Intelligent Human-Presence Assessment

An IoT Application using RESTful-APIs and Machine Learning

BACHELOR THESIS

YI ZHANG

August 2018

Thesis supervisors:

Prof. Dr. Jacques PASQUIER–ROCHA
Software Engineering Group

Pascal Gremaud

UNIVERSITÉ DE FRIBOURG
UNIVERSITÄT FREIBURG

Software Engineering Group
Department of Informatics
University of Fribourg
(Switzerland)

“IoT and machine learning can lead to better business insights and faster decisions.”

- *Ashley Gorakhpurwalla, president, Dell EMC*

Abstract

Nowadays the Internet of Thing (IoT) has entered people's real life, thanks to the advent of multiple new technologies. More and more smart devices and sensors are connected to the Internet for collecting, transmitting, analyzing and monitoring data for different purpose.

Web services enable seamless connectivity between millions of different devices and web servers. Nowadays RESTful-APIs are among the most popular interfaces due to their intuitive form and compactness.

Thanks to machine learning, the IoT can provide even more intelligent and more useful information. This project provides the detailed implementation, with an explanation, of a presence-detection system, which consists of a Thingy 52/gateway, a Restful web-service based on Python/Flask and machine learning.

Keywords: IoT, web service, RESTful, Machine Learning, Python, Flask, SVM

Preamble

Foreword

This thesis is made as a completion of the bachelor education in Informatics Science at the University Fribourg, Switzerland. The research described herein was conducted under the supervision of Prof. Dr. Jacques Pasquier-Rocha in the department of Informatics Science, University Fribourg, between October 2017 and August 2018.

Acknowledgments

I would like to thank my supervisor Professor Dr. Jacques Pasquier-Rocha for the provision of facilities during the project research. I am extremely grateful for my coaching by Mr. Pascal Gremaud, for his knowledge and support.

I would like to thank Arnaud Durand, which developed the “Thingy:52 gateway”, implemented with NodeJS.

I would also like to thank Nordic Semiconductor for the hardware and development document support.

Finally, I take this opportunity to express my gratitude to my family for their love, unfailing encouragement, and support.

Notations

$\{\ \}$	Set delimiters
$\in$	Element of
$x \rightarrow y$	Mapping from *x* to *y*
$\mathbb{R}^n$	*n*-dimensional vector space
$\|x\|$	Norm of *x*
$\Leftrightarrow$	Material equivalence

Table of Contents

List of Figures

List of Tables

List of Source Code

1 Introduction

1.1 Goal of the Project

The goal of this project is to implement a “smart presence detector” based on the **`Thingy:52`** device. The idea is to use the different available sensors for collecting the necessary environmental data, which can be then persisted in a database so that a machine learning model can assess human presence in a given indoor location.

With this presence-recognition system, a prediction pattern can be generated after using a learning phase. After that, the system can recognize human presence prediction based on real-tine data collection. Based on these predictions, a notification can be sent when the presence status changed. Thanks to a web-service, a notification can be sent in various channels without difficulty. This can be then applied as a smart monitoring/security application or service (i.e. anti-theft). Furthermore, such a notification can be sent directly to local security departments or to the police, or even to an intelligent security system for triggering an alert or any expected reaction.

1.2 Architecture of the Project

This project is designed as a typical *Internet of Thing (IoT)* application based on the **`Thingy:52`** and its gateway, which can communicate a web server through a *RESTful-API*. All the collected data is stored in a database and can be queried easily. The general architecture can be represented in *Figure 1.1*.

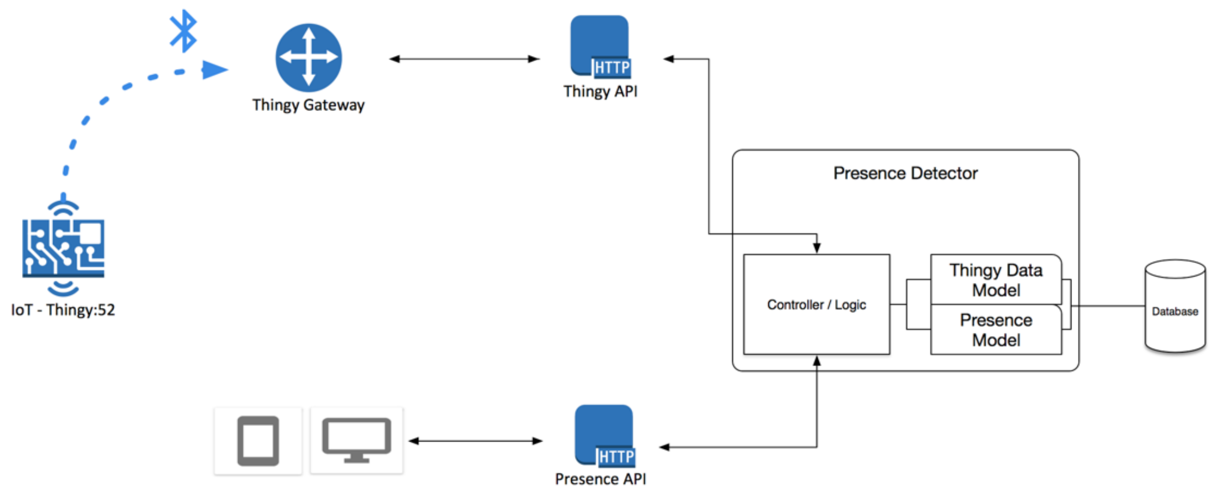


Figure 1.1 *Architecture of the project (taken from [1], accessed on November 30, 2017)*

The sensor data will be first sent to a gateway via its specific *Bluetooth-protocol* (i.e. RFCOMM), then forwarded by the gateway to the server via *HTTP-requests* (web service). The **`Thingy:52`** communicates to web service via a proxy (*Thingy-Gateway*). Since the goal of the project, which is mainly to implement the web service and the machine learning module, more details about the firmware of the **`Thingy:52`** will not be discussed in this thesis.

1.2.1 Components

This project consists of 4 parts.

- Smart device(s) as the sensor.
- A gateway of smart devices, which communicates between the web service and the smart devices.
- A web service with data persistence, which retrieves all raw data and state of presence via *RESTful APIs*.
- A data training application/module supported by machine learning, namely *Support Vector Machine (SVM)*.

1.2.2 Required Hardware

This project is developed based on the following hardware:

- ***`Thingy:52`*** as a smart device
- A Bluetooth USB-adapter for connecting with the ***`Thingy:52`***
- A Mac book pro running MacOS 10.12.6
- A wireless router for a network connection from other clients

`Nordic Thingy:52` is a compact, power-optimized, multi-sensor development kit. It is an easy-to-use development platform, designed to help build IoT prototypes and projects, without the need to build hardware or to write firmware.

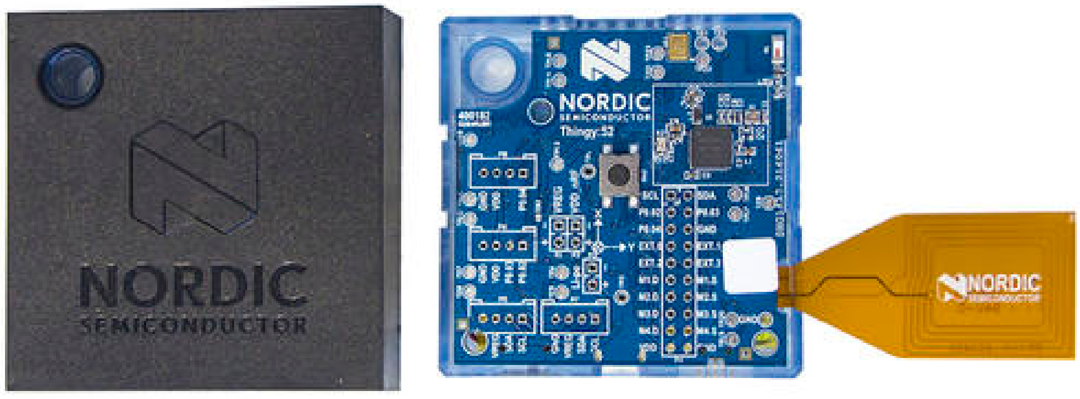


Figure 1.2 *Appearance of the Thingy:52*

1.3 How to Implement the Project

This project consists of a gateway, which has been developed and can be applied out-of-box directly, and a web-server, which is implemented by *Python 3.5+* on *MacOS 10.12.6*.

For simplifying the network settings, a virtual machine container needs to be installed. A virtual machine is deployed (*VirtualBox VM*) to emulate a gateway, which runs an *Ubuntu* with pre-configured settings. Although the gateway has been implemented and prepared to perform, it is still necessary to study the gateway code, because the *thingy-gateway* is an important component of the project as a client which holds the APIs' constraints to follow.

The web-service part of this project is implemented in 5 steps:

1. Implementing the web service based on the RESTful-API and a database, which can serve the gateway for data collection and persistence in a database.
2. Collecting the raw environmental data, states, and sessions of presence for generating a primitive data frame based on time series.
3. Finding a suitable binary classification in accordance with the state of presence by different data graphs.
4. Applying SVM with the training data to obtain a solution, which is called *hyperplane*.
5. Checking the validation of the hyperplane by testing data and optimizing the training model.

From chapters 2 to 4, the implementation of the project will be explained step by step.

2 Implementation of the Web Service

2.1 Preparation and Prerequisites

This chapter covers the sketch-up of a web service with *RESTful-API*.

Before everything starts, it is very important to choose a good toolkit and to configure a correct environment/framework, with which one can simplify the coding and clarify the thinking dramatically. As it is said in the first chapter, this project is aimed at the web service communication and application of machine learning. For such combination, *Python* shall be a suitable choice.

2.1.1 About Python

Python is an interpreted high-level programming language for general-purpose, developed by *Guido van Rossum*. *Python 2* is pre-installed in MacOS. However, this project is implemented with *Python 3*, which can be installed via *Homebrew* easily. After installing the *Python 3*, it is strongly recommended to use **`virtualenv`**, a Python extension lib which can create specified working-environments separately for different projects, in order to avoid unexpected conflicts caused by incompatible libraries.

Although any kind of text editor can be used to code in python, a programming specified editor with syntax-analyzer and auto-completion can relatively improve efficiency. *PyCharm* from *JetBrains* is a well-known *Integrated Development Environment* (*IDE)* much more capable of than just an editor and it is used in this project.

It is assumed that the reader of this thesis has the knowledge about *Python 3*. In addition, knowledge of *JavaScript/NodeJS and Object-Oriented Programming (OOP)* would be also very helpful for understanding this thesis. For any question about *Python 3*, the documentation can be found on the web[1].

2.1.2 Introduction to Web Services and RESTful-APIs

According to the definition by *W3C*, a web service is a software system designed to support interoperable machine-to-machine interaction over networks.

In short terms, a web service is a common language for machines in the network. In order to connect the gateway and server, an agreed contract between the gateway (client) and server must be defined. The server acts as a service provider, which can receive and answer the authenticated requests from a gateway (client). For instance, in this project, the server can accept a request for data-submission from the *thingy-gateway*. Then the data can be stored in the database, aka. *data-persistence*. Meanwhile, another client via a web browser or native mobile apps can submit the human presence by web service too.

The web service concept has been developed for over a decade. A well-known primitive implementation of web service is called *Remote Procedure Call (RPC)*. For an experienced Java programmer, it is known as *Java Remote Method Invocation (Java-RMI)*. *RPC* enables the client/server communication within *Java virtual machine (JVM)* and/or other platforms, as long as they have implemented an *RMI-interface*. However, this is difficult to apply to internet related applications, due to its strong platform-dependency. It is impossible to force all platforms to use *RMI-interfaces*. Furthermore, one *RMI-interface* may not be 100% compatible with another. Meanwhile, there were also some other concurrent *RPC*-solutions to *Java RMI*, i.e. *COBRA*, *DCOM* etc., which have fragmented the standard.

[1] Python 3 documentation https://docs.python.org/3/ (accessed on 20 August 2018)

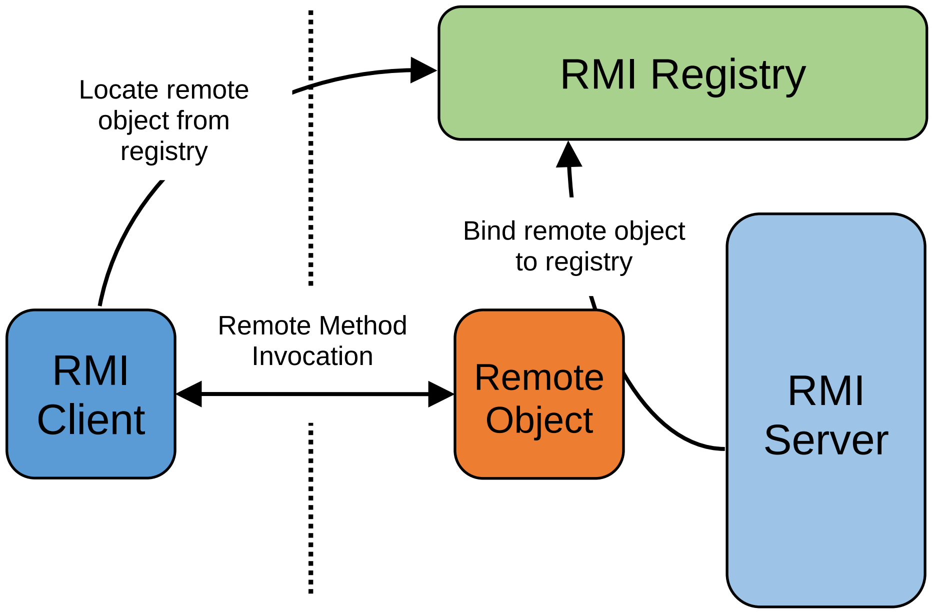


Figure 2.1 ***Java-RMI environment (adapted from [2], accessed on April 05, 2018)***

In order to unify the norms and standards for data-transferring between different web platforms, the web service concept was therefore proposed and implemented with the *Simple Object Access Protocol (SOAP)* and the *Web Service Description Language (WSDL)* in the year 2000. One of the biggest differences between web service and *RPC* is the coupling degree to the platform. Web services concentrated on how to find and bind the service, instead of the detailed business logic. It has a very loose coupling to the platform. Commonly, it contains the following elements:

- A *WSDL*-conformed service endpoint, which describes APIs
- A *SOAP* message object over an application protocol
- A Service broker written in *Universal Description, Discovery and Integration (UDDI)* language

SOAP is a messaging protocol specification for exchanging structured information in the implementation of web services in networks. It uses XML information Set for its message format and it relies on application layer protocols (i.e. *Http, Https, Smtp* and etc.).

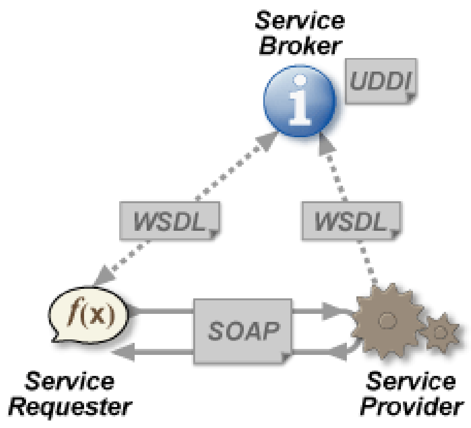


Figure 2.2 ***Architecture of a SOAP-WSDL web service (taken from [3], accessed on April 05, 2018)***

This web service framework developed by *SOAP* and *WSDL* had a great success and it became a symbol for web services since last 15 years. However, *SOAP* is really heavy to read and it is difficult to decide the development style of *WSDL*, especially for a middle scale project. The architecture-designer always needs to evaluate the cost of risk and make a hard choice between "contract-first" (aka. Top-down) or "contract-last" (aka. Bottom-up). In 2004, the W3C extended the definition of web service, because a new concept, *REpresentation State Transfer (REST)*, finally came into view.

The *REST*-compliant web service, in another word, the *RESTful-API* is designed and proposed to represent web resources by using a uniform set of "stateless" operations, which differ from other arbitrary web services, in which the service may expose an arbitrary set of operations.

In the *RESTful* world, *SOAP* and *WSDL* are not needed anymore. *RESTful* is **NOT** a strictly written standard (vs. *SOAP*) but a style. So, it can be only defined by the following constraints instead of the protocols:

- Client-server architecture, in which the communication always starts with the request of the client and ends with the response from the server. (request-response pattern)
- Statelessness, which means that only clients take care of session state. Multiple servers can be accessed in parallel, which enables scalability and improves the performance.
- Cacheability, which can improve performance by caching a resource with expiration during the communication-chain.
- Layered system, any resource should be a set of attributes, which are separated from the primitive object model.
- Code on demand (optional)
- Uniform interface
    - Resource identification in requests, meaning an individual resource is identified in a request. For example, using *URI* in web-based *REST* systems.
    - Resource manipulation through representations. It defines the core thinking of *REST*. It enables the client to perform *Create /Retrieve /Update /Delete (CRUD)* operations on resources.

- Self-descriptive messages, which require less documentation and consequently improve efficiency.
- Hypermedia as the engine of application state, which can inform clients of other possible operations without prior knowledge.

A web-based *RESTful-API*, such as the one implemented in this project, usually uses *XML* or *JSON* as its message object. With 5 *Http-request-methods* (stateless operations), or so-called *Http-verbs*, web resources in the network can be handled simply and intuitively. A resource is wrapped in a form of collection, or just an element.

Http-Verbs	**Resource Collection, such as** **http://api.thingy.com/presences/**	**Resource Element, such as** **http://api.thingy.com/presences/1**
GET	List the URIs and possibly other details of the collection’s members	Retrieve a representation of the addressed member of the collection, expressed in an appropriate Internet media type (i.e. MIME).
PUT	Replace the entire collection with another collection.	Replace the addressed member of the collection, or if it does not exist, create it.
PATCH	(not generally used)	Update the addressed member of the collection.
POST	Create a new entry in the collection. The new entry’s URI is assigned automatically and is usually returned by the operation.	(not generally used)
DELETE	Delete the entire collection	Delete the addressed member of the collection.

