Smart Home Activity Inference using Network Data

A supervised methodology for IoT device activity inference, annotation, and logging for Smart Homes

MARY ZACHARIAS

School of Computer Science and Informatics,
Cardiff University

This dissertation is submitted for the degree of
MSc Computing

Abstract

This dissertation seeks to infer physical events in real time, from an ensemble of off-the-shelf smart home devices, using only encrypted network data.

To develop the proposed system, a "smart-home style network" is set up within the university's laboratory environment. A multi-stage classification process is then used — a set of classifiers are trained to first, 'fingerprint' connected devices using passive techniques, and then to detect binary and more specific differences in the states of these connected devices.

To validate the system, these classifiers are then deployed within a smart home 'diagnostics' application. This layman-friendly solution offers users real-time visibility and correlated insights into the current and historic physical activities of their smart home devices —without any manual set-up or device integration necessary.

This paper achieves a 99.3% f1 score in detecting a connected device as 'IoT', 96.0% f1 score in fingerprinting the device, and 90.1% f1 score in identifying the state of the targeted smart home device. Through these results, this paper seeks to demonstrate that network data could be used to supplant more complex device data collection techniques within diagnostics or even simple Human Activity Recognition tools in smart home settings.

To the best knowledge of the author, the contributions of this project include:

1. A demonstration that device activities in smart homes can be successfully inferred from encrypted network traffic data.

2. A survey of current use-cases and methodologies in activity recognition for smart homes.

3. A methodology to (a) sniff network data and stream it in real-time, (b) pre-process data online to serve real-time predictions using pre-trained models, (c) to re-train machine learning models online based on user input.

4. An analysis of performance metrics from various machine learning models with a recommendation for the top-performing model.

5. A standalone interactive tool with opportunity to add-on to an existing open-source home automation platform.

Acknowledgements

I would like to thank Dr. Charith Perera, my dissertation supervisor, for his constant support throughout the course of the project. Dr. Perera's guidance throughout the academic year allowed this project to take shape seamlessly.
I would also like to thank my family for the support they provided throughout this academic year.

DATE: **21 October 2022**

Table of Contents

Page

List of Tables **viii**

List of Figures **ix**

1 Overview **1**

1.1 Introduction 1

1.2 Project Aim 2

1.3 Objectives 3

1.4 Contributions 3

1.5 Outline of Thesis 4

2 Background **6**

2.1 Introduction 6

2.2 Activity recognition 6

2.2.1 The need for activity recognition 6

2.2.2 Use-cases for Activity Recognition in Smart Homes 7

2.2.3 Methodology for HAR systems 8

2.2.4 Common challenges in HAR 10

2.3 Machine Learning approaches in HAR 11

2.3.1 Shallow learning 11

2.3.2 Deep learning 11

2.3.3 Transfer learning 12

2.3.4 Active learning 12

2.4 Conclusion and Discussion of Scope of Work 12

3 Related Work **13**

3.1 Introduction 13

3.2 Existing Literature 13

3.2.1 State-of-the-art in Fingerprinting Techniques 14

3.2.2 State-of-the-art in Device State Recognition Techniques 17

3.2.3 State-of-the-art in Activity Recognition Techniques 18

3.3 Conclusion and Evaluation of Literature Discussed 19

4 Methodology **20**
4.1 Introduction 20
4.2 Overall Methodology 21
4.3 Multi-Stage classification methodology for Activity Identification 23
4.3.1 Stage 1: Detecting IoT versus Not IoT 23
4.3.2 Stage 2: Device Identification 24
4.3.3 Stage 3: Device State/Activity Identification 24
4.3.4 Stage 4: User-Activity Identification 25
4.4 Conclusion 25

5 Data Preparation **27**
5.1 Introduction 27
5.2 Developing the Smart Home Network 27
5.2.1 Devices used 27
5.2.2 Tools used 28
5.3 Data Capture 28
5.3.1 Methodology for Data Capture from Smart Home Lab 28
5.3.2 External Datasets 30
5.3.3 Methodology for Data Capture during Live Deployment 30
5.4 Data Pre-processing 31
5.5 Summary 31

6 Pipeline Preparation and Application Development Methodology **33**
6.1 Introduction 33
6.2 Pipeline Development Overview 33
6.2.1 MLOps and Productionalisation 34
6.2.2 Pipeline orchestration tools 35
6.3 Application development methodology 36
6.3.1 Environment 36
6.3.2 Features and UI 36
6.3.3 Back End 39
6.4 Summary 41

7 Performance and Analysis **42**
7.0.1 Detecting IoT versus Not IoT 43
7.0.2 Detecting Device type — Device Fingerprinting 46
7.0.3 Detecting Activity States - On vs Off 47
7.1 Summary 48

8 Conclusion and Future Work **51**
8.1 Conclusion 51

8.2 Limitations 52
8.3 Future Work 52

9 Reflection **54**
9.1 Project Management Style 55
9.2 Shortcomings of work conducted 56
9.3 Key Learnings and Takeaways 58

Bibliography **60**

Appendix A **65**
A.0.1 Software Used 65
A.0.2 CLI/ OS commands 66
A.0.3 Apache Kafka Scripts 67
A.0.3.1 Kafka Producer 67
A.0.3.2 Kafka Consumer 69
A.0.4 User Interface 70
A.0.5 Classification results 73
A.0.5.1 Stage-1 (IoT vs NoT) Classification results 73
A.0.5.2 Stage-2 (Device Fingerprinting) Classification results 74
A.0.5.3 Stage-3 (Device State) Classification results 76
A.0.6 Code 79
1. Script for Homepage dynamic graph 79
2. Script for IP -> MAC mapping 80
3. Script to predict IoT versus Non-IoT devices 81
4. Script to retrieve data from database 81
5. Script to output data from above two functions to front end from Database 82
6. Script to create individual line graphs of each devices' activity for a desired time period, generated through a user query (flask form) 82
7. Script to write user-input(label) for Device Location from the front-end to the associated table in the database 84
8. Script to write user-input(label) for Device Name from the front-end to the associated table in the database 85
9. Script to predict device's names and locations from 30-second traffic flows using relevant Stage2 and Stage3 classifiers, and write this to a table in the database, for functions like graphing and logging 86
10. Script to capture live network traffic every 30 seconds, convert it to a flow-based feature-set (CSV) and a packet 'metadata' based feature-set (CSV), save these CSV files temporarily (for future re-training use), and and then serve predictions on each sample 88
11. DeviceStates Table in SQLite 90

12. DeviceLocations Table in SQLite 91
13. DeviceNames Table in SQLite 91

List of Tables

Table **Page**

3.1 Survey of passive device identification techniques using network tracking 16

5.1 Network traces from the Smart Lab at Cardiff University used by the Device recognition model. 30

7.1 Performance results from 8 binary classifiers for the IoT vs. Non-IoT detection stage . . 45
7.2 Performance results from 9 multi-class classifiers for the Device fingerprinting stage . 46
7.3 Performance results from 9 binary classifiers for the Device state recognition stage . . 48

1 Software used for generating results 65

LIST OF FIGURES

FIGURE **Page**

2.1 Activities of Daily Living. Fig source: (Ni *et al.*, 2015) 7
2.3 General use cases for HAR in smart homes 8
2.4 Generic HAR Methodology 8
2.5 Considerations for Human Activity recognition 9

3.1 High-level logic for sequential activity detection using network data 14
3.2 Device Fingerprinting Techniques 15
3.4 Source: (Acar *et al.*, 2020) 17
3.5 Source: (Apthorpe *et al.*, 2018) 17

4.1 Multi-stage process for activity identification 20
4.2 Pipeline for user activity inference tool 20
4.3 Overall Process Diagram for a generic Activity Recognition 'Add-On' for Home Assistant. 21
4.5 Overall Process Diagram for a generic standalone Activity Recognition tool 22
4.6 Proposed methodology for the standalone Activity Recognition tool 23

5.1 General pre-processing methodology 31

6.1 Proposed Process Diagram 34
6.2 Typical workflow for real-time prediction 35

7.1 Classification of IoT and Non-IoT traffic patterns 44
7.2 Feature Importance - Light GBM for IoTvNoT on traffic metadata 45
7.3 Feature Importance - Light GBM for Device Fingerprinting on network flows 47
7.4 Feature Importance - Extreme Gradient Boost for Device State Identification using network flows 49

1 Homepage: Real-time graph showing bytes/second sent over the local network 70
2 Homepage: Current Activities and Active spaces in the Smart Home 70
3 Homepage: Anomalies activities and warnings 71
4 Homepage: Navigation 71
5 Connected Devices: Overview of all device states 71

6 Connected Devices: User input (supervised learning) to manually input device names and locations (to partially retrain models online 72
7 Connected Devices: Detection of IoT and Non IoT devices in the local network 72
8 Dashboard: Historic Trends 72
9 Results - Cumulative 73
10 Stage 1 - Results - Light Gradient Boosting Machine 73
11 Stage 2 - Results - Light Gradient Boosting Machine 74
12 Stage 2 - Results - Extreme Gradient Boost 74
13 Stage 2 - Results - OneVsRest with Extreme Gradient Boost base estimator 75
14 Stage 2 - Results - Random Forest 75
15 Stage 3 - Results - AdaBoost 76
16 Stage 3 - Results - Gradient Boost 76
17 Stage 3 - Results - OneVsRest with Random Forest Base estimator 77
18 Stage 3 - Results - OneVsRest with ExtremeGradientBoost base estimator 77
19 Stage 3 - Results - ExtremeGradientBoost 78
20 Stage 3 - Results - Feature Importance Scores - ExtremeGradientBoost 78

Chapter 1

Overview

1.1 Introduction

Human activity recognition, or *HAR*, is a broad field of study concerned with identifying the specific movement or action of a person(s) based on sensor data. HAR has multi-faceted applications, and is often explored in diverse built environments such as hospitals and hospices, office spaces, factory floors, educational institutions, commercial complexes, sports and leisure facilities, and domestic environments like smart homes and care homes. While research in HAR has been traditionally explored through imagery using technologies like computer and machine vision, it is now evolving to the use of non-image data —i.e., through the use of non-invasive sensing— which is already breaking barriers within the healthcare industry (Sumathy *et al.*, 2021).

Current research in HAR predominantly uses device readings taken directly from source —that is, directly from the local sensing device, or from the associated smartphone or web application of the commercial off-the-shelf (COTS) device— as input, for machine learning models to serve predictions on. Whilst this data acquisition process inevitably involves some degree of manual-intervention, it is *relatively* straightforward if (a) it is managed by the researchers themselves, and (b) it is done from a manageable set of IoT devices, and/or (c) it uses devices from developer-friendly 'ecosystems' such as 'SmartThings', 'Nest' and 'Withings', which include APIs (usually at a fee) to extract these readings.

Data acquisition in this manner, however, will not scale well for real-world applications (such as the recognition of Activities of Daily Living (ADL)), where the installation and set up of such a system may be managed by a *non-technical* end-user like a smart home owner or care home manager. Nor will this scale in scenarios where the number and types of sensing devices are likely to increase. In essence, extracting and collating data from a collection of heterogeneous, off-the-shelf, IoT devices within a local network for a layman, and even potentially a researcher, can be non-trivial.

To address this problem, this paper looks at alternate, simpler means of data acquisition, and then checks whether this technique can be used to build a competent smart home-based activity recognition system.

To this end, network traffic is explored in the rest of the paper as a viable alternative to device data.

1.2 Project Aim

The primary motivation for the proposed system is to assist diverse user groups in getting access to insights on their home devices' activities and in extension their own activities, through simple, non-invasive, hardware-agnostic means. As such, a fundamental requirement is to support in answering questions like "What are the current states of all connected devices?", and “Are there any seasonal activity patterns?” and even “What was happening in the house up to and during a particular event?”. While historically, research and development in activity recognition catered to specific user groups like health-care assistants and 3rd-party researchers, a general-purpose activity recognition application could prove useful to a much wider audience of smart home owners, care home managers, IoT enthusiasts, students, and the residents of the smart home themselves. Products that come close to giving residents insights on their own home activities include open-source software hubs like Home Assistant (H.A, 2022) and OpenHAB (OH, 2022), however these fall short of reaching 'wide-enough' user groups because of the time and technical skills they require to set up, the specific OS platform they need, and the monetary commitment they ask for, to retrieve data from some of the more popularly used devices' cloud ecosystems. It is noteworthy also that these do not currently offer any correlated or 'user' activity insights.

To be truly 'general-purpose', the project thus prioritises usability, scalability and comprehensibility. By 'usable' the paper intends that (a) the tool can run across most operating systems with minimal to no additional hardware requirements, and (b) will require no pre-requisite technical skills; by 'scalable' the paper intends that the tool can be used to (semi) automatically learn to detect new devices and new activity patterns, and by 'comprehensive', the paper intends that the tool is able to provide an insight into activities derived from across the entire home network, including the past and in real time.

To implement this project, this paper builds upon the research and methodologies used in both (Avizheh *et al.*, 2017) and (Acar *et al.*, 2020). In contrast to the first, this paper specifically focuses on the logging of device-activities as opposed to generic network traffic, and in comparison with the second, this paper productionalises the research using Machine Learning Operations (MLOps), and proposes a real-time activity inference tool in order the demonstrate results 'in the wild'.

1.3 Objectives

In order to successfully complete this project, the following objectives needed to be met:

- A literature review of existing user-activity inference techniques.
- Experimentation in activity recognition using network traffic data to assess degree of inference possible.
- The selection and implementation of appropriate machine learning classifiers for deployment and online learning.
- The design and live deployment of the machine learning pipeline.
- A performance evaluation of the overall design through relevant metrics.

1.4 Contributions

To the best knowledge of the author, the main contributions of this research project include:

1. A pipeline which uses live traffic and user-input to semi-automatically compute and expose device activity patterns of interest in smart homes.
2. Comparison with several implementations from existing literature for device identification and activity recognition accuracy using the UNSW (UNS, 2021) and a local dataset.
3. Evaluation of the proposed solution with a dataset from 8 commercial smart home devices present in the Smart Lab at Cardiff University.
4. Development of a Dockerised standalone Flask application with scope to convert to an 'Add On' (HAA, 2022) for the open-source platform, Home Assistant. (H.A, 2022).

These contributions are further outlined in chapters 5-8.

As for point 4, this project specifically chooses Home Assistant, a leading open-source automation hub for IoT, to set the benchmark against, to compare and demonstrate ease of install and usability in this development. As highlighted in Section 1.2, most home automation hubs including Home Assistant currently provide insights into individual connected devices activities including their states and usage patterns, however, these do not provide correlated or 'larger picture' insights. Moreover, due to limited free integration capabilities, these hubs do not automatically connect to all devices in the network, notably denying free access to ecosystems like Alexa, SmartThings, Nest and Google Home. By addressing this gap in the (open-source) market through a tool that is able to capture information indiscriminately from all Wi-Fi/Ethernet-connected devices within a network, the author hopes to incentivise everyday users to gain high-level insights into their everyday home activities easily.

The 'logging' capabilities of this proposal are also of note. A 'logbook' of events in a home environment can give an indication about the lifestyle profile of the residents. to this end, this tool stores network events in a low-level form. It also passes these through an abstraction mechanism that produces higher-level summaries (e.g., Netflow-like records from packet-level traces), which is persisted in a database. In addition to the home resident, the data procured from this proposal could prove lucrative to numerous 3rd-party researchers such as device manufacturers, environmental scientists, and even forensics investigators in the event of a legal proceeding.

1.5 Outline of Thesis

The remainder of this paper is structured as follows:

Chapter 2: Background

This chapter provides a brief summary of Activity Recognition in smart homes and how machine learning has been utilised for Activity Recognition.

Chapter 3: Related Work

This chapter covers a detailed study of different device and activity recognition tools and research approaches. Existing datasets within the field of activity recognition are also reviewed in order to finalise a 'reserve' dataset. The literature's are compared and analysed to finalise the design of the project.

Chapter 4: Methodology

This chapter describes the high-level pipeline followed by this research to infer user-activities and anomalies in real-time.

Chapter 5: Data Collection

This chapter discusses the process of how data is prepared before it is inputted into the system. It also outlines the pre-processing methodology used by this research.

Chapter 6: Pipeline Preparation and Application Development Methodology

This chapter describes the development of the multi-stage device activity recognition tool, along with the dependencies, frameworks and algorithms used.

Chapter 7: Performance and Analysis

This chapter describes and analyses the performance of different classifiers at each stage of the overall system. The performance of this system is compared to existing literature implementations that utilise similar datasets.

Chapter 8: Conclusion and Future Work

This chapter provides a summary of work completed for this project. Additionally, it discusses the limitations of this research work. Lastly, it outlines future short-term and long-term directions for this project.

Chapter 9: Work Reflection

This chapter provides a brief overview of the project management style along with any successes, failures and lessons learnt.

Chapter 2

Background

2.1 Introduction

This chapter provides a background into the field of Activity Recognition for Smart Homes. It highlights its use-cases and also outlines the challenges and gaps in current literature. It also describes current machine learning approaches used in activity detection. It then concludes by refining the scope of the dissertation.

2.2 Activity recognition

2.2.1 The need for activity recognition

Activity recognition systems are a large field of research and development, currently with a focus on advanced machine learning algorithms, innovations in the field of hardware architecture, and on decreasing the costs of monitoring while increasing safety and privacy (Sarnaik, 2020). Within this field, Human Activity Recognition (HAR) in Smart Homes has been on the forefront of research in the fields of Human Computer Interaction (HCI) and Internet of Things (IoT) in recent years.

HAR is often described as the art of identifying, naming and oftentimes, predicting, human movements and actions from raw image or text-based data using machine learning. Here, 'movement' typically refers to general ambulation activities performed indoors, such as walking, jogging, walking upstairs, as well as standing, sitting, and laying down, while 'actions' refer to more complex and/or functional activities such as those types of activities performed in a kitchen, bathroom or in a workspace at home. Actions in themselves, could be simple yet specific, for example, 'peeling an apple', 'wearing a jacket', 'writing a check' while others could be generic but 'complex' in nature, for example, 'visiting the bathroom' or 'cooking'. Here, 'complexity' is defined as a sequence of actions which potentially involves different interactions with objects, equipment or other people.

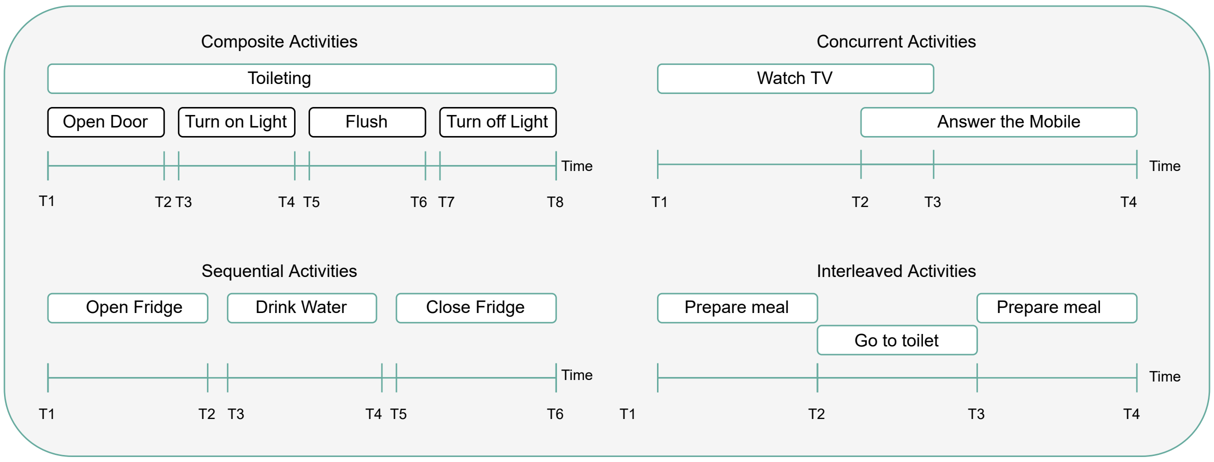


Figure 2.1: *Activities of Daily Living. Fig source: (Ni* et al.*, 2015)*

Activities are typically grouped under 4 categories: Composite, Concurrent, Sequential and Interleaved. Defining and labelling simple patterns such as walking, sleeping, sitting, and more complex patterns such as cooking, eating lunch, watching TV, working, etc. would involve factors such as time of day, duration, surrounding devices and location.

In the context of smart homes, this data is typically collected from internet-connected (IoT) devices which are largely grouped under two categories:

- (a) *Body-based devices or wearables:* These include smartphones, smart watches, implantables and even smart clothing, which often include advanced sensors such as gyroscopes, accelerometers, magnetometers, heart-rate monitors, glucose monitors, etc.
- (b) *Environment-based devices:* These include motion and heat detectors, door/window contact sensors, vibration sensors, and smart appliances such as surveillance systems (smart cameras and alarms), body-health trackers (sleep mats, weighing scales, blood pressure monitors), voice assistants, energy monitoring devices (smart plugs and bulbs) and entertainment systems (smart TVs, speakers and mood lights) etc.

Using such a collection of visual and non-visual sensory data, HAR systems are able to retrieve and process contextual (environmental, spatial, temporal, etc.) data to understand human behavior both inside and outside built environments Accurately tracking and logging user activities and behaviours using these devices in a domestic environment has applications in fields such as forensics, health care systems, cyber defence, domestic resource management, and smart device (IoT) development amongst others. Some of these applications are further described in the next section.

2.2.2 Use-cases for Activity Recognition in Smart Homes

Sensor-based activity recognition can be used to model a wide range of human activities. These systems could prove crucial for numerous applications, some of which include: (a) *Forensic investigations:* where activity data retrieved from smart homes is increasingly being used as supportive

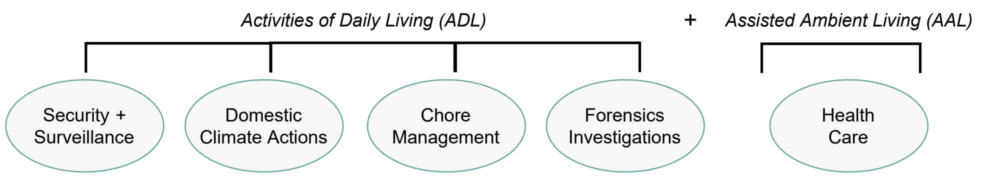


Figure 2.3: *General use cases for HAR in smart homes*

evidence in court cases (Burke, 2022) (b) *Health care systems:* where the move from image-based to sensor-driven systems has brought autonomy and a higher degree of privacy to patients lives, breaking barriers in Assisted Ambient Living (or 'AAL'). (c) *Security systems - Physical and Cyber:* where malicious activities and threats both in the physical realm and in cyber space, like unexpected entry, flood events and even hack attempts, are being detected through sophisticated HAR systems, (d) *Chore and resource management; Activities of Daily Living (ADL):* where the output of underlying HAR systems increase the intuitively and use of home automation systems —further assisting residents in managing their chores and/or domestic climate actions.

2.2.3 Methodology for HAR systems

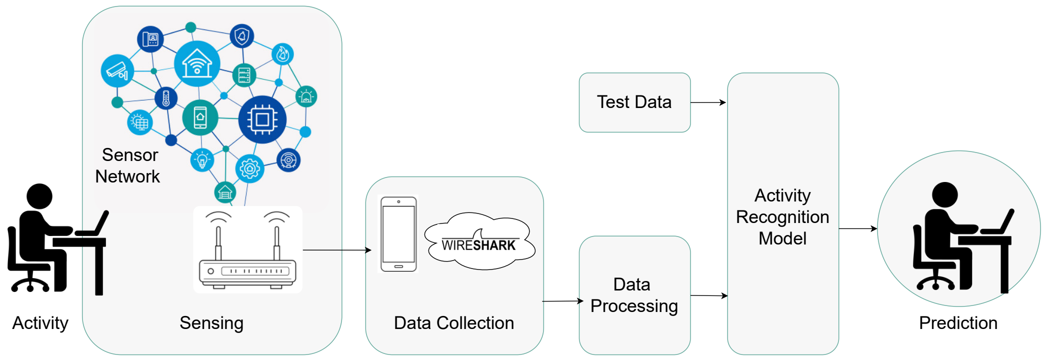


Figure 2.4: *Generic HAR Methodology*

At its simplest, an activity recognition system requires a sensor network, an actor(s), a data acquisition and processing methodology and a prediction model [Figure 3.1]. In a smart home setting, sensors collect readings based on user actions (e.g., 'Door state = open' and 'Hue Bulb state = On'), along with the timestamp of the action. These readings are then logically strung together and labelled as composite activities like 'cooking' or 'working', according to some pre-defined logic. Machine Learning models are then trained on this input and used to infer future activities. These models are either custom trained from scratch for each smart home, or trained using transfer learning using information from pre-trained models. Once trained, these models are either retrained periodically to catch any new information, and/or trained continuously to prevent temporal drift.

2.2.3.1 Strategies for HAR

Depending on the end-application, the strategy for how activities are observed and logged in smart home environments can very. While the technology used to recognise human behavior include visual, non-visual, and multimodal sensor technologies, this paper focuses specifically on non-visual sensing, which in itself involves the following considerations:

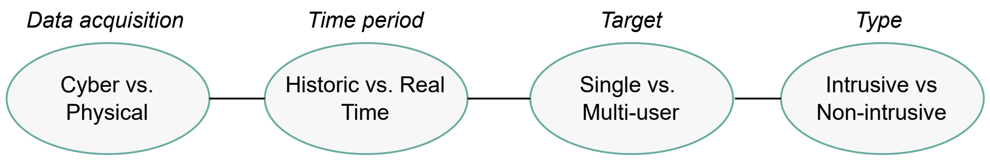


Figure 2.5: *Considerations for Human Activity recognition*

- *Historic vs. Real-Time detection*
  In some situations, it is not suitable to recognise activities several minutes or hours after they occur. Examples includes emergencies like falls in care homes, or rising water or smoke levels in the case of a calamity. In these instances, real-time detection is a necessity to propose reactive systems. For other scenarios like extracting energy consumption patterns or even event-based evidence from a crime scene, historic logging is often enough. For these applications too, frequency and duration of data capture can be a big consideration.

- *Intrusive vs. Non-intrusive sensing*
  Certain situations allow a degree of intrusive sensing, such as care homes (where patients are supervised and tracked round the clock, with wearables in certain cases), and detention centres - for real-time monitoring. Other applications such as Assisted Ambient Living (AAL) and Activities of Daily Living (ADL) in smart homes typically call for non-intrusive sensing. Here, sensor positioning (i.e. the location of devices) is a major consideration. To ensure continual usage, (a) these devices must accurately collect data and be accessible enough for maintenance and (b) these devices must be sufficiently concealed and/or be designed functionally as well as aesthetically to prevent any hindrance to the regular functioning of the household.

- *Single vs. Multi-User sensing*
  Recognition of group activities is fundamentally different from single, or multi-user activity recognition in that the goal is to recognize the behavior of the group as an entity, rather than the activities of the individual members within it. Choosing which of the two could be a major consideration in applications such as forensics and AAL.

Depending on the end-goal, research in this field typically targets a specific demographic of people.

To suit the technical skills and needs of wider audiences, this paper outlines yet another important category/consideration — the *Data acquisition* methodology.

- *Device (physical) vs. Network data*
  Smart homes usually contain off-the-shelf IoT devices from more than one ecosystem, for example, a typical household might have an Alexa Echo Dot voice assistant, an LG smart TV, a Withings BP Monitor and a Weekett Kettle. It can be a challenge to extract meaningful data from (such) a diverse collection of smart home devices because each ecosystem will have its own data storage method. The standard method then, to collect readings from this set of devices is to (a) collate readings individually from each respective smartphone application or web application, or (b) connect these devices to a central hub such as OpenHAB or Home Assistant and download all readings across the period of interest from the add-on database offering. If ad-hoc sensing devices such as arduinos or NodeMCUs are used in the smart home to measure temperature or humidity data, live data from these sensing devices can be logged using Google Sheets. This paper groups the aforementioned methods under a sub-category called *'Device Data'* under *Data acquisition*.

  An alternate, albeit indirect, technique could be to extract readings through network activity. This paper places this technique under the sub-category *'Network traffic data'*. This method attempts to draw the same conclusions about user activities using (potentially encrypted) network traffic as opposed to data from source. This is particularly useful for applications that are more autonomous - i.e. those requiring less initial input from users. The advantage of this method is that new devices could be automatically detected and device states potentially inferred as accurately as the usual standara, from network traffic. Moreover, irregularities in network traffic, which could potentially be the work of cyber criminals, could also be spotted using this technique. There are drawbacks to this method as well — network traffic does not reveal the location of the device in question within the network, this can make it harder to ascertain which activity is taking place and where. For example, the activity of visiting the cloakroom could be defined by two sequential actions: 'door contact sensor state = Open' and 'philips hue bulb = On'. In this scenario, a smart home with 2 Philips Hue bulbs and/or 2 door contact sensors would cause confusion if an activity in another room, such as 'visiting the guest bathroom', was defined in a similar fashion. In this instance, some level of user input would be required to map IP or MAC addresses of the device to the room location.

2.2.4 Common challenges in HAR

The data acquisition and processing stages can present several challenges. These include: (a) *The challenge with data acquisition itself:* As data is collected indiscriminately for all users in a smart home, it can be difficult to ascertain which user the data belongs to. Secondly, in terms of the method of extraction, readings extracted directly from the device or indirectly via network data would require some level of user input and/or specific hardware in certain cases (ex. for Bluetooth and ZigBee interfaces). (b) *The issue of data storage:* Forensic records may require data from varying timelines. Questions such as, 'How long to persist data for?' and ' What happens to older logs?' would need to be answered. (c) *The issue of activity ambiguity:* When activities are performed around the house there is no clear indication when one activity ends and another one starts, moreover, there could be ambiguity in the observed sensor data with respect to which activity is taking place. For example, cooking and getting a drink both involve opening the fridge. (d) *The issue of noisy data:* Human error could lead to faulty readings (like opening a wrong cupboard, entering the wrong room); similarly a power cut could lead to lost network packets and hence a gap in sensor readings, (e) *The issue of imbalanced data*: Class imbalance (and therefore biased activity recognition models) could occur when an activity is being performed for a longer duration than others, for example, a participant may 'work at a desk' longer than he/she 'drinks water'.

