

CS 498 Internet of Things

Lab 4

Team Ace

Alexey Burlakov (NetID: alexeyb2)

Christopher Lewis (NetID: calewis4)

Li Yi (NetID: liyi2)

Pui Sze Pansy Ng (NetID: ppn2)

Zhijie Wang (NetID: zhijiew2)

Table of Contents

Table of Contents	2
1. Introduction	3
2. Build the Cloud	3
2.1 Background on AWS IoT	3
2.2 Device (thing)	3
2.3 Configure rules	5
2.4 Create many devices	7
2.5 Use the device simulator	9
2.6 GreenGrass	10
3. Understand the Data	12
4. Data Inference with ML	14
4.1 Process incoming messages in Lambda	14
4.2 Return your results back to device	15
5. Use IoT Analytics to visualize the Data	17
5.1 Background	17
5.2 Setup data path	17
5.3 Set up the data set and test if you can receive data	18
5.4 Create Jupyter notebook	19
5.5 Visualize the data	20
6. Problems that encountered	21
7. Individual Contributions:	23

1. Introduction

In this lab, we used Amazon Web Service's IoT platform to use an emulator to simulate 120,000 Samsung Gear watches and collect their data. Then, we constructed a new to monitors people's heart rates and attempts to predict heart attacks

2. Build the Cloud

2.1 Background on AWS IoT

1. Compared with the TCP-based client-server communication model, what're the pros and cons for MQTT-based publish/subscribe?

TCP-based client-server communication model only for communications between a server and one client. It transmits messages using Ethernet and aims for accuracy of the data package. For pros, MQTT is good for real time systems because it uses asynchronous messaging protocol. MQTT also is best used when the internet connection is unstable and suitable for event-driven systems. On the other hand, MQTT has some down sides. MQTT has a very light messaging protocol which is only good for sending sensor data instead of photos/audios/videos.

2. What are the tradeoffs in MQTT when there are a large number of clients?

When there is a large number of clients, there is a possibility that MQTT will have server load issues. But, the server will load balance to other intermediate servers. The tradeoffs probably are to require higher cost.

Where there are a large number of clients, a MQTT broker is needed to handle a large number of publishers and subscribers. MQTT brokers do generally expect a persistent connection, and will timeout clients that don't send ping responses (or other activity) periodically. MQTT does support the concept of 'retained' messages. The client can publish something to a topic with the retained flag, and this message will then be stored by the broker.