Table 2.1 ***Web resource manipulation based on combinations of HTTP-methods and different resource type***

To compare *SOAP* with *REST*, it can be found, that:

- *SOAP* is a stateful, session-based, message-based web service, which is designed as a set of complex actions. The expected data are exposed to be accessible by *SOAP*-message, only when it is requested.
- *RESTful* is a stateless, sessionless, resource-based web service, which is designed as an accessor (setter/getter) to the web resources and performs simple *CRUD* operations on it. All data are exposed to the network as web resources permanently.

In this project, all data shall be exposed as web resources via *URLs* and their manipulations are *CRUD* operations. Therefore, a *RESTful* architecture is very suitable for this project.

2.1.3 Introduction to Flask and Flask-RESTPlus

As we all know, a web service is a kind of web application, which must conform to a standard web service protocol for web servers, such as *Common Gateway Interface (CGI)* to execute programs like console applications running on a server that generates web pages or resources dynamically. In the Python world, *Python Web Server Gateway Interface (WSGI)* is specified in the **`PEP-333`** and has been adopted as a standard of Python web application development. By using *WSGI*, it is independent to develop the web service from the choice of web servers.

Developers can concentrate on their work without consideration of choosing *CGI*, *FastCGI* or ***mod_python***.

For reason of reusing snippets and abstracting the modules, a micro web framework, Flask, was developed and released under BSD license by *Armin Ronacher* 2010, which is based on *WSGI* or more precisely on **Wergzeug**, a python *WSGI* utility library.

Flask is light-weighted to develop, easy to debug and flexible to extend. It supports most web servers. In this project, the **flask-restful** and **flask-restplus** add-ons can be used. With both libraries, the restful-APIs can be generated easily in a layered system. Due to the integration of *swagger-UI*, a utility to generate a standard API-documentation and to render a test interface automatically, the **flask-restplus** add-on is chosen for this project.

2.2 Studying the APIs from Thingy-Gateway

In order to enable the request-response-pattern between the web server and the thingy-gateway, it is necessary to define the contents of requests from the client (*Thingy-Gateway*). The source code of the *Thingy-Gateway* can be obtained from its repository on *GitHub*[2].

Because the implementation of the thingy-gateway is off-topic to this project and the source code is well described, we will not discuss all of it here.

In the file **index.js**, the code related to the APIs is listed in *Code 2.1*.

[2]Nordic Thingy:52 Node.js gateway https://github.com/DurandA/thingy-gateway (accessed on 20. August 23, 2018)

```
        function sendSensorData(data) {
            var jsonDate = (new Date()).toJSON();
            var jsonData = {
                timestamp: jsonDate,
                sensors: data
            };
            return rest.postJson(base_uri + '/' + this.id + '/sensors/', jsonData);
        }
        module.sendTemperature = function(temperature) {
            return sendSensorData.call(this, {
                temperature: temperature
            });
        };
        module.sendPressure = function(pressure) {
            return sendSensorData.call(this, {
                pressure: pressure
            });
        };
        module.sendHumidity = function(humidity) {
            return sendSensorData.call(this, {
                humidity: humidity
            });
        };
        module.sendColor = function(color) {
            return sendSensorData.call(this, {
                color: color
            });
        };
        module.sendGas = function(gas) {
            return sendSensorData.call(this, {
                gas: gas
            });
        };
        module.setButton = function(state) {
            return rest.putJson(base_uri + '/' + this.id + '/sensors/button', {
                state: state
            });
        };
        module.getSettings = function() {
            return rest.get(base_uri + '/' + this.id + '/setup');
        };
        module.getLed = function() {
            return rest.get(base_uri + '/' + this.id + '/actuators/led');
        };
        module.getLedSource = function(onmessage, onerror) {
            var source = new EventSource(base_uri + '/' + this.id + '/actuators/led');
            source.onmessage = onmessage;
            if (onerror) source.onerror = onerror;
        }
```

Code 2.1 ***Snippets in index.js, the entry-point of the thingy-gateway application***

The snippets are fairly easy to understand. The keyword **`this`** is referencing the thingy device, which has successfully been connected to the thingy-gateway and parsed by the notification in the context since the gateway can be connected with multiple **`Thingy:52`** (smart devices) at the same time. It is intuitive to find that there is a total of 2 GET methods, 4 POST methods and 1 PUT method, which can be implemented by overriding 3 correspondent methods:

`rest.get() / rest.postJson() / rest.putJson()`

These 3 methods are built-in methods in **`Restler`**[3], an Http client library for NodeJS.

In *Table 2.2*, all necessary endpoints are listed with URI, parameters, and HTTP-verbs, which are constrained to the API-requests made by the *Thingy-gateway*.

Methods	**URI**	**HTTP-methods/verbs**	**Parameters**
module.sendTemperature	base_url/dev_id/sensors/	POST	temperature
module.sendPressure	base_url/dev_id/sensors/	POST	pressure
module.sendHumidity	base_url/dev_id/sensors/	POST	humidity
module.sendColor	base_url/dev_id/sensors/	POST	colour
module.sendGas	base_url/dev_id/sensors/	POST	gas
module.setButton	base_url/dev_id/sensors/button	PUT	state
module.getSettings	base_url/dev_id/setup	GET	/
module.getLed	base_url/dev_id/actuators/led	GET	/

Table 2.2 ***API endpoints used by the Thingy-gateway***

In the file **`events.js`**, all outgoing messages are bound to thingy notifications as *events/event-listeners*. *Code 2.2* lists the event-binding functions.

[3] Official website of *Restler:* https://github.com/danwrong/restler accessed on 03. November 2017

```
thingy.connectAndSetUp(function(error) {
    console.log('Connected! ' + error ? error : '');
    thingy.on('temperatureNotif', client.sendTemperature.bind(thingy) /*onTemperatureData*/ );
    thingy.on('pressureNotif', client.sendPressure.bind(thingy) /*onPressureData*/ );
    thingy.on('humidityNotif', client.sendHumidity.bind(thingy) /*onHumidityData*/ );
    thingy.on('gasNotif', client.sendGas.bind(thingy) /*onGasData*/ );
    thingy.on('colorNotif', client.sendColor.bind(thingy) /*onColorData*/ );
    thingy.on('buttonNotif', client.setButton.bind(thingy) /*onButtonChange*/ );
    client.getSettings.call(thingy).on('complete', setup.bind(thingy));
    if (enableEventSource) {
        client.getLedSource.call(thingy, function(e) {
            thingy.led_breathe(JSON.parse(e.data));
        });
    } else {
        setInterval(() => client.getLed.call(thingy).on('complete', thingy.led_breathe.bind(thingy)), 1000);
    }
/*rest codes*/
});
```

Code 2.2 ***Snippets for binding event-listeners in event.js***

The specification of parameters are defined in the message constructor, as *Code 2.3* shows. They are located in **`/thingy52/lib/thingy.js`**.

```
Thingy.prototype.onTempNotif = function(data) {
    var integer = data.readInt8(0);
    var decimal = data.readUInt8(1);
    var temperature = integer + (decimal / 100);
    this.emit('temperatureNotif', temperature);
};
Thingy.prototype.onPressNotif = function(data) {
    var integer = data.readInt32LE(0);
    var decimal = data.readUInt8(4);
    var pressure = integer + (decimal / 100);
    this.emit('pressureNotif', pressure);
};
Thingy.prototype.onHumidNotif = function(data) {
    var humid = data.readUInt8(0);
    this.emit('humidityNotif', humid);
};
Thingy.prototype.onGasNotif = function(data) {
    var gas = {
        eco2: data.readUInt16LE(0),
        tvoc: data.readUInt16LE(2)
    }
    this.emit('gasNotif', gas);
};
Thingy.prototype.onColorNotif = function(data) {
    var color = {
        red: data.readUInt16LE(0),
        green: data.readUInt16LE(2),
        blue: data.readUInt16LE(4),
        clear: data.readUInt16LE(6)
    }
    this.emit('colorNotif', color);
};
Thingy.prototype.onBtnNotif = function(data) {
    if (data.readUInt8(0)) {
        this.emit('buttonNotif', 'Pressed');
    } else {
        this.emit('buttonNotif', 'Released');
    }
};
```

Code 2.3 ***Snippets for some pre-defined notifications in /thingy52/lib/thingy.js***

Thus, the specification of JSON parameters listed in *Table 2.2* can be obtained after investigating *Code 2.3*. *Table 2.4* lists an example of the JSON parameter for temperature object.

In the file **`index.js`**, the method `sendTemperature(temperature)` is constructed by another method `sendSensorData(data)`, which parses the temperature into data and returns the expected *JSON-Object*, as *Code 2.4 shows*.

```
function sendSensorData(data) {
        var jsonDate = (new Date()).toJSON();
        var jsonData = {
            timestamp: jsonDate,
            sensors: data
        };
        return rest.postJson(base_uri + '/' + this.id + '/sensors
/', jsonData);
    }
 module.sendTemperature = function(temperature) {
        return sendSensorData.call(this, {
            temperature: temperature
        });
    };
```

Code 2.4 ***Snippets for POST requests by thingy-gateway in index.js***

Thus, the returned *JSON-Object* can be seen as *Code 2.5*.

```
{
    "timestamp": "jsonDate",
    "sensors": {
        "temperature": temperature
    }
}
```

Code 2.5 ***JSON-Object returned by sendTemperature(temperature)***

The parameter `temperature` is returned as a float number with 2 decimals according to the snippets in **`/thingy52/lib/thingy.js,`** as *Code 2.6* shows.

```
Thingy.prototype.onTempNotif = function(data) {
    var integer = data.readInt8(0);
    var decimal = data.readUInt8(1);
    var temperature = integer + (decimal / 100);
    this.emit('temperatureNotif', temperature);
};
```

Code 2.6 ***Snippets for temperature notifications in /thingy52/lib/thingy.js***

By completing the investigation, an example of the JSON parameter for POST temperature information is shown in *Code 2.7*

```
{
    "timestamp": "2018-04-10T08:33:01.313Z",
    "sensors": {
        "temperature": 24.00
    }
}
```

Code 2.7 ***An example of the returned JSON-Object***

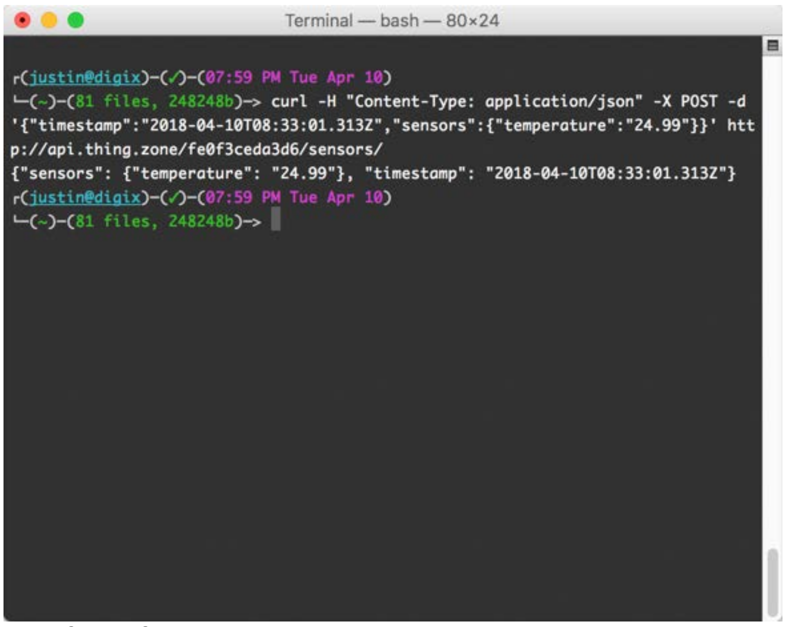
Using the given `{base_url}` and an arbitrary `{dev_id} (e.g. POST http://api.thing.zone/fe0f3ceda3d6/sensors/)`, it is not difficult to validate the specification of requests after studying. A simple but powerful toolkit to do this is cURL.

cURL can be used out-of-box for validating the request in the terminal for macOS users. After opening a terminal, a cURL command line can be typed as it showed in *Code 2.8*.

```
$ curl -H "Content-Type: application/json" -X POST -d
'{"timestamp":"2018-04-
10T08:33:01.313Z","sensors":{"temperature":"24.99"}}'
http://api.thing.zone/fe0f3ceda3d6/sensors/
```

Code 2.8 ***An example of a POST request by cURL command line instruction and its response***

When a positive response returned as the example shows, it means that the specification of the request is correct and that the request has been accepted.



```
┌(justin@digix)-(✓)-(07:59 PM Tue Apr 10)
└─(~)-(81 files, 248248b)-> curl -H "Content-Type: application/json" -X POST -d
'{"timestamp":"2018-04-10T08:33:01.313Z","sensors":{"temperature":"24.99"}}' htt
p://api.thing.zone/fe0f3ceda3d6/sensors/
{"sensors": {"temperature": "24.99"}, "timestamp": "2018-04-10T08:33:01.313Z"}
┌(justin@digix)-(✓)-(07:59 PM Tue Apr 10)
└─(~)-(81 files, 248248b)->
```

Figure 2.3 ***Example of RESTful-API request-response context via cURL command line in a terminal (macOS)***

The POST- or PUT-requests are listed in *Table 2.3*. The variable `jsonDate` refers to the String-type date, as a wildcard in JSON specifications, i.e. `2017-11-17T23:12:02.795Z`

Methods	**HTTP verbs + URI**	**JSON specification in example**

module.sendTemperature	POST {base_url}/{dev_id}/sensors/	{
		"timestamp": "jsonDate",
		"sensors": {
		"temperature": 24.00
		}
		}

module.sendPressure	POST {base_url}/{dev_id}/sensors/	{
		"timestamp": "jsonDate",
		"sensors": {
		"pressure": 958.24
		}

		}
module.sendHumidity	POST {base_url}/{dev_id}/sensors/	{ "timestamp": "jsonDate", "sensors": { "humidity": 3600 } }
module.sendColor	POST {base_url}/{dev_id}/sensors/	{ "timestamp": "jsonDate", "sensors": { "color": { "blue": 106, "red": 52, "green": 66, "clear": 62 } } }
module.sendGas	POST {base_url}/{dev_id}/sensors/	{ "timestamp": "jsonDate", "sensors": { "gas": { "tvoc": -99.99, "eco2": 99.99 } } }
module.setButton	PUT {base_url}/{dev_id}/sensors/button	{ "state": "Pressed" } or { "state": "Released" }

Table 2.3 ***List of JSON parameters of POST and PUT methods that are requested from the Thingy-gateway***

Since the *Thingy-gateway* is not a normal web client which usually requires a response for further procedure, there is no specified definition of the response, and no need to handle the occurred errors in the response. However, a response can and will be implemented later for respecting the *RESTful* constraints (request-response-pattern).

In addition to save data via POST requests, data can be fetched by the client via GET requests for most endpoints. *Table 2.4* lists 2 the GET methods.

Through a test server online (`base_url`: `http://api.thing.zone`), the responses of these 2 GET methods can be obtained. Later on, they will be constructed statically in the web service as well.

GET Requests with URI	**Response**
GET {base_url}/{dev_id}/setup/	{ "humidity": {"interval": 1000}, "temperature": {"interval": 1000}, "pressure": {"interval": 1000}, "color": {"interval": 1000}, "gas": {"mode": 1} }
GET {base_url}/{dev_id}/actuators/led	{ "delay": 1000, "color": 1, "intensity": 20 }

Table 2.4 ***List of JSON parameters of GET methods that are requested from the Thingy-gateway***

These responses contain the settings of the frequency of the data-updating as well as the configuration of LED breathing when connected. For more details about *Thingy:52 NodeJS Library*, please refer to the repository on *GitHub*[4].

At this point, all specifications for the obligated APIs between the web service and the thingy-gateway for requests and/or responses have been described. The next section covers building web server with *RESTful APIs*.

2.3 Web Server with Flask and Data Persistence

In the previous Sections 2.1 and 2.2, the *Flask micro-framework* has been briefly introduced and the specifications of the APIs, which are constrained by the *Thingy-gateway*, have been discussed and validated.

2.3.1 Overview of the Project Structure

The project is created under the root path **`/thingy-restful-api`**. The root folder contains the files and folders as *Figure 2.4* shows.

[4] *Nordic Thingy: 52 NodeJS Library* https://github.com/NordicPlayground/Nordic-Thingy52-Nodejs accessed on 07 December 2017

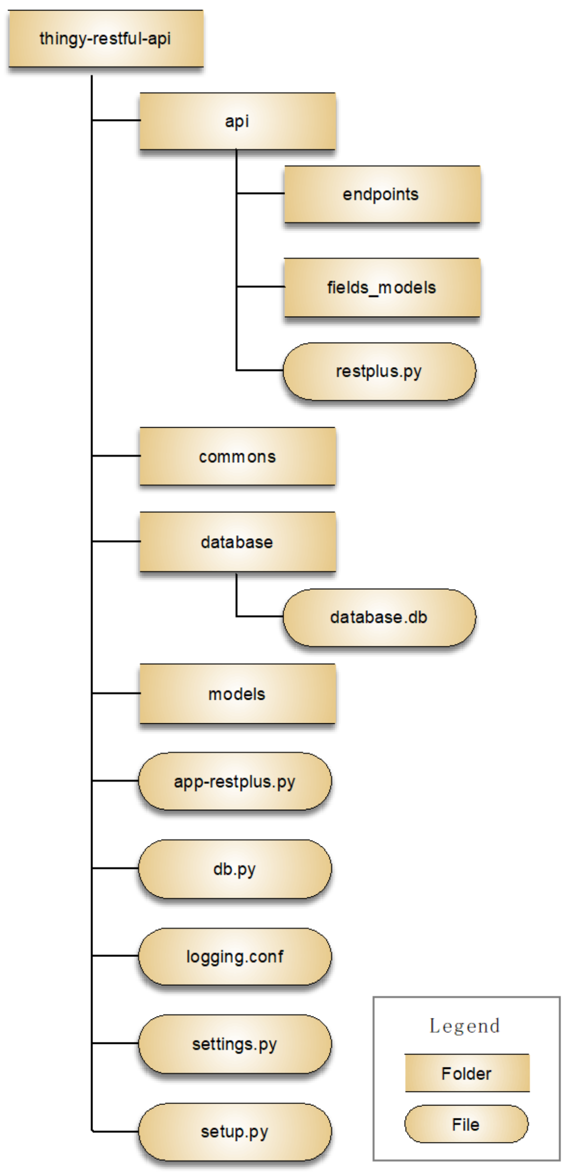


Figure 2.4 ***File architecture for RESTful web service upon Flask and flask-restplus***

The structure is designed based on a *layered-system-pattern*. For example, the **`api`** folder can be replaced by another, if another RESTful-library is chosen, i.e. *flask-restful*.