2.3 Machine Learning approaches in HAR

Different data collection, data preprocessing and machine learning algorithms can effect a model's capability to recognize activities. Therefore, data capture, analysis and feature extraction requires an expert-level understanding of human activity for both healthcare and social science applications.

Current and previous works in sensor-based HAR use both heuristically derived handcrafted feature-based traditional modeling approaches (such as those used in statistics and stochastic process like Decision Tree, Hidden Markov models, kNN, SVM, Ensemble models), as well as hierarchically self-evolving feature-based deep learning approaches (like Convolution Neural Networks (CNN), Long Short-Term Memory (LSTM), and Recurrent Neural Networks (RNN)) to classify data.

Some of these approaches are outlined below:

2.3.1 Shallow learning

In HAR systems designed using shallow learning, the commonly used features are time domain features such as mean, variance, and time sequences, frequency domain features and other transformations (wavelet transform) (Ramasamy Ramamurthy and Roy, 2018). Tree-based models such as Random Forest and Decision Trees, Boosting algorithms like CatBoost, XGBoost, and Gradient Boost, and Ensemble models are typically used in such systems, however, care must be taken to maintain these models over time, through periodic training and testing, to ensure prediction accuracy stays consistent or improves when new data — brought by changing behaviour patterns or new sensors — is made available.

2.3.2 Deep learning

In deep learning, the features are learned from the raw data hierarchically by performing some nonlinear transformation. The nonlinear transformation determines the type of deep learning network. Popular deep learning models include LSTM, CNN, and RNN. The drawback of using deep learning however, is that the results of the activity recognition algorithm remain largely unexplainable.

2.3.3 Transfer learning

In Transfer learning, annotation requirements and computational costs in new environments are reduced during model training as this technique reuses existing information from a previously trained model. As demonstrated in (Wang and Miao, 2018), transfer learning can only be leveraged if there exists some kind of relationship between the source (i.e., original smart home environment) and target areas (i.e., new smart home environment) which allows for the successful transfer of knowledge from the source to the target.

2.3.4 Active learning

Like Transfer Learning, Active learning algorithms also aim to mitigate the learning complexity and cost (Cherman *et al.*, 2016). These algorithms help select an optimal number of informative unlabeled data samples and then query the user for labels.

2.4 Conclusion and Discussion of Scope of Work

This paper is interested in how data is collected and general activities inferred for any smart home environment. To do so, the author aims to meet, if not improve upon, the state-of-the-art in activity recognition, with the only deviation from literature being that the input datasets used will be those of network data instead of device data.

To meet the challenges described in Section 2.2.4, whilst adhering to the time constraints set for this dissertation, this paper will narrow the scope of research to a single-user application. Moreover, it will attempt to classify activities in, or close to, real-time and will create a log of these inferences. To meet the general requirements of a wide range of use-cases, this paper will look at (unobtrusive) text-based data as opposed to image-based data. To be scalable and semi-automatic, this paper will deviate from the standard data acquisition technique of using sensor readings from source, and will instead use ubiquitously available network traffic data. Finally, to mitigate activity ambiguity, this paper will make use of binary sensors, and will track events alongside their timestamps.

As for the Machine Learning approach, this paper will further examine related works for the current state-of-the-art techniques in network-based HAR, and will develop a suitable methodology based on these findings.

Chapter 3

Related Work

3.1 Introduction

This chapter provides a detailed overview of different device and activity recognition tools and research approaches. The literature's are compared and analysed to finalise the design of the project. Finally, existing datasets within the field of device-identification are reviewed in order to finalise a 'reserve' dataset for this project.

3.2 Existing Literature

Whilst research in sensor-based HAR is growing within the smart home realm, recognition through network data, instead of the de-facto method of using device data to gauge activities is rare. To address the challenges and limitations of sensor-based HAR, parallel work in device *free* HAR (Cui *et al.*, 2021) (Damodaran and Schäfer, 2019) has gained traction in recent years. To gauge activities, these solutions make use of commonly used devices like laptops, PCs, smartphones and tablets to extract features from Wi-Fi signals like Channel State Information (CSI) and Received Signal Strength (RSS), on which HAR models are run. The limitation of these technique however, is that the range of recognisable activities is limited to single-motion activities such as walking, standing, sitting and running.

While the use of network traffic from commercial off-the-shelf (COTS) devices could be used to gauge a wider set of activities, a (potential) limitation is that the process sets the standard activity recognition process back by at least two steps. As demonstrated in (Acar *et al.*, 2020) and (OConnor *et al.*, 2019) this problem can be thought of as a multi-stage multi-class classification problem — i.e. to be able to extract similar human activity patterns as the de-facto from network data, the type of device and the state of the device will first need to be ascertained. After this, the same logical framework as the state-of-the-art can be used to club concurrent device states together (according to their timestamps) to form a composite activity. The advantage, however, of these preliminary steps (viz. device fingerprinting and device state recognition) is that they can be used to identify unknown, unauthorised or malicious devices within the home network.

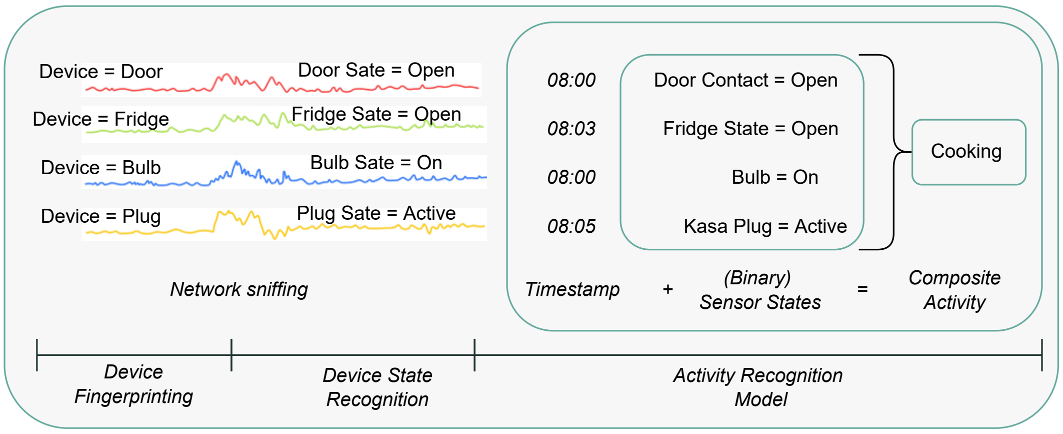


Figure 3.1: *High-level logic for sequential activity detection using network data*

As this section reviews research in network data within smart homes, it discusses the state-of-the-art in ascertaining device type (a process called device 'fingerprinting') and then discusses device state recognition. It then looks at wider activity recognition methods using both device data and network data.

3.2.1 State-of-the-art in Fingerprinting Techniques

Device fingerprinting (or 'DFP') is the process of identifying devices in the network without using common, spoof-able, identifiers such as IP address, Medium Access Control (MAC) address, Electronic Serial Number (ESN), International Mobile Station Equipment Identity (IMEI) number or Mobile Identification Number (MIN). DFP instead identifies a device by using implicit identifiers, such as network traffic (or 'packets') or radio signals. Such identifiers are closely related to the device hardware and software features and are almost impossible to alter/manipulate (Chowdhury *et al.*, 2020). Within network traffic, IoT traffic typically constitutes background traffic generated by the device autonomously (e.g., NTP queries for time synchronization) and traffic generated due to user interactions (e.g., the smart camera transmits image data to the cloud server when home invasion occurs) (Sun *et al.*, 2019). DFP can be done through both active and passive techniques: Active techniques involve actively sending TCP/ICMP packets to the unknown device and assessing the responses against some known baseline (or 'fingerprints'), while passive techniques are stealthier; they involve sniffing network traffic to analyse patterns and information.

Active probing leverages banner grabbing and other fingerprinting approaches to identify exposed IoT devices, however these techniques fail to identify IoT devices hosted behind NATs or firewalls

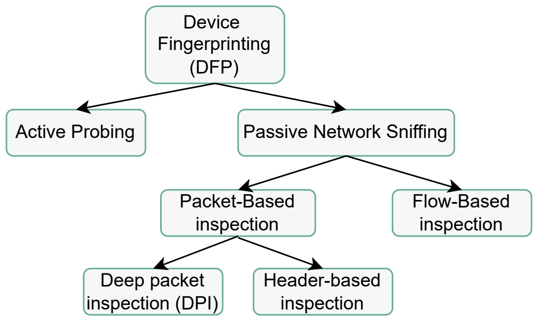


Figure 3.2: *Device Fingerprinting Techniques*

DFP techniques can be both active and passive in nature: Active techniques involve actively sending TCP/ICMP packets to the unknown device and assessing the responses against some known baseline (fingerprints). For stealthy detection, passive scanning is preferred. This method relies on sniffing techniques to analyse the information sent in normal network traffic.

(Perdisci *et al.*, 2020). Moreover, active techniques are usually tailored for specific IoT devices; in some cases, this might unintentionally activate devices as well (Bremler-Barr *et al.*, 2020).
Passive techniques in DFP use either packet-based features (Chowdhury *et al.*, 2020) (Aksoy and Gunes, 2019) or flow-based features (Sun *et al.*, 2019) (Shahid *et al.*, 2018) (Acar *et al.*, 2020), or both (Hamad *et al.*, 2019). Packet-based features use the content of individual packet payloads and headers, while flow-based uses statistical and temporal features such as packet flow direction, inter-arrival time and inter-packet length. In packet-based analysis, a common technique is to appraise the Network layer (IP and ICMP), Transport layer (TCP and UDP) and Application layer (DNS, HTTP, TLS/SSL and DHCP) protocols header felds (or features) from TCP/IP header packets (from device originated network packets) to learn the most suitable subset of features among a large number of features to train classification models. These methods typically exclude MAC and IP addresses from the feature set. Results from these works show that the features with the highest 'feature importances' include TTL (Time To Live), source and destination ports, TCP Window Size (Chowdhury *et al.*, 2020).

Flow-based features, on the other hand, are based on time-series features of traffic from the same device, some of which include packet flow direction, size of packets sent and received (mean packet length and median absolute deviation of packet size), inter-arrival time and inter-packet length. Some papers use flow-based information extracted during periodic time windows (such as 10s or 30s intervals), as in (Acar *et al.*, 2020) and (Shahid *et al.*, 2018), while others,like (Sun *et al.*, 2019) make use of statistical features from the entire flow such as the number of inbound/outbound packets and bytes, the total duration of the flow in seconds, the mean packet size, and the peak and mean packet rate. Entire flows also include the initial TLS/TCP handshake which includes plain-text data. Whilst this enhances the accuracy of fingerprinting and even enables high malware detection accuracy on encrypted traffic, these methods are impractical for real-time applications, as flows can last anywhere between a few hours to a few days. Entire-flow based classification may not be necessary however, as results from (Shahid *et al.*, 2018) show that classifiers trained on shorter intervals of flow-based traffic can reach accuracies of 99.9% using tree based classifiers. Regarding feature importances, results from (Hamad *et al.*, 2019) and (Shahid *et al.*, 2018) suggest that the features with the highest 'feature importances' include maximum, mean and variance of packet sizes, however 'feature selection' or extraction may not be necessary depending on the type of classifier chosen.

While most research work in this realm attempts to improve device identification accuracy against the benchmark, very few outline model performance over time. (Kolcun *et al.*, 2021) states that on average the models' accuracy degrades after a couple of weeks by up to 40 percentage points. The authors contend that the network traffic generated by these devices changes over time and therefore a single model cannot stay accurate for a longer period of time. To maintain accuracy these models thus need to be continuously updated.

3.2.1.1 Commonly used ML algorithms and datasets

DFP classifiers are constructed on (a)*classical machine learning techniques*: such as Adaptive Boosting (ABOOST), Latent Dirichlet Allocation (LDA), K-Nearest Neighbors (KNN), Decision Tree(CART), Extra Trees, Nave Bayesian (NB), Support-Vector Machines(SVM), RandomForest,Gradient Boosting (GBOOST), and (b)*neural networks*: like LSTM RNN and CNN.

As such, performance using both NN and classical methods remains largely similar; the advantage of classical methods over NN is less computational costs and better explainability, but more nuanced pre-processing requirements.

Regarding commonly used, publicly-available datasets, popular datasets to train and test algorithms include: (a) IoT Sentinel (Miettinen *et al.*, 2017) which covers network traffic from 31 COTS IoT devices and, (b) IoT Traces from University of New South Wales [Table 1].

Paper	Time-based		Packet-based		Flow-based	Algorithms	Dataset	Result (accuracy)
	Real-Time	Historic	Header	Payload				
(Chowdhury *et al.*, 2020)	-	✓	✓	-	-	Weka,	IoT Sentinel, UNSW Traces	S:83.35, U:97.78
(Dong *et al.*, 2019)	✓	-	-	✓	-	LSTM RNN	Local testbed	92%
(Shahid *et al.*, 2018)	-	✓	-	-	✓	Random Forest	Local testbed	99.9%
(Hamad *et al.*, 2019)	-	✓	✓	-	✓	OneVsRest Random Forest	IoT sentinel	90.3%
(Sun *et al.*, 2019)	-	✓	✓	-	✓	Multi-Layer Perceptron	(Sivanathan *et al.*, 2017)	99%
(Acar *et al.*, 2020)	✓	-	-	-	✓	KNN	Local testbed	93%

Table 3.1: *Survey of passive device identification techniques using network tracking*

3.2.2 State-of-the-art in Device State Recognition Techniques

Device state recognition attempts to classify binary states like 'On', 'Off', 'Active', 'Inactive', 'Armed' and 'Disarmed'. An overview of a few device states could be sufficient to draw conclusions about current activities like cooking, walking, sleeping, and eating within a home. More specific information like temperature or humidity readings is not possible to ascertain using this method, however this information is not crucial for general activity recognition.

The previous device fingerprinting process demonstrates that most devices exhibit unique (network traffic) behaviours and that these behaviours can be extracted from the metadata in instances where the payload is encrypted. (Acar *et al.*, 2020) and (Apthorpe *et al.*, 2018) suggests that interactions between devices and users create a significant increase in network data volumes as compared to the 'resting' state during which only the minimum amount of continuation packets like heartbeat messages are sent to minimize the device's power and bandwidth consumption. (Acar *et al.*, 2020) postulates that determining (binary) activity levels, i.e. active or inactive states, using volume analysis can be used to draw general conclusions about sequential activities such as walking through a hallway or heading to sleep. Determining certain device-specific activities is possible using traffic shaping as well; the authors of (Apthorpe *et al.*, 2018) were able to analyse traffic volume graphs to infer events such as 'went to sleep', 'temporarily out of bed' and ' out of bed in the morning' using a Sense sleep monitor (figure 3.5); determine 'off' or 'on' appliances from the traffic of a Belkin Wemo switch; and determine if users were monitoring video feeds or if motion was detected through the traffic of a Nest indoor camera.

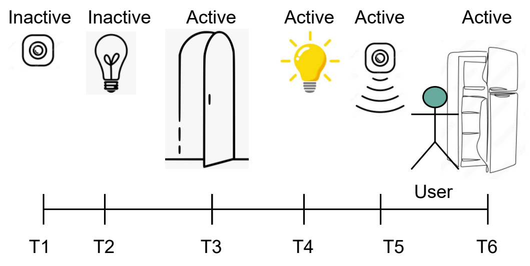


Figure 3.4: *Source: (Acar* et al.*, 2020)*

Detecting sequential activities like 'getting a drink of water', through activity levels.

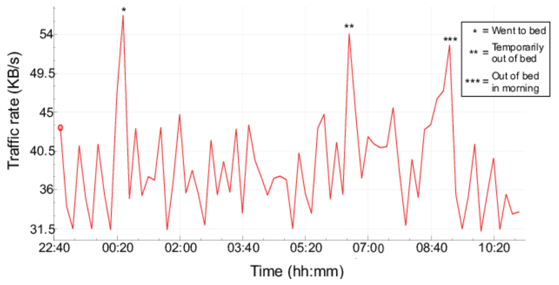


Figure 3.5: *Source: (Apthorpe* et al.*, 2018)*

Using network volume analysis to determine key states in sleep sensor.

However, such volume-based inferences cannot be used to describe more specific, non-binary, IoT events like colour and intensity. Additionally, peaks in graphs such as 'On' and 'Off' might be hard to distinguish. Moreover, a volume based analysis will need additional safeguards against packet padding, traffic shaping and traffic injection.

Works such as (Trimananda *et al.*, 2020a), (OConnor *et al.*, 2019) address this limitation; they identify that the client (the IoT device) and the server take turns in a request-reply communication style, and that this communication pattern is unique to device and device events. To demonstrate this, (Trimananda *et al.*, 2020a) extracted packet-level 'signatures' for device events which consisted of simple sequences of (unique and deterministic) packet lengths and directions, from packet pairs which consisted of a request packet from a device/phone, and a reply packet back to the device/phone. The authors were able to achieve an average recall of 97% using this method. The authors in (Acar *et al.*, 2020) also extracted similar matrices of timestamp, direction and packet length, from traffic data and attempted to use statistical time-series features such as 'Length of time-series, Mean and median of time-series, Skewness of time-series' to distinguish different states such as 'On/Off', 'Live View' and 'Measure Weight'. The authors were able to achieve 94% in F1 score and recall using this technique.

Irrespective of volume-based, packet-signature-based or statistical device-state recognition methods, characterizing IoT information exposure at scale is cumbersome: as outlined in (Ren *et al.*, 2019) it requires manually setting up large numbers of devices, using carefully controlled interactions with them, and capturing the salient network traffic they generate. To address scalability, these works then develop open-source repositories to encourage crowd-sourcing or the use of security service providers (OConnor *et al.*, 2019) for data sharing.

3.2.3 State-of-the-art in Activity Recognition Techniques

Sequential or compound activities can be modelled if labels such as device location, device activity and timestamps are made available. Two categories of research, distinguished by their data acquisition methodology, exist in this space: (a) research in privacy and exposure that looks at the extent of (HAR) inference possible from encrypted network traffic, and (b) research in sociological and behavioural HAR which uses directly-sourced sensor-based data to derive deeper insights.

For the first approach, researchers typically set up privacy attacks to gauge exposure from 'traffic shaping' experiments - and then seek ways to minimise this exposure ((Apthorpe *et al.*, 2018), (Apthorpe *et al.*, 2017), (Copos *et al.*, 2016)), (Acar *et al.*, 2020). The strategy these papers employ to gauge ongoing sequential activities is to first gauge which devices are active and interacting concurrently. Each device's activity is thought of as a 'sub-event'. Sub-events are captured by checking for changes in individual device activity states. Timestamps are then captured for when these sub-events begin (device state=On) and end (device state=Off). The location that each of these sub-events occur in are then tracked. The overall event or activity (e.g. 'Walking through a Hallway') is then logically put together by stringing these sub-events together meaningfully, for example, Hallway light=ON, Motion sensor=ON, Hallway light=OFF. These papers also note that some user-activity events are simpler to track. Activities like 'boiling water' or 'sleeping' can simply be ascertained through changes in individual device states. Here 'Kettle = On' and 'Sleep mat = On' could provide enough information to track the ongoing activity.

In sensor-based approaches, device metadata (labels) like location, states and names are made available from the start. Apart from modelling sequential activities (like above) using ML techniques like deep learning (LSTM) (Fang and Hu, 2014), Hidden Markov Model (Kabir *et al.*, 2016), and Naive Bayes Classifier (Shen and Fang, 2020), some papers extend the scope of HAR to include transfer learning (2.3.3). (Wang and Miao, 2018) suggests that an activity recognition system used by various smart home environments with potentially different sensor networks and/or label spaces (i.e. activities) can be built through this technique. Here, the authors are successfully able to use pre-labelled sensor-based readings to recognise activities in entirely new smart home settings. This approach has not yet been tested using network traffic data, however, from the results achieved in Chapter 07, it can be assumed that high-level traffic-based activity recognition is transferable till at least the device state recognition phase.

3.3 Conclusion and Evaluation of Literature Discussed

This chapter evaluates the current progress in literature against each of the different stages in network traffic-based HAR, which is defined in (Acar *et al.*, 2020) as Device recognition, Device State recognition and User Activity recognition. The results from these findings establishes a benchmark for the rest of this paper.

As an extension to this review, it is important to note that none of the experiments discussed have been developed to be deployed in a live environment. As a majority of these models are trained on datasets captured from lab-based environments — the lack of background noise from non-IoT devices in these environments could skew the models' performance when deployed in a real smart home. To this end, this Thesis will look at real-time device state recognition 'in the wild'. The strategy of multi-stage classification from (Acar *et al.*, 2020) will be continued in this dissertation work, however, emphasis will be given to the feasibility of productionalising such a tool. To mitigate the challenge of changing traffic patterns over time (highlighted in (Kolcun *et al.*, 2021), an attempt will thus need to be made to continuously update these models with real-time data, using user-input, at the edge.

CHAPTER 4

METHODOLOGY

4.1 Introduction

The state-of-the-art in IoT privacy-based literature suggests that user activity recognition through passive network sniffing can be approached as a multi-stage process: (a) device recognition (b) device state identification (c) device activity recognition (d) user activity recognition.

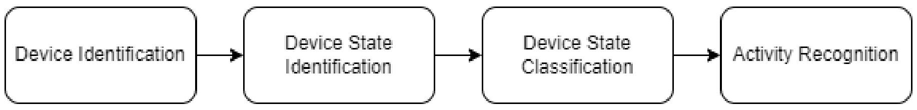


Figure 4.1: *Multi-stage process for activity identification*

This thesis uses this methodology to develop a generic real-time classification tool that can be deployed from the user's end. To predict generic user activities and provide real-time insights, this paper extends this pipeline to also include data analysis and (physical) anomaly detection in the smart home.

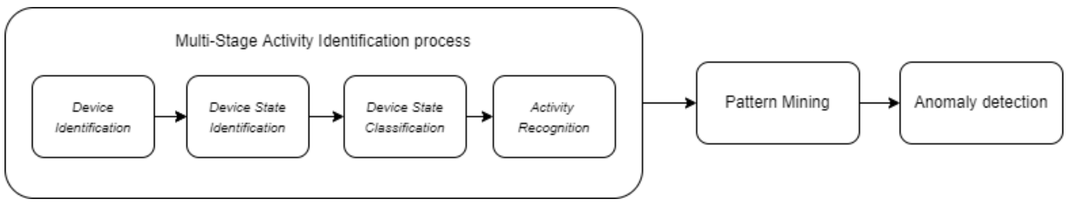

Multi-Stage Activity Identification process
Device Identification
Device State Identification
Device State Classification
Activity Recognition
Pattern Mining
Anomaly detection

Figure 4.2: *Pipeline for user activity inference tool*

This chapter introduces the larger pipeline that facilitates each of the stages described in Figure 4.2.

4.2 Overall Methodology

Activity recognition, for the most part, uses the supervised machine learning paradigm. Output features like device names, device activities and device locations will need to be labelled manually to some extent to train ML models to predict them. This process is time-intensive and can quickly become impractical as the number of devices and/or the number of spaces grow. To reduce the amount of initial manual labelling required, an option could be to train models on a few samples initially and then make use of the APIs of platforms like Home Assistant or OpenHAB to extract labels from previously 'unseen' devices. Another approach could be to ask for end-user input each time a *new/unseen* type of activity or location comes up. A sample could be determined as *'unseen'* if the generated prediction probability score does not meet the prediction probability acceptance threshold assigned by the developer for that label.

A simple overall process diagram that incorporates both options is described in Figure 6.2.

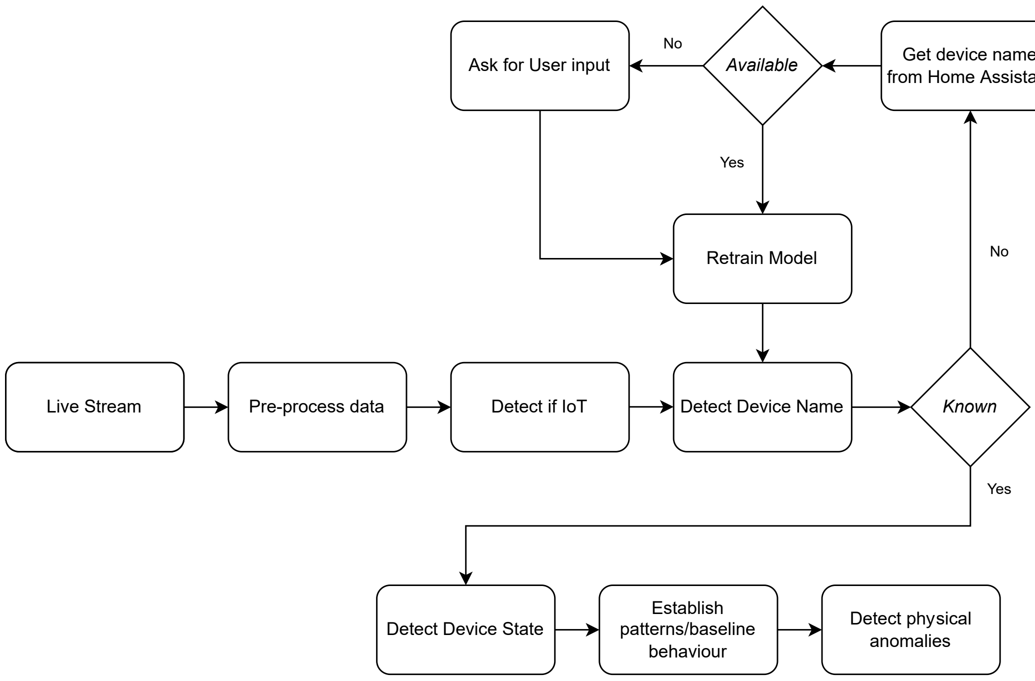


Figure 4.3: *Overall Process Diagram for a generic Activity Recognition 'Add-On' for Home Assistant.*

This diagram provides a high-level overview of what an end-to-end ML pipeline would look like if its pre-trained classifiers were supplemented with data from the Home Assistant API and if it also included scope for user input. This end-to-end deployment pipeline starts with a live packet capture from the local network (here, 'Live Stream'), which is then pre-processed by converting to a set of statistical features, which then serves as input to a set of pre-trained classifiers (here, 'Detect if IoT' and 'Detect Device Name'). The second classifier then attempts to fingerprint the device (here, 'Detect Device'). If the device name is known, data is then passed to another classifier to determine the device's state. After this, insights on usage are provided and physical anomalies are detected. If the device however is unknown, data is then retrieved for that IP address from Home Assistant using its API. If the IP address doesn't exist in the Home Assistant database, users are then prompted for input. This manually inputted label is then fed back to the initial (device fingerprinting) classifier for partial retraining or incremental learning *using methods like 'Warm Start' and 'Partial fit' from the Sklearn library.*

Such a deployment that includes device data brought in from a locally installed software hub like Home Assistant would be best suited as an 'Add-On' or plug-in to that software hub. The first benefit of such an add-on is that users could indiscriminately gather high-level information on all their IP connected devices, without having to specifically subscribe to the cloud version of these software hubs - thus making the diagnostic capabilities of the hub truly open-source. The use of network monitoring would also help users of such hubs to gain critical insights into all their IP-connected devices; direct comparisons between device-based readings and network-based readings of the same device could help highlight both cyber and physical anomalies in the smart home. This would prove useful to a wide-range of user groups including forensics researchers and building management.

The challenge, however, with developing a plugin for an established software hub like Home Assistant and OpenHAB is its currently limited user base.
Home Assistant and OpenHAB are not (at present) trivial applications to install and use. These require at least a basic working knowledge of the Linux operating system to install and to troubleshoot errors. These are therefore predominantly used by IoT hobbyists and not general laymen. A standalone application would therefore benefit a wider audience. Such a tool would largely replicate the diagnostic capabilities of both software hubs - which is to provide insights into their current and historic states. A notable dis-benefit however would be the loss of home automation capabilities that an integrated tool would provide.

The process diagram for such a tool [Figure 4.6] would exclude the 'Home Assistant API' phase of the previous diagram.

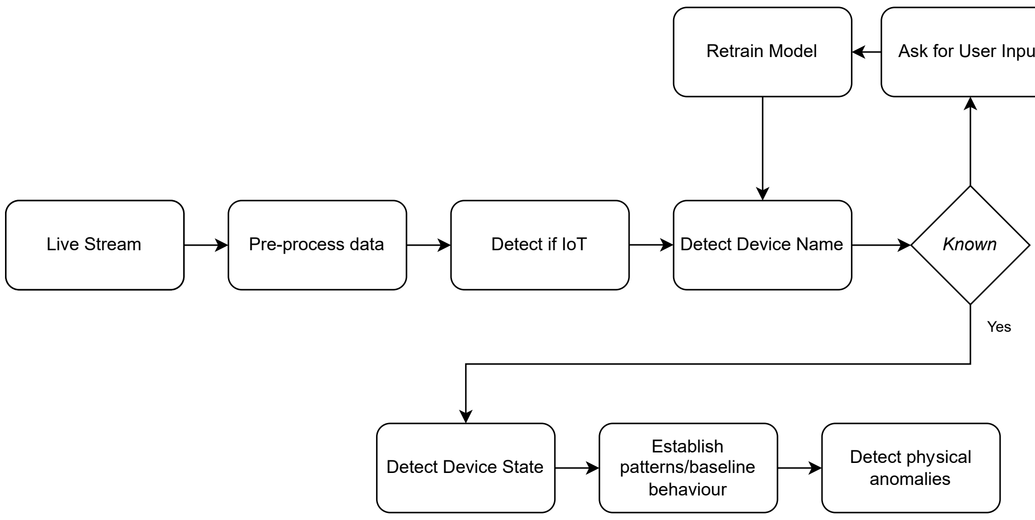


Figure 4.5: *Overall Process Diagram for a generic standalone Activity Recognition tool*

Like in (Acar *et al.*, 2020), the drawback of the 'cascading classifier' technique in Figure 4.6 is that the accuracy of the final prediction is dependant on the probability of getting an accurate reading at each of the stages (i.e. probability(final) = probability(stage1) x probability(stage2) x probability(stage3). This tool will therefore look at uncoupling these classifiers so as to independently predict labels like IoT/NoT, Device name, Device State and Device location for each device. Users could then be asked to verify these outputted labels periodically.

