Hes·so
Haute Ecole Spécialisée
de Suisse occidentale
University of Applied Sciences and Arts
Western Switzerland

Medina: sMart EDge fabric for Iot Applications

30235.1 IP-ICT
Enabling Sciences / Information and communication technologies (ICT)

Interim report

WP3, Validating InaaS architecture with the smart lighting use-case

WP4, Validating InaaS architecture with the IoT data integrity use-case (Trust Noise)

Authors

Pablo Strasser, Alexandros Kalousis (HES-GE/HEG), WP3
Ludovic Gindre, Marco Emilio Poleggi, Nabil Abdennadher, Tewfiq El Maliki (HES-GE/HEPIA), WP4

Project start date:	July 1st, 2019	Project duration:	24 months
Document identifier:	Medina/2020/M3.1	Version:	v0.5
Date due:	30 November 2020	Status:	draft
Submission date:	20 November 2020	Distribution:	Restricted

Medina Consortium

This document is part of a collaborative research project funded by the Innosuisse, grant number 30235.1 IP-ICT. The following partners are involved in the project:

HES-SO Genève Project Manager
4 Rue Prairie 4
1202, Geneva - Switzerland
Contact person: Nabil Abdennadher
E-mail: nabil.abdennadher@hesge.ch

SixSq
Rue du Bois-du-Lan 8
1217, Meyrin, Switzerland
Contact person: Marc-Elian Bégin
E-mail: meb@sixsq.com

Smart Geneva
Canton de Genève, Département de la sécurité
Direction générale du dévélopement économique, de la recherche et de l'innovation
Quai Ernest-Ansermet 18bis
1211, Geneva 3, Switzerland
Contact person: Nicolas Bongard
E-mail: nicolas.bongard@etat.ge.ch

Services Industriels de Genève (SIG)
Chemin de château Bloch 2
1214, le Lignon, Switzerland
Contact person: Olivier Gudet
E-mail: olivier.gudet@sig-ge.ch

Service de l'air, du bruit et des rayonnements non ionisants (SABRA)
State of Geneva
Avenue Saint-Clotilde 23
1211, Geneva, Switzerland
Contact person: Philippe Royer
E-mail: Philippe.Royer@etat.ge.ch

Contents

WP2: Design InaaS for cloud to edge application deployment

The objective of WP2 is to design a continuous integration platform for IoT self-adaptive Machine Learning based applications composed of a set of edge and cloud modules. This platform is called: Intelligence as a Service (InaaS). Task 2.2 is fully completed. Task 3.2 has been delayed due to delays of WP3, WP4 and WP5. At the demand of the implementation partner (SixSq), Task 2.1 has been extended to support two additional functionalities which are not planned in the initial proposal : "edge security" and service discovery. The results of this WP will not be detailed in this report. They will be the purpose of a separate report.

WP3: Validating InaaS architecture with the smart lighting use-case

2.1 Introduction

In this progress report we present the work that has been done within WP3, *Validating InaaS architecture with the smart lighting use-case* by the Data Mining and Machine Learning team[1] at HEG/HES-SO, Geneva. The goal of this WP is to develop machine learning models that can be deployed in resource constrained environments for traffic monitoring in order to automatically adjust the lighting conditions based on the trafic load. Our goal is to produce a light-weight machine learning-based system that can count cars in real time from video feeds produced by commodity cameras. The main challenge lies on the fact that all detections should be done locally on the edge with financial and computational constraints on the computational device as well as on the camera.

We chose to decompose the problem on the following set of simpler tasks:

Car detection: establish the bounding boxes of the cars that appear within a single video frame.

Car tracking: assign a unique identifier to each individual car as it appears and moves over the different video frames.

Lane detection: establish the position and direction of the traffic lanes over which the car counting takes place.

Car counting: establish the number of cars that pass over each one of the detected lanes.

In figure 2.1 we give an example of the final result produced by the different modules. The red bounding boxes are generated by the **Car detection** module, the id of the car (car 567 in our example) is generated by the **Car tracking** module. The positions of the green lines are generated by the **Lane detection** module and finally the car count (given next to the green line in the form of $label = count$) is generated by the **Car counting** module.

2.2 Car detection

Our objective in this module is to detect cars in an individual frame of the video. For that, we need to train a model using a large collection of annotated images containing cars and other objects, on the order of tens of thousands of images. In order to avoid to labor intensive task of image collection and generation we instead use a model that has been pretrained on an existing dataset, COCO. The pretrained model that we selected places bounding boxes around objects together with the label of the object. The pretrained model is able to detect 90 different classes such as cars, persons, bicycles, trucks and buses. In figure 2.2 we show two images containing cars from the COCO dataset; overall there are 12786 images containing cars in the COCO dataset.

In table 2.1 we give the performance reported in the literature on the COCO dataset of different pretrained models. For the needs of Medina, we want a fast inference model (Speed column) with high mean average precision (mAP column). We choose to use **ssdlite_mobilenet_v2_coco** which with an inference speed of 27 ms per image is one of the fastest model with a very good mAP of 22. Models with higher accuracy are more than twice slower making them unusable in a settings with tight computational budget.

[1] dmml.ch

Figure 2.1: Image of the final product, showing detection, crossing lines and count.

Figure 2.2: Two instances of the COCO dataset.

Let shortly discuss **ssd**[1] which is a central component of all the fast model in table 2.1. **ssd** learns to place bounding boxes around different types of detected objects. It does so by extracting multiple feature maps from the image using a base model (**mobilenet v2** in our case) at different resolutions. Theses features are considered as block of features each describing the content in part of the image, similarly as done in regular convolutional networks. Using theses features **ssd** first select using multiple classifier each specialized for a single class of object, aspect ratio and scale at every block of features. Each classifier correspond to a predefined bounding box centered on the block of feature. The target label of the classifier is determined if the overlap of the predefined bounding box overlap enought the ground truth bounding box. Each different scale of feature map is used to predict predefined bounding box of the same scale. The position and sizes of each of the detected predefined bounding box is then fine tuned by predicting a diplacement and scale modification. We describe the architecture in details in figure 2.3 and its effect in an image in figure 2.4.

