UNIVERSITÀ DI PISA

MSc in Computer Engineering

Industrial Applications

Car Driver Drowsiness Detection

TEAM MEMBERS:

Francesco Bruno

Federico Cavedoni

Academic Year 2023-2024

Content

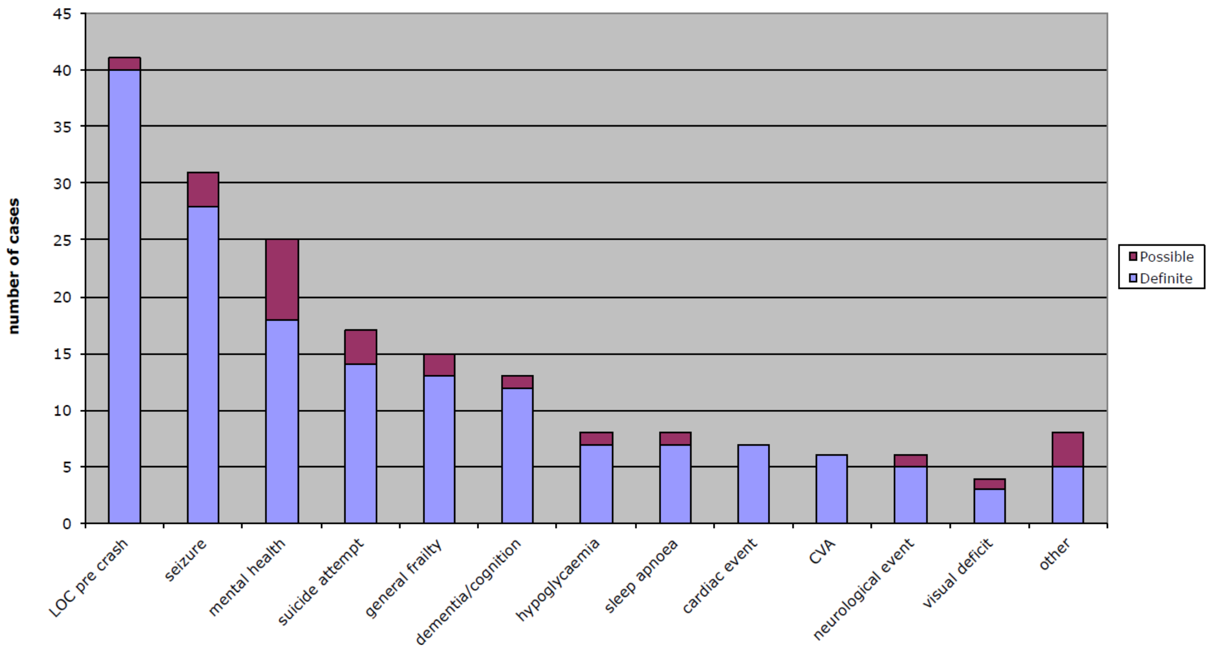
1. Abstract

This document deals with the development of an innovative system capable of monitoring the driver's state of drowsiness and acting promptly in the event of an abnormal condition to avoid unpleasant accidents.

The document covers the complete study that was carried out to develop the system mentioned above, starting from the study of the state of the art of the technology for detecting drowsiness up to the development of the system's software.

2. Introduction

The objective of this project was to create a drowsiness detection system, i.e. a system capable of monitoring the driver while driving, analyzing his vital parameters using specially placed sensors. The system can classify the driver's state of drowsiness and therefore warn him to avoid accidents while driving. Drowsy driving is in fact one of the major causes of road accidents, so this study aims to develop a solution that limits this type of accident as much as possible.



https://www.google.com/url?sa=t&rct=j&q=&esrc=s&source=web&cd=&ved=2ahUKEwjT8sfNxeuCAxUdSfEDHcBeD2AQFnoECBkQAQ&url=https%3A%2F%2Fcasr.adelaide.edu.au%2Fcasrpubfile%2F1151%2FCASRmedicalconditionscrashcausation-1253.pdf&usg=AOvVaw1w0m53NVsb3qGhGR1NHdpB&opi=89978449

The system is composed of one sensor (Polar T34 with receiver), which collects the driver's vital signs, a controller which sends the data to an external server and the server which is responsible for carrying out the classification and sending the result back to the controller, which will carry out certain actions based on the response received.

3. Related Works

3.1 PPG Cognitive Fatigue Prediction

A study that was fundamental for the creation of the prototype is this notebook present on kaggle. This notebook is based on a dataset obtained from 5 students at Stanford University. These people performed a long gaming session and noted the level of tiredness they felt during the session. These guys were wearing a ppg sensor, so they could associate the sensor value with an adequate level of drowsiness. Heart rate variability features were then obtained from the ppg sensors and used for classification.

