University Institute of Lisbon

Department of Information Science and Technology

A Machine Learning Approach for Indirect Human Presence Detection Using IoT Devices

Rui Nuno Neves Madeira

A Dissertation presented in partial fulfillment of the Requirements
for the Degree of
Master in Computer Engineering

Supervisor
Prof. Dr. Luís Miguel Martins Nunes, assistant professor
ISCTE-IUL

September 2016

Resumo

A recente maior democratização da tecnologia contribuiu para o aumento da disponibilidade de dispositivos dedicados à melhoria dos nossos espaços de vivência e trabalho, capazes de controlo remoto pela internet e de interoperabilidade com outros.

É neste contexto que a detecção de presença humana é fundamental pois: permite a automatização de acções, a aprendizagem de padrões de uso, a detecção de problemas de doença ou intrusão, etc. Dispositivos específicos de detecção de presença normalmente tem falhas dependendo da sua natureza, e não costumam estar coordenados de forma a melhorar a performance.

Coordenar os aparelhos de forma a obter um nível mais inteligente de uso requer um outro dispositivo ou software capaz de comunicar e controlar os outros. A Muzzley é uma empresa que criou uma aplicação móvel onde os utilizadores podem registar todos os seus dispositivos e depois controla-los a partir do programa.

Esta dissertação propõe uma abordagem para a detecção de presença baseada na utilização de métricas extraídas das mensagens entre os dispositivos e a plataforma da Muzzley. A ideia é que a detecção não será feita por informação de sensores específicos mas sim pela analise de padrões de interacções com os dispositivos. Conjuntos de dados anónimos criados na plataforma serão submetidos a uma fase extensa de processamento de forma a criar atributos interessantes para o treino e teste de algoritmos de aprendizagem automática.

As contribuições principais deste estudo são os algoritmos de processamento construídos para a criação da informação relevante para a tarefa, a demonstração da capacidade do uso destas interações para a detecção de presença, e os métodos usados de forma a melhorar a performance da abordagem.

Palavras-chave: Detecção de Presença Humana, Inteligência Ambiental, Internet das Coisas.

Abstract

The recent increased democratization of technology led to the appearance of new devices dedicated to the improvement of our daily living and working spaces, capable of being remotely controlled through the internet and interoperability with other systems.

In this context, human presence detection is fundamental for several purposes, such has: further automization, usage pattern learning, problem detection (illness, or intruder), etc. Current intrusion detection devices usually have flaws depending on type and many times are not coordinated for better performance.

Coordinating the devices for higher level operation however requires a device, or software, that is able communicate and control them. Muzzley is a company that tries to solve this issue by creating a mobile application where the user can register all its devices and control them from there.

In this dissertation we propose an approach to human presence detection using metrics based on messages between devices and the Muzzley platform. The idea is that the detection does not rely on information from specific presence detectors, but that it is able to achieve its purpose by analyzing the patterns of interactions with the devices. For this, anonimyzed datasets created by the Muzzley platform are submitted to an extensive processing in order to create meaningful features that will then be used with a machine learning algorithm for training and testing.

The main contributions of this study is the processing done to create meaningful information for the task, the demonstration of the capabilities of the interactions between these devices and platforms for human presence detection, and the methods used to improve the performance of the approach.

Keywords: Human Presence Detection, Ambient Intelligence, Internet of Things.

Acknowledgements

I would like to thank my parents and my family for all the love they gave me all my life, and for their efforts to provide me a comfort living that allowed me to develop myself as a human being and to focus properly on my academic path.

I would also like to thank my supervisor, Luís Nunes, for all his support and help in this project, and for all I learned from him in these months and in my time at ISCTE-IUL.

A big thanks to my friends from outside the university, their friendship was crucial for my social life and my personal development, allowing me to learn things from other areas and to have precious escapades from the academic work.

And finally, a very important thank you to everybody that worked beside me at the university during the thesis, since this kept us motivated and inspired through this hard and possibly lonely journey.

Without all of you this wouldn't have been possible. My deepest appreciation.

Contents

List of Figures

List of Tables

Abbreviations

AdaBoost	**Ada**ptative **Boost** (see page 61)
AVG	**AV**era**G**e (see page 46)
IoT	**I**nternet **o**f **T**hings (see page i)
RGB	**R**ed **G**reen **B**lue (see page 15)
SMOTE	**S**ynthetic **M**inority **O**ver-sampling **TE**chnique (see page 53)
SVC	**S**upport **V**ector **C**lassifier (see page 41)
XGBoost	E**X**treme **G**radient **Boost** (see page 61)

Chapter 1

Introduction

Human presence detection is an ongoing challenge in several scenarios and applications, one of them is in ambient intelligence and home automation. This project intends to create a system able to infer about presence using general information and statistics of usage and interaction with different types of devices.

Ambient intelligence in computing refers to technology that is non-intrusively integrated in an environment, doesn't need user intensive interaction, and adapts, in an automated way, to the necessities of each user and context. The goal of this technology is the creation of complex systems with simple interfaces that enhance the quality of our daily lives [1] [2] [3].

In order for this vision to become a reality, several authors have pointed out certain attributes that the systems need to have [1] [2] [3] [4]:

- Integration, the devices should be embedded in the environment in a pervasive but non-intrusive way.
- "Awareness", the technology must be able to understand its context in order to correctly interact with it.
- It should be tailored to the needs of each user, and be able to adapt to their behaviour changes and external factors.

- It should be able to predict the activities and needs of the users.
- And distributedness, several devices and systems should be connected and exchange information, but in a way that the failure in one of the network nodes does not lead to a total system failure.

Internet of things (IoT) is the network of physical objects with electronics, software, sensors and connectivity capabilities that allow these devices to communicate with each other and to exchange information. Because they are connected in a network, the devices can be interacted and monitored remotely from another point in the network. By joining the information of each device, and through the interaction of each device with its environment, we are able to build complex systems, with better precision, that may enhance our quality of life [5] [6] [7] [8] [9].

The previous concepts are connected, the internet of things is a technology that enables the creation and improvement of more complex ambient intelligence systems. A network of several devices of different types and purposes is a fundamental tool to understand the context in which the system is operating. The fusion of information from the devices is very important to decide which actions to carry out, and through the combination of the actions of several devices the system can cooperate with the user and fulfill his needs.

Sensor fusion is the combination of information from several sensors in a way that the end result has more quality and precision than the information of each single one. It is a fundamental concept in ambient intelligence, and in a lot of other areas such as robotics. In order to be able to understand the needs of its user and context, an ambient intelligence system has to structure and process very efficiently the data it receives from its device so that quantity is also quality [10] [11].

Home automation can be seen as an instantiation of the concept of ambient intelligence in which the environment we want to "make intelligent" is a residence,

and the functions we want to control and monitor are the temperature, the ventilation and humidity, the energy consumption, the garden sprinklers, etc.

1.1 Motivation

The problem this project intends to address is related to this vision of ambient intelligence for personal residences. Nowadays is increasingly common to have one or more devices related to this area on our houses. [11] predicts that before 2020 the internet of things will be comprised of more than 50 billion devices. Even today there are quite a few different devices available in the market for a lot of different purposes. Good examples are thermostats, humidity controllers, carbon dioxide sensors, motion sensors, door sensors, etc.

A problem with these devices is that each manufacturer company has its way of communicating and programming its device. Muzzley is a Portuguese company established in 2012 with the purpose of the creating a mobile application and a platform that allows for the interaction with all types of devices, independently of their brand, from a single system. The application has been evolving and nowadays it boasts of more advanced functions, related to ambient intelligence, such as coordinating the registered devices through programmed rules or behaviours learned from the information received from the devices and user interactions. It is in this context that human presence detection becomes an important thing.

The information about human presence at home is interesting for several reasons:

- Allows the automation of several actions such as turning on the lights when someone arrives, turning off when they leave, turning on the kitchen automatic cooking robot, turning on the smart sockets, etc.
- May aid in some forms of intrusion detection.

- Allows the learning of user routines, since it is fundamental to know when a user is at home or when he is simply interacting with the devices remotely.
- Can help detect behaviour anomalies related to health conditions.

Specific and high fidelity detectors however are usually expensive for domestic use. The more affordable devices have limitations, for example a person sitting down on a couch will not be detected by a regular motion sensor. Besides this, some devices may not be precise enough on their own.

1.2 Objectives

The idea of this dissertation is to process the data generated from interactions between the user, the devices and the Muzzley platform in order create general metrics, not related to each device usage or characteristic, like for example average number of interactions per day, or number of interactions in the last half hour. And then use them to train machine learning models for presence detection. The information to be used comes from every device possible, even if it isn't explicitly direct to presence, because the idea is to do an indirect detection based on usage and behaviour patterns of the devices available in the house.

The point is that if this proves to be a good method, it will allow to infer information about presence for users that don't have presence related sensors. And may increase the overall fidelity of houses equipped with presence detectors compensating for the inherent flaws of each type of sensor.

1.3 Scientific Contribution

This dissertation presents the following contributions:

- reviews state of the art projects in the area of ambient intelligence and internet of things related with the presence detection problematic.

- creates and fully describes an approach to process data from a system that monitors and controls several types of IoT devices with the intent of generating features for machine learning classification.
- presents the potential capabilities of the generated data used with machine learning algorithms for human presence detection.
- Elaborates on the techniques and processes used for result exploration and to improve the scores of the machine learning classification.
- provides conclusions on the approach reasoning on future work and implementation possibilities

Part of the work present in this document is based on the final report delivered for the Introduction to Research in Engineering course of the Master in Computer Engineering, titled: "Human Presence Detection in Ambient intelligence Systems Without relying on Specific Sensors".

The work conducted in this dissertation resulted in the following publication:

R. Madeira, L. Nunes, (2016), "A Machine Learning Approach for Indirect Human Presence Detection". Proceedings of the International Conference on Digital Information Management, 2016, Porto. In press.

1.4 Structure of the Dissertation

In Chapter 2 we present projects related to this one stating their contributions and declaring their similarities and differences with this dissertation. In Chapter 3 the Muzzley platform is described, the raw dataset contents are explained and the processing used to generate the feature and the new csvs for machine learning are described in depth. Chapter 4 begins by explaining the machine learning scripts and tools used, it then presents the initial results obtained. After this several sections about exploring the results and improving the scores follow. Chapter 5 concerns the final experiments done and the improvements are discussed. Finally

in Chapter 6 conclusions are drawn on the results obtained and the developing process of this dissertation.

This study followed the Design Science Research method [12] [13]. For further description of the phases applied in this case refer to the document presented for the Introduction to Research in Engineering course. Described briefly, the dissertation started with the identification and motivation behind the objective. The objectives were presented and now will continue to be elaborated upon, along with the state of the art. Chapters 3, 4 and small details of 5 concern the design and development phase, where the developed plugins, machine learning scripts and improvements are described. Chapter 4 and 5 are the demonstration and evaluation stage. And finally, this document along with the published paper serve as the communication step.

Chapter 2

State of The Art

2.1 Home automation, Ambient Intelligence and Internet of Things

In recent years the increased general availability of computing technology, the decrease in size of powerful devices of this area, the easiness of implementation of network infrastructures, the practical advances in artificial intelligence and the familiarization of people with these new technologies, has led to an increase in the accessibility of devices related to ambient intelligence and internet of things to the general audiences [3] [14] [15]. The main devices that can be found in common technology stores are light bulbs, capable of network connection and being remotely controlled. Sensors, like thermostats, carbon dioxide sensors, motion sensors, etc. And surveillance cameras with network connection.

The increased presence of these devices, that are easy to install and start to use, without the need for large infrastructures to support them, with the possible of coordination between them through third party products, have made "traditional" home automation systems less popular. Examples of "traditional" home automation systems are the central heat control, intelligent shutters, intrusion detection, gas leak detector, etc. The popularity of these new devices has however helped create a bigger interest in the area, and is now common to find in Portugal,

stores dedicated to the area, offering great quality products, and also house which are built with home automation in mind [16].

Because of this there has been lot of interesting work in the area of ambient intelligence for home and work spaces that by fitting the environment with different types of these devices, with different purpose, aim to reach the technological visions for this area. One of these projects is Dream Green House [17], which has the goal of creating the world's most intelligent house, but also focusing on ecology. The project consists of a series of subprojects, each one focusing on one aspect to improve the house, for example, a module to control energy consumption, another for temperature control, luminosity control, water consumption monitor, etc. The house continues being developed, as new technology arises in the area, new functionalities are structured and implemented and older modules are updated.

A system, built using Linux, Java and a Raspberry Pi serves as the main controller of the devices placed in the house, receiving information from the other devices and systems. In 2013 [18] developed a presence detection system at a room or zone level for the house. This system infers about presence by using a large quantity of information gathered from several sources. The type of data used and its source may indicate presence explicitly, such is the case for pressure sensors placed in beds and chairs, or implicitly, in the case for example of the detection of network traffic coming from a PlayStation system.

Another example of house that implements these technologies is the Gator House [19], which has the objective of creating a smart home that can help and monitor the daily life activities of the senior population or people with special needs. The house has a lot of systems and functionalities, such as smart pantries and fridges that can create automatic shopping lists, smart ovens, smart sockets, pressure sensors on the floor, etc. One of the most interesting contributions of the project is the creation of a model of architecture of middleware to control and coordinate all the systems implemented in the house. This model is structured in layers, the base layer has to do with the raw data produced by the devices,

and then, progressively through the layers the system becomes more complex and abstract in the sense of the created information.

In [20] the authors describe a system that joins information from several sensors, using Bayesian networks and Markov chains to reduce each sensor noise, in order to detect human presence. The system was built by layers and receives information from devices such as thermostats, door sensors, and cameras. Its capabilities were tested by using it to monitor an office in the University of Palermo.

2.2 Human Presence Detection Devices

The goal is to create a system that can detect human presence without relying on specific detection devices. However, to build the system and to better understand the results that will be obtained it is important to know which devices are available that focus specifically on detection, and what are their capabilities and weaknesses. In [21] the authors do a survey on available methods to detect presence and other related concepts such as occupancy, the number of people present, and the identification of each person. They also explain the physical traits used by the devices to infer about these concepts. Then the devices are classified according to the attribute they use to operate. The authors also refer some work related to sensor fusion. The survey ends with a summary on the capabilities of each type of device and conclusions for the area, one of which is the importance of the creation of sensor fusion systems, on a greater scale and complexity, in order to solve these problems with better precision.

2.3 Sensor Fusion

In addition to those described in [21] there are a lot more projects related to sensor fusion to solve presence detection problems and other related issues such as person tracking and identification. For example in [22], Bayesian networks and Markov

chains were used to join the information from several sensors in order to detect people and track their movement. An interesting aspect of this project is that they used Gaussian probability distribution functions to model the information gathered from each sensor by varying the average and covariance of each function according to the properties of the sensor.

In [23] the authors used a Kalman filter to join the information from the Wi-Fi module, magnetometer, gyroscope and accelerometer of a smartphone with the purpose of tracking a person's movement on a known space. Initially sensor fusion was thought to be more important for the project since the approach was different than the present one that focuses on the interaction statistics instead of sensor values, but these projects continue to be relevant as state of the art.

2.4 The Concept of Time in a Sequence of Events

Another important aspect for this problem and area, is the usage and processing of data related with time. There are many ways to model information about time: timestamps, offsets of time between events, time windows of varied size, etc. The authors of [24] studied several ways identification and segmentation of human activities in video and motion capture files. An interesting conclusion is that in this case the usage of a time window is not the best way to solve the problem.

In [24] it's described a way to process time series, in real time, using a hierarchical tree structure. This algorithm seems interesting for this type of problem since the devices involved generated a lot of data in short time. The authors in [25] have done an extensive analysis of the usage of different types of clustering algorithms on time series data.

