switching arm after pointing to the center.

The movements are controlled by a Publisher/Subscriber architecture: The publishers are found in the *project/nao_motion* module, where, when a specific position is desired, the corresponding function to perform the movement sends information through a publisher that publishes on a specific topic - defined as so:

‘/ arm_rotation / *shoulder* or *elbow* / *left* or *right* / *angle*’ for the arm;

‘/head_rotation/ *angle*’ for the head.

The subscribers are found in *nao_nodes/arm_motion_node* and *nao_nodes/head_motion_node*, which, after receiving the message on the specific topic, communicate the movement to Nao using the “ALMotion” Proxy.

This functionality is not active in the test mode.

Training the Classifier

The classifier needed to be trained on five different sounds (train, cow, car, dog and sheep), so we repeated the sounds until the performance of the classifier was acceptable.

In total, the classifier is trained on 6 samples for each object, summing up to 30 sounds.

The classifier neural network used was part of the provided modules found in the *sound_recognition/classifier* package.

Text To Speech Functionality

The text to speech functionality is used to make the robot vocalize the sentences.

This specific functionality is accomplished with two different modules (implemented as services), selected depending on the mode chosen by the user.

If the user selects the *test* mode, the application runs the text to speech module that initializes the *Pyttsx3* engine to use the pc speakers to synthesize the sentences.

If the user selects the *robot* mode, the application runs the *text to speech* and *audio player* modules that use Nao's speakers to perform the task.

The logic of the two services is very similar. When the function associated with the service is invoked, it takes as input a *String* representing the text to pronounce. The call returns another string corresponding to an *ack*.

Test Mode - Text2SpeechNode_Pyttsx3

This module is a replacement for Nao when the robot is not available. It is found in *nao_nodes/scripts/text2speech _pyttsx3_node*. It initializes the engine - setting parameters such as *rate* (the number of words/minute), the *voice* of the engine, the *language* used by the engine, and, when started, it associates the service call with the corresponding callback (the 'say' function).

Robot Mode - Text2SpeechNode

This module initializes the "ALTextToSpeech" Proxy, needed to use this functionality with the robot. The service call is associated with the 'say' callback. The function specifies the *words/minute rate* and the *language* used by the engine (English) to speak and the *volume* of the speakers, they will be used no matter what the settings of the robot are.

Listening

The *sound_recognition* package provides two different ways to perform the listening task:

- The always enabled one, implemented in the *voice_activity_detection* module, it allows to capture every sound near the microphone, that is used for training the classifier;
- The enabled-on request one, implemented in the *audio_detection* module, it allows to capture the sounds only in a fixed-length time window, that is used during the exercise.

Listen-Speak interchange

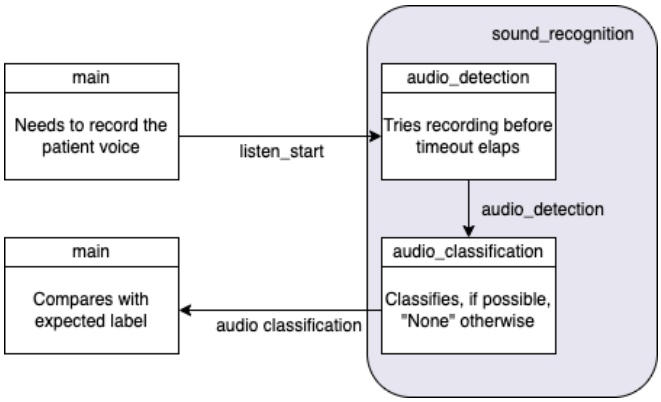
After the pronunciation of the sentence, when the application needs to record the patient, the main node publishes on the *'/listen_start*' topic. The '*sound_recognition/audio_detection'* is a subscriber to this topic and after receiving a message it can start the listening step of the task. Since the patient can repeat the sound only in a, settable, five seconds time lapse it was crucial that the robot (or the

computer) would not start the listening countdown before finishing pronouncing the sentences, or worst listening itself.