Under the root folder, the `/resource` folder holds all the API-layer implementation based on *flask-restful*, and `/app.py` is the main entry-point/ application file, which is not shown in *Figure 2.4* for ease of reading, since they are discontinued and deprecated by changing the library to *flask-restplus*. However, they all can be found in the source code of this project and run simply with *flask-restful*. Considering the featured support for swagger-UI from the flask-restplus library, the whole API layer needs to be rewritten. Thanks to the layered-system-pattern, it is much easier than it sounds. Even though it is called “rewrite”, all other modules/components are kept as before, such as the **`/models`** folder for object modeling, **`database.py`**, which is in charge of data persistence, etc. Actually, **`apps.py`** and **`apps-restplus.py`** can even run at the same time, as long as these two web server instances are not bound to a same port on an identical network-interface.

In the following sections, the implementation of the *RESTful* web service is explained in details, from one module to another.

2.3.2 Main Application and Settings

Main Application File of Web Service

The file **`app-restplus.py`** is the main entry point of *flask-restplus*. It is only the application file of the web service, not the whole project. Besides this application, there are another 2 independent ones in the **`/common`** folder, one is in charge of data analysis (see *Chapter 3*) and the other of machine learning procedure (see *Chapter 4*).

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# created on 7:05 PM 09.11.17, Yi Zhang

' main apps of thingy-restful-api '

__author__ = 'Yi Zhang'

from flask import Flask,Blueprint
import logging.config
from api.restplus import api
from db import db

import settings
from api.endpoints import temperature
from api.endpoints import gas
from api.endpoints import color
from api.endpoints import button
from api.endpoints import humidity
from api.endpoints import pressure
from api.endpoints import sensors
from api.endpoints import setup
from api.endpoints import led
from api.endpoints import presence

app = Flask(__name__)
logging.config.fileConfig('logging.conf')
log = logging.getLogger(__name__)


def configure_app(flask_app):
    flask_app.config['SERVER_NAME'] = settings.FLASK_SERVER_NAM
E
    flask_app.config['SQLALCHEMY_DATABASE_URI'] = settings.SQLALC
HEMY_DATABASE_URI
    flask_app.config['SQLALCHEMY_TRACK_MODIFICATIONS'] = settings.
SQLALCHEMY_TRACK_MODIFICATIONS
    flask_app.config['SWAGGER_UI_DOC_EXPANSION'] = settings.RESTP
LUS_SWAGGER_UI_DOC_EXPANSION
    flask_app.config['RESTPLUS_VALIDATE'] = settings.RESTPLUS_VA
LIDATE
    flask_app.config['RESTPLUS_MASK_SWAGGER'] = settings.RESTPLUS
_MASK_SWAGGER
    flask_app.config['ERROR_404_HELP'] = settings.RESTPLUS_ERROR
_404_HELP
```

```
39
40  def initialize_app(flask_app):
41      configure_app(flask_app)
42      # add swagger UI blueprint
43      #blueprint = Blueprint('api',__name__)
44      #flask_app.register_blueprint(blueprint)
45
46      api.init_app(flask_app)
47      db.init_app(flask_app)
48
49  def main():
50      initialize_app(app)
51      # init db if needed
52      @app.before_first_request
53      def create_tables():
54          db.create_all()
55      log.info('>>>>> Starting development server at http://{}/ <<<<<'.f
    ormat(app.config['SERVER_NAME']))
56
57      app.run(host=settings.FLASK_SERVER_HOST_NAME,port=setting
    s.FLASK_SERVER_PORT, debug=settings.FLASK_DEBUG)
58
59  if __name__ == "__main__":
60      print('start')
61      main()
```

Code 2.9 ***/thingy-restful-api/app-restplus.py***

The purpose of *Code 2.9* is to create the web server, which can listen to the given port and address for the responses to the received requests. In line 26, the `flask_app` is initialized and constructed. Then it is configured in method `configure_app(flask_app)`. Afterward, two flask extensions – *flask-restplus* and *flask-sqlalchemy* are set to the constructed `flask_app`.

Flask uses decorators to approach the function injection, such as simplifying the event-driven control. The method decorated by `@app.before_first_request` will be called before the first request to this instance of the application. By calling `create_tables()`, which results in calling `db.create_all()`, the database file will be created, if it does not exist. Otherwise, nothing would be touched.

Settings

In *Code 2.9*, it is shown, that

- *Code 2.10* shows file **`logging-conf`**, which contains the settings for logging, of which the level, handler, and formatter can be configured directly. In addition, multiple loggers can be configured in this file, too. This configuration file may be found in most flask apps and is not project-specific.

```
[loggers]
keys=root,rest_api_demo

[handlers]
keys=console

[formatters]
keys=simple

[logger_root]
level=DEBUG
handlers=console

[logger_rest_api_demo]
level=DEBUG
handlers=console
qualname=rest_api_demo
propagate=0

[handler_console]
class=StreamHandler
level=DEBUG
formatter=simple
args=(sys.stdout,)

[formatter_simple]
format=%(asctime)s - %(name)s - %(levelname)s - %(message)s
```

Code 2.10 *⁄thingy-restful-api/logging.conf*

- In file **`db.py`** (*Code 2.11*), the database utility interface is set. Commonly, the *flask-sqlalchemy*, is imported here. It is an extension for *Flask*, which adds support for *SQLAlchemy library*. *SQLAlchemy* enables the *Object Relational Mapper (ORM)*, in which the objects model and database schema can be developed in a cleanly decoupled way from the beginning.

```
from flask_sqlalchemy import SQLAlchemy
db = SQLAlchemy()
```

Code 2.11 */thingy-restful-api/db.py*

- The file **`setting.py`** (*Code 2.12*) is in charge of all environment parameters for *Flask*, *flask-restplus* and all other customer specified global parameters. Since all parameters are self-descriptive, few comments are necessary.

```
# Flask settings
FLASK_SERVER_HOST_NAME = '192.168.1.236'
FLASK_SERVER_PORT = 7777
FLASK_SERVER_NAME =
''.join([FLASK_SERVER_HOST_NAME,':',str(FLASK_SERVER_PORT)])
# Do not use debug mode in production
FLASK_DEBUG = True

# Flask-Restplus settings
RESTPLUS_SWAGGER_UI_DOC_EXPANSION = 'list'
RESTPLUS_VALIDATE = True
RESTPLUS_MASK_SWAGGER = False
```

```
RESTPLUS_ERROR_404_HELP = False

# SQLAlchemy settings
SQLALCHEMY_DATABASE_URI = 'sqlite:///./database/data.db'
SQLALCHEMY_TRACK_MODIFICATIONS = False

# GLOBAL ENV DEFAULT
# presence: -1: init state/unknown, 0: left, 1: entered
THINGY_PRESENCE = -1
```

Code 2.12 ***/thingy-restful-api/settings.py***

2.3.3 Database Schema for Sensors and Presence.

Database Schema for Sensors

Since the gateway can be connected with multiple devices, each having its own measured data, it is rational to create a one-to-many relationship between the device and measured data, which includes gas, pressures, temperatures, humidity, the button's state and LED-colors.

The schema of the device table can be then described in *Table 2.5*.

Field	Type	Comment
id	Integer	primary key, auto-increment
device_name	Varchar	name of device, for example 'eb108ef0e0c3'
created	Datetime	created time by the server
updated	Datetime	updated time by the server

Table 2.5 ***Database schema for devices***

According to the one-to-many relationship between the device and measured temperature, the schema of temperature can be sketched as in *Table 2.6*.

Field	Type	Comment
id	integer	primary key, auto-increment
device_id	integer	Foreign key to id in table device
temperature	float	measured temperature
timestamp	varchar	timestamp from the gateway in form of a string
created	datetime	created time from the server
updated	datetime	updated time from the server

Table 2.6 ***Database schema for temperature***

Given the specification of APIs, which are listed in *Table 2.3*, all other schemas for the rest data can be designed in the same way.

Database Schema for Presences

In addition, it is necessary to persist presence-information while synchronizing the measured data from the gateway. Thus, a supervised training set can be split from these presence-tagged entities, and so for the test set in a purpose of validation, too.

The state of presence is defined by a period from the time "entered" to the one "left". Hence, the schema of presence can be designed as the *Table 2.7* listed.

Field	Type	Comment
id	integer	primary key, auto-increment
entered_on	varchar	entered time from the gateway in form of a string
left_on	varchar	left time from the gateway in form of a string
created	varchar	created time from the server
updated	datetime	updated time from the server

Table 2.7 ***Database schema for presences***

Finally, *Figure 2.5* shows the UML-diagram for the completed database schema of this project.

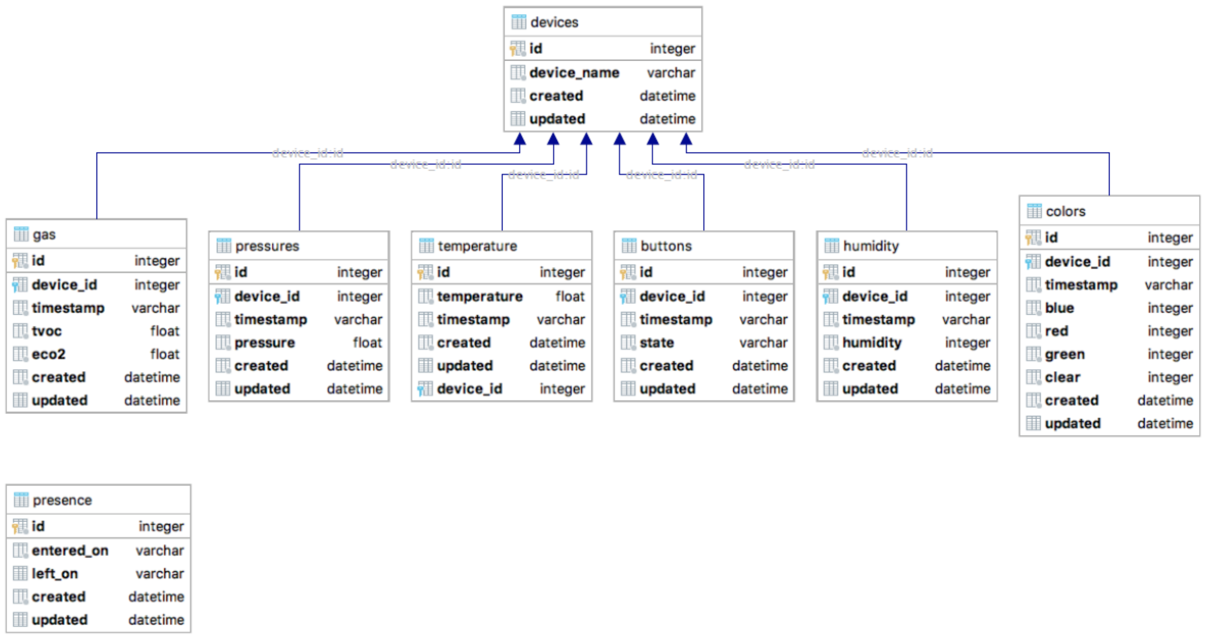


Figure 2.5 ***UML-diagram of the database schema***

2.3.4 Objects' Modelling for Data of Sensors and Presence.

Generally, it could take a lot of time to repeat a series of SQL sentences only for creating tables or/and constraints in a database. Later on, the developer has to spend more time on the hybrid-style syntax for the SQL-operations intensively and carefully, to avoid some small bugs, which may but cause some fatal security issues, such as a vulnerability in SQL-injection-attack. Furthermore, it could be slightly different from one SQL-dialect to another, because of the different database services. A couple of small faults can lead a sleepless night for painful debugging. Fortunately, *SQLAlchemy* offers a set of powerful toolkits, which provides the data-mapper-patterns, to enable object modeling at the beginning.

In *Table 2.5* and *2.6*, the database schema of device and temperature has been defined and specified. Thanks to the ORM, they can be defined as a familiar usual class to all OOP-programmers and are ready to be instantiated.

File **`device.py`** (*Code 2.13*), constructs the `DeviceModel`.

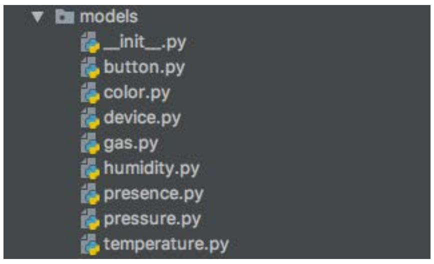
```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# created on 11:20 AM 04.11.17, by Yi Zhang

' device model of thingy-restful-api '

__author__ = 'Yi Zhang'

from db import db
from datetime import datetime
from sqlalchemy import desc,asc
from models.temperature import TemperatureModel
from models.gas import GasModel
from models.humidity import HumidityModel
from models.pressure import PressureModel
from models.color import ColorModel
from models.button import ButtonModel

class DeviceModel(db.Model):

    # Table name
    __tablename__ = 'devices'

    # Definition of fields
    id = db.Column(db.Integer, primary_key=True)
    device_name = db.Column(db.String, nullable=False)
    created = db.Column(db.DateTime, nullable=False, default=datetime.utcnow)
    updated = db.Column(db.DateTime, onupdate=datetime.utcnow, default=datetime.utcnow)

    # one-to-many relationship
    temperatureList = db.relationship(TemperatureModel,lazy='dynamic')
    gasList = db.relationship(GasModel, lazy='dynamic')
    buttonList = db.relationship(ButtonModel, lazy='dynamic')
    pressureList = db.relationship(PressureModel, lazy='dynamic')
    humidityList = db.relationship(HumidityModel, lazy='dynamic')
    colorList = db.relationship(ColorModel, lazy='dynamic')

    def __init__(self,device_name,_id=None):
        self.device_name = device_name
        self.id = _id

    def __repr__(self):
        return "<Device('%s')>" % (self.json())

    def json(self):
        return {'id':self.id,'device_name':self.device_name}

    @classmethod
    def find_by_name(cls,device_name)->'DeviceModel':
        return cls.query.filter_by(device_name=device_name).first()

    @classmethod
    def find_by_id(cls,_id)->'DeviceModel':
        return cls.query.filter_by(id=_id).first()

    @classmethod
    def find_all_devices(cls)->list('DeviceModel'):
        return cls.query.order_by(desc('created')).all()

    def save_to_db(self):
        db.session.add(self)
        db.session.commit()
```

Code 2.13 ***DeviceModel class in device.py***

In line 22, the *table name* for the model is determined. From line 24 to line 28, the *fields* are declared as specified in the schema. The essential codes in line 31 realized the declaration of the one-to-many relationship between models of devices and of temperature. Since it is a one-to-many relationship, the related multiple temperatures, are wrapped as a collection of objects, such as a list. The self-descriptive decorator `@classmethod` claims the static class method for retrieving expected device or collection of devices, which functions as same as the SELECT-queries do, but much more simple and elegant. The `save_to_db()` method, similar to the INSERT-query, but is wrapped in a more robust transaction context, which ends-up by either a successful return or a handleable exception via `db.session.commit()`. Thus, all necessary properties and methods for device model are completely defined.

Nevertheless, the model class is implemented within about 40 lines and there exists no obscure hybrid-syntax such as “SELECT-query” or “INSERT-query” in the snippet.

Similar to the device model, all other models are implemented in the same way under the folder **`/models`**.



```
▼ models
    __init__.py
    button.py
    color.py
    device.py
    gas.py
    humidity.py
    presence.py
    pressure.py
    temperature.py
```

Figure 2.6 ***Files for database models under the models folder***

Thus, all models have been generated. In the next step, web resources will be implemented as accessible endpoints. Furthermore, auto-documented and interactive graphics interface upon *field model* will also be introduced, which is supported by *Swagger-UI*.

2.4 Endpoint and Field Object

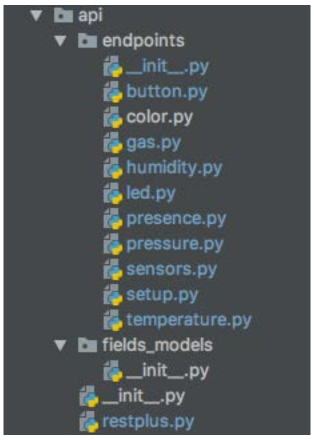
Thanks to the extension *flask_restplus*, it is simple and swift to generate the endpoints of expected web resources with the different *Http-verbs* (GET / POST /PUT /PATCH /DELETE) for various operations, i.e. *CRUD-operation* (create /retrieve /update /delete).

As introduced in the previous section, the *RESTful API* should be implemented in a layered system according to the *RESTful*-constraints. Rather than the native flask framework, the *flask-restplus* extension has extended more features, such as

- Response marshaling
- Request parsing
- Error handling
- Field masks
- Swagger-UI documentation.

All these new features brought by *flask-restplus* will be introduced and explained in the following case study. It is always a good start point from the brief structure of *resources* and *fields*, based on *flask-restplus*.

From the view of the hierarchy of project, all related files and folders are located under the folder API:



```
▼ api
  ▼ endpoints
      __init__.py
      button.py
      color.py
      gas.py
      humidity.py
      led.py
      presence.py
      pressure.py
      sensors.py
      setup.py
      temperature.py
  ▼ fields_models
      __init__.py
    __init__.py
    restplus.py
```

Figure 2.7 ***Files structure for resources(endpoints) and fields under the api folder***

One file named **`restplus.py`** is located directly under the folder **`api`**.