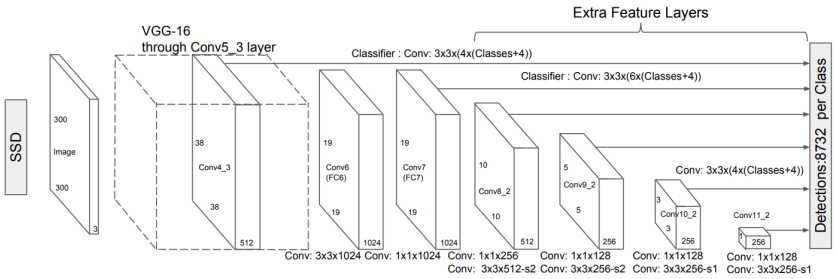
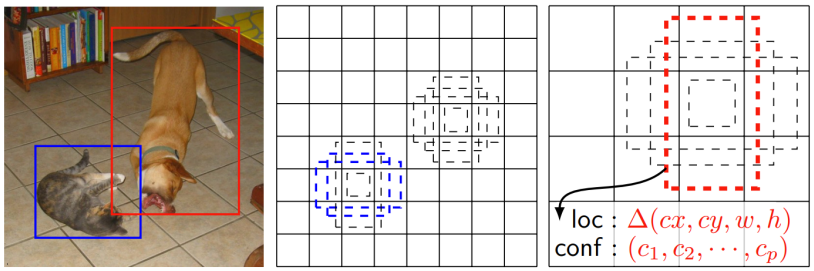


Figure 2.3: Architecture of **SSD**[1] extracted from the original paper. In this figure the base model is **VGG-16** and 6 different resolutions are used for detection. Each of theses features map will be used to choose a good base bounding box together with fine tuning its position and scale.



(a) Image with GT boxes (b) 8×8 feature map (c) 4×4 feature map

Figure 2.4: Shema extracted from **SSD**[1] original paper. Figure a) is the original image with two objects red and blue. In figure b) and c) we see some of the predefined bounding boxes together with the bounding boxes that match the target. Figure b) correspond of one scale of features maps and bounding boxes and is only able to match the cat, whereas figure c) is another scale only able to match the dog. Note also that in this example two bounding boxes are found for the cat (in blue). The step of non-maximum suppression(**nms**) will remove one of the two bounding boxes giving the final result.

Because multiple bounding boxes may be detected for the same object an additional step called non-maximum suppression(**nms**) is used. **nms** remove spurtious bounding boxes based on the overlap with other detected objects.

One of the central parts of **ssd** is the convolution architecture that is used to extract features from images. This can be a computational demanding operation especially for the resource constrained devices that we consider in this project. **mobilenet v2**[2] is a speed improved variant of standard convolutional model used in computer vision where convolution are replaced with depthwise separable convolutions which is a cheaper version of convolution. **ssdlite** is **ssd** using the speed improvement of convolution that **mobilenet v2** bring to **ssd**. So, the model that we use **ssdlite mobilenet v2** apply the faster version of convolution throught the whole pipeline.

As we already mentioned we used a pretrained **ssdlite mobilenet v2** model on COCO. The pretrained model has a very good detection rate of cars in daylight videos. However it was failling when it had to operate on night videos for the simple reason that the original dataset did not contain training examples of cars in night scenes.

To resolve the problem of car detection in night scenes, which is actually the main target of WP3, we had to generate an additional annotated dataset contaning night images of cars. To annotate additional images, we used a simple labeling app and annotated 215 day and night images. In figure 2.5 we show an screenshot of the labeling app we used.

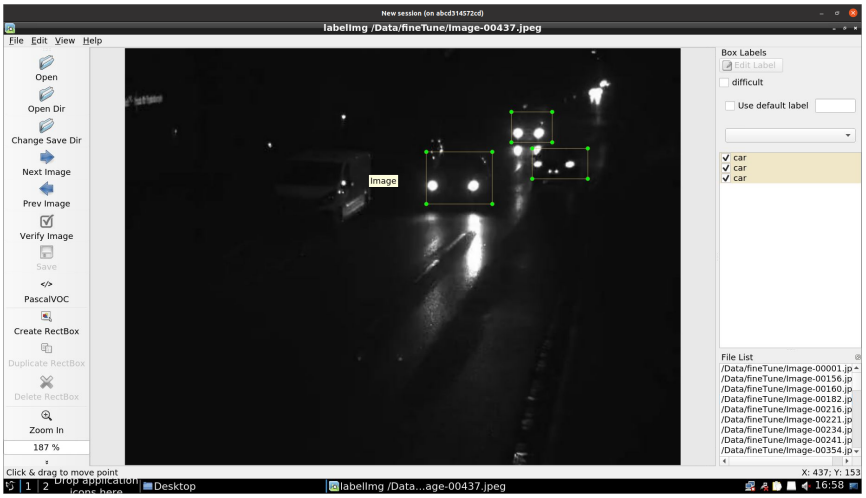


Figure 2.5: Windows of the labeling tool used to manually label images. This app is a packaged version of labelImg(https://github.com/qaprosoft/labelImg).

Once the new dataset was available we tuned the pretrained **ssdlite mobilenet v2** model on this additional dataset. We did so by updating all parameters of the model on the new dataset for less than an hour. The resulting model did not have the problem during the night.

As the **Car detection** module is the most computational expensive module of the whole pipeline, we evaluate its performance on various devices. The two devices our project were interessted on are the Jetson Nano and the Raspberry Pi. In table 2.2 we see that the performance of the Raspberry Pi is a lot lower than the Jetson Nano and is insuficiant for our use case. As having less that 2 frame per second mean that in a fluid trafic on a fast lane, we may have only a few frame capturing the car. For the rest of this project, we decided to drop the Raspberry Pi and concentrate only on the Jetson Nano.

2.3 Car tracking and counting

To track and count cars, we use the DeepStream framework provided by Nvidia . This framework is usable on all Nvidia GPUs and devices including the Jetson Nano.

The DeepStream framework is compatible with the pretrained model we use. This facilitate the integration of our detection model into the DeepStream framework.

A basic tracking algorithm is given by the Intersection over Union (IOU) which computes the overlap of bounding boxes. This approach however can show its limit in case of overlapping objects as it may merge together object that overlap considerably. To address this limitaion we use an improved version named Nv-adapted Discriminative Correlation Filter NvDCF implemented in DeepStream. **NvDCF** in addition to the overlap also uses visual feature correlation resulting in excellent car tracking performance.