Source: https://www.kaggle.com/code/katariinaparkja/ppg-cognitive-fatigue-prediction

3.2 ECG-Pi (Thesis title: IoT Device for Electrocardiography Summary Statistics Monitoring)

The aim of this project was to propose and develop a solution to the problems associated with heart monitoring. The focus was on creating a novel solution for electrocardiography to provide preliminary or inital results.
The main consideration was to use low-cost components, such as an ECG sensor, which would allow the device to be used in a patient's home.
From the ECG signal, were extracted the RR-intervals and then its main features (see later).
Source: https://github.com/sam-luby/ECG-Pi

4. System Description

The System works from the interaction of three main components that perform the numerous functions present:

- **ECG Sensor**

  The ECG sensor is used to collect the driver's heartbeat while driving.

  Every time the driver has a heartbeat, the sensor reports it to the controller who can then save the timestamp.

- **Sensor Controller**

  The sensor Controller receives a pulse signal for each heartbeat of the driver and saves the timestamps.

  From these signals it obtains 5 time-related heart rate variability parameters. These parameters are:

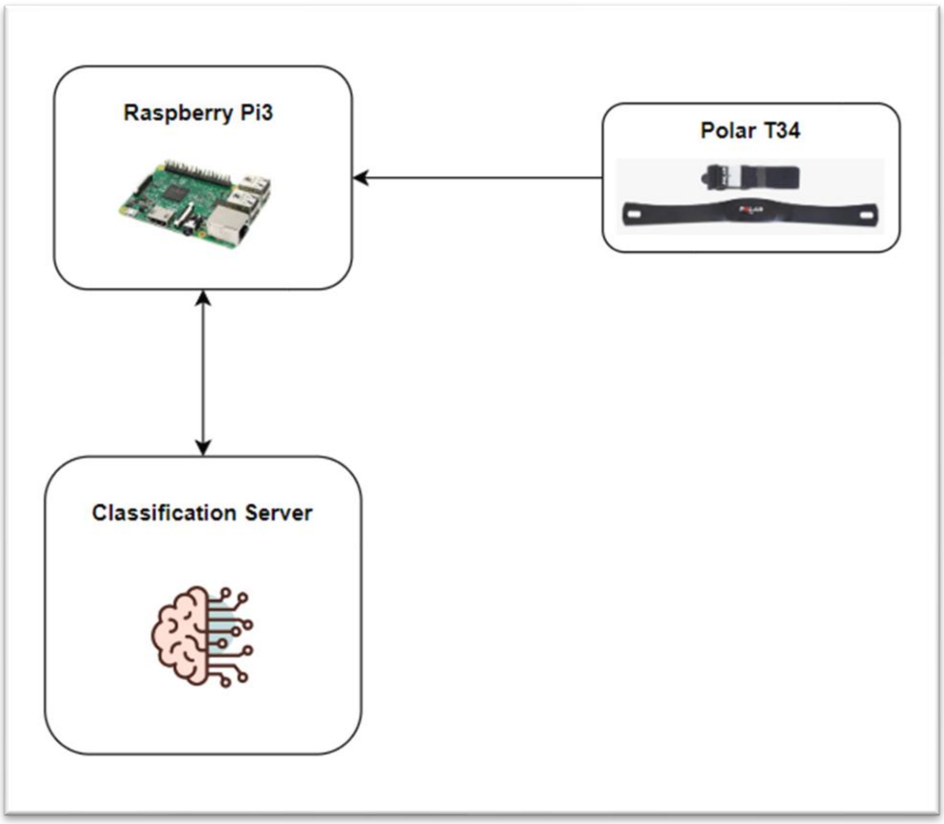
  - **BPM:** Beats per minute
  - **IBI:** Interval between two beats
  - **RMSSD:** Root Mean Square Successive Difference
  - **SDNN:** Standard Deviation of the IBI
  - **PNN50:** Percentage of beats that are more than 50 milliseconds apart.

  These parameters are computed every 60 seconds of sampling.

  Once these parameters have been calculated, the controller sends them to the classification server and waits for the response. When it receives back the response, it sends an alarm signal if a state of drowsiness has been classified.

- **Classification Server**

  The classification Server is responsible for receiving the features from the controller and then classifying them via a previously deployed and trained neural network. Once the prediction has been performed, it sends back the result to the controller and waits for a new array of features.