So, when the pronunciation task is over, a message on *'/listen_start'* topic is published, then in *'sound_recognition/audio_detection'* module the 'listen' callback is invoked. In this callback the audio capture engine tries to record sounds in next *timeout* (=5) seconds. If it fails, it publishes a dummy message on *'/audio_detection'* topic, otherwise it publishes the recorded audio and its start and end timestamps. The dummy message is made up of zeros as data and it has the end time equal to start time, so it is recognizable, because no message can have such timestamps if it is a real recorded audio. Then, *'sound_recognition/audio_classification'* module is waiting for this message on *'/audio_detection'* topic. We implemented two different behaviors depending on the contents of the message, but they are handled by the same modules:

If the true sound has been recorded, *'sound_recognition/audio_classification'* classifies it using cosine similarity and publishes the hypothesis, the confidence, and the class label on *'/audio_classification'* topic. The class label is equal to the hypothesis if the confidence is higher than the *threshold*, otherwise it is None. In *'project/main'* module this message is captured and compared to the correct label.

If no sound has been recorded, *'sound_recognition/audio_classification'* does a control on the message and recognizes that classification is not possible, so it publishes "None" as both class label and hypothesis with confidence 1.0 (100%) on *'/audio_classification'* topic. A class that matches this label doesn't exist so in *'project/main'* module this message is captured and comparing to the correct label it will produce an error, as well as it happens when the predicted class label is different from desired one.



Design Choices

Timeout implementation

To give exactly 5 seconds to the patient the timeout hasn't been implemented simply using the *timeout* parameter during the wait for message of *'/audio_classification'* topic. The application receives a message on this topic after the whole process of recording and classifying. So, timeout is the time that the listen module waits before publishing the dummy message on "*/audio_detection"* topic, only if nothing has been recorded. This means that the *timeout* time is actually the exact time to pronounce the answer.

Recording sounds

When the patient feedback is recorded it is saved into a folder named "*records*" and it is named as the predicted class. In this way the therapist can further investigate other aspect of pronunciation.

```
# Storing file
                onlyfiles = [f for f in listdir(os.path.join(dir_path,'records')) if isfile(
                    join(os.path.join(dir_path,'records'), f))]

                # Check for other file with the same name
                i = len(onlyfiles) + 1

                write(os.path.join(dir_path,'records',
                                   f"{format(i, '04d')}"+hyp+".wav"), 16000,voice.astype(np.int16))
```

Eyes feedback

To give visual feedback to the patient, e.g., letting them know that the robot is listening, and they can reproduce the sound, the application also publishes on the topic '*/led/color*' to control the color of the robot's eyes. During the speaking phase, the eyes are **white** while during the listening phase their color is **yellow**. Each time the task for an object is accomplished, the color of the eyes turns **green** to give positive feedback. Otherwise, it turns **red**.

```
WHITE = 0x00ffffff
GREEN = 0x0066ff66
YELLOW = 0x008080
RED = 0x00ff0000
```

Application Logic

The main file of the application is implemented as so:

First, the application checks that the objects specified by the user belong to one of the five acceptable categories and saves their order.

Second, depending on the mode chosen, one of the two engines for the text2speech functionality is started. Since the sentences to pronounce are different for the two engines, it is necessary to also associate the right dictionary of words to pronounce depending on the engine chosen: *sounds* for the robot and *calls* for the pc.

We decided to let the pyttsx3 engine also pronounce the sound of the object, whereas the robot reproduces an already recorded sound for each object.

Third, the application subscribes to the *'/system_ready'* topic and waits for a message on this topic before selecting one of the objects from the list to work with it. The subscription to the topic is essential in order to allow the robot to start the execution of the defined activity only after the classifier of the trained sounds is loaded. During this time lapse the robot/computer *introduces* the exercise that they will do shortly thereafter.

Fourth, the robot can start performing the task whose steps are managed by the main function 'work_with'. Once the object is selected, the robot is moved in the position corresponding to the index of the object and pronounces the sentences "This is an *object*, repeat *recorded_sound_of_call*".

Fifth, the application publishes on '*/listen_start*' to notify that the listen phase can start. To understand if the patient has reproduced the sound correctly, the application also subscribes to the *'/audio_classification'* topic waiting for the result of the prediction made by the classifier on the listened sound. The sound has been reproduced correctly if the predicted sound class coincides with the selected object.

If the patient doesn't repeat the call correctly, it is given three chances to get it correctly, else, the application ends. If the task has been accomplished successfully, the robot pronounces "Very well", otherwise "Retry".