For car counting, we use the nvdsanalytics module of DeepStream. This module allows to define a set of virtual lines and directions. It will then count every object passing throught the line following the given direction. We use virtual lines for two reasons. First, they correspond directly to lanes facilitating the debuging of the **lane detection** module. Second, they avoid counting multiple time the same car if the car is miss tracked for a short time. Correct tracking is only needed just before and just after crossing the virtual line. All the difficulty lies on providing the optimal virtual lines. This is achieved by the lane counting module (cf. 2.4).

Model	Speed (ms)	COCO mAP
ssdlite_mobilenet_v2_coco	27	22
ssd_mobilenet_v1_coco	30	21
ssd_mobilenet_v1_0.75_depth_coco	26	18
ssd_mobilenet_v1_quantized_coco	29	18
ssd_mobilenet_v1_0.75_depth_quantized_coco	29	16
ssd_mobilenet_v1_ppn_coco	26	20
ssd_mobilenet_v1_fpn_coco	56	32
ssd_resnet_50_fpn_coco	76	35
ssd_mobilenet_v2_coco	31	22
ssd_mobilenet_v2_quantized_coco	29	22
ssd_inception_v2_coco	42	24
faster_rcnn_inception_v2_coco	58	28
faster_rcnn_resnet50_coco	89	30
rfcn_resnet101_coco	92	30
faster_rcnn_resnet101_coco	106	32
faster_rcnn_inception_resnet_v2_atrous_coco	620	37
faster_rcnn_nas	1833	43

Table 2.1: Inference time and predictive for different pretrained model on object recognition. The table uses data from TensorFlow 1 Detection Model Zoo. The speed is given for a specific hardware with GPU, the speed without GPU may be different however this gives an order of magnitude of the inference time of the different models. COCO mAP is the average mean Precision, this is an average over different detection tresholds of the mean on the precision recall curve, the higher the more accurate the model is. In bold we mark the model we use.

Device	FPS
Workstation (CPU)	55
Workstation (GPU)	188
Jetson Nano (GPU)	24
Raspberry Pi (CPU)	2

Table 2.2: Computational performance for **ssdlite mobilenet v2** on different devices. FPS is the number of frame per second.

2.4 Lane detection

In order to correctly count cars, we need to establish the positions and directions of the virtual lines which are then used in the **car counting** module. These virtual lines will essentially correspond to the car lanes. We will assume that the direction of the car lanes are orthogonal to the virtual lines. To establish the lane position information we did not follow the classical approach in lane detection which detects the boundaries of the lanes directly. Instead we relied on the car movement and car trajectories as these are extracted from the traffic videos which we use in order to extract the car lanes. The lane detection algorithm seeks to place virtual lines such that every trajectories cross at least one virtual line in the same direction The number of virtual lines is increased such that almost every trajectories cross at least one virtual line.

Our lane detection algorithm relies on an error measure that evaluates how close the virtual lines are with respect to the car trajectories such that they will trigger the counting algorithm. The error measure is a differentiable function which seek to push the virtual lines such that at least one virtual line is triggered. To be triggered, the car's trajectory need to cross the virtual line in a similar direction (there is a 30 degree tolerance). Thus we define our error measure with two terms. The first term try to correct the direction of near virtual lines whereas the second term try to correct the position of the virtual lines.

Note the lane detection algorithm need to be executed only once for a given camera with a short video segment.

To explain the error measure let us consider figure 2.6 which is an annotated version describing the different computations involved in order to be able to count cars. In red, we show the trajectory of the car and the arrow indicates the direction of the car. The two green segments, $c-0$, $c-1$, show two virtual lines. The green arrows associated with the green segments correspond to their respective directions represented by their angles. The lengths of the two blue segments give the distances of the car's trajectory from the virtual lines c_0 and c_1, we will denote these two distances by d_0 and d_1. We define that distance as the minimal distance of the trajectory from the virtual line. As we see the distance with respect to virtual line $c-0$ is zero since the trajectory of the car crosses that virtual line. On the contrary, the trajectory of the car does not cross the virtual line $c-1$ and thus the respective distance is larger. In addition we will also define two angular distances, a_0, a_1, which will correspond to the angle between the direction of the car and the two virtual lines, c_0, c_1 at the positions of the car that we computed d_0 and d_1 (i.e. the points where the red line and the blue segments intersect). Using these two distances we define two respective combined distance as $b_i = d_i + a_i, i = 0, 1$.

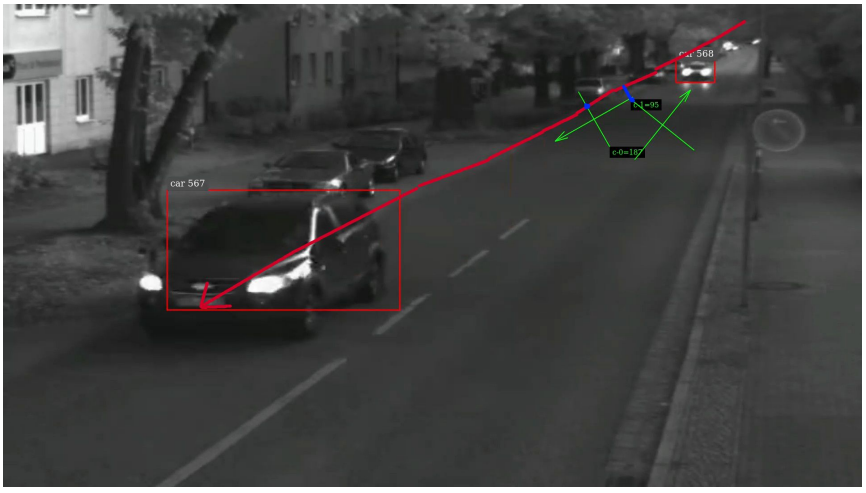


Figure 2.6: Image of the final product, showing detection, crossing lines and count. In red, we show the trajectory of a car and in blue show the distance to the two virtuals lines. The green lines are two virtual lines $c-0$ and $c-1$ together with their directions.