2.5 A Unique Platform

This project however is so far unique and quite different from the others discussed before. A lot of different types of devices from different companies interact in the Muzzley platform. Each one with its details on the way it works and communicates with its manufacturer platform and with Muzzley. Besides this, the system should work in every house and every user, instead of most of other projects that focus on a specific house and, or, type of user. The system must also behaviour correctly independently of the environment on which the user decided to use the devices, and independently of the way the devices are located in the space. And finally, the system must be able to adapt to each user daily life habits and learn his patterns of device usage.

Chapter 3

Data Processing

3.1 The Muzzley Platform

Muzzley is an application for mobile devices that allows the control and monitoring of devices from different manufacturers in a single system. A user just has to register his devices in the application, he then has access to a user interface suited to each type of device that allows interaction with its properties.

For this to happen the device manufacturer has to integrate his devices with Muzzley. This can be done cloud to cloud, in the case of manufacturers who already have a built platform to support their devices, or cloud to device using Muzzley's own platform to interact with the device. To do the integration the manufacturer has to provide information about the type of integration, interfaces or HTTP URLS needed, etc. Then he has to define an ontology that establishes the properties and components of its devices, and the relationships between them. A user interface used to interact with the device must also be provided, a pre-defined one can be used or a new one using HTML and CSS can be created. This is what the application will use when navigating to the device page.

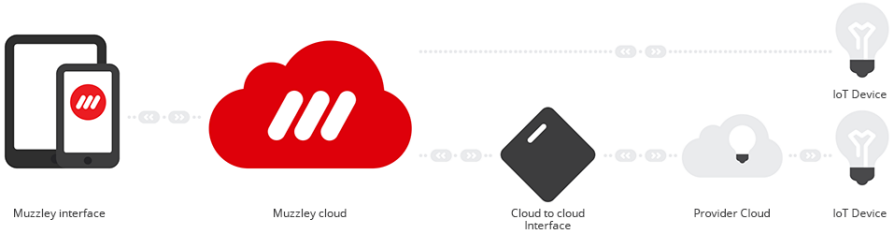

FIGURE 3.1: Overview of the interactions in the Muzzley ecosystem[1]

The manufacturer must also define its device hierarchy. This is used by Muzzley in order to have its own way of identifying the devices and their properties without relying on each company's identification method, since that would be very difficult to integrate and would probably lead to conflicts. The hierarchy in the Muzzley platform is composed by profile: this is an identifier for a type of device, for example a light bulb or a thermostat. Channel, a unique identifier for a certain concrete device. Component an identifier for a certain component of a device, for example a cooling system with more than one fan, each fan is a component. And property the identifier for a certain property of a component, in the cooling system example it could be the speed and angle of the fan.

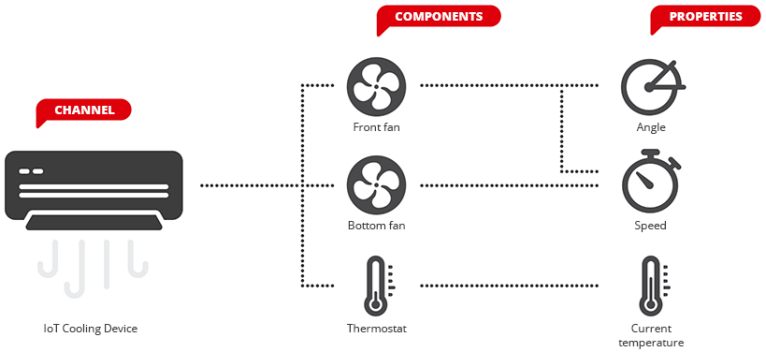

FIGURE 3.2: Muzzley device hierarchy[2]

To communicate with the manufacturer clouds and the devices Muzzley uses a set of JSON schemas that specify the structure and type of data the messages between these entities uses. So the manufacturers must also specify for each property which type of schema it uses. For a color light bulb this could be for the

[1] `https://www.muzzley.com/documentation/images/integration_assets/_img1.png`
[2] `https://www.muzzley.com/documentation/images/integration_assets/_img5.png`

example the colour-rgb (Red, Green, Blue) schema that indicates that, aside from other fields, the messages would have the r, g, and b fields with a value between 0 and 255.

Muzzley also has an automation engine that allows the users to set up rules that control the devices. These rules can use a lot of different events as triggers: time, location, properties of other devices, etc. And can then actuate changing the properties of others devices. These rules capability comes straight out of the box only requiring the user to set up the rules or to adopt some already pre-defined ones.

3.2 Segmentation Engine

The Muzzley system generates data related to the messages interactions between the user application and the platform, and then between the platform and each device manufacturer systems when this applies. The quantity of messages generated by the platform in a short period of time is quite big, and so gathering and storing this information is also a challenge. In order to facilitate this process Muzzley has developed a program called Segmentation-Engine. This program can listen to the messages being transmitted, and operate on them. It stores the messages, filtering and, or, masking some fields if necessary, and compress the information if needed. It can also calculate metrics from received messages or saved datasets, delete saved information according to some parameters, etc. The program was built to do this continuously, while listening to messages or new dataset entries, serving as an automated dataset creation tool.

The Segmentation-Engine is a program written in Python, that uses a lot of libraries and operates using other applications such as the Redis and Mongodb databases. The program is used through the command line and it has 3 main functions. Schema operations, regarding actions on locally stored schemas. The datasets operations, for the creation of a dataset, to append information to an

existing one, or to clear datasets. And the metrics operations to do calculations and create new data based on the stored datasets.

To create the datasets the program listens to the Muzzley platform and stores the captured messages in csv files. Options can be applied to these operations such as type of messages to store. Details on these types on section 3.3. A list of fields to exclude from the dataset. And masking so that the ids present on the dataset aren't the ones used in the real application messages.

The metric operations are the crucial part for the data processing phase of this project. These allow the creation of new information by reading the stored datasets and do calculations on them. In order to do this new parts of the program, called plugins, can be developed, each corresponding to a new metric. Each metric has a configuration that defines if it is active, which types of schemas from each dataset it has access to, etc. The program can then run a metric, a list of them, or all of them. If the metric needs to access a csv dataset the Segmentation Engine already is prepared to deal with this abstracting the reading of the datasets and saving on a Redis database the files and lines that were already processed. In case the plugins need access to a MongoDB database the program also facilitates this procedure.

The program also has a set of production services, and more may be added in the future. These are deployment configurations that can easily be used to run the program with a specific purpose. One of them is the muzzley-dataset-writer a configuration that keeps the Segmentation Engine running, listening to the messages sent in the Muzzley platform and creating datasets based on them. This is the process responsible for creating the initial sets for this project.

The data created will then undergo a processing phase to create the features and information that will serve as input to the machine learning algorithms. This phase was done by developing plugins for the Segmentation Engine that will calculate the necessary metrics. These plugins use both the Redis database when reading from the stored datasets and the MongoDB database to store new metrics

and be able to correlate metrics and calculate new ones. Developing these algorithms within the Segmentation Engine however puts some constraints on them, mainly they have to be ready to work with stream-like data, because the program is prepared to handle data this way.

Developing the plugins in Python, a language we didn't have previous experience with, and building the algorithms in accordance with the Segmentation Engine structure meant an initial adaption phase with slower development. In retrospective this paid off, because there was the opportunity to learn a new programming language, that is very suited for the jobs of data processing and machine learning, also the created plugins follow the Segmentation-Engine way of operation so in the future they can be easily integrated for other purposes or built upon to improve this project.

3.3 Structure of the Raw Dataset

The messages from a "raw" dataset are divided in three types: messages from the application to the device in order to obtain its state called user_reads, from the application to the device to change its state named user_writes, and the messages from the device to the whole platform in order to signal a state change called device_updates.

The Segmentation Engine stores the data in csvs in folders. Inside the main folder there are 3 subfolders that correspond to each type of dataset. Inside each one of these there are subfolders for each type of schema. The schemas, as described before, are related to how each message should be composed depending on the property it refers to. So there are folders for color-rgb, of, for example, light bulbs, for brightness, for battery levels of certain devices, etc. Inside these folders there are csvs, one per day, where each line correspond to a captured message.

3.3.1 user_reads

These messages concern direct requests from the application or the platform to a specific device in order to obtain the value of one of its properties. These requests might be triggered by the user, for example opening the application, or by the rules module if it needs to know a property at a certain time. The response to this request with the value of the property is given directly and is not part of the device_updates data set, this also means that response is not gathered by the Segmentation Engine, and so values of properties must be learned from the updates or the writes.

These messages contain information about the ids and names of the device hierarchy to which the request is sent, and all other related information, such as property classes, schema, etc. It also contains information about the time of generation of the message in the timestamp field, the user_id corresponding to which user generated this message and the trigger field, that marks if this message was generated by a user or a rule.

This and the next type of dataset, user_writes, also have an additional special field that will be called user_NDI (non-disclosed information). This field contains special information that won't be disclosed in this document. The information was gathered from users who agreed to help this project, allowing for the data collection, and it was only handled by Muzzley. The idea is that with this information some more realistic datasets can be created, however it is still very different from having explicit time information about when there was presence at home.

Muzzley's insight was that these read messages were the least important for this project since that they don't always mean a direct interaction with a device or the platform. They will however be very important to have data about the user and his devices. Regardless of this, the messages will also be present in the rest of the processing because they might contribute for the presence detection.

3.3.2 user_writes

These messages are instructions, generated by the user or a rule, to a device in order to change the value of one of its properties. They are very similar to the user_reads messages, but have some fields more depending on the schema of the message. For example, if the message is for an rgb coloured light bulb to change its colour then it will have the additional r, g and b fields. Like the user_reads, these messages have important information about the users and their devices. They also might contribute a lot for the detection problem, since they are specific requests for a property change, and the trigger field will also tell if these were done by a user or by a rule.

3.3.3 device_updates

These messages are quite different from the other types, they are sent throughout the platform and inform the concerned entities, a user application for example, about the value of a device property. Because they inform the value, these messages have, like the user_writes, more fields depending on the type of schema. However, since these don't originate in a user application, they don't have the user_id.

Because of not having a user_id field, each message will have to be mapped to one, or several users. But this is a fundamental step since these updates may give a very good insight in terms of presence detection. Another problem with the dataset is that the updates are generally periodic, or originated by a property change. This means that devices who broadcast their status frequently will generate a lot of information that maybe won't be as useful, and the same is true for devices who change their properties very frequently, like for example, thermostats or weather stations because of fluctuations in temperature, atmospheric pressure, etc.

3.4 Plugins Developed

As described before, because of the nature of the generated data, the Segmentation-Engine is prepared to work in a stream like way. In order to continue this paradigm the developed plugins must be prepared to be able to process stream-like data instead of batch-like which puts some constraints on the complexity and efficiency of some algorithms. Clearly some algorithms could have an overall better structure, and some parts could have being developed in a more efficient or clear way.

Adding to the requisites described, the quality of the code also suffered because the developing was more incremental than completely planned from the beginning. The reason for this is that more insight on the problem was gained from working on it, and also due to the new fields added to the data during development, requested after the initial analysis of the results. And thus, more functionalities were interesting to add to the plugins, increasing their complexity. Regardless, the overall quality of the plugins is good and the processing can be completed in a reasonable time window.

The next diagram shows the flow of information between algorithms, evidencing also their order. Next to each a brief summary of its functions can be found:

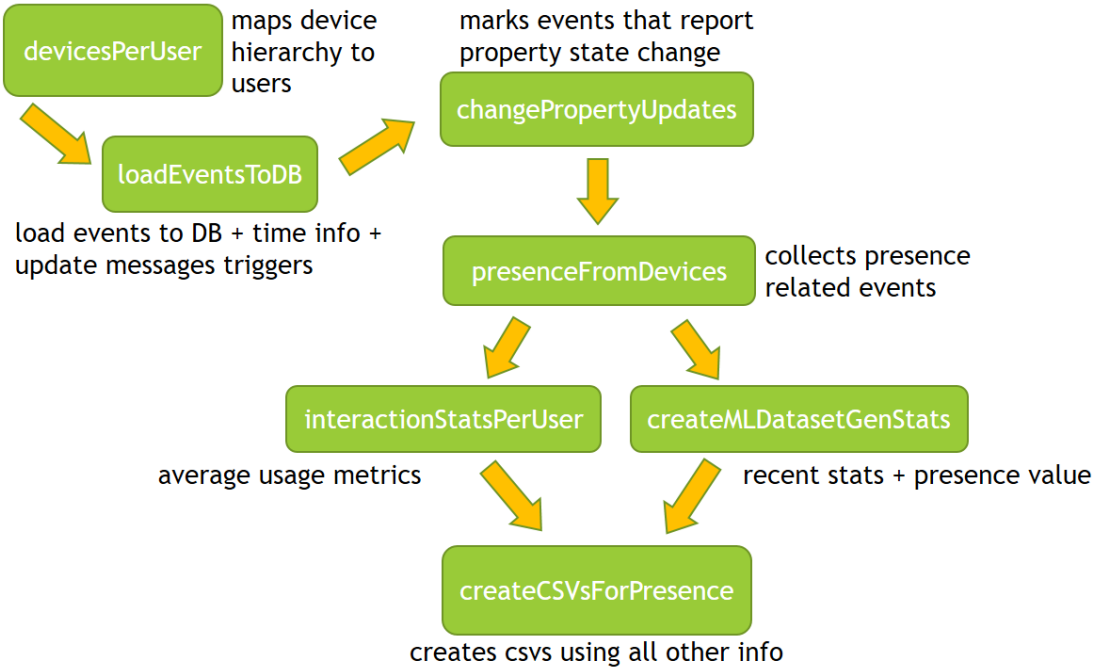

FIGURE 3.3: Sequence of the processing plugins

The next diagram shows each plugin according to its interactions with other components:

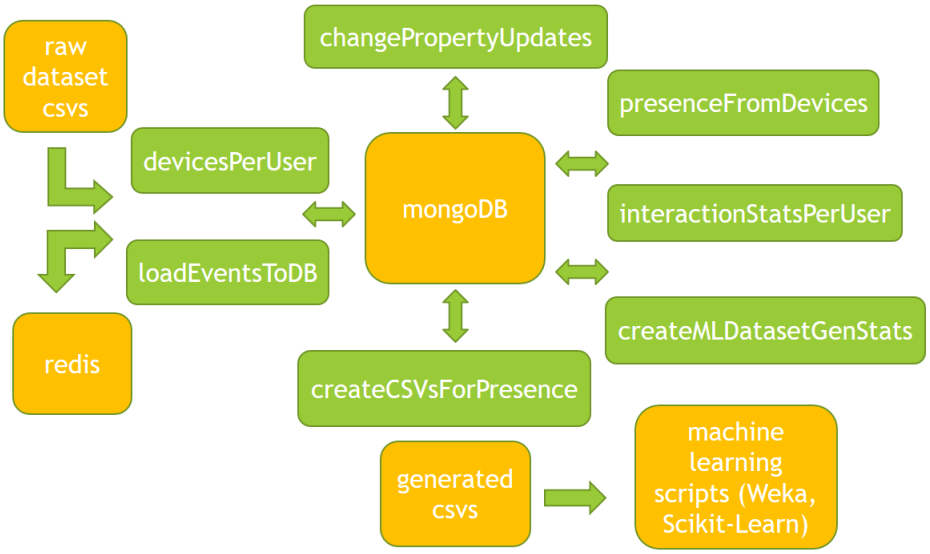

FIGURE 3.4: Component interactions

A detailed description of the plugins will now follow, along with implementation decisions, objective and possible parameters.

