National University of Singapore

Project 1: To simulate IoT Pipeline on AWS platform

Name :	LIANG MING
Matric. No. :	A0195025H
Email :	e0383690@u.nus.edu
Course :	EE5111
Assignment :	AWS IOT
Date :	21, Sep, 2019

Contents

1. Introduction

1.1 Background

The **Internet of Things** (**IoT**) is a system of interrelated computing devices, mechanical and digital machines, objects, animals or people that are provided with unique identifiers (UIDs) and the ability to transfer data over a network without requiring human-to-human or human-to-computer interaction.

The definition of the Internet of things has evolved due to the convergence of multiple technologies, real-time analysis machine learning commodity sensors, and embedded systems. Traditional fields of embedded systems, wireless sensor networks control systems, automation (including home and building automation, and others all contribute to enabling the Internet of Things. In the consumer market, IoT technology is most synonymous with products pertaining to the concept of the "smart home", covering devices and appliances (such as lighting fixtures, thermostats, home security systems and cameras, and other home appliances) that support one or more common ecosystems, and can be controlled via devices associated with that ecosystem, such as smartphones and smart speakers.

There are a number of serious concerns about dangers in the growth of IoT, especially in the areas of privacy and security and consequently industry and governmental moves to begin to address these.

In this project, we will two small IoT settings to record and push data from two jet engines. The engines' data provided is from a high fidelity system level engine simulation designed to simulate nominal and fault engine degradation over a series of flights.

1.2 Goals

1. Simulate two small IoT setups that record and push data from two jet engines.
2. Visualise the data for two engines for all the sensors by querying the data from AWS Dynamo DB.
3. Ingest real-time data from embedded systems

1.3 Tools

AWS IoT platform
A laptop with Anaconda 3 installed.
Python >= 3.5

2. Publish pre-defined engine data to AWS

2.1 Basic settings

2.1.1 Reconstruct data trainFD001:

Modify the Jupyter notebook in Step 1 to read and publish data from trainFD001.txt to your thing under AWS IoT platform at the rate of 10 second per row. Overwrite column 'id' of the engine as 'FD001' + id (e.g. FD001_12); add one more columns 'timestamp' as timestamp in UTC (e.g. UTC 2019-01-28 14:41:15.237); also add one more column that contains my Matric number A0195025H. The Jupyter code is as follow:

```
In [4]: import pandas as pd
        import time as t
        df = pd.read_csv('/Users/frankie/Downloads/EE module/Semester2/Selected Topics in Industrial Co
                         ,delim_whitespace = True,header = None)
        sensor_number = ['s' + str(i) for i in range(1,22)]
        columns_list = ['id','time','os1','os2','os3'] + sensor_number
        df = pd.DataFrame(df.values,columns = columns_list)
        df[['id','time']].astype('int')
        df['id'] = df['id'].map(lambda s: 'FD001_'+str(s))
        time_now = t.time()
        utc_time = t.strftime('%s',t.gmtime(time_now))
        df['timestamp'] = 'UTC 2019-09-16 16:50:15.237'
        df['Matrix_number'] = 'A0195025H'
        df
```

Figure 2-1 The Jupyter code

2.1.2 Create the AWS IoT policy:

Open the AWS IoT console. To do this, on the AWS navigation bar, choose Services. In the Find a service by name or feature box, enter IoT, and then press Enter. In the AWS IoT console. In the service navigation pane, expand Secure, and then choose Policies. Create a new policy and name it "publish trainFD001" as follow:

Figure 2-2 The created policy

2.1.3 Create a thing in AWS IoT

Choose Manage. In the service navigation pane, with Manage expanded, choose Things, and then register a thing to create a single thing:

Figure 2-3 The created Thing

2.1.4 Create certificate and download it

Follow the web browser's onscreen directions to save the file ending in certificate.pem.crt.txt on the computer. Download the public.pem.key, private.pem.key, Amazon Root CA 1 too. And click the ' activate' to active the certificate. Then add the previous policy 'publish trainFD001' for the thing. Register Thing.

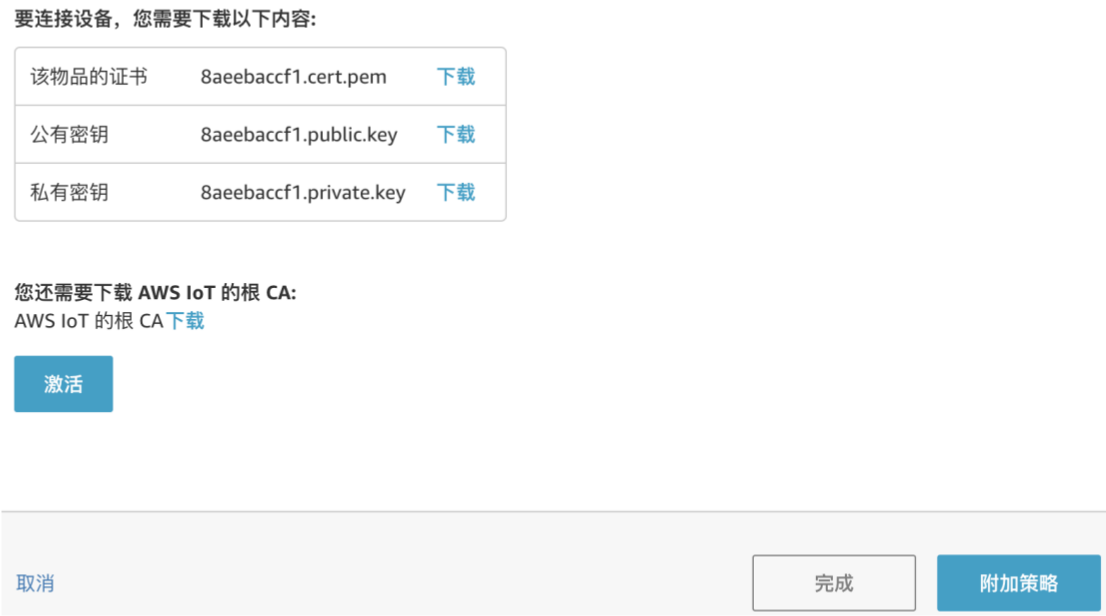


Figure 2-4 Create the policy

2.1.5 Set up Email Subscription

Create an AWS IoT rule to trigger the email subscription through Amazon SNS. To do this, with the AWS IoT console open, in the service navigation pane, choose Act. Name the rule: 'Rule Step 2'. For Rule query statement, with Using SQL version set to 2016-03-23, in the Rule query statement box, enter the following AWS IoT SQL statement as a single line, without any line breaks:

```
SELECT * FROM '$aws/things/A0195025H/shadow/update/accepted'
```

Then adding an action, choose SNS message notification. Enter a name for the SNS topic: 'publish_data'.

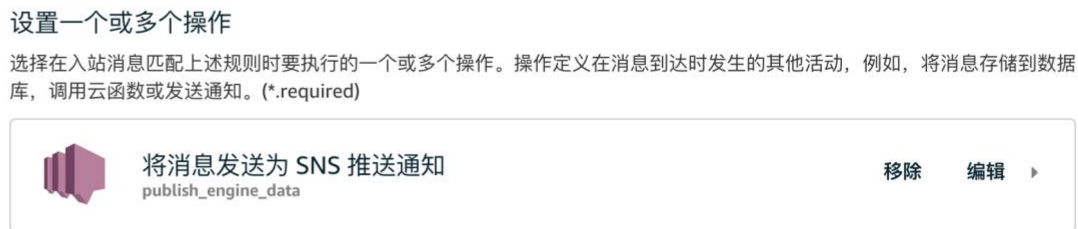


Figure 2-5-1 The created SNS rule

Then set up Amazon SNS to send the messages through your Amazon SNS topic to your email inbox. On the AWS navigation bar, choose Services. In the Find a service by name or feature box, enter SNS, Select the check box next to 'publish_engine_data' topic. For Actions, choose Subscribe to topic. Topics list with the Subscribe to topic action selected. In the Create subscription dialog box, for Protocol, choose Email. For Endpoint, I enter my nus email address. Choose Create subscription.

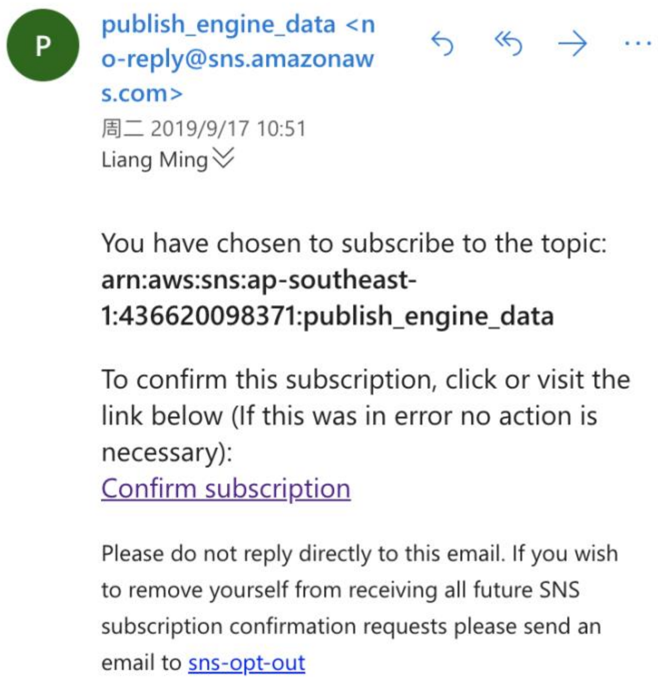


Figure 2-5-2 Email subscription

2.2 publishing pre-defined engine data to AWS

After setting done, let’s try to send data trainFD001 to AWS platform.

2.2.1 Create a new Dynamo Table named after my matric number

Find "DynanmoDB" in the "service" drop-down menu in the AWS navigation and create a new Dynamo table.

Figure 2-2-1.1 choose the DynamoDB Table

The primary key of this table created consists of ‘id’ as partition key and ‘timestamp’ as sort key:

Figure 2-2-1.2 Create a new Dynamo Table

2.2.2 Read and publish data from train_FD001.

According to the requirement, besides overwrite the data FD001, we also need to convert the data type from a string to json so that the DDB table can accept.

