EE5111 SELECTED TOPICS IN INDUSTRIAL CONTROL & INSTRUMENTATION

CA 1 IoT project on Amazon AWS

Name: Yu Shixin

Matriculation Number.: A0195017E

Date:2019/9/16

1. Introduction

1.1 Background

The internet of things (IoT) is a computing concept that describes the idea of everyday physical objects being connected to the internet and being able to identify themselves to other devices. The term is closely identified with RFID as the method of communication, although it also may include other sensor technologies, wireless technologies or QR codes.

The IoT is significant because an object that can represent itself digitally becomes something greater than the object by itself. No longer does the object relate just to its user, but it is now connected to surrounding objects and database data. When many objects act in unison, they are known as having "ambient intelligence."

All in all, the Internet of Things is a network of physical objects – vehicles, machines, home appliances, and more – that use sensors and APIs to connect and exchange data over the Internet.

In this project, I will use the AWS Cloud platform to implement a simple IoT pipeline and visualize the data to better understand how the Internet of Things works.

1.2 Goals for project

For this project, I will finish those tasks as below:

a. Implement a simple IoT pipeline with AWS Cloud platform.
b. Visualise the data. I will simulate two small IoT setups that record and push data from two jet engines.
c. Write a guideline on how to set up the pipeline.

2. Study for AWS

2.1 Introduction of AWS Cloud platform

AWS full form is Amazon Web Services. AWS is a growing cloud computing platform which has a significant share of Cloud Computing with respect to its competitors. AWS is geographically diversified into regions to ensure system robustness and outages.

AWS IoT enables Internet-connected devices to connect to the AWS Cloud and lets applications in the cloud interact with Internet-connected devices. Common IoT applications either collect and process telemetry from devices or enable users to control a device remotely. Devices report their state by publishing messages, in JSON format, on MQTT topics. Each MQTT topic has a hierarchical name that identifies the device whose state is being updated. When a message is published on an MQTT topic, the message is sent to the AWS IoT MQTT message broker, which is responsible for sending all messages published on an MQTT topic to all clients subscribed to that topic.

We can create rules that define one or more actions to perform based on the data in a message. For example, you can insert, update, or query a DynamoDB table or invoke a Lambda function. Rules use expressions to filter messages. When a rule matches a message, the rules engine triggers the action using the selected properties. Rules also contain an IAM role that grants AWS IoT permission to the AWS resources used to perform the action.

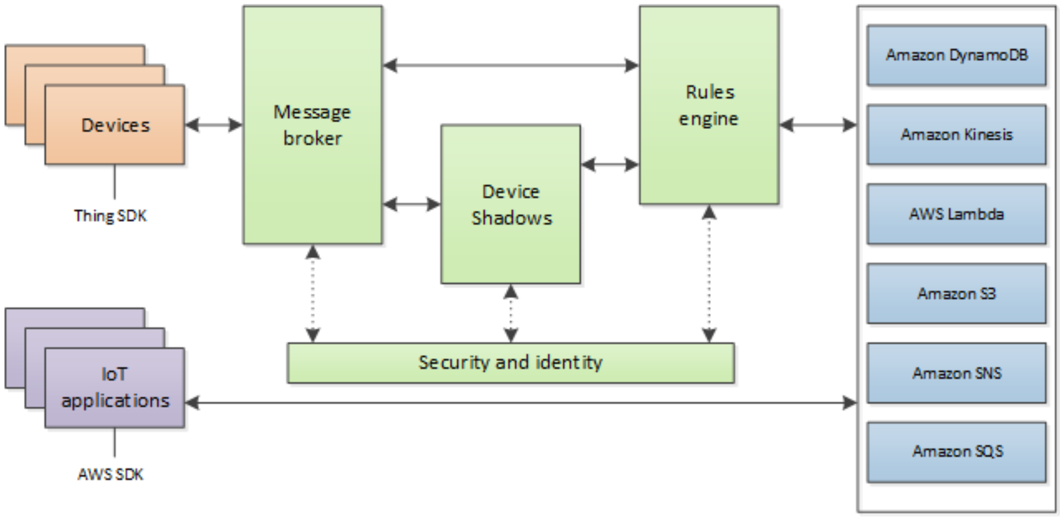


Fig.2.1 structure of AWS IoT system

The structure of AWS IoT system is shown as above, the things SDK are devices that we connect to the AWS cloud and the data is send to the shadow that be an intermediate station. The shadow will manage the connection of the things. Also, there is security and identity part to ensure the access to AWS cloud is permitted. Only the things that pass the thing registry can access. The customs can set rules to decide which information will be draw to the database. The data base service on the AWS have several and custom can choose based on their choice.