Code snippet

'main'

```
if __name__ == '__main__':

    objs , test, max_errors, patient= parse_args()
    check(objs)
    if test == '1':
        rospy.wait_for_service('tts_pyttsx3')
        tts = rospy.ServiceProxy('tts_pyttsx3', Text2Speech_pyttsx3)
        sents = calls
        saycall = pc_saycall
        stand = no_op
    else:
        rospy.wait_for_service('tts')
        tts = rospy.ServiceProxy('tts', Text2Speech)
        rospy.wait_for_service('audio_play')
        sents = sounds
        saycall = nao_saycall
        stand = wakeup
        rospy.wait_for_service('wakeup')

    m = Motion()
    rospy.init_node('main_node', anonymous=True)
    pub = rospy.Publisher('/listen_start', String, queue_size=3)
    color_pub = rospy.Publisher('/led/color', Int32, queue_size=1)
    rospy.Subscriber('/audio_classification', ClassifiedData)
    errors = 0

    tts('Hello' + patient)
    introduction()

    rospy.wait_for_message('/system_ready', Bool)
    color_pub.publish(WHITE)
    tts("Okay I am Ready Let's go")
    rospy.sleep(1)
    while not rospy.is_shutdown():
        for obj in objs:
            while(work_with(obj, m, objs.index(obj))!=obj):
                print()
                errors += 1
                color_pub.publish(RED)
                if errors == max_errors:
                    tts('Retry')
                    exit_routine()
                    sys.exit()
            color_pub.publish(GREEN)
        tts('Very well')
        break

    exit_routine()
```

'work with' function

```
def work_with(obj, m, pos):
    rospy.loginfo("We are working with "+obj)
    point_to_pos(m, pos)
    saycall(sents[obj])
    rospy.sleep(1)
    global color_pub
    color_pub.publish(WHITE)
```

```
    global pub
    pub.publish("done")
    try:
        data = rospy.wait_for_message('/audio_classification', ClassifiedData)
        rospy.loginfo('Predicted class:' + data.hypothesis + ' with ' + str(data.probability) + '%
'+ 'of confidence')
        label = data.class_label
    except:
        label = None
        rospy.loginfo("no sound")
    finally:
        stand()
        return label
```

'exit_routine'

```
def exit_routine():
    global color_pub
    color_pub.publish(WHITE)
    stand()
```

'introduction'

```
phrases= ['I am Nao','In a minute we will do an exercise together',
          'We will play with five objects', 'The ones you see in front of me',
          'Once I pronounce the sound of the indicated object', 'You will repeat it',
          'Look at the color of my eyes', 'When they are yellow I am listening you',
          'If they turn green the task is over','If they turn red the task will be repeated',
          'Please wait wait a few moments', 'I am getting ready to do the exercise']

def introduction():
    for phrase in phrases:
        rospy.sleep(2)
        tts(phrase)
```

'point_to_pos'

```
def point_to_pos(m, p):
    m.arm_elbow(yelbow[p], relbow[p],speed[p], left_arm[p])
    m.arm_shoulder(pshoulder[p], rshoulder[p], speed[p], left_arm[p])
    m.head(phead[p], yhead[p], speed[p])
```

Starting the application

Preliminary settings

Before executing the application launchers, if the hardware is the computer, it is mandatory to install pyttsx3 library using the shell file *vm_setup.sh*.

```
#!/bin/bash

sudo apt update
pip3 install --upgrade pip
pip install pyttsx3==2.7
sudo apt install libespeak1
```

Launch files

There are many ways to start the application depending on the goal and the hardware available.

To train the classifier, the *classifier_train.launch* file can be used.

```
<launch>
  <arg name="test" default = "0"/>
  <arg name="nao_ip" default="$(optenv NAO_IP 10.0.1.200)" />
  <arg name="nao_port" default="$(optenv NAO_PORT 9559)" />
  <node pkg="nao_nodes" type="microphone_node.py" name="microphone_node" required="true" args="--
ip=$(arg nao_ip) --port=$(arg nao_port)" unless="$(arg test)" output="screen" />
  <node pkg="sound_recognition" type="voice_activity_detection.py" name="voice_activity_detection"
required="true" output="screen" args="--test $(arg test)"/>
  <node pkg="sound_recognition" type="sound_recognition_node.py" name="sound_recognition_node"
required="true" output="screen" />
</launch>
```

This file launches *sound_recognition_node* and *voice_activity_detection* that are the base logic to train the classifier. If the selected mode is *test*, two modules are enough because *voice_activity_detection* selects the input source by itself, depending on *test* value. Otherwise, it launches the *microphone_node* file to start all the node that communicates with the robot's interface.