2.2 Device (thing)

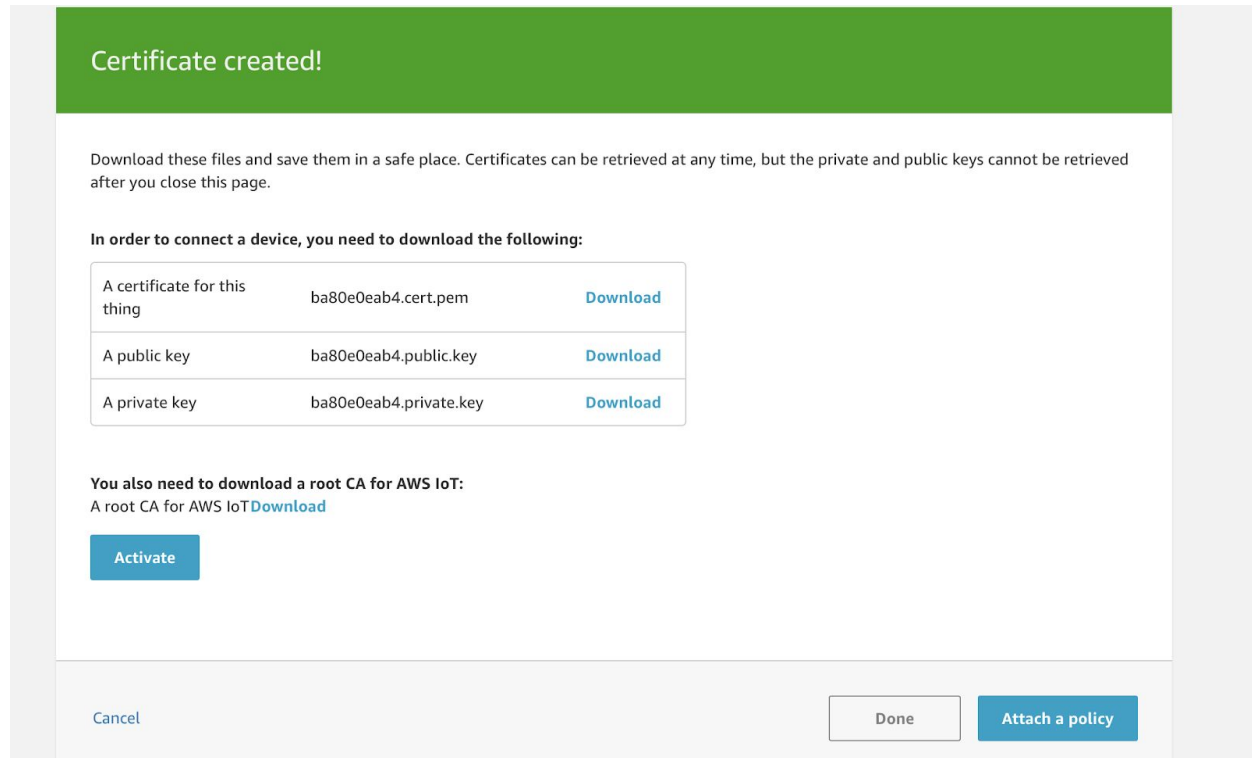


Figure 2.2.1: Screenshot of three key files created

2.3 Configure rules

Create a policy

Create a policy to define a set of authorized actions. You can authorize actions on one or more resources (things, topics, topic filters). To learn more about IoT policies go to the [AWS IoT Policies documentation page](#).

Name

Add statements

Policy statements define the types of actions that can be performed by a resource. **Advanced mode**

Action

Resource ARN

Effect

☒ Allow ☐ Deny

Remove

Figure 2.3.1: Screenshot of create a policy page

RULE

lab4_rule

ENABLED

Actions ▾

Overview

Tags

Description

Edit

forward the message to IoT analytics and use lambda functions. The rule acts as a trigger for the Amazon Lambda that will initiate the machine learning classification process.

Rule query statement

Edit


The source of the messages you want to process with this rule.

```
SELECT * FROM 'data/heartbeat'
```

Using SQL version 2016-03-23

Actions

Actions are what happens when a rule is triggered. [Learn more](#)

 Send a message to a Lambda function
heartbeatesult

Remove Edit ▶

Add action

Figure 2.3.2

The screenshot shows the AWS IoT Rule Editor interface for a rule named **result_role**. The rule is currently **ENABLED**. The interface includes a sidebar with **Overview** and **Tags** tabs. The main content area is divided into sections: **Description** (with an **Edit** link), **Rule query statement** (with an **Edit** link), and **Actions**. The **Rule query statement** section contains the SQL query: `SELECT *, timestamp() as my_timestamp FROM 'data/result'`. Below the query, it indicates **Using SQL version 2016-03-23**. The **Actions** section explains that actions occur when a rule is triggered and provides a **Learn more** link. A single action is configured: **Send a message to IoT Analytics**, which uses the `dataresult_channel` target. This action has **Remove** and **Edit** options. At the bottom, there is an **Add action** button.

Figure 2.3.3

2.4 Create many devices

Our team modified `createThings-cert.py` provided by Lab4 document, added policy names and thing certificates directory. By following the instructions, 500 things are added into AWS IoT core, and shown on aws management console. Please see the following screenshot.

Things			Create
Search things <input type="text"/> <input type="button" value="Q"/> <input type="button" value="Fleet Indexing"/> ⓘ			List ▼
<input type="checkbox"/> Name	Type		
<input type="checkbox"/> HelloWorld_Subscriber	NO TYPE		***
<input type="checkbox"/> HelloWorld_Publisher	NO TYPE		***
<input type="checkbox"/> awsec2mqtt	NO TYPE		***
<input type="checkbox"/> lab4greengrass_Core	NO TYPE		***
<input type="checkbox"/> REIGlIS1WQE7Zrk	NO TYPE		***
<input type="checkbox"/> u1n9y6a4sSsozsm	NO TYPE		***
<input type="checkbox"/> ul0V9O9NoQ57oFI	NO TYPE		***
<input type="checkbox"/> 2D5g7ppuHGwW6yQ	NO TYPE		***
<input type="checkbox"/> frvgTzyhHogUOh5	NO TYPE		***
<input type="checkbox"/> gzfkR66ADNR7fz	NO TYPE		***
<input type="checkbox"/> nqzypKxaryXD4CT	NO TYPE		***
<input type="checkbox"/> p7dgg4yGHDZeLTO	NO TYPE		***
<input type="checkbox"/> qJYr9f5n3hQeiDw	NO TYPE		***
<input type="checkbox"/> 9lkd08ANIUCB5Zo	NO TYPE		***