3.4.1 devicesPerUser

This plugin iterates through the dataset types that have information about user_ids, that is the user_reads and writes. The objective of this plugin is to create information per user about its devices, including the devices hierarchy. The algorithm will create a document with fields like, number of devices, number of components, the complete device hierarchy of that user, different schemas used by these devices, etc. This document will then be inserted in MongoDB.

FIGURE 3.5: Partial example database document for the devicesPerUser plugin


```
    "total_Schema=power-w" : 1,
    "totalDifferentSchemas" : 1,
    "totalComponents" : 1,
    "totalProperties" : 1,
    "devices" : [
        {
            "profile_name" : "WeMo Insight Switch",
            "channel_id" : "881406K110009E",
            "profile_id" : "569521afbd95b4a08a9ca9e8"
        }
    ],
    "totalDevices" : 1,
    "differentSchemas" : [
        "power-w"
    ],
    "properties" : [
        {
            "channel_id" : "221519K121109E",
            "property_id" : "energy",
            "property_name" : "Energy"
        }
    ],
    "timezone" : "None",
    "totalSchemas" : 1,
    "totalIndividualUnits" : 1,
    [...]
```


The plugin starts by doing a query to its correspondent collection in MongoDB to see if there are already users in the database. Then it starts to use the Segmentation Engine built in capabilities to iterate through the stored csv datasets, skipping the device_update dataset because this one doesn't have user_ids and

thus is not useful. Per each message iterated the plugin will use the user_id present to find in the dictionary if there is already that user, if there isn't, it creates a new user, which is also a Python dictionary, with that id and certain count fields such as totalDevices, totalSchemas, set to 0 and list fields, such as devices, differentSchemas, etc. empty. Then it will check if there is timezone info for the present user, if there isn't and the current message has a timezone, it will assign that timezone to the user as another field in the user dictionary.

Then the algorithm will check the current user individual units list. This is a list of dictionaries each one corresponding to a unique device to the property level in the Muzzley hierarchy. So each dictionary has profile_id, channel_id, component_id, property_field and schema field. The idea is to see if the current source from where the message came has already been assigned to this user, this is done checking if there already is a dictionary on the list with the ids present on the message, if there isn't, one is created and added to the list.

If there is already such item on the list the algorithm will continue to process messages since nothing more is to be added to the user. However if the individual unit is new for this user, the plugin will continue by checking the similar lists, devices, components, properties, to see if that id has already been mapped to this user, and if not creating dictionaries with the correspondent id and the channel id. For example adding the current message component_id and channel_id as fields of a dictionary in the components list.

It will also use profile, component and property labels to add counts to the user dictionary, for example for a light bulb message being processed it can add a plus one count to the total_Comp=bulb field of the user. The schema is also added to a list of different schemas if it isn't already there.

A user that was modified in this run of the algorithm is marked has changed and in the end the user list will be iterated in order to update the database with the new users info. This is an implementation decision made to avoid updating the database each time new information is added to a user. This works well, however because the plugin takes some time to iterate through all messages when

the dataset is big, for example 1 hour for a month long data, additional methods should be added to the plugin to save the new gathered information periodically in case there is a problem during the run.

3.4.2 loadEventsToDB

The main idea of this plugin is to insert into MongoDB all of the events of all types of a dataset, so that it's easier to do queries and create information in other plugins. The algorithm also does some more things: the device_updates type of messages don't have the user_id in then, so using the information created by the devicesPerUser it can, according to the profile, component and property id, map a user_id to each device_update message. This plugin also uses the timezone information of each user, retrieved in the devicesPerUser, and then using the timestamp of each message it can calculate the date of the message. This is very important to create usage statistics. Finally it is very interesting for the presence problem to know which update occurred without a previous user_write message, because this indicates that either it changed by itself, ex: the value of a thermometer, or it changed because of a manual interaction, ex: the user turned the light bulb manually so he has to be at home. Comparing the values and timestamps between user_writes and device_updates messages the algorithms is able to mark which update was due to a user_write.

The plugin starts by doing a query to the devicesPerUser collection in MongoDB in order to retrieve every user we have information so far, and maps these users to a python dictionary using the user_id as key. Then it starts reading the stored user_reads dataset and generates a dictionary from each message, with each field with its correct type, and extra time related fields, minute, hour, day of week, calculated using the message timestamp and the user timezone, if one is known in the devicesPerUser information.

After processing the user_reads dataset the algorithm moves to the writes and updates since for this implementation these have to be processed together. Write

FIGURE 3.6: Partial example database document for the loadEventsToDB plugin


```
"schema" : "https://ontology.muzzley.com/schemas/v1/timestamp",
"timestamp" : NumberLong(1462098710812),
"component_label" : "Plant",
"dayOfWeek" : "Sunday",
"month" : "May",
"timezone" : "Europe/Greenwich",
"trigger" : "user",
"property_name" : "Water last reading",
"hour" : 11,
"format" : "seconds",
"day" : 1,
"component_id" : "plant-sensor-1",
"profile_id" : "117e12e51bbb443ec06d7d",
"changeProperty" : true,
"profile_name" : "Koubachi Plant Sensor",
"component_classes" : "com.muzzley.components.sensor.plant",
"channel_id" : "31189",
"property_id" : "water-last-reading",
"io" : "r",
"minute" : 31
[...]
```


messages are read, events for them are created, like in the reads, and they are stored in a buffer. When the buffer is full, its length is a global variable, the algorithm iterates through the updates and checks if the element in the middle of the writes buffer has a lower timestamp than the current update message, if so half of the buffer is flushed inserting the write events in the database and loading new ones. When the middle element timestamp is no longer lower the algorithm will process the current update event.

Processing the current event has two phases, first the plugin searches the writes buffer to see if there is a write to the same device property, meaning profile, channel, component and property id between messages has to be the same, with at most a 4 minute time difference. If so, this means the current update event was caused by a write message, a trigger field is created on the update with the write message trigger and a user_id field with the corresponding user id. 4 minute is a heuristic defined time frame, discussed with Muzzley that is used in order to take into account delays in the network, delays while gathering messages and delays of the device generating the update. An update event is appended to a list per correspondent write message found.

In the second phase the algorithm goes through the devicesPerUser information and if the corresponding profile, channel, component and property of the message matches a unit in a user information then an update event is appended to a list with that user id, but this time with a trigger that marks it as an independent update. This is only done if this user id was not already gathered in the first phase. Finally time information is calculated for the created events using the time zones of users mapped to these events and the events are inserted into the database.

The plugin does this until there are no more write messages to process. Then it checks to see if there are more update datasets to process, since a lot of schemas present in the updates are not present in the writes, and if so it process those events. The implementation decisions in this algorithm rested in how to best do the search for write messages related to update ones. Other possibilities instead of the writes buffer used are inserting immediately the write messages and then do a query to see if there is a match. Or inserting all messages and then in the end do queries and change the update messages that got matched. Both alternatives rely on query times to be effective, which means creating and updating indexes. For large number of messages to process the solution implemented or the second alternative are probably the best since they don't require constant queries and index updating. However, even with the chosen approach, indexes had to be created anyway because of queries done in other plugins.

The possible parameters for this plugin are the length of the write events buffer and the bulk insert limit. The bulk insert limit is the number of inserts to the database done at once. This batch like insert type was adopted to try to further speed up the algorithm. After the algorithm run indexes are created for the user_id, timestamp, and user_NDI fields in this collection since these will be heavily queried in other plugins.

3.4.3 presenceFromDevices

This plugin is responsible to collect events that give information about presence. This information is a mixture of heuristics, for example device_updates from light bulbs that were independent, this probably means that the user is home. And sensor information, for example cameras with motion sensors. The plugin creates this information by doing a set of queries to the loadEventsToDB collection, and then generates a document per user with lists of events that each have a window of time and a value between 0 and 1 that correspond to certainness of presence. The events are divided in two types: continuous, gathered from devices like alarms which their state give a continuous source of presence information, and non-continuous like motion sensors or light bulb interactions which have a time window, heuristically defined, in which the occurrence of the event is relevant.

Adding to this information some users were given a chance to contribute with the user_NDI field. With this, additional presence information can be gathered in certain conditions and better tag the datasets.

As with the devicesPerUser this plugin starts by doing a query to its own collection in the database to retrieve the users that are already there. Then it does a list of queries to the loadEventsToDB collection, each one corresponding to a specific type of event we want to gather presence information from. The results of each query are passed to a function that processes them, accepting as parameters the results, the collection of users, the type of dataset, the name to give to this presence information, the presence value, the time window and if it's a continuous event.

This function will iterate through the results skipping those already processed according to saved timestamp, if information is already present for the user_id of the current result it will retrieve that user document from users dictionary, otherwise it will create a new user. Then it will create an event for this result, saving the original timestamp, the schema, the original value, and if it is a continuous event. It then creates an identifier using the profile, channel, component and

FIGURE 3.7: Partial example database document for the presenceFromDevices plugin

```
"presenceInfo" : [
    {
        "value" : 0.8,
        "endTimestamp" : NumberLong(1464152502667),
        "continuous" : false,
        "startTimestamp" : NumberLong(1464151902665),
        "decay" : 3.33333333333333e-006,
        "origValue" : "100.34",
        "origTimestamp" : NumberLong(1464152202677),
        "schema" : "weight-kg",
        "io" : "i"
    },
    {
        "value" : 0.8,
        "endTimestamp" : NumberLong(1464153204641),
        "continuous" : false,
        "startTimestamp" : NumberLong(1464152604641),
        "decay" : 3.33333333333333e-006,
        "origValue" : "100.34",
        "origTimestamp" : NumberLong(1464152904641),
        "schema" : "weight-kg",
        "io" : "i"
    }
],
"hasNDI" : false,
"hasNonContinuous" : true,
"processingTimestamp" : NumberLong(1464152904641),
"hasContinuous" : false
[...]
```

property id present in the result, this identifier is stored in a list in the user info, if it isn't already there, and is used after to exclude events with this identifier in other plugins.

If the result is of a continuous type the function will check its value and add a value to the event, 1 for home and 0 for away. It will then set the event start timestamp to the original timestamp and add it the presence info list of the user and the event list of the corresponding type. If the event list for that type is not empty it will set the last event end timestamp to this current one start timestamp.

If it is of a non-continuous type it will use the provided interval in the parameters of the query, heuristically defined, to calculate the event start and end timestamps from the original timestamp and then will add these events to the lists. The algorithm will also store the messages ids, simply because sometimes there are messages from the same device hierarchy with the same value and the

same timestamp, and this way we can avoid processing again the same messages if the plugin is run again or is in a continuous production service. Finally as with the devicesPerUser, the users marked has changed are updated on the database.

3.4.4 changePropertyUpdates

Depending on the type of device and its use cases sometimes the update messages generated by it may report repeated values. For example an alarm may create a lot of updates since it was activated up to the point it was turned off, however only two times the arm property changed. The same is true in the case of the read messages, several read messages may be sent for a property that hasn't changed in value meanwhile. Also for the write messages, repeated write messages for the same property with the same value are in some cases common, specially when they are triggered by rules. This plugin was created to mark the events that effectively report a property change instead of just broadcasting a repeated value, since this may generate more meaningful information about the messages. The idea is to save a state per property of each device and then check if an update message reports a new value or if a write message contains a new value. For the reads case basically the read is marked if there hasn't been another read since the property changed.

So contrary to the other plugins, which are more focused on the user_id for their operation, this one is oriented to a property device identifier composed by profile, channel, component, property and user ids. It stores a timestamp and an event per identifier that corresponds to the last update message from this device property. Then it will iterate through the events checking per identifier if the new message, an update or a write differs from the last one. If so the event is marked by creating the new field changeProperty with true value and then updating the loadEventsToDB collection, and the event is also stored in this identifier data so that it can be compared to the next.

FIGURE 3.8: Partial example database document for the changePropertyUpdates plugin

```
    "newReadMessage" : false,
    "data" : {
        "schema" : "https://ontology.muzzley.com/schemas/v1/status-onoff",
        "property_classes" : "com.muzzley.properties.status",
        "timestamp" : NumberLong(1464539043525)
        "component_label" : "Bulb",
        "dayOfWeek" : "Sunday",
        "value" : "False",
        "month" : "May",
        "trigger" : "independent_update",
        "property_name" : "Status",
        "hour" : 17,
        "day" : 29,
        [...]
    },
    "timestamp" : NumberLong(1464701245075)
    [...]
```

In the case of the read messages there is a Boolean variable associated with the identifier that is set to true each time the saved data changes and set to false when a read message for that identifier is processed. A message that is processed while the variable is true is updated in the loadEventsToDB collection with the changeProperty field, like in the update and read cases. This way only one read message is marked per property state change.

As with the loadEventsToDB, event comparison is done by comparing the dictionaries representing the messages, excluding certain fields, such as timestamp, etc. in order to see if the message reports new values or not. This is done because the fields in the message are dependent on the schema and this way works on all of them.

3.4.5 interactionStatsPerUser

This plugin is responsible for creating usage statistics for each user. It iterates through all database loaded messages and creates metrics per user with different levels of granularity, taking into account the type and time of day. The metrics are created by counting events and time passed, then averages are calculated using this information. Examples of created metrics are: average number of update

messages per hour, average number of write messages per weekend day, average number of independent update messages per hour on a weekday, etc. Metrics are also divided by if the counts concern messages marked as changeProperty.

FIGURE 3.9: Partial example database document for the interactionStatsPerUser plugin


```
"iWeekendCountCP" : 11,
"iWeekdayCountCP" : 74,
"rWeekendCountCP" : 8,
"userWeekendAvgDay" : 3,
"WeekdayAvgHour" : 0,
"iNightWeekdayindependent_updateCount" : 17,
"rWeekendAvgMin" : 0,
"userAvgDay" : 1,
"iDawnWeekdayindependent_updateCount" : 12,
"rAvgDay" : 1,
"rDawnWeekenduserCountCP" : 8,
"rCount" : 13,
"ruserAvgDay" : 1,
"iNightWeekdayindependent_updateCountCP" : 17,
"currentTimestamp" : NumberLong(1464715201302)
"totalCount" : 98,
"iAvgDay" : 5,
"iDawnWeekendindependent_updateCountCP" : 1,
"iNoonWeekdayindependent_updateCount" : 25,
"ruserAvgMin" : 0
[...]
```


This plugin starts by retrieving users' information from the devicesPerUser collection and then it will iterate through the users skipping those that don't have timezone information, also checking if there is already information for this user in this plugin collection and if not, it creates a new dictionary for it. For each user it will then do a query to the presenceFromDevices and if that yields a result it will then query loadEventsToDB collection to retrieve all his events, sorted by timestamp. This query is limited to results with timestamp greater than the saved one so that events are not processed twice. Then start and end timestamps are retrieved from the first and last events returned by the query, and the plugin starts iterating the results.

Firstly, if the exclude option is set, it will check the presence information for the identifiers and if the current one matches it will skip the event. Then it will use the dayOfWeek information to create a typeOfDay, Weekend or Weekday and the hour to create a timeOfDay, morning if between 7 and 13, noon if 13 and 19,

night if 19 and 1, and dawn between 1 and 7. After this it starts doing counts based on the event information. These counts take into account several levels of the Muzzley hierarchy and other information. For example, the first count is the type of dataset at this time of day in this type of day with this trigger. This is composed has a field name for the user dictionary, so a field for this type of count can be iMorningWeekendindependent_updateCount, in which "i" means it is from the updates dataset. An example of a different type of granularity of this count would be for example dataset and trigger, which for this case would compose the field iindependent_updateCount. A duplicate version of each type of metric is also created with the CP letters on the end of its name, meaning that this metric only counts events that were marked has changeProperty.