2.2 Learn to use AWS (AWS IoT Plant Watering Sample)

Because I have no experience with AWS and IoT development, I follow the 'AWS IoT Plant Watering Sample' to familiarize myself with the AWS IoT' API and workflow.

First, go through the 'Getting start with AWS IoT' official document to know how to sign in the AWS IoT console, what is the rule, the thing, the certification, the policy and how to test rules. After understanding the basic conception of AWS IoT, I follow the 'AWS IoT Plant Watering Sample' to understand the whole workflow of AWS. Creating the AWS IoT policy, the thing, sending and receiving test data for the thing, setting up e-mail alerts for low moisture readings and simulating random moisture level.

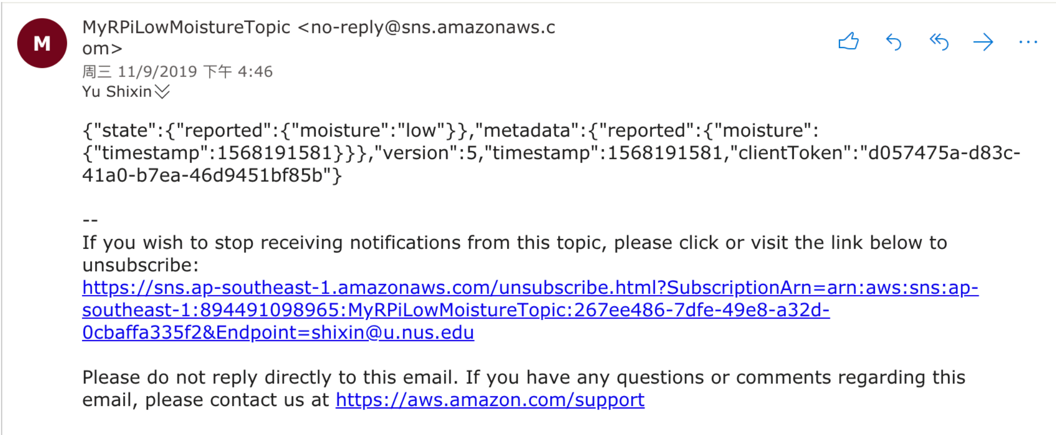


Fig.2.2 alarm email from AWS

2.3 Set up own AWS platform

In this part, I will show you the detailed action for each step to set up your own AWS platform and use AWS IoT services to finish further tasks in project. (The steps are similar with Plant Watering Sample.) **Before your starting, make sure clean all the data in the learning part.**

2.3.1 Create the AWS IoT policy (secure->policy)

Create a policy named '**policy_A0195017E**' to allow my computer as simulator to operate AWS IoT action. For **action**, I set as '**iot:***', and for **resource ARN**, replace the suggested value with an asterisk (*).