To start the classifier there are a specific launch file, *classify.lauch*, that runs *audio_classification* and *audio_detection*. These modules take as input *threshold*, *test*, and *timeout*. It comprises all the modules needed to classify the sounds.

```
<launch>
  <arg name="test" default = "0"/>
  <arg name="threshold" default = "0.75"/>
  <arg name="timeout" default = "5"/>
  <node pkg="sound_recognition" type="audio_detection.py" name="audio_detection" required="true"
output="screen" args="--test $(arg test) --timeout $(arg timeout)"/>
  <node pkg="sound_recognition" type="audio_classification_node.py" name="audio_classification_node"
required="true" output="screen" args="--threshold $(arg threshold)"/>
</launch>
```

To start the whole application, only the *system.launch* file, located in the *project* package, is required.

The launch file always launches the *main* node and *classify.launch*, furthermore, it either starts *nao_bringup.launch* or the *text2speech_pyttsx3* node depending on the working mode of the system.

```
<launch>
  <arg name="object1" default = "train" />
```

```
  <arg name="object2" default = "cow" />
  <arg name="object3" default = "sheep"/>
  <arg name="object4" default = "car" />
  <arg name="object5" default = "dog" />
  <arg name="threshold" default = "0.75" />
  <arg name="timeout" default = "5" />
  <arg name="errors" default = "3" />
  <arg name="test" default = "0"/> <!--0 means testing false any other value will refer to true -->
  <arg name="patient" default = "Salvatore"/>
  <arg name="nao_ip" default = "10.0.1.200"/>
  <arg name="nao_port" default = "9559"/>

  <include file="src/nao_nodes/launch/nao_bringup.launch" unless="$(arg test)" >
    <arg name="nao_ip" value="$(arg nao_ip)"/>
    <arg name="nao_port" value="$(arg nao_port)"/>
  </include>
  <include file="src/sound_recognition/launch/classify.launch" >
    <arg name="test" value="$(arg test)"/>
    <arg name="threshold" value="$(arg threshold)"/>
    <arg name="timeout" value="$(arg timeout)"/>
  </include>
  <node pkg="project" type="text2speech_pyttsx3_node.py" name="text2speech_pyttsx3_node"  if ="$(arg
test)" required="true" output="screen"/>
  <node pkg="project" type="main.py" name="main_node" required="true" args="--1 $(arg object1) --2
$(arg object2) --3 $(arg object3) --4 $(arg object4) --5 $(arg object5) --test $(arg test) --errors
$(arg errors) --patient $(arg patient)" output="screen" />
</launch>
```

The presence of $object_i$ arguments allows the user to totally customize the order of the objects. When launching the application specifying, for example, *object1:="dog"*, this leads to having as first object the dog. Other names outside the predefined ones are not allowed: the application ends informing the user that Nao doesn't know such object.

When launching this file, it can be also set the classifier threshold, by default set at 75%, the time to wait for an answer before assign the error, and the maximum number of errors. Thus, the application is fully customizable.

The launch file *nao_bringup.launch* starts service modules: *wakeup_node.py*, *microphone_node.py*, *text2speech_node.py*, *audio_player.py*, and modules that wait messages on topic: *head_motion_node.py*, *arm_motion_node.py*, *led_node.py*.

Test

To test the physical features, we first developed some code based on the application *Choregraphe*, that displays a virtualization of the robot which the possibility to manipulate its limbs. The application helped understand the movements associated to each angle (roll, pitch, yaw) of both the head and the arm, and allowed to select the desired positions of the two by visualizing the movements. Once the angles had been decided, we ran *nao_bringup.launch* to test if the robot moved as desired.

Running the *system.launch*, the application also gave feedbacks regarding the implementation of the publisher/subscriber architecture, even if it is not provided with a virtualization of the microphone or with speakers.