After a lot of different types of counts are done the plugin will then calculate days passed based on the difference between the first and last event timestamp. This is important to know how many week and weekend days passed to have statistics based on each. The algorithm then iterates the collected counts calculating the averages. In order to do this it calls a function that will discount time passed based on the properties of the count, for example if it is referred to weekend it will removed all time passed in weekdays and then it will divide the count with the remaining time to obtain the average. This is done in three levels, minutes, hours and days.

Finally, as with other plugins, users marked as changed will be updated in the database. The algorithm however does some simplifications. First when counting the days passed, even if a user only has a couple of events at the beginning of the day, that day is counted. Then when discounting the time, for example for time periods, time is simply divided by 4, the number of periods. And the same for time passed depending on week or weekend days. In the long run, if the user has normal activity throughout the days the simplifications won't distort too much the real values, but it is something to take into account. Doing the algorithm other way would probably mean a much more complicated solution with a slightly better precision that might not make much difference.

It is worth noting that the metrics marked as CP were a later introduction to the plugin and so they were not present in the initial experiments, but this will be stated again when explaining the experiments.

3.4.6 createMLDatasetGenStats

This plugin generates more immediate metrics that will serve as the main building block of the csvs for machine learning. Depending on the chosen mode it will either create a database entry per message from the loaded ones, or an entry per window of time. The metrics on this entry are for example: number of write messages in the past 5 minutes, number of independent updates in the past half hour, etc. This algorithm also creates information about presence per entry by querying the presenceFromDevices collection.

FIGURE 3.10: Partial example database document for the createMLDataset-GenStats plugin


```
"wLastMessageDiffCP" : 1292,
"totalDiffSchemasCountHalf" : 1,
"iindependent_updateCountHalfCP" : 1,
"ruleLastMessageDiff" : 0,
"rruleLastMessageDiffCP" : 835,
"ruleCountQuarter" : 5,
"iindependent_updateCountFive" : 1,
"ruleCountHalf" : 5,
"iLastMessageDiffCP" : 0,
"totalDiffSchemasCountQuarterCP" : 1,
"iindependent_updateLastMessageDiff" : 0,
"presenceStatusContinuous" : "n/a",
"totalCountQuarter" : 6,
"iDiffSchemasCountHalf" : 1,
"rDiffSchemasCountHalf" : 1,
"totalCountFiveCP" : 1,
"totalCountHalfCP" : 1,
"iCountHalfCP" : 1,
[...]
```


The plugin starts by retrieving information from the devicesPerUser collection, the interactionStatsPerUser collection and its own collection. Then it iterates each user_id to query the loadEventsToDB collection to retrieve the events that weren't processed yet, sorted by timestamp. It will then process them by window or by event depending on which mode has been selected.

If window mode is selected statistics are calculated each time we move to a new time frame. In order to do this the algorithm gets the last event timestamp and the current event timestamp is initialized with the first event time. It then enters a while cycle that continues as long as the current is lower than the last timestamp. Inside this while there is another cycle that essentially adds events to a list as long as the event timestamp is lower than the current timestamp and there are more events to process. This is the list that will be used to calculate this time window metrics. In this mode the time metrics are calculated from the window timestamp and the user timezone.

If event mode is selected, the events are iterated and simply added to the list to be used to calculate statistics, time metrics are gathered directly from the event. As with the interactionStatsPerUser, if event exclusion is turned on before adding to the lists in both window and event mode, those events are skipped if they were used to gather presence from. Then for each window or each event, another function goes through the selected events list. Since metrics are based in messages occurring in the last 1, 5, 15 and 30 minutes, the ones out this timeframe are removed. This function will then do counts to create the metrics using this list, for example, wuserFivecount, the number of user writes messages in the last 5 minutes.

Besides count related metrics very similar to the interactionStatsPerUser plugin, this plugin has 2 other types of metrics. Firstly there are the different schemas metrics. These count the number of messages that use a different schema per time window and granularity as the other counts. So for example if in the last 15 minutes there was an update message for an rgb bulb and an update message for an alarm that was activated, the metric iDifferentSchemasCountQuarter will be 2. And if for example another 2 messages for a different rgb bulb were processed, this metric will still be 2 since no different schema was counted. This is implemented by keeping a list of schemas, appending when a new one is processed and then retrieving the list length to create the metric value. The reason behind these metrics is that they represent indirectly the number and types of devices being

interact with. This may be very relevant for presence, for example somebody arriving home probably interacts with different type of devices in a very short time, switching lights on, changing the thermostat temperature, turning a cooking pot on, watering plants, etc.

The second type of different metrics are the last message difference. These basically hold in seconds how much time ago a message of a certain granularity was generated. So for example if the metric wRuleLastMessageDiff holds the value 5 it means that the last time a write message with the rule engine as trigger was seen was approximately (since its rounded from milliseconds) 5 seconds ago. This is implemented by storing the last event of each type, the difference in time is calculated between the current processing time and the last events of each type. The idea of these metrics is that even with the counts it may be useful to know when was the last time a message of a certain type occurred.

As with the interaction stats per user, for each of these types of metrics, the counts, the counts of different schemas and the last messages differences, an equal counterpart exists that holds the same type of information but only takes into account events that were marked as changedProperty.

Regardless of the selected mode, the last step for creating an entry in this plugin collection is retrieving information about presence. Values for presence are stored individually for each type of presence, non-continuous continuous and user_NDI, and an "n/a" is used when a user doesn't have that type. A total presence value is also stored. In order to do this a function iterates through the presence events of the user and retrieves the values of the events that have time windows in which the timestamp of the event/window fits. If more than one event is gathered the values are summed and if the value is bigger than 1 it is lowered to one. Finally the new entry is inserted in the database in a bulk operation, the same way it's done in the loadEventsToDB plugin.

The main parameters for the plugin is the mode: window / event and the size of the time window. Initially this plugin would also restrict the users from which

metrics would be calculated depending on their number of devices and average interactions, but both this function, and the possibility to create a dataset of users with specific type of presence information, were implemented in the createCSVsForPresence plugin. The reason for this is the time this plugin takes to run, in this way we need to run it fewer times.

The different schemas, the last messages differences and the metrics marked as CP were a later introduction to the plugin and so they were not present in the initial experiments, but this will be stated again when explaining the experiments.

3.4.7 createCSVsForPresence

This plugin uses the information generated in the others to create csv files that will later be used with machine learning algorithms. It creates a line of csv per createMLDatasetGenStats document, writing in the file the information generated in this plugin plus other like the ones presence in the interaction stats and devicesPerUser, such as: total number of devices of this user, total number of components, etc. It can also create datasets with some variations: only writing in the csv lines of users with more than X devices, only using information presence from continuous sources, etc.

FIGURE 3.11: Example database document for the createCSVsForPresence plugin

```
    "timestamp" : NumberLong(1464683282852),
    "user_id" : "t4bn98yar4",
    "lastIdsProcessed" : [
        ObjectId("57cdc2b8c1bec10834f3fdf7"),
        ObjectId("57cdc2b8c1bec10834f3fdf8")
    ]
```

It starts by retrieving information from the devicesPerUser, the interactionStats, the createMLDatasetGenStats and its own database collection. It then iterates the createMLDatasetGentStats information since each entry in this collection will correspond to a csv line. Per entry it will first check the eligibility of

this line according to the presence information available for the user, for example if a continuous only information dataset is to be created, entries of users who don't have that type of information will be skipped. The same is done in case minimums are set for devices or interactions averages, for example if minimum devices is 5 and a user only has 3, its entries will be skipped. Except when stated these minimums for tests are always 1. This means users need to have at least one device to be eligible, of course, but they also need to have at least 1 interaction per day as average.

After this it will check its own collection information to see which entries of this user were already processed using the timestamps. If the entry isn't skipped after all these checks it will be passed to a function that will create the csv line. This function will write the presence status and call other functions depending on the selected mode. The 0 mode will write in the line features related to the devicesPerUser plugin, such as number of devices of a user, and the createMLDatasetGenStats metrics. Mode 1 will write metrics from 0 mode plus the calculated statistics from the interactionStatsPerUser. Finally the mode 2 will write all the previous metrics plus the same ones but to the schema level granularity, but was not used at all in the dissertation.

The main difference between the MongoDB collections and the csvs is that in the database a metric will not be stored if its 0, for example if there is no update messages count in the last five minutes the document won't have that field. In the csv however all features must be there even if they are 0. In order to do this the plugin has lists has global variables that contain each part of the possible name combinations for metrics, for example the list ["w", "r", "i"] represents the types of message, and then it combines them and checks if the current document has that metric name as field, gathering the value if it has or writing 0 in case it doesn't.

Finally the line is inserted into a created csv file. Segmentation Engine functions are used that automatically take care of writing the csv headers. There is also a limit of lines per file, if the limit is reached a new file is created, this is to help making the created files more manageable to inspect, copy, etc.

3.5 Non Processed Schemas

Because of the nature of some devices certain schemas have a lot more entries than others. Typical examples are the update messages of devices like thermostats, noise level meters, and humidity meters. The number of messages generated by these devices is orders of magnitude greater than the others and thus heavily increase processing time. This led to the decision that for this project, we were not going to process the related schemas. This is a problem mostly with update and read messages, but for consistency they are also not processed in the write messages datasets.

However it is interesting to note that some of these types of devices may by themselves serve as a presence detection system, the noise levels for example may reveal important information about presence. It would have been interesting to use these messages in the presenceFromDevices plugin to try to retrieve more presence relevant events. However noise levels would have to be scanned and processed first in order to try to identify per house, what are the noise levels boundaries per class. This is not a trivial task, it would also not be very precise since we have no specific information about presence that we can use. The same is true for temperature levels, humidity levels and carbon dioxide levels, all of them would be very useful but is not an easy and straightforward task to include that related information here. But it is indeed very interesting future work that will complement and enhance the approach described in this document.

Chapter 4

Initial Experiments and Exploration

4.1 Initial Exploration

The development of the Segmentation-Engine and the creation of these initial datasets started at the same time as this project. So initially there wasn't much available information to work with. Regardless, initial csv datasets were generated with older versions of the described plugins and were very important to see the capabilities of the planned approach.

In order to do that Weka [26] was used because of its usability, automatic plotting and correlation of data features, and possibility to try a lot of machine learning algorithms and tools easily. Weka is a machine learning tool developed in the University of Waikato in New Zealand. It can be used as a library called directly from java code, with a simple terminal like interface, or with a graphical interface. The graphical interface has three modes explorer, where a dataset can be loaded and then filters can be applied, supervised or unsupervised algorithms can be used, feature selection can be done, etc. Experimenter, where a set of algorithms can be applied to a dataset, varying their parameters or other variables, to obtain meaningful results. And finally the KnowledgeFlow that is a way to set up machine learning experiments but using graphic objects to create the process flow.

This initial dataset was gathered from 3 days of Muzzley platform interactions. After running the plugins in different modes the generated csvs size ranged from 2MB to 300MB. This data was from roughly 900 users, but depending on the mode only the data from some users could be used. The results obtained in this phase showed that the approach is interesting but more exploration was needed. Tests were done using regression, treating presence as a continuous values, and discretizing the presence value gathered in the plugins into two or more classes. Values discretized to two classes became the norm because there aren't a lot of different values set for the presence related events, and they are heuristically defined so it didn't make sense to use regression or more classes.

An example result, gathering information only from continuous sources and discretizing into two classes, using a j48 decision tree, the Weka implementation of the C4.5 algorithm [27], and 10-Fold cross validation as evaluation method, we were able to achieve overall 99% of correctly classified instances. But for the same case, using non-continuous sources for presence gathering the results for the presence class were below 50%.

However as it will be discussed further along, we discovered the datasets must be separated by type of presence and further development was made on the plugins, so focus will be on the results of the next phase.

Feature selection algorithms were also used to eliminate most features, since with the described plugins over 300 features are generated but only 30 to 60 remained after applying feature selection filters. This step helped training the classifier faster, with better precision and to understand what information was more useful.

After this phase more data was available and Weka was switched for Scikit-Learn [28] in order to have faster and more capable algorithms, and also an easy way to build a machine learning pipeline. Scikit-Learn is a free software machine learning Python library that features several classification, regression and clustering algorithms. It uses and is prepared to work with NumPy and SciPy.

Having no previous experience with Python or Scikit-Learn we were motivated to learn Python to write the plugins described before, since the rest of the Segmentation-Engine was built with this language, and to use Scikit-Learn for the machine learning part because of its popularity and good results. Some extra considerations have to be taken into account, for example to use Scikit-Learn every feature has to be numerical, Weka instead could detect and assume String features as nominal. Despite the differences, adaptation was easy and two Scikit-Learn based Python script like files to do the machine learning experiments were created. These will be described in the next sections.

4.2 Machine Learning Scripts created

4.2.1 fullAlgorithm

The first script is called FullAlgorithm because it uses tools from all parts of the machine learning process. The program starts by using Pandas [29], a Python data manipulation library, to read from the created csvs. Then Pandas functions are called to transform the created structure to matrices supported by Scikit-Learn. After this the presence attribute is discretized to {0, 1} meaning unknown or presence. Feature selection is done choosing an algorithm from these: SelectKBest, with Chi2 [30] as the scorer function, LinearSVC [31] or a Random Forest [32], the last two cases are classifying algorithms which in their normal operation will rank the features in terms of their contribution to classification, so they can be used in this feature selection phase.

The Random Forest algorithm was heavily used throughout the dissertation, it is an ensemble method that consists in creating many decision trees and then use the mode of their classification or the mean prediction in the case of regression as the output. Each tree is built using a subsample of the instances and of the features. The idea is to use the powerful prediction capabilities of decision trees to build better models that suffer less from overfitting.

The next phase is parameter search, the classifier chosen, in this case a Random Forest, is fitted and tested to see which options for the classifier best suit the problem. This process also has 3 modes, the first is random parameter search, the second grid search and in the last both types of searches are used. Parameters are scored by testing the data with a 3-fold cross validation, and the best parameters are chosen. Finally the algorithms will be evaluated with a 5-fold cross validation for reference.

At the end of the process the chosen estimator will be stored to physical memory using the pickle Python library. The array of chosen attributes is also stored this way. A file is created since the beginning of the process that serves as log. This process is done with a small dataset of all the csv information generated so that heavy tasks, such as feature selection, and even heavier, parameter search, can be done, in a reasonable time, with several variations trying to find the best machine learning algorithm configuration.

4.2.2 usingSavedConfig

The second script is called usingSavedConfig, the objective of this one is to use the selected features and classifier from the FullAlgorithm to classify and test on a bigger dataset. In order to do that the algorithm loads the pickled feature array and starts reading from the dataset already excluding the non-selected features. Then the classifier is loaded and 10-fold cross validation is used to train and test it on the new data. These new algorithms were used with a month long gathered data from the Muzzley platform. The total data for each scenario ranged between less than 1 GB and 15GB, depending on the processing phase with the plugins. The results obtained will now be presented.

4.3 Initial Scikit-Learn Experiments

Due to the way the instances are tagged, the experiments are mainly divided between "excluding" and "not excluding". The “excluding” experiments are more difficult scenarios where the events that were used to calculate presence information are not taken into account in the counts and averages calculated by the plugins. This is done using an identifier that uses profile, component and property id. So no brightness events of a light bulb will enter the statistics if the brightness information of this bulb was used to gather presence information for a user in the heuristic labeling step. But if, for example, the color-RGB property of this bulb wasn't used for presence labeling, the events related to this will enter the statistics. Although the metrics used don't contain information about values and thus the algorithm can't use the same information used for tagging we are also doing tests with this setup to further explore the capabilities of the approach.

As stated before, the gathered presence value from the plugins was discretized to 0.0 or 1.0, and will be referred as unknown and presence from now on. It is important to note that for the non-continuous and the user_NDI datasets the 0.0 unknown class are instances that we didn't have information to tag as presence. While in the continuous dataset 0.0 unknown refers to instances that according to the gathered information there shouldn't be presence in the house.