The modified code is as follows:

```
from AWSIoTPythonSDK.MQTTLib import AWSIoTMQTTShadowClient
import random
import time
from datetime import datetime
import pandas as pd
import json

# A random programmatic shadow client ID.
SHADOW_CLIENT = "A0195025H"
HOST_NAME = "aa82s646w3yp3-ats.iot.ap-southeast-1.amazonaws.com"
ROOT_CA = "/Users/frankie/Downloads/EE module/Semester2/"\
          "Selected Topics in Industrial Control/5111 project cert"\
          "/AmazonRootCA1.pem"
PRIVATE_KEY = "/Users/frankie/Downloads/EE module/Semester2/"\
              "Selected Topics in Industrial Control/5111 project cert/"\
              "ab7a8d9e50-private.pem.key"

CERT_FILE = "/Users/frankie/Downloads/EE module/Semester2/"\
            "Selected Topics in Industrial Control/5111 project cert/"\
            "ab7a8d9e50-certificate.pem.crt"
SHADOW_HANDLER = "A0195025H"
def myShadowUpdateCallback(payload, responseStatus, token):
    print()
    print('UPDATE: $aws/things/' + SHADOW_HANDLER + '/shadow/update/#')
    print("payload = " + payload)
```

```
    print("responseStatus = " + responseStatus)
    print("token = " + token)

# Create configure and connect a shadow client
myShadowClient = AWSIoTMQTTShadowClient(SHADOW_CLIENT)
myShadowClient.configureEndpoint(HOST_NAME, 8883)
myShadowClient.configureCredentials(ROOT_CA, PRIVATE_KEY,CERT_FILE)
myShadowClient.configureConnectDisconnectTimeout(10)
myShadowClient.configureMQTTOperationTimeout(5)
myShadowClient.connect()

# Create a programmatic representation of the shadow
myDeviceShadow = myShadowClient.createShadowHandlerWithName(
    SHADOW_HANDLER, True)
print('Program start!')
#  The following part is for overwriting column 'id' and add
#  column timestamp column UTC
df = pd.read_csv("/Users/frankie/Downloads/EE module/Semester2/Selected"\
                 " Topics in Industrial Control/NUS_guest_lecture-master/input/train_FD001.txt"
                 ,delim_whitespace = True,header = None)
sensor_number = ['s' + str(i) for i in range(1,22)]
columns_list = ['id','time','os1','os2','os3'] + sensor_number
df = pd.DataFrame(df.values,columns = columns_list)
df[['id','time']].astype('int')
df['id'] = df['id'].map(lambda s: 'FD001_'+str(s))
nums, dimsm = df.shape

for i in range(nums):
    tmp = df.iloc[i]
    now = datetime.utcnow()
    tmp = tmp.append(pd.Series(['A0195025H', str(now)], index=['MatricID', 'timestamp']))
    tmp = tmp.to_dict()
#     jsonPayload = {"state": {"reported": tmp}}
    jsonPayload = json.dumps(tmp)
    print(jsonPayload)
    myDeviceShadow.shadowUpdate(str(jsonPayload),myShadowUpdateCallback, 5)

    time.sleep(0.2)
```

Figure 2.2.2-1: The publishing code

2.2.3 Run this jupyter notebook and check the AWS DynamoDB Table is set correctly

```python
myShadowClient.connect()

# Create a programmatic representation of the shadow
myDeviceShadow = myShadowClient.createShadowHandlerWithName(
    SHADOW_HANDLER, True)
print('Program start!')
#  The following part is for overwriting column 'id' and add
#  column timestamp column UTC
df = pd.read_csv("/Users/frankie/Downloads/EE module/Semester2/Selected"\
                 " Topics in Industrial Control/NUS_guest_lecture-master/input/train_FD001.txt"
                 ,delim_whitespace = True,header = None)
sensor_number = ['s' + str(i) for i in range(1,22)]
columns_list = ['id','time','os1','os2','os3'] + sensor_number
df = pd.DataFrame(df.values,columns = columns_list)
df[['id','time']].astype('int')
df['id'] = df['id'].map(lambda s: 'FD001_'+str(s))
nums, dimsm = df.shape

for i in range(nums):
    tmp = df.iloc[i]
    now = datetime.utcnow()
    tmp = tmp.append(pd.Series(['A0195025H', str(now)], index=['MatricID', 'timestamp']))
    tmp = tmp.to_dict()
#     jsonPayload = {"state": {"reported": tmp}}
    jsonPayload = json.dumps(tmp)
    print(jsonPayload)
    myDeviceShadow.shadowUpdate(str(jsonPayload),myShadowUpdateCallback, 5)
    time.sleep(0.2)
```

```
Program start!
{"id": "FD001_1.0", "time": 1.0, "os1": -0.0007, "os2": -0.0004, "os3": 100.0, "s1": 518.67, "s2": 641.82, "s3": 158
9.7, "s4": 1400.6, "s5": 14.62, "s6": 21.61, "s7": 554.36, "s8": 2388.06, "s9": 9046.19, "s10": 1.3, "s11": 47.47, "s
12": 521.66, "s13": 2388.02, "s14": 8138.62, "s15": 8.4195, "s16": 0.03, "s17": 392.0, "s18": 2388.0, "s19": 100.0,
"s20": 39.06, "s21": 23.419, "MatricID": "A0195025H", "timestamp": "2019-09-18 02:29:23.169981"}

UPDATE: $aws/things/A0195025H/shadow/update/#
payload = {"code":400,"message":"Missing required node: state","clientToken":"e300a880-73dd-4861-aa52-0a7060ee9811"}
responseStatus = rejected
token = e300a880-73dd-4861-aa52-0a7060ee9811
{"id": "FD001_1.0", "time": 2.0, "os1": 0.0019, "os2": -0.0003, "os3": 100.0, "s1": 518.67, "s2": 642.15, "s3": 1591.
82, "s4": 1403.14, "s5": 14.62, "s6": 21.61, "s7": 553.75, "s8": 2388.04, "s9": 9044.07, "s10": 1.3, "s11": 47.49, "s
12": 522.28, "s13": 2388.07, "s14": 8131.49, "s15": 8.4318, "s16": 0.03, "s17": 392.0, "s18": 2388.0, "s19": 100.0,
"s20": 39.0, "s21": 23.4236, "MatricID": "A0195025H", "timestamp": "2019-09-18 02:29:25.452911"}

UPDATE: $aws/things/A0195025H/shadow/update/#
payload = {"code":400,"message":"Missing required node: state","clientToken":"bf3d6a73-3416-48cc-b265-218472735ffe"}
responseStatus = rejected
token = bf3d6a73-3416-48cc-b265-218472735ffe
```

Figure: 2.2.3-1: Running the jupyter notebook

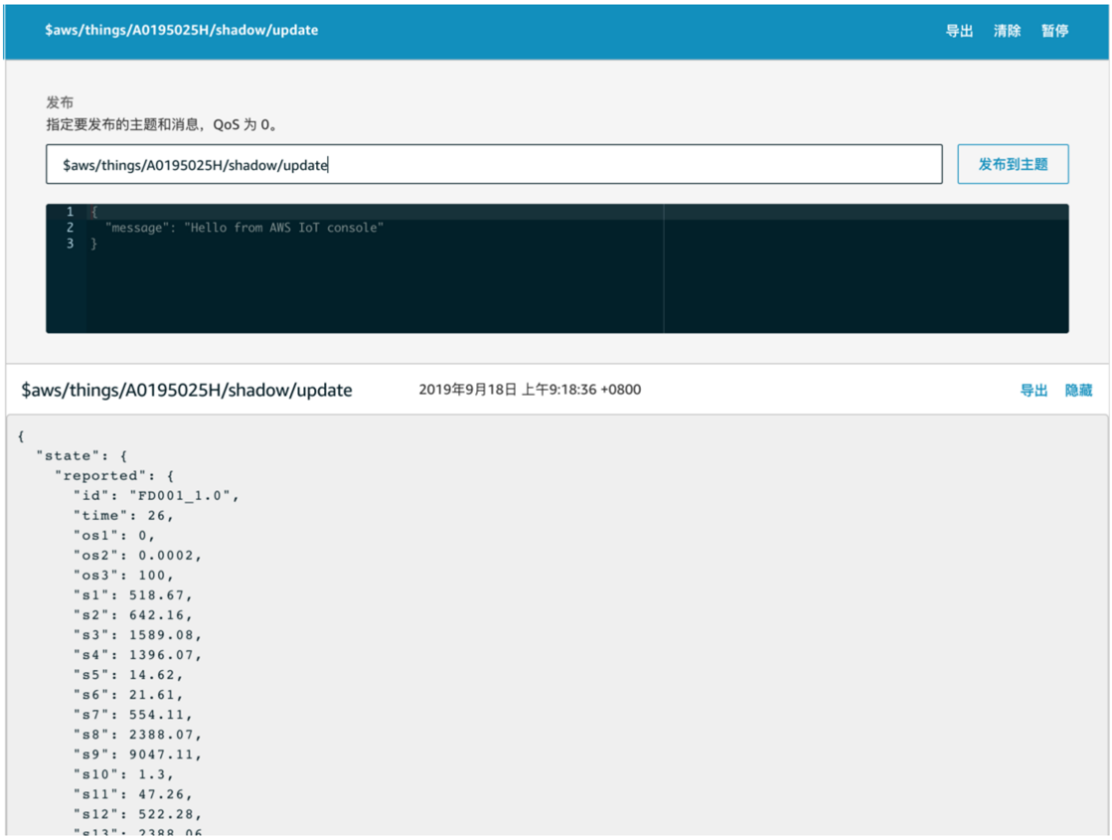
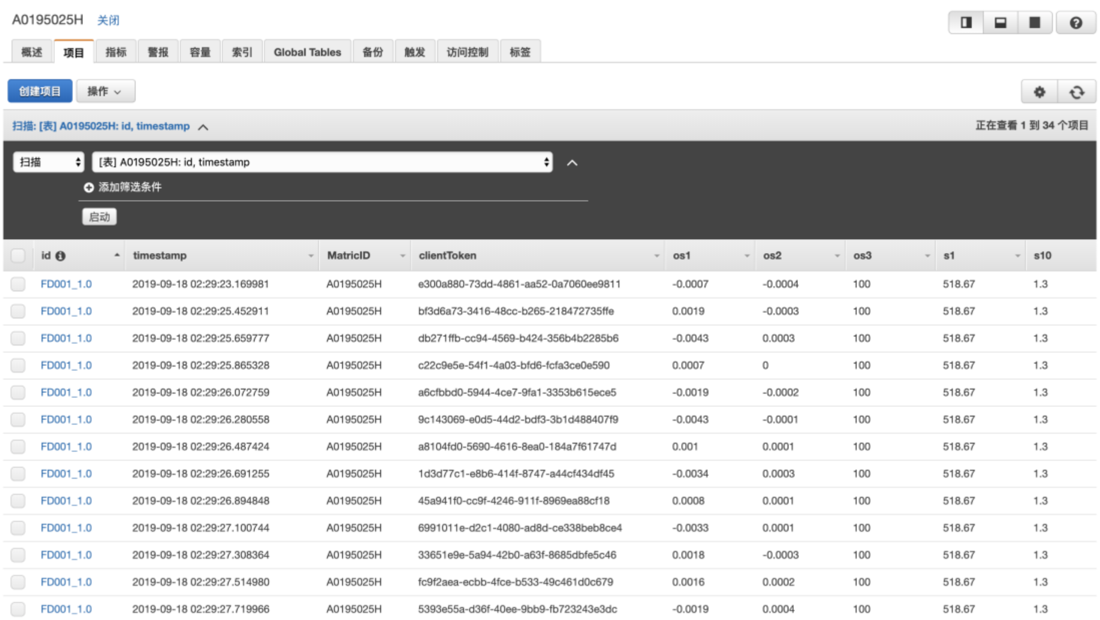


Figure 2.2.3-2: check if the AWS platform can receive data



id	timestamp	MatricID	clientToken	os1	os2	os3	s1	s10
FD001_1.0	2019-09-18 02:29:23.169981	A0195025H	e300a880-73dd-4861-aa52-0a7060ee9811	-0.0007	-0.0004	100	518.67	1.3
FD001_1.0	2019-09-18 02:29:25.452911	A0195025H	bf3d6a73-3416-48cc-b265-218472735ffe	0.0019	-0.0003	100	518.67	1.3
FD001_1.0	2019-09-18 02:29:25.659777	A0195025H	db271ffb-cc94-4569-b424-356b4b2285b6	-0.0043	0.0003	100	518.67	1.3
FD001_1.0	2019-09-18 02:29:25.865328	A0195025H	c22c9e5e-54f1-4a03-bfd6-fcfa3ce0e590	0.0007	0	100	518.67	1.3
FD001_1.0	2019-09-18 02:29:26.072759	A0195025H	a6cfbbd0-5944-4ce7-9fa1-3353b615ece5	-0.0019	-0.0002	100	518.67	1.3
FD001_1.0	2019-09-18 02:29:26.280558	A0195025H	9c143069-e0d5-44d2-bdf3-3b1d488407f9	-0.0043	-0.0001	100	518.67	1.3
FD001_1.0	2019-09-18 02:29:26.487424	A0195025H	a8104fd0-5690-4616-8ea0-184a7f61747d	0.001	0.0001	100	518.67	1.3
FD001_1.0	2019-09-18 02:29:26.691255	A0195025H	1d3d77c1-e8b6-414f-8747-a44cf434df45	-0.0034	0.0003	100	518.67	1.3
FD001_1.0	2019-09-18 02:29:26.894848	A0195025H	45a941f0-cc9f-4246-911f-8969ea88cf18	0.0008	0.0001	100	518.67	1.3
FD001_1.0	2019-09-18 02:29:27.100744	A0195025H	6991011e-d2c1-4080-ad8d-ce338beb8ce4	-0.0033	0.0001	100	518.67	1.3
FD001_1.0	2019-09-18 02:29:27.308364	A0195025H	33651e9e-5a94-42b0-a63f-8685dbfe5c46	0.0018	-0.0003	100	518.67	1.3
FD001_1.0	2019-09-18 02:29:27.514980	A0195025H	fc9f2aea-ecbb-4fce-b533-49c461d0c679	0.0016	0.0002	100	518.67	1.3
FD001_1.0	2019-09-18 02:29:27.719966	A0195025H	5393e55a-d36f-40ee-9bb9-fb723243e3dc	-0.0019	0.0004	100	518.67	1.3

Figure2.2.3-3: check if data is loaded to DynamoDB

The whole uploading procedure lasts about one hours. All the data is loaded to the DynamoDB box.