Figure 2.4.1


```
##### Connecting to AWS
import boto3

import json
##### Create random name for things
import random
import string

##### Parameters for Thing
thingArn = ''
thingId = ''
#thingName = ''.join([random.choice(string.ascii_letters + string.digits) for n in xrange(15)])
defaultPolicyName = 'Lab4_policy'
#####

def createThing(i):
    global thingClient
    thingName = ''.join([random.choice(string.ascii_letters + string.digits) for n in range(15)])
    thingResponse = thingClient.create_thing(
        thingName = thingName
    )
    data = json.loads(json.dumps(thingResponse, sort_keys=False, indent=4))
    for element in data:
        if element == 'thingArn':
            thingArn = data['thingArn']
        elif element == 'thingId':
            thingId = data['thingId']
        createCertificate(thingName, i)
    return thingName

def createCertificate(thingName, i):
    global thingClient
    certResponse = thingClient.create_keys_and_certificate(
        setAsActive = True
    )
    data = json.loads(json.dumps(certResponse, sort_keys=False, indent=4))
    for element in data:
        if element == 'certificateArn':
            certificateArn = data['certificateArn']
        elif element == 'keyPair':
            PublicKey = data['keyPair']['PublicKey']
            PrivateKey = data['keyPair']['PrivateKey']
        elif element == 'certificatePem':
            certificatePem = data['certificatePem']
        elif element == 'certificateId':
```

Figure 2.4.2: screenshot of createThing-cert.py

2.5 Use the device simulator

Our team installed AWSIoTPythonSDK, and modified lab4 emulator client.py provided by lab4 instruction. First, the data and certificate directories are changed:

#Path to the dataset, modify this
data_path = "data/class_{}.csv"

#Path to your certificates, modify this

certificate_formatter = "certificates/thing_{}.certificate.pem"

key_formatter = "certificates/thing_{}.private.pem"

Second, the endpoint ARN and credential are modified:

```
self.client.configureEndpoint("a191olynvpydfg-ats.iot.us-west-2.amazonaws.com", 8883)
```

```
self.client.configureCredentials("root-ca-cert.pem", key, cert)
```

At last, we modified publish function to publish the data from csv file properly.

```
index = np.random.randint(0, len(data[self.state]))
```

```
payload = {
```

```
    "device_id": self.device_id,
```

```
    "class": str(self.state),
```

```
    "features": list(data[self.state].iloc[index].values)
```

```
}
```

```
self.client.publishAsync("data/heartbeat", json.dumps(payload), 0,
```

```
ackCallback=self.customPubackCallback)
```

After changes above, we were able to execute the python script, and sent data with topic "data/heartbeat"

2.6 GreenGrass

We set up our GreenGrass core on an EC2 instance and open port 22 and 8883 to allow SSH and MQTT protocol to access the EC2 instance. Once we finished EC3 instance set up, we installed AWS IoT Greengrass Core software. At last, Greengrass started by command:

```
cd /greengrass/ggc/core/
sudo ./greengrassd start
```

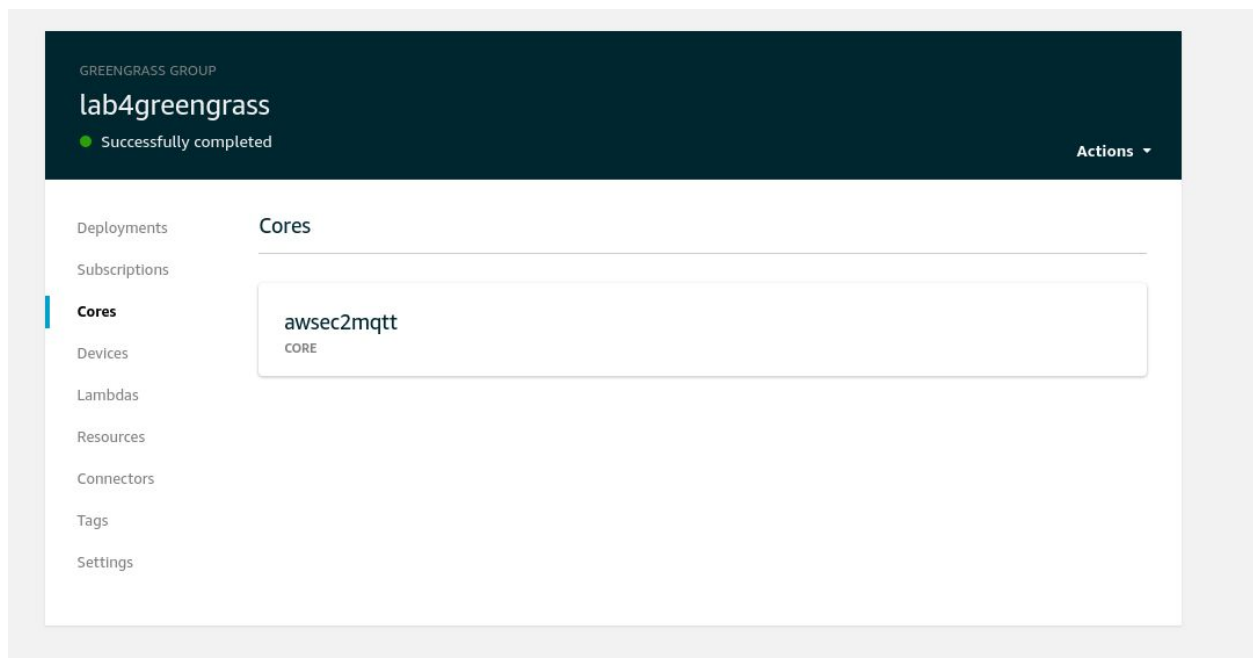


Figure 2.6.1

Following AWS instruction, we then set up few IoT devices (Python clients). We also set up numerous subscriptions between the publisher device, subscriber device, IoT cloud, and lambda functions.