The "ALTextToSpeech" Proxy can be tested through a Dialog pane that displays the sentences the robot would have pronounced. So, at the end, it was possible to test that the main node wrote on the "/listen_start" topic, and that the application run the listen callback.

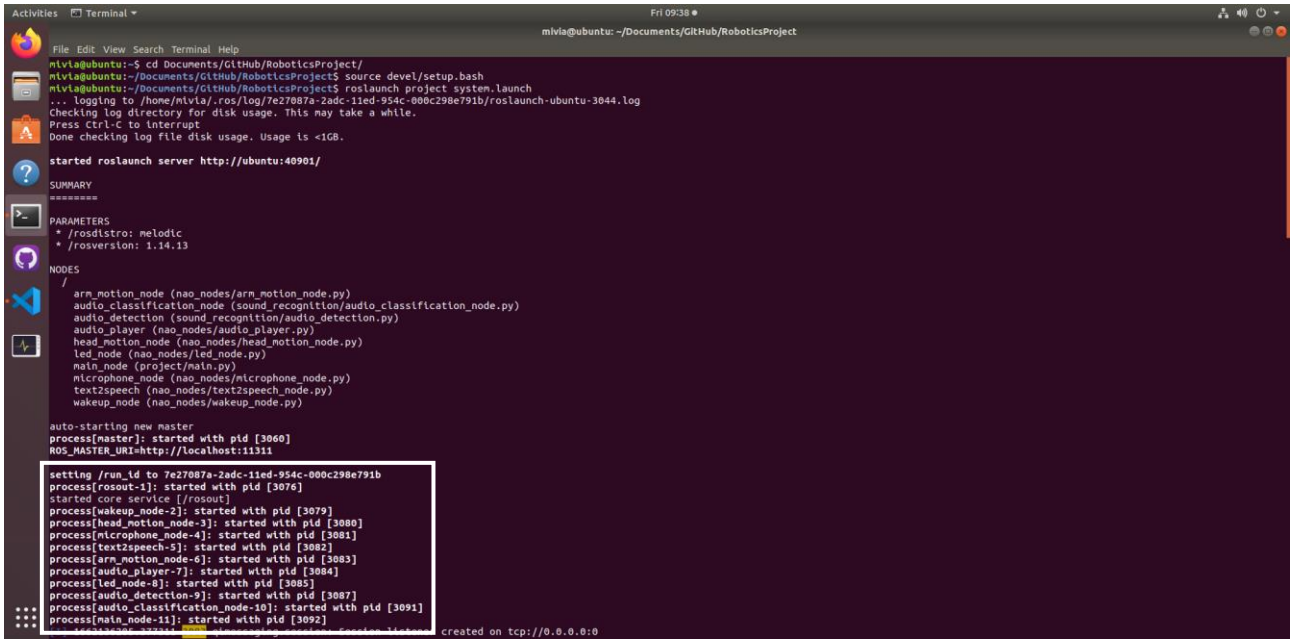
No more tests could be run on the application because of the missing virtualization engines (like microphone or LEDs).

The computer feature was initially implemented as a test feature (though it is fully operating and usable). In fact, to start the application in computer mode the associated parameter is *test*. It was sufficient to install the text to speech engine as described, it also changes the output source for speakers' part, whereas explicitly changes the input source for the microphone part and avoids the execution of other features specific for the robot, like motion or eye color. They are developed as a publisher-subscriber approach, associating no subscribers was enough to suppress the actions. In this way it was possible to record sounds, classify them, listen the text synthesis and test the application to refine its behavior.