```
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# created on 7:22 PM 09.11.17, by Yi Zhang

' Flask-restplus settings '

__author__ = 'Yi Zhang'

import logging
import traceback
import settings

from flask_restplus import Api
from sqlalchemy.orm.exc import NoResultFound

log = logging.getLogger(__name__)

api = Api(version='1.0', title='Thingy restful API',
          description='A simple Thingy restful API powered by Flask-RESTplus')

```

```

@api.errorhandler
def default_error_handler(e):
    message = 'An unhandled exception occurred.'
    log.exception(message)

    if not settings.FLASK_DEBUG:
        return {'message': message}, 500


@api.errorhandler(NoResultFound)
def database_not_found_error_handler(e):
    log.warning(traceback.format_exc())
return {'message': 'A database result was required but none was found.'}
, 404
```

Code 2.14 ***Settings for flask_restplus in /api/restplus.py***

Code 2.14 contains general settings for this extension, the logger, the initializing of the *Api object* and the error handling.

All the endpoints, through which the web resource can be accessed, are located under the subfolder **`api/endpoints/`**. They are all defined as different *resource objects*, which are usually named upon their endpoints and returned type. It is very important to differentiate these *resources objects* from the *data models*, which are defined and specified by the database schema. When a *resource model* has the same name as a *database model*, they are but totally different in definition, even though they work mostly together very closely. The data model contains all information about the properties and relationships of it to other database models, which are determined by the database schemas. The resource object contains the information about the resource routing, request parsing, response marshaling and etc.

In the Layered System

The data model defines WHAT (exactly the objects) the server has and the resource object defines HOW (the way and the form) represent/expose them as web resources.

The word “represent” reminds us of a familiar concept in a web application — View. By investigating the source code of resource-object in *flask-restplus*, it will be directly found, that the class `Resource` is a subclass inherited from `flask.views.MethodView`, which can instantiate a `View`, after handling a request based on the different HTTP-methods (GET/POST/PUT/…).

Before the response is sent, the response is marshaled by the *fields model*. Fields models act as a response filter and formatter to return the proper response. Furthermore, it is very useful to decorate the resource endpoint with `@api.expect(some_expected_fields)` to enable the automation of the swagger documentation. All *Fields objects* are declared in file **`__init__.py`** in the subfolder **`api/fields_models/`**.

The field models are not defined and named in different files like other models, such as data models or resource models. Because all these fields models are just dictionaries. They can be defined as a nesting class, which usually needs to be referenced again within another nesting class. Thus, it will be more convenient to put them together in one file, instead of using multiple small files with many of `import` sentences.

By going through the *sensor resource*, *temperature resource*, and related *fields objects*, it is easy to learn about the web *RESTful APIs* with *flask-restplus*.

2.4.1 Resource Object and Endpoint

```python
#!/usr/bin/env python
# -*- coding: utf-8 -*-
# created on 7:47 PM 11.11.17, by Yi Zhang

' sensor resource of thingy-restful-api '

__author__ = 'Yi Zhang'

from flask import request
from flask_restplus import Resource,reqparse
from models.temperature import TemperatureModel
from models.gas import GasModel
from models.device import DeviceModel
from models.humidity import HumidityModel
from models.pressure import PressureModel
from models.color import ColorModel
from api.restplus import api
import api.fields_models as fm


@api.route('/<device_name>/sensors/',endpoint='sensors')
class Sensors(Resource):

    def temperature(self,temperature,rec_timestamp,device):
        _temperature = TemperatureModel(temperature=float(temperature, timestamp=rec_timestamp)
        _temperature.save_to_db(device)
        _temperature = TemperatureModel.find_last_one_by_device_id(device.id)
        return _temperature.json(),201

    def humidity(self,humidity,rec_timestamp,device):
        _humidity = HumidityModel(humidity=int(humidity, timestamp=rec_timestamp)
        _humidity.save_to_db(device)
        _humidity = HumidityModel.find_last_one_by_device_id(device.id)
        return _humidity.json(),201

    def pressure(self,pressure,rec_timestamp,device):
        _pressure = PressureModel(pressure=float(pressure), timestamp=rec_timestamp)
        _pressure.save_to_db(device)
        _pressure = PressureModel.find_last_one_by_device_id(device.id)
        return _pressure.json(),201

    def gas(self,data,rec_timestamp,device):
        _gas = GasModel(tvoc=data['tvoc'], eco2=data['eco2'],timestamp=rec_timestamp)
        _gas.save_to_db(device)
        _gas = GasModel.find_last_one_by_device_id(device.id)
        return _gas.json(),201


    def color(self,data,rec_timestamp,device):
```

```
50          _color = ColorModel(blue=data['blue'],
    red=data['red'],green=data['green'],clear=data['clear'],timestam
    p=rec_timestamp)
51          _color.save_to_db(device)
52          _color = ColorModel.find_last_one_by_device_id(device
    .id)
53          return  _color.json(), 201
54
55
56      # parser validation rules
57      parser = reqparse.RequestParser()
58      parser.add_argument('timestamp',
59                          type=str,
60                          required=True,
61                          help="This field cannot be blank!")
62      parser.add_argument('sensors',
63                          type=str,
64                          required=True,
65                          help="This field cannot be blank!")
66
67      @api.expect(fm.temperature_sensors_post)
68      @api.expect(fm.humidity_sensors_post)
69      @api.expect(fm.pressure_sensors_post)
70      @api.expect(fm.gas_sensors_post)
71      @api.expect(fm.color_sensors_post)
72      def post(self,device_name):
73          sensors_options = {
74              'temperature': self.temperature,
75              'humidity': self.humidity,
76              'gas': self.gas,
77              'color': self.color,
78              'pressure': self.pressure
79          }
80          jsonData = request.get_json()
81          rec_timestamp = jsonData['timestamp']          # str
82          rec_sensor = [key for key in jsonData['sensors'].keys()
    ][0]        # dict
83          rec_sensor_data = jsonData['sensors'][rec_sensor]
84          device = DeviceModel.find_by_name(device_name)
85          if not device:
86              device = DeviceModel(device_name)
87              device.save_to_db()
88              device = DeviceModel.find_by_name(device_name)
89          try:
90              return sensors_options[rec_sensor](rec_sensor_data
    ,rec_timestamp=rec_timestamp,device=device)
91          except KeyError:
92              return {'message':'invalid sensor name'},400
```

Code 2.15 ***/thingy-restful-api/sensors.py***

In the file **`api/endpoints/sensors.py`** the sensor resource is defined as a unified endpoint dependent on different payloads. By parsing and validating the payload, the sensor resource returns the expected JSON-Object or throws an error message.

In line 21 of *Code 2.15*, the endpoint of sensors is defined. Lines 24 to 53 show the definition of the database persistence and call-back of response for different sensor types (i.e. temperatures/humidity/pressures/ and etc.). Lines 57 to 65 demonstrate located validation rules for the request's payload. The next four lines after line 67 describe the expected request payload, which will be shown in the swagger automated documentation. The post method defines the logic in the POST-request. There is no `switch/case` syntax in python, instead some tricks will be used as an equivalent code block similar to `switch/case` by a combination of `try/except` blocks and function dictionaries, shown in lines 73 to 92. The main idea here is to the return the expected sensor data based on the parsed-in sensor type, which can be fetched by the serialized payload in JSON via `request.get_json()`.

Code 2.16 simply implements temperature resource with a GET-method.

```
1     #!/usr/bin/env python
2     # -*- coding: utf-8 -*-
3     # created on 11:45 PM 10.11.17, by Yi Zhang
4
5     ' temperature resource of thingy-restful-api '
6
7     __author__ = 'Yi Zhang'
8
9     import logging
10
11    from flask import request
12    from flask_restplus import Resource
13    from models.temperature import TemperatureModel
14    from models.device import DeviceModel
15    from api.restplus import api
16    import api.fields_models as fm
17
18    @api.route('/<device_name>/sensors/temperature',endpoint='temperature
      ')
19    @api.param('device_name', 'device no.') # the description of para
      in Swagger-UI
20    class Temperature(Resource):
21
22        @api.marshal_list_with(fm.temperature_get)
23        def get(self,device_name):
24            _device = DeviceModel.find_by_name(device_name)
25            if _device:
26                _temperature = TemperatureModel.find_last_one_by_
      device_id(_device.id)
27                if _temperature:
28                    return _temperature.json(), 200
29                return {'message': 'temperature data of device {} not foun
      d'.format(device_name)}
30            return {'message': 'device {} not found'.format(device_name
      )}
```

Code 2.16 ***/thingy-restful-api/temperature.py***

In line 18, the URI of the API is defined. Instead of a `post()` method, a `get()` method needs to be overwritten in the class for implementing the GET-method. The code between lines 23 and 29 realize this GET-method with 2 different exception handlings. When an unknown `device_name` is input or if no data is stored for a certain requested device, a response with a proper error message will be returned to the client.

A new decorator is introduced in line 22. This decorator handles response marshaling within fields model, in which it works like an internal filter layer inside the resource object, to avoid exposing the unexpected internal data structure.

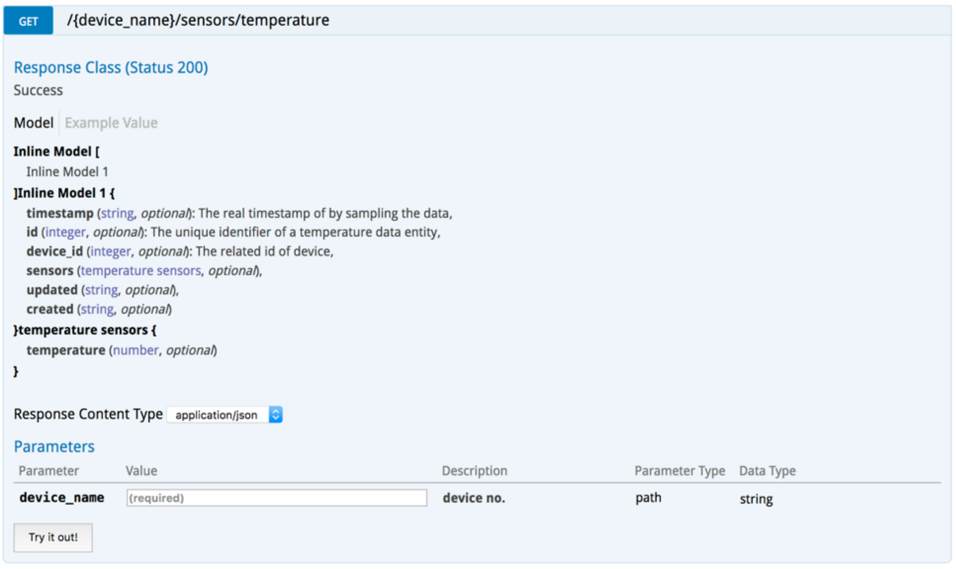
2.4.2 Field Objects and Swagger-UI

```
common_fields = {
    'timestamp': fields.String(readOnly = True, description = 'The real timestamp of by sampling the data'),
    'created': fields.DateTime(readOnly = True),
    'updated': fields.DateTime(readOnly = True),
    'device_id': fields.Integer(readlOnly = True, description = 'The related id of device')
}
temperature_sensors = api.model('temperature sensors', {
    'temperature': fields.Float(attribute = 'temperature')
})
temperature_get_fields = {
    'id': fields.Integer(readOnly = True, description = 'The unique identifier of a temperature data entity'),
    'sensors': fields.Nested(temperature_sensors),
    **common_fields
}
temperature_get = api.model('Temperature', temperature_get_fields)
```

Code 2.17 ***Snippets of field models for the temperature resource in api/fields_models/__init__.py***

The correspondent nested fields object for the temperature resource in the GET-method, which is defined as `temperature_get` in the *Code 2.17*, is in charge of documentation automation based on Swagger-UI. A field object can be regarded as a “response formatter + response filter”. Furthermore, it can contain descriptions for the response payload in details, which is represented automatically in API-docs for the client developer.

Figure 2.8 shows an automatically generated online API-documentation for the temperature resource that is accessed with GET-method. It is generated from *Code 2.17*, which contains all necessary description elements. A crystal-clear, professional, and interactive API-documentation is prepared for the front-end developers. In addition, the documentation and the marshaling are dynamically bound to the field model. If the field model is changed, the correspondent content in the documentation and the specification in the real response will be modified automatically.



GET /{device_name}/sensors/temperature

Response Class (Status 200)
Success

Model | Example Value

```
Inline Model [
  Inline Model 1
]Inline Model 1 {
  timestamp (string, optional): The real timestamp of by sampling the data,
  id (integer, optional): The unique identifier of a temperature data entity,
  device_id (integer, optional): The related id of device,
  sensors (temperature sensors, optional),
  updated (string, optional),
  created (string, optional)
}temperature sensors {
  temperature (number, optional)
}
```

Response Content Type: application/json

Parameters

Parameter	Value	Description	Parameter Type	Data Type
device_name	(required)	device no.	path	string

Figure 2.8 ***Screenshot of an API-doc supported by Swagger UI***

After all the thingy-related resources are implemented, the presence resource must be created from the sketch, because this resource is project-specified.

It is defined that the presence resource enables the web server:

1. to know the presence state per request
2. to create or update the presence in the database
3. to respond if the operations above have succeeded

Thus, the HTTP-method is determined as PUT for creating a new presence or updating the presence. The endpoint for updating the presence state is `/presence/enter` for the state “entered” and `/presence/leave`” for “left”. In addition, an endpoint `/presence` with the GET-method is designed to fetch the last presence state. All endpoints for presence resource are defined in *Table 2.6*

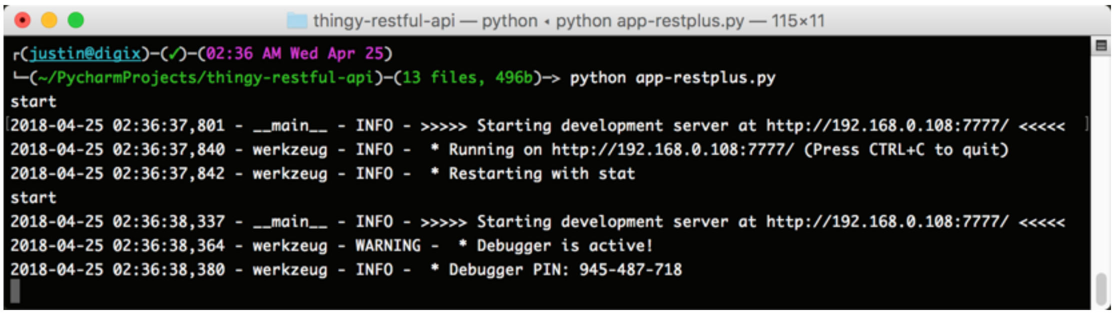
Http-methods	Endpoints
PUT	{base_url}/presence/enter
PUT	{base_url}/presence/leave
GET	{base_url}/presence

Table 2.8 ***Endpoints URI and correspondent Http-methods for presence resource***

After all resources and the related fields objects have been implemented, the web server is successfully built. If all settings are properly configured, the web service application can be launched by the following command line instruction under the root folder of the project:

```
$ python app-restplus.py
```

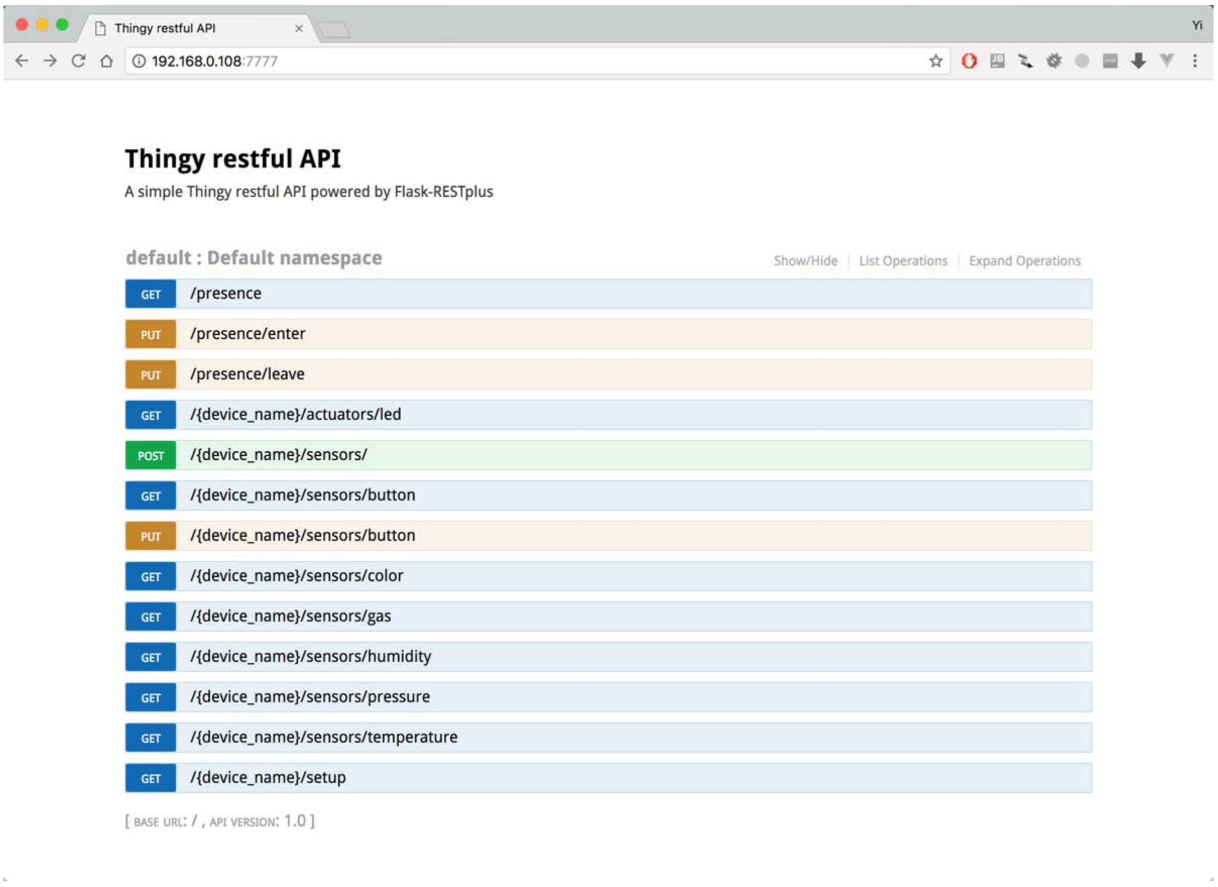
After that, some similar information as in *Figure 2.9* will be shown in the terminal/console.