The proposed process diagram followed by this paper for the deployment is as outlined below.

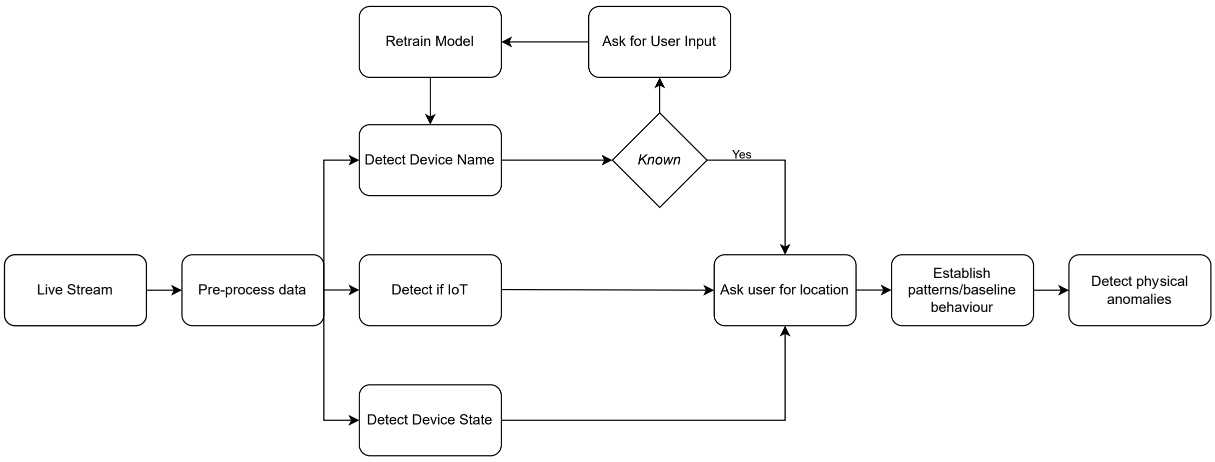


Figure 4.6: *Proposed methodology for the standalone Activity Recognition tool*

Each of the classifiers are de-coupled in the proposal. The 'Device Name' classifier is a multi-class classifier whilst the 'Detect if Iot' and 'Detect Device State' classifier are binary classifiers. Users are asked for confirmation or label input from the multi-class classifier based on a pre-set probability threshold. Users are then asked to input the device's location. Insights are then generated based on current device patterns.

4.3 Multi-Stage classification methodology for Activity Identification

The proposed tool will thus rely on three pre-trained classifiers (datasets outlined in Section 5.3) that are capable of being re-trained online automatically as new data becomes available. Once deployed online, real-time data (here, local network traffic) will be made available to these classifiers to serve predictions on. This local traffic —consisting of IP address ranges starting from '192.168' or '10.0— will be captured as .pcap files using open-source packet capture tools. These files will then pre-processed to CSV files containing a specific set of traffic and 'flow'-based features (Appendix A.0.2), that each of these classifiers have previously been trained on. These newly created CSV files will then be fed into each of these classifiers after which label predictions will be generated if they meet a certain acceptance threshold.

Each of the stages are described below:

4.3.1 Stage 1: Detecting IoT versus Not IoT

This stage will attempt to determine if each 'discoverable' device in the local network is an IoT device or not. This is thus a binary classification problem. For this, a suitable classifier will be trained on 91 packet features (Appendix A.0.2) that are extracted from pre-existing .pcap files. The top classifier will then be 'pickled' and deployed online to serve predictions (along with their probability scores) on real-time packet captures.

Note: The feasibility, here, of training a classifier on a packet-based dataset is outlined in (Bremler-Barr *et al.*, 2020), (Bezawada *et al.*, 2018) and (Aksoy and Gunes, 2019) and further experimentation in this dissertation work corroborates these results. The performance from a range of classifiers along with that of the top performing binary classifier is detailed in Section 7.0.1.

4.3.2 Stage 2: Device Identification

This dissertation finds that while command line applications like *arp-a* and *nmap* can be used to map IP and MAC addresses, these do not provide a complete list of *every* device connected to the network. Only the router can provide a reasonably conclusive list. SNMP can report a routers DHCP lease list, and even so, IP spoofing can affect this IP mapping. The MAC address itself can also be used to look up the manufacturer or vendor name of the network adaptor through its first 6 letters (which is its Organizational Unique Identifier), however this is hardly a differentiating factor for multiple devices with the same adaptor.

This stage, therefore, will attempt to determine individual device names (e.g. 'Blink Camera', 'Alexa Echo Studio' etc.) based on traffic patterns. This is essentially a muti-class classification problem and sits under the supervised learning paradigm. Here 'flow-based' traffic —which is a sequence of packets carrying information between two hosts during a set time interval— will be used to train a suitable multi-class classifier. As highlighted in Figure 4.6, this tool will also need to inform the user when previously unseen devices get connected to the local network. As suggested in OConnor *et al.* (2019), this design constraint motivates an ensemble or averaging based classifier, as they are more resistant to under-fitting. A range of such classifiers will therefore be tested in this paper. To indicate the presence of unknown behaviours, the probability prediction score will be used to define a threshold of acceptance.

Note: The justification for choosing flow-based instead of packet-based features to differentiate IoT devices comes down to the difference in results achieved by both methods in Section 3.2.1 of the Related Work Chapter. The final time interval chosen by this paper to extract flows, along with the tools chosen to capture these flows, are further outlined in Chapter 5. The results from the top-performing classifier are later outlined in Chapter 7, Section 7.0.2.

4.3.3 Stage 3: Device State/Activity Identification

This stage will attempt to distinguish between the 'ON' and 'OFF' states of an IP connected device. Experiments in IoT traffic-shaping suggest this is possible (Section 3.2.2) and results achieved in Section 7.0.3 corroborate this. Flow-based features will again be used to train binary classifiers to detect if devices appear active —which is characterised by random peaks of traffic— or inactive, but locally discoverable on the network —which is characterised by a consistant stream of fixed-length packets sent at fixed intervals. A drawback, however, of using this technique is that traffic flows will be extracted after a set duration of time, therefore the exact moment when the switch between the On and Off stages occurs will not be captured, which can be a deterrent for time-sensitive use cases. For this *generic* use-case, a range of binary classifiers will thus be tested on flows extracted from different time intervals.

Note: The top performing classifier that uses the least flow interval period is outlined in Section 7.0.3.

4.3.4 Stage 4: User-Activity Identification

This stage will attempt to classify user activities. It is essentially a sequence classification problem, however, for the purpose of this paper, less complex activities like “walking through a hallway”, “taking a shower”, “sleeping”, “preparing breakfast”, and “spare time/TV” will be gauged through rule-based techniques. An example of a possible sequence of activities that can be modelled based on a simple rule-based algorithm is: ’Given Kitchen Light = On, Fridge door = Open, Stove = On, Predict Ongoing Activity = ’Cooking”, and additionally: ’Given ’activity’ = ’cooking’, Predict ’Active room’ = Kitchen’ ’.

Papers like (Wang and Miao, 2018) additionally make use of different time period labels (like morning, noon, afternoon, evening and night) to gauge probable activities. This technique will be used here to ’guesstimate’ ongoing activities as well. As an extension, this will also be used to gauge anomalous physical events such as ’Garage Light switched ON at 02:00’, or ’Bedroom Light OFF for 12 days’.

Note: This stage has scope for further research — this is further outlined in Chapter 08.

4.4 Conclusion

This chapter presents this Thesis work’s proposed development methodology in the form of a high-level multi-stage process diagram. It outlines the requirements of each of these stages, based on the current benchmark established in Chapter 03. It also notably prescribes the use of both packet-based and flow-based traffic to train a set of binary and multi-class classifiers. This chapter notes that whilst the proposed method serves predictions on live streams, the prediction method itself may not be perfectly ’real-time’, and will depend on the results achieved in Chapter 07.

The chapter also alludes to the need for user input to both confirm and label any ’unknowns’. The smart home market is set to grow by 27% in the next 8 years (Sta, 2022 - 2030) — the resulting proliferation of new IoT devices in the market will mean that that is high likelihood of numerous missing labels even in extensively labelled datasets. To catch any 'unobserved' values and label them as 'Unknowns', an accceptance threshold will therefore need to be defined at the time of training.

Chapter 5

Data Preparation

5.1 Introduction

This chapter discusses how data is collected and processed before it is inputted into the system: it briefly describes the Smart Home Network setup for data capture, and then outlines the data collection and data conversion techniques, along with tools used. This chapter also describes how data is pre-processed to be used as initial training data for the ML classifiers described in the previous chapter [Section 4.3].

5.2 Developing the Smart Home Network

A 'smart home style' network is used in this research work to capture initial network traffic for training data. To emulate a real-world smart home, this set up includes a router, a set of IoT sensors and actuators, a set of non IoT devices, a host machine (PC), and a packet capture and parser tool. To capture maximum packets, the host PC in this setup is connected to the university's wider network (Eduroam) and is made to act as an internal 'modem' in the University's Smart Home Lab. A dedicated router is then connected to the modem via an Ethernet cable. This allows all packets reaching the router to be made available to the host machine via the 'Eth'/'eth0' network interface. Whilst a real-world deployment in a smart home may not be able to capture all packets transmitted over the network in this way, this initial step is taken to ensure all samples transmitted or received by an IoT device are captured for robust initial training.

5.2.1 Devices used

The devices used for data capture aim to include a representative cross-section of WiFi/Ethernet-connected IoT device types typically found in smart homes. Based on current market share trends (Jackman, 2022), the most popular device categories are smart TVs, smart speakers, smart lighting, smart security systems such as smart cameras and smart locks, e-health devices such as smart scales and blood pressure monitors, outlets and switches, gateways including hubs and routers, and appliances such as kettles, vacuums, dishwashers, fridges and coffee makers. The set of IoT devices thus chosen for data collection from the Smart Home Lab at Cardiff University include: LG TV, Alexa Echo Studio, Lumiman Bulb, Blink Camera, Withings Smart Scale, Withings Blood pressure monitor, Kasa Plug, Weekett Kettle, Roborock Vacuum, and a Smarter Coffee Machine. In addition to this, non-IoT devices are considered as well; this includes a Dell Laptop, an Android Phone, and a MacBook.

5.2.2 Tools used

Numerous packet capture tools with abilities to capture, analyse, create and even send packets are available in the open market. Some of the more popular open-source 'sniffing' tools include Wireshark and its command line equivalent tshark (Combs, 2022), the Scapy library (Biondi, 2022), and TCPdump (Van Jacobson and McCanne, 2022). All three have free versions for MAC, Windows and Linux-based operating systems. For the purpose of this dissertation however, tshark will be used because of its versatile packet parsing and analysing, and reading and writing capabilities.

5.3 Data Capture

In the design of the classification system, a full set of network traces is needed to train and test the system at each stage. In current literature, this process is approached differently based on the individual classification requirement; for example, for passive device fingerprinting, traffic is typically captured during the initial TLS handshake (which may be between 1-4 KB). This necessitates packet capture during the first one-two minutes of a device reboot or (new) connection. In contrast, a classification that is interested in detecting binary states (on/off) of a device might be interested in activity over time. Devices that are 'off' or 'inactive' (but still connected to the network) may still send and receive packets, however this typically happens at set intervals in specific patterns. Papers like (Acar *et al.*, 2020) run multiple On/Off experiments for average time periods of 2 hours to capture this periodic traffic. A classification that is interested in differentiating IoT from non IoT devices on the other hand, may be interested in capturing entire TLS sessions or flows, — these could last for days for certain devices. Some papers also use similar TLS sessions as initial training input for fingerprinting devices (Sun *et al.*, 2019). This, however, may not be practical in real-world developments as noted in Section 3.2.2. Similarly, while capturing initial handshake packets during reboot is feasible in a Lab setting, this may not be feasible in a smart home setting where devices may have already previously been connected.

5.3.1 Methodology for Data Capture from Smart Home Lab

To ensure that the quality/type of the initial training data meets that of the online (re-training) data when new devices become available, a generalised traffic capture methodology is proposed which seeks to capture *just enough* traffic to successfully (initially) train all three classifiers. For the time period of this capture, a median point between all three classifiers appears to be between 10 minutes to 12 hour (packet capture) duration. Although a very ambitious claim, 10 minutes of traffic may just be enough (depending on experimentation) to train classifiers to differentiate between devices —though of course, a longer capture may improve prediction performance scores. Moreover, if classification of new devices is possible with 10 minutes worth of traffic, lesser network traffic will need to be stored for later retraining. At the other end, anything longer than 12 hours may prove too memory intensive for most personal machines — findings from the Smart Home Lab suggest that on average, 8-10 IoT devices can generate up to 75000KB in 12 hours while dormant.

For this Thesis therefore, traffic is captured for varying durations, ranging from 10 minutes-2 hours for active experiments (like on/off), and for <12 hours for inactive/dormant captures, to get a sufficient record of heartbeat messages and other traffic patterns.

Traffic is captured using tshark, and saved as separate PCAP files for each individual device (identified here by source IP address). 91 individual header fields/metadata [Appendix A.0.2] from these PCAPs are then separately parsed using tshark and written to a separate CSV file. This packet-header data will be used to train the first *IoT versus Non-IoT* classifier, to test it against current benchmarks.

The other two classifiers (viz., the *Device Fingerprinter* and the *Device State Identifier*) will use features from 'traffic-flows' —generated from the original PCAPs— instead of packet metadata, as initial input. Here, 'flows' are defined by a sequence of packets with the same values for Source IP, Destination IP, Source Port, Destination Port and Protocol (TCP or UDP) (Palmieri and Fiore, 2009). This is so that the results from the testing phases from both classifiers are comparable with the current state-of-the-art.

To extract these flow-based features, this Thesis makes use of a specific network traffic flow generator —CiCFlowMeter (Hieu, 2022)— which facilitates both live and historic generation of bidirectional flows and outputs these as *features* [Appendix A.0.2] in a CSV format. To ensure these flows are limited to 10 or 30 second durations (as needed for the experimentation phase) between all endpoints, all PCAP files are first split into smaller 10 and 30 second captures using 'editcap' on the terminal [Appendix A.0.2]. Flow-based features are then extracted from these shorter PCAPs as CSV files using CiCflowmeter. These files are thereafter merged to form a single CSV file to use as raw training data. Here, each 10 or 30 second flow, acts as a *sample* for the classifiers. The reason here, for limiting flows to 10 and 30 seconds, is for the classifier to be able to predict statistics between 2 endpoints, *almost* in real-time.

Device	State	Time Duration	Size (Flow)	Samples (flows)
Kasa Plug	On	12 hours	585.2KB	662
Kasa Plug	Off	12 hours	590KB	663
Smarter Coffee	Active	(Avg 2-5 mins) Total 15mins	15.3KB	28
Weekett Kettle	Active	2 mins	8.84KB	10
Weekett Kettle	Inactive	6 mins	6.55KB	16
Roborock Vacuum	Active	10 mins	145KB	279
Roborock Vacuum	Inactive	4 mins	4.2KB	10
LG TV	On	5 mins	124KB	268
Lumiman Bulb	On	15 mins	25KB	47
Lumiman Bulb	Off	15 mins	24.4KB	47
Withings Blood Pressure Monitor	Active	5 mins	4.06KB	7
Alexa Echo Studio	Active	mins	913KB	1960
Alexa Echo Studio	Inactive	mins	238KB	511

Table 5.1: *Network traces from the Smart Lab at Cardiff University used by the Device recognition model. (Note: This does not represent the complete capture set.)*

5.3.2 External Datasets

Popular public network traffic datasets used by related works include *IoT Traffic Traces* from UNSW (A. Sivanathan and Sivaraman, 2022), the *IoT Sentinel* dataset (Miettinen *et al.*, 2018), and the *PingPong dataset* (Trimananda *et al.*, 2020b). In this paper, the UNSW dataset is used to supplement the existing dataset (captured from the Smart Home Lab at the University) for Stage 1 (IoT versus Non-IoT recognition) to ensure a wide enough sample range. The other stages do not require training from devices outside the Smart home Lab initially as they are due to be iteratively re-trained online using real-time data.

5.3.3 Methodology for Data Capture during Live Deployment

In general, an online tool can capture real-time traffic from the client's local network using the Scapy library. It can even use tshark from the user's terminal via the subprocess.call() method. To facilitate this however, the user will first need to *manually* set their network card or intended network interface (example, ethernet) to 'promiscuous' or 'monitor' mode to allow complete packet capture over the desired interface. This is because many operating systems require superuser privileges to allow packet capture from other devices in the network. The open source packet capture tool of choice (viz., Scapy, TShark, NFStream or TCPDump) can then be used to write these packet captures to local files or to online databases for further analysis.

The implementation in this paper follows all these steps. After user-permissions are granted, a python script uses the subprocess.call() method to utilise tshark to sniff and write packet captures to a .pcap file in 30 second batches. Each file is then automatically converted — both to a CSV file containing packet-headers (for IoT versus Non-IoT classification), as well as to a CSV file containing flow-based features using CiCFlowMeter (for device and device-state classification).

These files are then processed further, as defined in the next section, after which they are used by each ML model, either to serve predictions, or (after getting user-input labels), to further re-train the model itself.

5.4 Data Pre-processing

After both types of CSV files are generated, simple pre-processing techniques are used to convert these datasets into suitable training data. For easy manipulation, these CSV files are first

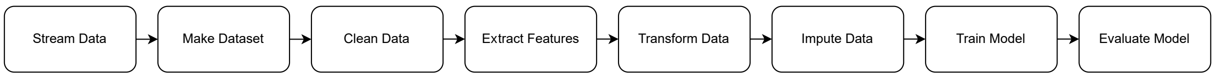


Figure 5.1: *General pre-processing methodology*

converted to Pandas Dataframes. These are then pre-processed using the scikit learn library: To clean the numeric (flow-based) datasets, any duplicates and samples with null values greater than 70% are removed. Device and Device state labels are manually added, and then encoded and transformed using the LabelEncoder method from the scikit-learn preprocessing library. These are then split into training and testing datasets in a 70/30 ratio using the train-test-split method. After this, using sklearn Pipeline, features with non-float data types are dropped (feature extraction): this includes identifiers like the source IP, destination IP and Flow ID. The remaining features are then imputed (using SimpleImputer where null values are assigned '0') and scaled (using StandardScaler). The packet-metadata dataset is similarly pre-processed, with an additional step of imputing missing labels with 'missing' and encoding all categorical features using OneHotEncoder within the Pipeline.

Hereafter, a range of classifiers are fit on this training data and evaluated using sklearn performance metrics such as accuracy, precision, f1 score, recall - where f1 score is given the most importance due to the imbalanced nature of IoT data. The models are also cross-validated using 5-fold cross validation. A confusion matrix and feature importance graph are additionally generated. Hyperparameter tuning is done for the top performing classifiers as well. This stage is further discussed in Chapter 07.

5.5 Summary

This chapter outlines how datasets are created for the ML training phase. The tools and techniques used for live network traffic capture, its conversion to flows, and the automatic extraction of features is defined here. Furthermore, a brief overview of the dataset pre-processing phase was covered as well.
Note1: The datasets used are available at (Dat, 2022).
Note2: The samples captured for this dissertation are not as extensive as the PingPong Dataset (Trimananda *et al.*, 2019). To improve the quality of this work, this dataset will benefit from a

larger number of structured 'On/off' experiments.
Note3: Efforts were also made to capture differences in specific states as well, such as 'Kettle cooling down', 'Kettle warming up', 'Kettle boiling' —however as this was done manually an extensive dataset could not be obtained.
In general, these traffic capture experiments will greatly benefit from automation; cron-based rules could be used to change device states automatically after set intervals. For this, automation hubs like OpenHAB and Home Assistant would be prove most useful.

Chapter 6

Pipeline Preparation and Application Development Methodology

6.1 Introduction

The previous chapter describes the steps taken to first convert an existing network traffic (pcap) dataset to statistical flows and then prepare this dataset through standard methods like imputation, scaling and labelling, for supervised training and testing. While this 'pipeline' can be manually executed on Jupyter Notebooks to build a suitably robust model for each of the stages each time new data is available, this methodology will not scale well in a real-time smart home environment where new, unknown, heterogeneous devices are likely to be connected in an ad-hoc basis.

This chapter thus outlines techniques used to productionalise ML development. It then attempts to build a tool that is (a) capable of serving predictions in real time, and (b) capable of automatic model retraining based on new input data. To this end, the best practices in ML Operations are explored, and the development of the application's features, along with the tools, frameworks and algorithms used in deployment is described.

6.2 Pipeline Development Overview

Deploying activity recognition models in a live setting is effectively an exercise in ML operations - which by its textbook definition is a set of practices that aims to deploy and maintain machine learning models in production reliably and efficiently.

While pre-trained models can be easily put in production to actively serve predictions, issues such as model drift could severely affect prediction accuracy as new devices enter the market. Depending on the business case requirement the logic for model retraining may vary; in cases like recommendation systems (such as in Netflix or Amazon), real-time incremental updates can be necessary, while in others, batch or trigger-based updates are often enough. Model retraining for device activity recognition could take the later (trigger-based) approach. In such a scenario, detecting unknown/new devices would trigger a partial retrain of the existing model to include future recognition of the new device and its state(s) patterns. To this end, libraries such as Scikit-Learn library will facilitate this type of retraining through approaches like 'Partial Fit' and 'Warm Start'.

The next challenge lies in deciding where to deploy this model. This could take two approaches: (1) a central model into which data from various 'smart' households are fed, or (2) an individual 'barebones' base model which is customised depending on the individual household's activities. As the proposed business logic in this thesis incorporates some level of user input, a central model would require dedicated administration/quality control to manage variations in user-defined activities across households. The resultant model may output generalised labels in this instance and extending such a model to incorporate customisation is outside the scope of this paper. As the proposed model puts activity definition in the hands of the (layman) user, this thesis thus takes the second approach - a general base model which is (custom) trained when new data becomes available on the user's end.

6.2.1 MLOps and Productionalisation

To deploy such a (set of) models, the general pipeline would be to set up the environment and create an initial ML model; then simulate streaming data coming in as users use the ML model and publish this to a Kafka feed; then, periodically extract data from this feed and use it to update the ML model, gauge its relative performance and put it up for use if it outperforms the current version of the model; and last, log the results, model parameters, and sample characteristics of each update run.

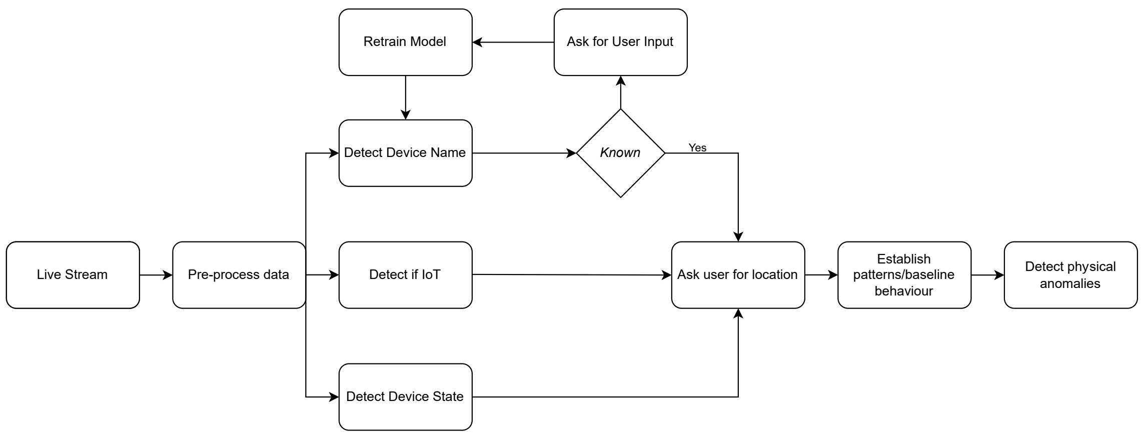


Figure 6.1: *Proposed Process Diagram*

To take this into production the model could either be (a) initially developed offline and later retrained online, or (b) developed and retrained offline (in batches or during specific times of the day), and then deployed online, or (c) Developed online using a workflow management platform like Apache Airflow, hosted online and then called using an API endpoint.

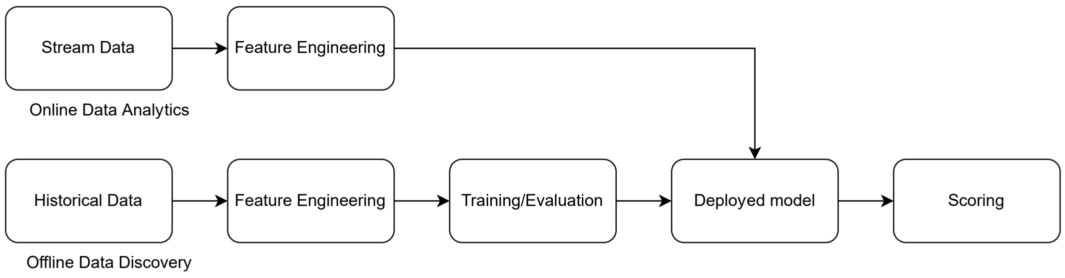


Figure 6.2: *Typical workflow for real-time prediction*

6.2.2 Pipeline orchestration tools

To avoid the consequential overhead from manually updating the model each time new data becomes available, open-source tools like Luigi, AirFlow, MLFlow, and KubeFlow, can be used to automate many parts of the ML pipeline development process. Amongst this, Apache Airflow is a well-documented task scheduling platform that allows users to create, schedule, orchestrate, monitor and maintain data/ML workflows through its UI. Here, separate data pre-processing, training and testing scripts can automatically be run (or 'orchestrated') via DAGs (Directed Acyclic Graph) in AirFlow, to train and retrain ML models. Developed models can then be hosted online using platforms like Docker or Heroku and API endpoints could then be used to serve predictions.

This thesis postulates that Flask - a lightweight python framework - can similarly be used to take ML development out of Jupyter Notebooks and into a production environment. Flask facilitates code modularisation and its python backend lends to its data science capabilities. Moreover it allows for the backend scheduling of tasks, and this can be orchestrated from the frontend as well. The advantage of packaging ML pre-processing, (re)training and deployment, together with the prediction and dashboarding components of the tool, is that such an architectural setup is easily portable/deploy-able and also facilitates label/prediction customisation for each user. While a separate, centrally located, ML deployment would alleviate the risk of a single point of failure, it will not allow for much personalisation - and hence is not explored in this Thesis.

While ML deployment via Flask is a well documented process, online development and online learning using Flask is less so. This thesis explores option 1 of the previous section, viz, '*ML models initially developed offline, deployed online and then retrained online*', to assess if online learning and data pre-processing using Flask is practical. This implementation is described in detail in the next section.

6.3 Application development methodology

6.3.1 Environment

Flask (FLA, 2022), a python-based micro web framework, is utilised to develop this project. The application is containerised using Docker to facilitate integration with Home Assistant. Information about current device states from Home Assistant can also be brought in using Home Assistant's REST (Res, 2022) and WebSocket API — this, however, is not demonstrated in this tool. Instead, additional user input is collected via Flask Forms on the UI — to demonstrate feasibility both as a standalone app and as a Home Assistant (HAA, 2022) Add-On.

6.3.2 Features and UI

Features of this tool include:

- *A real-time chart which provides live updates of data volumes transferred within the private network.*
- *A live mapping of currently connected devices' MAC addresses and their currently assigned IP addresses*
- *A view of historically connected IoT and N-IoT devices.*
- *A real-time view of currently connected devices — with IP addresses, MAC addresses, device names, device locations and device states.*
- *Interactive Line charts to display trends in activity patterns over custom time periods.*
- *Widgets that display ongoing activities and 'active' rooms.*
- *A 'Warnings' card that displays alerts and high-level anomalies.*
- *A User 'Input' card to enter values for 'Unknown' device names or locations.*

The methodology and tools used to develop these features are outlined below.

1. *A dynamic, real-time chart which provides live updates of data volumes transferred within the private network.* [Figure: Appendix Section A.0.4 (1), Code: Appendix Section A.0.6 (10.1)]
   For this feature, live network traffic is captured using the Scapy (Sca, 2022) library. The Sniff() function is used to parse both the total bytes in each packet and the timestamp of the packet, and this data is fed into a Dataframe which is then converted into the JSON format. This function is called every 2-3 seconds by a JavaScript script, which renders this data to a dynamic front-end line chart using the Highcharts (Hig, 2022) library.

2. *A mapping of currently connected devices' MAC addresses and their currently assigned IP addresses.* [Figure: Appendix A.0.4 (7), Code: Appendix A.0.6 (10.2)]
   IP addresses in domestic settings are typically dynamically assigned — these get re-allocated

when the device reboots or disconnects from the network. MAC addresses, on the other hand, are usually static - though these too can be altered which this tool addresses and mitigates through the next two features. For this feature, the 'ARP -a' command line functionality is brought to the front-end without any user input. The scapy.srp() method is used to return packets from OSI Layer 2. From this, the IP and MAC addresses are parsed and added into a dictionary which is then returned to the front-end. This function is called every 30 seconds by repeatedly reloading (or refreshing) its URL endpoint on Flask using a simple JavaScript function.