2.3 Simulate two IoT things running simultaneously

2.3.1 Add one more thing and one more certificate

Create another thing named A0195025H_2 under AWS IoT platform and one more certificate. Create a copy of the Jupyter notebook above, renaming the client name, certificate, data source train_FD002.txt.Modify the rules to be triggered by other condition:

SELECT * FROM '$aws/things/+/shadow/update'

'+' means anything. both the thing in Step2 and in this step. Now, the rule should help to push data from both things.

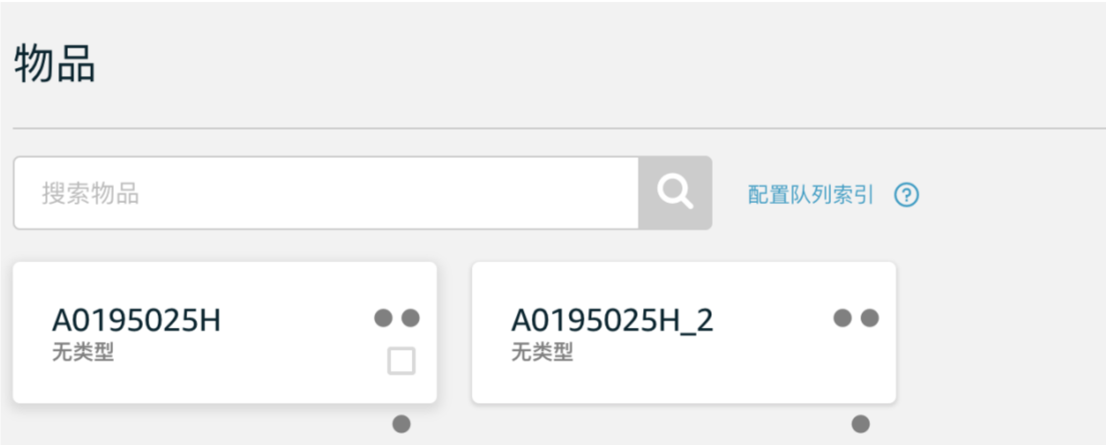


Figure 2.3.1-1: Create a new thing 'A0195025H_2'

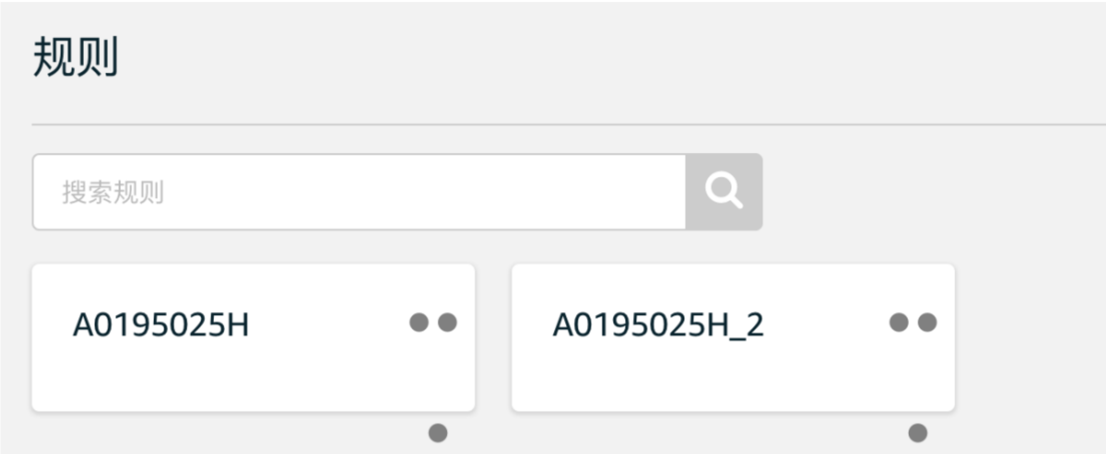


Figure 2.3.1-2: Create a new rule

2.3.2 Running two Jupyter notebook together

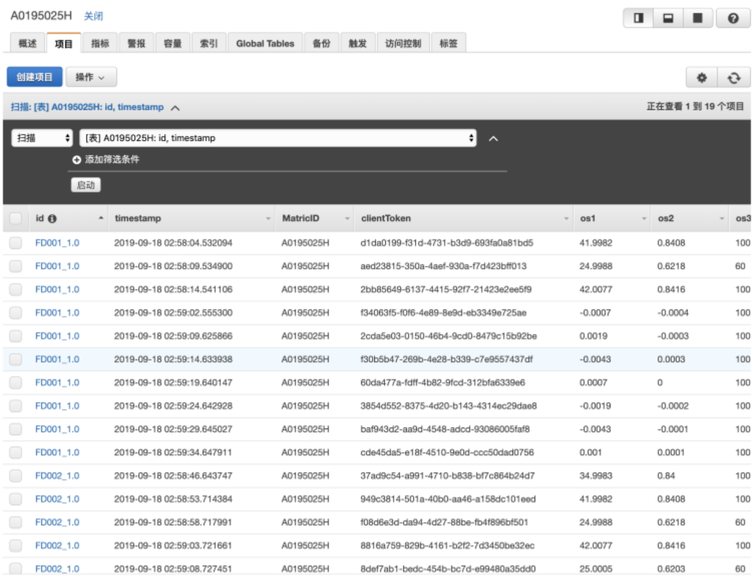
If we want to load data train002 to the DynamoDB, we need to modify the code some of the jupyter notebook we use before. Both the root of the certificate and client ID are supposed to be changed. In addition to that, in this step, we also rise the time Interval for loading data from 0.2s to 5s so that the data from train001 and train002 is easy to distinguish in DynamoDB.The modified code is as follow:

```python
from AWSIoTPythonSDK.MQTTLib import AWSIoTMQTTShadowClient
import random
import time
from datetime import datetime
import pandas as pd
import json

# A random programmatic shadow client ID.
SHADOW_CLIENT = "A0195025H_2"
HOST_NAME = "aa82s646w3yp3-ats.iot.ap-southeast-1.amazonaws.com"
ROOT_CA = "/Users/frankie/Downloads/EE module/Semester2/"\
          "Selected Topics in Industrial Control/A0195025H_2/"\
          "AmazonRootCA1.pem"
PRIVATE_KEY = "/Users/frankie/Downloads/EE module/Semester2/"\
              "Selected Topics in Industrial Control/A0195025H_2/"\
              "16e02f8b9a-private.pem.key"

CERT_FILE = "/Users/frankie/Downloads/EE module/Semester2/"\
            "Selected Topics in Industrial Control/A0195025H_2/"\
            "16e02f8b9a-certificate.pem.crt"
SHADOW_HANDLER = "A0195025H_2"
def myShadowUpdateCallback(payload, responseStatus, token):
    print()
    print('UPDATE: $aws/things/' + SHADOW_HANDLER + '/shadow/update/#')
    print("payload = " + payload)
    print("responseStatus = " + responseStatus)
    print("token = " + token)

# Create configure and connect a shadow client
myShadowClient = AWSIoTMQTTShadowClient(SHADOW_CLIENT)
myShadowClient.configureEndpoint(HOST_NAME, 8883)
myShadowClient.configureCredentials(ROOT_CA, PRIVATE_KEY,CERT_FILE)
myShadowClient.configureConnectDisconnectTimeout(10)
myShadowClient.configureMQTTOperationTimeout(5)
myShadowClient.connect()

# Create a programmatic representation of the shadow
myDeviceShadow = myShadowClient.createShadowHandlerWithName(
    SHADOW_HANDLER, True)
print('Program start!')
#  The following part is for overwriting column 'id' and add
#  column timestamp column UTC
```

```
df = pd.read_csv("/Users/frankie/Downloads/EE module/Semester2/Selected"\
                 " Topics in Industrial Control/NUS_guest_lecture-master/input/train_FD002.txt"
                 ,delim_whitespace = True,header = None)
sensor_number = ['s' + str(i) for i in range(1,22)]
columns_list = ['id','time','os1','os2','os3'] + sensor_number
df = pd.DataFrame(df.values,columns = columns_list)
df[['id','time']].astype('int')
df['id'] = df['id'].map(lambda s: 'FD002_'+str(s))
nums, dimsm = df.shape

for i in range(nums):
    tmp = df.iloc[i]
    now = datetime.utcnow()
    tmp = tmp.append(pd.Series(['A0195025H', str(now)], index=['MatricID', 'timestamp']))
    tmp = tmp.to_dict()
#     jsonPayload = {"state": {"reported": tmp}}
    jsonPayload = json.dumps(tmp)
    print(jsonPayload)
    myDeviceShadow.shadowUpdate(str(jsonPayload),myShadowUpdateCallback, 5)

    time.sleep(5)
```