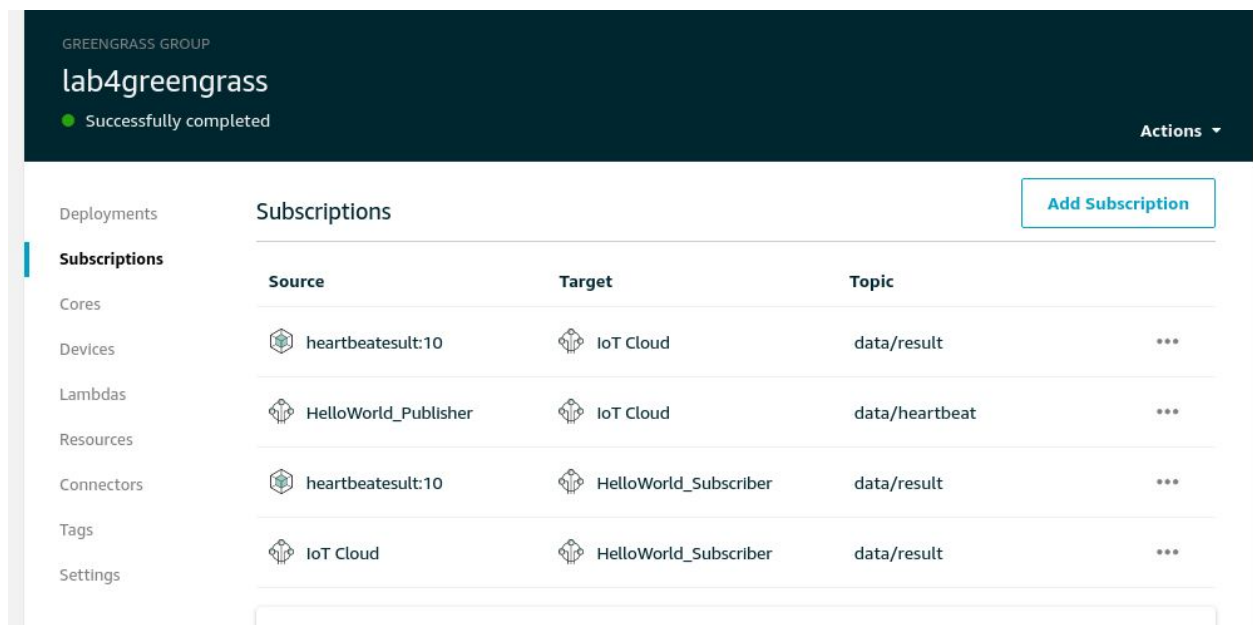


Figure 2.6.2

3. Understand the Data

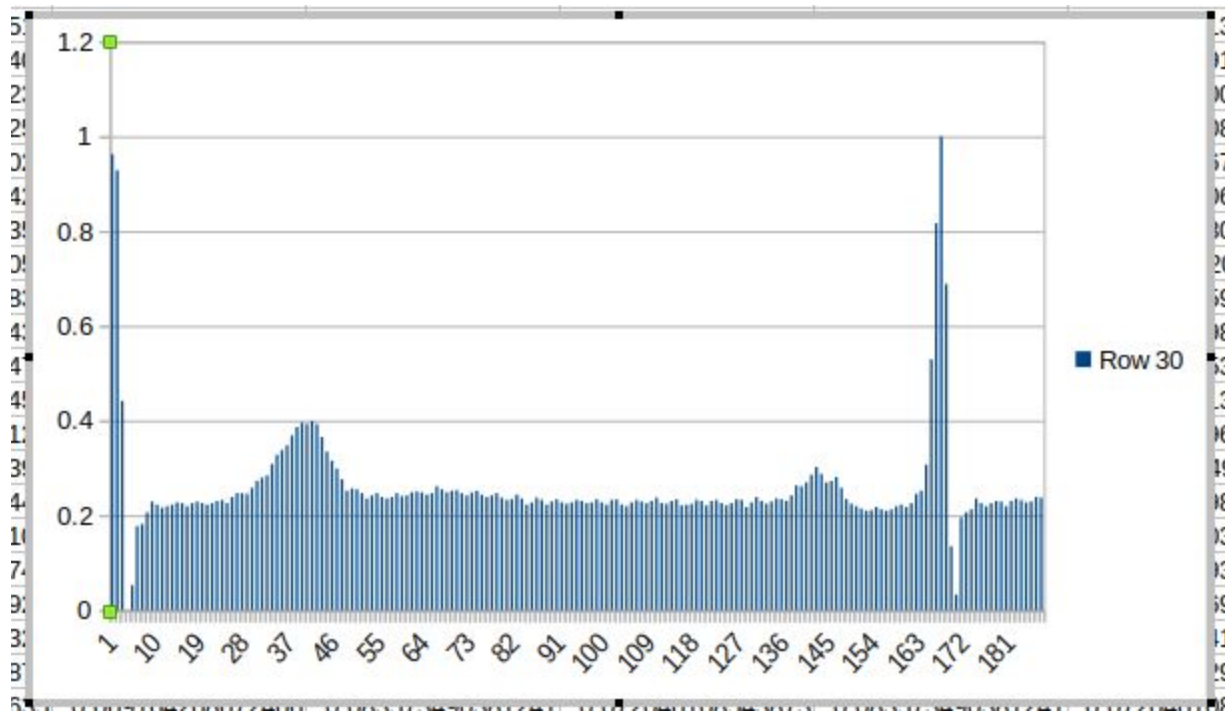


Figure 3.1

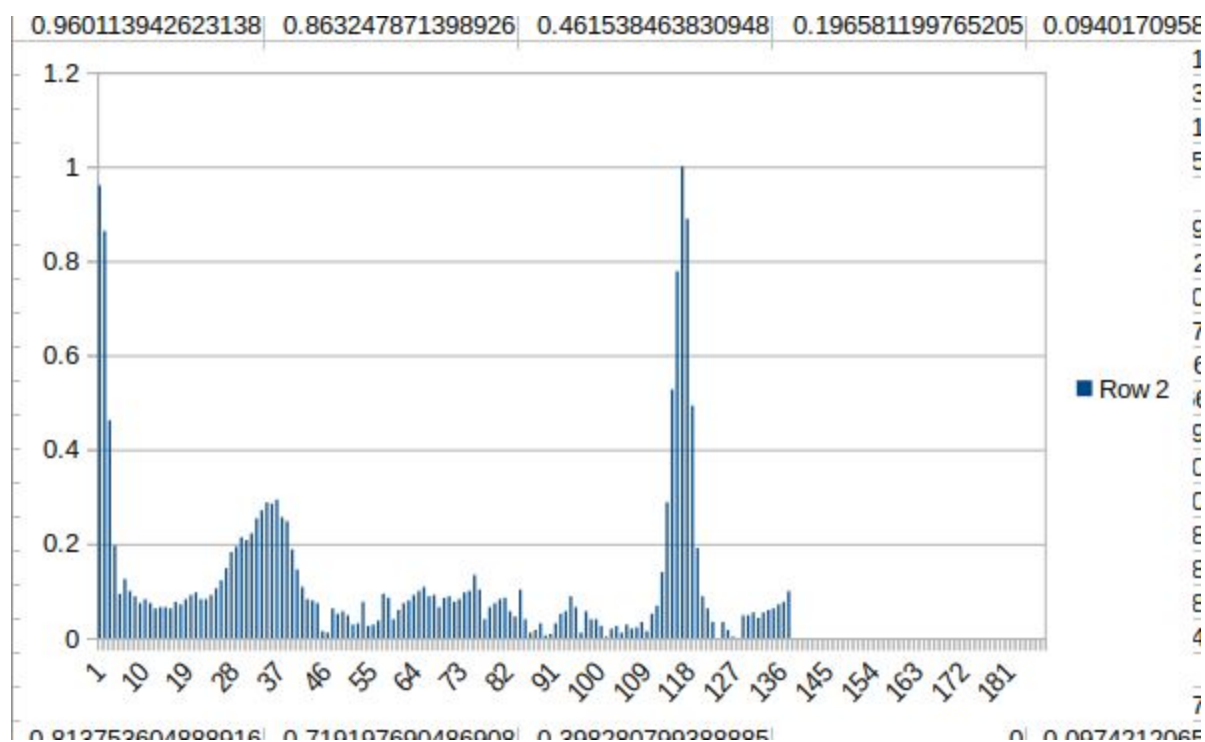


Figure 3.2

We downloaded data.zip provided by lab4 instruction. From the plot we created, we can see the figure matches "QRS complex" - electrical pattern observed on ECG from human heartbeat. There are 5 csv files in data.zip: we put the data source directory in our publishing python script. We learned each row represents heartbeats from one watch. We used the topic 'data/heartbeat' to publish messages to IOT Core. IoT Core then forwards messages to Lambda functions and IoT analytics.