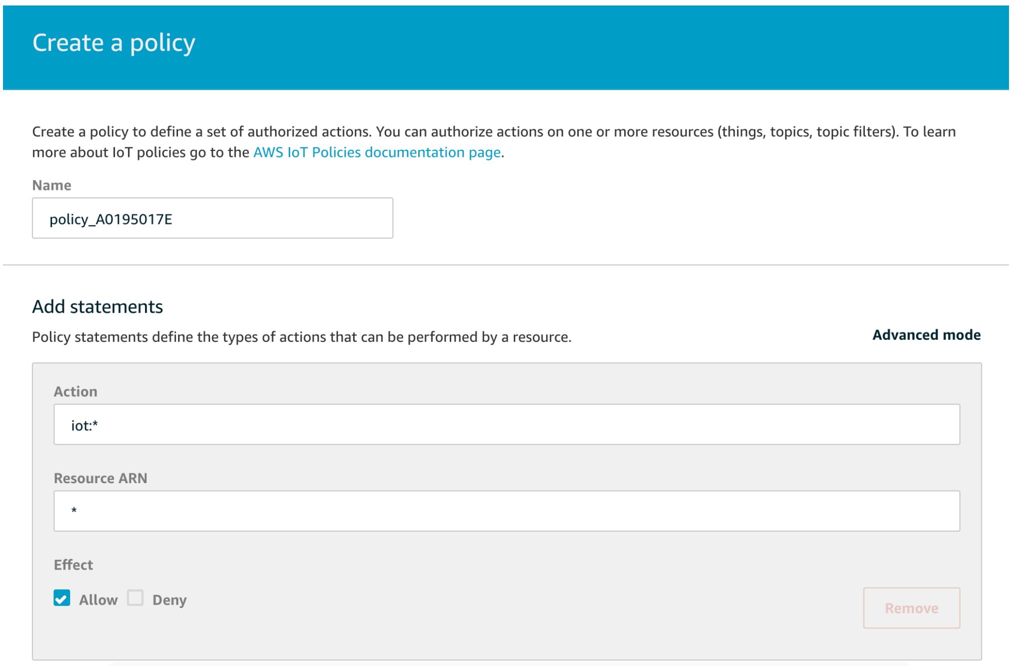


Fig.2.3 create the AWS IoT policy

2.3.2 Create the thing (manage->things)

Create a thing named '**thing1_ A0195017E**'. Before you finish a thing creation, you have to create a certificate shown in Fig.2.4. Downloading all of the four files, **A certificate for this thing** (saved it ending in certificate.pem.crt.txt), **A public key**, **A private key**, and **A root CA for AWS IoT** shown in Fig.2.5. Then click **activate**.

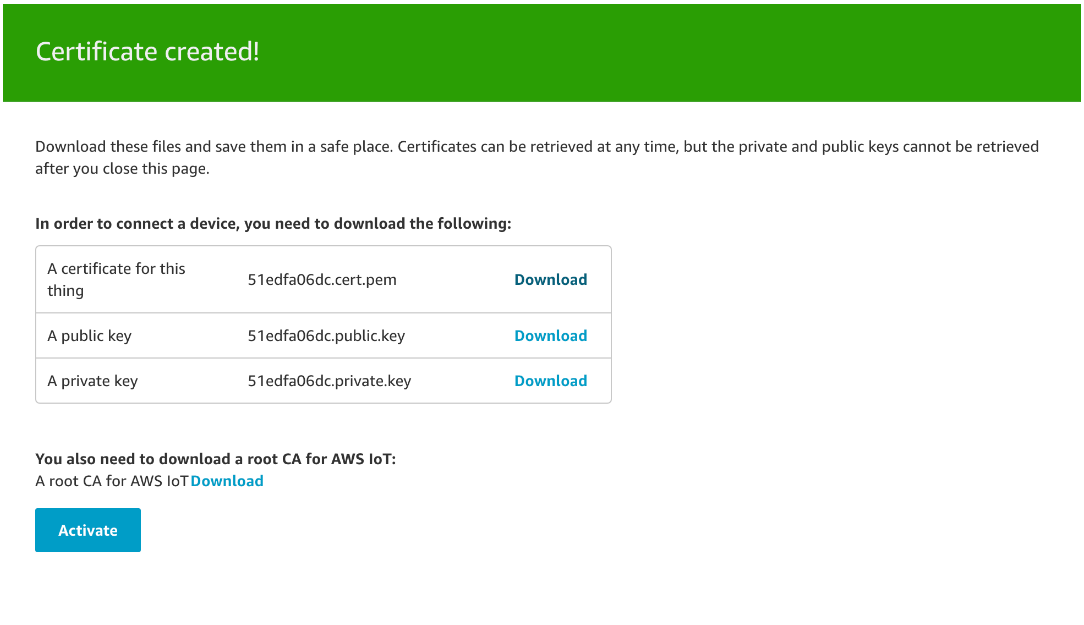


Fig.2.4 create certificates

Fig.2.5 certificates

Add a policy for your thing.