```
thingy-restful-api — python ◂ python app-restplus.py — 115×11
┌(justin@digix)-(✓)-(02:36 AM Wed Apr 25)
└─(~/PycharmProjects/thingy-restful-api)-(13 files, 496b)-> python app-restplus.py
start
2018-04-25 02:36:37,801 - __main__ - INFO - >>>>> Starting development server at http://192.168.0.108:7777/ <<<<<
2018-04-25 02:36:37,840 - werkzeug - INFO -  * Running on http://192.168.0.108:7777/ (Press CTRL+C to quit)
2018-04-25 02:36:37,842 - werkzeug - INFO -  * Restarting with stat
start
2018-04-25 02:36:38,337 - __main__ - INFO - >>>>> Starting development server at http://192.168.0.108:7777/ <<<<<
2018-04-25 02:36:38,364 - werkzeug - WARNING -  * Debugger is active!
2018-04-25 02:36:38,380 - werkzeug - INFO -  * Debugger PIN: 945-487-718
```

Figure 2.9 ***Screenshot of the web service launch***

After opening the web browser, the API docs can be represented by typing the URL with the port number, i.e. http://192.168.0.108:7777 as *Figure 2.10* shows.



```
Thingy restful API
192.168.0.108:7777

Thingy restful API
A simple Thingy restful API powered by Flask-RESTplus

default : Default namespace          Show/Hide | List Operations | Expand Operations
GET   /presence
PUT   /presence/enter
PUT   /presence/leave
GET   /{device_name}/actuators/led
POST  /{device_name}/sensors/
GET   /{device_name}/sensors/button
PUT   /{device_name}/sensors/button
GET   /{device_name}/sensors/color
GET   /{device_name}/sensors/gas
GET   /{device_name}/sensors/humidity
GET   /{device_name}/sensors/pressure
GET   /{device_name}/sensors/temperature
GET   /{device_name}/setup

[ BASE URL: / , API VERSION: 1.0 ]
```

Figure 2.10 ***Screenshot of the homepage of automated APIs-docs***

Thus, the web server is in running and everything is set up to collect the temperature, the concentration of CO_2 (value of *eCO2* in fetched gas information. All *eCO2* in the following context refer to *the concentration of* CO_2) and the presence state.

3 Measurement and Data Analysis

3.1 Modelling Measurement

The purpose of this project is to find a functional model to infer the presence of human beings upon the measured environmental data. It can be described as a mathematical function as the following mapping relationship:

$$f(T) \rightarrow \{presence \mid \text{"entered"}, \text{"}left\text{"}\},$$

in which *T* can be a set of environmental data or its derivative set based on sliding-window and *f* the correspondent mapping function. Mapping function *f* shall be found by the machine-learning module in the later chapter.

As it has been seen in the previous chapter, the **`Thiny:52`** is equipped with various sensors, so that it can be used to measure different environmental data. In the previous chapter, a web server has been implemented, which is able to collect all transferred data from *Thingy-gateway* via the *RESTful API*. It is well known that the presence of human beings can mainly affect the *Emission of CO_2 (eCO2)* and temperature in the environment for the biological breathing and heat radiation. Usually, the *eCO2* and the temperature will increase, when one entered, and decrease when one left. These 2 dimensions are regarded as the input parameters in the modeling. Therefore, the research will be firstly focused on these 2 dimensions and their development.

The state of presence is the third dimension in the modeling, regarded as the output parameter of the mapping function.

3.2 Calibration before Collecting Data

The **`Thingy:52`** is equipped with a CCS811 sensor for measuring *eCO2* and *Volatile Organic Compound (VOC)*, such as formaldehyde. CO_2 measurements values range from 400 to 8192 ppm (parts-per-million). It is very important to calibrate the gas sensor unit (burn-in) according to the manual. Otherwise, the performance in terms of resistance levels and sensitivities will change dramatically during data collection. For performing the burn-in/calibration, it is strongly recommended to run the **`Thingy:52`** in the working environment for 48 hours[5].

3.3 Optimization of Measurement

The first try-out may fail or not yield expected values. This can be improved by solving some found issues. By refining the modeling of measurement, a progressive result can be analyzed once more and it can approach the proper model even closer. The next subsections cover the issues of measuring data, and their solutions for improvement.

3.3.1 Proper Conditions for Measurement

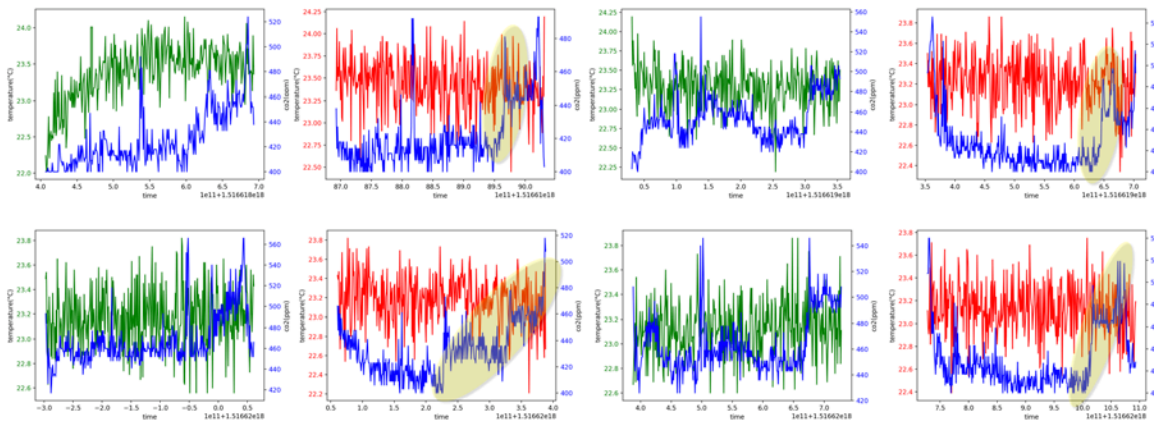
Different environmental conditions, such as ventilation, heating/radiator or air condition, can change the results significantly.

The interval between the presence-state is but the most important key value to choose. If it is too short, the influence of the presence-change is too small to be observed by the measurement, if it is too long, the environment will then reach a new thermal equilibrium and dynamic balance for *eCO2*, of which the measured data changes no longer. That could confuse the later machine learning and get a deviated result. After some tests, the proper interval is determined as ca.15-20 minutes.

During the research, it has been found that the collected data without proper conditions can influence the data analysis dramatically.

3.3.2 Measurement before optimization

Below is a series of plotting for a measurement in one session with neither ventilation nor calibration for *eCO2*. The red/green plotting shows the temperature development. The red curve shows when a state of presence is tagged as `left`, green when a state as `entered`. The blue plotting is the development of *eCO2*.



[5] Source: page 8, https://cdn.sparkfun.com/assets/learn_tutorials/1/4/3/CCS811_Datasheet-DS000459.pdf , accessed on 04. April 2018

Figure 3.1 ***Data plotting for temperature and eCO2 upon time series (segmented by the presence, before pre-processing)***

Even though the data swings heavily, some abnormal curves can be discovered in *left-segmentation*, which are highlighted in yellow. These curve areas tell an abnormal increasing trend for *eCO2* even though people left, which should be due to bad ventilation and/or wrong drifting caused by the bad performance without calibration.

Segmented by Session vs. Segmented by Presence

It is found that the visualization segmented by presence is not ideal to represent a continuous development during the changes of state. Besides that, heavily swung dataset hinders observation. Therefore, the relevant file **`/commons/old-data-analyse.py`** is deprecated. A new one **`/commons/data-analyse.py`** is re-written and improved, resulting in a continuous and smoothed plotting (with segmentation by session, after pre-processing). *Figure 3.2* shows a new data-visualization after pre-processing (smoothing) upon the exact same dataset as one applied in *Figure 3.1*. In the next section all around data analysis will be explained more in detailed.

In *Figure 3.2*, presence is displayed as a square wave, in which the high level means state `entered` and low-level state `left`. By checking the curve of *eCO2*, an abnormal result can be found directly, which is highlighted in yellow, due to the unusual increment after people left.

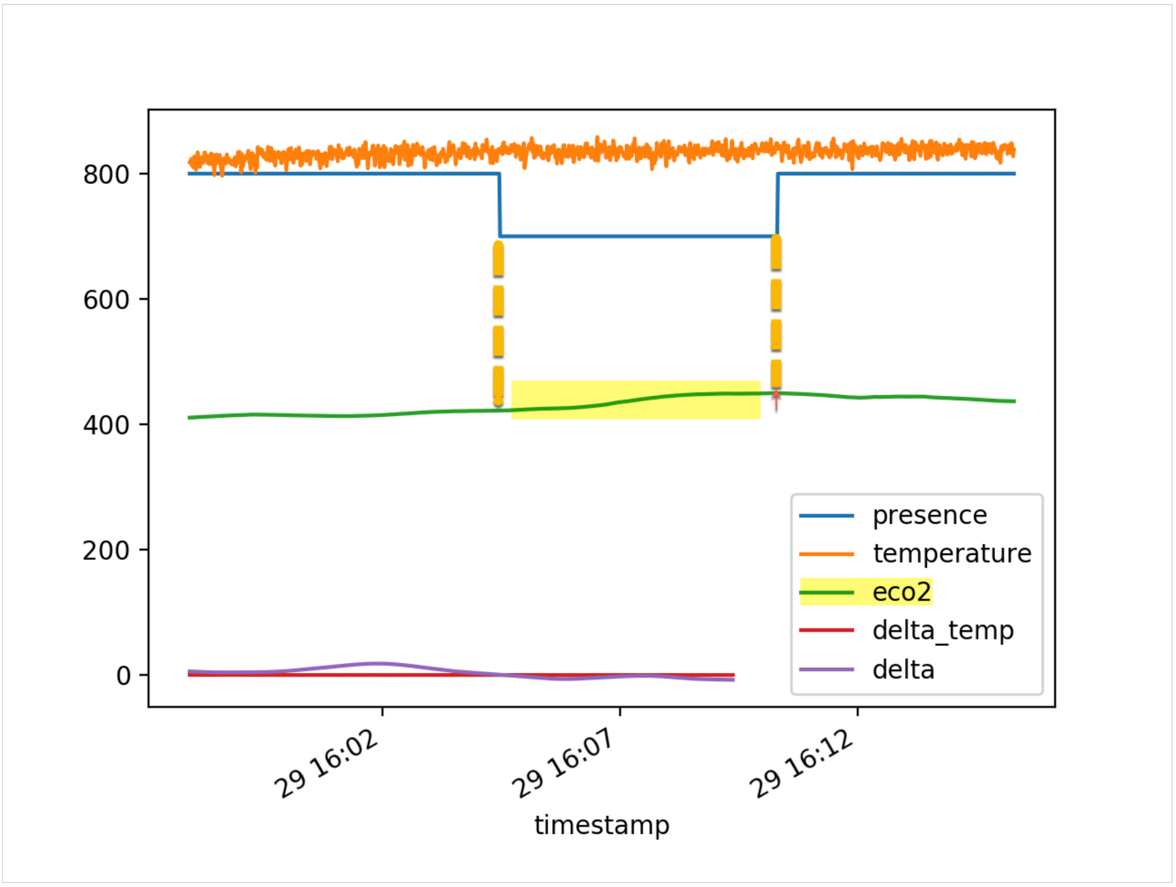


Figure 3.2 ***New continuous data plotting (segmented by session, after pre-processing)***

3.3.3 Measurement after optimization

After research, it is found, that the abnormal result probably happens due to:

- Non-compliant experimental procedure without *eCO2* burn-in / calibration
- Improper room environments, such as a room with bad ventilation or thermos-insensitivity

To ensure a better result, a thermal sensitive environment must be chosen, in order to avoid such an abnormal phenomenon, and a 48h calibration must also be performed. *Figure 3.3* shows a much better result from the measurement with the optimized environmental condition and after calibration. Now it looks much closer as expected, though it is not smoothed.

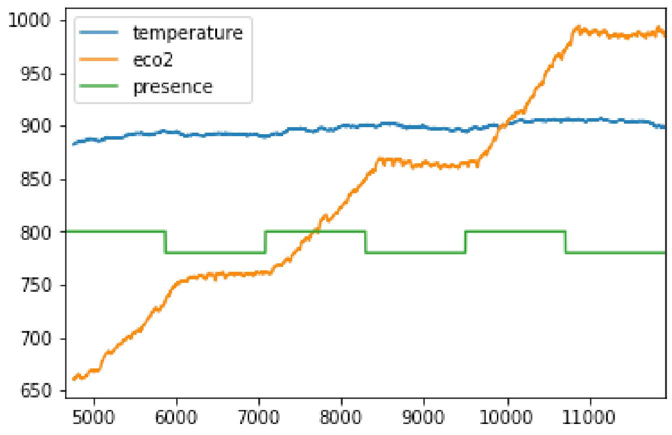


Figure 3.3 ***Plots for data from optimized environments and after calibration***

Finally, the importance of optimizing the environmental conditions by measurement was confirmed after multiple try-outs, because all improper measurement failed in data analysis. For a better explanation with an intuitive view, some plots i.e. *Figure3.2 and Figure 3.3* have to be introduced in advance. In the next section, the implementation of data visualization and pre-processing will be introduced.

3.4 Data Analysis, Visualization, and Pre-processing

When combining Python, NumPy, Pandas and Matplotlib, we gain a powerful data analysis toolkit. In this project, all the following libs are required for data analysis.

- **NumPy** is the fundamental package for scientific computing, which supports a powerful N-dimensional array object.
- **Pandas** is the data analysis library in Python, with which the data can be manipulated as a `DataFrame` object, similar to a programmable worksheet in excel. Furthermore, it supports featured rolling-window and "group-by engine", which allows split-apply-combine operations on data sets.
- **Matplotlib** is a Python 2D plotting library which can produce a high-quality image for various plots.

3.4.1 Formatting Data with Tagged Presences in DataFrame

Before analyzing data, the data must be fetched from the database and tagged with the state of presence. As it was introduced before, a `DataFrame` object is applied to wrap all data for

analysis, from data generation to data pre-processing. The second application of this project **`/common/data-analyse.py`** has been designed to accomplish all necessary procedures around this `DataFrame` object.
Code 3.1 shows a part of function `data_frame_from_device(device_id)`. For a standalone continuous measurement, the `DataFrame` can be generated in 4 steps.

1. All temperature entities and gas entities shall be retrieved by the given `device_id` and wrapped into 2 `list` objects.
2. The temperature list will be then iterated. In the iteration, a new list which contains the temperature and timestamp will be generated. The state of presence shall be determined by `temperature.is_present()`, and attached the new list.
3. The gas list will be iterated, too. In the iteration, a new list which contains only the *eCO2* and timestamp will be created.
4. These 2 lists will be merged into a `DataFrame` indexed by time stamps.

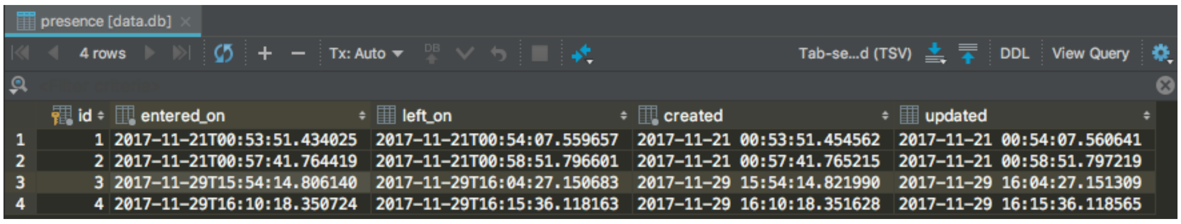
Some datetime-class methods, such as `strftime("%Y-%m-%d %H:%M:%S")` must not be applied for converting `datetime` to `string`, because *string type* variables cannot be plotted. Instead we just keep the timestamp as *datetime object*. Otherwise, an exception will be raised, when it starts to be plotted.

```python
def data_frame_from_device(device_id) -> pd.DataFrame:
    temperature_list = TemperatureModel.find_all_by_device_id(device_id)
    new_temperature_list = []
    for temperature in temperature_list:
        entity = Entity()
        entity.temperature = temperature.temperature
        entity.timestamp = temperature.get_time() # .strftime("%Y-%m-%d %H:%M:%S")
        entity.presence = 1 if temperature.is_present() else 0
        new_temperature_list.append(entity)

    gas_list = GasModel.find_all_by_device_id(device_id)
    new_gas_list = []
    for gas in gas_list:
        entity = Entity()
        entity.eco2 = gas.eco2
        entity.timestamp = gas.get_time() # .strftime("%Y-%m-%d %H:%M:%S")
        new_gas_list.append(entity)

    df_temperature = pd.DataFrame(data=new_temperature_list)
    df_gas = pd.DataFrame(data=new_gas_list)
return pd.merge(df_temperature, df_gas, on='timestamp')
```

Code 3.1 ***Snippets for data generating in /commons/data-analyse.py***

Presences in Multiple Sessions

A session is a time interval, in which the data can be measured and stored via APIs continuously with a determined presence. Within a session, the state of presence must be either "*entered*" or "*left*". During the research, it is found that the session information is an indispensable part to determine the state of presence, especially when the states of presence are segmented in multiple time-discrete sessions. *Figure 3.4* shows a typical example of entities.



	id	entered_on	left_on	created	updated
1	1	2017-11-21T00:53:51.434025	2017-11-21T00:54:07.559657	2017-11-21 00:53:51.454562	2017-11-21 00:54:07.560641
2	2	2017-11-21T00:57:41.764419	2017-11-21T00:58:51.796601	2017-11-21 00:57:41.765215	2017-11-21 00:58:51.797219
3	3	2017-11-29T15:54:14.806140	2017-11-29T16:04:27.150683	2017-11-29 15:54:14.821990	2017-11-29 16:04:27.151309
4	4	2017-11-29T16:10:18.350724	2017-11-29T16:15:36.118163	2017-11-29 16:10:18.351628	2017-11-29 16:15:36.118565

Figure 3.4 ***Screenshot of the presence table with filled data.***

For example, the state of presence can be misunderstood by the DataFrame generating function from the end of last presence in the previous session (`id:2, left_on`) to the begin of the first presence in the next session (`id:3, entered_on`). It will be recognized as an 8-day-long *left state* period. Sessions can be regarded as a project specified environment parameter, which should be stored in **`/settings.py`**. *Code 3.2* shows an example of sessions, which is nothing more than a list object.