4. Data Inference with ML

4.1 Process incoming messages in Lambda

1. Create Amazon Lambda with Python 2.7
2. Add Trigger with Forward Rule

Add trigger

Trigger configuration

 **AWS IoT**
aws devices iot

IoT type
Configure a custom IoT rule, or set up an IoT button.

☒ Custom IoT rule
☐ IoT Button

Rule
Pick an existing rule, or create a new one.

Forward

Rule query statement
select * from "data/hearttrate"

Lambda will add the necessary permissions for AWS IoT to invoke your Lambda function from this trigger. [Learn more](#) about the Lambda permissions model.

☒ **Enable trigger**
Enable the trigger now, or create it in a disabled state for testing (recommended).

Figure 4.1.1

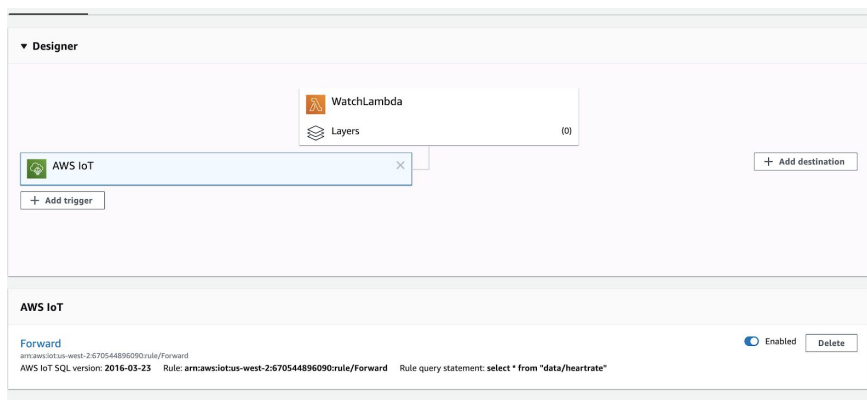


Figure 4.1.2

3. Add functional code

Function code [Info](#)

The deployment package of your Lambda function "WatchLambda" is too large to enable inline code editing. However, you can still invoke your function.

Code entry type:

Runtime:

Handler:

Function package:

For files larger than 10 MB, consider uploading using Amazon S3.

Figure 4.1.3

4. Configure proper role policy to allow write back to AWS IoT (AWSIoTDataAccess)

Role ARN [arn:aws:iam::670544896090:role/service-role/WatchHeartPrediction-role-tah23kjd](#)

Role description [Edit](#)

Instance Profile ARNs [Edit](#)

Path /service-role/

Creation time 2020-05-02 02:29 PDT

Last activity Not accessed in the tracking period

Maximum CLI/API session duration 1 hour [Edit](#)

Permissions [Trust relationships](#) [Tags](#) [Access Advisor](#) [Revoke sessions](#)

▼ Permissions policies (2 policies applied)

[Attach policies](#) [Add inline policy](#)

Policy name	Policy type
AWSIoTDataAccess	AWS managed policy
AWSLambdaBasicExecutionRole-9f4f02d2-91f9-4a16-9f49-5787e33cacc7	Managed policy

► Permissions boundary (not set)

Figure 4.1.4

4.2 Return your results back to device

Add IoT communication client back to service.py

```
#-*- coding: utf-8 -*-
import boto3
import numpy as np
import scipy
import sklearn
import json
import pickle

class Classifier:
    def __init__(self):
        with open("model.pkl", "rb") as f:
```

```

        self.model = pickle.load(f)

    def predict(self, data):
        z = self.model.predict([data])
        return z[0]

classifier = Classifier()
iot_data = boto3.client('iot-data', region_name='us-west-2')

def lambda_handler(event, context):
    # TODO1: Get your data
    data = event["features"]
    data = np.array(data).astype(np.float)
    out = classifier.predict(data)

    response = iot_data.publish(
        topic='data/heartattack',
        qos=1,
        payload=json.dumps({
            "device_id": event["device_id"],
            "classification": out
        })
    )