We will now give a more detailed description of the error measure that our clustering algorithm uses. As we said its first term measures how good the directions of the virtual lines are. It is given by:

$$e_a = w_0 a_0 + w_1 a_1 \tag{2.1}$$

Where w_0, w_1, are the two component of the similarity measure w indicating the soft assignment of the car's trajectory to the two virtual lines $c-0$ or $c-1$. This soft assignment is computed by creating a similarity

measure from the combined distances b_1, b_2. We compute this similarity measure with the $\mathrm{softmin}$ function. In our example, w_0 is almost one and w_1 is almost zero. As a result the first term of the error ends up beeing a_0 the difference in direction with respect to cluster $c-0$. The maximum value of the angle error is 2π since we express angles in radians. The full expression of the first term is then given in vectorized form as:

$$e_a = \mathrm{softmin}(b) \odot a \tag{2.2}$$

Where $\odot$ is an element wise product. To be counted, a car trajectory need to have a value of a smaller than 30 degree.

The second term measure how good the positions of the virtual lines are:

$$e_d = \mathrm{sigmoid}(\min\{d_i | i = 0, 1\}) \tag{2.3}$$

Where the $\min$ operation hard select the shortest distance in position. We use the $\mathrm{sigmoid}$ operation to ensure that outliers do not dominate the error and ensure that the error is bounded (maximal possible value is 1). To be counted, a car trajectory need to cross at least one virtual line ($e_d = 0$).

The final error that we use to establish the virtual lines is:

$$e = e_a + e_d = \mathrm{softmin}(a + d) \odot a + \mathrm{sigmoid}(\min(d)) \tag{2.4}$$

We choose in our error measure to minimize quantities based on the distance and not the distance squared because the former is more robust to outlier. Because error bigger than a given threshold results in a car being missed, it is better to have low error in a majority of cases even at the cost of some huge outliers than having bigger errors everywhere but risking to miss every car. This is a similar situation of the difference between the median and the mean for a bimodal distribution. The median is near of one of the modes, whereas the mean is in between them. In our car counting scenario, the median will count all the car of one mode and miss the other whereas the mean will miss all of them.

To minimize this error, we first greedily create a good initial set of virtual lines. We do so by first finding a single virtual line ($c-0$ in our example). We find this single optimal virtual line by computing the error of a list of candidate virtual lines constructed from the car's trajectories. We select the virtual line from the candidate virtual line having the lowest error. We then add an additional virtual line by computing the error of the same list of candidate virtual lines and selecting the virtual line with the lowest error. We repeat the process to obtain the fixed number of virtual lines needed.

The list of candidate virtual lines is obtained by taking every frame of every car trajectories and create a virtual line by creating a line orthogonal to the car direction with a fixed width taken from a range of predefined value. Currenty the grid of values for the width is taken from 1 to 100 pixel with a 10 pixel increment.

We finally fine tune the virtual lines by backpropagation. By experience, the first greedy solution already give a very good solution.

2.5 Computational Performance

Our final solution is based on Nvidia DeepStream framework which is available on all devices having an Nvidia graphical card, including the Jetson Nano. As discussed in section 2.2 the Raspberry Pi 4 did not have a good enough detection performance and is unsupported in the current solution configuration. We believe that to accurately track a car in all situations a minimum of 5 to 10 frames per second is necessary.

To ensure that we have the computational power to solve the problem, we benchmark our final solution as a docker image on a variety of systems. All systems are x86 platform except the Jetson Nano (in bold). The speed in frames per second is given after a short (a few minute) start up time for the counting of car on a h264 encoded video. Similar speeds are attained when using feed directly from a camera. The full results are shown in table 2.3.

As we see, all platform attain at least our minimum desired speed of 5 frame per second. To put the FPS speed in perspective old black and white silent movies had a speed of 16 FPS, modern european television broadcast have a speed of 25 FPS. The performance of the Jetson Nano (14 FPS) is then slightly worse than old black and white movies and half as good as that of television. As we see, all platforms except the Jetson Nano achieve more than 60 FPS. The Jetson Nano (in bold) has a slower computational performance but is still considerable faster than the 5 FPS that we believe is the minimun needed to solve accurately the car counting problem.

A demo video of the result can be seen here.

CPU	GPU	FPS
i7-10875H	RTX 2080 Super Max-Q	111
i7-6700	GTX 960	90
Ryzen 9 3900X	GTX 1080	164
i9-9820X	RTX 2080 Ti	90
Cortex-A57	**Jetson Nano**	**14**

Table 2.3: Speed measurement on different architectures measured in frame per second (FPS).

2.6 Evaluation

For the moment we have only performed a qualitative visual-based evaluation of the counting correctness which seem to be mainly correct as can be seen in the video (cf. https://www.youtube.com/watch?v=rTM3eF-dElQ). We have not yet undertaken a systematic quantitative evaluation of the counting correctness. We do have some counting of the previously deployed system, however, we did not yet compare the count with respect to this system.

However, given the fact that the results of the visual evaluation are very good, we believe that the system will exhibit a very good performance in the final deployment setting.

2.7 Deployment

For all software components of this work, we provide docker images. Docker images is a way to provide whole operating systems together with softwares akin to virtual machines but with lower overhead. Our docker images can be deployed on a local linux machine, into a linux hpc cluster, into a Jetson Nano or into the cloud. We provide images for two different architectures (x86 and Jetson Nano).

We provide two types of images:

Command line: containing scripts and softwares meant to be launched from the command line. All configurations are done throught command line options and configuration files.

GUI: containing a x2go server together with a graphical application. The user can connect to the server using a x2go client and use the graphical application.

We provide the following images:

1. The labeling application which is a GUI docker image containing an application allowing to annotate images for fine tunning.

2. The training command line docker image which contain Tensorflow used to preprocess and fine tune the model.

3. The lane detection training algorithm, which is a command line docker image which contain the training algorithm that detect car lanes. The output of this application is a configuration file indicating the parameters of the virtual lines.

4. The counting application, which is a command line docker image containing an application which does the counting on a file or video stream. This docker image is build using configuration files and model files created by the other images. Two versions of this image is provide; one for the Jetson Nano and another for the rest.

5. Finally a developper GUI image containing tools to edit the code of the lane detection and counting image.

All theses images are currently stored on a private dockerhub repository and will be shared with partners.

2.8 Future work

The next steps are to undertake a systematic evaluation of the current solution in real world with a variety of different camera locations and weather conditions, as well as the validation of the solution by deploying the whole pipeline in the field.