```
1     SESSIONS = [
2         ['2018-02-15 14:00:00+00:00', '2018-02-15 16:00:00+00:00'],
3         ['2018-02-16 14:00:00+00:00', '2018-02-16 16:00:00+00:00']
4     ]
```

Code 3.2 ***Customized session variables in /settings.py, line 2 – session 1, line 3 – session 2***

Then, a new function can wrap the previous `data_frame_from_device(device_id)` as in *Code 3.1*. It can be called as `sessions_by_device(dev, sessions)`, since it can output a list of `DataFrame` objects which are segmented by the given sessions and device. *Code 3.3* is the implementation of this function.

```
def sessions_by_device(dev, sessions=settings.SESSIONS):
    dfs = []
    for i, session in enumerate(sessions):
        data = data_frame_from_device(dev.id)
        # filter by session, avoid unexpected wrong session segmentation
        data_session = data[(data.timestamp>session[0]) & (data.timestamp<session[1])]
        # filtering null data.
        data_session = data_session[(data_session.temperature!=0) & (data_session.eco2 != 0)]
        data_session.to_csv(dev.device_name+'_session_'+str(i)+'.csv')
        dfs.append(data_session)
    return dfs
```

Code 3.3 ***Snippets for segmented data generating by sessions in /commons/data-analyse.py***

3.4.2 Data Visualization and Data Pre-processing

Data Visualization

Data visualization is usually applied to analyze data via intuitive plots, statistical graphics or Info-graphics. It is a very clear and efficient way to find out the proper solution for data mining, such as a proper regression model or a proximate classification algorithm.

In this project, the data visualization will be applied as follows.

- The 1st plotting consists of the raw temperature and *eCO2*, their derivate data and presence state (values on *axis-y*) dependent on the time-series (values on *axis-x*), in order to find out the proper data pre-processing from which the outputs have mostly constant trend to the correspondent presence state.
- The 2nd plotting consists of the scatter of the state of presence, dependent on a tuple of proper data after data pre-processing, which are refined after first plotting. The tuple of data contains temperature-related and *eCO2*-related values. Temperature-related values are located on *axis-x*, *eCO2*-related ones are located on *axis-y* and the tagged state of presence is displayed in different colors. This scatters graphics can be used to decide, whether the dataset can be applied to SVM training directly.

Figure 3.5 shows examples of the 1st plotting for 2 different sessions and *Figure 3.6* shows the 2nd plotting.

Data Pre-processing

Data pre-processing is the last procedure in this chapter, which usually includes data-filtering, data-stabilizing, data-transforming, etc. In this project, the data pre-processing consists in the following steps:

1. **Data-filtering**: The entities with invalid `null` values will be swiped.
2. **Data-stabilization**: the swing of data will be stabilized/smoothed by averaging the value in a certain rolling-window.
3. **Data-transforming**:
   I. For a more intuitive view to observe the trends in different presence state, the value of presence will be transformed to 800 for state `entered` and to 700 for state `left`, the temperature will be multiplied by 35 for a clear and grouped view together with *eCO2*.
   II. For compensation of the influence from the environment, it is necessary to transform the directly measured values to incremental values according to the standard-addition method.
4. Besides the common pre-processing methods, **delay compensation** and an **off-set at the start** will be applied, since the time by reflects of sensors has a certain delay after the change of presence-state. The cause for the delay is probably due to the speed of molecular diffusion.

The `data_pre_process(ds)`shall be called before the construction of `DataFrame` objects. Hence, the function can be implemented as it shows in *Code 3.4*.

```python
def data_pre_processing(ds):
    # filtering null data.
    data_session = ds[(ds.temperature != 0) & (ds.eco2 != 0)]
    # eCO2: smoothing data in a rolling-window in 400 seconds
    data_session['eco2'] = data_session['eco2'].rolling(window=400, center=False, min_periods=1).mean()
    # delta temperature:
    # 1. diff in 120 seconds and delay it in 400 seconds.
    # 2. smoothing delta_temperature in a rolling window for 45s
    data_session['delta_temp'] = data_session['temperature'].diff(120).shift(-400)
    data_session['delta_temp'] = data_session['delta_temp'].rolling(window=45, center=False, min_periods=1).mean()
    data_session['delta_eco2'] = data_session['eco2'].diff(120).shift(-400)
    data_session['delta_eco2'] = data_session['delta_eco2'].rolling(window=45, center=False, min_periods=1).mean()
    # off-set for 200sec at session start, to avoid random error.
data_session = data_session.iloc[200:]
return data_session
```

Code 3.4 ***Snippets for data_pre_processing(ds) in /commons/data-analyse.py***

Thus, the newly added function can then be called in `sessions_by_device()`. The modification is shown in *Code 3.5*.

```python
def sessions_by_device(dev, sessions=settings.SESSIONS):
    dfs = []
    for i, session in enumerate(sessions):
        data = data_frame_from_device(dev.id)
        # filter by session, avoid unexpected wrong session segmentation
        data_session = data[(data.timestamp>session[0]) & (data.timestamp<session[1])]
        data_session = data_pre_processing(data_session)
        data_session.to_csv(dev.device_name+'_session_'+str(i)+'.csv')
        dfs.append(data_session)
return dfs
```

Code 3.5 ***Snippets for modified sessions_by_device() in /commons/data-analyse.py***

3.5 Final Data after Pre-processing

After data pre-processing, the trend of *eCO2*, *delta-eCO2*, *delta-temp* (abbr. delta-temperature) are represented in the state of presence as expected. Values with prefix "*delta*" are incremental values.

Finally, we have two plots after data pre-processing, shown in *Figure 3.5*.

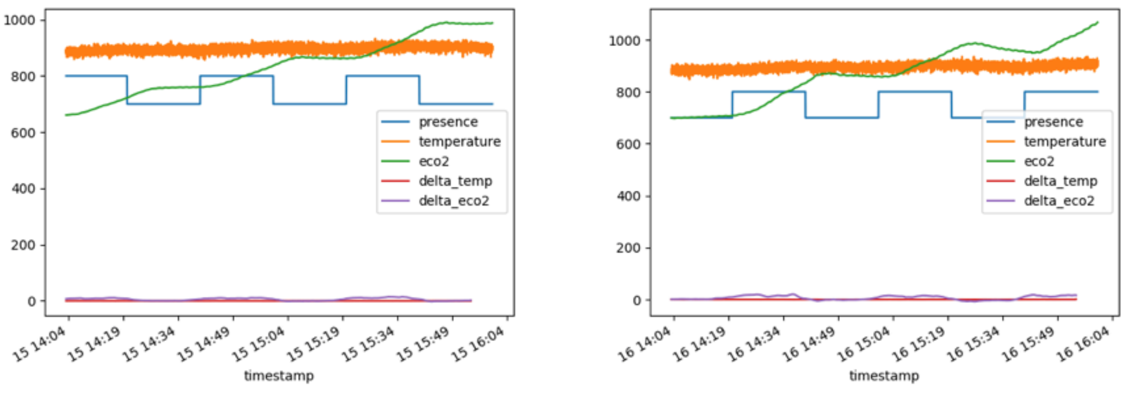


Figure 3.5 The *1st Plotting for 2 different sessions, left – session 1, right – session 2*

After being observed, it is determined, that the curves of *eCO2* and of *delta-eCO2* shows expected trends upon the state of presence. When `entered`, *eCO2* is increased quickly, when `left`, *eCO2* is decreased slightly.

With the same data, the classification by state of presence shows in the two scatters below. The points are colored upon the state of presence, "yellow" for `entered` and violet for `left`. Each colored point has a coordination includes delta-temperature (value in *axis-x*) and *delta-eCO2* (value in *axis-y*), as same as the 2nd plotting mentioned before.

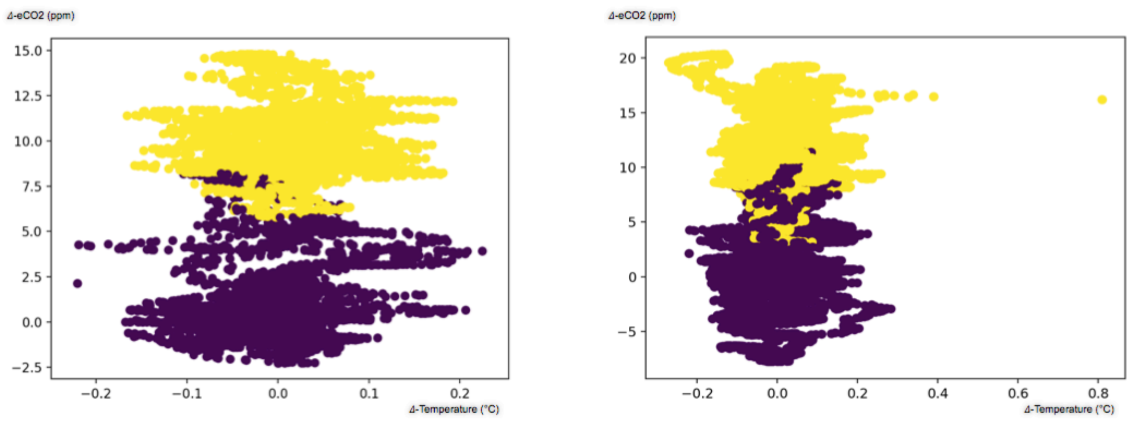


Figure 3.6 The 2nd *Plotting (scattering) for 2 different sessions, left – session 1, right – session 2*

We can suppose now that *SVM* should be applied for this data model because the points with same presence state are clustered in 2-dimensional coordination and the border between 2 classifications can be found from the graphics of both sessions intuitively in *Figure 3.6*. This figure shows a visualization of 2 different sessions. From the visualization of classification in both sessions, we can suppose that *delta-temp* appears irrelevant to the classification. From now on, the data set gets ready for being parsed into a machine learning model.

4 Presence Recognition with Machine Learning

4.1 Machine Learning with Python

Machine learning uses statistical techniques to “learn” with data, without being explicitly programmed. Since the learning system is strongly dependent on the quantity of the statistical data, the more data as well as “study material” being input, the better the machine learns.

Machine learning can be grouped in the different criterion. Generally, the criteria can be the learning tasks or the outputs from learning applications.

4.1.1 Classification upon Learning Tasks

Depending on whether there is a learning *signal* or *feedback* available to a learning system, the learning tasks can be classified into 2 categories, as it shows in *Figure 4.1*.

- **Supervised learning**, which trains a model on known input and output data, so that it can predict future outputs.
- **Unsupervised learning**, which finds hidden patterns or intrinsic structures in input data.

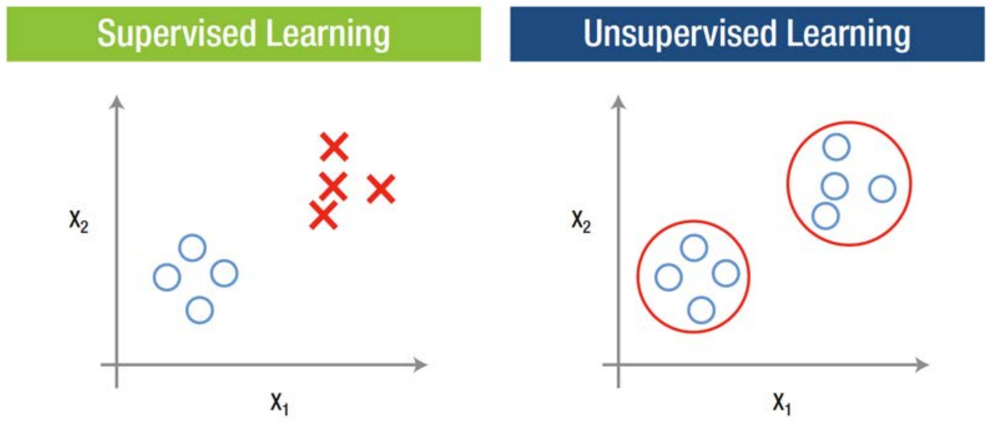


Figure 4.1 ***Supervised learning vs. unsupervised learning (taken from [4], accessed on April 10, 2018)***

4.1.2 Learning Techniques and Algorithms.

Under the learning tasks, machine learning can be divided in different learning techniques. As seen in *Figure 4.2*, ***supervised learning*** uses ***classification*** and ***regression*** techniques to develop predictive models and ***clustering*** is the most common ***unsupervised learning*** technique.

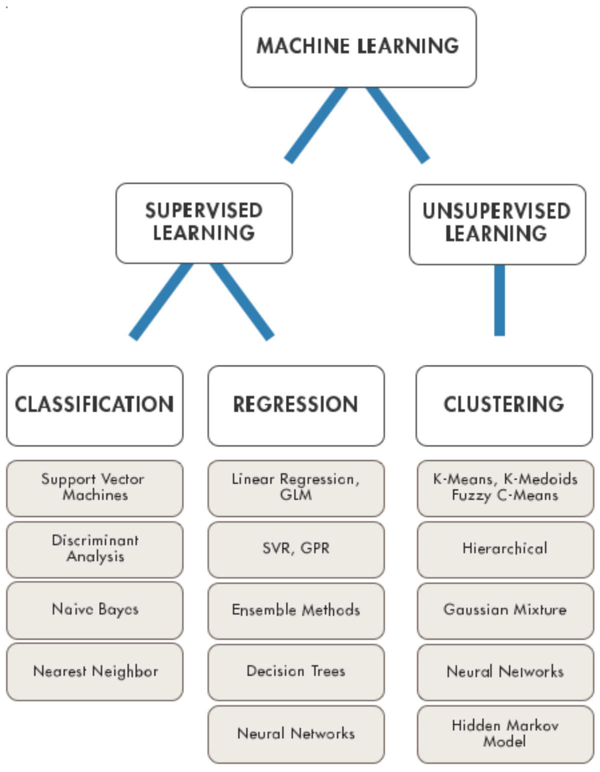


Figure 4.2 ***Categorized hierarchy in machine learning (taken from [4], accessed on April 10, 2018)***

Classification techniques, which predict discrete responses, are generally applied if the data can be tagged, categorized, or separated into specific groups or classes. Common algorithms for performing classification include *Support Vector Machine (SVM), discriminant analysis, Naïve Bayes and Nearest neighbor.*

Regression techniques, which predict continuous responses, are used if the task concerns a data range or the nature of the response is a real number. Common regression algorithms include *linear model, nonlinear model, regularization, stepwise regression, boosted and bagged decision trees, neural networks* and *adaptive neuro-fuzzy learning*.

Clustering techniques are used for exploratory data analysis to find hidden patterns or groupings in data. Common algorithms for performing clustering include *k-means* and *k-medoids, hierarchical clustering, Gaussian mixture models, hidden Markov models, self-organizing maps, fuzzy c-means clustering* and *subtractive clustering.*

4.1.3 Scikit-learn, a Toolkit for Machine Learning in Python

Scikit-learn is a very efficient and easy-to-use toolkit for data mining, data analysis, and machine learning. It is built on *NumPy*, *SciPy,* and *Matplotlib*, which are the most widely used libraries for scientific research in python. In this project, *scikit-learn* is applied to train and test and optimize the machine learning model.

4.2 Theory and Algorithms of SVM

By investigating the plotted scatters in the previous *Figure 3.6*, we have found out, that *SVM* can be applied since the data are discrete and tagged in two classifications by different states of presence. This section mainly provides a theoretical background of *SVM.*

4.2.1 Overview

SVM is a *supervised machine learning* algorithm which can be used for classification problems. In SVM, each data item acts as a point in *n*-dimensional space (where *n* is the number of features) with the value of each feature being the value of a particular coordinate. Thus, the classification by finding the hyperplane that differentiate the two classes can be then performed. This project covers only the linear separable classification.

Support Vectors are simply the coordinates of individual observation. Support Vector Machine is applied to approach a frontier (hyperplane) which best segregates the two classes.

Figure *4.3* shows the *SVM* classification in 2-dimensional coordinates.

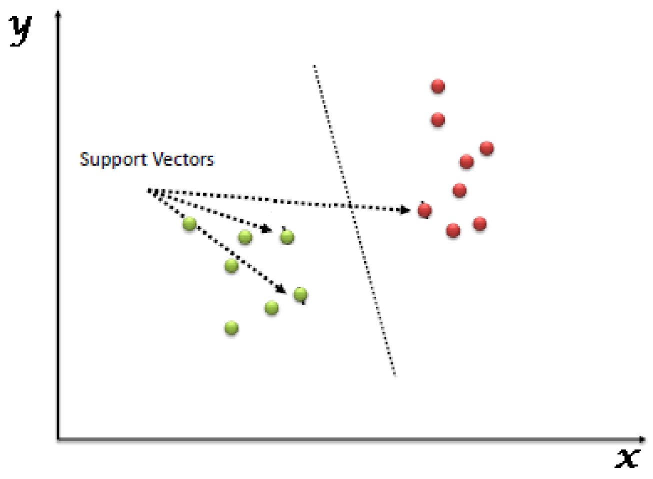


Figure 4.3 ***A simple SVM classification with support vectors and hyperplane (taken from [5], accessed on April 10, 2018)***

Any separating line can be described as the locus of points(x) in the 2-dimensional plane that satisfies the following equation:

$$\beta_0 + \beta^T x = 0, \qquad \beta \text{ and } x \in \mathbb{R}^2$$

It is obvious that there are many possible separating lines can be found between 2 classifications. Thus, it is necessary to use additional criteria to uniquely specify the best-fit separating line (or, hyperplane in a higher-dimensional space)

4.2.2 Maximal Margin Algorithm

Considering a training set consisting of $\{(x, y)\}$ where $y \in \{-1,1\}$. For any point x_i, the functional margin can be calculated as $\hat{\gamma}_i = y_i\,(\beta_0 + \beta^T x_i)$. Thus, $\hat{\gamma}_i > 0$ when x_i is correctly classified. The geometrical margin from x_i to hyperplane is: $\gamma_i = \frac{\hat{\gamma}_i}{\|\beta\|}$. When x_i is correctly classified, γ_i is equal to the perpendicular distance from x_i to the line.

Let M be the width of the functional margin. The maximal margin algorithm can be formulated as a quadratic programming problem. It is required to simultaneously maximize the margin M while ensuring that all of the data points are correctly classified.

$$\underset{\beta_0, \beta, \|\beta\|=1}{maximize} \quad M$$

$$subject\ to\ y_i(\beta^T x_i + \beta_0) \geq M, i = 1, \ldots, N.$$

By getting rid of $\|\beta\| = 1$ in the constraints with the following reformulation:

$$\underset{\beta_0, \beta}{maximize} \quad \frac{M}{\|\beta\|}$$

$$subject\ to\ y_i(\beta^T x_i + \beta_0) \geq M, i = 1, \ldots, N.$$

Since the functional margin M can be adjusted in any scale, so let $M = 1$:

$$\underset{\beta_0, \beta}{maximize} \quad \frac{1}{\|\beta\|}$$

$$subject\ to\ y_i(\beta^T x_i + \beta_0) \geq 1, i = 1, \ldots, N.$$

Transforming the objective as follows, (to a quadratic form)

$$\underset{\beta_0,\beta}{maximize} \quad \frac{1}{\|\beta\|} \Leftrightarrow \underset{\beta_0,\beta}{minimize} \quad \|\beta\| \Leftrightarrow \underset{\beta_0,\beta}{minimize} \quad \frac{1}{2}\|\beta\|^2$$

There are two classes shown in *Figure 4.4* (white and grey points), which are linearly separable. The maximal margin solution is shown by the bold line in the middle. The dotted lines show the extent of the margin. The large circles indicate the support vectors for the maximal margin solution.

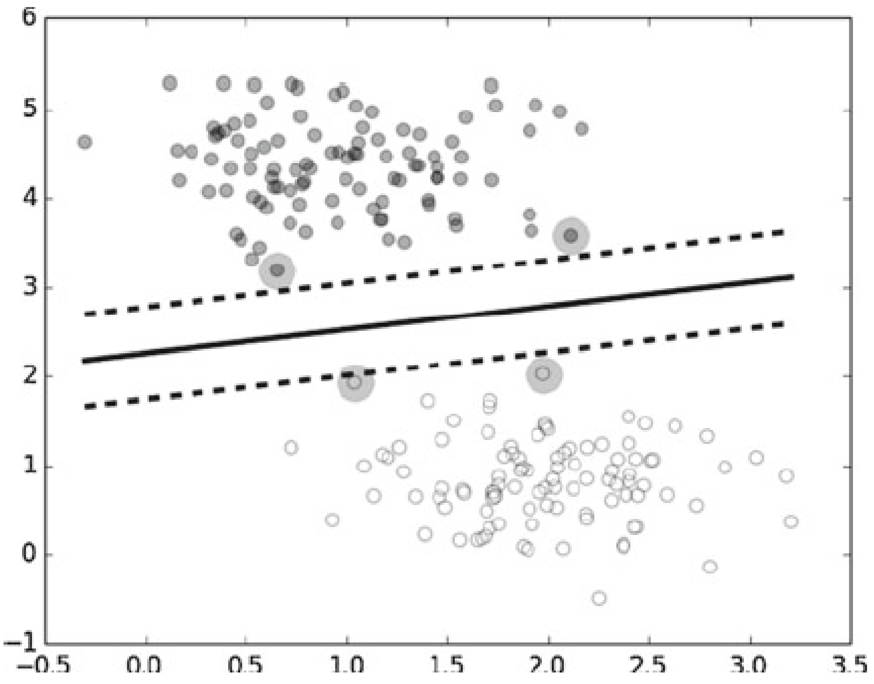


Figure 4.4 ***SVM classification with maximal margin solution (taken from [p.253, Jos16])***

4.2.3 Slack Variables and Penalty for Misclassification

However, the situation becomes usually more complex when the two classes are not separable. By allowing some unavoidable mixing between the two classes in the solution, the slack variable has to be introduced.

The slack variables ζ_i are the slack variables and represent the proportional amount that the prediction is on the wrong side of the margin. Thus, elements are misclassified when $\zeta_i > 1$.

The objective of the quadratic programming has to be modified as follows:

$$\underset{\beta_0,\beta}{minimize} \quad \frac{1}{2}\|\beta\|^2 + C\sum\zeta_i$$

$$subject\ to\ y_i(\beta^T x_i + \beta_0) \geq 1 - \zeta_i,\ \ \zeta_i \geq 0, i = 1, \ldots, N.$$

Because the ζ_i terms are all positive or 0, the objective becomes to maximize the margin ($\Leftrightarrow$minimize $\|\beta\|$) while minimizing the proportional drift of the predictions to the wrong side of the margin ($\Leftrightarrow$ minimize $C\sum\zeta_i$).

It has been already calculated, that the functional margin must be bigger than 0 for a correct classification. Hence, it can be resulted in the following relationships, with the latest introduced slack variables ζ_i.

$$\begin{cases} \zeta_i = 0\ for\ any\ correct\ classified\ x_i \\ 0 < \zeta_i \leq 1\ for\ any\ correct\ classified\ x_i\ within\ the\ extended\ area. \\ \zeta_i > 1\ for\ any\ incorrect\ classified\ x_i \end{cases}$$

Thus, larger values of *C* lead algorithmic focus on more correctly classified points near the decision boundary and smaller values on further data.

In other words, a larger *C* causes a bigger penalty for a misclassification, and a smaller one tolerates more misclassifications. *C* is the so-called *penalty cost for misclassification*.

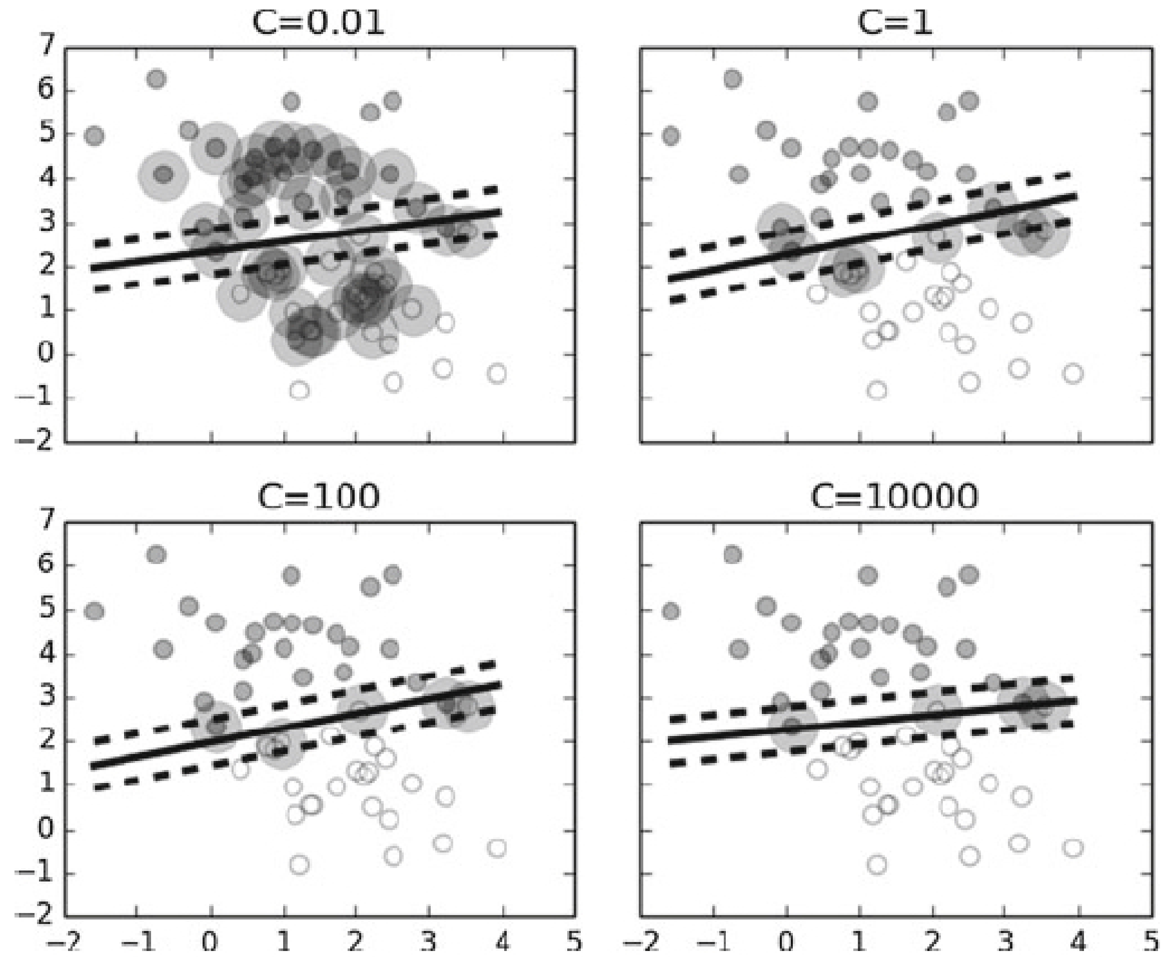


Figure 4.5 ***Different classification due to the modification of C-parameter (taken from [p.254, Jos16])***

In *Figure 4.5*, the maximal margin algorithm finds the separating line that maximizes the margin shown. The elements which touch the margins are the support elements. The dotted line is not relevant to the solution anymore. It is easy to find that the bigger the C is, the closer the circled elements to the hyperplane and vice versa.

Until now, a linear separable SVM model is built completely. However, SVM can be applied for *non-linear separable classification* with *kernel trick* as well, which is but out of the scope of this project.

4.3 Applying the SVM on Pre-proceeded Data

To find the optimized hyperplane, which is equivalent to the solution of the quadratic programming, the “method of Lagrange multipliers” will be used to transform the objective with constraints into a *Lagrangian dual problem*. Then, the *Sequential Minimal Optimization Algorithm (SMO)* can be applied to find the optimized hyperplane. Implemented with the popular LIBSVM tool, *SMO* has been proved as a very efficient iterative algorithm for training *SVM*.

The introduction to theoretical part of *SVM* ends up here because the aim of this project is to apply the *SVM* for analyzing the data instead of analyzing the algorithm itself.

Fortunately, all of this complicated reformulation and iterative computing are neatly inside of *Scikit-learn*, which is also developed based on LIBSVM for *SVM* training.

Applying the *SVM* on the pre-proceeded data consists in 5 steps, described in the next subsections.

4.3.1 Splitting the Pre-proceeded Data for Training and Testing

In order to train the model, the data points must be first split into training- and test-dataset randomly, in both of which the input-data (`X_train | X_test`) and output-data (`y_train | y_test`) are prepared for the training and validation.

Since the pre-proceeded data have been saved in `CSV` format, a `DataFrame` can be initialized, by reading all data from the CSV-file. Then, they can be split into 2 data-frames, in which one (`y`) contains a reference (presence) and another (`X`) the features (`delta_temperature`, `delta_eco2`). By using `train_test_split()` method, the train-dataset (`X_train | y_train`) and test-dataset (`X_test | y_test`) are built, in which the size of the test-dataset is as twice big as the training-dataset. The `Train_test_split()` method is a commonly used method to split arrays or matrices into random train and test subsets. A float between 0.0 (0%) and 1.0 (100%) can be parsed into the parameter `test_size`, in which the size of test-dataset can be set using a percentage. A random seed can be then parsed into parameter `random_state` as a random number generator. If it is set as `None`, the random number generator is the *RandomState instance* used by `np.random`. A literal `random_state` with value `101` is applied here, which can keep the split results identical for reproducibility with pseudo-random-method.