3. *A view of historically connected IoT and N-IoT devices.* [Figure: Appendix A.0.4 (7), Code: Appendix A.0.6 (10.3)]
   For this feature, Tshark is used to capture packets from the chosen interface in real-time. Metadata from 91 chosen headers [listed in Appendix-A] of this pcap file is then sent to a CSV file. From this file, only addresses that match the '198.168.'range of private addresses are parsed and then run through a LightGBM ML model (which is pre-trained on network traffic datasets from 23 IoT and 9 Non IoT devices from the SmartLab and UNSW), to classify them as IoT or Non-IoT devices, without using standard identifiers such as IP and MAC addresses. The same model that checks for IoT and Not IoT signatures also outputs the probability score for each prediction using the 'predict_proba()' method from scikit-learn, and this is displayed on the front-end as well. This function is also called every 30 seconds by repeatedly reloading (or refreshing) its URL endpoint on Flask using a simple JavaScript function.

4. *A real-time view of currently connected devices — with IP addresses, MAC addresses, device names, device locations and device states.* [Figure: Appendix A.0.4 (5), Code: Appendix A.0.6 (10.4)]
   For this feature, two pre-trained classifiers: a multi-class classifier that detects device names' and a binary classifier that detects activities (On/Off), are used to serve predictions for all source IPs that follow the '192.168' range of private addresses from the captured* traffic. These predictions are then written to a SQLite database* automatically. These results are then queried and only the latest states of each connected IoT device are displayed on the front-end HTML table — This is done using a simple SQL query that groups readings by MAC address, orders these by timestamp and then returns all unique MAC address row with the latest timestamp. This table is continuously updated in 30 second intervals using the same script as the previous functions/features.

   (Note*: The predictions, in particular, are served on statistical flows that are extracted from *30 second-long* packet captures. The process of capturing live network traffic and processing these into flows for immediate ML prediction is further outlined in Section 6.3.3.2.This section also outlines the data persisted in the SQLite database. The process of choosing both classifiers is outlined later in Section 7.0.2 and 7.0.3.)

5. *Interactive Line charts to display trends in activity patterns over custom time periods.* [Figure:

Appendix A.0.4 (20), Code: Appendix A.0.6 (10.6)]
For this feature, a simple SQL query is used to return historic device states for each identified MAC address. Users are able to select custom timeframes (hours/days/months) to view device trends. This is similar in functionality to tools like Home Assistant(HA) and OpenHAB(OH), however it has main two points of distinction, (1) this tool includes a built-in database which does not require separate setup like in OH and HA (further outlined in Section 6.3.3), (2) data from all internet-connected devices is persisted - this includes ecosystems like Nest, Alexa, and SmartThings, unlike HA which requires separate payment for this functionality. The drawback however is that data from devices that use protocols other than WiFi or Ethernet like ZigBee (e.g. SmartThings multipurpose sensors and Aqara devices) are not captured.

6. *Widgets that display ongoing activities and 'active' rooms.* [Figure: Appendix A.0.4 2]
For this feature, a simple SQL query to retrieve the data of all devices that are currently active, from the 'DeviceStates' table in the SQLite database. Graphic widgets for each active device's room/location (e.g. 'Living Room') and activity-type (e.g. 'Entertainment') are then highlighted in blue on the Home Page.

7. *A 'Warnings' card that displays alerts and high-level anomalies.* [Figure: Appendix A.0.4 4]
This feature displays energy alerts (e.g. devices that have been On for longer than 24 hours) and suspicious-activity warnings (e.g. devices that are active between the hours of 1 and 6 am) by retrieving the timestamps of each active device.

8. *A User 'Input' card to enter values for 'Unknown' device names or locations.* [Figure: Appendix A.0.4 (6), Code: Appendix A.0.6 (10.7)] The developed application renders any device recognition predictions below a 90% probability threshold as 'Unknowns'. A high initial threshold is chosen because the total number of initial devices and the total number of samples in the initial training dataset were both relatively low. Users are prompted to input device names for any of these 'Unknown' devices, and this starts the device recognition classifier's (partial) retraining process. The scikit-learn library facilitates this type of retraining through approaches like 'Partial Fit' and 'Warm Start', where models learn incrementally on new data without 'unlearning' from previous data. This effectively increases the number of devices the classifier can identify.

   The location variable on the other hand is manually input by users initially, with the capability to alter this at a later date. This label is used by a simple rule-based python function that tracks device states by location (e.g. 'bulb' = 'on' in 'Study') and then provides a generic 'guesstimate' about the type of human activity occurring, which in this instance might be 'working'.

   Both device Names and Device Locations are added to the database for each associated IP entry. These are then automatically retrieved and displayed on the front-end for user review during the next page reload (i.e. within the next 30 seconds).

6.3.3 Back End

A regular Smarthome with about 3-10 IoT devices could generate anywhere between 50k KB to 2-3 GB of network traffic a day. Homes with 20+ IoT devices and a couple of typical Non IoT devices could easily go through 100+ GB of network packets a week. Persisting this kind of data over long periods is impractical, unless a dedicated hard-drive or cloud service is made available. The decision to store PCAP files was thus disregarded. However, as the intent was to allow users to access historic data (to gauge how devices are used and activities change over time), device state predictions/observations and their timestamps, along with identifiers like MAC addresses and device names needed to be stored and easily retrieved. A simple SQLite database was set up for this purpose. While device state changes along with device identifiers were frequently persisted to the database (in 30 second-intervals while the app was running), the difference in magnitude from complete packet captures which could span 40+rows in 30 seconds (15-30 KB), to these predictions which typically only spanned 2-3 rows in the database (or 1-3KB) in 30 second intervals, suggested that this process was far more practical.

The task then of capturing live network traffic from a smart home, processing this data in real-time, and streaming this output to ML models for real-time prediction was approached in two ways, (1) via a separate python script that used Apache Kafka for network streaming, and, (2) via an integrated script that handled data collection and processing within the Flask app.
Both approaches are outlined in the next sections.

6.3.3.1 Real-time data injection using Kafka

Apache Kafka (Kaf, 2022) is an open-source distributed event streaming platform that is especially useful for building real-time streaming data pipelines. Using Kafka to decentralise the packet capture process was briefly considered to avoid constraining the client device's location to an area close to the router. At a high level, hosting a Kafka producer instead, on a device close to the router would alleviate this by allowing any client device (i.e. device on which this tool would be installed) the freedom to be located anywhere within the house. The host device on which the Kafka producer is run, in this instance, would handle packet capture and streaming, while the client device would contain a Kafka 'consumer' script running in the background. This 'consumer' would automatically receive specific data from selected 'topics' without having to repeatedly 'poll' the host.

To test this out, a simple Kafka Producer was written based on the following high-level logic:

- Use Scapy/Pcapy/tcpdump libraries to capture network packets in 10 second intervals and store to a (temporary) Pcap file.
- Use CiCFlowMeter library to convert these 10 second captures to statistical flows.
- Create a Kafka producer.

- Send each of these flows to a topic titled 'Flows' and then call the producer.flush() method to ensure all previously sent messages have completed.

A Kafka Consumer was then written to receive (or 'consume') this data stream from the producer, by subscribing to that same topic.

Both the Producer and Consumer scripts developed for this trial are included in Appendix A.0.3.

This technique was ultimately disregarded as it proved unnecessary for a use-case as small as a household. Kafka however still remains an extremely viable option for (event based) data streaming over a larger network such as a commercial or institutional environment. Much like how the MQTT protocol works, it is easy to set up a 'consumer', 'subscribe' to specific 'Topics' and get event driven data in JSON format pushed to the consumer (much like a Webhook). Upon receipt, this data can be de-serialised and used in real-time.

6.3.3.2 Real-time Network sniffing and Activity Logging through Flask

Decentralising the data capture and data consumption process proved unnecessary for this application. Considerations like application portability and usability took priority over the challenge of physical location constraints solved by Kafka. Running a Kafka broker, and asking a user to start a Producer and Consumer script simultaneously would be impractical in a production environment.

In Flask, a separate traffic sniffing function using the Tshark command line functionality is set up using subprocess calls. Live packets from a chosen interface (such as 'eth0' and 'wlan0') are captured in intervals of 30 seconds and stored in temporary PCAP files. These files are used for two purposes after which they are immediately discarded: (a) 'Cicflowmeter', a tool developed for statistical flow analysis, is used to convert the data in these packets to bi-directional flows. These flows are then written to another temporary CSV file for immediate (online) ML prediction. These predictions along with their timestamps are continuously persisted to the SQLite database. The flows (CSV) file is stored for 3 days at a stretch before discarding after which another file is created — these 3-day old files can be used to retrain the device recogniton and device state identification models at a later stage. (b)Tshark is also separately used to parse these packet captures and extract metadata from 91 headers into a CSV file. This file is then used by a binary classifier to distinguish IoT from Non-IoT devices in the network, after which it is discarded.

The SQLite database is continuously updated with 5 fields each time a source MAC address emits a packet. This includes the device's MAC address, device name, device state, device location and timestamp. The front end component allows Users to input values for any 'Unknowns' — values such as Location and some Device Names are initially entered this way. Users are also able to alter any false predictions in real-time. This 'supervised' technique is especially useful to retrain multi-class classifiers on new devices connected in the network.

6.3.3.3 User considerations

Such a passive network sniffing setup presents users with certain constraints as well. The sniffing device (host client) will need to be placed close to the router/DHCP server to catch the maximum number of packets. There will inevitably be some loss of packets — the exact amount will vary depending on the location and router model, and is thus out of scope of this paper. The sniffing device will probably be assigned one channel, and will only be able to sniff on that channel. Care will need to be taken to ensure that the devices-of-interest are also connected to that same channel to allow complete packet capture.

6.4 Summary

This chapter employs 'MLOps' practises to deploy and test device and device-state recognition models in a 'production' environment. Allowing models to learn incrementally from new data in real-time ensures model maintainability over time. For this, this section looks at how flow-based and packet-based data is persisted for a short enough amount of time to facilitate potential retraining when new devices are identified in the network.

This chapter also exposes the challenges of such a development methodologies. While continuous data collection was previously explored as a separate stand-alone service with a dedicated API endpoint (in micro-service architecture style), encapsulating this functionality within a single application meant easier portability and usability and was thus chosen. However challenges with this architectural style persist; frequent page reloads causes additional overhead and slows down the application significantly.

CHAPTER 7

Performance and Analysis

To choose the most appropriate classification technique for this multi-stage problem, this thesis examined a number of machine learning classification techniques and compared their validated accuracy. Across all three stages, the Scikit-learn python library was used to implement, and train and test models. 10 iterations and 10-fold cross validation were used to generalize the validation results. Hyper-parameter tuning using Grid Search was then used to improve the performance of the highest performing models. To check model performance, a confusion matrix and a classification report using the Sklearn package was generated for each classifier. Metrics such as Accuracy, Precision, F1 score and Recall for all classifiers were compared. Additionally, parameters like learning speed and resistance to over-fitting were considered while choosing classifiers — K-Nearest Neighbours and Support Vector Machines were therefore deliberately not tested for any of the stages as each classifier took beyond 60 minutes to train on relatively small (>500 sample) datasets. Feature importance from all the top performing models was also taken into consideration to ensure that disproportionate weightage was not given to any single feature during training. To reduce noise and improve performance further, models were retrained after subsequent feature selection.

The challenge with IoT datasets was that some devices generated more activity than others during the same time duration (for e.g. Alexa Echo Studio was found to be more active even during the idle state as compared to the Withings Weighing scale and BP machine). This led to imbalanced datasets. These datasets were not separately under or over-sampled however, as the intention was that the chosen model would handle speedy retraining based on real-time data. Therefore more emphasis was placed on choosing models that are known to perform well with imbalanced datasets in real-time settings — such as boosting, tree and ensemble models. To verify performance, a classification report was generated for each model to check how many instances of each class were correctly identified. Additionally, the F1 score, which is the harmonic mean of precision and recall, was used instead of the 'accuracy' metric, to check the classification success rate of rarer samples.

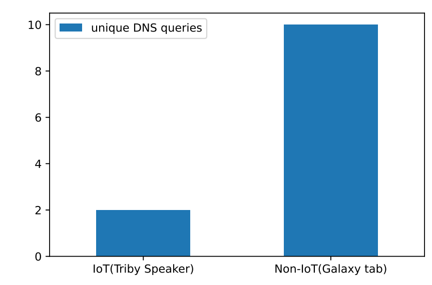
On the development front, instead of building a single multi-stage classifier, separate classifiers for each stage (i.e. detecting IoT versus Non-IoT, detecting IoT device type, detecting activity states (i.e. on/off)) were developed in order to avoid the problem of prediction accuracy reduction in the final stage — i.e. the probability of accurate final stage prediction = probability of accurate prediction of Stage1 x Stage2 x Stage3), which was a challenge highlighted in (Acar *et al.*, 2020). Uncoupling these classifiers also provides scope for individual monitoring and maintenance.

Comparison tables for each of the stages along with the final chosen algorithms are highlighted below:

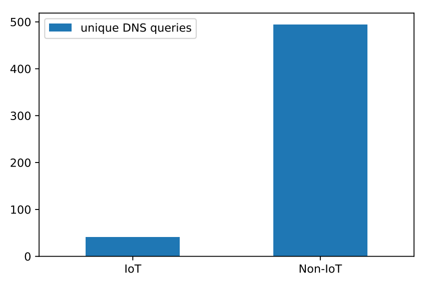
7.0.1 Detecting IoT versus Not IoT

The methodology to differentiate IoT from Non-IoT devices in the network using ML involved the standard pre-processing steps: importing all the necessary libraries in python, importing network trace data from the Smart Lab at CU and the datasets published from UNSW, preprocessing these pcap files to extract metadata from specific headers, separating the x and y variables and splitting the data into train and test data using the Scikit-Learn library in python, fitting a range of classifiers on the train data and making predictions on the test data. [See Jupyter Notebook 1]

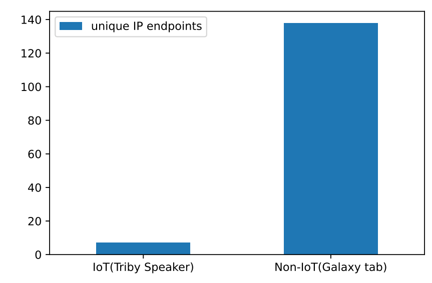
The feasibility of differentiating IoT from Non-IoT devices, rapidly (< 1 minute latency), using traffic features consisting entirely of packet-based TCP/TLS metadata, was verified through analysis conducted on IoT and Non-IoT traffic traces collected from 23 IoT and 9 Non-IoT devices, both from the Smart Lab at Cardiff University and the public repository at UNSW (Sivanathan *et al.*, 2018). This analysis was conducted to test against certain baseline ground truths listed in (Sivanathan *et al.*, 2017), namely, "*(a) An IoT device communicates with less than 10 servers on average per day, (b) IoT devices predominantly tend to use a few specific application-layer protocols (found by destination port number) (c) IoT devices initiate DNS queries for only a limited number of domains (mostly domain name of their vendors or service providers) and repeat the queries in a consistent manner* ". The analysis also compared trends in features tested in (Bremler-Barr *et al.*, 2020), namely, maximum window TCP size, number of unique DNS queries, number of unique interacted endpoints of remote IPs, and number of unique outgoing ports.



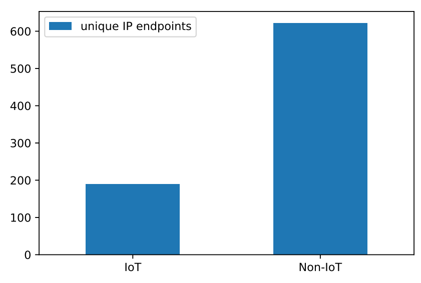
(a) *No.of unique DNS queries over 24 hours between an active IoT and Non IoT device*



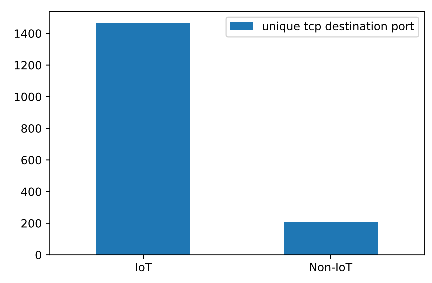
(b) *No.of unique DNS queries over 24 hours, using 23 IoT and 9 Non-IoT devices*



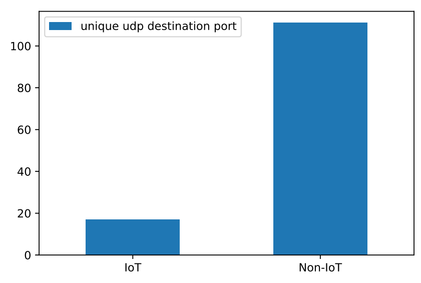
(c) *No.of unique IP endpoints accessed over 24 hours between an active IoT and Non IoT device*



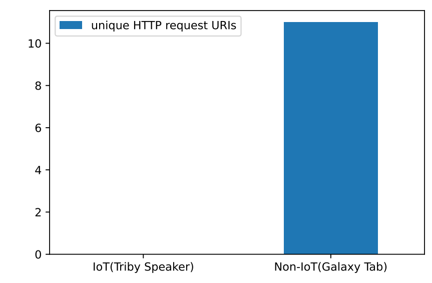
(d) *No.of unique IP endpoints accessed over 24 hours*



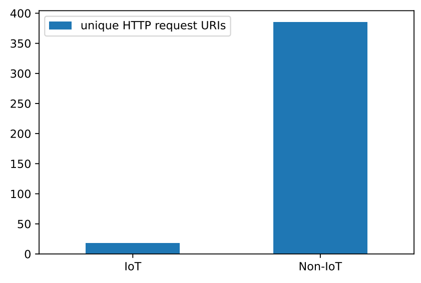
(e) *No.of unique TCP destination ports accessed over 24 hours*



(f) *No.of unique UDP destination ports accessed over 24 hours*



(g) *No.of unique HTTP request URIs accessed over 24 hours between an active IoT and Non IoT device*



(h) *No.of unique HTTP request URIs accessed over 24 hours*

Figure 7.1: *Classification of IoT and Non-IoT traffic patterns*

In general, the findings suggest that IoT devices connected to limited endpoints, (and thus have fewer unique DNS requests, remote IPs and ports); These results were consistent with those outlined in (Sivanathan *et al.*, 2017) and (Bremler-Barr *et al.*, 2020). A range of binary classifiers were then trained using supervised learning, on traffic metadata from 91 pcap headers/features [Appendix A 10].

The model chosen was Light GBM with a recall rate and F1 score of 99% [Table 7.1].

Algorithm	Accuracy	Precision	Recall	F1 score
Light GBM	99.2%	99.1%	99.9%	99.3%
Gradient Boost	98.9%	98.7%	99.2%	99.3%
XGBoost	99.2%	99.1%	99.9%	99.5%
AdaBoost	98.9%	98.8%	99.8%	99.3%
Random Forest	99.2%	99.3%	99.6%	99.5%
Decision Tree	98.9%	99.3%	99.3%	99.3%
Cat Boost	98.9%	99.3%	99.3%	99.3%
Logistic Regression	97.4%	97.6%	99.1%	98.4%

Table 7.1: *Performance results from 8 binary classifiers for the IoT vs. Non-IoT detection stage*

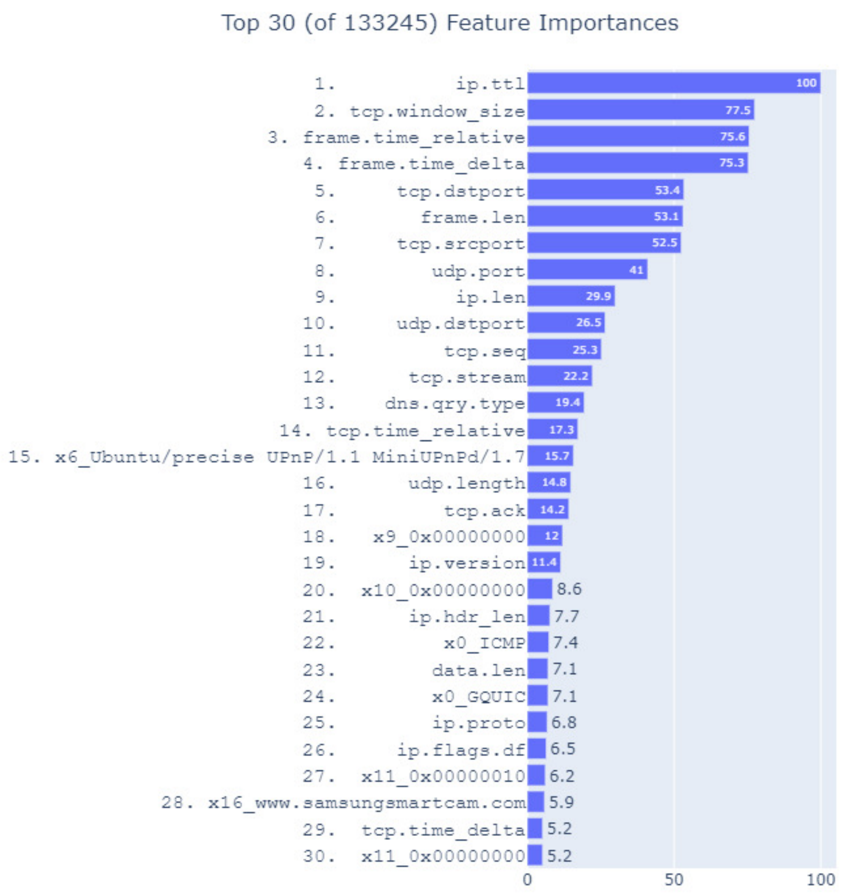


Figure 7.2: *Feature Importance - Light GBM for IoTvNoT on traffic metadata*

7.0.2 Detecting Device type — Device Fingerprinting

A range of Multi-Class Classification models were initially trained on *30-second, bidirectional network traffic flows* collected from 8 commonly used house-hold devices, from the Smart Home Lab at Cardiff University — the Alexa Echo Studio, TPLink Plug, Lumiman Bulb, Roborock Vacuum Cleaner, LG smart TV, Withings BP monitor, Weekett Kettle, and Smarter Coffee Machine.

These statistical flows consisted of 78 features which included, the mean, standard deviation, minimum and maximum Flow Inter-Arrival Time (or IAT); statistics for forward and backward packets (e.g. packet length, header length, bytes, flag count); statistics for idle and active flows and sub-flows such as the minimum and maximum time a flow was active before becoming idle, etc. This also included a few packet-based features such as source and destination ports, IP and MAC addresses and Protocols. These statistical flows were generated from the Flask app, using command line applications like tshark for packet capture and CiCFlowmeter for pcap to flow conversion.

The models tested for this stage included the OneVsRest classifier with different base estimators, XGBoost, AdaBoost, Gradient Boost, and Light Gradient Boosting Machine algorithms, tree-based algorithms like Decion Tree and Random Forest, and stacking ensemble algorithms with different base estimators. The Light GBM and Extreme Gradient boost classifiers were seen to outperform the rest. The initial LGBM training reached an F1 score of 96.0% (without parameter tuning) and an accuracy of 97.04% after hyperparameter tuning. The scores of each input feature was tabulated [Figure 7.3]; features such as source port, backward (initial) window bytes, destination port, and flow inter-arrival-time statistics such as minimum flow Inter-Arrival-Time, minimum backward Inter-Arrival-Time, and flow duration, were given the highest scores — similar to the scores from the Random Forest and XGBoost classifiers. This classifier was later *partially* re-trained online using traces from the IoT Sentinel dataset. The results of each trial are outlined in Table 7.2.

Algorithm	Accuracy	Precision	Recall	F1 score	Hyperparameter tuning (accuracy)
LightGBM	96.71%	96.0%	96.0%	96.0%	97.04% (f1-score) using 'max_depth': -10, 'min_child_weight': 1e-05, 'n_estimators': 1000, 'num_leaves': 40
XGBoost	96.15%	96.3%	96.4%	96.3%	-
OneVsRest (XG-Boost)	96.15%	95.8%	96.2%	95.9%	96.75% using 'max_depth': 4, 'min_child_weight': 1, 'n_estimators': 1000
Gradient Boost	94.87%	95.0%	95.0%	95.0%	-
Random Forest	94.54%	94.7%	94.7%	94.6%	-
OneVsRest (Random Forest)	93.92%	94.6%	94.6%	94.4%	95.34 % using 'max_features': 60, 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_estimators': 100
Logistic Regression	80.81%	79.0%	80.0%	78.0%	-
OneVsRest (Logistic Regression)	79.80%	78.30%	79.80%	77.0%	-
AdaBoost	43.35%	52.3%	42.5%	40.7%	-

Table 7.2: *Performance results from 9 multi-class classifiers for the Device fingerprinting stage*

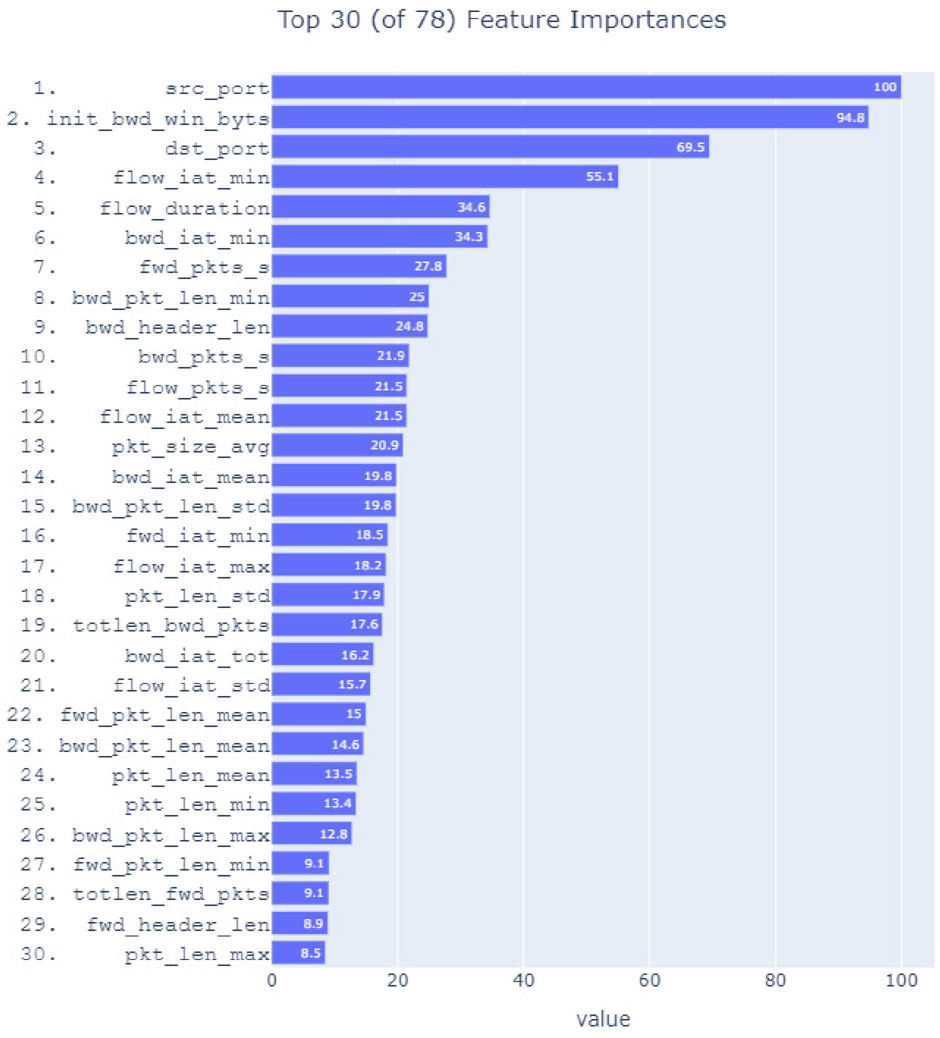


Figure 7.3: *Feature Importance - Light GBM for Device Fingerprinting on network flows*

7.0.3 Detecting Activity States - On vs Off

Experiments were conducted to test basic activity recognition between the active or 'On' state and the inactive/idle or 'Off' state. The 'Off state' terminology is here used interchangeably with the 'inactive state' because a fully unplugged or discharged device would not exhibit any network activity at all, leading to the device being virtually 'invisible' over the network, i.e. this device would be 'Offline' in such an instance, and not 'Off'.

The challenge at this stage was to identify device states suitably quickly whilst also reducing the chances of mis-identification. For this, models were first trained using 10-second statistical flows. The resulting F1-score from standard top performing algorithms such as tree-based, boosting, and ensemble classifiers was found to average to 60%. As this score was unacceptable in a home environment, 30-second statistical flows were next considered. This was found to be better performing, with the F1 score averaging to 84.4%, with the highest score being 90.1% recorded by the OneVsRest Classifier (with XGBoost Base Estimator). 60-second flows were also considered, however these were not found to be significant improvements over the 30 second flows to warrant the increase in time.

The results additionally suggested that devices still received bursts of data during the inactive state. If bidirectional flows were used to predict device states, this meant that a largely inactive device that received some data was still predicted as an active device. This reduced prediction accuracy. To increase performance, these datasets were pre-processed again to ensure that only flows emanating from the device in question were taken into consideration. Findings then suggested that while devices in the inactive state typically sent between 1-5 packets to the router every 30 seconds (typically the Heartbeat or Time To Live (TTL) message), some devices that were active or 'On' for a while also displayed similar patterns; the LG TV for instance only demonstrated peaks in traffic when channels were switched, or the volume was changed.