5. Prototype Description

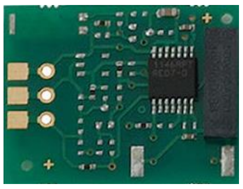
The developed prototype consists of different software modules and different hardware components that cooperate with each other.

5.1 Hardware Description

The hardware components of the prototype are:

- An ECG sensor, the **Polar T34**, capable of collecting the driver's heartbeat via the cardiac impulses collected by the sensor.

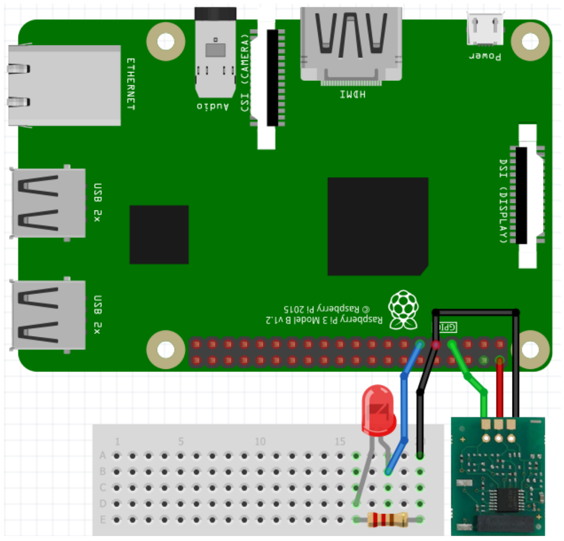
- The Polar Heart Rate Receiver

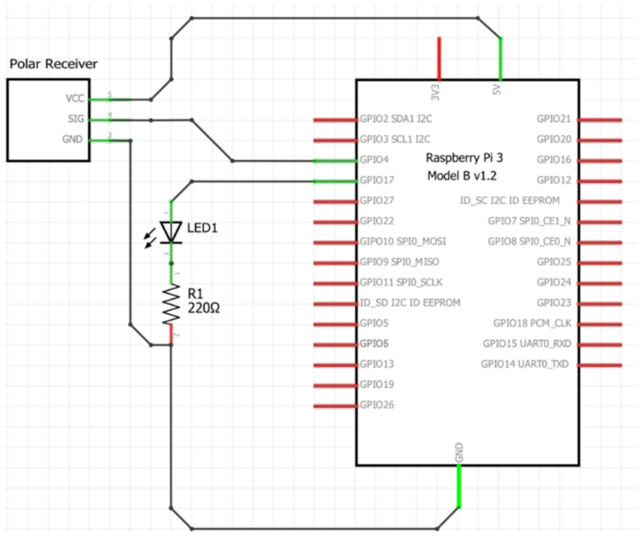
  

  This component is designed to receive heartbeat signals from the Polar ECG sensor. Together, the sensor and receiver provide a low-cost and convenient heart rate monitoring system that can be connected to most any microcontroller. This receiver generates for each heartbeat received, a pulse signal.

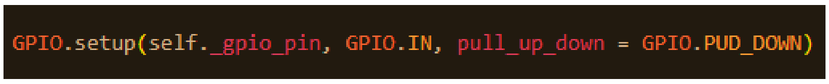
- A controller, the **Raspberry Pi 3**, which receives the signals from the sensor receiver, saves the timestamps, calculates the features to send to the server and, once the classification has been performed and received back, it performs an action based on the resulting classification.

5.2 Circuit Description





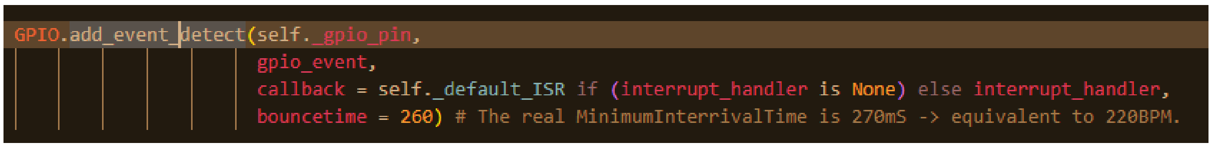
The image shown above is the prototype circuit.
In detail, it consists of the polar receiver connected to the GPIO-4 pin, which through code, we enabled the internal PullDown resistor in order to force the attachment to ground when there is no heartbeat. Look here the last parameter.



```
GPIO.setup(self._gpio_pin, GPIO.IN, pull_up_down = GPIO.PUD_DOWN)
```

In order to give visual feedback, the circuit has a simple red led, which emulates a warning light inside the car dashboard. This led is made to blink in case of Anomaly found in the driver's vital signs and in case the car-driver lifts his hands from the steering wheel.