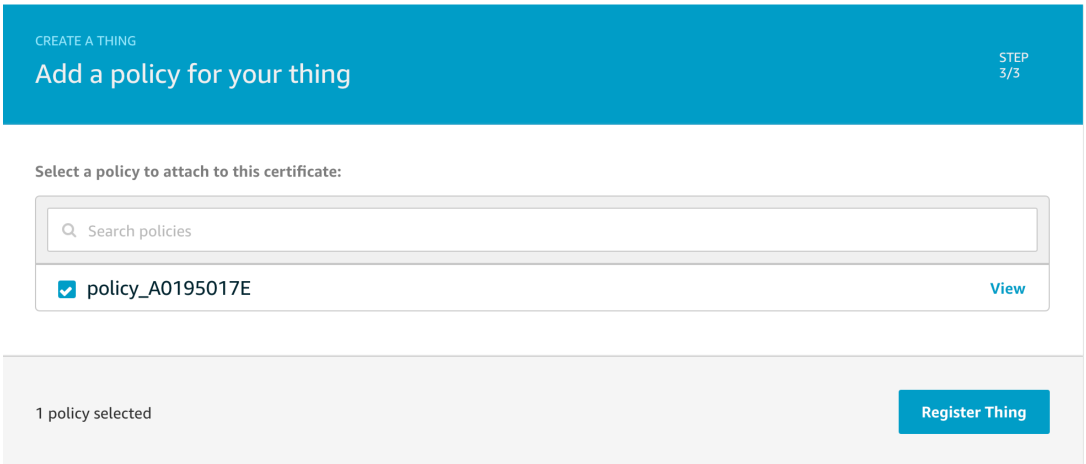


Fig.2.6 certificates

Repeat those step to create the '**thing2_ A0195017E**'. Finally, you will get two things (Fig.2.7).

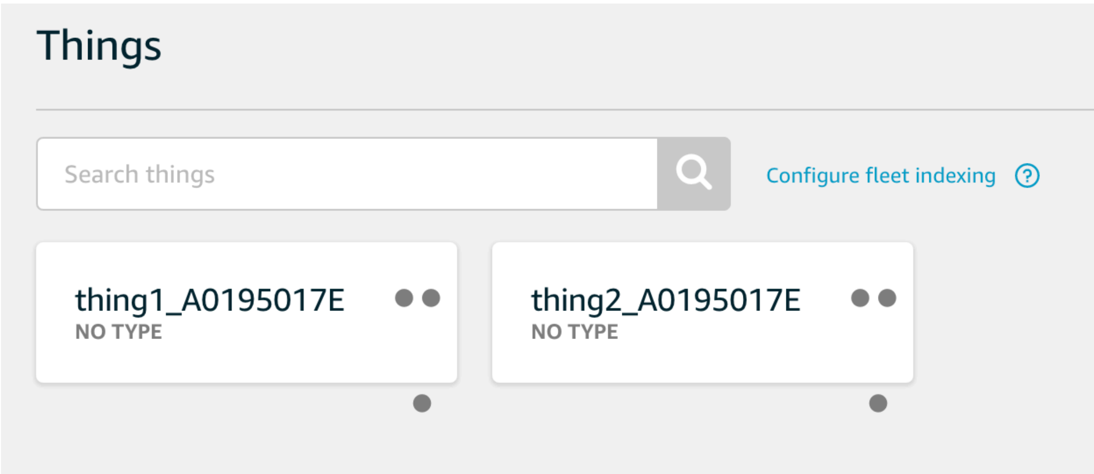


Fig.2.7 two things

2.3.3 Send and Receive Test Data for the Thing

We can send test data to the thing shadow for my computer to test the connection.

Choose '**thing1_ A0195017E**' -> **interact**. We use MQTT protocol to communicate with shadow. We use this three command to test.

- **Update to this thing shadow**
- **Get this thing shadow**
- **Get this thing shadow accepted**

Then choose **test**. For each subscription topic shown above, we enter its MQTT topic value and **subscribe to topic**.