Algorithm	Accuracy	Precision	Recall	F1 score
OneVsRest (XGBoost)	90.4%	90.4%	90.4%	90.4%
XGBoost	90.4%	90.4%	90.4%	90.4%
Gradient Boost	88.62%	87.0%	87.0%	87.0%
AdaBoost	87.79%	87.8%	87.7%	87.7%
LightGBM	87.04%	88.0%	88.0%	88.0%
OneVsRest (Random forest)	88.0%	88.0%	88.0%	88.0%
Random forest	86.49%	86.7%	86.6%	86.6%
OneVsRest (Logistic Regression)	71.91%	71.9%	71.9%	71.9%
Logistic Regression	70.5%	71%	71%	71%

Table 7.3: *Performance results from 9 binary classifiers for the Device state recognition stage*

Note: This feature-set was also used to determine if specific activity states like 'Kettle heating', 'Alexa playing music', 'Blink Camera Live View' could be assessed as well. 13 such 'activities' or states were tested in all. The same classifiers were used to test this dataset [see: Jupyter Notebook 3]. The Gradient Boost classifier was found to narrowly outperform the LightGBM classifier with an F1 score of 80.1%. More work can be done to improve the dataset however, as these models are not likely to perform well in a live setting.

7.1 Summary

The findings suggest that IoT devices and their current states can be successfully identified using both packet-based and flow-based statistics, in real-time. The highest performing classifiers at each of the three stages were found to be as follows: (a) To detect IoT from Non-IoT device, the Light Gradient Boosting Machine outperformed both the Extreme Gradient Boost and the Random Forest Classifiers, with an F1 score of 99.3% and accuracy of 99.2%, (b) To detect device name (i.e., fingerprint devices) the Light Gradient Boosting Machine again outperformed both the Extreme Gradient Boost and the One Versus Rest Classifiers, with an F1 score of 97.0% and accuracy of 96.7%, (c) To detect device state, the One Versus Rest (with XGBoost base classifier) outperformed

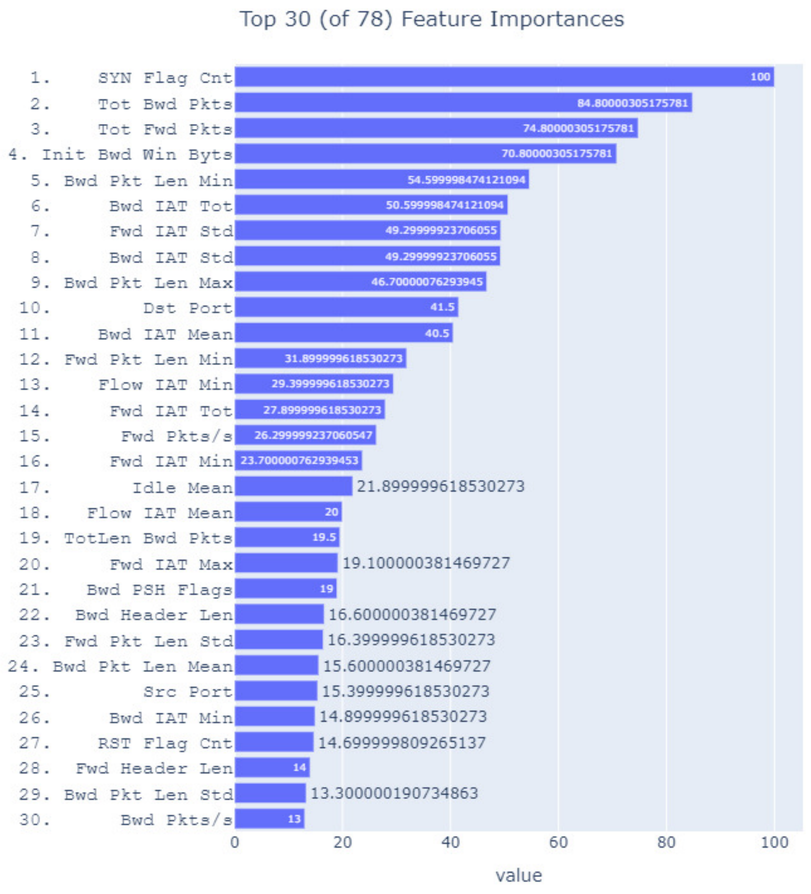


Figure 7.4: *Feature Importance - Extreme Gradient Boost for Device State Identification using network flows*

other boosting and tree based classifiers with an F1 score of 90.1% and accuracy of 88.1%.
In comparison with the state-of-the-art, the performance scores for Stage 1 (IoT vs NoT) was found to be at par with current works using similar packet metadata-based classification.
The scores for Stage 2 (Device fingerprinting), on the other hand, was found to be 2.7% lower than the high F1 score of 99.9% by the Random Forest classifier recorded by (Shahid *et al.*, 2018). This can be attributed to the fact that in (Shahid *et al.*, 2018), a fewer number of IoT devices (viz., 4) were used, data was captured over a longer duration (7 days), and the dataset was largely balanced. In direct comparison, the datasets in this paper were captured over an average time period of 2 hours for 8 devices, and these datasets were largely imbalanced (with a ratio of 1:50 for some devices) - this was done to mimic real-world conditions where these models will need to be retrained, quickly, on live, noisy, data. Light Gradient Boosting Machine, therefore, is seen here to outperform the Random Forest Classifier, as it is built to handle heavily imbalanced datasets *out-of-the-box* without the need for other techniques like under or over-sampling data.
For Stage 3: Device state identification, the f1 score was also found to be 3.7% lower than that recorded by (Acar *et al.*, 2020). While the dataset was equally imbalanced, the devices themselves were a mix of sensors, gateways and bridges from across the zigbee, bluetooth and wifi protocols, and therefore was a bit dissimilar to the wifi-based dataset used in this paper. More notably

however, a different set of statistical features (generated using *tfresh*), were used to train the classifier; 795 initial features were used initially, which were later cut down to 197 binary features using feature selection. No mention was made on final training time. In contrast, this paper uses 84 (out-of-the-box) flow-based features generated from CiCFlowMeter to train the classifiers, where the top performing classifier takes an average of 1.5 seconds to train. Further experiments will therefore need to be run on this paper's dataset using the features selected in (Acar *et al.*, 2020) to gain more comparable results. Additional experiments can also be run with a more custom set of features through CiCFlowMeter to improve the achieved results.

Note: Compound user actions such as 'working', 'sleeping', and 'cooking' were separately ascertained using rule-based algorithms through the Flask application. As an example, activities like 'working' were 'guesstimated' through the locations and timestamps of devices, for e.g., if the location of the devices were labelled as 'Study', and the states of 2 or more devices located in the 'Study' were 'ON' (like 'laptop plug', and 'bulb' for instance), and all within a set timeframe, the activity was automatically labelled as 'working'. This may be presumptuous however; further work can be done to improve on this.
Note 2: The Jupyter Notebooks used to (initially) train these models are included along with this submission.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

This thesis explores the development of a smart home-based activity recognition tool that uses a simpler data collection methodology as compared to the current state-of-the-art.

Through a comprehensive literature review and findings from subsequent experimentation, this paper demonstrates that COTS devices and device activities can be automatically recognised through patterns in their network activity. Subsequently, it finds that more complex human activities can be constructed using this data, almost, if not as successfully, as traditional sensor-based methods. It also posits that this traffic-based methodology, through its open-source nature, offers the benefit of being almost fully hardware and software agnostic. It is therefore offered as a time and cost-effective alternative to traditional approaches.

This Thesis therefore uses this methodology to develop a layman-friendly activity recognition tool as a counterpart to the diagnostics component of mature open-source hubs like Home Assistant and OpenHAB. The proposed tool in this Thesis navigates the challenges of such hubs through its easy set-up and simple UI —both of which do not require prerequisite technical skills to use. The proposal eliminates any monetary commitment, by making use of open-source network sniffing tools to identify devices and their current states. This tool goes a step beyond tools like Home Assistant and OpenHAB as well, by ensuring that devices are identified through robust fingerprinting methods, instead of currently used common identifiers like MAC and IP addresses, so that they are hard/impossible to spoof. To the best of the author's knowledge, such a diagnostics tool with custom labelling capabilities, does not exist in the open-source market.

To implement this tool, this Thesis uses a multi-stage device-state identification method using encrypted network data. It extends current literature around passive activity detection techniques and brings it into a production-ready environment. It identifies the database requirements, outlines tools to capture and process network traffic, recommends top performing binary and multi-class classifiers, and offers ways to retrain models using user input. It also highlights the challenges and limitations (Section 8.2) of using this software development technique.

For evaluation, this Thesis utilises metrics such as precision, recall and F-measure, which provides an insight into the quality of the classification. The performance of these probabilistic models is evaluated and these results provide a baseline for comparison with other recognition methods in live settings. The datasets needed for repeating these experiments are made available as well (Dat, 2022).

8.2 Limitations

The architecture of this tool is designed to be an all-in-one portable solution for general activity monitoring. This faces its own set of challenges: (a) *Lag*: Continuous network sniffing, frequent database updation, and regular, automatic page refreshes, causes each Flask webpage to load in an average of 10 seconds — which is much slower than the average 100ms recorded for a Flask application of this size. This could be made significantly faster if computationally heavy tasks like network capture and database updation is separated from the main flask app. From an MLOps perspective, implementing a microservice style of architecture could address this problem. Separate deployment would additionally address the 'single point of failure' problem which is another limitation of the current application. (b) *Physical location of deployment*: This application captures local traffic packets passively. To minimise packet loss the application will need to be hosted on a device close to the router,or with the least number of physical barriers between them. (c) *Non-protocol agnostic*: The current set up collects network data from devices that use Ethernet and Wifi. Protocols like ZigBee, Z-Wave, Bluetooth are not addressed here as these require additional hardware (like ZigBee or Bluetooth sniffers), that the average User may not possess. This, however, means that the current application excludes devices from the SmartThings and Aqara ecosystems as these battery-based devices typically run on such low-power protocols. (d) *Manual install for certain tools*: This application uses tools like Wireshark, TShark and CiCFlowMeter which users will need to install separately. This is because the 'Promiscuous' mode of monitoring will need to be manually enabled, and this could require administrative privileges. The process of mitigating some of these challenges are addressed in the next section.

8.3 Future Work

To enhance scalability and performance in a production environment, the ML pipeline could be orchestrated via Apache Airflow and hosted online using Docker. An API endpoint would then push data to the Flask application using Webhooks. A decentralised approach to packet capture and real-time transmittance using Apache Kafka could also be explored in environments where physically connecting or locating the client machine close to the main router may not be possible. The management and results of ML model retraining could also be automated and tracked online using tools like MLFlow.

To make this application truly protocol or device-'agnostic', dedicated hardware like Bluetooth and ZigBee sniffers can be used. After making a change to the desired interface (e.g. from 'Wi-Fi' to 'Bluetooth') on the tshark packet-capture command [Appendix A.O.2], the subsequent steps of packet capture and packet-to-flow conversion will be the same as the current set-up. Though not tested for this paper, it is assumed that the ML component will work much the same way as well — works such as (Acar *et al.*, 2020) have explored similar flow-based device and device-state recognition methods using Bluetooth and ZigBee traffic, and have achieved beyond 90% accuracy scores.

On the ML front, an attempt could be made to include even 'noisier' data in the initial learning stage. The models currently classify device names with an average 75% accuracy rate when deployed in a live, 'noisy' environment. This is much lower than the initial scores achieved during training - implying that further improvements in selecting a diverse enough dataset should be made. While, false scores are currently mitigated through a process of 'threshold-acceptance checks' (where devices with prediction probability scores lower than 90% are classified as 'Unknown', thus prompting users to re-label them), much of this online re-training process can be reduced for already known devices if initial datasets include randomised traffic. Additionally, this paper currently uses 84 flow-based features generated from CiCFlowMeter, a network traffic flow generator to train two classifiers. The results achieved suggest that improvements can be made on customising the feature-set further, while managing the overall training time. Moreover, the number of samples used to train the models could be increased to improve performance score.

Additional work can also be done to improve Anomaly Detection techniques on such an application. The current set up is built on an ad-hoc rule-based approach. It is developed as an analytics tool that detects anomalous physical behaviour such as surges in device activities during unusual time periods and even exceptionally long activity periods for certain devices. This analysis could be made more sophisticated — for instance, time-based physical anomalies, such as events that could lead up to a flood or fire, could be detected. Improvements within the cyber realm could also be made — devices that are malfunctioning or compromised could be detected through unusually large packet transfers, for example.

CHAPTER 9

REFLECTION

This Thesis explored two distinct components of the IoT network analysis project equally — Machine Learning and Software Engineering. The project's end-goal was not trivial; in essence, the objective was to develop an MLOps project that facilitated real-time detection of physical events using encrypted traffic. This involved a considerable learning curve — specific upskilling was needed in areas such as (a) IoT networks and protocols, (b) Supervised, semi-supervised and online learning ML techniques, (c) ML pipeline orchestration and management techniques, and (d) Data collection/streaming services. It was imperative therefore to narrow down the scope of the project and then, to create a realistic project plan.

To refine the project scope, a broad but quick market study was necessary. The aim was to understand if activity recognition using network data instead of more commonly used physical device data was feasible, and more importantly, informative enough, for regular smart home users. To extract insights from multiple papers in a short span of time, a simple Excel-based exercise proved useful. Here, any papers reviewed were tabulated and key bullet points regarding the paper's *Purpose*, *Experiments*, *Datasets*, and *Outcomes* were roughly outlined. The 'Abstract', 'Conclusion' and 'Future Work' sections of these papers provided enough content for this exercise.

This type of a review structure exposed any outstanding themes, and gaps, quickly. It found that while there was already substantial existing work in traffic-based HAR from the IoT privacy and vulnerability research sub-domains, most of these works were largely theoretical. Moreover, as their primary motivation was to gauge and limit exposure, these papers approached the study from the point of view of an external (malicious) actor. Having found a sizeable gap to fill here, the next step was to quickly assess if any IoT diagnostics or HAR-based tools existed in the open-source market. A quick study revealed that while tools like NMAP, Home Assistant and OpenHAB (for IoT diagnostics) existed, the later two did not address security (against spoofing) nor usability for wider, non-technical audiences. Additionally, none of these tools provided an overview of the concurrent activities happening in the smart home, and therefore did not offer any HAR insights.

These set of findings, through this preliminary market study, helped set the benchmark and generate a more refined project outline and purpose for this dissertation.

Next, to develop a project plan, a series of iterative process diagrams proved useful. This led to clear requirements definition — which generated a list of required and 'nice-to-have' features. This also helped address my existing skills gap, and helped determine which components to prioritise, and which to 'value engineer' as 'out-of-scope' for the given timescale. The outcome of this exercise proved useful for writing the 'Limitations' and 'Future Work' Sections of this report.

9.1 Project Management Style

My initial project plan followed the Waterfall model. The intention was to allocate one week for overall research (i.e., for undertaking a preliminary market study, assessing IoT lab resources, and curating necessary learning material), two weeks for an extended literature review, one week for data collection, four weeks for ML experimentation and software development, and four weeks for writing the paper.

This model did not work well past the third week. The allocated timescale for each task did not account for any additional learning nor did it cater to possible execution delays or failures during the experimentation and software development phases. Moreover, the rigidity of this structure did not bring potential roadblocks in the software development phase to light early in the process.

For example, two 'nice-to-have' features that were to be included were a webserver for ML model management, and comparison graphs between the predicted device states (using network data) and Home Assistant states (using device data). For the first, implementing a fully automated pipeline using Apache AirFlow and MLFlow took more time than allocated. Simpler alternatives should have instead been tested (quickly) during the beginning stages. For the second feature, it was found that Home Assistant required a dedicated Linux OS, and would not run on a virtual machine or Windows Subsystem for Linux unlike what the documentation suggested. This meant that a majority of potential non-technical users would not have as easy access to this software as originally understood — which meant that the project scope needed to be re-revised. This should have also been tested during the early stages of the project.

The total number of 'unknowns' in the project, therefore, exceeded the total number of 'knowns' — and this did not suit the Waterfall model. To address each of the unknowns, it became necessary to shift to a more 'Agile' style of working.
Using the 'Sprint' workflow helped reduce abortive tasks after the Literature Review phase. *3-day*

sprints were used to quickly trial secondary/nice-to-have features (like real-time dynamic displays), to allow just enough time to try and potentially fail at a task without it having an adverse impact on the overall timeline. Planning work in this iterative manner helped generate key findings quicker, which helped troubleshoot issues in a timely manner. To meet the deadline, other simple workflow techniques were used as well. Where open-source tools/libraries/templates were available for supportive functions, these were utilised. Previous skills/knowledge was reused as well — an example of this was using Flask itself to develop the project. Since this framework was not new to me, it was easier to focus on testing/developing the main objectives of the project without worrying about the implementation itself.

Best practises in MLOps were used during the ML development phase as well. These were (a) taking a modular approach (b) having pre-trained models ready to show as proof of concept (c) developing generic algorithms to show some success, and then training them further for their specific tasks, and (d) bridging gaps in training data with publicly available data sources. These steps were useful in generating results quickly. The 'simplest' models were trained first, using small external datasets, to demonstrate proof-of-concept. During this stage, an automated pre-processing pipeline was created that could be re-used for other classification tasks. This meant that extra time was not spent on cleaning and pre-processing data for each subsequent stage of the project. External datasets were also identified as a back-up for instances where certain devices were unavailable in the Smart Home Lab.

9.2 Shortcomings of work conducted

The proposal, at its earliest stage, started with the question - 'Can I check if the hotel I am staying at has any hidden devices'? This exposed others such as 'Are there ways to assess if any connected devices are 'active' in my network?', and 'Is there an easy, human-readable way to 'see' *all* the devices connected to this network?'. These questions brought an exposure to current research works that successfully detected IoT and Non IoT devices in the 'wild' such as (Bremler-Barr *et al.*, 2020), and others which successfully fingerprinted devices such as (Bruhadeshwar *et al.*, 2018). Using Home Assistant (a home automation hub), for an unrelated project at that time, revealed that this desired type of immediate 'network surveillance' functionality was not available to most smart home users, unless they invested time in manually connecting these devices individually to these hubs.

A theoretical paper that solved some of these questions was (Acar *et al.*, 2020) however this work did not consider this research from the view of a smart home user as its primary objective was just to gauge the extent of device activity exposure from a smart home. Factors such as scalability, speed of recognition/detection, memory constraints, potential user-input, potential online model re-training, database management, all needed to be considered.

While this dissertation attempts to use this as a start at productionalising such a tool, there is still scope for further improvement:

(a) *The methodology to test the performance of the tool in a live setting requires better definition;*
For the purpose of this dissertation, experiments were conducted manually - where devices are connected, activated, de-activated and disconnected, in an ad-hoc manner. A few readings (about 3-5) were recorded for each result (e.g. 'observed ip address', 'predicted name', 'predicted state'), and readings were then assessed against physical evidence (or ground truths) 'by eye'. To provide more conclusive results of these experiments, a higher number of such experiments —at least in the tens— could be conducted and, results tabulated. The current set of findings were too variable to draw any definite conclusions about its performance 'in the wild'. In addition to this, the application could benefit from being tested in more than one 'live' environment.

(b) *The initial data gathering process could benefit from automation;*
The locally gathered datasets are by no means 'rich'. These were gathered through ad-hoc experiments, where devices were manually activated and de-activated after either the full length of the device's activity (for example, for the time it takes for a 'Roborock' vacuum cleaner to clean one room, or for a 'Smarter' coffee machine to finish one brew), or for about 10-15 minutes at a stretch for devices like Smart bulbs and plugs. The number of such experiments conducted were not extensive enough to train robust classifiers. Moreover, as this was a manual effort, the datasets were in danger of being mislabelled. These models were therefore presented as 'proof of concepts', and are by no means a finished product.

The PingPong dataset (Trimananda *et al.*, 2019), in contrast, was significantly 'richer' and could be used as a benchmark for future research works. It used a systematic, *automated*
technique to capture traffic from 22 devices, where automations through smartphone apps orchestrated the experiments. A script was run to generate 'touch-points' on the smartphone apps themselves to activate/de-active devices, so very little manual intervention was necessary. Each experiment was then run 'n' times for 'x' seconds -where n was in the range of 50-100 experiments, and 'x' depended on the device. For our purpose, Home Assistant could easily supplant the need for a smartphone script so as to make this process *even simpler*, however the key takeaway from this is that the final dataset generated by the 'PingPong' experiments was robust and included perhaps a complete set of possible traces from each device.

(c) *Further development is required in the proposed 'online learning' component of this research work;*
The current lack of skills within the set timeframe of this project meant that the 'online learning' component of this research work could not be adequately explored nor executed in the final tool. The intention was to successfully retrain devices online, so as to recognise guest devices that may

enter the network at a later date. The methodology that was intended to do this has already been put in place in the code itself: 3 days worth of network traces are currently captured by the tool, converted to flows in a CSV format, and then stored, after which this file gets deleted (to keep memory costs at a minimum) and is replaced by another similar file. The intention is that this flow-based featureset can be used as input to partially retrain the 'device fingerpinting' classifier to detect, for example a new, never before seen, 'blood pressure monitor' that a guest may bring in.

The way this would be done is as follows:
The tool is already capable of recognizing new devices in the network, categorising these as 'Unknowns', and then allowing users to input or 'label' these by IP address. This label is then written to a database table that is then displayed on the front end. However, the main intent of this ('device name') label is to be used as a 'prediction output label'. In simple terms, all samples in the (3-day old) flow-based CSV file, that corresponds to the new IP address in question, would be automatically labelled with this inputted 'device name' label through a script. This 'labelled' CSV file will then be fed into the same device fingerprinting classifier again, to partially retrain it using Scikit Learn's 'partial fit' method. This updated model will then start to serve correct predictions each time a new 'blood pressure monitor' (for example) enters the network.

This implementation is definitely worth exploring further, as the knowledge gained through developing such an online training mechanism will prove invaluable in industry projects and future research works.

9.3 Key Learnings and Takeaways

On the research/project management front, a key learning was to write the paper in tandem with the experimentation phase, and then restructure as required after implementation. The previous waterfall method made it difficult to track the early stages of a project; moreover, the time to allocate, to write the paper was not easy to ascertain. Another learning was that shorter (soft) deadlines proved effective for a range of tasks. This was in keeping with the 'Agile' spirit, where timely rectifications could be made if these tasks did not go to plan. A third was to approach tasks/features/models iteratively; smaller/simpler deployments could be used as proof-of-concepts before more complex tasks were attempted.

On a technical front, new skills were acquired in machine learning techniques. Simple yet effective data pre-processing techniques were studied and an insight was gained into ML pipelines, which helped automate these tasks. Different performance metrics were utilised: K-fold cross validation scores, confusion matrices, and other standard metrics such as f1 scores and recall rates were assessed to finalise the models at each classification stage. Techniques to improve scores, such as feature extraction using feature importances, Hyperparameter tuning using Grid Search, and stacked ensembles were explored as well. Exposure was also gained in machine learning management and orchestration tools like Apache AirFlow. Within the Apache framework, Kafka was explored for streaming data —with experiments in writing simple Consumer and Producer scripts to stream captured flows in real-time proving successful. Whilst neither of the Apache products were used in the final proposal, the exposure gained would prove most lucrative for future projects in industry.

Lastly, a deeper understanding of IoT networks and protocols was gained. Active and passive techniques in device fingerprinting were studied, and an understanding of the OSI model was developed. Simple experiments in Pandas on the captured datasets were conducted to reveal patterns in IoT and non-IoT traffic; these patterns were then exploited to classify devices and device states with minimal training data almost as successfully as the state-of-the-art.

Bibliography

2021. *The unsw dataset*, [Online]. Available at: https://research.unsw.edu.au/projects/unsw-nb15-dataset. Accessed: 15 August 2022.

2022. *Apache kafka - documentation*, [Online]. Available at: https://kafka.apache.org/. Accessed: 1 September 2022.

2022. *Flask v:2.2.x - documentation*, [Online]. Available at: https://flask.palletsprojects.com/en/2.2.x/. Accessed: 09 September 2022.

2022. *Highcharts - interactive javascript charts library*, [Online]. Available at: https://www.highcharts.com/. Accessed: 18 September 2022.

2022. *Home assistant*, [Online]. Available at: https://www.home-assistant.io/. Accessed: 06 July 2022.

2022. *Home assistant add-ons*, [Online]. Available at: https://www.home-assistant.io/addons/. Accessed: 09 September 2022.

2022. *Iot dataset collected from smart home lab, cardiff university*, [Online]. Available at: https://cf-my.sharepoint.com/:f:/g/personal/zachariasm_cardiff_ac_uk/EpZvgw_rO15AuO7pbDvOE9QBnZ8wSJSL-OfFEk4x9OF5uQ?e=3BAVwcy. Accessed: 17 October 2022.

2022. *Openhab*, [Online]. Available at: https://www.openhab.org/. Accessed: 10 July 2022.

2022. *Rest api - home assistant*, [Online]. Available at: https://developers.home-assistant.io/docs/api/rest. Accessed: 07 October 2022.

2022. *Scapy - packet crafting for python2 and python3*, [Online]. Available at: https://scapy.net/. Accessed: 08 August 2022.

2022 - 2030. *Smart home market size, share trends analysis report by region and segment forecasts*, [Online]. Available at: https://www.grandviewresearch.com/industry-analysis/smart-homes-industry. Accessed: 04 October 2022.

A. Sivanathan, F. L. A. R. C. W. A. V., H. Habibi Gharakheili and Sivaraman, V. 2022. *Unsw sydney - iot traffic traces*, [Online]. Available at: https://iotanalytics.unsw.edu.au/iottraces. Accessed: 09 September 2022.

Acar, A., Fereidooni, H., Abera, T., Sikder, A. K., Miettinen, M., Aksu, H., Conti, M., Sadeghi, A.-R. and Uluagac, S. 2020. Peek-a-boo: I see your smart home activities, even encrypted! In: *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*. pp. 207–218.

Aksoy, A. and Gunes, M. H. 2019. Automated iot device identification using network traffic. In: *ICC 2019-2019 IEEE International Conference on Communications (ICC)*. IEEE, pp. 1–7.

Apthorpe, N., Huang, D. Y., Reisman, D., Narayanan, A. and Feamster, N. 2018. Keeping the smart home private with smart (er) iot traffic shaping. *arXiv preprint arXiv:1812.00955* .

Apthorpe, N., Reisman, D. and Feamster, N. 2017. A smart home is no castle: Privacy vulnerabilities of encrypted iot traffic. *arXiv preprint arXiv:1705.06805* .

Avizheh, S., Doan, T. T., Liu, X. and Safavi-Naini, R. 2017. A secure event logging system for smart homes. In: *Proceedings of the 2017 Workshop on Internet of Things Security and Privacy*. pp. 37–42.

Bezawada, B., Bachani, M., Peterson, J., Shirazi, H., Ray, I. and Ray, I. 2018. Behavioral fingerprinting of iot devices. In: *Proceedings of the 2018 workshop on attacks and solutions in hardware security*. pp. 41–50.

Biondi, P. 2022. *Scapy - packet crafting for python2 and python3*, [Online]. Available at: `https://scapy.net/`. Accessed: 05 July 2022.

Bremler-Barr, A., Levy, H. and Yakhini, Z. 2020. Iot or not: Identifying iot devices in a short time scale. In: *NOMS 2020-2020 IEEE/IFIP Network Operations and Management Symposium*. IEEE, pp. 1–9.

Bruhadeshwar, B., Bachani, M., Peterson, J., Shirazi, H., Ray, I. and Ray, I. 2018. Iotsense: Behavioral fingerprinting of iot devices. *ArXiv abs/1804.03852* .

Burke, M. 2022. *News article - amazon's alexa may have witnessed alleged florida murder, authorities say*, [Online]. Available at: `https://www.nbcnews.com/news/us-news/amazon-s-alexa-may-have-witnessed-alleged-florida-murder-authorities-n1075621`. Accessed: 25 July 2022.

Cherman, E. A., Tsoumakas, G. and Monard, M.-C. 2016. Active learning algorithms for multi-label data. In: *IFIP International Conference on Artificial Intelligence Applications and Innovations*. Springer, pp. 267–279.

Chowdhury, R. R., Aneja, S., Aneja, N. and Abas, E. 2020. Network traffic analysis based iot device identification. In: *Proceedings of the 2020 the 4th International Conference on Big Data and Internet of Things*. pp. 79–89.

Combs, G. 2022. *tshark manual page*, [Online]. Available at: `https://www.wireshark.org/docs/man-pages/tshark.html`. Accessed: 05 July 2022.

Copos, B., Levitt, K., Bishop, M. and Rowe, J. 2016. Is anybody home? inferring activity from smart home network traffic. In: *2016 IEEE Security and Privacy Workshops (SPW)*. IEEE, pp. 245–251.

Cui, W., Li, B., Zhang, L. and Chen, Z. 2021. Device-free single-user activity recognition using diversified deep ensemble learning. *Applied Soft Computing* 102, p. 107066.

Damodaran, N. and Schäfer, J. 2019. Device free human activity recognition using wifi channel state information. In: *2019 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computing, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*. IEEE, pp. 1069–1074.

Dong, S., Li, Z., Tang, D., Chen, J., Sun, M. and Zhang, K. 2019. Your smart home can't keep a secret: Towards automated fingerprinting of iot traffic with neural networks. *arXiv preprint arXiv:1909.00104* .

Fang, H. and Hu, C. 2014. Recognizing human activity in smart home using deep learning algorithm. In: *Proceedings of the 33rd chinese control conference*. IEEE, pp. 4716–4720.

Hamad, S. A., Zhang, W. E., Sheng, Q. Z. and Nepal, S. 2019. Iot device identification via network-flow based fingerprinting and learning. In: *2019 18th IEEE international conference on trust, security and privacy in computing and communications/13th IEEE international conference on big data science and engineering (TrustCom/BigDataSE)*. IEEE, pp. 103–111.

Hieu, L. 2022. *Cicflowmeter*, [Online]. Available at: `https://www.unb.ca/cic/research/applications.html`. Accessed: 20 July 2022.

Jackman, J. 2022. *Smart home statistics*, [Online]. Available at: `https://www.theecoexperts.co.uk/smart-homes/statistics`. Accessed: 18 October 2022.