The first experiments yielded very good results, over 0.9 precision and recall was achieved for both classes, excluding and not excluding labeling information. However, after further exploration we realized the algorithm was using the metrics to detect each user almost individually and then classifying according to if the user has mostly presence tagged instances or unknown ones. This was possible because users with devices that contribute with continuous type information had much more presence tagged instances than the ones which only had non-continuous type information.

This led to the conclusion that for now the information gathered from each separated type of event: non continuous sources, ex: light bulbs, continuous sources,

for example: alarms, and user_NDI shouldn't be all gathered in the same dataset. So the experiments were repeated with different types of datasets.

Results will now be presented stating the difference between them, depending on whether events were excluded or not, and which type of events were used to gather presence information from. These results are obtained with the process previously explained in the Scikit-Learn Experiments section with the difference that the parameter search phase was not done due the time this phase takes. Also the features created from the changeProperty marked events, the different schemas features and the last message difference features weren't already implemented yet.

The next table shows the selected features for the two first presented scenarios. This is just to complement the results since further discussion of the features will be done in section 4.4 and in Chapter 5.

TABLE 4.1: Features used in first two results that will be presented

experiment	**features**
window non-continuous not excluding	hour, minute, typeOfDay=Weekday, typeOfDay=Weekend, timeOfDay=Morning, timeOfDay=Noon, timeOfDay=Night, timeOfDay=Dawn, totalDevices, individualUnits, differentSchemas, rruleCountMin, rruleCountFive, rruleCountQuarter, rruleCountHalf, iindependent_updateCountMin, iindependent_updateCountFive, iindependent_updateCountQuarter, iindependent_updateCountHalf, rCountFive, rCountQuarter, rCountHalf, iCountFive, iCountQuarter, iCountHalf, totalCountMin, totalCountFive, totalCountQuarter, totalCountHalf, wWeekdayAvgDay, rWeekdayAvgDay, rWeekendAvgDay, iWeekdayAvgDay, iWeekendAvgDay, wruleAvgDay, ruserAvgDay, ruserAvgHour, rruleAvgDay, iindependent_updateAvgDay, iindependent_updateAvgHour, rAvgDay, iAvgDay, iAvgHour, totalAvgDay, totalAvgHour
window non-continuous excluding	hour, minute, typeOfDay=Weekday, typeOfDay=Weekend, timeOfDay=Morning, timeOfDay=Noon, timeOfDay=Night, timeOfDay=Dawn, totalDevices, individualUnits, differentSchemas, wruleCountHalf, rruleCountFive, iindependent_updateCountMin, iindependent_updateCountFive, iindependent_updateCountQuarter, iindependent_updateCountHalf, wCountQuarter, rCountFive, rCountQuarter, rCountHalf, iCountFive, iCountQuarter, iCountHalf, totalCountMin, totalCountFive, totalCountQuarter, totalCountHalf, rWeekdayAvgDay, rWeekendAvgDay, iWeekdayAvgDay, iWeekendAvgDay, ruserAvgDay, rruleAvgDay, iindependent_updateAvgDay, rAvgDay, iAvgDay, totalAvgDay

In the first new experiment events were not excluded and only presence gathered from non-continuous events was used:

TABLE 4.2: Classification report using non-continuous events for labeling not excluding any information from the metrics

classes	**precision**	**recall**	**f1-score**	**support**
unknown	1.0	1.0	1.0	4979672
presence	0.87	0.49	0.62	20330
avg/total	1.0	1.0	1.00	5000002

TABLE 4.3: Confusion matrix for the same scenario of Table 4.2

actual classes	**classes predicted**	
	unknown	**presence**
unknown	4978142	1530
presence	10427	9903

As with the initial experiments, we have now removed the events that were used to collect presence information from the statistics and repeated the experiment with the previous dataset:

TABLE 4.4: Classification report using non-continuous events for labeling and excluding information used for labeling from the metrics

classes	**precision**	**recall**	**f1-score**	**support**
unknown	1.0	1.0	1.0	4980543
presence	0.54	0.14	0.22	19459
avg/total	0.99	1.0	1.00	5000002

TABLE 4.5: Confusion matrix for the same scenario of Table 4.4

actual classes	**classes predicted**	
	unknown	**presence**
unknown	4978246	2297
presence	16726	2733

Then the same experiment was done but this time a minimum of devices, schemas and interaction per day was enforced. And then a maximum limit of these features was used. So only users who fulfilled these conditions were in the dataset. This tested if the model would behave better when there was information from more or less devices available. The results weren't better, recall dropped to 0.12 and 0.13 respectively. Precision however climbed to 0.60 in the case of the maximum devices, which suggests the algorithm is more certain of presence cases with less devices.

In order to use the rest of the presence information, datasets were created separately for users who had continuous information and user_NDI information. For the continuous case:

Table 4.6: Classification report using continuous events for labeling and not excluding any information from the metrics

classes	**precision**	**recall**	**f1-score**	**support**
unknown	0.72	0.62	0.67	412795
presence	0.83	0.89	0.86	884421
avg/total	0.80	0.80	0.80	1297216

Table 4.7: Confusion matrix for the same scenario of Table 4.6

actual classes	**classes predicted**	
	unknown	**presence**
unknown	257197	155598
presence	100776	783645

And with the user_NDI information:

TABLE 4.8: Classification report using user_NDI for labeling and not excluding any information from the metrics

classes	**precision**	**recall**	**f1-score**	**support**
unknown	1.0	1.0	1.0	441060
presence	0.67	0.39	0.49	2343
avg/total	1.0	1.0	1.00	443403

TABLE 4.9: Confusion matrix user_NDI not excluding for the scenario of Table 4.8

actual classes	**classes predicted**	
	unknown	**presence**
unknown	440618	442
presence	1435	908

These were then reproduced again but excluding the events used to gather presence information. For the continuous case the results remained mostly the same. For the user_NDI case recall dropped quite significantly reaching 0.08. Finally, we experimented with a set of users who had all types of information and this experiment achieved good results, similar to the previous presented in Table 4.8.

After this, all these variations were tested again but with the difference that each line of the csv was created per event instead of using a time window. In the non-continuous dataset this approach lowered the scores overall but raised recall in some cases. For the continuous cases this approach also lowered the obtained scores. In the user_NDI dataset though, especially with the excluded events dataset, the results had a significant increase. Finally for the dataset of users who had all types of information the results remained practically the same.

The number of users present in each dataset varied greatly, for the non-excluding datasets: the non-continuous has 1220 users, the continuous 189, the

user_NDI 56, and only 5 users had information of all types. For the excluding cases the non-continuous has 778 users, the user_NDI 26 and for the rest numbers remain the same. The difference in number of instances of each class was also very big, mostly just the continuous dataset had more lines of the presence class, for non-continuous and user_NDI in most cases the unknown class was more present as the showed results illustrate.

Despite these differences in the datasets and the results obtained, the many experiments gave us an insight to the problem. As expected the algorithm as a hard time identifying possible presence for the non-continuous case. Even counting in the metrics the messages that were used to determine the user's presence and build the training-set's labels, is not a guarantee of a high accuracy, which means it probably makes sense to continue counting these instead of handicapping so heavily the training data just to assure the model generalizes the rule. For the other cases results were quite good, except in the user_NDI only dataset were it didn't have a very good recall, probably because this is the scenario closest to explicit information and so its more difficult.

4.4 Feature Selection

Initial tests with the three techniques for feature selection described in the scripts section showed that there wasn't a significant difference in performance between them. Also when using the SelectKBest algorithm with the Chi2 metric, scores didn't vary much by choosing the 100 best features or the 50 best. These results coupled with the fact that using a Random Forest's feature importance values is a fast and more adaptable method for each scenario, since the number of attributes chosen is based on their scores instead of an initial argument, led to the adoption of this technique for all the tests, including the ones already presented.

The large number of features is due to the different possible granularity and different time period counts, so the main objective with feature selection was to try to find out what type of metric specificity worked best.

TABLE 4.10: Top feature importance values of some of the initial results according to the Random Forest algorithm

experiment	**top features**
window non-continuous not excluding	(minute,0.162), (hour, 0.128), (iindependent_updateCountFive, 0.074), (totalCountFive, 0.059), (iCountFive, 0.044), (iindependent_updateCountQuarter, 0.043), (iCountHalf, 0.029), (differentSchemas, 0.026), (rruleCountHalf, 0.025), (iCountQuarter, 0.022),
window continuous not excluding	(hour, 0.121), (minute, 0.074), (individualUnits, 0.047), (iWeekdayAvgDay, 0.038), (iindependent_updateAvgDay, 0.038), (iAvgDay, 0.03), (rWeekdayAvgDay, 0.028), (rAvgDay, 0.028), (typeOfDay=Weekend, 0.027)
window user_NDI not excluding	(minute, 0.144), (hour, 0.141), (totalCountFive, 0.06), (rCountFive, 0.03), (rruleCountFive, 0.027), (wCountFive, 0.027), (wruleCountFive, 0.025), (totalCountHalf, 0.022), (iAvgDay, 0.022), (iWeekendAvgDay, 0.021)
window non-continuous excluding	(minute, 0.296), (hour, 0.267), (individualUnits, 0.024), (totalCountFive, 0.016), (totalCountQuarter, 0.016), (totalCountHalf, 0.016), (typeOfDay=Weekend, 0.015), (totalDevices, 0.015)
window continuous excluding	(hour, 0.122), (minute, 0.075), (rWeekdayAvgDay, 0.039), (iWeekendAvgDay, 0.039), (individualUnits, 0.036), (iWeekdayAvgDay, 0.036), (iindependent_updateAvgDay, 0.033), (iAvgDay, 0.033), (totalAvgDay, 0.033), (typeOfDay=Weekday, 0.03)
window user_NDI excluding	(minute, 0.432), (hour, 0.387), (typeOfDay=Weekend, 0.025), (typeOfDay=Weekday, 0.015), (timeOfDay=Noon, 0.009), (totalCountFive, 0.009), (totalCountHalf, 0.009), (timeOfDay=Night, 0.008), (timeOfDay=Dawn, 0.008), (iCountFive, 0.008), (iCountHalf, 0.008)

As Table 4.10 shows selected features depend on the type of dataset in question and if the dataset excludes or not the events used to gather presence. By further examining the Table and the complete output of the algorithms some remarks can be done:

- Hour and minute were in most cases the top relevant features. However, when removing them a lot of tests achieved better performance, in the order of 0.1, and in the rest the drop in performance was only of 0.1. For the hours its normal that a pattern related to the habits of users exists and it may sometimes lead the algorithm to, however for minutes this is more

strange and may be connected to the periodicity in the generation of some messages, or with the mechanisms of the rules engine.

- For the averages created in the interactionStatsPerUser plugin the most relevant usually were the hourly and daily ones. The most selected from these time periods were usually the total ones, totalAvgHour and totalAvgDay, but type of dataset (reads, writes, updates) per type of day and type of dataset per type of day per trigger were also highly chosen.

- In terms of specificity metrics that counted by type of dataset and type of trigger were the most significant.

- In the count metrics from createMLDatasetGenStats the quarter and half hour periods were the most relevant. But for the non-continuous and the user_NDI datasets the minute and five minutes counts were important.

- Results for the event approach are not present in the table but they mirror the observed pattern with some differences: the main one is that in all scenarios of the continuous dataset there are higher scores for the count features instead of average ones. Other differences are more count features scores for the excluding cases, and slightly different, for example write related instead of read, counts in all experiments.

Despite these points since the feature selection phase is not very time consuming, and the training and validation with the selected features takes an acceptable time, we decided to continue outputting all possible features and let the selection algorithm reduce the complexity. Except for hour and minute that are also outputted but then excluded in the load phase in the results of the next chapter.

4.5 Addressing the Imbalance Issue

Because of the way presence information is created it is normal that the datasets that use non-continuous events and the user_NDI information have significantly

more instances of the unknown classes than the presence ones. The classes are not represented equally, this means our datasets are imbalanced. It is very common for a dataset to not be completely balanced, however in this case the numbers of each class vary greatly and so it is a problem for training the algorithm since it will favor the class with more instances. It is also a problem for performance metrics, since overall precision and recall don't reflect well the classification quality, but this is something we took into account from the beginning and the reason why the results analysis focuses on the scores obtained in each class.

A lot of techniques and procedures can be applied to try to mitigate this dataset imbalance, examples of these are: gathering more data, which in this case wouldn't change the distribution, generating synthetic samples, it could help in these datasets but since the class tagging is done heuristically we chose to not use any other method that could distort the results. A more suitable algorithm can be chosen, but random forests is already a good algorithm for imbalanced datasets. The data can be resampled, meaning changing the instances that go into the train and tests sets. Weights can be applied to classes or samples, etc.

So in order to try to cope with these differences between number of instances of each class, we have tried two techniques available in Scikit-Learn and two other ones available in the Imb-Learn [33], Imbalanced Learn, an independent library that extends Scikit-Learn with a lot of different re-sampling techniques.

First we tried creating the classifiers in scikit with different class weights, this means the classifier will be more penalized making mistakes of a certain regarding with another. So in our case we tried the weights 1 for the 0, unknown class, and 5 for the 1, presence class. Next we tried a very similar technique: sample weights, the difference here is that each sample has its own weight instead of the sample being set by class but in theory, if samples of the same class have the same weight, the techniques should have the same results, and indeed they had. Then we tried using other weights and the results will be presented in Table 4.11 and discussed after it.

Then we resorted to the Imb-Learning library that offers re-sampling functions such as down-sampling (or under-sampling), over-sampling and the SMOTE [34] Synthetic Minority Over-sampling Technique . As stated, re-sampling means changing the composition in terms of instances of the train and test sets. Down-sampling means that the dataset will be downsized approximating the number of instances of each class to the numbers of the less common class. So this usually means discarding instances of the majority class. If we have for example 1000 instances, 900 of the 0 class and 100 of 1, a normal 1:1 down sampling would mean 200 instances in the final dataset, 100 of the 0 class and 100 of the 1. Over-sampling means that the number of instances of the minority class will be augmented to the number of the majority class, in a simple over-sampling this is done by repeating instances. So in the last example the final dataset would be composed of 1800 instances, 900 from the 0 class and 900 from the 1 class.

SMOTE is one of the more complex resampling techniques available in the Imb-Learn, it is an oversampling technique that instead of repeating samples from the minority class it creates new synthetic samples. These types of more advanced techniques have reported to achieve very good results, but we had problems trying them because of the size of the dataset and the number of features. Besides, as stated, before we prefer not to work with synthetic samples.

So experiments for the most interesting cases were done adding an additional re-sampling phase with Imb-Learn that was either random down-sampling, or random over-sampling. Random in this case means that in the down-sampling case the majority class samples are randomly selected, and in the over-sampling that the repeated instances are also random. Scores improved significantly when resorting to these techniques. Table 4.11 will show the obtained performance with each one for the case of the window based approach using information from the user_NDI for the presence values.