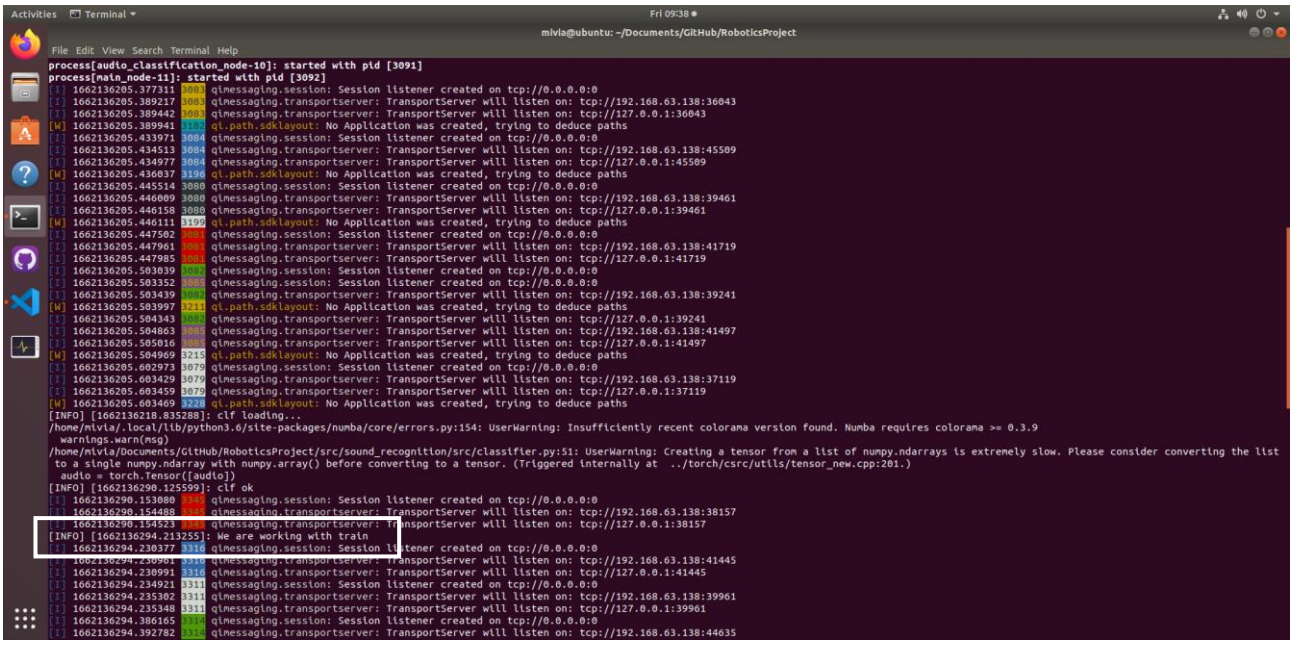
Further tests have been performed on the actual robot, where it was possible to test the differentiation in the color of the eyes, and the responsiveness of the robot during and after listening.

Demo

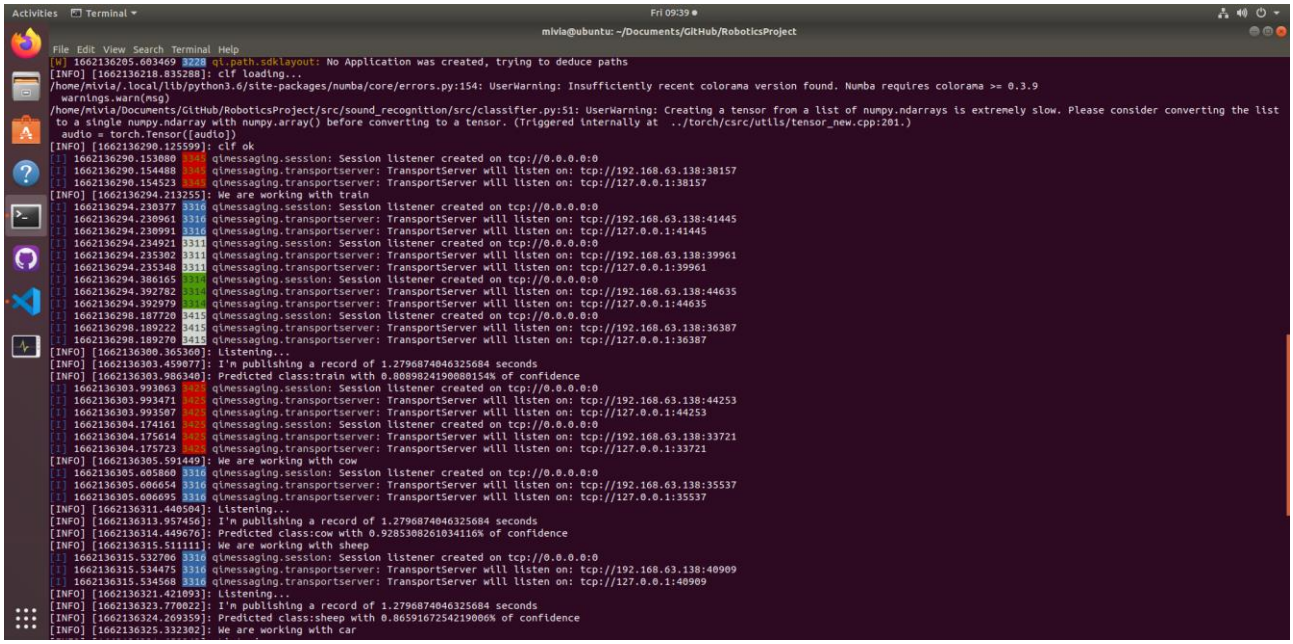
At the end, once all software modules are implemented, running the system.lauch without passing any parameter (all default values), connecting to the robot, the final output is captured on the following screens:



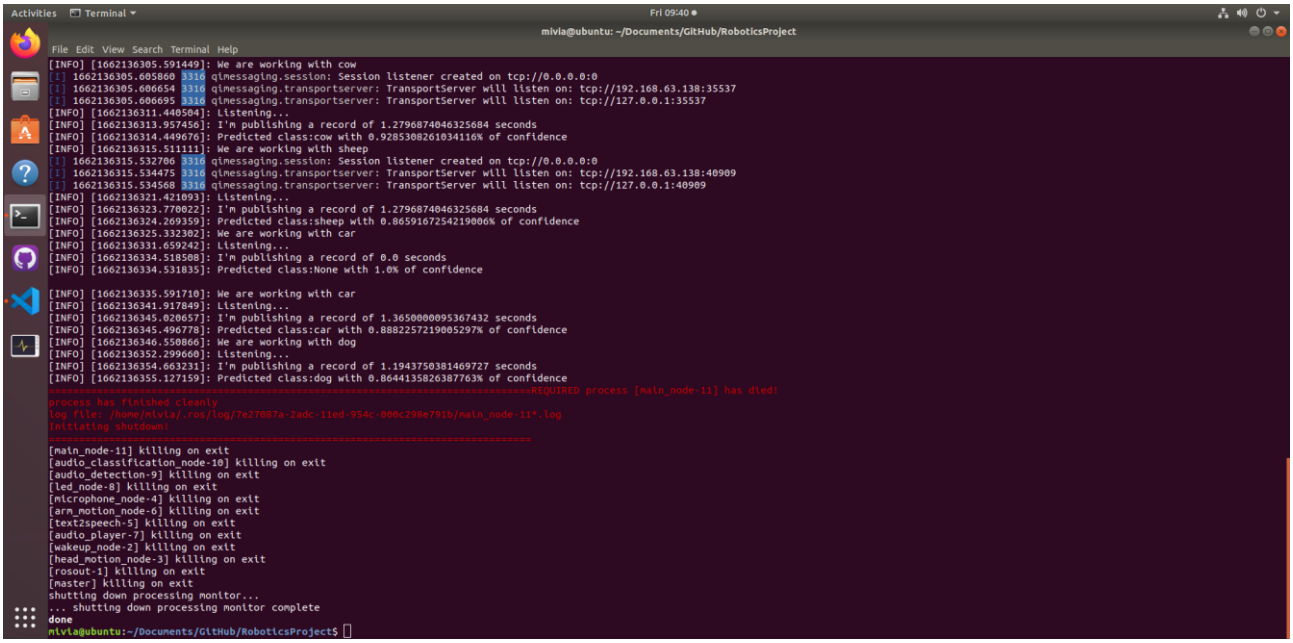
At the beginning all the processes related to the node of the network are instantiated, each one with a specific associated *pid* (process id).



Then, the classifier neural network is loaded, and the main task started. The default order of the objects is: 1) train – 2) cow – 3) sheep – 4) car – 5) dog. “Working with train” informs that the routine for the first object has started.



In the third screen there are all the info send back from the program to give a feedback to the user of the current phase of the exercise and the results of the listening/classification steps.



In the last screen the task is successfully completed and after that, the program stops killing all the active processes related to itself.