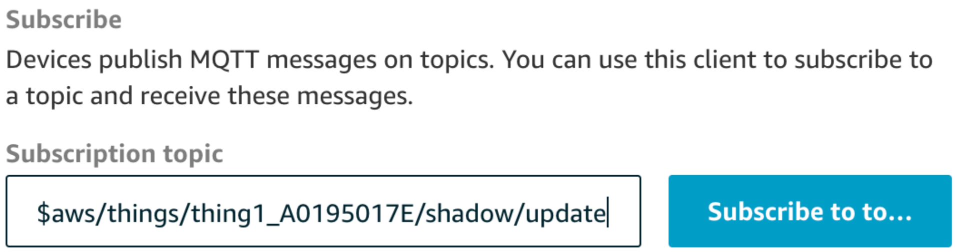
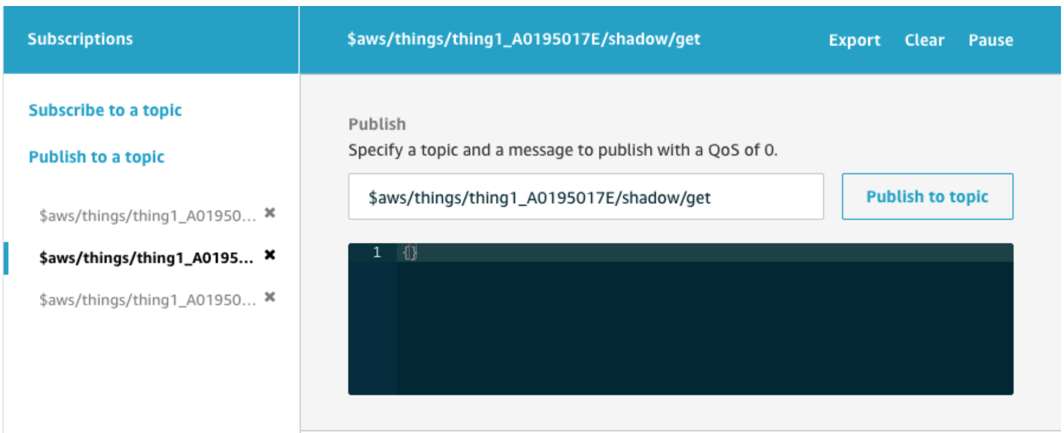


Fig.2.8 Subscription topic

Now push some test data into the shadow. Select **update** topic, enter the message as below in the message payload area and **publish to topic**. Select **get** topic, enter blank brace in the message payload area and **publish to topic**. In the end, I will get message like Fig.2.9-3 in **Get this thing shadow accepted**.



```
{
  "state": {
    "reported": {
      "Content": "This is a test message by A0195017E"
    }
  },
  "metadata": {
    "reported": {
      "Content": {
        "timestamp": 1568700657
      }
    }
  },
  "version": 2,
  "timestamp": 1568700660
}
```


Fig.2.9 do a test

3. Step 1 : Publish pre-defined engine data to AWS.

3.1 Download data & understand

We can download engine data 'train_FD001.txt' from GitHub, and understand each column's meaning by reading 'CMAPSS_Data_Readme.doc'. For each row, the meaning of each value is id, cycle, os 1, os 2, os 3,sensor 1,…sensor 21. Next, we also use them as the table's header.

3.2 Prepare data for requirement

After understanding the meaning of data, I load data into DynamoDB by using JupyterLab.

The AWS IoT provides several API to publish data to AWS, we can find this in its instruction from official website (https://docs.aws.amazon.com/iot/latest/developerguide/iot-plant-step5.html). First, I configure the parameters as below to create a connection with AWS IoT platform.

```
from AWSIoTPythonSDK.MQTTLib import AWSIoTMQTTShadowClient
import random, time

# Configure parameters.
SHADOW_CLIENT = "ShadowClient1_A0195017E "
HOST_NAME = "a3okd7abcwvnh4-ats.iot.ap-southeast-1.amazonaws.com"
ROOT_CA = "AmazonRootCA1.pem.txt"
```

```
    PRIVATE_KEY = "51edfa06dc-private.pem.key"
    CERT_FILE = " 51edfa06dc-certificate.pem.crt "
    SHADOW_HANDLER = "thing1_A0195017E"

    # Automatically called whenever the shadow is updated.
    def myShadowUpdateCallback(payload, responseStatus, token):
        print("responseStatus = " + responseStatus)

    # 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)
```