From these experiments the techniques that have shown more potential are down-sampling and over-sampling, with the second one having slightly better results. Class and sample weights struggle because of the great difference in number

TABLE 4.11: Comparison of classification reports for several techniques to cope with the imbalance problem

technique	classes	precision	recall	f1-score	support
base	**unknown**	1.00	1.00	1.00	441060
	presence	0.40	0.33	0.36	2343
	avg/total	0.99	0.99	0.99	443403
classe weight: balanced	**unknown**	1.00	0.84	0.91	441060
	presence	0.02	0.55	0.03	2343
	avg/total	0.99	0.84	0.91	443403
sample weight: 5	**unknown**	1.00	1.00	1.00	441060
	presence	0.36	0.38	0.37	2343
	avg/total	0.99	0.99	0.99	443403
sample weight: 200	**unknown**	1.00	0.83	0.90	441060
	presence	0.02	0.58	0.03	2343
	avg/total	0.99	0.82	0.90	443403
sample weight: 400	**unknown**	1.00	0.72	0.83	441060
	presence	0.01	0.67	0.02	2343
	avg/total	0.99	0.72	0.83	443403
sample weight: 2000	**unknown**	1.00	0.59	0.75	441060
	presence	0.01	0.72	0.02	2343
	avg/total	0.99	0.60	0.74	443403
class weight: 1000	**unknown**	1.00	0.62	0.77	441060
	presence	0.01	0.71	0.02	2343
	avg/total	0.99	0.62	0.76	443403
down sampling	**unknown**	0.75	0.80	0.77	2343
	presence	0.78	0.73	0.75	2343
	avg/total	0.76	0.76	0.76	4686
over sampling	**unknown**	0.85	0.81	0.83	441060
	presence	0.82	0.86	0.84	441060
	avg/total	0.84	0.83	0.83	882120

of samples, since much greater weights are required to increase scores in the presence class that then result in lowering the performance in the unknown class. Advanced setups could have been tried, for example a different percentage of re-sampling, instead of 50%/50% and then applying different weights to the classes, but since results improved significantly with the re-sampling tests, these hypothesis were not further explored.

Taken only the scores into consideration the obvious choice would be to pick

over-sampling for further improvement of the scores in the rest of the project. However, this technique has problems in our case that are not easily represented here. The first is that this technique creates additional copies of instances from the minority class in order to have a 50%/50% distribution, but since these scores were obtained using cross validation, this means that the high performance might be due to overfitting, since for each fold there were probably instances in the training and test sets that were the same. The second, and more important aspect, because the other could be circumvented, is that we are working with big datasets that take a lot of computational resources to process and over-sampling is a technique that in this case heavily increases the number of instances to process. And so, down-sampling was chosen has the technique to further try to improve the results obtained from now on.

4.6 Results Exploration

The idea for this phase was to explore the results obtained so far and the values distribution in the datasets to find ways to improve the classification. Although this didn't contribute significantly with improvements to the algorithms, work done and tools used will be described in this section to provide further insight into the problem and to serve as example for this type of exploration. An extra Python script was created, called dataVisualization, again using Scikit-Learn and Pandas, very similar to the fullAlgorithm, but with some different capabilities.

The first one is to output the dataset with the extra predicted column that contains the values that the algorithm has predicted for every instance, and another column called Correct, that hold the values "no" or "yes", depending on the correctness of prediction. The purpose of this is to try to manually find patterns that lead to the failure or success of the classification. This was done for small cases to be able to open the generated CSVs in Microsoft Excel and use tools available in this program like column sorting to better analyze the data. Some interesting insights from these tasks are:

- Metrics of average statistics still feel like a way to identify each user, although this is mitigated by the randomness and the way algorithms like the random forest work. However as seen in the feature selection section the classifiers takes more than these features into account.
- A lot of the features don't provide extra information, especially in the case of the event based mode when a lot of messages are generated in a short period of time.
- As expected it is very difficult to understand the algorithms prediction just with this examination.

The next step was similar, the predicted dataset was output but this time resorting to TreeInterpreter [35], a Python package that enables the decomposition of predictions from scikit's Decision Tree or Random Forests. The output is composed of an array of values per features, with length equal to the number of classes, in this case two. These values represent the contributions of this feature to the decision of prediction. The final column is the prediction, an array with the total value of prediction per class. This is a very interesting concept but difficult to analyze and in the end it didn't result in new conclusions or ideas.

A Decision Tree and the trees created with the Random Forest algorithm are composed by rules in their nodes which represent divisions in the data depending on the values of features. After the creation of the model we can extract these rules to try to visualize the generated tree. So a Decision Tree was used for simplicity and an extra step was created to extract to output the tree like structure of rules. The files created however are quite complex and difficult to follow, so no insights were gained with this process. An extract of the generated output is presented next:

FIGURE 4.1: Example of generated tree structure

```
if ( iWeekdayAvgDay <= 18.5 ) {
                if ( rruleAvgDay <= 5.0 ) {
                if ( iCountFive <= 30.5 ) {
                if ( ruserCountHalf <= 297.5 ) {
                if ( rWeekdayAvgDay <= 78.5 ) {
                if ( iWeekendAvgDay <= 0.5 ) {
                if ( iindependent_updateCountQuarter <= 0.5 ) {
                if ( minute <= 55.5 ) {
                if ( totalCountMin <= 94.5 ) {
                if ( hour <= 16.5 ) {
                if ( minute <= 50.5 )
                if ( hour <= 1.5 ) {
                if ( totalDevices <= 4.0 ) {
                if ( minute <= 48.5 ) {
                return [[  0.  54.]]
                } else {
                if ( rruleCountHalf <= 2.5 ) {
                return [[ 1.  0.]]
                } else {
                return [[ 0.  2.]]
                }
        [...]
```

Finally the visualization package of pandas was used to create graphs to better analyze the results. Some of them will now be presented, taken from a test with the event based approach using continuous events for tagging excluding messages used to gather presence from the metrics and with no down-sampling:

This first graph compares the number of different schemas among devices the user has to the number of correct and incorrect predicted instances. As can be seen the number of correct predictions grows slightly as the number of different schemas does. So it can be said that the number of different types of devices, or devices with several different types of properties influences the positively on the prediction. This isn't true for all scenarios, for the user_NDI datasets for example, this feature almost didn't influence classification.

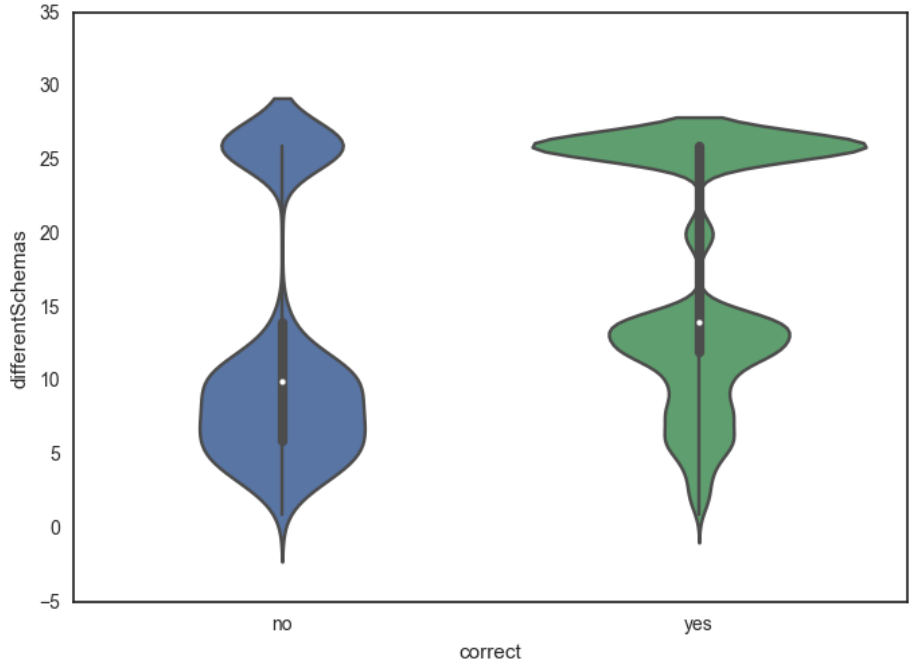

FIGURE 4.2: Violin graph of the number of different schemas between devices users own vs number of correctly predicted instances

In the next one, figure 4.3, the metric totalAvgDay is confronted with the correct number of predictions. This metric is an average number of messages this user and his devices generate per day. As with the other metric, a higher number means better prediction, and a number close to 0 increases heavily the number of incorrect instances. This behaviour is more consistent among scenarios.

The metric totalCountHalf, that holds the number of interactions counted in the last half hour is the next one to be compared to the number of correct predictions in figure 4.4. A higher value means just some slightly better results in some parts of the curve. Another thing to note is that when values go over approximately 750 all instances are correctly classified.

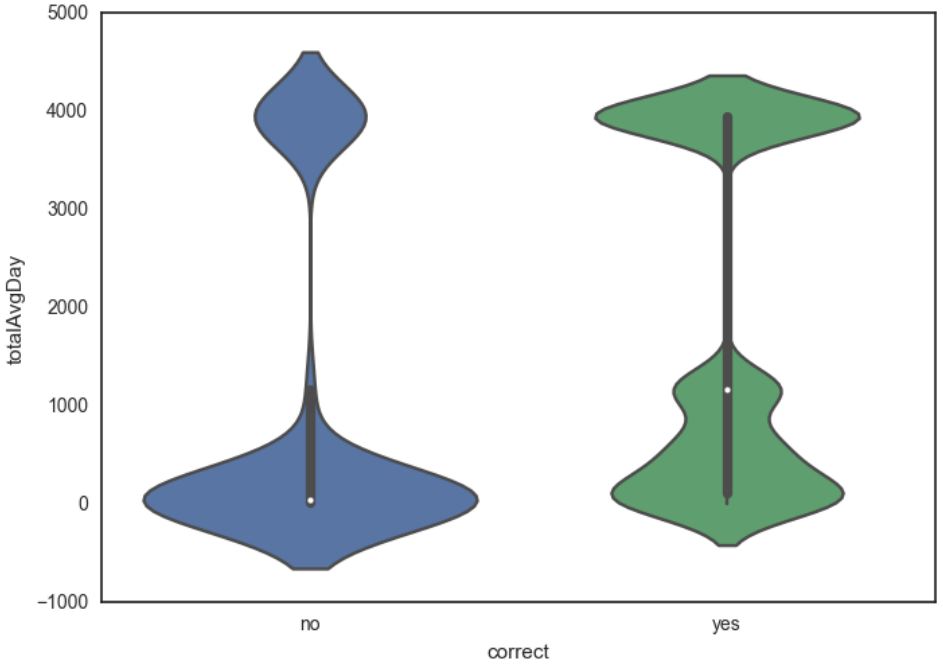

FIGURE 4.3: Violin graph of the totalAvgDay metric vs number of correctly predicted instances

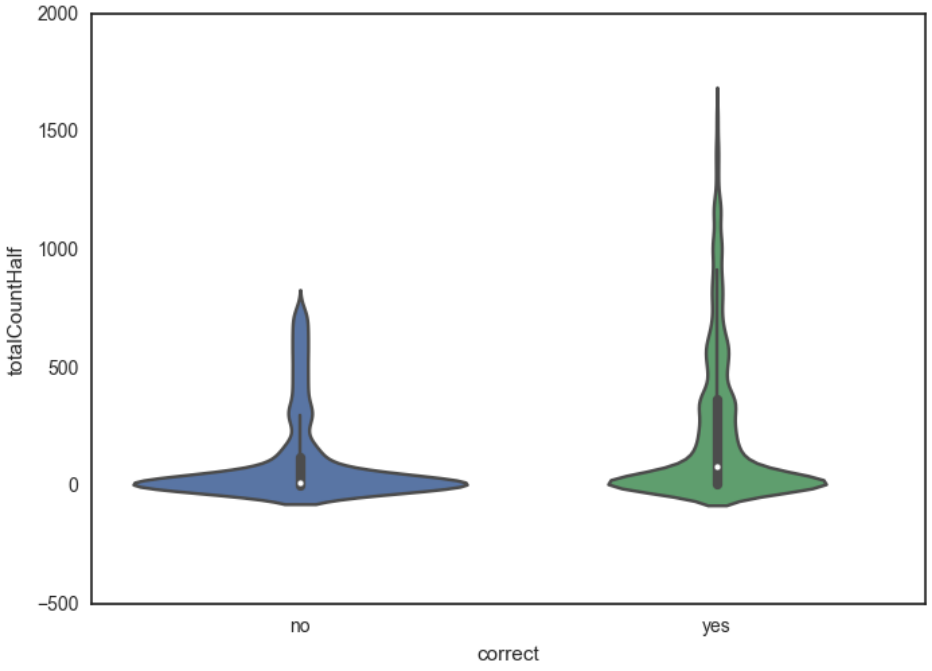

FIGURE 4.4: Violin graph of the totalCountHalf metric vs number of correctly predicted instances

Following the trend for this scenario a higher number of devices means more correct classifications as can be seen in figure 4.5. But this is not true for all scenarios.

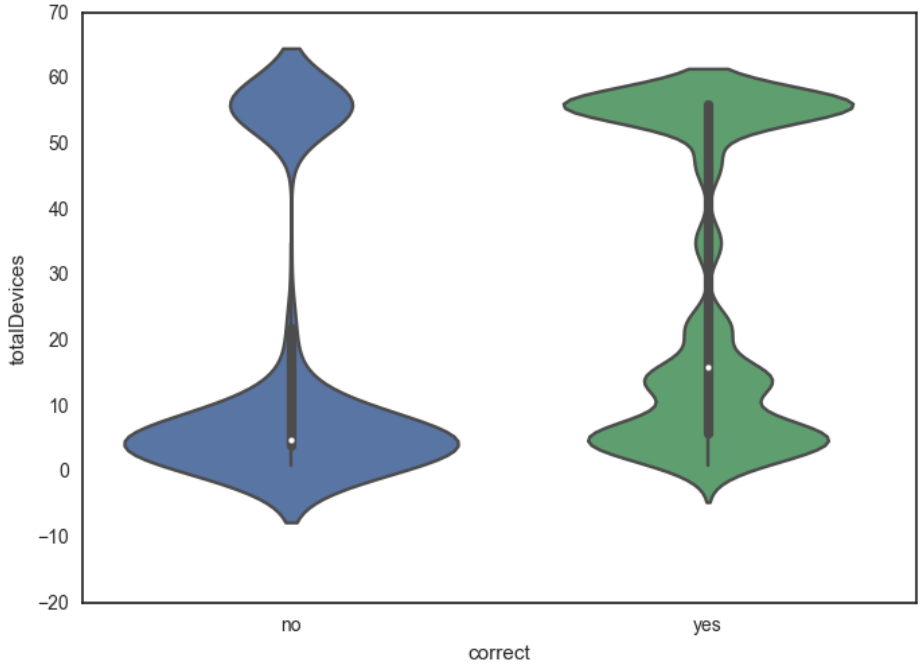

FIGURE 4.5: Violin graph of the total number of devices users own vs number of correctly predicted instances

It is difficult to draw general conclusions since depending on the scenarios some features influenced more or not, which means for now it is interesting to keep using all features and resorting to feature selection. It seems things depend mostly on the type of dataset, as concluded before, and generally, more devices and more messages, meaning more information available, leads to better results with the continuous type, while for the user_NDI and the non-continuous its the opposite.

These types of analysis are very interesting but in our project didn't lead to practical changes to improve the results. The analysis of the prediction output, the features contribution and the trees structure were difficult to analyze and draw conclusions, the graphics were more clearer although covering all scenarios and features with them would take a lot of time. Reflecting on the conclusion of the previous paragraph, in the future this type of analysis may serve for example to

understand from which number of devices, different devices and usage would the application be able to provide a presence indicator with good confidence.

4.7 Exploration with Other Algorithms

Although in the very initial experiments with Weka several algorithms were tested to find the most suited for the problem, the switch to Scikit-Learn brought more computational efficiency and access to some other algorithms or different implementation of the ones in Weka. And so it made sense to again try the datasets with a set of algorithms to see if there is one which naturally was more suited for our objective.