WP4: Validating InaaS architecture with the IoT data integrity use-case (TrustNoise)

3.1 The context

The aim of WP4 is to evaluate the integrity of 900 noise sensors installed by SABRA in Carouge. The goal of SABRA is to draw up the road noise map of a Geneva. The noise sensors were installed by Orbiwise (https://orbiwise.com/home) and SABRA are connected via a LoRa network. These sensors are of average quality (C2 type). However, the quality required by the regulation is C1. Noise sensors are connected to a LoRaWAN network. They measure the noise level each second and send a report every 15 minutes, consisting of the following sound pressure levels in dB(A):

- Lmin: the minimum recorded level;
- Lmax: the maximum recorded level;
- Leq: the equivalent continuous recorded level (average level);
- L10,L50,L90,L95: the level exceeded for 10%, 50%, etc,. of the period (percentiles);

Hence, a data point <Lmax,Lmin,Lav,L10,L50,L90,L95> is expected at a LoRaWAN gateway every 15-minutes

As depicted in Figure 3.1, SABRA deployed in Carouge City about 1000 wireless LoRa-based noise sensors installed along main roads and busy places. To allow redundancy and finer acoustic resolution, in some hot spots, two or three sensors are installed at different heights along the same vertical axis: 1-layer (blue) spots have one sensor at 3 m, 2-layer (green) spots have a second sensor at 6 m and 3-layer (yellow) spots have a third sensor at 9 m.

Raw data collected from the sensors cannot be used as they are. Several pre-processing are required to make them available to end-user applications. One of these pre-processing steps is to evaluate the integrity of the data collected by noise sensors. The goal is to assess the "trustworthiness" of the retrieved data in order to deduce their reliability. The trust score assigned to a data is used to make necessary "arrangements": isolate the IoT device, fix the problem, etc.

The goal of WP4 is to recognise the misbehaving sensors in order to dismiss the data they send. WP4 will devise a trust model able to assess the trustworthiness of the 900 noise sensors.

3.2 The solution

One of the solutions to this problem is to decide whether we trust or not the data collected by the device using a trust model (TM). Generally speaking, a Trust Model is an agent which detects when a device is misbehaving and tags it as such. Our approach is based on the following actions:

1. Interact with the device: retrieve data from the noise sensor

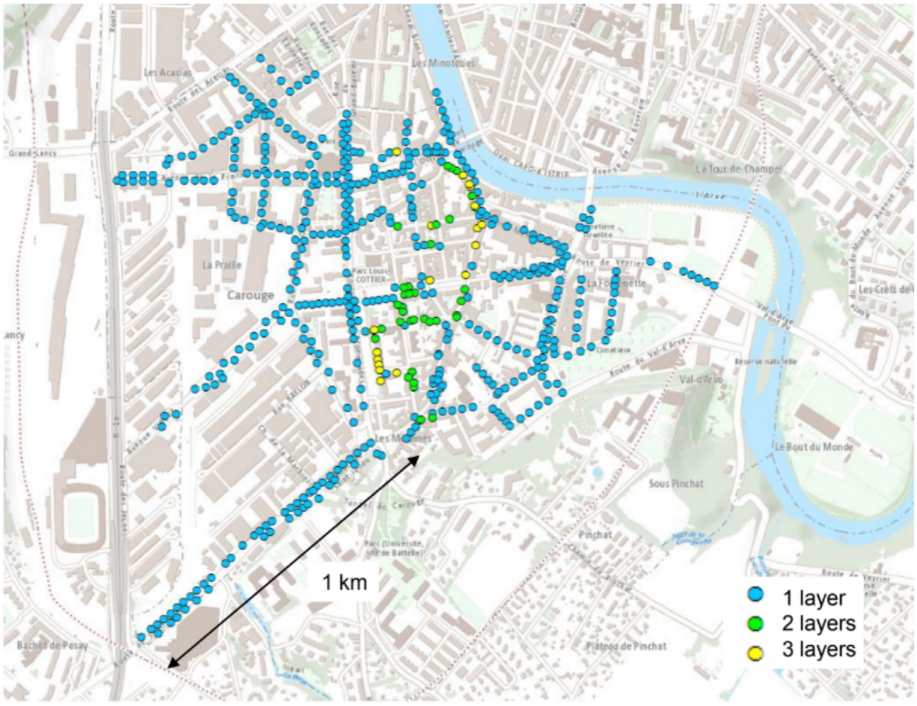


Figure 3.1: The LoRa-based noise sensors deployed in Carouge (Geneva)

2. Evaluate the quality of the interaction
3. Use this information to deduce the trust score of the interaction.

In order to accommodate different kinds of deployments, independently of communication protocols and of system architecture, a generic trust model should be parametric with respect to:

- devices. For any aspect (operating conditions, events or errors) that might affect the interaction with a device, a trust factor (TF) is defined. The number of trust factors and their nature depend on the device's type and on how it is linked to the rest of the system. During interactions, a trust factor might exhibit some variation that we want to measure: this is referred to as "trust factor score";
- applications. A given application might have different sensitivities to disturbances, which can be expressed via some coefficients for the factor scores.

The resulting model is depicted in Figure 3.2. An interaction, such as a measurement coming from a sensor, is fed to the system which, for each trust factor TFi, evaluates its trust factor score Si. These are then weighted by their corresponding application coefficients Ci and combined together to yield a "global trust score". To completely specify such a system, we:

1. Identify the relevant trust factors. These are the model's variables;
2. Define the functions to compute the trust factor scores;
3. Fix the applications coefficients, which are the model's constants;
4. Define a global trust function.

The trust model used within MEDInA is detailed in [3]

The generic trust model evaluates the quality of the received sensing data, not the device's reliability. Thus, we look at the disturbances affecting the interaction along the chain from the environment surrounding the device to the collecting point, through the network layer. A disturbance can be an operating condition, an event or an error.