```

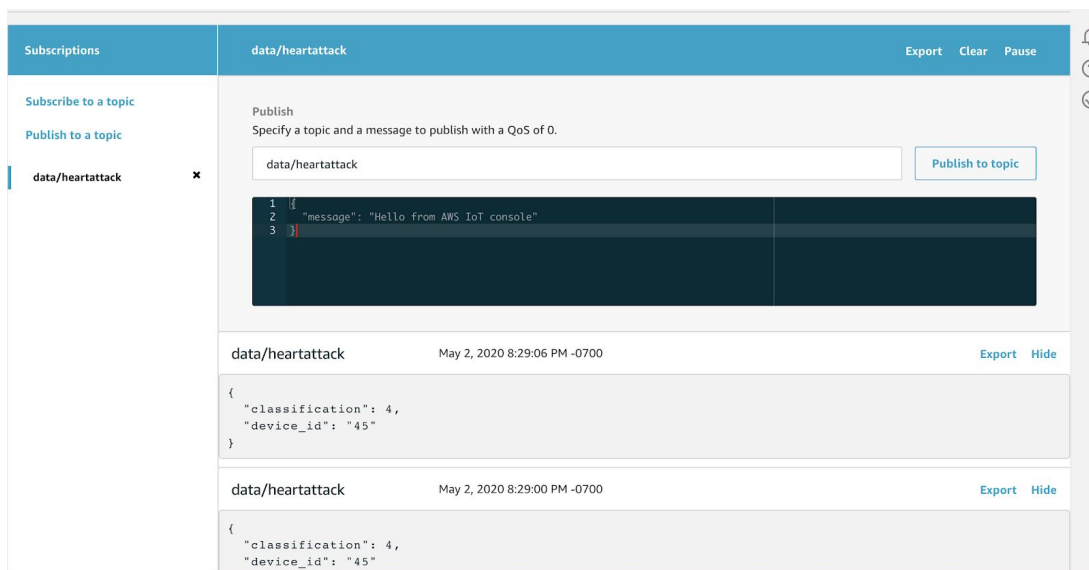


Figure 4.2.1

Ideally one device should only see its own result, for privacy and security. Given the Pub/Sub mechanism of MQTT protocol, it is impractical as the number of topics will grow linearly as the number of devices. Given the current data is not sensitive, there are no personal identifiable attributes or location, each device can see others' prediction results, allowing us to use shared topic for returning results. Each device needs to listen and filter on the result topic and only respond to topics addressing itself.

5. Use IoT Analytics to visualize the Data

5.1 Background

We configure the AWS IoT Analytics Stack using the Wizard. Many steps are automatically created and later configured with detail connection info.

*Channel name

watchheartrate_channel	Clear	Select
------------------------	-------	--------

*Role (requires IoT Analytics access)

WatchHeartRate_role	Create Role	Select
---------------------	-------------	--------

Figure 5.1.1

5.2 Setup data path

Configure AWS IoT Analytics Pipeline to stream data from Channel to Storage

Channel inputs		Edit
Name	Type	
watchheartrate_channel	Channel	
Activities		Edit
Name	Type	
Data store outputs		Edit
Name	Type	
watchheartrate_datastore	Data store	
Tags		Edit
No tags		
Monitoring		

Figure 5.2.1
Define data attributes

Pipelines enrich, transform, and filter messages based on their attributes. Upload a sample JSON message or enter attributes manually to get started.

Attributes

string ▼

Add new

<input type="checkbox"/>	Attribute name		
<input type="checkbox"/>	classification	4	...
<input type="checkbox"/>	device_id	"45"	...

Figure 5.2.2

5.3 Set up the data set and test if you can receive data

The data from the previous Lambda function test is present here.

UPDATE DATA SET

Author query

Craft a query to generate the data set.

Query

```
1 SELECT * FROM WatchHeartRate_datastore
```

Query duration: 2564 ms

Result preview

classification	device_id	__dt
4.0	45	2020-05-03 00:00:00.000
4.0	45	2020-05-03 00:00:00.000

Figure 5.3.2

5.4 Create Jupyter notebook

Create Notebook

CREATE NOTEBOOKS

Set up Notebook

Name the Notebook, set one or multiple data sets as a source for this Notebook, and select a Notebook instance to which the new Notebook will belong.

Name

Select data set sources

[Clear](#) [Edit](#)

Select a Notebook instance

[Create new](#) [Edit](#)

Create Notebook Instance to host notebook execution

Create a new Notebook instance to organize and manage Notebooks.

Notebook instance name

Instance type

Select a role

Notebook

Create new Clear Edit

Tags

Key	Value
<input type="text" value="Key"/>	<input type="text" value="Value"/>

Clear

Add Tag

5.5 Visualize the data

After steps above, we opened notebook in Jupyter Notebook.

Dataset and dataset_url are already loaded. We need to plot the data in the datastore. As the requirement of Lab4.

“When you are done, connect your cloud to our emulator. Write some scripts to predict heart attacks. Produce a chart of the number of lives you save.”