Moreover, to avoid bounce of the input signal from the receiver, we have exploited the functionality offered by the *add_event_detect* method, which is responsible for binding the GPIO_input_event with the *Callback* function.



```
GPIO.add_event_detect(self._gpio_pin,
                      gpio_event,
                      callback = self._default_ISR if (interrupt_handler is None) else interrupt_handler,
                      bouncetime = 260) # The real MinimumInterrivalTime is 270mS -> equivalent to 220BPM.
```

So, for each two consecutive input rising signal (equivalent to two heartbeats) with an interval lower that 260mS, are automatically ignored.

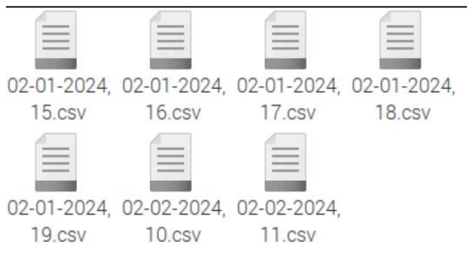
5.3 Controller Software Description

We have chosen to separate the monolithic software controller in a group of classes, each one of them specialized in its task.
The controller prototype software modules are:

- **Communication_API.py**
The module represents a simple interface that both controller and server have to be implemented to communicate in an easy way between them.

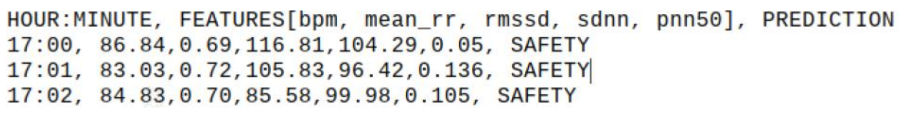
- **Sensor_driver.py**
Sensor driver which is responsible for reporting to the controller the beats that the sensor collects. It provides both the interrupt-mode and the polling-mode to manage the receiver. We have chosen to handle it by using the Interrupt-Mode.

- **Hearth_beat_analysis.py**
Class used for the calculation and management of heart rate variability features. It also provides the functionality to write on permanent memory the history of the vital signs of the car driver.


02-01-2024, 15.csv
02-01-2024, 16.csv
02-01-2024, 17.csv
02-01-2024, 18.csv
02-01-2024, 19.csv
02-02-2024, 10.csv
02-02-2024, 11.csv

In detail, will be created a file .csv indicating in its name the day and the hour of that sampling session in this format “MM-DD-YYYY, HH”.csv. Doing so, will be created a signal vital history like in the above image.

Furthermore, in every file there will be a simple format like in the above image, in order to maintain an ordered and clean history.



```
HOUR:MINUTE, FEATURES[bpm, mean_rr, rmssd, sdnn, pnn50], PREDICTION
17:00, 86.84,0.69,116.81,104.29,0.05, SAFETY
17:01, 83.03,0.72,105.83,96.42,0.136, SAFETY
17:02, 84.83,0.70,85.58,99.98,0.105, SAFETY
```

• **Driver_status**

This is a simple enum class used for the classifier result.

• **Controller_firmware.py**
The controller firmware is the core of the controller-module.
It's in charge of initialize the sensor-driver, configure its interface exposed to the Classifier-Server and then cyclically waits for a new input heart-bit to calculates features and performs actions based on the classification.

5.4 Classifier-Server Software Description

Below follows the code of the Classifier-Server that offers the classification service because the computational power of the raspberry is not enough to run locally (in its memory) the entire classifier.
So we have decided to delegate this task to a simple "classification-service" that is waiting to receive a feature array, then classify it and at the end sends back the classification info.

• **Communication_API.py**
The class provides, as mentioned above, basic functionalities to send and handle the incoming http-messages.

• **Project_network.py**
This py-script must be used to deploy the Neural Network used for feature classification, implemented in the classification server.

- **Neural_network.py**
  The module is our abstraction of the pre-trained neural network.
  It offers the classification functionality and to generate a correct input for the classifier.

- **Thread_receiver.py**
  The thread_receiver is the main class that coordinates all the other.
  In detail, this class, initialize the communication interface,
  loads if exists the classifier model (else it deploy the classifier), and waits for an incoming classification-request.

5.5 Data Source

For the neural network training, a dataset belonging to the '**PPG Cognitive Fatigue Prediction**' study was used in which 5 students from the Stanford University put on a ppg sensor and had a long gaming session. They also noted their level of tiredness during this session, so they could classify the data and train the network correctly.