The typical, or more popular, ones available in Scikit-Learn were chosen, and then an additional algorithm was installed and tried: XGBoost [36] , an highly optimized version of the gradient boosting algorithm that has been gaining a lot of attention in the recent years because it was used to win a lot of machine learning competitions, and used in production by a lot of companies. The main idea of the algorithm is to create an ensemble of many weak prediction models, in this case decision trees, and combine them into a strong model using boosting, an iterative process to improve performance based on a cost function.

Besides this an extra technique was tried called Voting, a meta-classifier available in Scikit-Learn that takes a list of classifiers as input and then the idea is that the prediction is chosen as the majority prediction of the ensemble of classifiers. The voting ensemble in this case consisted in Logistic Regression, LinearSVC, adaBoost and XGboost.

A table will now be presented that will exemplify results obtained for the window mode using user_NDI for presence information, already with the downsampling phase active:

TABLE 4.12: Comparison of classification reports using different classification algorithms

algorithm	classes	precision	recall	f1-score	support
decision tree	**unknown**	0.76	0.81	0.78	2343
	presence	0.79	0.74	0.77	2343
	avg/total	0.78	0.77	0.77	4686
random forest	**unknown**	0.73	0.84	0.78	2343
	presence	0.81	0.68	0.74	2343
	avg/total	0.77	0.76	0.76	4686
logistic regression	**unknown**	0.68	0.87	0.76	2343
	presence	0.82	0.59	0.69	2343
	avg/total	0.75	0.73	0.73	4686
LinearSVC	**unknown**	0.69	0.87	0.77	2343
	presence	0.82	0.60	0.69	2343
	avg/total	0.75	0.73	0.73	4686
knn	**unknown**	0.80	0.62	0.70	2343
	presence	0.69	0.85	0.76	2343
	avg/total	0.74	0.73	0.73	4686
extra trees	**unknown**	0.74	0.82	0.78	2343
	presence	0.80	0.72	0.76	2343
	avg/total	0.77	0.77	0.77	4686
adaboost	**unknown**	0.71	0.84	0.77	2343
	presence	0.80	0.65	0.72	2343
	avg/total	0.76	0.75	0.74	4686
xgboost	**unknown**	0.75	0.80	0.77	2343
	presence	0.78	0.73	0.75	2343
	avg/total	0.76	0.76	0.76	4686
voting	**unknown**	0.68	0.93	0.78	2343
	presence	0.89	0.56	0.69	2343
	avg/total	0.78	0.74	0.74	4686

It is interesting to note that there isn't an algorithm that can be singled out as much better than others for this problem. As expected, and since it was chosen before because of the Weka results, the random forest algorithm has a very good performance, but also the extra trees and, surprisingly, the decision tree. In most algorithms tested in this phase randomness plays a part, which means further tests could show variations that would probably not be very significant. Also most of them have important parameters that require heavy tuning in order to perform better or to adapt to a certain problem. Here the default parameters were used, since a lot of time and effort would be necessary to tune all the algorithms for all scenarios, but as will be discussed in the next section, parameter tuning wouldn't probably change the conclusion obtained in these tests.

However the purpose of these tests was not to find the perfect combination of algorithm and parameters, it was about checking if there was a specific algorithm that behaved exceedingly well for our case. These experiments were then done for other cases: presence from continuous, from non-continuous, event based approach, etc. The results were similar, Random Forest and XGBoost took the lead, with occasional rivaling results from the Decision Tree and LinearSVC, depending on the scenarios. There was no significant result that would compensate to have a different classifier for a specific scenario so the conclusion was to either continue using the Random Forest algorithm or switch to the XGBoost. XGBoost was chosen, not because it achieve much better results than the random forests but because it uses better the computational resources available, taking less memory to run, and optimizing the computation by using all available cores efficiently (Scikit's default parallelization settings require copying the dataset per core used).

4.8 Parameter Search

Parameter search was designed as an important step in the tests in order to try to achieve better performance. For the very first tests, the ones, not shown in this document, were it was found that datasets should be created separately for

each presence source, parameter search was done, using the techniques described in the 4.2 section. The results however in these tests didn't improve much by the parameter search phase, and because it was a heavy processing task, it was skipped in the other tests until this point.

In spite of the initial results of parameter search, we tackled this section again and searched for more parameter variations and possibilities for both Random Forests and XGBoost. The number of variables that can be tuned in both algorithms is considerably large, and it would be very intensive work to understand them all completely and manually tune them for each scenario. So as before, the tests were done using GridSearch and RandomSearch, with values gathered from available literature.

As an example, the tests done on the window mode with presence gathered from user_NDI and down-sampling showed a 0.2 increase in precision for the avg f1-score with the Random Forest algorithm and a 0.1 increase for the XGBoost algorithm, with no notable difference in recall for both. But these were obtained after 20 to 50 minutes of parameter search, depending on the number of parameters and values to test. It was then concluded that for this project the potential performance increases of the parameter search phase versus the required tuning and, or, searching time, did not produce enough improvements to justify its use since there are a lot of scenarios to test, and thus this phase was skipped from this point on.

Chapter 5

Final Experiments

These final experiments were obtained by applying what was learned in the tests described in chapter 4, by using newly engineered features and by resorting to a new validation type.

New datasets were created with the difference in time since last message and the number of different schemas from the createMLDatasetGenStats plugin, and with the changeProperty related features from both creatMLDatasetGenStats plugin and the interactionStatsPerUser one. These features were described in chapter 3, on their respective sections.

For classifying algorithm XGBoost was chosen, because of the reasons stated in Section 4.7, and so, attribute selection was also changed to use feature importances from an XGBoost algorithm, reducing the now approximately 1000 features to usually around 50 to 100. As for dealing with imbalance in the datasets, down-sampling was the technique used. For the final validation a new script was created. The idea was that even though down-sampling validated with cross-validation is very effective, it doesn't represent the original imbalanced problem and thus doesn't represent the actual capabilities of the classification with this type of heuristic tagging. What this new script does is split the data into two equally stratified parts, which means they retain the original percentage of class distribution, one with 70% of the total instances that will be transformed with

down-sampling and used to train the algorithm, and the other with 30% of the instances that will be used as test data without down-sampling.

Tables will now be shown with results obtained using this configuration. One table per each mode, window and event based, will be presented with each entry corresponding to a type of dataset depending on the source of presence used. Another set of tables will then be presented for the same scenarios but in which events that were used to gather presence are excluded for the metrics.

5.1 Not Excluding Information from the Metrics

As with the initial results the number of users depending on type was: 1220 for the non-continuous, 189 for the continuous and 56 for user_NDI.

TABLE 5.1: Results for each type of dataset obtained using 70% for training with down-sample and 30% for testing using the window based approach and without excluding any information from metrics

type of presence data	classes	precision	recall	f1-score	support
non-continuous	**unknown**	1.00	0.95	0.98	2915601
	presence	0.08	0.92	0.15	12951
	avg/total	1.00	0.95	0.97	2928552
continuous	**unknown**	0.86	0.90	0.88	123839
	presence	0.95	0.93	0.94	265326
	avg/total	0.92	0.92	0.92	389165
user_NDI	**unknown**	1.00	0.92	0.96	132318
	presence	0.06	0.88	0.11	703
	avg/total	0.99	0.92	0.96	133021

Starting with the non-continuous dataset, compared with the initial results there was a significant drop in precision for the presence class to almost zero, but recall increased to almost double. Results for the unknown class remain practically

the same. The f1-score also dropped significantly due to these changes. Which might suggest these results are worst than before, however, the recall increase is very important because it means the algorithm can detect almost all presence tagged situations. The much greater imprecision might mean that there are a lot more cases where there should be presence, but the plugin used for tagging didn't find any event that could be used to gather presence value.

For the continuous case the results are an overall improvement, every score for both class and the average were increased with the new features and methods. With the user_NDI the results are similar to the non-continuous, precision dropped abruptly for the presence class but recall doubled. For this case the same reason could be applied, the user_NDI is a value tied to the reads and writes messages, so it is probable that many instances that actually represent presence cases weren't tagged as such.

TABLE 5.2: Results for each type of dataset obtained using 70% for training with down-sample and 30% for testing using the event based approach and without excluding any information from metrics

type of presence data	classes	precision	recall	f1-score	support
non-continuous	**unknown**	0.99	0.93	0.96	506315
	presence	0.64	0.94	0.76	61921
	avg/total	0.95	0.94	0.94	568236
continuous	**unknown**	0.95	0.96	0.95	52710
	presence	0.98	0.97	0.97	98115
	avg/total	0.97	0.97	0.97	150825
user_NDI	**unknown**	1.00	0.99	0.99	33696
	presence	0.87	1.00	0.93	3141
	avg/total	0.99	0.99	0.99	36837

For the event approach the results improved in all scenarios. In the non-continuous case, precision increased 0.1 and recall practically doubled, leading to an approximately 0.05 increase in the average scores since the good scores for

the unknown class were maintained. In the continuous dataset all scores were increased, specially in the unknown class, were the values rose in approximately 0.4. The user_NDI case also had major increases, in this case in the presence class.

In this approach we can see the imbalance between classes is of lesser orders of magnitude than in the previous. This probably contributed for the precision values to remain high, contrary to the window version. Besides this, the nature of this dataset: instances are only created when a message is processed, makes it easier to have entries correctly tagged, in opposition to the window mode where in some intervals of time there are none, or very few messages and no information from presence could have been gathered.

Table 5.3 shows the top feature values for the presented scenarios obtained using an XGBoost algorithm for feature selection. The main difference that can be observed from the table and remarks in 4.4 is that last message difference related features appear on the very top in all scenarios except the window continuous and the event non-continuous. Averages continue prominent in the continuous datasets. Change Property marked features are also common in all experiments. Further exploring the output of the algorithms reveals the new different schemas related features also have relevant scores specially in the event approach, and that the patterns shown in 4.4 remain, with count related features important for non-continuous and user_NDI, and averages more relevant in the continuous.

TABLE 5.3: Top feature importance values of the not excluding section according to the XGBoost algorithm

experiment	top features
window non-continuous not excluding	(LastMessageDiffCP, 0.081), (LastMessageDiff, 0.061), (totalDevices, 0.058), (rruleLastMessageDiff, 0.05), (ruleLastMessageDiff, 0.047), (rruleLastMessageDiffCP, 0.036), (ruleLastMessageDiffCP, 0.033)
window continuous not excluding	(iWeekdayAvgDay, 0.068), (wruleLastMessageDiff, 0.065), (WeekdayAvgDay, 0.056), (iindependent_updateLastMessageDiffCP, 0.05), (rruleLastMessageDiffCP, 0.044), (WeekendAvgDay, 0.042), (rDawnWeekdayuserAvgDay, 0.032), (rruleAvgDay, 0.032), (iAvgDay, 0.03)
window user_NDI not excluding	(rruleLastMessageDiff, 0.085), (irule_writeUpdateAvgDay, 0.067), (ruleLastMessageDiff, 0.065), (LastMessageDiff, 0.055), (iWeekdayAvgDay, 0.037), (rLastMessageDiff, 0.035), (ruserLastMessageDiff, 0.032), (ruserLastMessageDiffCP, 0.028), (iWeekendAvgDay, 0.028)
event non-continuous not excluding	(iindependent_updateCountFive, 0.063), (iindependent_updateAvgDay, 0.052), (totalDevices, 0.043), (differentSchemas, 0.031), (ruleLastMessageDiff, 0.028), (wruleLastMessageDiffCP, 0.023), (iindependent_updateCountHalf, 0.023), (irule_writeUpdateLastMessageDiffCP, 0.023), (ruserCountHalf, 0.022)
event continuous not excluding	(rruleLastMessageDiffCP, 0.153), (iuser_writeUpdateLastMessageDiffCP, 0.049), (wruleLastMessageDiff, 0.047), (WeekdayAvgDay, 0.038), (totalDevices, 0.034), (wLastMessageDiffCP, 0.031), (ruserLastMessageDiffCP, 0.025), (timeOfDay=Noon, 0.022)
event user_NDI not excluding	(rruleLastMessageDiffCP, 0.066), (iuser_writeUpdateLastMessageDiffCP, 0.046), (wuserLastMessageDiffCP, 0.04), (irule_writeUpdateLastMessageDiffCP, 0.04), (rWeekendAvgDay, 0.03), (ruleLastMessageDiffCP, 0.029), (rruleLastMessageDiff, 0.027), (iWeekdayAvgDay, 0.027), (iLastMessageDiffCP, 0.026)

5.2 Excluding Messages Used to Generate Presence Information from Metrics

As with the initial results the number of users depending on type was: 788 for the non-continuous, 189 for the continuous and 28 for user_NDI.

TABLE 5.4: Results for each type of dataset obtained using 70% for training with down-sample and 30% for testing using the window based approach and excluding messages used to gather presence from metrics

type of presence data	classes	precision	recall	f1-score	support
non-continuous	**unknown**	1.00	0.86	0.92	1818230
	presence	0.02	0.62	0.04	7900
	avg/total	0.99	0.86	0.92	1826130
continuous	**unknown**	0.71	0.72	0.71	122275
	presence	0.87	0.86	0.86	261917
	avg/total	0.82	0.82	0.82	384192
user_NDI	**unknown**	1.00	0.80	0.89	67486
	presence	0.01	0.86	0.02	187
	avg/total	1.00	0.80	0.89	67673

For the exclusion window scenario the results emulate what was obtained in the non-exclusion case. In the presence class for non-continuous precision and f1-score drop but recall is greatly increased compared with the initial results. In the continuous dataset all values are increased, and so the averages are augmented by 0.2. For the user_NDI the results are also similar to the non excluding, with dropped precision but a greatly increased recall for the presence class.

TABLE 5.5: Results for each type of dataset obtained using 70% for training with down-sample and 30% for testing using the event based approach and excluding messages used to gather presence from metrics

type of presence data	classes	precision	recall	f1-score	support
non-continuous	**unknown**	0.98	0.74	0.84	474423
	presence	0.26	0.85	0.39	50497
	avg/total	0.91	0.75	0.80	524920
continuous	**unknown**	0.82	0.87	0.84	52707
	presence	0.93	0.90	0.91	98106
	avg/total	0.89	0.89	0.89	150813
user_NDI	**unknown**	1.00	0.98	0.99	29708
	presence	0.63	0.99	0.77	1181
	avg/total	0.99	0.98	0.98	30889

For this approach the results are also greatly increased compared with the initial. But as expected there is a drop compared with the non excluding scenario. This comes in the form of a heavily decreased precision for the presence class in the non-continuous dataset, and a significant decrease for the same score in the user_NDI dataset.

Table 5.6 shows the importance values now for the excluding experiments. There isn't a significant difference with the results show in Table 5.3, they can be summed as an overall change in specificity or focus of the granularity. For example, rruleLastMessageDiff for the same scenario instead of lastMessageDiff, or iWeekdayAvgDay instead of iWeekdayAvgDay. Also, as expected, the count related features drop in scores, and are no longer present or have a more general granularity. Further analysis finds that the different schemas features also drop significantly in the window tests but remain present in the event ones.