We plotted prediction result vs. timestamp. Please see the following figure includes the python script.

```
In [1]: import boto3
import pandas as pd
from matplotlib import pyplot as plt

# create IoT Analytics client
client = boto3.client('iotanalytics')
```

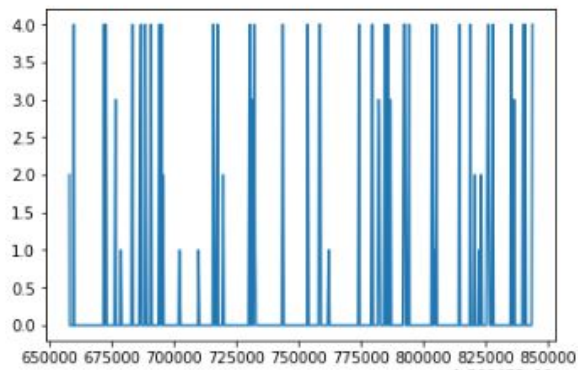
Now we can get the data location (URL) for the given dataset and start working with the data (In order to need to perform grant iot analytics corresponding IAM permission):

```
In [2]: dataset = "dataresult_dataset"
dataset_url = client.get_dataset_content(datasetName = dataset)['entries'][0]['dataURI']

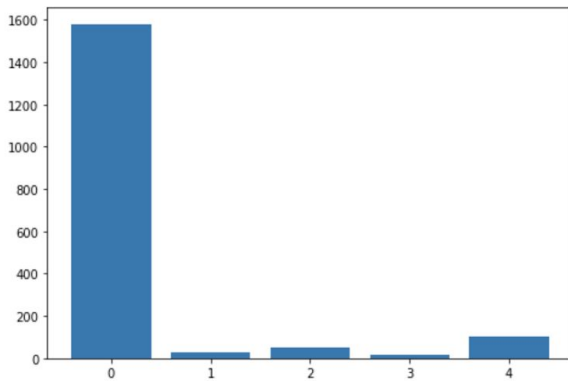
# start working with the data
```



```
In [3]: df = pd.read_csv(dataset_url)
x = df.my_timestamp
y = df.prediction
fig, ax = plt.subplots()
ax.plot(x,y)
#df.my_timestamp.plot(legend=True)
#df.prediction.plot(legend=True)
plt.show()
```



```
df=pd.read_csv(dataset_url)
df= df.classification.value_counts()
df = df.reset_index()
df = df.sort_values(['index'])
fig = plt.figure()
ax = fig.add_axes([0,0,1,1])
x = df["index"]
y = df["classification"]
ax.bar(x,y)
plt.show()
```



6. Problems that encountered

1. Boto3, AWSCLI on different versions of Python.
The example provided for creating Certificates and IoT things are on Python 2.
Resolution is to use mixed Python2 and Python3 in development.
2. Automatically created policies sometimes are too restrictive or using default generated ARN.

```
→ lab4 python test.py
No handlers could be found for logger "AWSIoTPythonSDK.core.protocol.mqtt_core"
Traceback (most recent call last):
  File "test.py", line 16, in <module>
    myMQTTClient.connect()
  File "/usr/local/lib/python2.7/site-packages/AWSIoTPythonSDK/MQTTLib.py", line 513, in connect
    return self._mqtt_core.connect(keepAliveIntervalSecond)
  File "/usr/local/lib/python2.7/site-packages/AWSIoTPythonSDK/core/protocol/mqtt_core.py", line 199, in connect
    raise connectTimeoutException()
AWSIoTPythonSDK.exception.AWSIoTExceptions.connectTimeoutException
→ lab4 python test.py
```

Solution: change policy to something like below -- key is to set resources to "*" or a predetermined set of topic names.

```
{
  "Version": "2012-10-17",
  "Statement": [
    {
      "Effect": "Allow",
      "Action": [
        "iot:*",
        "iot:Publish",
        "iot:Subscribe",
        "iot:Connect",
        "iot:Receive"
      ],
      "Resource": "*"
    }
  ]
}
```

3. AWS Lambda failed to write back to the IoT Data stream

Solution: grant permission through AWS Lambda execution role by attaching correct policy.

Role ARN: `arn:aws:iam::670544896090:role/service-role/WatchHeartPrediction-role-tah23kjd`

Role description: [Edit](#)

Instance Profile ARNs: [View](#)

Path: `/service-role/`

Creation time: 2020-05-02 02:29 PDT

Last activity: Not accessed in the tracking period

Maximum CLI/API session duration: 1 hour [Edit](#)

Permissions policies (2 policies applied)

[Attach policies](#) [Add inline policy](#)

Policy name	Policy type	
AWSIoTDataAccess	AWS managed policy	✕
AWSLambdaBasicExecutionRole-9f4f02d2-91f9-4a16-9f49-5787e33cacc7	Managed policy	✕

Permissions boundary (not set)

4. AWS Lambda throws error

"errorMessage": "Unable to import module 'service': dynamic module does not define module export function (PyInit_multiarray)",

"errorType": "Runtime.ImportModuleError"

The instruction of importing and configuring Lambda functions follows after GreenGrass configuration. In GreenGrass configuration, the python version is 3.7.

The code provided in the Lab instruction is based on Python 2.7.

Solution: modify runtime of the AWS Lambda function to 2.7.

7. Individual Contributions:

Team members are working on the project individually on section 1 -4. But, when issues were encountered, we discussed and helped each other to troubleshoot. Each team member has full understanding of lab4, and is able to demo any part of the lab.