Kabir, M. H., Hoque, M. R., Thapa, K. and Yang, S.-H. 2016. Two-layer hidden markov model for human activity recognition in home environments. *International Journal of Distributed Sensor Networks* 12(1), p. 4560365.

Kolcun, R., Popescu, D. A., Safronov, V., Yadav, P., Mandalari, A. M., Mortier, R. and Haddadi, H. 2021. Revisiting iot device identification. *arXiv preprint arXiv:2107.07818* .

Miettinen, M., Marchal, S., Hafeez, I., Asokan, N., Sadeghi, A.-R. and Tarkoma, S. 2017. Iot sentinel: Automated device-type identification for security enforcement in iot. In: *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, pp. 2177–2184.

Miettinen, M., Marchal, S., Hafeez, I., Asokan, N., Sadeghi, A.-R. and Tarkoma, S. 2018. *Iot sentinel dataset: Automated device-type identification for security enforcement in iot*, [Online]. Available at: `https://github.com/andypitcher/IoT_Sentinel/tree/master/captures_IoT_Sentinel/captures_IoT-Sentinel`. Accessed: 05 August 2022.

Ni, Q., Garcia Hernando, A. B. and De la Cruz, I. P. 2015. The elderly's independent living in smart homes: A characterization of activities and sensing infrastructure survey to facilitate services development. *Sensors* 15(5), pp. 11312–11362.

OConnor, T., Mohamed, R., Miettinen, M., Enck, W., Reaves, B. and Sadeghi, A.-R. 2019. Homesnitch: behavior transparency and control for smart home iot devices. In: *Proceedings of the 12th conference on security and privacy in wireless and mobile networks*. pp. 128–138.

Palmieri, F. and Fiore, U. 2009. A nonlinear, recurrence-based approach to traffic classification. *Computer Networks* 53(6), pp. 761–773.

Perdisci, R., Papastergiou, T., Alrawi, O. and Antonakakis, M. 2020. Iotfinder: Efficient large-scale identification of iot devices via passive dns traffic analysis. In: *2020 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE Computer Society, pp. 474–489.

Ramasamy Ramamurthy, S. and Roy, N. 2018. Recent trends in machine learning for human activity recognition—a survey. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery* 8(4), p. e1254.

Ren, J., Dubois, D. J., Choffnes, D., Mandalari, A. M., Kolcun, R. and Haddadi, H. 2019. Information exposure from consumer iot devices: A multidimensional, network-informed measurement approach. In: *Proceedings of the Internet Measurement Conference*. pp. 267–279.

Sarnaik, N. 2020. Human activity recognition using cnn. *Int. J. Sci. Res. Publ* 10, p. 9804.

Shahid, M. R., Blanc, G., Zhang, Z. and Debar, H. 2018. Iot devices recognition through network traffic analysis. In: *2018 IEEE international conference on big data (big data)*. IEEE, pp. 5187–5192.

Shen, J. and Fang, H. 2020. Human activity recognition using gaussian naive bayes algorithm in smart home. In: *Journal of Physics: Conference Series*. IOP Publishing, vol. 1631, p. 012059.

Sivanathan, A., Gharakheili, H. H., Loi, F., Radford, A., Wijenayake, C., Vishwanath, A. and Sivaraman, V. 2018. Classifying iot devices in smart environments using network traffic characteristics. *IEEE Transactions on Mobile Computing* 18(8), pp. 1745–1759.

Sivanathan, A., Sherratt, D., Gharakheili, H. H., Radford, A., Wijenayake, C., Vishwanath, A. and Sivaraman, V. 2017. Characterizing and classifying iot traffic in smart cities and campuses. In: *2017 IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS)*. IEEE, pp. 559–564.

Sumathy, B., Kavimullai, S., Shushmithaa, S. and Anusha, S. S. 2021. Wearable non-invasive health monitoring device for elderly using iot. In: *IOP Conference Series: Materials Science and Engineering*. IOP Publishing, vol. 1012, p. 012011.

Sun, J., Sun, K. and Shenefiel, C. 2019. Automated iot device fingerprinting through encrypted stream classification. In: *International Conference on Security and Privacy in Communication Systems*. Springer, pp. 147–167.

Trimananda, R., Varmarken, J., Markopoulou, A. and Demsky, B. 2019. Pingpong: Packet-level signatures for smart home device events. *arXiv preprint arXiv:1907.11797* .

Trimananda, R., Varmarken, J., Markopoulou, A. and Demsky, B. 2020a. Packet-level signatures for smart home devices. In: *Network and Distributed Systems Security (NDSS) Symposium*. vol. 2020.

Trimananda, R., Varmarken, J., Markopoulou, A. and Demsky, B. 2020b. Packet-Level Signatures for Smart Home Devices. *Proceedings of the 2020 Network and Distributed System Security (NDSS) Symposium* .

Van Jacobson, V. P., Sally Floyd and McCanne, S. 2022. *Tcpdump*, [Online]. Available at: `https://www.tcpdump.org/`. Accessed: 05 July 2022.

Wang, W. and Miao, C. 2018. Activity recognition in new smart home environments. In: *Proceedings of the 3rd International Workshop on Multimedia for Personal Health and Health Care*. pp. 29–37.

Appendix A

A.0.1 Software Used

Table 1: *Software used for generating results*

Filename/ Hardware/ Package/ Algorithm	Supplier/ Source/ Author Website	Use
Python	Python Software Foundation	Used for data preparation and Flask development
JSON	https://www.json.org/json-en.html	Used during Kafka streaming
Pandas	https://pandas.pydata.org/	Reading datasets converting to dataframes
Scikit-learn	https://scikit-learn.org/	Used for preprocessing data and training and testing ML models
Draw.IO	https://app.diagrams.net/	Used for creating flowcharts and diagrams
tshark	https://www.wireshark.org/docs/man-pages/tshark.html	Used for capturing and parsing network traffic
CiCFlowMeter	https://gitlab.com/hieulw/cicflowmeter	Used for generating traffic flows
SQLite	https://www.sqlite.org/index.html	Used as database

A.0.2 CLI/ OS commands

TShark Command to generate (91 feature) Dataset - For IoT versus Non-IoT detection

```
tshark -r 160923.pcap -T fields -E header=y -E separator=, -E quote=d -E
 ↪  occurrence=f -e frame.time -e eth.src -e eth.dst -e _ws.col.Protocol -e
 ↪  _ws.col.Info -e frame.len -e frame.cap_len -e ip.src -e ip.dst -e ip.len -e
 ↪  ip.hdr_len -e ip.flags.df -e ip.flags.mf -e ip.fragment -e
 ↪  ip.fragment.count -e ip.fragments -e ip.ttl -e ip.proto -e ip.version -e
 ↪  ip.tos -e ip.id -e ip.flags -e ip.flags.rb -e ip.frag_offset -e ip.checksum
 ↪ -e ip.dsfield -e tcp.window_size -e tcp.ack -e tcp.seq -e tcp.len -e
 ↪  tcp.stream -e tcp.urgent_pointer -e tcp.flags -e tcp.analysis.ack_rtt -e
 ↪  tcp.segments -e tcp.reassembled.length -e tcp.dstport -e tcp.srcport -e
 ↪  tcp.hdr_len -e tcp.flags.fin -e tcp.flags.syn -e tcp.flags.reset -e
 ↪  tcp.flags.push -e tcp.flags.ack -e tcp.flags.urg -e tcp.flags.cwr -e
 ↪  tcp.checksum -e tcp.time_relative -e tcp.time_delta -e tcp.options.mss_val
 ↪ -e dtls.handshake.extension.len -e dtls.handshake.extension.type -e
 ↪  dtls.handshake.session_id -e dtls.handshake.session_id_length -e
 ↪  dtls.handshake.session_ticket_length -e dtls.handshake.sig_hash_alg_len -e
 ↪  dtls.handshake.sig_len -e dtls.handshake.version -e
 ↪  dtls.heartbeat_message.padding -e dtls.heartbeat_message.payload_length -e
 ↪  dtls.heartbeat_message.payload_length.invalid -e dtls.record.content_type
 ↪ -e dtls.record.length -e dtls.record.sequence_number -e dtls.record.version
 ↪ -e dtls.change_cipher_spec -e dtls.fragment.count -e
 ↪  dtls.handshake.cert_type.types_len -e dtls.handshake.certificate_length -e
 ↪  dtls.handshake.certificates_length -e dtls.handshake.cipher_suites_length
 ↪ -e dtls.handshake.comp_methods_length -e dtls.handshake.exponent_len -e
 ↪  dtls.handshake.extensions_alpn_str -e
 ↪  dtls.handshake.extensions_alpn_str_len -e
 ↪  dtls.handshake.extensions_key_share_client_length -e
 ↪  tls.handshake.extensions_server_name -e http.server -e http.request -e
 ↪  http.request.method -e http.host -e http.request.uri -e http.user_agent -e
 ↪  udp.port -e frame.time_relative -e frame.time_delta -e dns.ns -e
 ↪  dns.qry.name -e dns.qry.type -e udp.srcport -e udp.dstport -e udp.length -e
 ↪  data.len> 16-09-23_pcap.csv
```

Tshark command to capture live packets and save to file:

```
tshark -i <interface> -w <destination.csv>
```

Editcap command to split pcaps into smaller files based on time duration:

```
editcap -i 30 '/<filepath>/<source>.pcap' '/mnt/c/<destination>.pcap'.
```

CiCFlowmeter (linux/WSL based) command to convert pcap files to statistical flows:

```
    cicflowmeter -f <source.pcap> -c <destination.csv>
```

84 statistical Features extracted from pcap files using CiCFlowmeter - used for Device fingerprinting and device state recognition:

```
'Flow ID', 'Src IP', 'Src Port','Dst IP','Dst Port','Protocol','Timestamp','Flow
 ↪  Duration','Tot Fwd Pkts','Tot Bwd Pkts','TotLen Fwd Pkts','TotLen Bwd
 ↪  Pkts','Fwd Pkt Len Max','Fwd Pkt Len Min','Fwd Pkt Len Mean','Fwd Pkt Len
 ↪  Std','Bwd Pkt Len Max','Bwd Pkt Len Min','Bwd Pkt Len Mean','Bwd Pkt Len
 ↪  Std','Flow Byts/s','Flow Pkts/s','Flow IAT Mean','Flow IAT Std','Flow IAT
 ↪  Max','Flow IAT Min','Fwd IAT Tot','Fwd IAT Mean','Fwd IAT Std','Fwd IAT
 ↪  Max','Fwd IAT Min','Bwd IAT Tot','Bwd IAT Mean','Bwd IAT Std','Bwd IAT
 ↪  Max','Bwd IAT Min','Fwd PSH Flags','Bwd PSH Flags','Fwd URG Flags','Bwd URG
 ↪  Flags','Fwd Header Len','Bwd Header Len','Fwd Pkts/s','Bwd Pkts/s','Pkt Len
 ↪  Min','Pkt Len Max','Pkt Len Mean','Pkt Len Std','Pkt Len Var','FIN Flag
 ↪  Cnt','SYN Flag Cnt','RST Flag Cnt','PSH Flag Cnt','ACK Flag Cnt','URG Flag
 ↪  Cnt','CWE Flag Count','ECE Flag Cnt','Down/Up Ratio','Pkt Size Avg','Fwd Seg
 ↪  Size Avg','Bwd Seg Size Avg','Fwd Byts/b Avg','Fwd Pkts/b Avg','Fwd Blk Rate
 ↪  Avg','Bwd Byts/b Avg','Bwd Pkts/b Avg','Bwd Blk Rate Avg','Subflow Fwd
 ↪  Pkts','Subflow Fwd Byts','Subflow Bwd Pkts','Subflow Bwd Byts','Init Fwd Win
 ↪  Byts','Init Bwd Win Byts','Fwd Act Data Pkts','Fwd Seg Size Min','Active
 ↪  Mean','Active Std','Active Max','Active Min','Idle Mean','Idle Std','Idle
 ↪  Max','Idle Min',
```

A.0.3 Apache Kafka Scripts

A.0.3.1 Kafka Producer

```
#Import the necessary modules
import logging
from datetime import datetime
import subprocess
import sys
import os
import time
basepath = os.path.dirname(__file__)

#Install tcpdump, kafka-python, and cicflowmeter on linux.
# pcapy can be used in place of tcpdump.
```

```
os.system("apt-get install tcpdump")
os.system("pip install kafka-python") #for python 3
os.system("pip install cicflowmeter") #for python 3

from time import sleep
from struct import *
from kafka import KafkaProducer
import json
import csv

producer = KafkaProducer(bootstrap_servers=['localhost:9092'],
                        value_serializer=lambda v:
                        ↪  json.dumps(v).encode('utf-8'))

print("Created Producer\n")

#Ask user for input on which interface to listen from
#put interface in monitor mode

if __name__ ==  '__main__':
    while True:
        dev = 'eth0'

        p = subprocess.Popen(['tcpdump', '-i', dev,
                  '-w', 'test.pcap'], stdout=subprocess.PIPE)
        time.sleep(10)
        p.terminate()

        # requires cicflowmeter to be installed using pip on linux OS
        subprocess.call(['cicflowmeter','-f','test.pcap','-c','flows.csv'])

        with open('flows.csv') as file:
            reader = csv.DictReader(file, delimiter=";")
            for row in reader:
                producer.send(topic='flows', value=row)
                producer.flush()

        print("message sent")
```

```
#********************************
# To install CICFLOWMETER
# os.system("pip install cicflowmeter")
# os.system("git clone https://gitlab.com/hieulw/cicflowmeter && cd
↪  cicflowmeter")
# os.system("python setup.py install")

# Requirements:
# Install
# numpy==1.18.0 scipy==1.4.1 scapy==2.4.3
```

A.0.3.2 Kafka Consumer

```
# Import KafkaConsumer from Kafka library
from kafka import KafkaConsumer

kafka_topic_name = "flows"
kafka_bootstrap_servers = "localhost:9092"

# Initialize consumer variable
consumer = KafkaConsumer(kafka_topic_name, fetch_max_wait_ms=0)

# Read and print message from consumer
for message in consumer:
    print(message) # or parse it according to tool's requirements
    print('message received', time.time())
```

A.0.4 User Interface

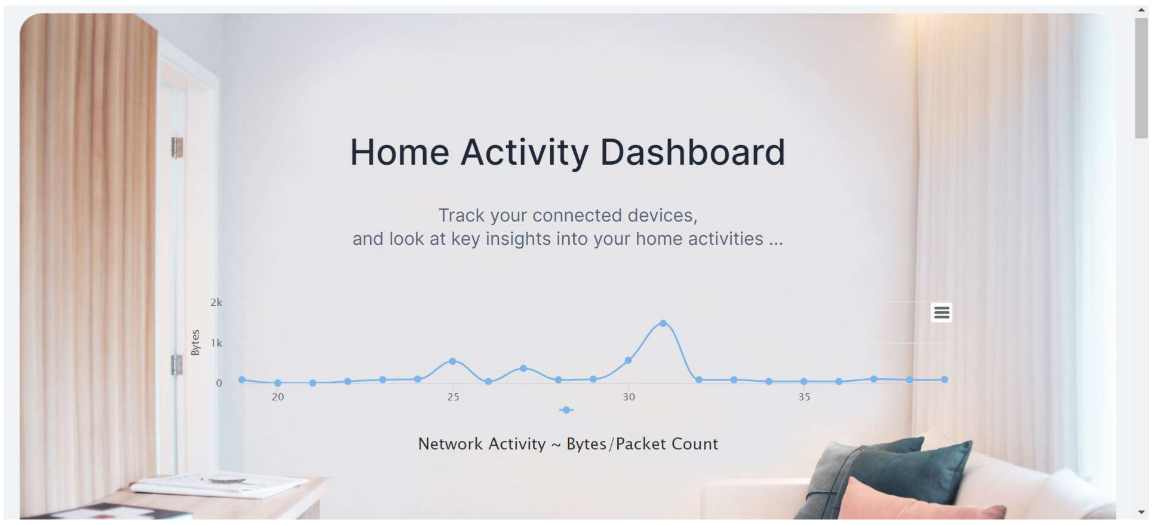
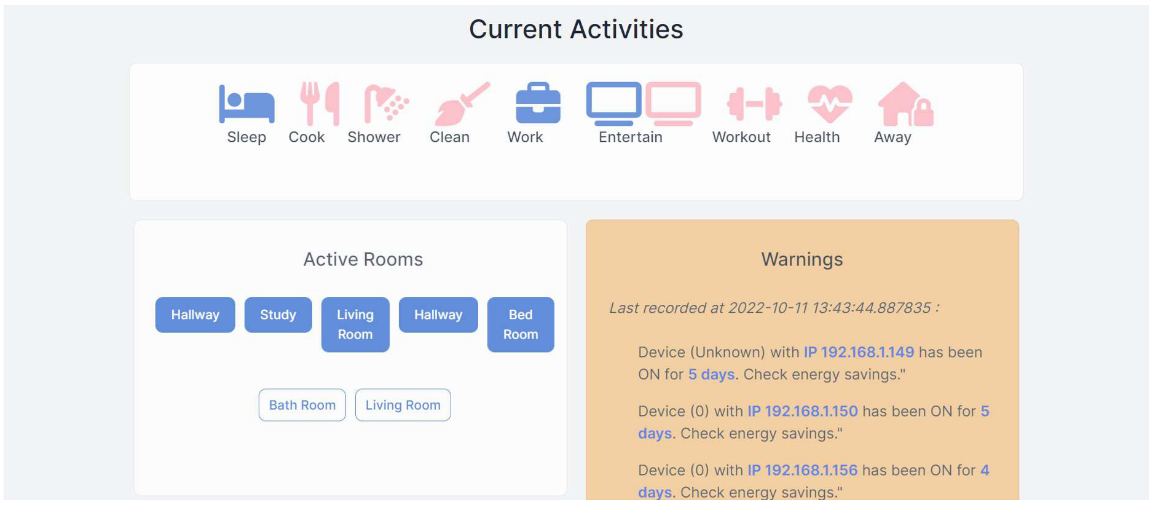

Home Activity Dashboard
Track your connected devices,
and look at key insights into your home activities ...
Bytes
2k
1k
0
20
25
30
35
Network Activity ~ Bytes/Packet Count

Figure 1: *Homepage: Real-time graph showing bytes/second sent over the local network*



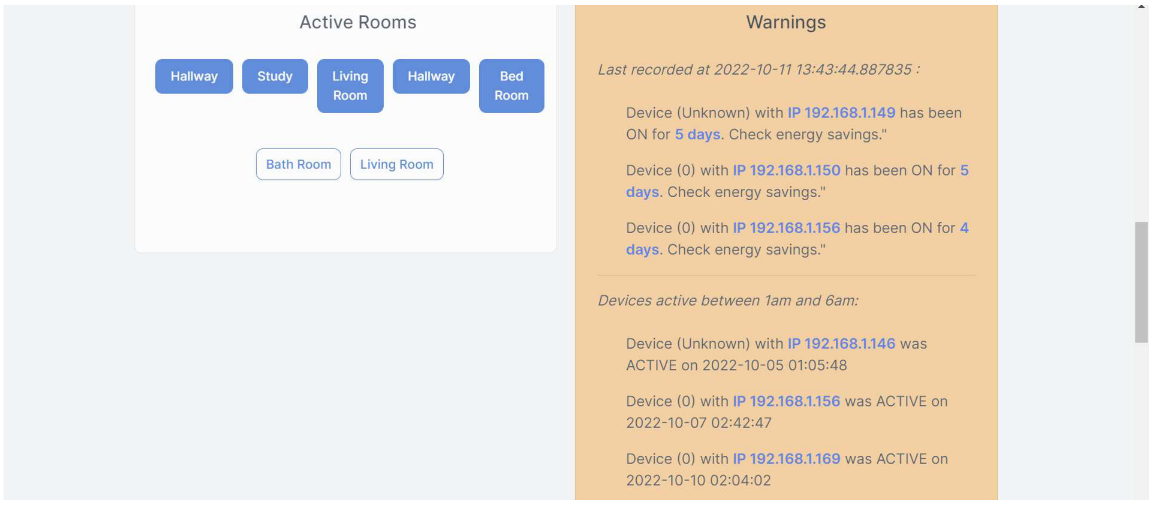
Last recorded at 2022-10-11 13:43:44.887835 :

Device (Unknown) with IP 192.168.1.149 has been ON for 5 days. Check energy savings."

Device (0) with IP 192.168.1.150 has been ON for 5 days. Check energy savings."

Device (0) with IP 192.168.1.156 has been ON for 4 days. Check energy savings."

Figure 2: *Homepage: Current Activities and Active spaces in the Smart Home*



Warnings

Last recorded at 2022-10-11 13:43:44.887835 :

Device (Unknown) with IP 192.168.1.149 has been ON for 5 days. Check energy savings."

Device (0) with IP 192.168.1.150 has been ON for 5 days. Check energy savings."

Device (0) with IP 192.168.1.156 has been ON for 4 days. Check energy savings."

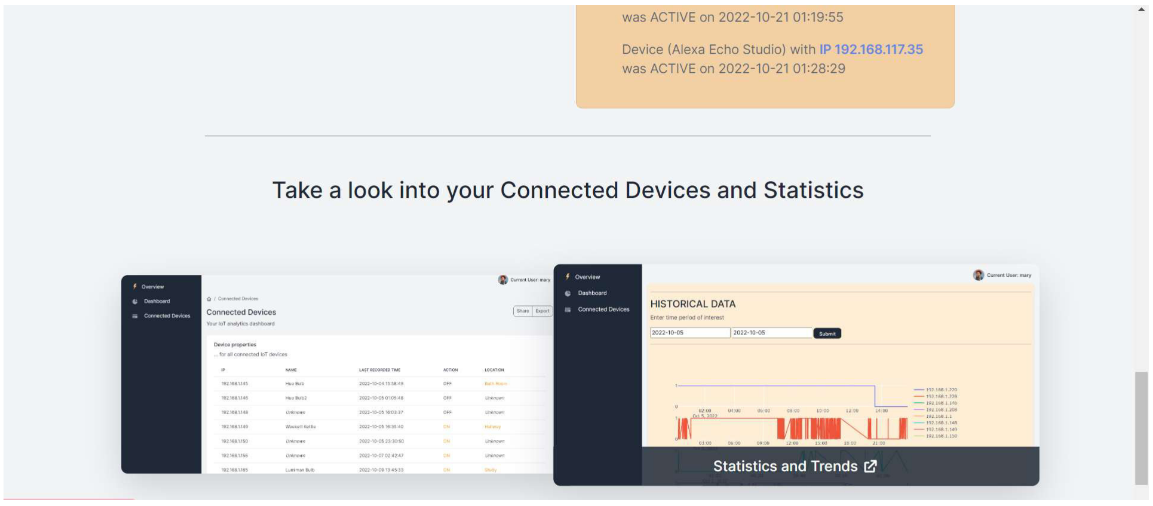
Devices active between 1am and 6am:

Device (Unknown) with IP 192.168.1.146 was ACTIVE on 2022-10-05 01:05:48

Device (0) with IP 192.168.1.156 was ACTIVE on 2022-10-07 02:42:47

Device (0) with IP 192.168.1.169 was ACTIVE on 2022-10-10 02:04:02

Figure 3: *Homepage: Anomalies activities and warnings*

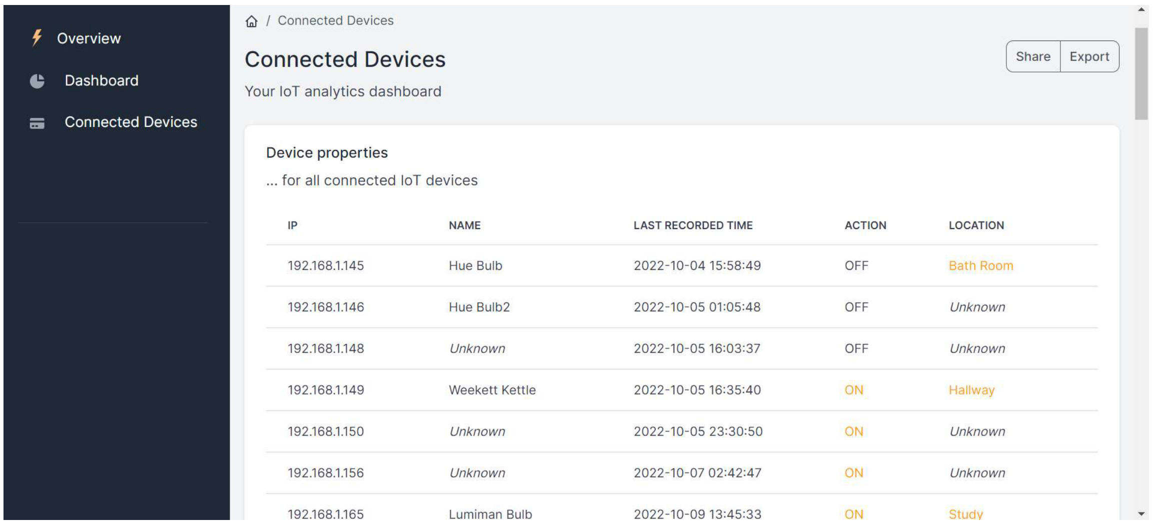


was ACTIVE on 2022-10-21 01:19:55

Device (Alexa Echo Studio) with IP 192.168.117.35 was ACTIVE on 2022-10-21 01:28:29

Take a look into your Connected Devices and Statistics

Figure 4: *Homepage: Navigation*



Connected Devices

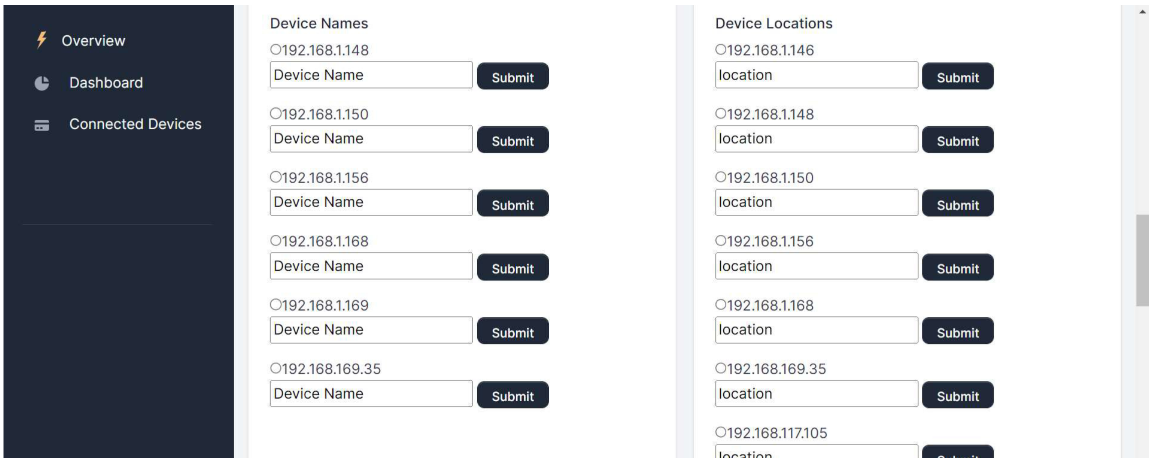
Your IoT analytics dashboard

Device properties

... for all connected IoT devices

IP	NAME	LAST RECORDED TIME	ACTION	LOCATION
192.168.1.145	Hue Bulb	2022-10-04 15:58:49	OFF	Bath Room
192.168.1.146	Hue Bulb2	2022-10-05 01:05:48	OFF	*Unknown*
192.168.1.148	*Unknown*	2022-10-05 16:03:37	OFF	*Unknown*
192.168.1.149	Weekett Kettle	2022-10-05 16:35:40	ON	Hallway
192.168.1.150	*Unknown*	2022-10-05 23:30:50	ON	*Unknown*
192.168.1.156	*Unknown*	2022-10-07 02:42:47	ON	*Unknown*
192.168.1.165	Lumiman Bulb	2022-10-09 13:45:33	ON	Study

Figure 5: *Connected Devices: Overview of all device states*



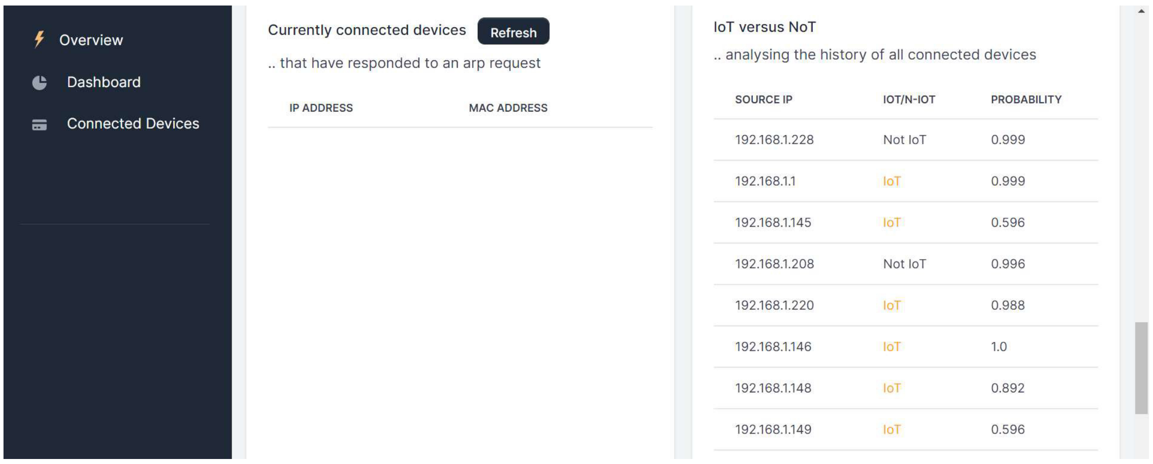
Device Names

○192.168.1.148	
Device Name	Submit
○192.168.1.150	
Device Name	Submit
○192.168.1.156	
Device Name	Submit
○192.168.1.168	
Device Name	Submit
○192.168.1.169	
Device Name	Submit
○192.168.169.35	
Device Name	Submit