Figure 2.3.2.1: Code to upload data train002

After loading data train001 and data train002 in Dynamo. Here is the figure:



id	timestamp	MatricID	clientToken	os1	os2	os3
FD001_1.0	2019-09-18 02:58:04.532094	A0195025H	d1da0199-f31d-4731-b3d9-693fa0a81bd5	41.9982	0.8408	100
FD001_1.0	2019-09-18 02:58:09.534900	A0195025H	aed23815-350a-4aef-930a-f7d423bff013	24.9988	0.6218	60
FD001_1.0	2019-09-18 02:58:14.541106	A0195025H	2bb85649-6137-4415-92f7-21423e2ee5f9	42.0077	0.8416	100
FD001_1.0	2019-09-18 02:59:02.555300	A0195025H	f34063f5-f0f6-4e89-8e9d-eb3349e725ae	-0.0007	-0.0004	100
FD001_1.0	2019-09-18 02:59:09.625866	A0195025H	2cda5e03-0150-46b4-9cd0-8479c15b92be	0.0019	-0.0003	100
FD001_1.0	2019-09-18 02:59:14.633938	A0195025H	f30b5b47-269b-4e28-b339-c7e9557437df	-0.0043	0.0003	100
FD001_1.0	2019-09-18 02:59:19.640147	A0195025H	60da477a-fdff-4b82-9fcd-312bfa6339e6	0.0007	0	100
FD001_1.0	2019-09-18 02:59:24.642928	A0195025H	3854d552-8375-4d20-b143-4314ec29dae8	-0.0019	-0.0002	100
FD001_1.0	2019-09-18 02:59:29.645027	A0195025H	baf943d2-aa9d-4548-adcd-93086005faf8	-0.0043	-0.0001	100
FD001_1.0	2019-09-18 02:59:34.647911	A0195025H	cde45da5-e18f-4510-9e0d-ccc50dad0756	0.001	0.0001	100
FD002_1.0	2019-09-18 02:58:46.643747	A0195025H	37ad9c54-a991-4710-b838-bf7c864b24d7	34.9983	0.84	100
FD002_1.0	2019-09-18 02:58:53.714384	A0195025H	949c3814-501a-40b0-aa46-a158dc101eed	41.9982	0.8408	100
FD002_1.0	2019-09-18 02:58:58.717991	A0195025H	f08d6e3d-da94-4d27-88be-fb4f896bf501	24.9988	0.6218	60
FD002_1.0	2019-09-18 02:59:03.721661	A0195025H	8816a759-829b-4161-b2f2-7d3450be32ec	42.0077	0.8416	100
FD002_1.0	2019-09-18 02:59:08.727451	A0195025H	8def7ab1-bedc-454b-bc7d-e99480a35dd0	25.0005	0.6203	60

Figure 2-3-2.2 : Two type of data successfully uploaded to the Dynamo

3. Visualise the data for the two engines

3.1 IAM policy

To download the data in Dynamo for visualizing, we need to use IAM(Identity and Access Management) policy. In the AWS platform, we search the ‘IAM ’ and click ‘Add User’ , which is shown in Figure 3.1.1:

Figure 3.1.1: Create IAM policy

Set Username as my matric number and set both programming access and AWS management console access. The figure is as follows:

Figure 3.1.2: Set Username

Creating the user step by step by the introduction and it shows that the user has been successfully created. Download the file in ‘csv’ format, which is show in Figure 3.1.3:

Figure 3.1.3: Successfully create the user

3.2 DynamoDB data download

First of all, we need to connect the data uploaded in DynamoDB before to the jupyter notebook. Here is the code:

```
import gc
import json
import boto3
import decimal
import pandas as pd
import rope
import numpy as np

class DecimalEncoder(json.JSONEncoder):
    def default(self, o):
        if isinstance(o, decimal.Decimal):
            if o%1>0:
                return float(o)
            else:
                return int(o)
        return super(DecimalEncoder, self).default(o)

AWS_ACCESS_ID = 'AKIAWLKEV7NBYI65MJXE'
AWS_ACCESS_KEY = '62ejyESha6McA6yrsyK8c+ujalk5c3hSky4LTKOQ'
client = boto3.client('dynamodb', region_name='ap-southeast-1',
```

```
                      aws_access_key_id=AWS_ACCESS_ID,
                      aws_secret_access_key=AWS_ACCESS_KEY)
dynamodb = boto3.resource("dynamodb", region_name='ap-southeast-1',
                          aws_access_key_id=AWS_ACCESS_ID,
                          aws_secret_access_key=AWS_ACCESS_KEY)
table = dynamodb.Table('A0195025H')
response = table.scan()
items = []

for i in response['Items']:
    items.append(json.dumps(i, cls=DecimalEncoder))
while 'LastEvaluatedKey' in response:
    response = table.scan(ExclusiveStartKey=response['LastEvaluatedKey'])

    for i in response['Items']:
        items.append(json.dumps(i, cls=DecimalEncoder))

data = pd.DataFrame()
i = 1
for line in items:
    data_list = json.loads(line)
    data_df = pd.DataFrame(data_list, index=[i])
    data = data.append(data_df)
    i = i+1

order = ['MatricID','id','timestamp','time','os1','os2','os3']\
        + ['s' + str(i) for i in range(1,22)]
data = data[order]
data.head(10)
```

In the code, Access_ID and Access_KEY are from the file csv we saved before. Figure 3.2.1 shows some data loaded in jupyter notebook

Out[1]:

	MatricID	id	timestamp	time	os1	os2	os3	s1	s2	s3
1	A0195025H	FD002_231.0	2019-09-18 10:41:29.880280	1	0.0027	0.0000	100	518.67	642.72	1591.10
2	A0195025H	FD002_231.0	2019-09-18 10:41:30.192301	2	0.0008	0.0008	100	518.67	642.07	1585.08
3	A0195025H	FD002_231.0	2019-09-18 10:41:30.498113	3	0.0004	0.0000	100	518.67	642.15	1582.05
4	A0195025H	FD002_231.0	2019-09-18 10:41:30.806861	4	42.0061	0.8413	100	445.00	549.45	1345.00
5	A0195025H	FD002_231.0	2019-09-18 10:41:31.116849	5	41.9984	0.8400	100	445.00	549.35	1345.72
6	A0195025H	FD002_231.0	2019-09-18 10:41:31.425955	6	41.9985	0.8410	100	445.00	549.54	1351.82
7	A0195025H	FD002_231.0	2019-09-18 10:41:31.735067	7	0.0016	0.0013	100	518.67	642.65	1582.68
8	A0195025H	FD002_231.0	2019-09-18 10:41:32.043045	8	20.0018	0.7009	100	491.19	607.37	1481.81
9	A0195025H	FD002_231.0	2019-09-18 10:41:32.350957	9	20.0014	0.7018	100	491.19	607.01	1481.25
10	A0195025H	FD002_231.0	2019-09-18 10:41:32.663244	10	42.0008	0.8411	100	445.00	548.34	1354.13

Figure 3.2.1: 10 rows of data

However, there is a problem that data is mixed with FD001 and FD002, so we need to divide it.

In [2]:

```
t = data.copy()
#type(t)
from statsmodels.tsa.stattools import adfuller
import matplotlib.pyplot as plt
from statsmodels.graphics.tsaplots import plot_acf, plot_pacf
#Split the data of the first engine and the second engine
fd1=t.loc[t['id'].str.contains('FD001')]
fd1 = fd1.reset_index()
fd1.drop(['index'],axis=1,inplace=True)
fd2=t.loc[t['id'].str.contains('FD002')]
fd2 = fd2.reset_index()
fd2.drop(['index'],axis=1,inplace=True)
print('Number of cycles in fd001: ', fd1['id'].unique().size)
print('Number of cycles in fd002: ', fd2['id'].unique().size)
```

```
Number of cycles in fd001:  100
Number of cycles in fd002:  260
```

Figure 3.2.2: Divide mixed data.

3.3 Compare two engine life

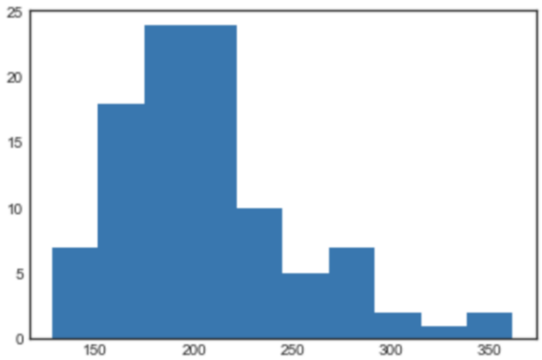
Here is the code and result for each engine average lifespan

```
In [12]: h1= fd1.reset_index().groupby(['id'])['time'].idxmax()
         #g1
         h2= fd2.reset_index().groupby(['id'])['time'].idxmax()
         #g2
         time1=fd1.iloc[h1]
         time2=fd2.iloc[h2]
         #te1
         his1=time1.loc[:,"time"]
         his2=time2.loc[:,"time"]
         print('Average life of fd001: ', his1.mean())
         print('Average life of fd001: ', his2.mean())

         Average life of fd001:  206.06
         Average life of fd001:  206.6769230769231
```

Figure 3.2.3: average lifespan

According to the result, we can see that the lifespan is very closed. To analysis more specifically, we draw the histogram of each engine. The code and result is as follows:

```
In [15]: import matplotlib.pyplot as plt
         plt.style.use('seaborn-white')
         #The most basic frequency histogram command
         plt.figure(1)
         plt.hist(his1)
         plt.figure(2)
         plt.hist(his2)

Out[15]: (array([25., 49., 68., 50., 25., 19., 12.,  6.,  4.,  2.]),
          array([128., 153., 178., 203., 228., 253., 278., 303., 328., 353., 37
         8.]),
          <a list of 10 Patch objects>)
```



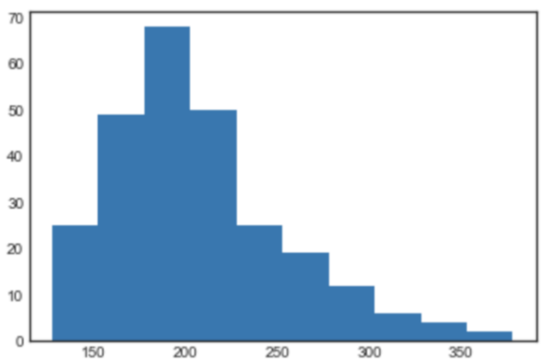


Figure 3.3.3: histogram of lifespan

According to the histogram, we can see that the first half of the two histograms are very similar. However, after the magnitude start to decrease, we can see the difference between two histograms. The histogram of FD001 decline with fluctuation, whereas the other keeps decreasing.

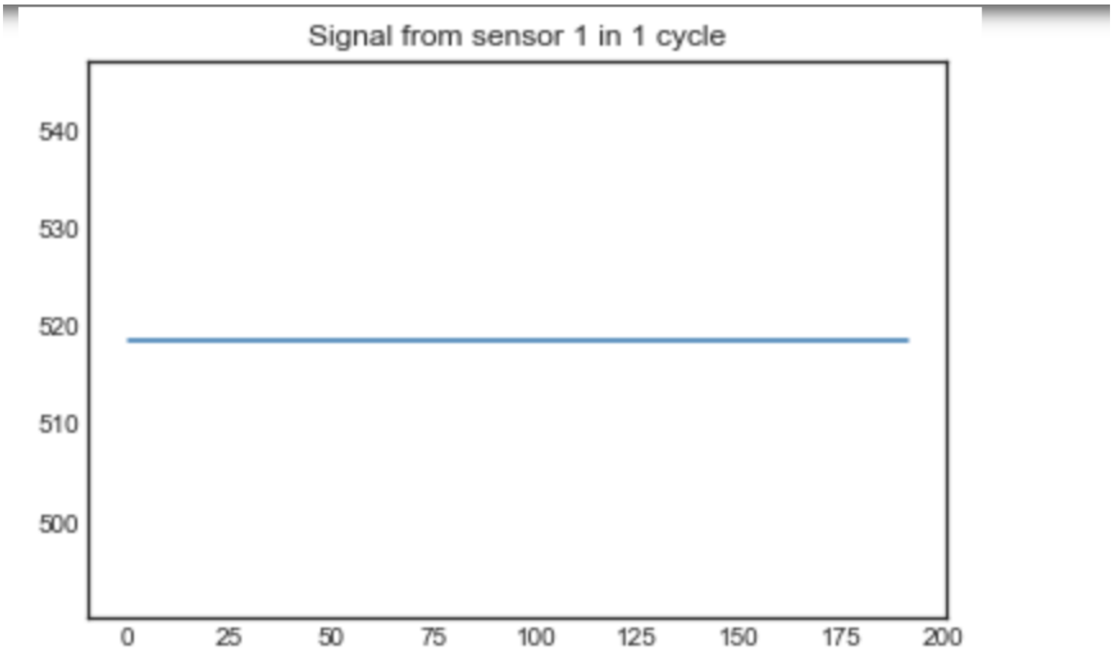
From what has been discussed above, we can get a conclusion that although the average lifespan is closed, the data in FD002 is better becaue its steady tendency.