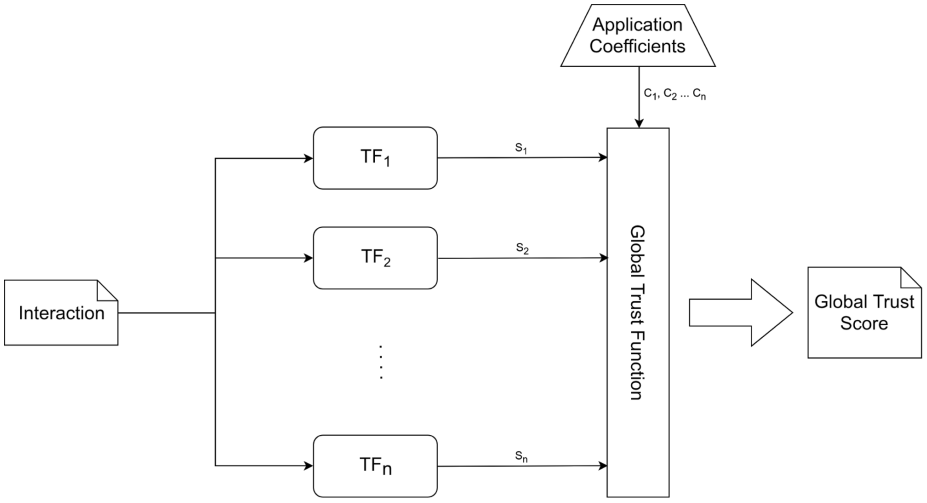


Figure 3.2: The generic Trust Model architecture

A trust factor expresses the quality of the interaction with the IoT device with respect to a given disturbance. Any disturbance must be quantified, by either measurement or estimation, resulting in a corresponding factor score. Concretely speaking, each TFi block in Figure cc represents a function which computes a trust factor score normalized in the range [0, 1].

Trust factor scores are fed to a global trust function which, through the application coefficients, computes a global trust score. Different forms of this function can be thought of. The simplest is a linear combination of factor scores weighted by the application coefficients. That is, for a given application A employing a device D we define:

$$Trust(D, A) = \frac{\sum_{i=1}^{n} C_i^A \times S_i^D}{\sum_{i=1}^{n} C_i^A} \in [0, 1] \tag{3.1}$$

The Trust Model applied to the noise sensors deployed in Carouge uses the following Trust factors:

1. Signature: This Device-specific factor represents the context of a sensor. It is defined by its environmental settings: road configuration (residential vs commercial, intersection vs straight segment, etc.), acoustic configurations of the streets, height placement, etc., and its noise patterns: acoustic levels follow recurring conditions at different time scales, such as rush hours, weekends vs working days, summer vs winter, etc. Thus, a synthesis of the above features constitutes the signature of the context. Noise sensors belonging to the same context should provide data matching a similar signature.

2. Accuracy: This device-specific factor expresses the closeness of a reported measurement to the value that a good sensor would report. We will rely on spatial neighbourhood correlations to calculate the accuracy.

3. Packet Error Rate: This network-specific factor gives a quality estimate of the communication channel between the sensor and the receiver gateway. It takes into account the ratio of the number of erroneously received packets to the total number of packets received in a time interval Δt.

4. Availability : This device-specific factor takes into account the ratio of the number of received reports to the total number of expected during an interval Δt.

3.3 Implementation/deployment

This section describes how the two trust factors “signature” and “accuracy” will be deployed. The “availability” and “PER” trust factors are straightforward and do not need any explanation.

3.3.1 Signature

The signature represents the context of a sensor. It is defined by its environmental settings: road configuration (residential vs commercial, intersection vs straight segment, etc.), acoustic configurations of the streets, height placement, etc., and its noise patterns: acoustic levels follow recurring conditions at different time scales, such as rush hours, weekends vs. working days, summer vs. winter, etc. Thus, a synthesis of the above features constitutes the signature of the context. Noise sensors belonging to the same context should provide data matching a similar signature. In other words, a context has a unique "voice", and sensors deployed in the same context should all "hear" the same "voice". The idea consists of identifying and locating all the signatures of a given city and/or neighbourhood. Later on, by comparing a sensor's new data series to the signature of the context to which the sensor belongs, we can assess the compatibility of the data series with the expected signature.

The identification of signatures must be done using "meaningful" data. As explained above, the sensors are of average quality and the data collected suffers from a data integrity problem. In order to extract the relevant signatures, we must first resolve this integrity issue. This problem can be solved by pre-filtering the noisy data to work only with meaningful data.

Extracting meaningful data

The difficulty in making this filter is that there is no definition of "meaningful" data. Indeed, filtering noisy data (non meaningful) out of noise recordings is not straightforward. Anomalies, which may come from several sources that are difficult to identify, make the validation of the filter difficult. given that the collection period is approximately one year, the amount of data is large enough to allow deletion of data. The following policy is then applied: if an anomaly is detected during one day, the whole day is tagged as "non meaningful" by the filter.

It's worth reminding here that the noise sensors measure the urban noise and not only the traffic road noise [4]. this study assumes that the former is mainly affected by the latter, as shown in figure 3.3

Based on that information, the intuition is that anomalous data would not fit this pattern and must be discarded when calculating the signatures of a given city and/or neighbourhood. Our solution, therefore, turned to data analysis in order to find this pattern and filter the data that did not respect this pattern. We decided to analyze the spectrum of the signal in order to characterize a good day. In analyzing the data by weeks, we also found that the patterns are recurring from day to day. i.e. the Mondays of a device have a pattern that is different from the pattern of Tuesdays of the same device and so on.

Since we want to analyse the shape of the signal, we have developed a filtering method that uses the Fourier Transform. Indeed, FT is well adapted to this kind of problem.

Our filter analyses each sensor independently and individually. The first step of the filter is to calculate the references that will be used to filter data. For each noise sensor, there is a reference for each day of the week .e. one for Mondays, one for Tuesdays and so on. If the data received from a sensor is similar or close to the reference, this data is considered meaningful. Concretely speaking the reference of a given sensor for a specific day of the week is the FT of the concatenation of all the occurrences of that given day.

To be able to compare two FT, they must have the same amount of samples. We must thus duplicate the signal of the day in order to obtain a signal that has the same number of days as the reference. We calculate the FT of the signal with the duplicated day to be compared and we compare this FT with its reference FT, eg for a given Monday, we duplicate it then compute its FT and finally compare the resulting FT with the reference FT of Mondays. In order to compare the two FTs, we use the Mean Square Error (MSE) function. As shown in figure 3.4, the MSE computes the difference between each vertical point: orange & blue. Orange points represent the reference signal while blue points represent the signal to filter. The final error value is the sum of all differences. We consider that meaningful data are those which MSE value is below a threshold.