6. Performance Evaluation

6.1 Real Time Contraints

Below, we present our study regarding the RealTime constraints of the sampling task.

We decided to manage sampling through the use of interrupts, to avoid polling.

Unlike the STM32, Arduino, or any board that allow you "Bare-Metal" programming, with Raspberry the interrupt management is done in software because the hardware-interrupt is intercepted first by the Linux Kernel. Consequently, every time the heart-rate-receiver sends an impulse on a GPIO-pin(4), the same thread will always be put into execution (let's call it "thread-ISR") which will execute our “Interrupt-Service-Routine”.

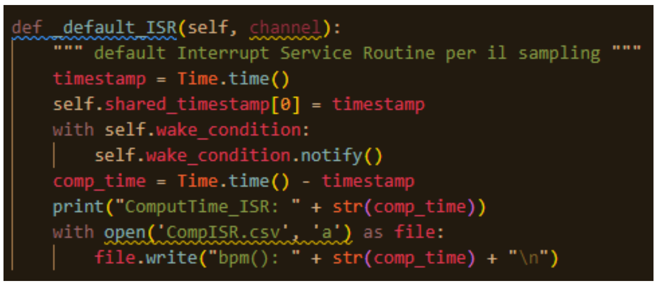
The latter will store the timestamp of this impulse in a variable shared with the main thread.

According to our study regarding “task-ISR”, we can define it as **Firm-Task**, because: if between two consecutive pulses, the "ISR-thread" in the first pulse has not stored the relevant timestamp in the shared variable, when the second pulse arrives, the timestamp relating to the first will be overwritten and therefore lost.
From our point of view, this is equivalent to invalidating the session and therefore for that minute it is as if the car-driver had not been monitored.
Therefore, if the deadline miss occurs sporadically, our system continues, with a **small performance degradation**, to monitor the car-driver.

6.2 Deadline Analysis

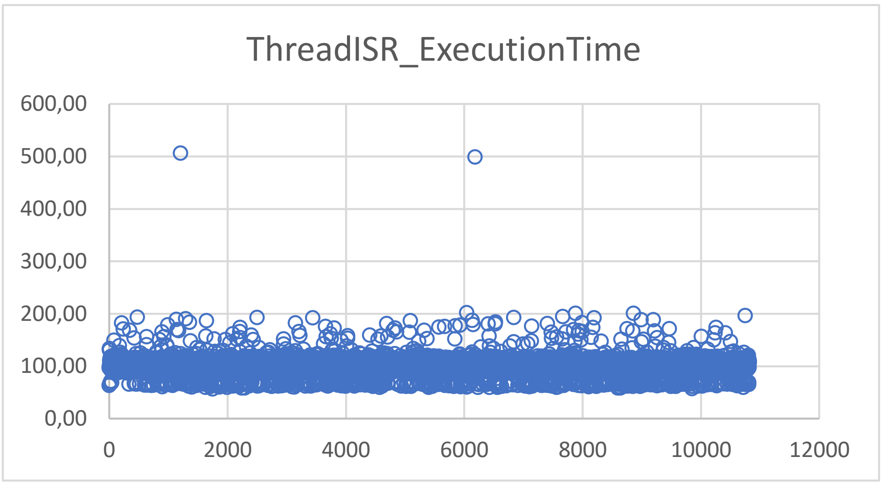
First ComputationTime study was performed on the duration of the taskISR, in order to somehow give an estimate on its WorstCaseExecutionTime.

Recalling that the input pulse to the raspberry has a minimum interarrival time of about 272mS, to be sure not to miss the deadline, the taskISR must finish within (in the ideal case) 272mS from the arrival of the pulse.



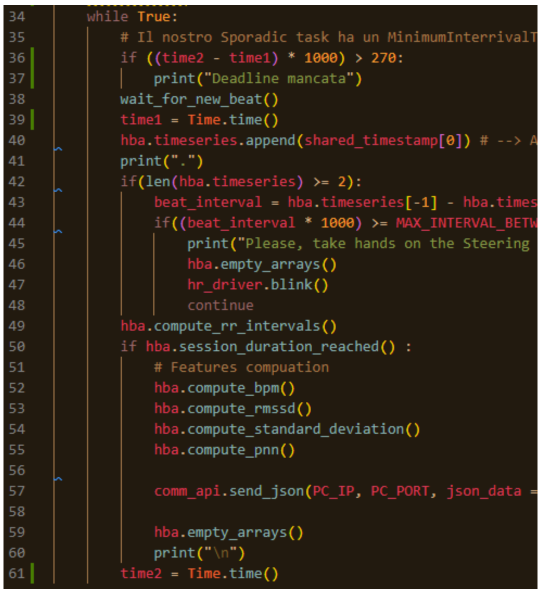
```
def _default_ISR(self, channel):
    """ default Interrupt Service Routine per il sampling """
    timestamp = Time.time()
    self.shared_timestamp[0] = timestamp
    with self.wake_condition:
        self.wake_condition.notify()
    comp_time = Time.time() - timestamp
    print("ComputTime_ISR: " + str(comp_time))
    with open('CompISR.csv', 'a') as file:
        file.write("bpm(): " + str(comp_time) + "\n")
```