```
        # get data from csv file
        file_name = 'cfe137a13701_session_0.csv'
        df = pd.read_csv(file_name).dropna()
        del(df['Unnamed: 0'])
        y = df['presence']
        X = df[['delta_temp', 'delta_eco2']]
        # init the test and training set from y, X
    X_train, X_test, y_train, y_test =
    train_test_split(X,y,test_size=0.66,random_state=101)
```

Code 4.1 ***Snippets for splitting training- and test-dataset in /commons/training.py***

4.3.2 Training the Data

With the powerful tool offered by scikit-learn, training data may be the simplest work in this project, as *Code 4.2* shows.
Method `SVC(kernel=’linear’)` will initialize a model for *SVM*-training. The parameter kernel declares that a linear-kernel is explicitly applied in this training-model. With `model.fit(X_train, y_train)` method, the training-dataset has been already evaluated.

```
        # init training model
        model = SVC(kernel='linear') # linear kernel
        # training model
        model.fit(X_train,y_train)
```

Code 4.2 ***Snippets for training data by SVM in /commons/training.py***

4.3.3 Validating the Hyperplane with the Test-set

The function `model.predict()` offers a simple way to validate the trained model in one line (*Code 4.3*). The output results in *Table 4.1*.

```
        # get the validation without grid search
        predictions = model.predict(X_test)
```

Code 4.3 ***First validation before optimization in /commons/training.py***

4.3.4 Adjusting the C-parameter and Re-training the Data.

As it was discussed before, the C-parameter, aka. *penalty cost* of misclassification, can be adjusted to improve the classification. Scikit-learn offers a `GridSearchCV class` to evaluate the best-fit `C` and `gamma`. *Gamma-parameter* can be ignored here because it is for the non-linear kernel. After a *grid search*, the best parameters can be found. *Code 4.6* shows an example of output after a *grid research*.

```
        # improve the presicise
        from sklearn.grid_search import GridSearchCV
        param_grid =
    {'C':[0.1,1,10,100,1000],'gamma':[1,0.1,0.01,0.001,0.0001]}
        grid = GridSearchCV(model,param_grid,verbose=3)
        grid.fit(X_train,y_train)

        # Re-training
        model = SVC(kernel='linear',C=grid.best_params_['C'])
        model.fit(X_train, y_train)
```

Code 4.4 ***Optimizing by grid search and re-training the model with best parameters in /commons/training.py***

4.3.5 Validating the Hyperplane after Optimization

Code 4.5 shows validation using a grid search (The code looks similar to *Code 4.3*). The output shows in *Table 4.2*.

```
        # validation with new predicted model
        grid_predictions = grid.predict(X_test)
```

Code 4.5 ***Second validation after optimization with a better C-parameter in /commons/training.py***

4.4 Results of Validation

4.4.1 Introduction to the Confusion Matrix

In the field of machine learning, a *confusion matrix* is a specific table layout that can be applied to visualize the performance of classification algorithms, esp. in a supervised learning case. Each row of the matrix represents the instances in a *predicated class* (aka. *predicted condition*) while each column represents the instances in an *actual class* (aka. *true condition*). *Table 4.1* shows a general 2x2 confusion matrix. Each predicted results can be filled in the following 4 cells:

- *True positive (TP),* if a predicted condition positive hits the actual condition (correctly identified);
- *True negative (TN),* if a predicted condition negative hits the actual condition (correctly rejected);
- *False positive (FP)*, if a false predicted condition positive occurs (incorrectly identified);
- *False negative (FN),* if a false predicted condition negative occurs (incorrectly rejected).

Because the scalar values are enormously different from case to case, some derivations from a *confusion matrix* are applied as performance-index. These derivations are listed below in terminology.

- *Precision*:= $\frac{TP}{TP+FP}$, aka. *Positive Predictive value (PPV)*
- *Recall*:= $\frac{TP}{P} = \frac{TP}{TP+FN}$, aka. *True Positive Rate (TPR)*
- *F1-score*:= $2 \cdot \frac{PPV \cdot TPR}{PPV+TPR} = \frac{2TP}{2TP+FP+FN}$, aka. *Harmonic Mean* of *precision* and *recall*.

All 3 factors have values between 0.0 (0%) and 1.0 (100%). The higher the values, the better performance algorithms have.

		True condition	
		Condition positive (P)	Condition negative (N)
Predicted condition	Predicted condition positive	**True positive (TP)**	**False positive (FP)**
	Predicted condition negative	**False negative (FN)**	**True negative (TN)**

Table 4.1 ***A general 2x2 confusion matrix (adapted from [11], accessed on August 15, 2018)***

4.4.2 Validation before Grid Search

Confusion Matrix:

	Real presence: Entered	Real presence: Left
Predicted presence: Entered	**2122**	**47**
Predicted presence: Left	**31**	**2227**

Table 4.2 ***Confusion matrix before optimization***

Classification Report:

	Precision	**Recall**	**F1-score**	**Support**
0	**0.99**	**0.98**	**0.98**	**2169**
1	**0.98**	**0.99**	**0.98**	**2258**
Avg. / total	**0.98**	**0.98**	**0.98**	**4427**

Table 4.3 ***First validation results before optimization***

By investigating the confusion matrix in *Table 4.2*, we find that all cells of the diagonal (from left-top to right-bottom) show the correct recognition (true entered and true left), and all cells of anti-diagonal (from left-bottom to right-top) show the confusion (false left and false entered). With high recognition and low confusion, a well-qualified *classification report* is shown in *Table 4.3*.

4.4.3 Optimized Parameters and Validation after Grid Search

Best Parameters Found:

```
{'C': 1000, 'gamma': 1}
```

Code 4.6 *Best parameters found after grid search*

Confusion Matrix:

	Real presence: Entered	Real presence: Left
Predicted presence: Entered	**2123**	**46**
Predicted presence: Left	**27**	**2231**

Table 4.4 *Confusion matrix after optimization*

Classification Report:

	Precision	**Recall**	**F1-score**	**Support**
0	**0.99**	**0.98**	**0.98**	**2169**
1	**0.98**	**0.99**	**0.98**	**2258**
Avg. / total	**0.98**	**0.98**	**0.98**	**4427**

Table 4.5 *Second validation results after optimization*

4.4.4 Results Visualization and Evaluation

By comparing the confusion matrices and the classification reports of the *SVM*-modelling before (*Table 4.2* and *Table 4.3*) and after optimization with grid search (*Table 4.4* and *Table 4.5*), there is no noteworthy difference found, although C is adjusted from 1.0 to 1000. However, if these 2 models are plotted in a 2-dimensional space, it is still very clear to uncover that the slope of optimized hyperplane looks more similar to the trends of the support elements (yellow-colored dots).

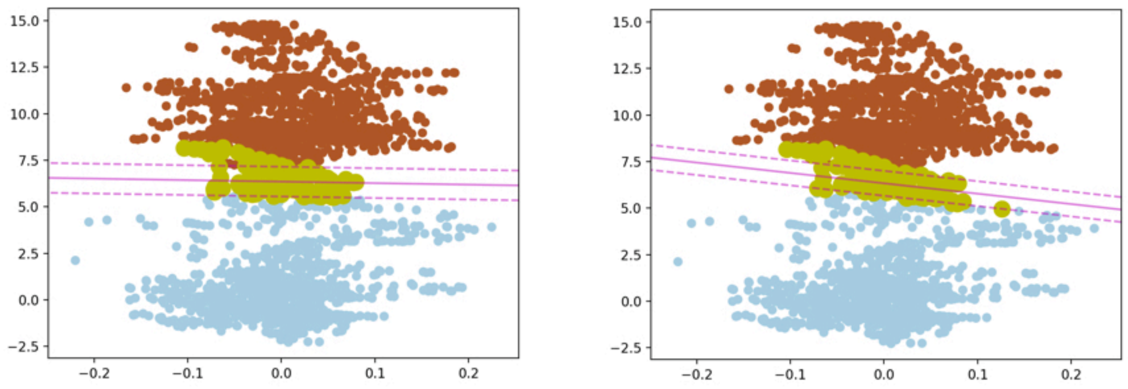


Figure 4.6 ***Before- vs. after-optimization, different classification and hyperplanes due to different C-parameters***

According to the high values in *precision* (0.99), *recall* (0.98) and *F1-score* (0.98), it is confirmed that the trained hyperplane fit the data very well and should be regarded as an acceptable model to predicate the presence with the pre-proceeded data.

All codes for this section, incl. training, validation, optimizing with grid search and all the data visualization can be found in **`/commons/training.py`** (The source code was given along with the thesis).

5 Conclusion and Future Work

5.1 Process Review

The project duration was approximately 11 months. The scope of this project is so wide, that it has to take a lot of time to read various theoretical literature, technique documentation, and device user-manual.

Initially, it was thought, that the coding could be more challenging. However, the most headache part is the measurement, which was considered the simplest phase on the before getting started. This lets me realize that the *data sampling* is very important for data analyzing. When encountering some unexpected results, it is always worthwhile to review the data sampling process once more.

5.2 Conclusion

Given the final results, the goal of this system has been successfully reached. After repeated attempts, it is found that state of presence is more highly related to the development of *eCO2* in the air than to the room temperature.

Features

Due to the benefits of machine learning and RESTful style web service, the system has following remarkable features:

+ This system has the ability to responsively recognize presence by adopting the given devices and environment. Otherwise, the measurement can be enormously deviated by different devices, different positions, different sizes of the environment or even different time-periods (day/night), which might fail any pre-defined logic for presence prediction. Due to the machine learning algorithm, it is simple to compensate the system error caused by differences from devices and environments during the “leaning-phase”.
+ The layered system makes “high cohesion, low coupling”, which enhances consequently the flexibility, maintainability, and scalability of the system. For example, nothing needs to be modified in model objects for changing/adding any web resource endpoint or for improving the machine learning algorithms.
+ Thanks to the accessibility, the system can communicate with other web services for the purpose as aggregation, notification, or data mining with other criteria.

Limitations

Any system has its own limitations. During the development and measurement, some issues have been uncovered during implementation as follows:

- This recognition is strongly depended on a single data modeling related to *eCO2*. Hence, it might be hard or even failed to find a hyperplane, while it is being performed in a room without ventilation or in a too big room, which cannot be measured with only one sensor.
- Although the linear kernel is simple to understand and works well in development, an important fact must be taken into consideration that all the time interval in the same presence cannot exceed 20 minutes. By shortening or prolonging the interval, the linear kernel can probably invalid and leads to an incorrect result.
- Much time needs to be spent on repeated experimental analysis to determine the pattern of data pre-processing.
- Due to the “client-service-pattern”, the system can be influenced by any network-traffic and network-security issues as well. Without authentication, any client can pretend as a thingy-gateway to request the data via RESTful API.

5.3 Improvement and Application

5.3.1 Improvement

By solving the above-listed limitations, this project might be optimized and improved in following aspects:

Increment of Sampling

Multiple sensors can be applied to augment the correctness of measuring, esp. in a room with large space.

Automation of Data Pre-process

Correspondent to the multiple sensor data, a well fit data pre-process pattern can be generalized with machine learning automatically.

Optimizing the Machine Learning Algorithms.

The non-linear kernel can replace the linear kernel for a more generalized solution.

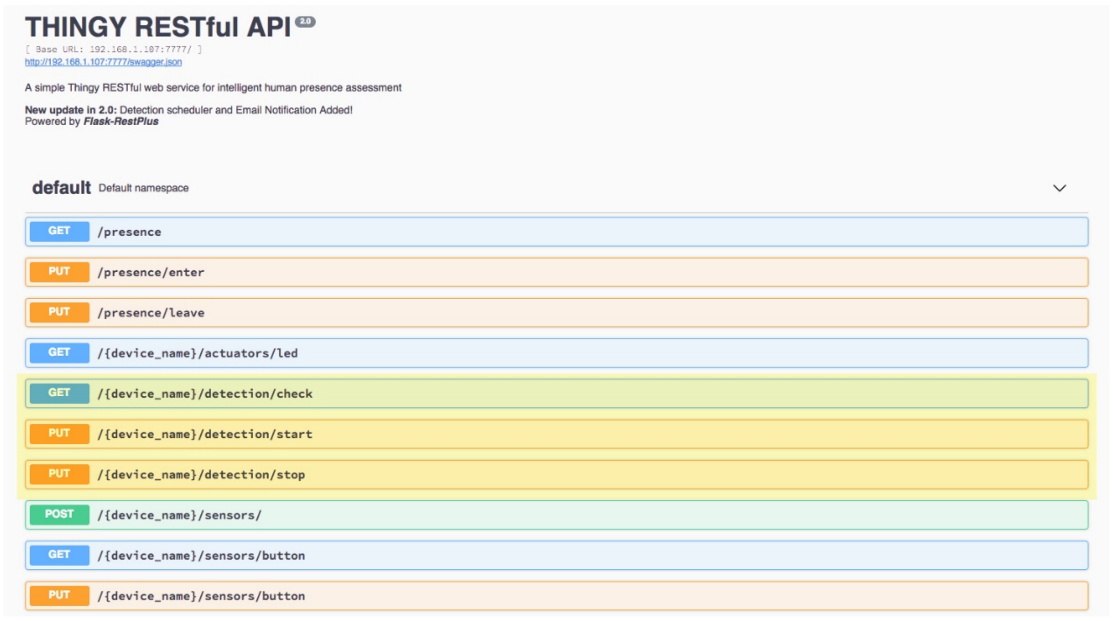
Enhancing Security Policy

JSON web token (abbr. JWT) can be integrated into the system for enhanced security.

5.3.2 Application

Based on this project, a monitoring or a notification system can be built with extended APIs.

For monitorin the presence, prediction event can be run as a time-scheduled job/cronjob. The results should be persisted in a normal RDBMS or Redis and the correspondent API can be prepared for querying the last state of presence. *Figure 5.1* shows newly added *Detection-APIs* for the monitoring system.


THINGY RESTful API 2.0

```
[ Base URL: 192.168.1.107:7777/ ]
```
http://192.168.1.107:7777/swagger.json

A simple Thingy RESTful web service for intelligent human presence assessment

New update in 2.0: Detection scheduler and Email Notification Added!
Powered by ***Flask-RestPlus***

default Default namespace

```
GET   /presence
PUT   /presence/enter
PUT   /presence/leave
GET   /{device_name}/actuators/led
GET   /{device_name}/detection/check
PUT   /{device_name}/detection/start
PUT   /{device_name}/detection/stop
POST  /{device_name}/sensors/
GET   /{device_name}/sensors/button
PUT   /{device_name}/sensors/button
```

Figure 5.1 *Screenshots of API Docs with newly added Detection-APIs*

Thus, people can use client or another service to retrieve the predicted presence for monitoring the presence in real-time. *Figure 5.2* shows newly updated system with monitoring function.

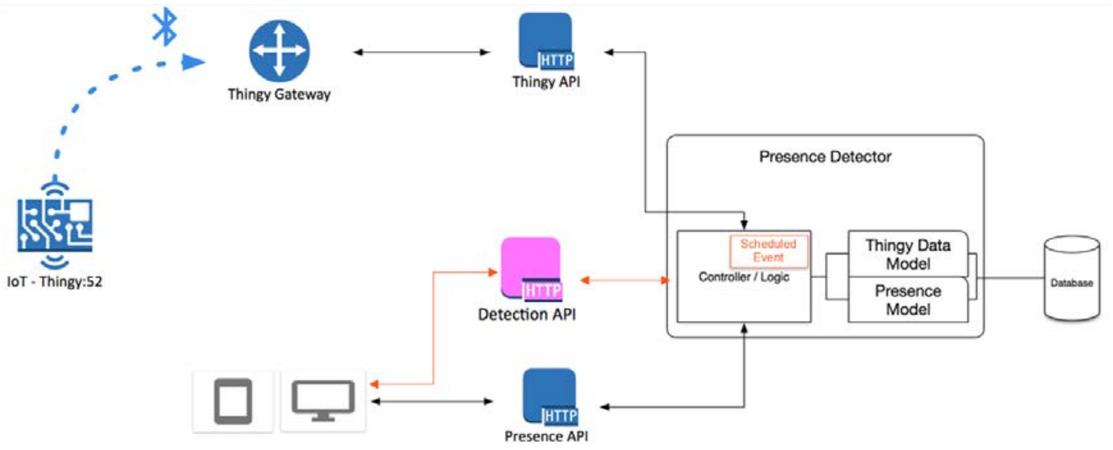


Figure 5.2 *Updated architecture of monitoring system (newly added Detection-APIs and Scheduled-Event)*

Furthermore, a notification system with predefined constraints can be built upon the monitoring API. By adding the event listener and flask-mail object, we can trigger a sending mail request to a certain SMTP-server, when the state of presence is changed. Hence, a pre-defined email can be sent to the pre-defined user. A completed architecture shows in *Figure 5.3*.

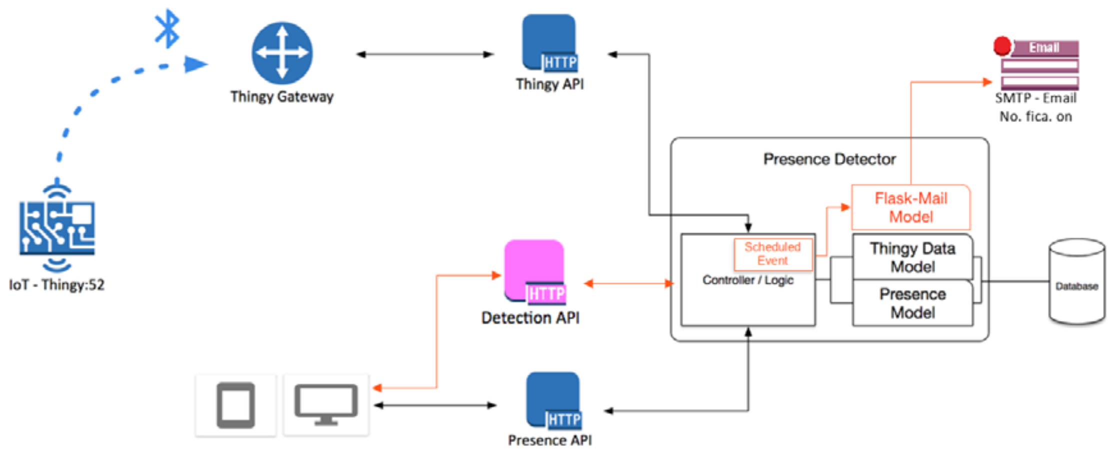
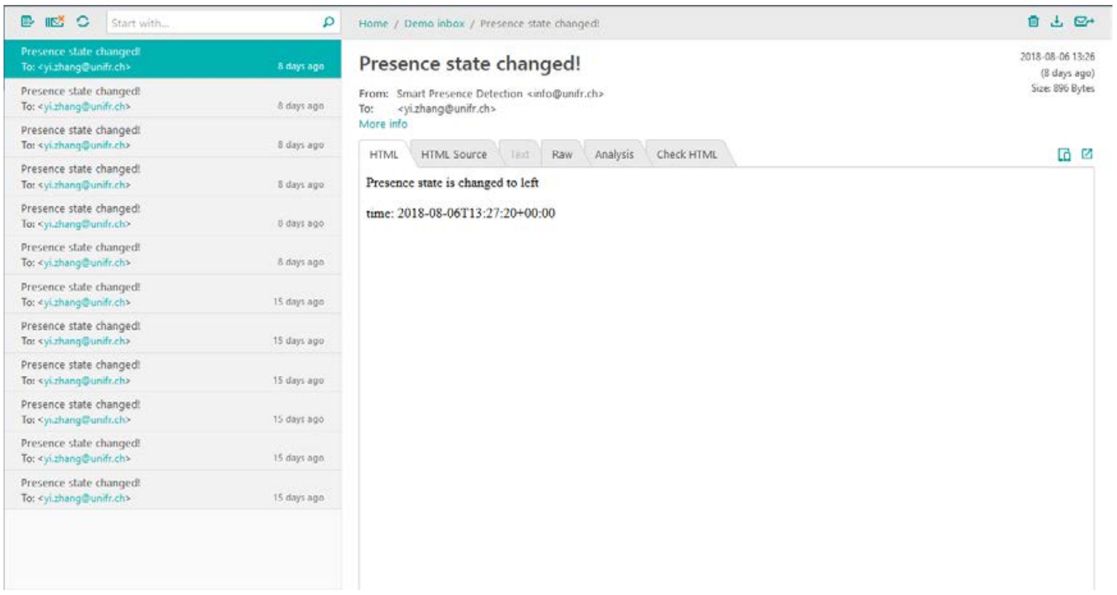


Figure 5.3 *Updated architecture of notification system (newly added Email-Notification)*

Figure 5.4 shows the example of email notifications in a webmail client. For purpose of development, ***Mailtrap*** is used as a pseudo mail server/client.


Start with...

Presence state changed!
To: <yi.zhang@unifr.ch> 8 days ago

Presence state changed!
To: <yi.zhang@unifr.ch> 8 days ago

Presence state changed!
To: <yi.zhang@unifr.ch> 8 days ago

Presence state changed!
To: <yi.zhang@unifr.ch> 8 days ago

Presence state changed!
To: <yi.zhang@unifr.ch> 8 days ago

Presence state changed!
To: <yi.zhang@unifr.ch> 8 days ago

Presence state changed!
To: <yi.zhang@unifr.ch> 15 days ago

Presence state changed!
To: <yi.zhang@unifr.ch> 15 days ago

Presence state changed!
To: <yi.zhang@unifr.ch> 15 days ago

Presence state changed!
To: <yi.zhang@unifr.ch> 15 days ago

Presence state changed!
To: <yi.zhang@unifr.ch> 15 days ago

Presence state changed!
To: <yi.zhang@unifr.ch> 15 days ago

Home / Demo inbox / Presence state changed!

Presence state changed!

2018-08-06 13:26
(8 days ago)
Size: 896 Bytes

From: Smart Presence Detection <info@unifr.ch>
To: <yi.zhang@unifr.ch>
More info

HTML | HTML Source | Text | Raw | Analysis | Check HTML

Presence state is changed to left

time: 2018-08-06T13:27:20+00:00

Figure 5.4 *Examples of email notifications in mailtrap web client*

The latency of notification takes about 30 seconds to 90 seconds, which can be regarded as an acceptable performance.

6 Terms of Use

6.1 Author

Yi Zhang

Student for BSc. Informatics in university Fribourg (CH)

6.2 License

References

[Gar17]

Gareth Dwyer, Shalabh Aggarwal, Jack Stouffer. *Flask: Building Python Web Services, Packt Publishing Ltd.*, 2017, ISBN 978-1-78728-822-5

[Gas16]

G. C. Hillar, *Building RESTful Python Web Services*, *Packt Publishing Ltd.*, 2016, ISBN 978-1-78646-225-1

[Bre12]

E. Bressert. *SciPy and NumPy, First edition, O'Reilly Media, Inc.*, 2012, ISBN: 978-1-449-30546-8

[Ant15]

F. Anthony. *Mastering pandas, Packt Publishing Ltd.*, 2015, ISBN 978-1-78398-196-0

[Jos16]

J. Unpingco. *Python for Probability, Statistics, and Machine Learning, Springer International Publishing Switzerland*, 2016, ISBN 978-3-319-30715-2

[Nor18]

John D. Poole. Model-Driven Architecture: Vision, Standards And Emerging Technologies. In *ECOOP '01: Workshop on Metamodeling and Adaptive Object Models*, OMG, 2001. [Retrieved January 08, 2008, from `http://www.omg.org/mda/mda_files/Model-Driven_Architecture.pdf`]

Referenced Web Resources

[1] Nordic Thingy:52 https://www.nordicsemi.com/eng/Products/Nordic-Thingy-52 (accessed on November 30, 2017)

[2] Remote Method Invocation (Java RMI) https://www.slideshare.net/sonalizoya/distributed-systemremote-method-invocation-java-rmi. (accessed on April 05, 2018)

[3] Wikipedia - Web service https://en.wikipedia.org/wiki/Web_service (accessed April 05, 2018)

[4] What Is Machine Learning https://www.mathworks.com/discovery/machine-learning.html (accessed on April 10, 2018)

[5] Support Vector Machine - Classification (SVM http://www.saedsayad.com/support_vector_machine.htm. (accessed on April 10, 2018)

[6] Flask's documentation http://flask.pocoo.org/docs/0.12/ (accessed on January 05, 2018)

[7] Flask-RESTPlus's documentation http://flask-restplus.readthedocs.io/en/stable/ (accessed on January 05, 2018)

[8] Understanding Support Vector Machine algorithm from examples (along with code) https://www.analyticsvidhya.com/blog/2017/09/understaing-support-vector-machine-example-code/ (accessed on April 15, 2018)

[9] Who Needs the Internet of Things https://www.linux.com/news/who-needs-internet-things (accessed on November 30, 2017)

[10] Wikipedia - Representational state transfer https://en.wikipedia.org/wiki/Representational_state_transfer (accessed on November 20, 2017)

[11] Wikipedia – Sensitivity and specificity https://en.wikipedia.org/wiki/Sensitivity_and_specificity (accessed on August 15, 2018)