Device Locations

○192.168.1.146	
location	Submit
○192.168.1.148	
location	Submit
○192.168.1.150	
location	Submit
○192.168.1.156	
location	Submit
○192.168.1.168	
location	Submit
○192.168.169.35	
location	Submit
○192.168.117.105	

Figure 6: *Connected Devices: User input (supervised learning) to manually input device names and locations (to partially retrain models online*



Currently connected devices

.. that have responded to an arp request

IP ADDRESS	MAC ADDRESS

IoT versus NoT

.. analysing the history of all connected devices

SOURCE IP	IOT/N-IOT	PROBABILITY
192.168.1.228	Not IoT	0.999
192.168.1.1	IoT	0.999
192.168.1.145	IoT	0.596
192.168.1.208	Not IoT	0.996
192.168.1.220	IoT	0.988
192.168.1.146	IoT	1.0
192.168.1.148	IoT	0.892
192.168.1.149	IoT	0.596

Figure 7: *Connected Devices: Detection of IoT and Non IoT devices in the local network*

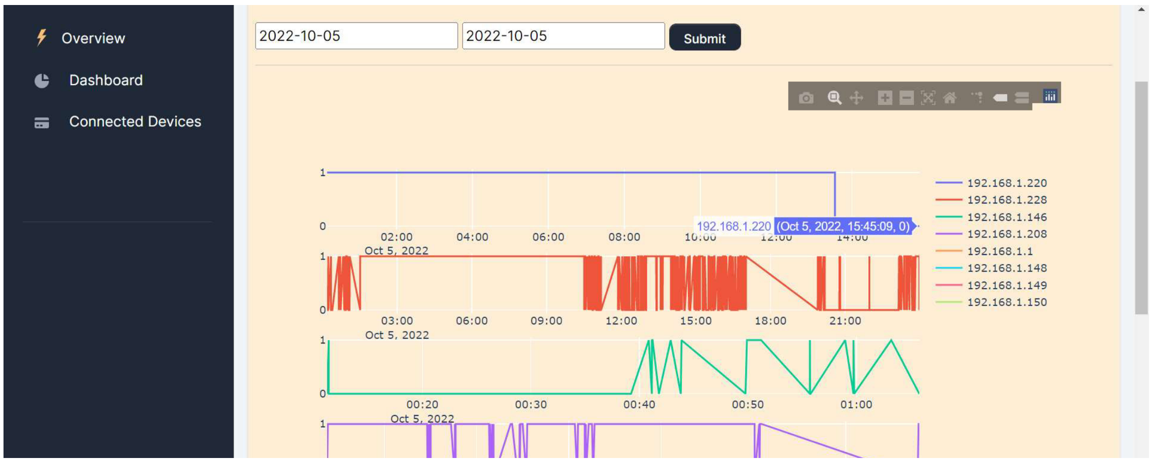
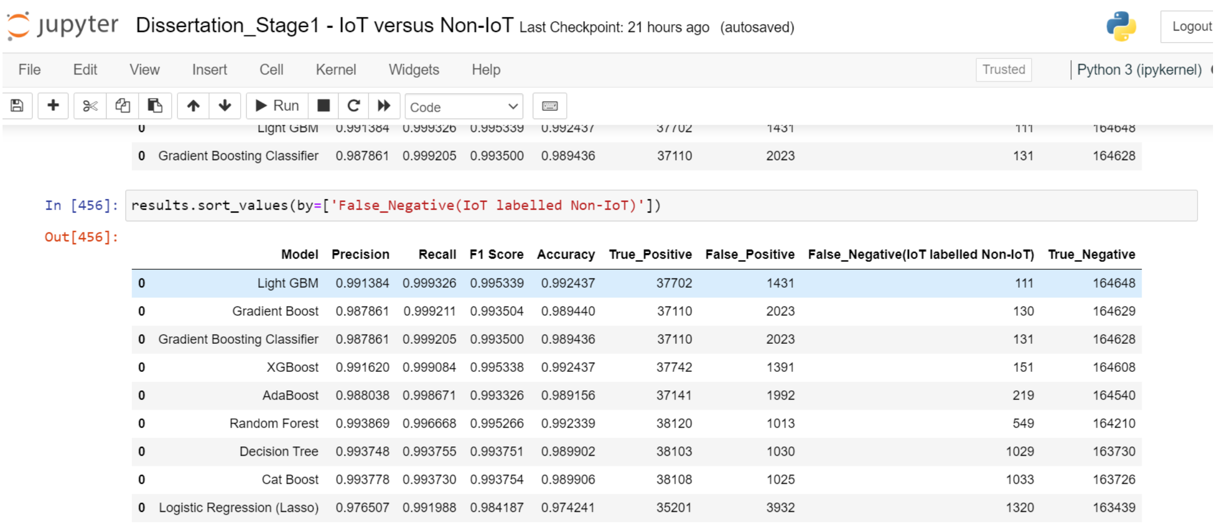


Figure 8: *Dashboard: Historic Trends*

A.0.5 Classification results

A.0.5.1 Stage-1 (IoT vs NoT) Classification results

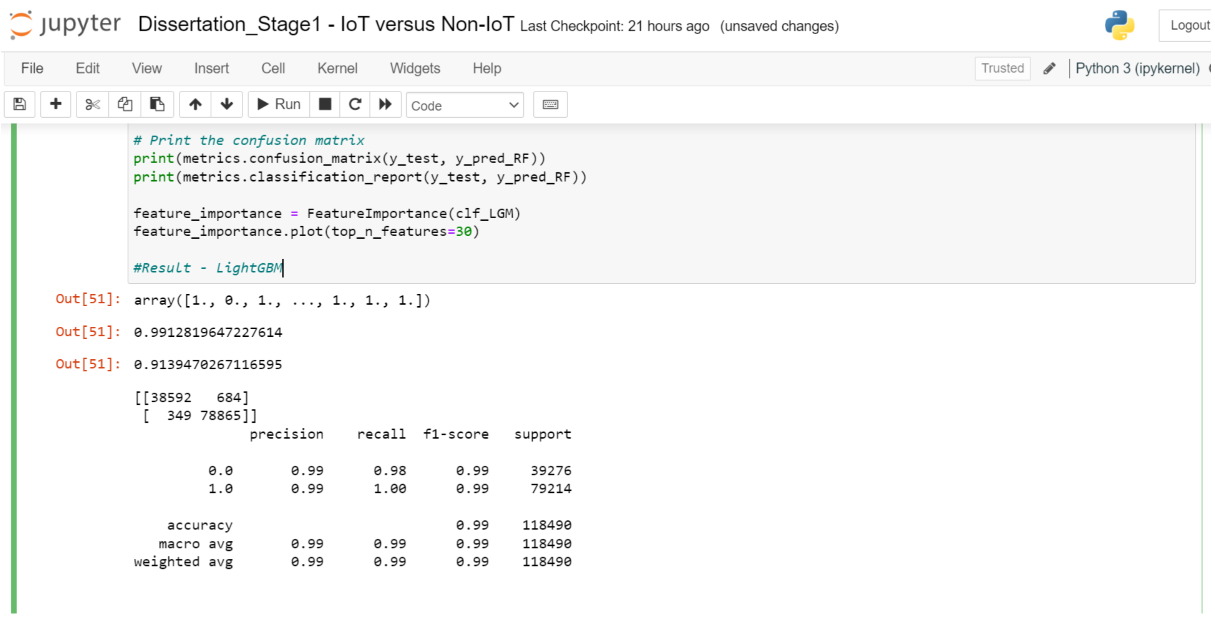


	Model	Precision	Recall	F1 Score	Accuracy	True_Positive	False_Positive	False_Negative(IoT labelled Non-IoT)	True_Negative
0	Light GBM	0.991384	0.999326	0.995339	0.992437	37702	1431	111	164648
0	Gradient Boosting Classifier	0.987861	0.999205	0.993500	0.989436	37110	2023	131	164628

```
In [456]: results.sort_values(by=['False_Negative(IoT labelled Non-IoT)'])
Out[456]:
```

	Model	Precision	Recall	F1 Score	Accuracy	True_Positive	False_Positive	False_Negative(IoT labelled Non-IoT)	True_Negative
0	Light GBM	0.991384	0.999326	0.995339	0.992437	37702	1431	111	164648
0	Gradient Boost	0.987861	0.999211	0.993504	0.989440	37110	2023	130	164629
0	Gradient Boosting Classifier	0.987861	0.999205	0.993500	0.989436	37110	2023	131	164628
0	XGBoost	0.991620	0.999084	0.995338	0.992437	37742	1391	151	164608
0	AdaBoost	0.988038	0.998671	0.993326	0.989156	37141	1992	219	164540
0	Random Forest	0.993869	0.996668	0.995266	0.992339	38120	1013	549	164210
0	Decision Tree	0.993748	0.993755	0.993751	0.989902	38103	1030	1029	163730
0	Cat Boost	0.993778	0.993730	0.993754	0.989906	38108	1025	1033	163726
0	Logistic Regression (Lasso)	0.976507	0.991988	0.984187	0.974241	35201	3932	1320	163439

Figure 9: *Results - Cumulative*

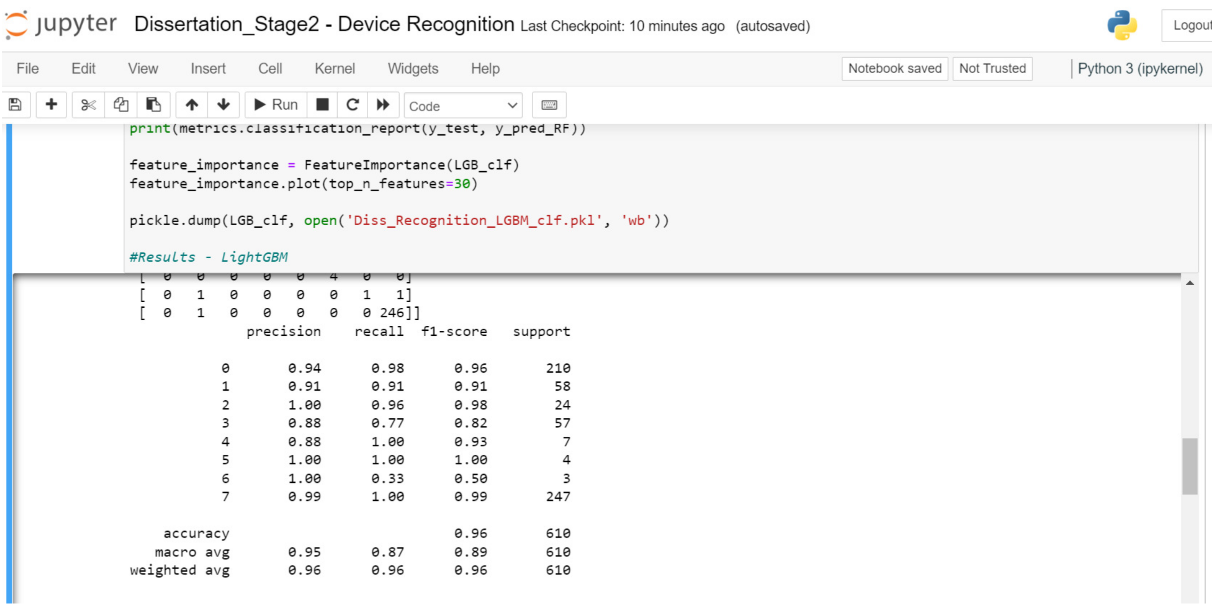

jupyter
Dissertation_Stage1 - IoT versus Non-IoT Last Checkpoint: 21 hours ago (unsaved changes)
Logout
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Trusted
Python 3 (ipykernel)
Run
Code

```
# Print the confusion matrix
print(metrics.confusion_matrix(y_test, y_pred_RF))
print(metrics.classification_report(y_test, y_pred_RF))

feature_importance = FeatureImportance(clf_LGM)
feature_importance.plot(top_n_features=30)

#Result - LightGBM
```

```
Out[51]: array([1., 0., 1., ..., 1., 1., 1.])

Out[51]: 0.9912819647227614

Out[51]: 0.9139470267116595

         [[38592   684]
          [  349 78865]]
                       precision    recall  f1-score   support

                  0.0       0.99      0.98      0.99     39276
                  1.0       0.99      1.00      0.99     79214

             accuracy                           0.99    118490
            macro avg       0.99      0.99      0.99    118490
         weighted avg       0.99      0.99      0.99    118490
```

Figure 10: *Stage 1 - Results - Light Gradient Boosting Machine*

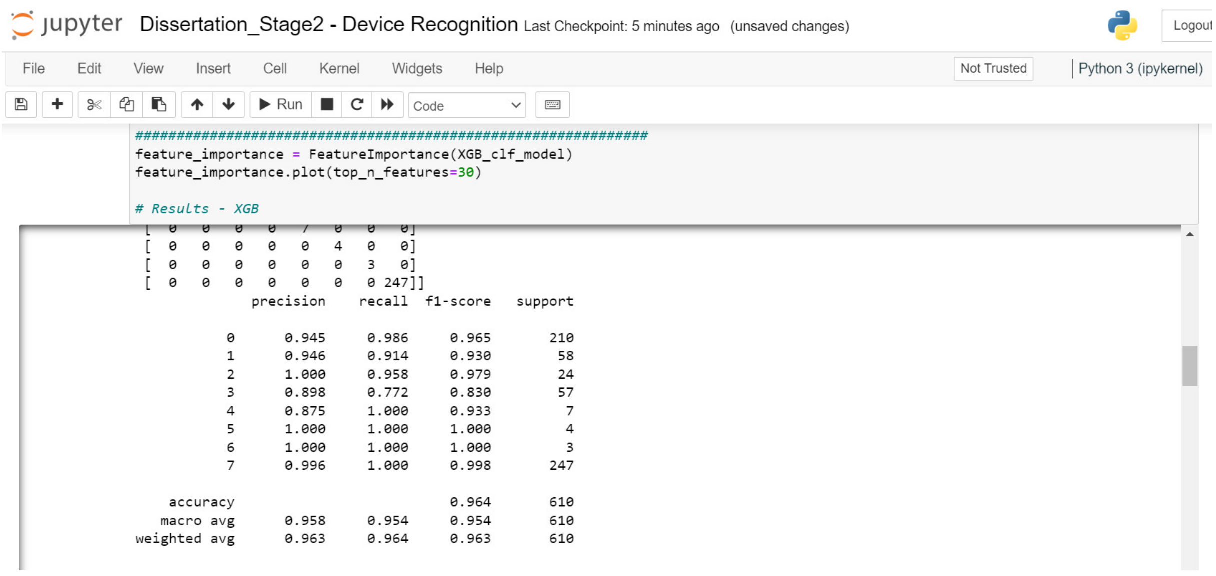
A.0.5.2 Stage-2 (Device Fingerprinting) Classification results


jupyter Dissertation_Stage2 - Device Recognition Last Checkpoint: 10 minutes ago (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Notebook saved
Not Trusted
Python 3 (ipykernel)
Run
Code

```
print(metrics.classification_report(y_test, y_pred_RF))

feature_importance = FeatureImportance(LGB_clf)
feature_importance.plot(top_n_features=30)

pickle.dump(LGB_clf, open('Diss_Recognition_LGBM_clf.pkl', 'wb'))

#Results - LightGBM
```

```
 [  0   0   0   0   0   4   0   0]
 [  0   1   0   0   0   0   1   1]
 [  0   1   0   0   0   0   0 246]]
              precision    recall  f1-score   support

           0       0.94      0.98      0.96       210
           1       0.91      0.91      0.91        58
           2       1.00      0.96      0.98        24
           3       0.88      0.77      0.82        57
           4       0.88      1.00      0.93         7
           5       1.00      1.00      1.00         4
           6       1.00      0.33      0.50         3
           7       0.99      1.00      0.99       247

    accuracy                           0.96       610
   macro avg       0.95      0.87      0.89       610
weighted avg       0.96      0.96      0.96       610
```

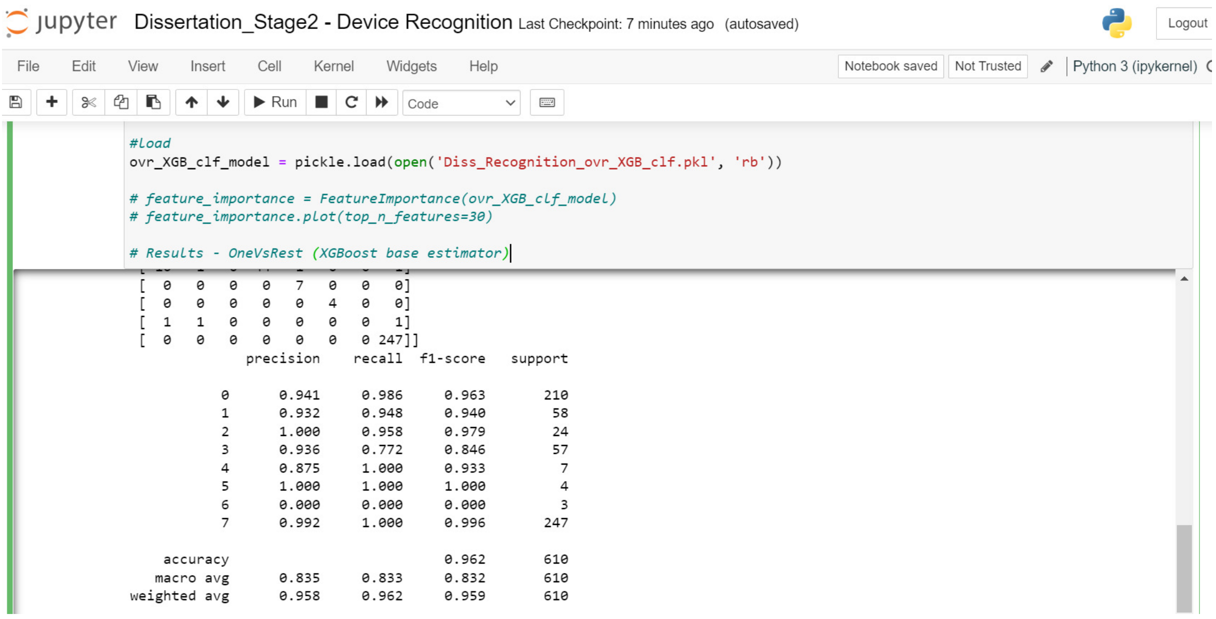
Figure 11: *Stage 2 - Results - Light Gradient Boosting Machine*


jupyter Dissertation_Stage2 - Device Recognition Last Checkpoint: 5 minutes ago (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Not Trusted
Python 3 (ipykernel)
Run
Code

```
##############################################################
feature_importance = FeatureImportance(XGB_clf_model)
feature_importance.plot(top_n_features=30)

# Results - XGB
```

```
 [  0   0   0   0   7   0   0   0]
 [  0   0   0   0   0   4   0   0]
 [  0   0   0   0   0   0   3   0]
 [  0   0   0   0   0   0   0 247]]
              precision    recall  f1-score   support

           0      0.945     0.986     0.965       210
           1      0.946     0.914     0.930        58
           2      1.000     0.958     0.979        24
           3      0.898     0.772     0.830        57
           4      0.875     1.000     0.933         7
           5      1.000     1.000     1.000         4
           6      1.000     1.000     1.000         3
           7      0.996     1.000     0.998       247

    accuracy                          0.964       610
   macro avg      0.958     0.954     0.954       610
weighted avg      0.963     0.964     0.963       610
```

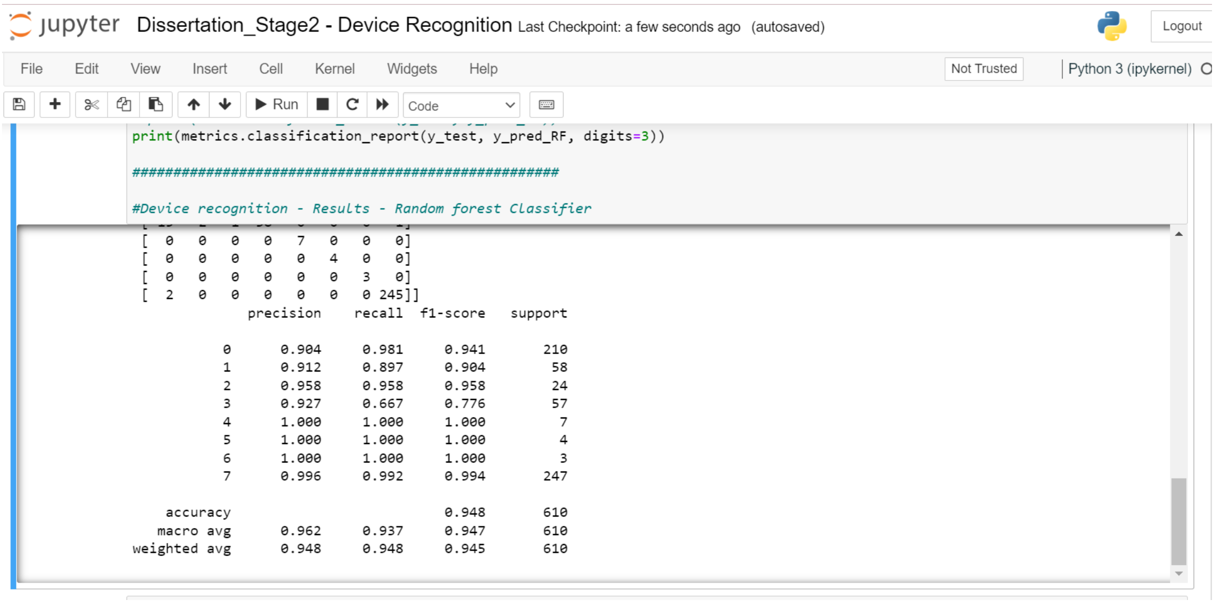
Figure 12: *Stage 2 - Results - Extreme Gradient Boost*


jupyter
Dissertation_Stage2 - Device Recognition Last Checkpoint: 7 minutes ago (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Notebook saved
Not Trusted
Python 3 (ipykernel)
Run
Code

```
#Load
ovr_XGB_clf_model = pickle.load(open('Diss_Recognition_ovr_XGB_clf.pkl', 'rb'))

# feature_importance = FeatureImportance(ovr_XGB_clf_model)
# feature_importance.plot(top_n_features=30)

# Results - OneVsRest (XGBoost base estimator)
```

```
 [  0   0   0   0   7   0   0   0]
 [  0   0   0   0   0   4   0   0]
 [  1   1   0   0   0   0   0   1]
 [  0   0   0   0   0   0   0 247]]
              precision    recall  f1-score   support

           0      0.941     0.986     0.963       210
           1      0.932     0.948     0.940        58
           2      1.000     0.958     0.979        24
           3      0.936     0.772     0.846        57
           4      0.875     1.000     0.933         7
           5      1.000     1.000     1.000         4
           6      0.000     0.000     0.000         3
           7      0.992     1.000     0.996       247

    accuracy                          0.962       610
   macro avg      0.835     0.833     0.832       610
weighted avg      0.958     0.962     0.959       610
```

Figure 13: *Stage 2 - Results - OneVsRest with Extreme Gradient Boost base estimator*

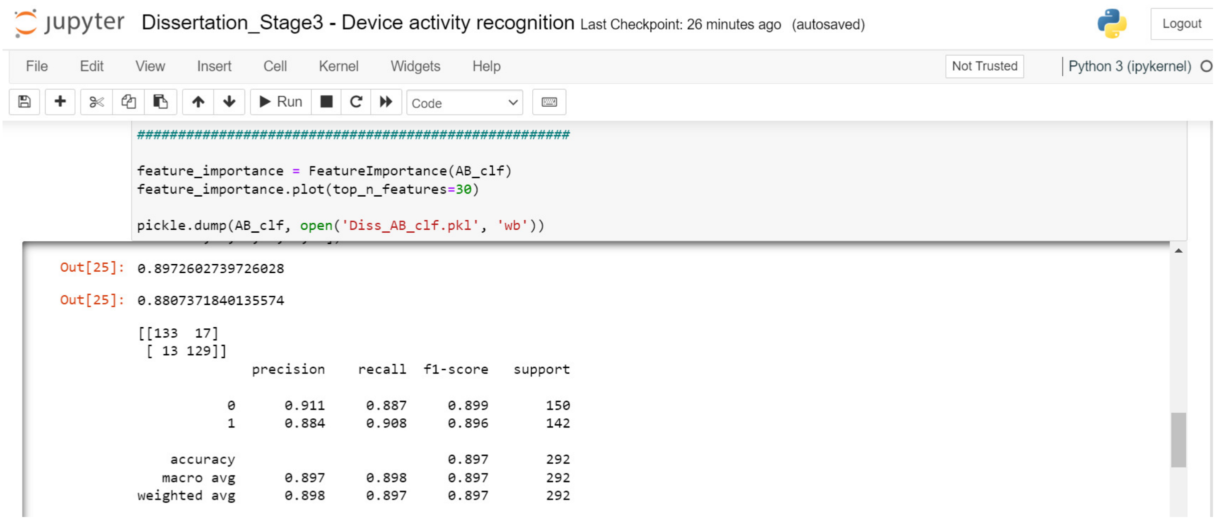

jupyter
Dissertation_Stage2 - Device Recognition Last Checkpoint: a few seconds ago (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Not Trusted
Python 3 (ipykernel)
Run
Code

```
print(metrics.classification_report(y_test, y_pred_RF, digits=3))

######################################################

#Device recognition - Results - Random forest Classifier
```

```
 [  0   0   0   0   7   0   0   0]
 [  0   0   0   0   0   4   0   0]
 [  0   0   0   0   0   0   3   0]
 [  2   0   0   0   0   0   0 245]]
              precision    recall  f1-score   support

           0      0.904     0.981     0.941       210
           1      0.912     0.897     0.904        58
           2      0.958     0.958     0.958        24
           3      0.927     0.667     0.776        57
           4      1.000     1.000     1.000         7
           5      1.000     1.000     1.000         4
           6      1.000     1.000     1.000         3
           7      0.996     0.992     0.994       247

    accuracy                          0.948       610
   macro avg      0.962     0.937     0.947       610
weighted avg      0.948     0.948     0.945       610
```

Figure 14: *Stage 2 - Results - Random Forest*

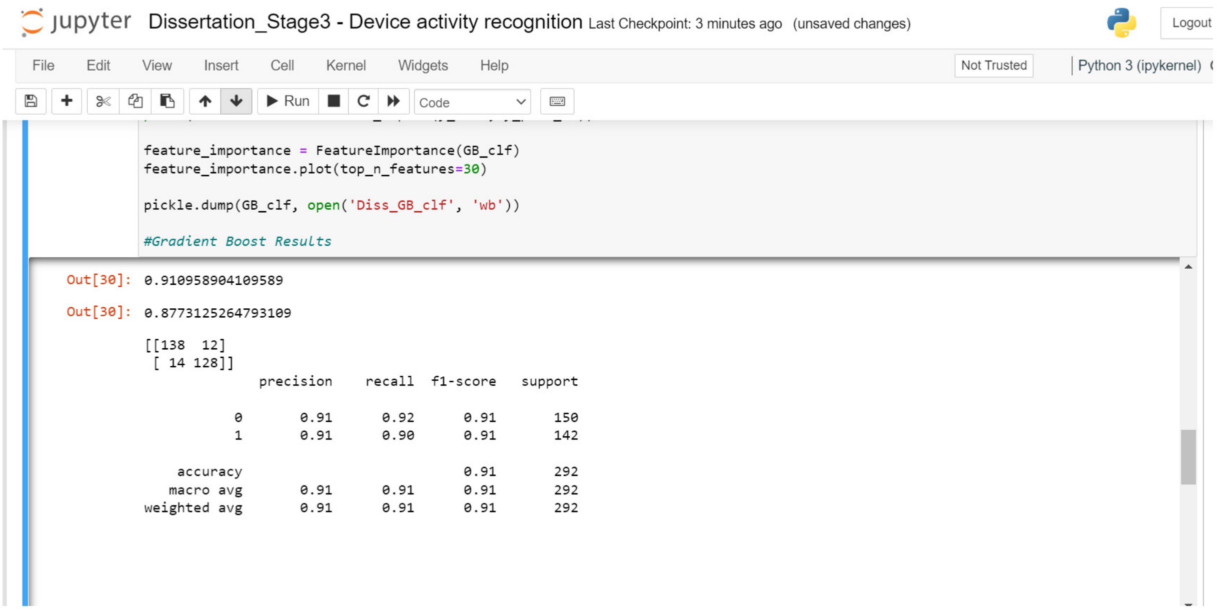
A.0.5.3 Stage-3 (Device State) Classification results


jupyter
Dissertation_Stage3 - Device activity recognition Last Checkpoint: 26 minutes ago (autosaved)
Logout
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Not Trusted
Python 3 (ipykernel)
Run
Code

```
#######################################################

feature_importance = FeatureImportance(AB_clf)
feature_importance.plot(top_n_features=30)

pickle.dump(AB_clf, open('Diss_AB_clf.pkl', 'wb'))
```

```
Out[25]: 0.8972602739726028

Out[25]: 0.8807371840135574

         [[133  17]
          [ 13 129]]
                       precision    recall  f1-score   support

                    0      0.911     0.887     0.899       150
                    1      0.884     0.908     0.896       142

             accuracy                          0.897       292
            macro avg      0.897     0.898     0.897       292
         weighted avg      0.898     0.897     0.897       292
```