To test our filter and visually represent its results, we have developed a website that allows you to interact with the filter. This website is available at the following address: filter.lsds.ch

Once the meaningful data has been extracted, we can apply clustering methods to identify the signatures.

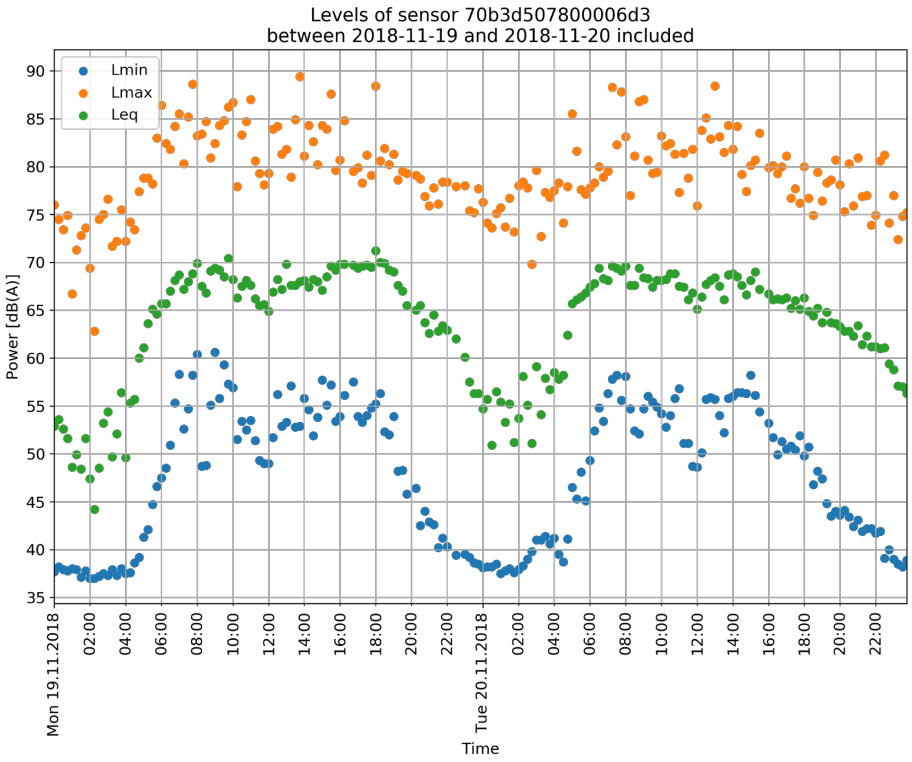


Figure 3.3: Noise levels follow a 24h pattern

Clustering

The definition of a signature is not precise. Any component that helps characterize the sound can be considered part of the signature. Following previous work [5] that defined signature as an average of levels during 24 hours, we decided to extend their approach to a weekly average (Figure 3.5). Given that each day of the week has its own pattern, we decided to represent an average week as our signature.

The purpose of the signature module is not to have one signature per sensor, which would be counterproductive. The goal is to group several sensors with a close context under the same signature. In this way we identify signature classes and can identify whether the data sent by a sensor corresponds to the signature class to which the sensor actually belongs.

Clustering techniques are widely used to group data together. These techniques are part of the unsupervised learning branch of Machine Learning. This implies that there are two phases: a first training phase and a second production phase. For the time being, we are only interested in the learning phase.

The clustering algorithm we have selected is K-Means. We have chosen this algorithm because it is relatively simple and can easily handle large data sets. We calculated the signatures of each sensor to build our data set. We then ran the algorithm on this dataset. The result of the learning is called a model. We extracted six different classes of signatures. Finally, we calculated which cluster the signature of each sensor belonged to. We displayed these results on a map (figure 3.6) in order to better represent the different contexts of the city of Carouge.