TABLE 5.6: Top feature importance values of the excluding section according to the XGBoost algorithm

experiment	top features
window non-continuous not excluding	(rruleLastMessageDiff, 0.108), (totalDevices, 0.096), (totalCountFive, 0.049), (individualUnits, 0.041), (LastMessageDiff, 0.032), (ruleLastMessageDiff, 0.031), (iWeekdayAvgDay, 0.028), (totalAvgDay, 0.028)
window continuous not excluding	(WeekdayAvgDay, 0.062), (wruleLastMessageDiff, 0.046), (WeekendAvgDay, 0.043), (iindependent_updateLastMessageDiffCP, 0.04), (iAvgDay, 0.04), (iuser_writeUpdateAvgDay, 0.035), (wruleLastMessageDiffCP, 0.033), (rDawnWeekdayuserAvgDay, 0.033), (totalDevices, 0.03)
window user_NDI not excluding	(userLastMessageDiffCP, 0.074), (LastMessageDiffCP, 0.064), (ruserLastMessageDiff, 0.059), (iLastMessageDiff, 0.057), (wuserLastMessageDiff, 0.045), (WeekendAvgDay, 0.045), (individualUnits, 0.038)
event non-continuous not excluding	(individualUnits, 0.048), (totalDevices, 0.039), (differentSchemas, 0.036), (irule_writeUpdateLastMessageDiffCP, 0.034), (iindependent_updateAvgDay, 0.033), (totalCountMin, 0.025), (ruserLastMessageDiff, 0.023), (wWeekdayAvgDay, 0.023), (rruleLastMessageDiff, 0.022)
event continuous not excluding	(rruleLastMessageDiffCP, 0.153), (totalDevices, 0.061), (iuser_writeUpdateLastMessageDiff, 0.054), (iuser_writeUpdateLastMessageDiffCP, 0.049), (wLastMessageDiffCP, 0.042), (wruleLastMessageDiff, 0.033), (rWeekendAvgDay, 0.033), (LastMessageDiff, 0.024), (totalCountHalf, 0.024)
event user_NDI not excluding	(LastMessageDiff, 0.078), (iindependent_updateLastMessageDiff, 0.067), (timeOfDay=Night, 0.05), (iindependent_updateCountQuarter, 0.041), (iuser_writeUpdateLastMessageDiff, 0.031), (userLastMessageDiffCP, 0.031), (iindependent_updateCountMin, 0.029)

5.3 Using more data

For the results presented so far, one month of anonymized data was used. There were more months of gathered data available to use for this project, however, both the Muzzley platform and the objectives and approach of this dissertation evolved and so not all data collected can be used for experiments. There was only one

more month of data that could be joined with this one for the experiments, and this is what we did for the very last phase. The main objective for this was to see how the models behaved with more data, specially for tests with down-sampling. This was only done with the event approach, since the window one generates a lot more data, taking much more time to process, also the event approach performed better in the last phase.

For the 70%/30% validation results dropped for the non excluding scenarios and remained mostly the same for the excluding. With the down-sampling validation scores increased for both non-continuous and user_NDI datasets in both scenarios. Overall it can be said that the models perform slightly better with this new month of data, although there are exceptions to this and in the long run some possible variations on the habits and interactions of the users should be taken into account, namely those related to the seasons.

5.4 Conclusions On The Final Experiments

Although not shown here, tests for 5.1 and 5.2 scenarios but with cross validation, as before instead of the new 70% / 30% process were done. These showed an overall increase in performance which means the new features are an improvement to the classification. The main increase however comes from down-sampling has seen in chapter 4, and from the effect of joining all the improvements.

The high scores shown by the event approach indicate this might be the best way to go for future work or a production application of this study. In this chapter we have showed results obtained where the test set conserved its imbalanced properties in order to stay true to our original created datasets. Real explicitly tagged data obtained from users however would probably follow a class distribution that its less imbalanced than some of our cases, so the difficulty of real data might be softened by the fact that is more balanced.

Chapter 6

Conclusions

This study presented an approach for human presence detection in a ambient intelligence like context. Further improvement can be applied to the system but still it performs well in its current state for certain scenarios and shows a promising capability for detection using unspecific data from several IoT devices. In order to create the system an extensive processing phase was applied to anonymized data gathered from a real application, the Muzzley app and platform, generating metrics for each new example. Datasets were labeled with presence information inferred by heuristic methods.

The generated datasets were used with the Weka and Scikit-Learn machine learning libraries in order to test the approach. After obtaining the initial results we tried to improve them, first focusing on feature selection, then on addressing the class imbalance issue present in some scenarios. After this, we explored the results better to see what could be improved, tests were done with different algorithms, and experimented with parameter search. Finally, the improvements and new features were used in a final testing phase creating the final scores that were analyzed and compared with the initial ones.

6.1 On the Development of the Project

The main challenges of the project were the processing needed, the diversity of devices that interact with the platform, the creation of adequate features and not having explicit presence information.

Originally the project was more related to sensor fusion. The idea was to join information from all types of devices possible in order to detect presence and then try to see by removing specific devices and removing the number of devices used, to what limit of minimum information presence could be detected with a reasonable accuracy. But we ended up using only the interactions because of the way tagging was done and because it would be the best way to be able to use all devices from all users and thus create this novel approach.

The project was supposed to count with a explicitly tagged dataset from users who agreed to share their presence hours. This ended up not occurring since is not an easy thing to ask of the users. We had however the user_NDI dataset, that, while not being completely specific, is a closer to reality information. Nevertheless we think the mechanisms we used for tagging are good for at least proving the potential of this indirect presence detection. Also, this type of tagging could maybe inspire and be applied to other projects related to data mining, or machine learning with IoT devices, for example for activity detection, the cooking activity events could be tagged according to the cooking pot messages, etc.

Most of the development time of the project was on creating the plugins to process the data, since there was the need to first understand the data, to adapt to new technologies and tools and to plan the algorithms to follow the same paradigm as the rest of the Segmentation Engine. The first results with Weka were done quickly since we already had experience with it, and then another learning phase came with the adoption of Scikit-Learn. The bulk of the rest of the time was invested in doing experiments and tweaking the algorithms and plugins to have good conclusions.

6.2 On the Results Obtained

The results obtained were quite interesting. Since the initial results that the approach has shown potential and the improvements done further increased it. In the final results the window approach achieved recall higher than 0.8 for all types of datasets in the presence class, while also having achieving top scores in the unknown class. Precision however was not so good for presence, which as analyzed in chapter 5 could be due to lack of information for tagging. The exclusion scenario was also better than the initial since there wasn't such a significant decrease in scores.

In the event approach the final results were better. Recall had very good scores for both classes and precision didn't drop as drastically. Also for the exclusion scenario the results suffered a minor overall drop with the user_NDI, possibly the most realistic of the datasets, still having very good scores.

In terms of features used, as analyzed, it isn't necessary at least for now, to use that many levels of granularity since some very specific features don't add much more information. The features implemented in the later phase, the number of different schemas and time differences between last messages were important for the classification along with the initial overall average metrics and the immediate counts mostly at the type of message and type of trigger metric.

The results obtained are also interesting because for all tests, except when stated, we didn't set a minimum number of devices or interactions in order for users to be considered, except the at least 1 average interaction per day, which means the results obtained were considering a lot of different setups and types of users.

In terms of implementing this system, although the process phase could be optimized, it is very plausible to be able to integrate the operations needed to generate the features information in a Muzzley like platform, since most of the metrics are basically counts and averages. Training the model does take some time, specially if parameter search is considered, but it wouldn't be necessary to

train new algorithms that often. The prediction with the chosen classifiers is also very fast, so this part wouldn't probably be the bottleneck.

This dissertation also resulted in the publication of the paper "A Machine Learning Approach for Indirect Human Presence Detection" presented at the International Conference on Digital Information Management held September of 2016 at Porto.

6.3 Future Work

A lot of ideas and improvements are possible for future work. The first of which is to try getting an explicitly tagged dataset or a way for users to give feedback about the prediction since it is fundamental to validate these results in the more realistic possible way before applying the algorithms in production. Other improvements are more features, such as similar ones to the different schemas but for different identifiers in the hierarchy that would represent the exact number of properties of devices that were changed. These different schemas / identifiers could be applied to the interactionStatsPerUser to have averages of these numbers. An even bigger dataset could be used and create features related to the current season, average weather for a time period, current weather, etc.

Other approaches could also be tried, such as using regression instead of classification to have a certainty level of presence, although this can be done with the prediction probability capabilities of the Random forest and XGBoost classifiers. Or using Markov Chains or Recurrent Neural Networks with a more sequence or memory based dataset.

Also, besides these improvements, for a production application of this study the information of specific detectors and the heuristics used for tagging these datasets would be features themselves since in this new real scenario all possible information is to be used.

Presence detection is related to other important information an ambient intelligence system should have about its context. Such as occupation, that is how many people are present, and which activity is currently being done, and if possible to know all this information for each division of the environment, in this case, for each room. A possible evolution is the creation of systems or modules to do these other types of tasks and then connect them together, and with the presence module, in a way that the predictions of each can be used by the others for better accuracy.

Although some results can be improved, the approach described provides insights for systems integrating these new types of devices to build upon. These new IoT devices are easily acquired and start working out of the box, so it is very interesting to have this novel way to be able to join information from all of them with a specific purpose. Presence detection, and, as mentioned, activity detection, or occupation number, are very important challenges to ambient intelligence and hopefully this dissertation will help shed a little light on the current possibilities with these devices and technologies.

Bibliography

[1] Eli Zelkha et al. “From Devices to "Ambient Intelligence"”. In: *Digital Living Room Conference* (1998) (cit. on p. 1).

[2] Diane J. Cook, Juan C. Augusto, and Vikramaditya R. Jakkula. “Ambient intelligence: Technologies, applications, and opportunities”. In: *Pervasive and Mobile Computing* 5.4 (2009), pp. 277–298 (cit. on p. 1).

[3] E.H.L. Aarts and J.L. Encarnacão. *True Visions: The Emergence of Ambient Intelligence*. Springer, 2006, p. 454 (cit. on pp. 1, 7).

[4] Friedemann Mattern and Christian Floerkemeier. “From the internet of computers to the internet of things”. In: *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 6462 LNCS (2010), pp. 242–259 (cit. on p. 1).

[5] Lopez Research. “An Introduction to the Internet of Things (IoT)”. In: *Lopez Research Llc* Part 1. of.November (2013), pp. 1–6 (cit. on p. 2).

[6] Neil Gershenfeld, Raffi Krikorian, and Danny Cohen. *The Internet of things.* Vol. 291. 4. 2004, pp. 76–81 (cit. on p. 2).

[7] Gerald Santucci. “The Internet of Things : Between the Revolution of the Internet and the Metamorphosis of Objects”. In: *Forum American Bar Association* (2010), pp. 1–23 (cit. on p. 2).

[8] Cognizant Reports. “Reaping the Benefits of the Internet of Things”. In: May (2014) (cit. on p. 2).

[9] Kaivan Karimi. *The Role of Sensor Fusion in the Internet of Things* (cit. on p. 2).

[10] James L. Crowley and Yves Demazeau. "Principles and techniques for sensor data fusion". In: *Signal Processing* 32.1-2 (1993), pp. 5–27 (cit. on p. 2).

[11] Dave Evans. "The Internet of Things - How the Next Evolution of the Internet is Changing Everything". In: *CISCO white paper* April (2011), pp. 1–11 (cit. on pp. 2, 3).

[12] Ken Peffers et al. "A Design Science Research Methodology for Information Systems Research". In: *J. Manage. Inf. Syst.* 24.3 (Dec. 2007), pp. 45–77 (cit. on p. 6).

[13] Richard Baskerville. "What design science is not". In: *European Journal of Information Systems* 17.5 (2008), pp. 441–443 (cit. on p. 6).

[14] Juan Carlos Augusto and Daniel Shapiro. *Advances in Ambient Intelligence: Volume 164 Frontiers in Artificial Intelligence and Applications*. IOS Press Amsterdam, The Netherlands, 2007, p. 200 (cit. on p. 7).

[15] Diane J. Cook and Sajal K. Das. "How smart are our environments? An updated look at the state of the art". In: *Pervasive and Mobile Computing* 3.2 (2007), pp. 53–73 (cit. on p. 7).

[16] Gustavo José Henriques Patrício. "Redes Sem Fios de Microcontroladores com Acesso Remoto Aplicada à Domótica". In: *Cadernos de Saúde Pública* (2009) (cit. on p. 8).

[17] Rob Collingridge. *Dream Green House*. 2009. URL: `http://dreamgreenhouse.com/index.php` (visited on 12/28/2015) (cit. on p. 8).

[18] Rob Collingridge. *Dream Green House: Occupancy & Presence*. 2013. URL: `http://www.dreamgreenhouse.com/projects/2013/presence/index.php` (visited on 12/28/2015) (cit. on p. 8).

[19] Sumi Helal et al. "The Gator tech smart house: A programmable pervasive space". In: *Computer* 38.3 (2005), pp. 50–60 (cit. on p. 8).

[20] Alessandra De Paola and ML Cascia. "User detection through multi-sensor fusion in an AmI scenario". In: *15th International Conference on Information Fusion (FUSION), 2012* (2012), pp. 2502–2509 (cit. on p. 9).

[21] Thiago Teixeira, Gershon Dublon, and Andreas Savvides. "A Survey of Human-Sensing: Methods for Detecting Presence, Count, Location, Track, and Identity". In: *ACM Computing Surveys* 5 (2010), pp. 1–35 (cit. on p. 9).

[22] Christian Martin et al. "Sensor Fusion Using a Probabilistic Aggregation Scheme for People Detection and Tracking". In: *Proc. of the 2nd European Conference on Mobile Robots (ECMR 2005), Ancona, Italy* 54 (2005), pp. 176–181 (cit. on p. 9).

[23] Zhenghua Chen et al. "Fusion of WiFi, Smartphone Sensors and Landmarks Using the Kalman Filter for Indoor Localization". In: *Sensors* 15.1 (2015), pp. 715–732 (cit. on p. 10).

[24] Dian Gong, G Medioni, and Xuemei Zhao. *Structured Time Series Analysis for Human Action Segmentation and Recognition.* 2014 (cit. on p. 10).

[25] T Warren Liao. "Clustering of time series data - a survey". In: *Pattern Recognition* 38 (2005), pp. 1857–1874 (cit. on p. 10).

[26] Mark Hall et al. "The WEKA data mining software". In: *SIGKDD Explorations Newsletter* 11.1 (2009), p. 10 (cit. on p. 39).

[27] J Ross Quinlan. *C4.5: Programs for Machine Learning.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1993 (cit. on p. 40).

[28] Fabian Pedregosa et al. "Scikit-learn: Machine Learning in Python". In: *... of Machine Learning ...* 12 (2012), pp. 2825–2830 (cit. on p. 40).

[29] Wes McKinney. "Data Structures for Statistical Computing in Python". In: *Proceedings of the 9th Python in Science Conference* 1697900.Scipy (2010), pp. 51–56 (cit. on p. 41).

[30] R L Plackett. "and the Chi-squared Test". In: 51 (1983), pp. 59–72 (cit. on p. 41).

[31] Bernhard E Boser, Isabelle M Guyon, and Vladimir N Vapnik. "A Training Algorithm for Optimal Margin Classifiers". In: *Proceedings of the Fifth Annual Workshop on Computational Learning Theory.* COLT '92. New York, NY, USA: ACM, 1992, pp. 144–152 (cit. on p. 41).

[32] Tin Kam Ho. "Random decision forests". In: *Proceedings of 3rd International Conference on Document Analysis and Recognition* 1 (1995), pp. 278–282 (cit. on p. 41).

[33] Guillaume Lemaitre. *Imbalanced-Learn*. 2014. URL: `https://github.com/scikit-learn-contrib/imbalanced-learn` (visited on 09/08/2016) (cit. on p. 52).

[34] Nitesh V. Chawla et al. "SMOTE: Synthetic minority over-sampling technique". In: *Journal of Artificial Intelligence Research* 16 (2002), pp. 321–357 (cit. on p. 53).

[35] Ando Saabas. *TreeInterpreter*. 2015. URL: `https://github.com/andosa/treeinterpreter` (visited on 07/28/2016) (cit. on p. 56).

[36] Tianqi Chen and Carlos Guestrin. "XGBoost : Reliable Large-scale Tree Boosting System". In: *arXiv* (2016), pp. 1–6 (cit. on p. 61).