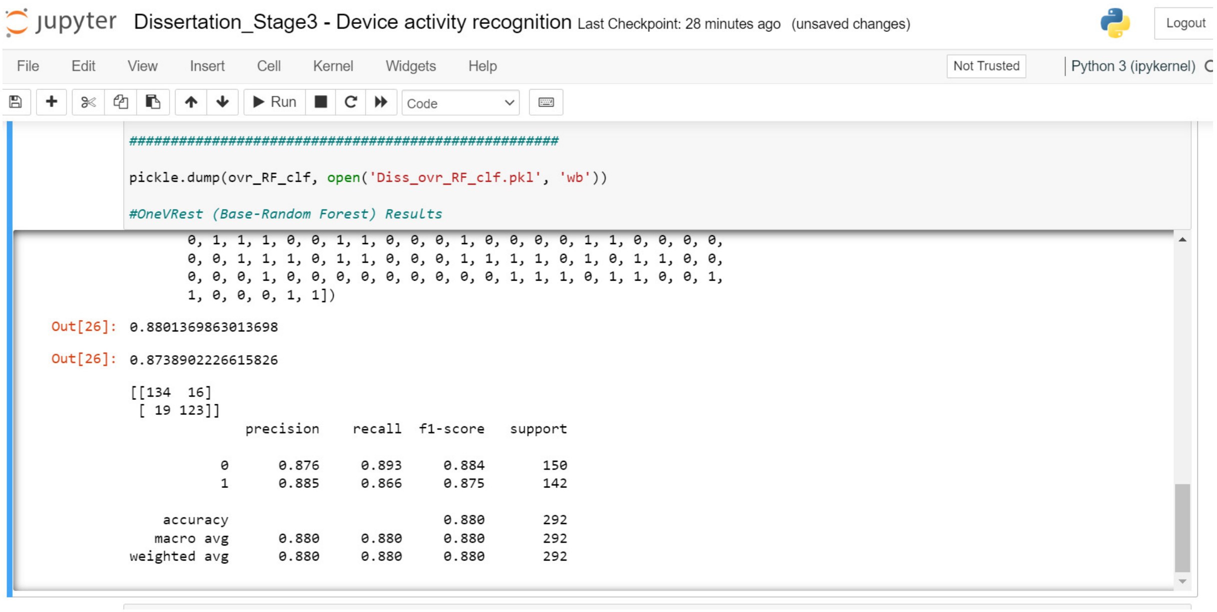
Figure 15: *Stage 3 - Results - AdaBoost*


jupyter
Dissertation_Stage3 - Device activity recognition Last Checkpoint: 3 minutes ago (unsaved changes)
Logout
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Not Trusted
Python 3 (ipykernel)
Run
Code

```
feature_importance = FeatureImportance(GB_clf)
feature_importance.plot(top_n_features=30)

pickle.dump(GB_clf, open('Diss_GB_clf', 'wb'))

#Gradient Boost Results
```

```
Out[30]: 0.910958904109589

Out[30]: 0.8773125264793109

         [[138  12]
          [ 14 128]]
                       precision    recall  f1-score   support

                    0       0.91      0.92      0.91       150
                    1       0.91      0.90      0.91       142

             accuracy                           0.91       292
            macro avg       0.91      0.91      0.91       292
         weighted avg       0.91      0.91      0.91       292
```

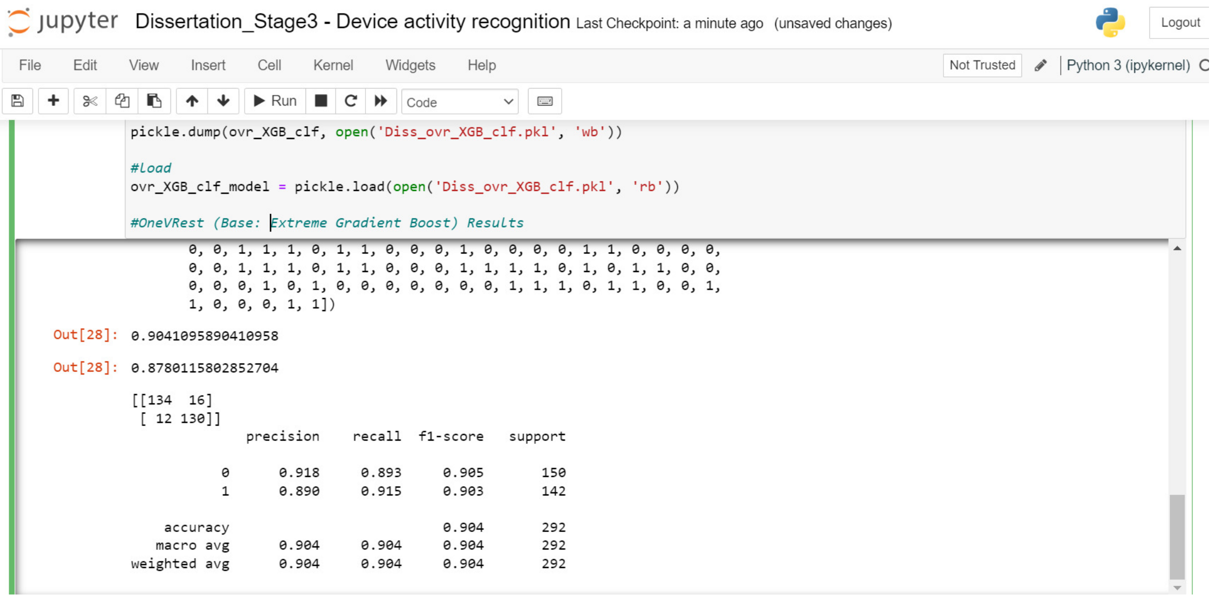
Figure 16: *Stage 3 - Results - Gradient Boost*


jupyter
Dissertation_Stage3 - Device activity recognition Last Checkpoint: 28 minutes ago (unsaved changes)
Logout
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Not Trusted
Python 3 (ipykernel)
Run
Code

```
######################################################

pickle.dump(ovr_RF_clf, open('Diss_ovr_RF_clf.pkl', 'wb'))

#OneVRest (Base-Random Forest) Results
```

```
       0, 1, 1, 1, 0, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0,
       0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0,
       0, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1,
       1, 0, 0, 0, 1, 1])

Out[26]: 0.8801369863013698

Out[26]: 0.873890226615826

[[134  16]
 [ 19 123]]
              precision    recall  f1-score   support

           0      0.876     0.893     0.884       150
           1      0.885     0.866     0.875       142

    accuracy                          0.880       292
   macro avg      0.880     0.880     0.880       292
weighted avg      0.880     0.880     0.880       292
```

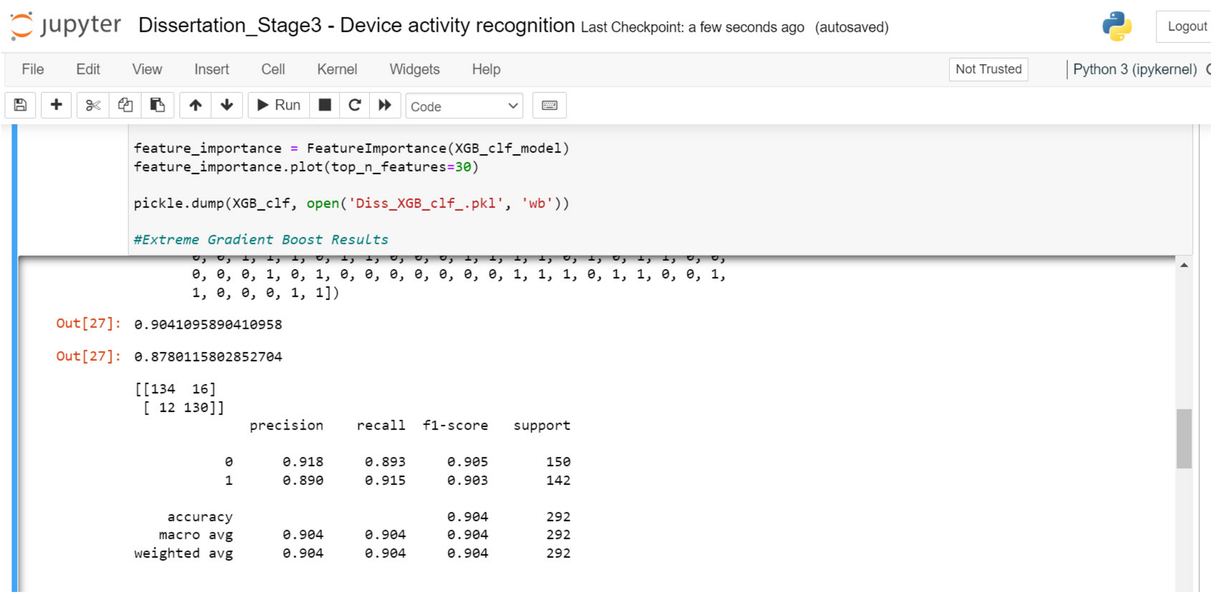
Figure 17: *Stage 3 - Results - OneVsRest with Random Forest Base estimator*


jupyter
Dissertation_Stage3 - Device activity recognition Last Checkpoint: a minute ago (unsaved changes)
Logout
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Not Trusted
Python 3 (ipykernel)
Run
Code

```
pickle.dump(ovr_XGB_clf, open('Diss_ovr_XGB_clf.pkl', 'wb'))

#Load
ovr_XGB_clf_model = pickle.load(open('Diss_ovr_XGB_clf.pkl', 'rb'))

#OneVRest (Base: Extreme Gradient Boost) Results
```

```
       0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 1, 0, 0, 0, 0,
       0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 0, 1, 1, 1, 1, 0, 1, 0, 1, 1, 0, 0,
       0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1,
       1, 0, 0, 0, 1, 1])

Out[28]: 0.9041095890410958

Out[28]: 0.8780115802852704

[[134  16]
 [ 12 130]]
              precision    recall  f1-score   support

           0      0.918     0.893     0.905       150
           1      0.890     0.915     0.903       142

    accuracy                          0.904       292
   macro avg      0.904     0.904     0.904       292
weighted avg      0.904     0.904     0.904       292
```

Figure 18: *Stage 3 - Results - OneVsRest with ExtremeGradientBoost base estimator*


jupyter
Dissertation_Stage3 - Device activity recognition Last Checkpoint: a few seconds ago (autosaved)
Logout
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Not Trusted
Python 3 (ipykernel)
Run
Code

```
feature_importance = FeatureImportance(XGB_clf_model)
feature_importance.plot(top_n_features=30)

pickle.dump(XGB_clf, open('Diss_XGB_clf_.pkl', 'wb'))

#Extreme Gradient Boost Results
```

```
        0, 0, 0, 1, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 1, 0, 1, 1, 0, 0, 1,
        1, 0, 0, 0, 1, 1])

Out[27]: 0.9041095890410958

Out[27]: 0.8780115802852704

         [[134  16]
          [ 12 130]]
                       precision    recall  f1-score   support

                   0       0.918     0.893     0.905       150
                   1       0.890     0.915     0.903       142

            accuracy                           0.904       292
           macro avg       0.904     0.904     0.904       292
        weighted avg       0.904     0.904     0.904       292
```

Figure 19: *Stage 3 - Results - ExtremeGradientBoost*

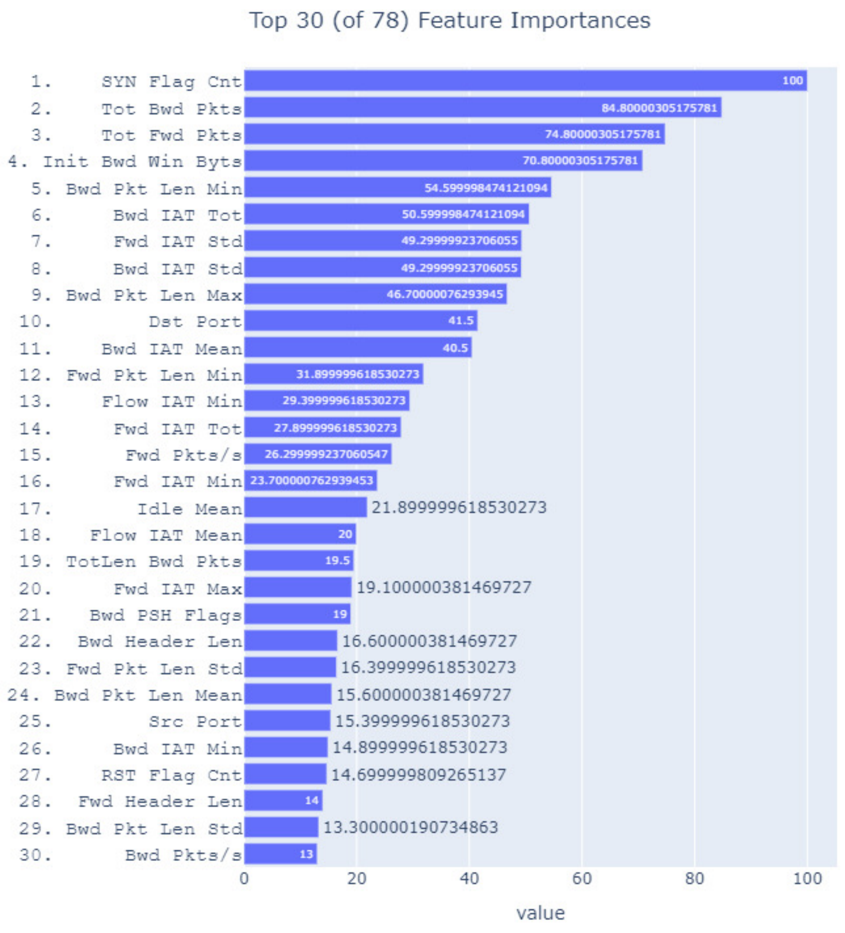


Figure 20: *Stage 3 - Results - Feature Importance Scores - ExtremeGradientBoost*

A.0.6 Code

1. Script for Homepage dynamic graph

```
@blueprint.route('/data', methods=["GET", "POST"])
def data():
    """
    For real-time bytes/second graph
    """
    pkt = sniff(count = 1)[0]
    pktBytes=[]
    pktTimes=[]
    from datetime import datetime
    #Read each packet and append to the lists.
    for p in pkt:
        if IP in p:
            try:
                pktBytes.append(p[IP].len)
                pktTime=datetime.fromtimestamp(p.time)
                pktTimes.append(pktTime.strftime("%Y-%m-%d %H:%M:%S.%f"))
            except:
                pass

    # Convert list to series
    bytes = pd.Series(pktBytes).astype(int)

    # Convert the timestamp list to a pd date_time with the option
     ↪  ‘‘errors=coerce’’ to handle errors.
    times = pd.to_datetime(pd.Series(pktTimes).astype(str),  errors='coerce')
    # times.dt_time()
    # Build the dataframe, set time as index
    df  = pd.DataFrame({'Bytes': bytes, 'Times':times})

    try:
        Bytes = df['Bytes'].iloc[0]
        Time = df['Times'].iloc[0]
    except IndexError:
        Bytes = 0
        Time = 0
        print("Index should be smaller.")   # Throws IndexError: single
         ↪  positional indexer is out-of-bounds
    data = [str(Time),int(Bytes)]
    response = make_response(json.dumps(data))
    response.content_type='application.json'
    return response
```

2. Script for IP -> MAC mapping

-Connected Device Page-

```
import argparse
from scapy.all import *

def arp_scan(ip):
    """
    Performs a network scan by sending ARP requests to an IP address or a range
     ↪  of IP addresses.
    Args:
        ip (str): An IP address or IP address range to scan. For example:
                    - 192.168.1.1 to scan a single IP address
                    - 192.168.1.1/24 to scan a range of IP addresses.
    Returns:
        A list of dictionaries mapping IP addresses to MAC addresses. For
         ↪  example:
        [
            {'IP': '192.168.2.1', 'MAC': 'c4:93:d9:8b:3e:5a'}
        ]
    """
    request = Ether(dst="ff:ff:ff:ff:ff:ff") / ARP(pdst=ip)

    ans, unans = srp(request, timeout=2, retry=1)
    result = []

    for sent, received in ans:
        result.append({'IP': received.psrc, 'MAC': received.hwsrc})

    Data={}
    for mapping in result:
        Data.update({mapping['IP']:mapping['MAC']})
    return Data
```

3. Script to predict IoT versus Non-IoT devices

```
@blueprint.route('/data.html', methods=["GET", "POST"])
def Predict():
    """
    For IoT versus Non-Iot Prediction
    """
    df=pd.read_csv('packetsFinal.csv', on_bad_lines='skip')
    loaded_model =
     ↪  pickle.load(open('apps/templates/ML_models/clf_LGM_23Apr(without
     ↪  router).pkl', 'rb')) #load the saved model
    predictions = loaded_model.predict(df)
    # print prediction probability as well
    predictionProbability = loaded_model.predict_proba(df)
    predictionDict={} # return a dictionary of tuples containing ip address and
     ↪  predictions
    for f, b,c in zip(df.iloc[:, 7], predictions,predictionProbability):
        if str(f).startswith('192.168'):
            predictionDict.update({f:[b,round(max(c),3)]})
    return predictionDict
```

4. Script to retrieve data from database

```
from apps import db

#Read data from sql
@blueprint.route('/DeviceStates', methods=["GET", "POST"])
def table():
    """
    Retrieves latest Device states for all unique MAC values from SQLite db.
    Function is run every 30 seconds using app scheduler (at end of file)
    """
    session = db.session()
    cursor = session.execute('SELECT
     ↪  mac,deviceName,deviceState,location,timeStamp, max(timeStamp) FROM
     ↪  DeviceStates group by mac order by timeStamp')
    resultsDeviceStates = cursor.fetchall()
    return resultsDeviceStates
```

5. Script to output data from above two functions to front end from Database

```
@blueprint.route('/transactions.html', methods=["GET"])
def add_data():
    """
    Output from below functions is directly returned
    to the table in the 'transactions.html' template
    """
    data = arp_scan('192.168.1.1/24')
    predictionDict=Predict()
    resultsDeviceStates = table()
    return render_template('home/transactions.html', data=data,
     ↪  predictionDict=predictionDict,resultsDeviceStates=resultsDeviceStates)
```

6. Script to create individual line graphs of each devices' activity for a desired time period, generated through a user query (flask form)

```
import plotly.graph_objects as go
from plotly.subplots import make_subplots


@blueprint.route('/dashboard', methods=['GET', 'POST'])
def graph():
    """
        Retrieve data from SQLite to Flask graph based on time, with a user
         ↪  query
    """
    session = db.session()
    input1 = '2022-10-05'
    input2 = '2022-10-06'
    if request.method == 'POST':
        input1 = str(request.form.get('input1'))
        input2 = str(request.form.get('input2'))
    cursor = session.execute("SELECT * FROM DeviceStates WHERE timeStamp >= '" +
     ↪  input1 + "' and timestamp <= '" +input2+"' ORDER BY timeStamp")
```

```
# else:
#     cursor = session.execute("SELECT * FROM DeviceStates order by
 ↪  timeStamp")

results = cursor.fetchall()
df = pd.DataFrame(results, columns = ['id', 'mac',
 ↪  'deviceName','timeStamp','deviceState','location'])

# convert to df
listOfMACs = df['mac'].unique()

fig = make_subplots(
rows=len(listOfMACs), cols=1,
shared_yaxes=True
)

for i,unique in enumerate(listOfMACs):
    tempdf = df[ df['mac'] == unique]
    check1 = tempdf['timeStamp']
    check2 = tempdf['deviceState']
    fig.add_trace(go.Scatter(x = check1, y=check2, name = unique,
     ↪  line=dict(width=2)),row=i+1,col=1)
    fig.update_layout({'paper_bgcolor':'rgba(0,0,0,0)',
    'plot_bgcolor':'rgba(0,0,0,0)','height':900, 'width':900,})
    fig.update_yaxes(categoryorder='array', categoryarray= ['0', '1'])

graphJSON = json.dumps(fig, cls=plotly.utils.PlotlyJSONEncoder)
return render_template('home/dashboard.html', graphJSON=graphJSON)
```

7. Script to write user-input(label) for Device Location from the front-end to the associated table in the database

```
@blueprint.route('/transactions.html', methods=["POST"])
def dictionary():
    """
    User input from the flask form is written
    to the DeviceLocations table in the database
    """
    # locationTable={}
    if request.method == 'POST':
        ip = str(request.form.get('result.mac'))
        location = str(request.form.get('location'))
        # locationTable.update({ip:location})
        record = DeviceLocations(ip, location)
        try:
            db.session.add(record)
            db.session.commit()
            print('adding locations worked')
        except:
            print('adding locations did not work')

    # Try and alter the device states table based on location
    num_rows_updated =
     ↪  DeviceStates.query.filter_by(mac=ip).update(dict(location=location))
    db.session.commit()

    return redirect('/transactions.html')
```

8. Script to write user-input(label) for Device Name from the front-end to the associated table in the database

```
@blueprint.route('/devicenames', methods=["POST"])
def devicenames():
    """
    User input from the flask form is written
    to the DeviceNames table in the database
    """
    # locationTable={}
    if request.method == 'POST':
        ip = str(request.form.get('result.mac'))
        name = str(request.form.get('name'))
        record = DeviceNames(ip, name)
        try:
            db.session.add(record)
            db.session.commit()
            print('Names worked')
        except:
            print('Names did not work')

    # Try and alter the device states table based on location
    num_rows_updated =
     ↪  DeviceStates.query.filter_by(mac=ip).update(dict(deviceName=name))
    db.session.commit()

    return redirect('/transactions.html')
```

9. Script to predict device's names and locations from 30-second traffic flows using relevant Stage2 and Stage3 classifiers, and write this to a table in the database, for functions like graphing and logging

```
def PredictDevice():
    """
    Runs predictions (for device name and device state) on all 30-second flows
    and writes entries to SQLite db
    """
    #every 30 seconds:
    # app = current_app
    with app.app_context():
        df_device=pd.read_csv('flowsOriginal.csv', on_bad_lines='skip') #Clean
         ↪  out
        loaded_model_deviceRec =
         ↪  pickle.load(open('apps/templates/ML_models/Diss_Recognition_LGBM_clf.pkl',
         ↪  'rb')) #load the saved model
        loaded_model_deviceState =
         ↪  pickle.load(open('apps/templates/ML_models/Diss_XGB_clf_.pkl',
         ↪  'rb'))

        for row in range(len(df_device.index)):
            if str(df_device.loc[[row]].iloc[:,
             ↪  0].values[0]).startswith('192.168'):
                predictionDeviceRec =
                 ↪  loaded_model_deviceRec.predict(df_device.loc[[row]])
                predictionDeviceState =
                 ↪  loaded_model_deviceState.predict(df_device.loc[[row]])

                #Add probability
                probabilityThreshold =
                 ↪  loaded_model_deviceRec.predict_proba(df_device.loc[[row]])
                maxProb = [max(i) for i in probabilityThreshold]
                if maxProb[0] <= 0.9:
                    deviceName = 'Unknown'
                    # if request.method == 'POST':
                    #     deviceName = str(request.form.get('Device_name'))
                else:
                    deviceName = str(predictionDeviceRec[0]) #Check if works
                #Add values to df
```

```
# db=get_db()
mac = str(df_device.loc[[row]].iloc[:, 0].values[0])
# timeStamp = 'hi'
timeStamp = str(df_device.loc[[row]].iloc[:, 5].values[0])
deviceState = str(predictionDeviceState[0])
location = "Unknown"
session = db.session()
# cursor = session.execute('SELECT id, ip, location, max(id)
 ↪  FROM DeviceLocations GROUP BY ip ORDER BY id')
cursor = session.execute('SELECT *, max(id) FROM DeviceLocations
 ↪  GROUP BY ip ORDER BY id')
resultsDeviceLocations = cursor.fetchall()
for result in resultsDeviceLocations:
    if mac == result.ip:
        location = result.location

print(mac,location)
record = DeviceStates(mac, deviceName, timeStamp, deviceState,
 ↪  location)
try:
    db.session.add(record)
    db.session.commit()
    print('worked')
except:
    print('did not work')
```

10. Script to capture live network traffic every 30 seconds, convert it to a flow-based feature-set (CSV) and a packet 'metadata' based feature-set (CSV), save these CSV files temporarily (for future re-training use), and and then serve predictions on each sample

```
# @blueprint.route('/Packetsniff', methods=("GET","POST"))
def Packetsniff():
    """
    Back-end function to sniff (live) packets in 30 second intervals
    and convert to statistical flows
    """
    # #Write to and read from temporary file
    # make_path('C:/Program Files/Wireshark') #path to tshark on
    platform. Requires tshark installed separately

    dev = 'Wi-Fi'

    #requires tshark downloaded in linux or saved in same windows subdirectory

    p = subprocess.Popen(['tshark', '-i', 'Wi-Fi', '-w',
    '/path/to/folder/pcapy_test3.pcap'],cwd="/path/to/Wireshark",
    stdout=subprocess.PIPE,shell=True)
    time.sleep(30)
    p.terminate()

    #Requires cicflowmeter downloaded in linux

     ↪  subprocess.call(['wsl','cicflowmeter','-f','/path/to/folder/pcapy_test3.pcap','-c',
     ↪  'flowsOriginal.csv'])
    os.system('wsl tshark -r /path/to/folder/pcapy_test3.pcap -T fields
    -E header=y -E separator=, -E quote=d -E occurrence=f -e frame.time
    -e eth.src -e eth.dst -e _ws.col.Protocol -e _ws.col.Info -e
    frame.len -e frame.cap_len -e ip.src -e ip.dst -e ip.len -e
    ip.hdr_len -e ip.flags.df -e ip.flags.mf -e ip.fragment -e
    ip.fragment.count -e ip.fragments -e ip.ttl -e ip.proto -e
    ip.version -e ip.tos -e ip.id -e ip.flags -e ip.flags.rb -e
    ip.frag_offset -e ip.checksum -e ip.dsfield -e tcp.window_size -e
    tcp.ack -e tcp.seq -e tcp.len -e tcp.stream -e tcp.urgent_pointer
    -e tcp.flags -e tcp.analysis.ack_rtt -e tcp.segments -e
    tcp.reassembled.length -e tcp.dstport -e tcp.srcport -e tcp.hdr_len
-e tcp.flags.fin -e tcp.flags.syn -e tcp.flags.reset -e
tcp.flags.push -e tcp.flags.ack -e tcp.flags.urg -e tcp.flags.cwr
-e tcp.checksum -e tcp.time_relative -e tcp.time_delta -e
tcp.options.mss_val -e dtls.handshake.extension.len -e
dtls.handshake.extension.type -e dtls.handshake.session_id -e
dtls.handshake.session_id_length -e
dtls.handshake.session_ticket_length -e
dtls.handshake.sig_hash_alg_len -e dtls.handshake.sig_len -e
dtls.handshake.version -e dtls.heartbeat_message.padding -e
dtls.heartbeat_message.payload_length -e
dtls.heartbeat_message.payload_length.invalid -e
dtls.record.content_type -e dtls.record.length -e
dtls.record.sequence_number -e dtls.record.version -e
dtls.change_cipher_spec -e dtls.fragment.count -e
dtls.handshake.cert_type.types_len -e
dtls.handshake.certificate_length -e
dtls.handshake.certificates_length -e
dtls.handshake.cipher_suites_length -e
dtls.handshake.comp_methods_length -e dtls.handshake.exponent_len
-e dtls.handshake.extensions_alpn_str -e
dtls.handshake.extensions_alpn_str_len -e
dtls.handshake.extensions_key_share_client_length -e
tls.handshake.extensions_server_name -e http.server -e http.request
-e http.request.method -e http.host -e http.request.uri -e
http.user_agent -e udp.port -e frame.time_relative -e
frame.time_delta -e dns.ns -e dns.qry.name -e dns.qry.type -e
udp.srcport -e udp.dstport -e udp.length -e data.len>
packetsOriginal.csv')

PredictDevice()

#flows to serve as future input to retrain classifiers
with open('flowsFinal.csv', 'a+', newline='') as outputfile:
    with open('flowsOriginal.csv', newline='') as feed:
        writer = csv.writer(outputfile, delimiter=',', quotechar='"')
        reader = csv.reader(feed, delimiter=',', quotechar='"')
        next(reader)
        for row in reader:
            writer.writerow(row)
outputfile.close()
```

```
    # packets for iot vs not detection
    with open('packetsFinal.csv', 'a+', newline='') as outputfile:
        with open('packetsOriginal.csv', newline='') as feed:
            writer = csv.writer(outputfile, delimiter=',',
            quotechar='"')
            reader = csv.reader(feed, delimiter=',', quotechar='"')
            next(reader)
            for row in reader:
                writer.writerow(row)



# for running this script in the background every 30 seconds
from apscheduler.schedulers.background import BackgroundScheduler
scheduler = BackgroundScheduler(job_defaults={'max_instances': 2})

scheduler.add_job(Packetsniff, 'interval', seconds=30)
# scheduler.add_job(PredictDevice, 'interval', seconds=30)
scheduler.start()
```

11. DeviceStates Table in SQLite

```
class DeviceStates(db.Model):

    __tablename__ = 'DeviceStates'
    # mac = db.Column(db.String(64))
    id = db.Column(db.Integer, primary_key=True)
    mac = db.Column(db.String(64))
    deviceName = db.Column(db.String(64))
    timeStamp = db.Column(db.String(64))
    deviceState = db.Column(db.String(64))
    location = db.Column(db.String(64))

    def __init__(self, mac, deviceName, timeStamp, deviceState, location):
        self.mac = mac
        self.deviceName = deviceName
        self.timeStamp = timeStamp
        self.deviceState = deviceState
        self.location = location
```

12. DeviceLocations Table in SQLite

```
class DeviceLocations(db.Model):

    __tablename__ = 'DeviceLocations'
    # mac = db.Column(db.String(64))
    id = db.Column(db.Integer, primary_key=True)
    ip = db.Column(db.String(64))
    location = db.Column(db.String(64))

    def __init__(self, ip,location):
        self.ip = ip
        self.location = location
```

13. DeviceNames Table in SQLite

```
class DeviceNames(db.Model):

    __tablename__ = 'DeviceNames'
    # mac = db.Column(db.String(64))
    id = db.Column(db.Integer, primary_key=True)
    ip = db.Column(db.String(64))
    name = db.Column(db.String(64))

    def __init__(self, ip,name):
        self.ip = ip
        self.name = name
```