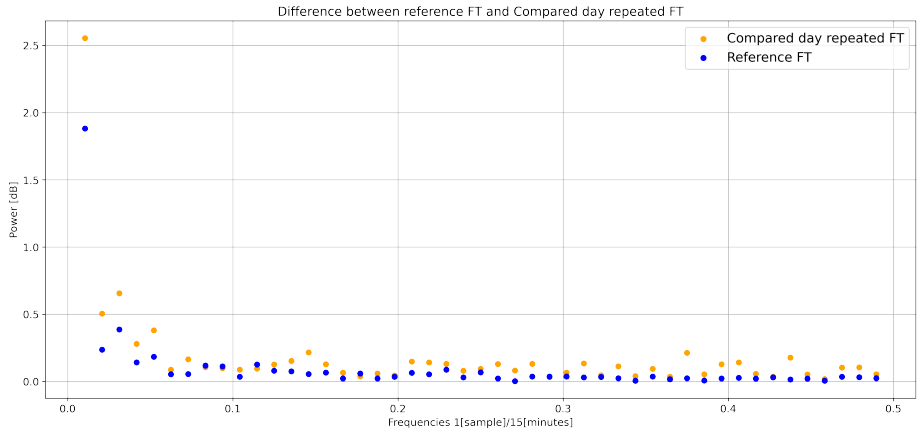


Figure 3.4: Difference between the Reference FT and the tested signal FT

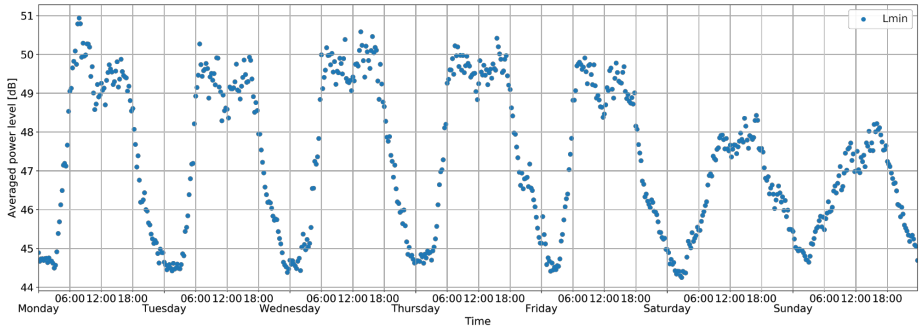


Figure 3.5: Example of signature for a given sensor

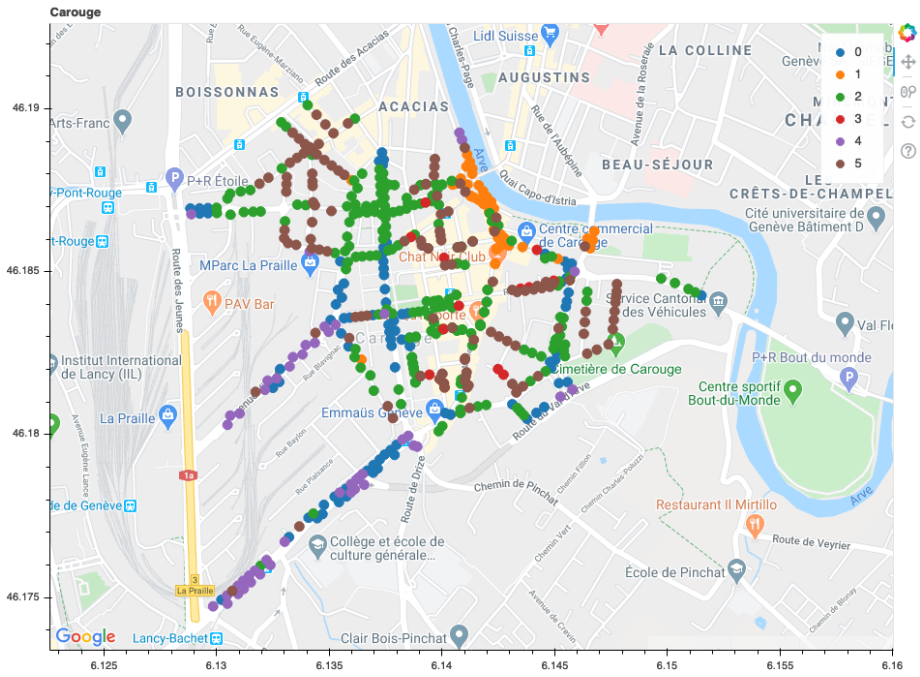


Figure 3.6: Clusters locations in Carouge city

Deployment

In this section, we will focus more on the production phase of clustering. During the production phase, we can give any one week signal and the model will tell which is the closest signature.

The trust model requires us to provide a score for each report, i.e. every 15 minutes. However, the model requires a one-week signal as input. So we have to find a way to have a one-week signal every 15 minutes. One solution considered is to use the previous data to construct a one-week signal by updating only the new value.

3.3.2 Accuracy

As stated above, the accuracy module represents the closeness of a reported measurement to the value that a good sensor would report. However, it is not possible to obtain this value in a real deployment context. We must therefore infer this value using spatial correlation techniques. Assuming that a majority of sensors behave correctly, we expect that neighbor sensors report similar measurements. We can therefore create a confidence interval in order to detect whether the reported value is within this interval. This work will be carried out during the period Dec. 2020-March 2021.

3.4 Future work

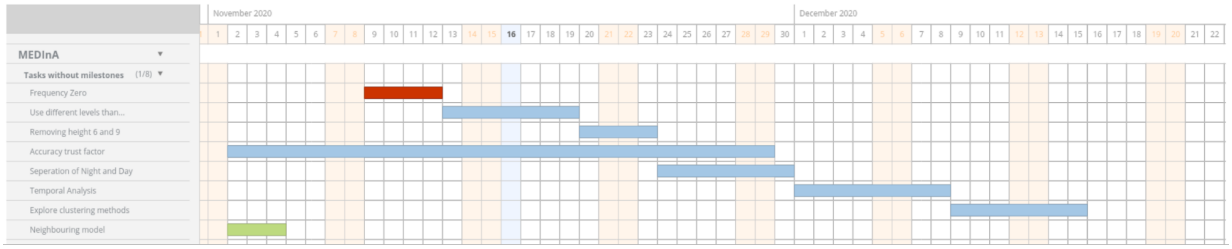


Figure 3.7: Schedule for the coming weeks

As we can see in our schedule in figure 3.7, the signature module is in the last phase and is due at the end of December. This module has required great effort in filtering data. We are currently in an iterative process to improve the quality of the clustering. Once this iterative process, we will move on to a packaging phase in which we will integrate the result of the clustering into the module. At the end of this phase, the module can be integrated into the Trust Model. On the other hand, the accuracy module is currently in progress and is due during the first quarter of 2021. When it is ready, we will also move on to the packaging phase in order to integrate the module into the trust model.

We also scheduled another filter approach proposed by the SABRA application partner. This approach is a temporal analysis.

Bibliography

[1] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "SSD: Single Shot MultiBox Detector", *arXiv:1512.02325 [cs]*, vol. 9905, pp. 21–37, 2016, arXiv: 1512.02325. DOI: 10.1007/978-3-319-46448-0_2. [Online]. Available: http://arxiv.org/abs/1512.02325 (visited on 11/11/2020).

[2] M. Sandler, A. Howard, M. Zhu, A. Zhmoginov, and L.-C. Chen, "MobileNetV2: Inverted Residuals and Linear Bottlenecks", *arXiv:1801.04381 [cs]*, Mar. 2019, arXiv: 1801.04381. [Online]. Available: http://arxiv.org/abs/1801.04381 (visited on 11/11/2020).

[3] M. N. Bouchedakh, "Generic trust framework for iot applications", Last accessed: 2020-02-25, M.S. thesis, HEPIA - Haute école du paysage, d'ingénierie et d'architecture, 2018. [Online]. Available: https://lsds.hesge.ch/wp-content/uploads/2018/05/Master-Report-Bouchedakh.pdf.

[4] Europæiske Miljøagentur, *Noise in Europe 2014*. European Environment Agency, 2014, OCLC: 918951121, ISBN: 978-92-9213-505-8.

[5] G. Zambon, R. Benocci, and G. Brambilla, "Cluster categorization of urban roads to optimize their noise monitoring", *Environ Monit Assess*, vol. 188, no. 1, p. 26, Dec. 12, 2015, ISSN: 1573-2959. DOI: 10.1007/s10661-015-4994-4. [Online]. Available: https://doi.org/10.1007/s10661-015-4994-4 (visited on 06/06/2020).