Measuring the maximum ComputationTime for about 4 hours, we obtained a *WorstCaseExecutionTime* of about 510uS. This value is the result of a Test-based measurement technique, so this guarantees us a high percentage of meeting the deadline, but not the 100% exact. See below the scatter plot of the execution time of the task-ISR.



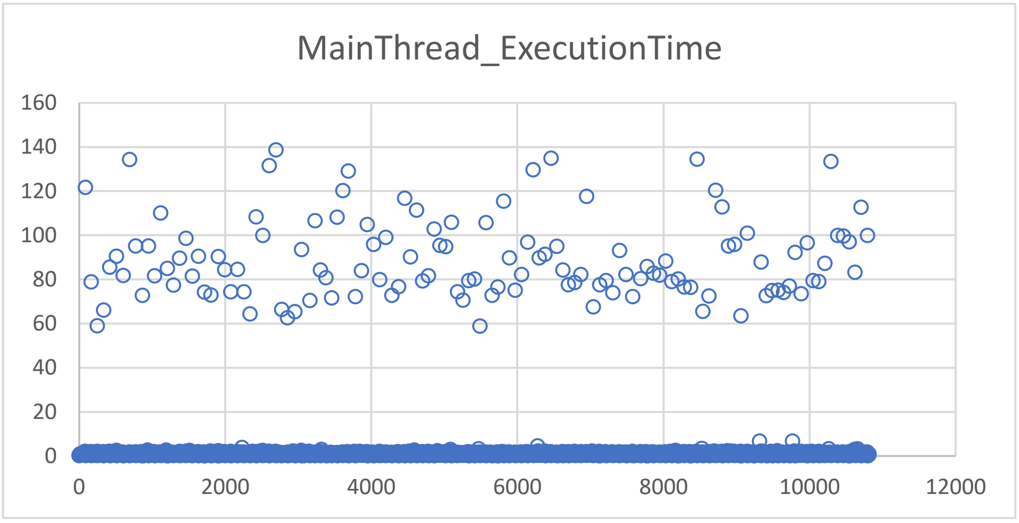
The second study of the ComputationTime was performed on the "MainThread," i.e., the thread that waits to receive a signal from the taskISR. The reason for such a study is to understand whether that thread finishes the run, before the arrival of the next pulse (always assuming the maximum bpm).

With the simple calculation in *line 36*, we check that the previous lap did not last longer than the minimum interarrival time.



```
34      while True:
35          # Il nostro Sporadic task ha un MinimumInterrivalT
36          if ((time2 - time1) * 1000) > 270:
37              print("Deadline mancata")
38          wait_for_new_beat()
39          time1 = Time.time()
40          hba.timeseries.append(shared_timestamp[0]) # --> A
41          print(".")
42          if(len(hba.timeseries) >= 2):
43              beat_interval = hba.timeseries[-1] - hba.times
44              if((beat_interval * 1000) >= MAX_INTERVAL_BETW
45                  print("Please, take hands on the Steering 
46                  hba.empty_arrays()
47                  hr_driver.blink()
48                  continue
49          hba.compute_rr_intervals()
50          if hba.session_duration_reached() :
51              # Features compuation
52              hba.compute_bpm()
53              hba.compute_rmssd()
54              hba.compute_standard_deviation()
55              hba.compute_pnn()
56
57              comm_api.send_json(PC_IP, PC_PORT, json_data =
58
59              hba.empty_arrays()
60              print("\n")
61          time2 = Time.time()
```

By making an estimate on the maximum execution time over about 4 hours sampling session, we obtained that the WorstCaseExecutionTime is more or less 140mS, quite below the 272mS threshold. See here the next page the scatter plot.

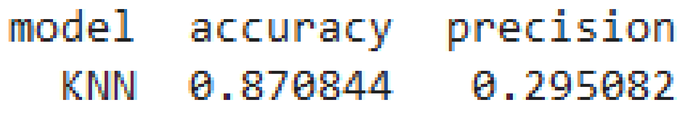


We remember you that, to have 100% certainty on the WCETs of the taskISR and MainThread, special tools should be used, because the prototype was built on a system with cache, which we know introduces Impredictability for ComputationTime calculation (or if it is possible, you can disable the cache, then do your calculation and then re-enable the cache).
However, the two computed values give us a good approximation of the WCET.