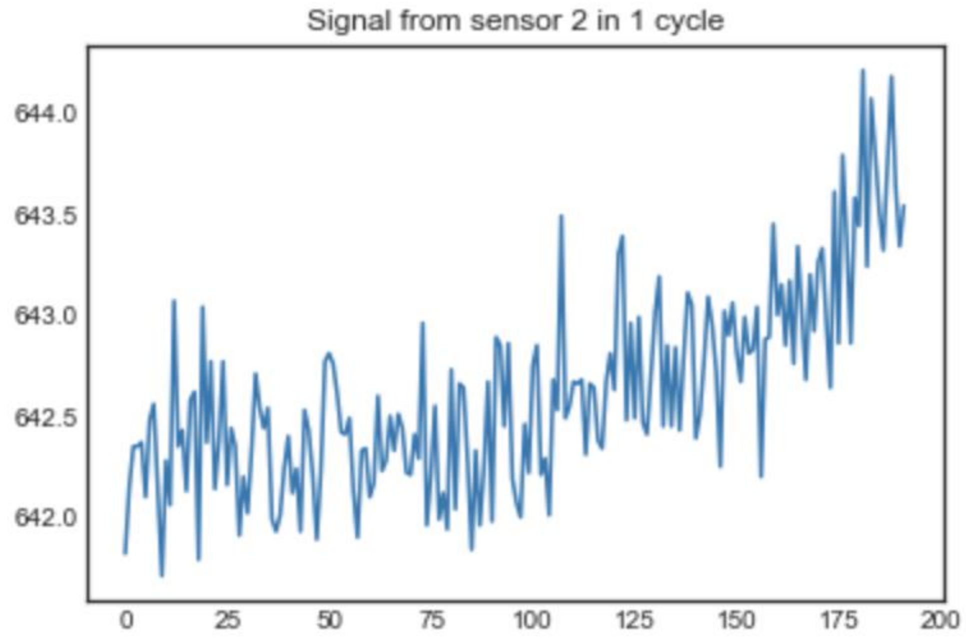
3.4 data visualization

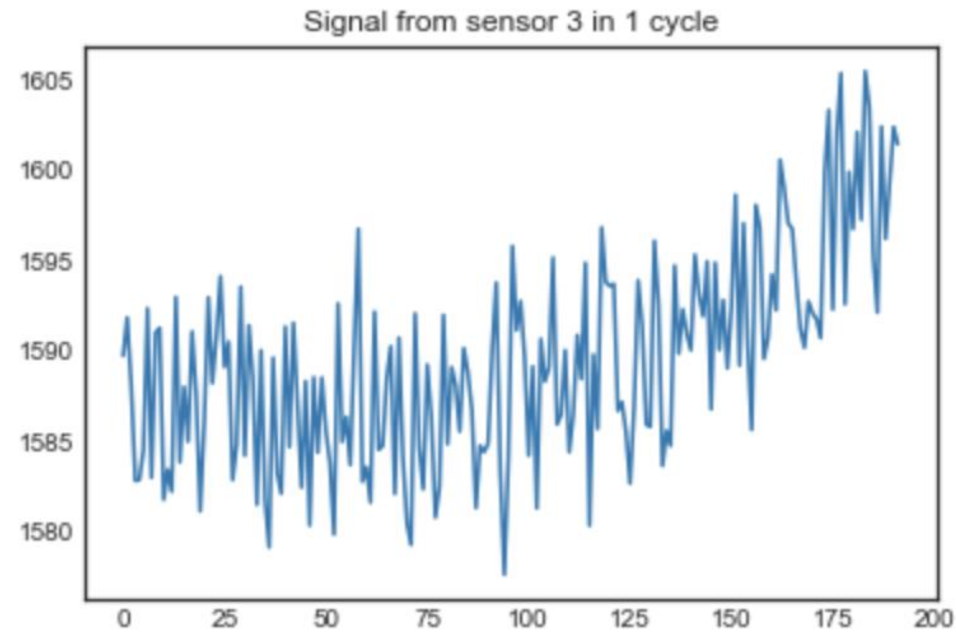
In this section, we check the variation tendency for 21 sensor in the first cycle. The code and images are as follows:

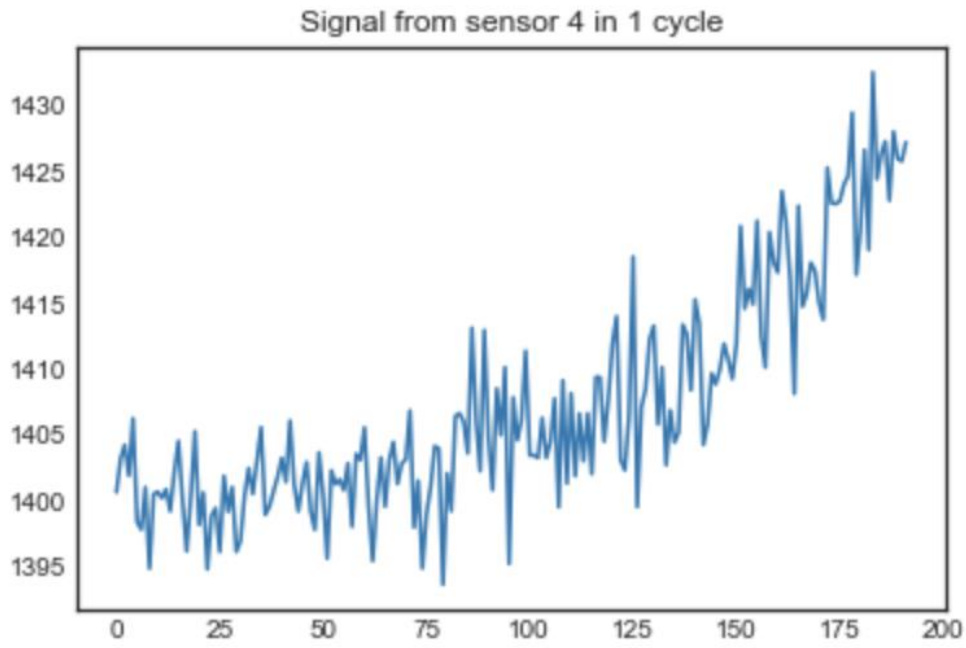
```
sensors = locals()
for i in range(1,22):
    sensors['s' + str(i) ] = fd1.loc[:,'s'+str(i)]
for i in range(1,22):
    sensors['s' + str(i)+'_c1'+str(i)] = fd1.loc[fd1['id']=="FD001_1.0",'s'+str(i)].reset_index()
    sensors['s' + str(i)+'_c1'+str(i)].drop(['index'],axis=1,inplace=True)
    plt.figure(i)
    plt.plot(sensors['s' + str(i)+'_c1'+str(i)])
    plt.title('Signal from sensor %d in 1 cycle'%i)
```

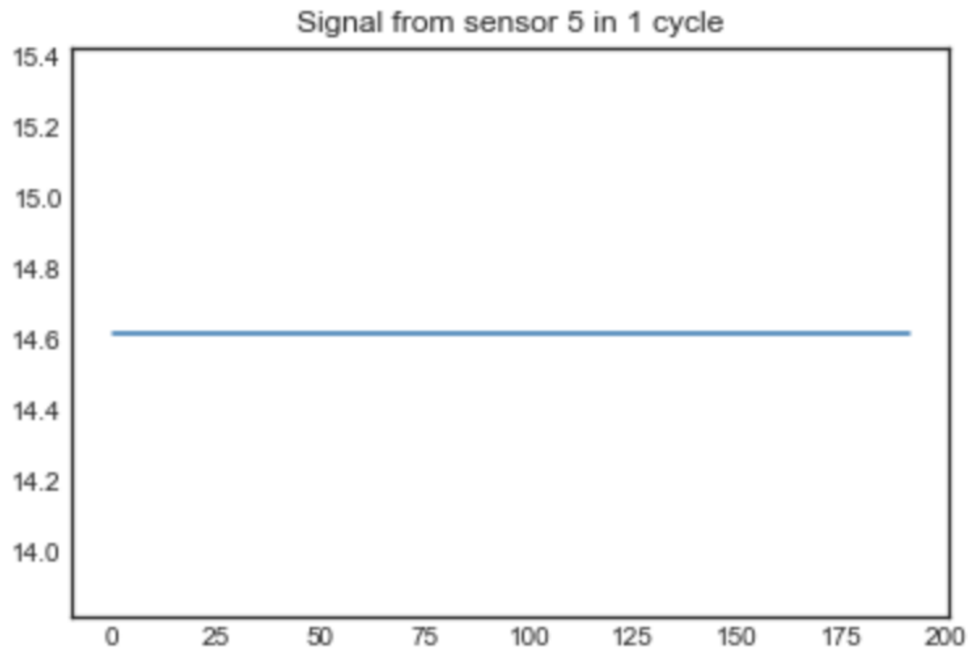
Figure 3.4.1: picture code

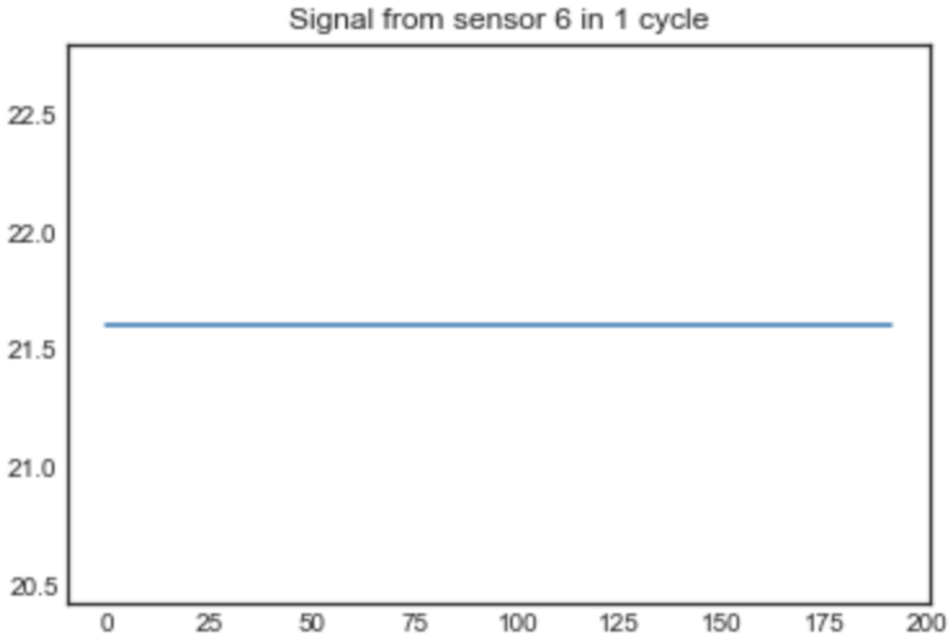


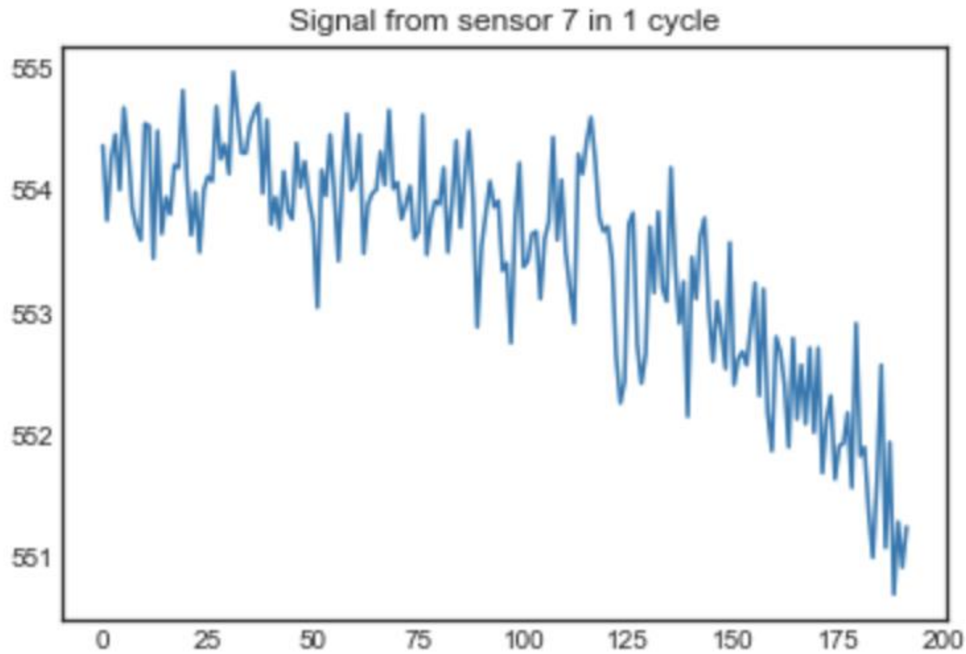

Signal from sensor 2 in 1 cycle
644.0
643.5
643.0
642.5
642.0
0
25
50
75
100
125
150
175
200

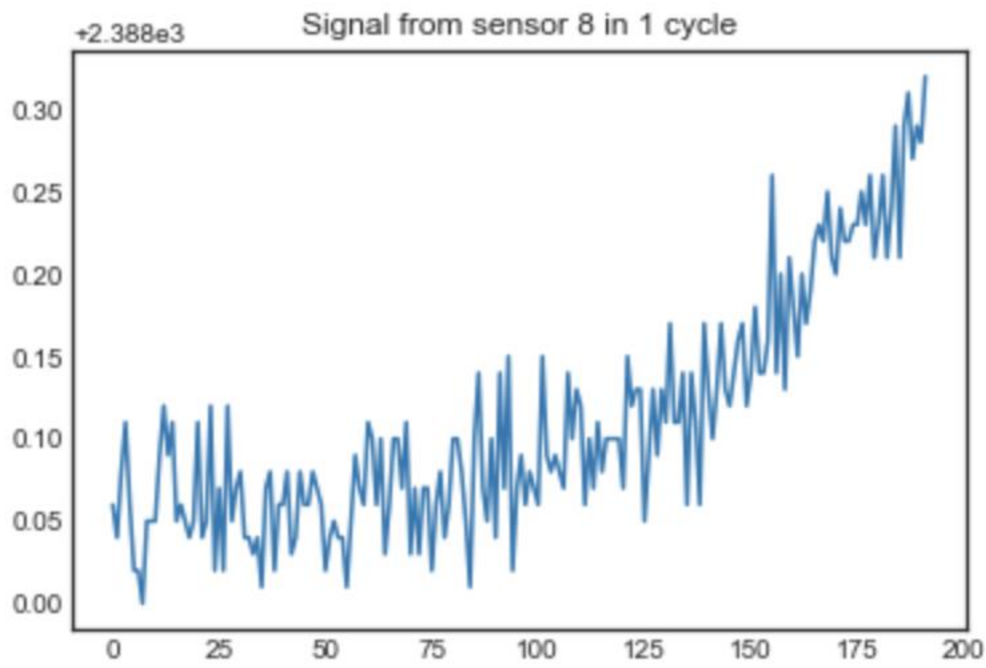

Signal from sensor 3 in 1 cycle
1605
1600
1595
1590
1585
1580
0
25
50
75
100
125
150
175
200


Signal from sensor 4 in 1 cycle
1430
1425
1420
1415
1410
1405
1400
1395
0
25
50
75
100
125
150
175
200

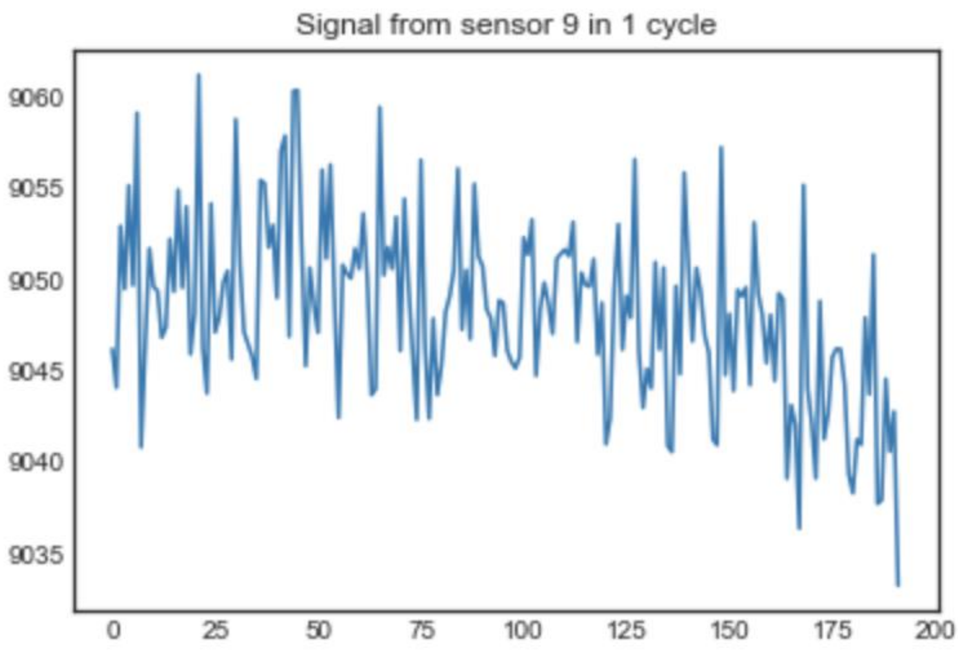

Signal from sensor 5 in 1 cycle
15.4
15.2
15.0
14.8
14.6
14.4
14.2
14.0
0
25
50
75
100
125
150
175
200


Signal from sensor 6 in 1 cycle
22.5
22.0
21.5
21.0
20.5
0
25
50
75
100
125
150
175
200

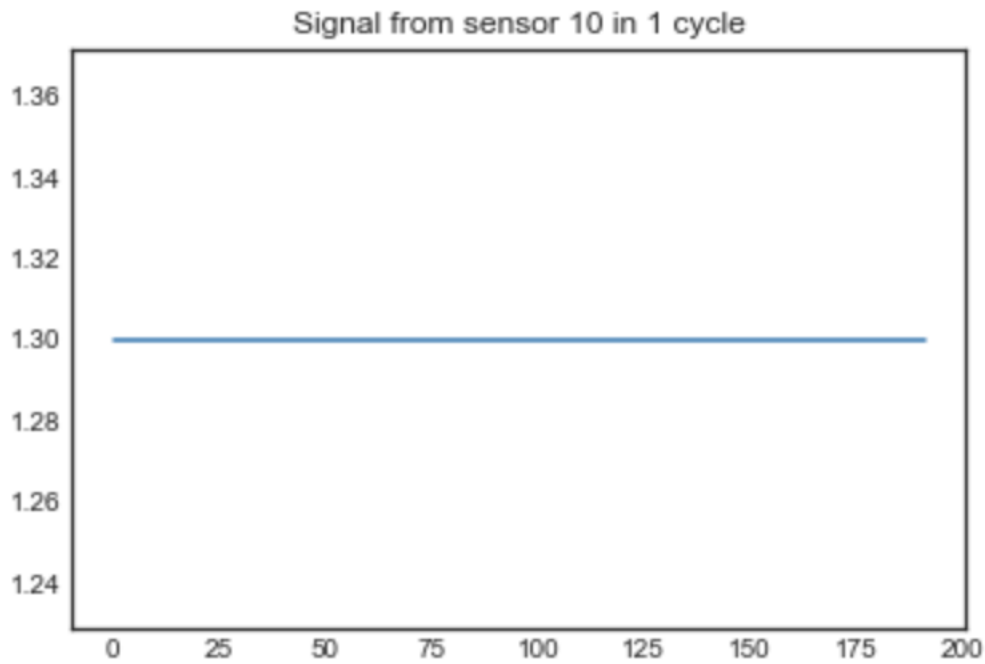

Signal from sensor 7 in 1 cycle
555
554
553
552
551
0
25
50
75
100
125
150
175
200

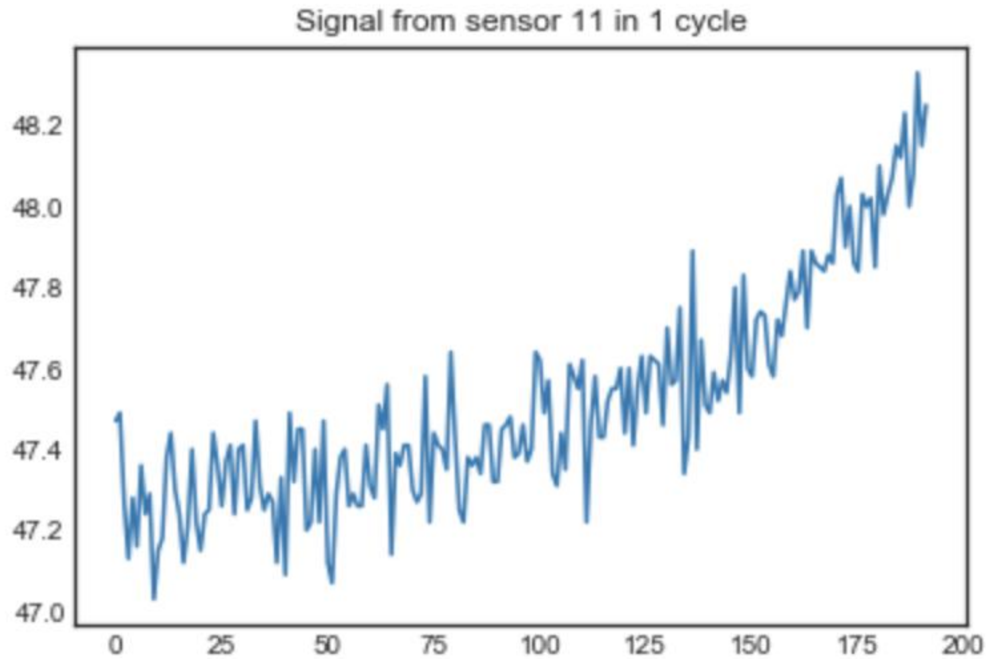


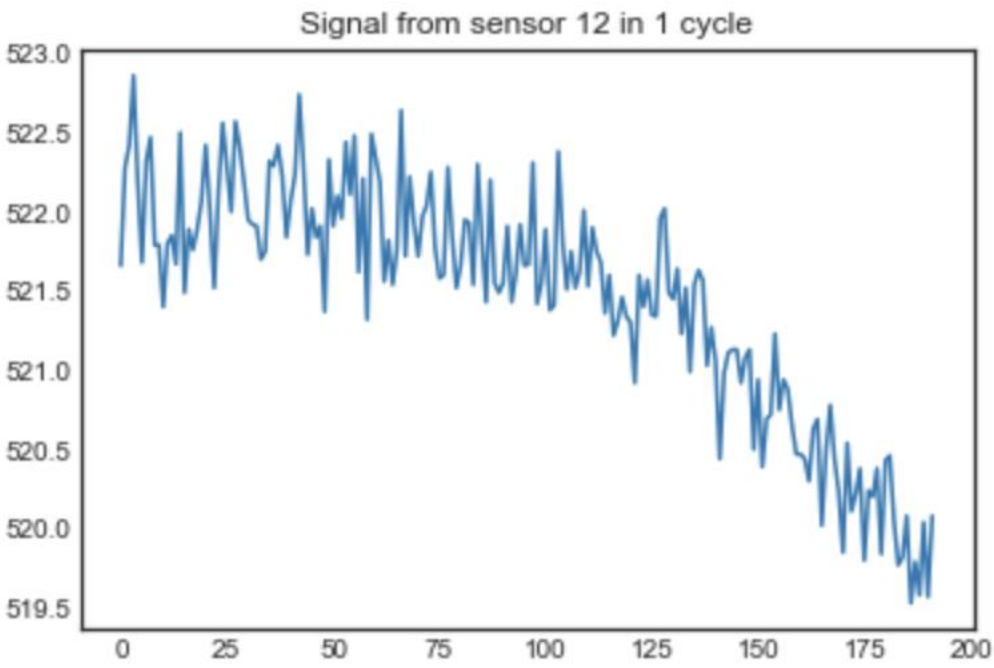
Signal from sensor 8 in 1 cycle
+2.388e3
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
25
50
75
100
125
150
175
200