6.3 Neural Network Evaluation

As regards the performance of the network, we chose a 'KNeighborsClassifier' type neural network which, during the testing phase, had excellent performances.

Among the networks we found, we chose this one because it is the one with the highest precision, with an excellent accuracy and therefore excellent for our use case.



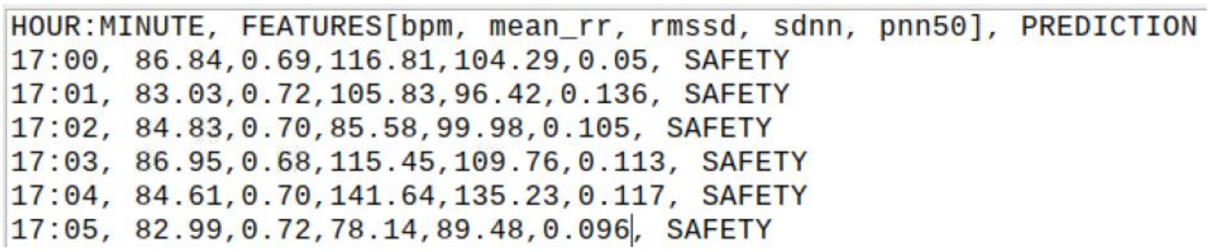
```
model   accuracy   precision
  KNN   0.870844    0.295082
```

7. Demo

Here is shown is a photo of the band used in the prototype.

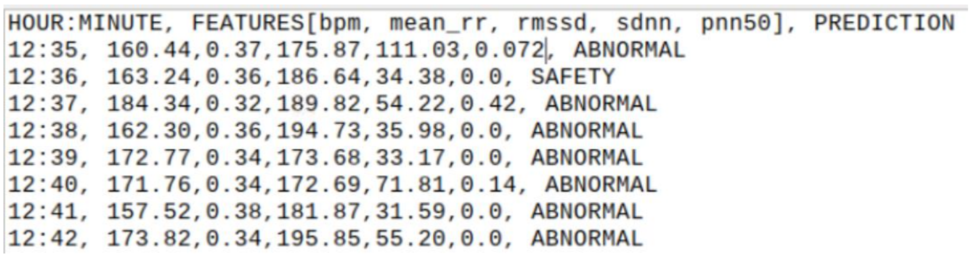
The demo is divided into 3 trials:

- First was carried out in a seated position to have a situation similar to that of a driver in a normal condition (SAFETY).
  We show below a screen of the file generated by the controller during this demo.



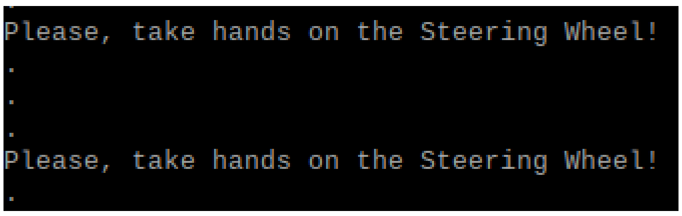
```
HOUR:MINUTE, FEATURES[bpm, mean_rr, rmssd, sdnn, pnn50], PREDICTION
17:00, 86.84,0.69,116.81,104.29,0.05, SAFETY
17:01, 83.03,0.72,105.83,96.42,0.136, SAFETY
17:02, 84.83,0.70,85.58,99.98,0.105, SAFETY
17:03, 86.95,0.68,115.45,109.76,0.113, SAFETY
17:04, 84.61,0.70,141.64,135.23,0.117, SAFETY
17:05, 82.99,0.72,78.14,89.48,0.096, SAFETY
```

- The second demo was carried out after a short training session in order to get the beats up and thus trying to re-create a tachycardia (and thus abnormal) situation for a car driver. We can see that the network correctly classifies almost all monitoring sessions.
  For each ABNORMAL classification received, a red LED was blinked.



```
HOUR:MINUTE, FEATURES[bpm, mean_rr, rmssd, sdnn, pnn50], PREDICTION
12:35, 160.44,0.37,175.87,111.03,0.072, ABNORMAL
12:36, 163.24,0.36,186.64,34.38,0.0, SAFETY
12:37, 184.34,0.32,189.82,54.22,0.42, ABNORMAL
12:38, 162.30,0.36,194.73,35.98,0.0, ABNORMAL
12:39, 172.77,0.34,173.68,33.17,0.0, ABNORMAL
12:40, 171.76,0.34,172.69,71.81,0.14, ABNORMAL
12:41, 157.52,0.38,181.87,31.59,0.0, ABNORMAL
12:42, 173.82,0.34,195.85,55.20,0.0, ABNORMAL
```

- The third consists in loss of samples by emulating the situation in which the user takes his hands off the steering wheel.
  We took advantage of the two band's electrodes to cause the loss of beats and with a simple check in the software,
  we verify that the maximum interval between two consecutive beats never exceeds the value stored in *MAX_INTERVAL_BETWEEN_BEATS* (1.2seconds), which is equivalent to a BPM of 50.
  In this case, in addition to the blinking of the red LED, the message below is shown to warn the driver.

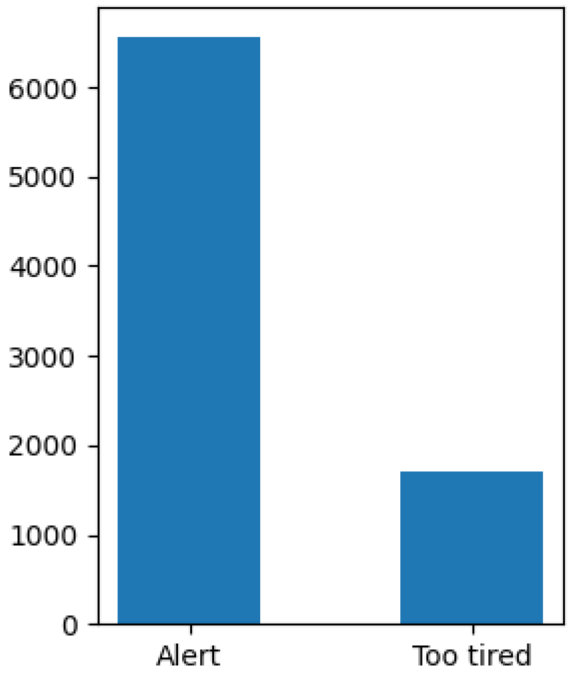


```
Please, take hands on the Steering Wheel!
.
.
.
Please, take hands on the Steering Wheel!
.
```

8. Conclusions

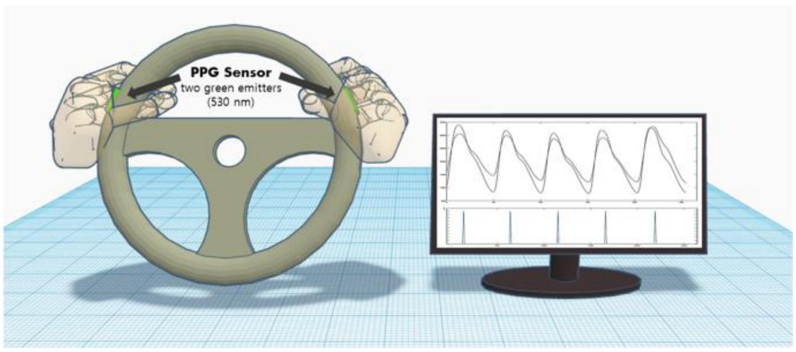
In conclusion, the prototype is correctly able to monitor the driver's vital signs and respond promptly if the driver is no longer able to drive due to severe drowsiness.

However, we noticed that the system is unable to correctly classify some inputs, and this is due to the fact that the training is not exhaustive and balanced enough. By analyzing the data, we can in fact see that the dataset used for training is highly unbalanced and this certainly had consequences in the final classification. A possible upgrade could therefore be to do a better training, which could result in better classification.



Furthermore, an improvement that would make sense to do is on the sensor, since the polar T43 can only sample the arrival of a heartbeats to the controller, but no other unit of measurement. So, with that sensor, the precision degree was at timestamps level. A good future improvement could therefore be to use a Photoplethysmographic Sensor (PPG), which is capable of extract the blood volume changes, therefore, can give us a better precision degree with the consequence to perform a more accurate classification.

Below, is shown a picture in which there is a possible position of the ppg sensor in the steering wheel.



Another improvement may be the change of the behavior when Abnormal vital signs were detected. For example, in that situation, one might consider decreasing the session interval from 60s to 30/40 seconds to monitor the driver more frequently, thereby trying to understand the causes of that Abnormal event.

9. References

PPG Cognitive Fatigue Prediction Source:
https://www.kaggle.com/code/katariinaparkja/ppg-cognitive-fatigue-prediction