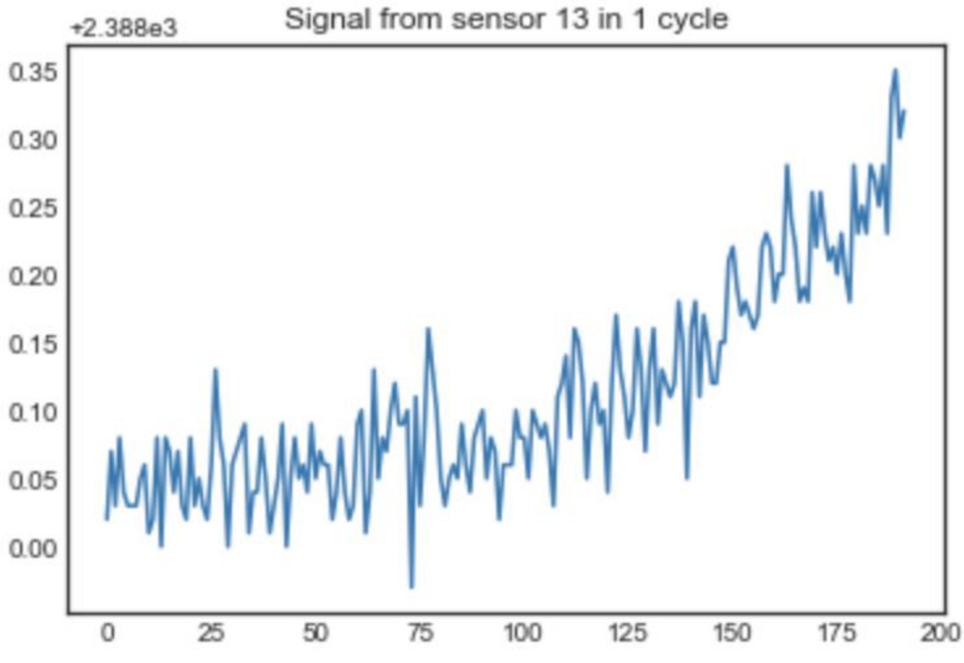


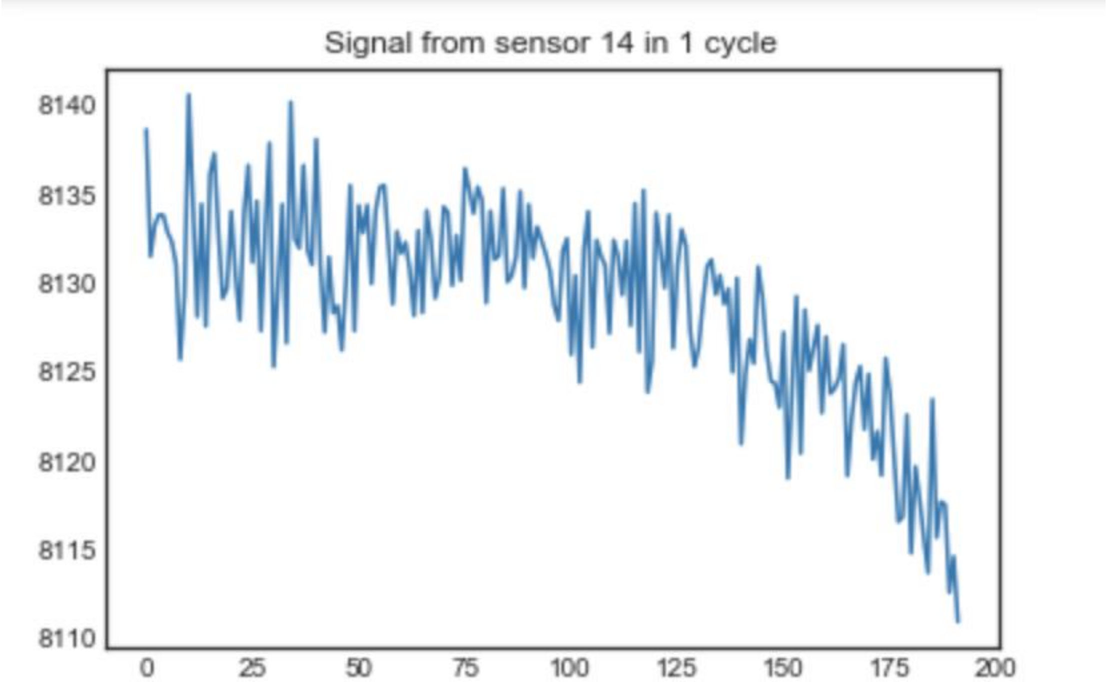
Signal from sensor 9 in 1 cycle
9060
9055
9050
9045
9040
9035
0
25
50
75
100
125
150
175
200

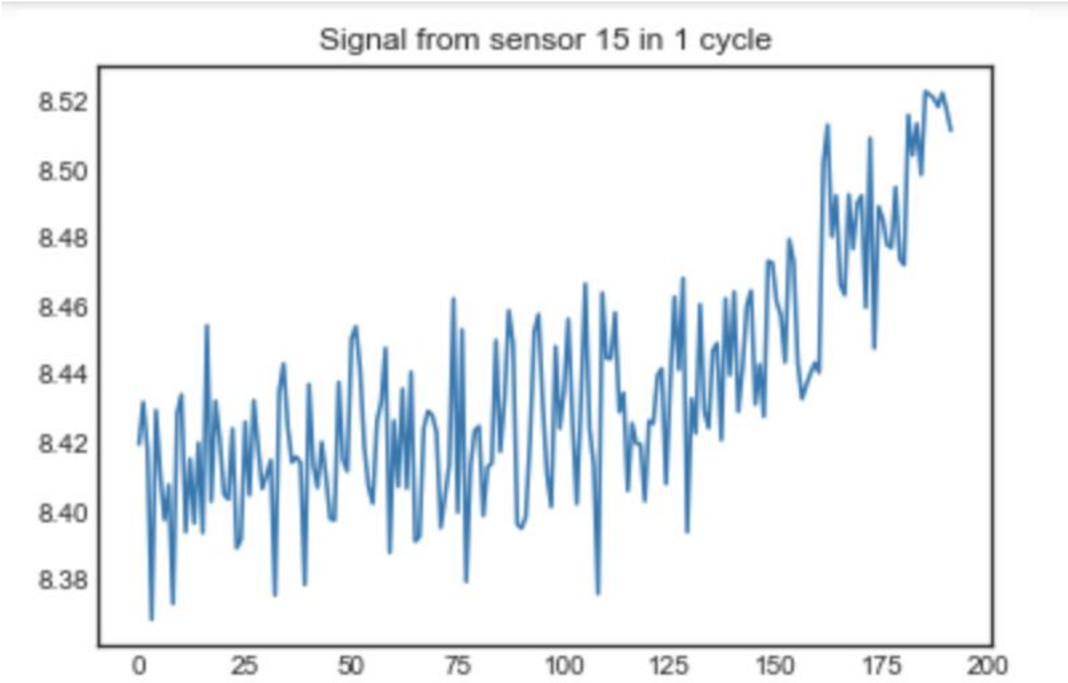

Signal from sensor 10 in 1 cycle
1.36
1.34
1.32
1.30
1.28
1.26
1.24
0
25
50
75
100
125
150
175
200

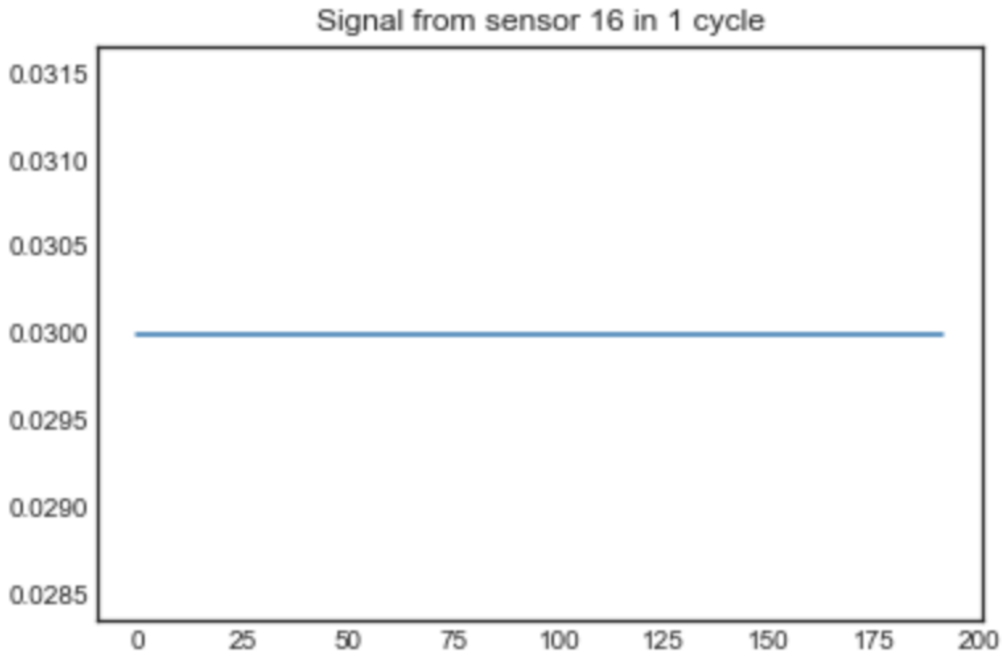

Signal from sensor 11 in 1 cycle
48.2
48.0
47.8
47.6
47.4
47.2
47.0
0
25
50
75
100
125
150
175
200

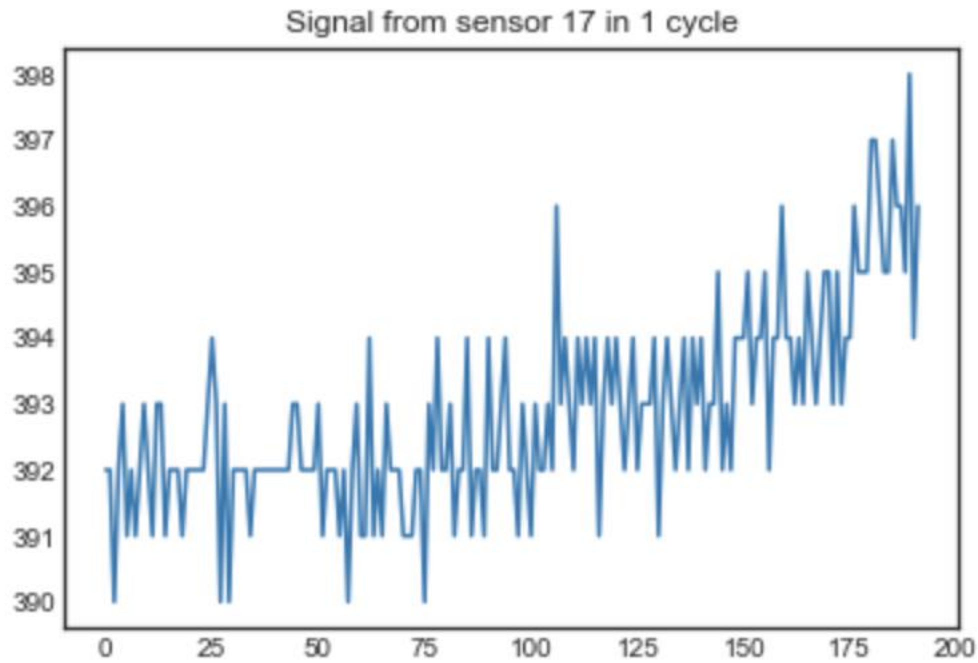

Signal from sensor 12 in 1 cycle
523.0
522.5
522.0
521.5
521.0
520.5
520.0
519.5
0
25
50
75
100
125
150
175
200


Signal from sensor 13 in 1 cycle
+2.388e3
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
0
25
50
75
100
125
150
175
200

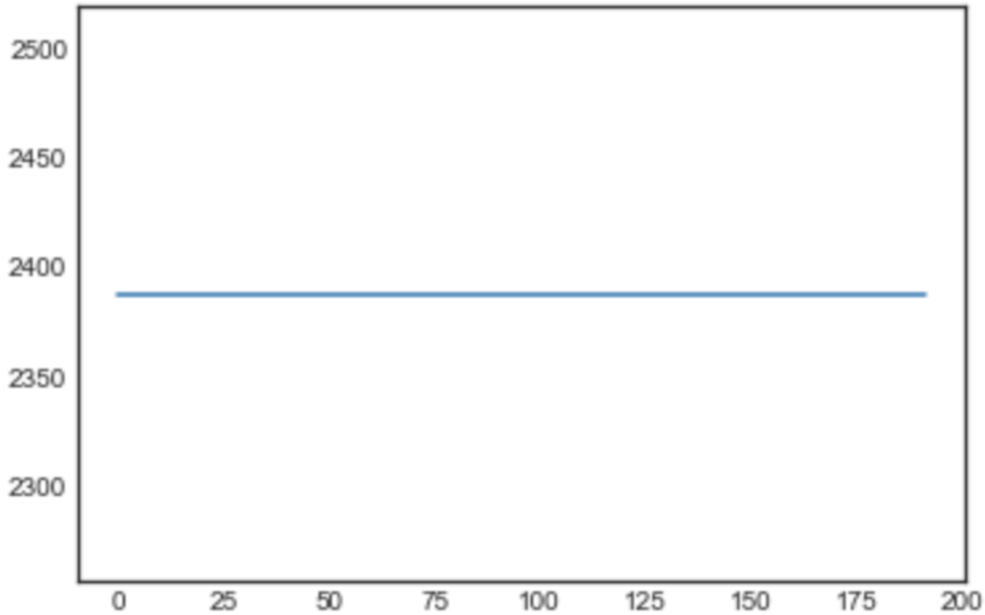

Signal from sensor 14 in 1 cycle
8140
8135
8130
8125
8120
8115
8110
0
25
50
75
100
125
150
175
200


Signal from sensor 15 in 1 cycle
8.52
8.50
8.48
8.46
8.44
8.42
8.40
8.38
0
25
50
75
100
125
150
175
200

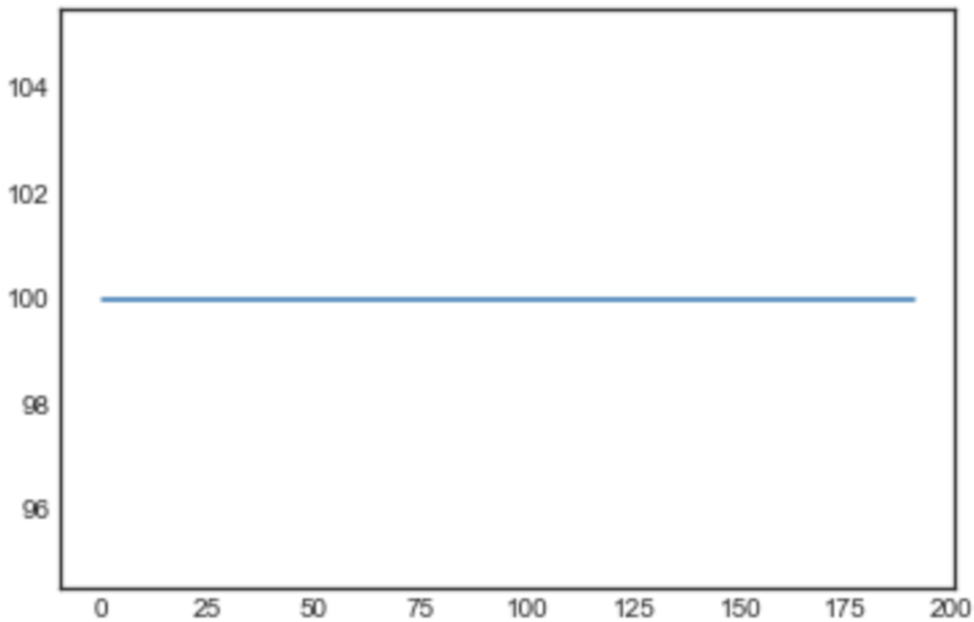

Signal from sensor 16 in 1 cycle
0.0315
0.0310
0.0305
0.0300
0.0295
0.0290
0.0285
0
25
50
75
100
125
150
175
200

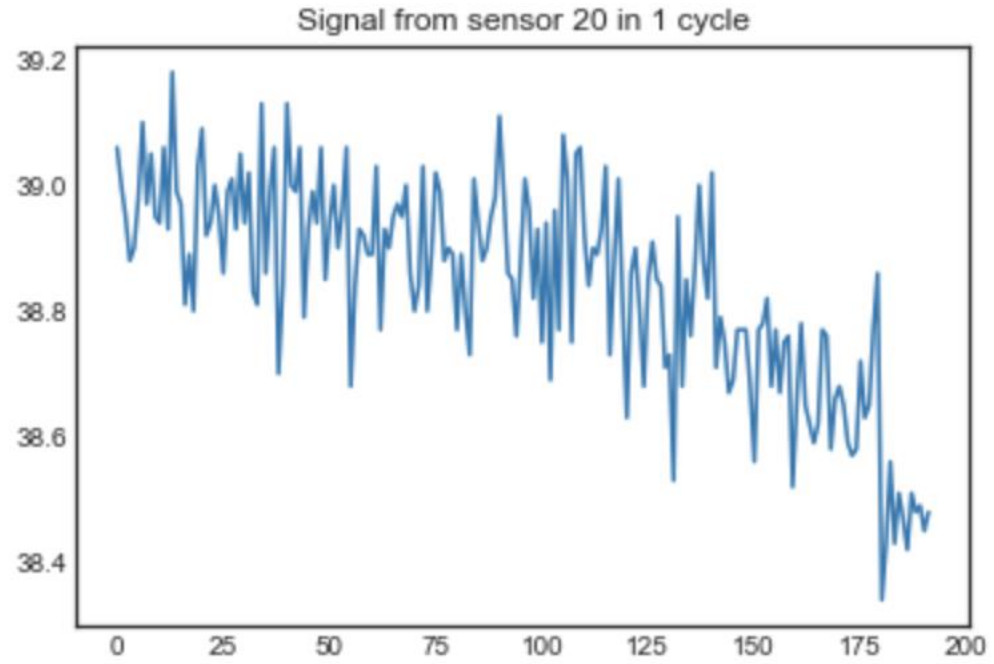

Signal from sensor 17 in 1 cycle
398
397
396
395
394
393
392
391
390
0
25
50
75
100
125
150
175
200

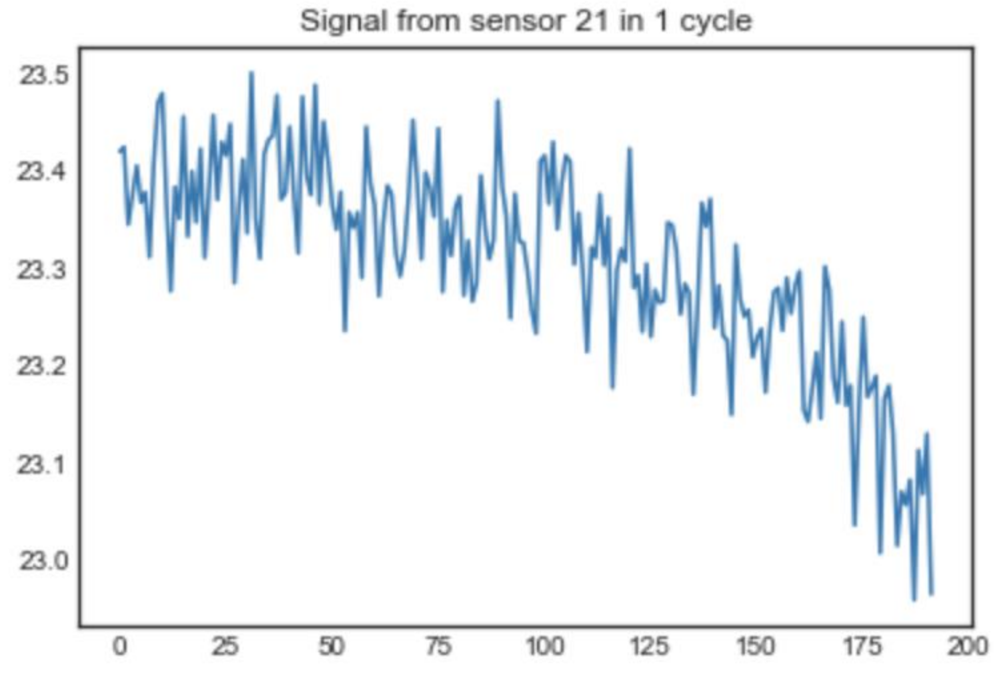
Signal from sensor 18 in 1 cycle



Signal from sensor 19 in 1 cycle







According to the 21 images, we can get that
sensor 7,9,12,14,20,21 decrease with fluctuation,
sensor 1,5,6,10,16,18,19 keep stable,
sensor 2,3,4,8,11,13,15 increase with fluctuation.

Therefore, for first set of sensors, the smaller value they are, the smaller lifespan engine is, for the second set of sensors, their value are not relevant to the lifespan of engine and for the third set of sensors, their values have the opposite meaning to the first, which means that the bigger value is, smaller the lifespan of engines are.

4. two link to the report online

github https://github.com/Lfrankie0315/EE5111-AWS-project

medium post https://medium.com/@475628967/project-1-to-simulate-lot-pipeline-on-aws-platform-fa4efea647b?sk=70a1381ff7ca7f031106d3c3fa4b3d85

5. Summary

Doing this project is really a hard challenge for me. However, the procedure of finishing project does make me know the basic data processing. In addition to that, I have also learned how to take advantage of AWS.

Generally speaking, this project gives me a lot of gains. It makes me take interested in data processing as well.

github link