Then, we need to load the file to Jupyter, overwrite ‘id’ into ‘FD001’+id, add 2 columns (timestamp and Matric No.) and publish each message to DynamoDB table. The code shown as below:

```
    #load & overwrite
    file = pandas.read_csv('train_FD001.txt', delim_whitespace = True, header =
None)
    sensor=[]
    for i in range(21):
        sensor[i]='sensor'+ str(i+1)
    file.columns = ['id', 'cycle', 'os1', 'os2', 'os3'] + sensor
    file['id'] = file['id'].map(lambda x: 'FD001_'+str(x))
    file['Matric No.']='A0195017E'

    for i in range(len(file)):
        print("line:" + str(i+1))
        msg=trainFD.loc[[i]]
        msg['timestamp']='UTC '+str(datetime.datetime.utcnow())
        msg.index=["reported"]
        msg=msg.to_json(orient='index')
        msg='{"state":'+msg+'}'
        myDeviceShadow.shadowUpdate(msg,myShadowUpdateCallback, 5)
        time.sleep(10)
```

3.3 Set up DynamoDB & Check under AWS DynamoDB service

*This step I finished before **Publish pre-defined engine data to AWS**.

Choose **act->rule**, create a rule to connect with DynamoDB named ‘**rule_ A0195017E**’.

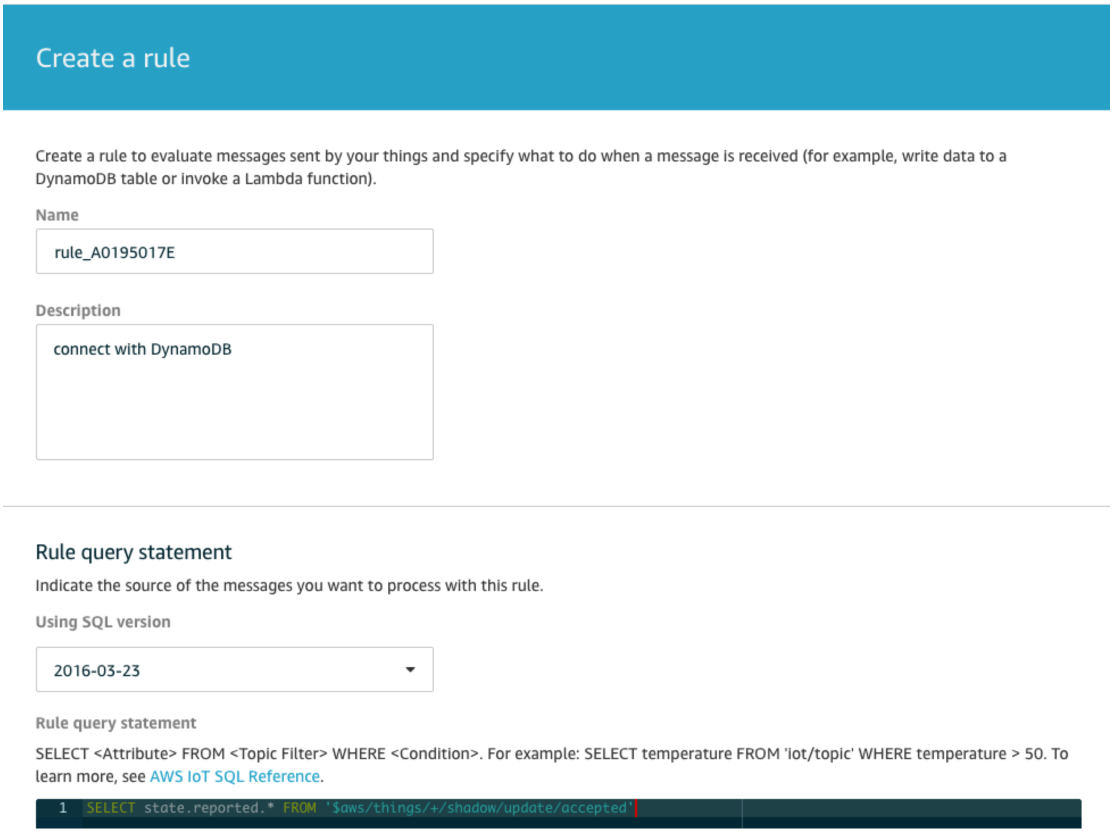


Fig.3.1 create a rule

Because we need to operate 2 things, we should write SQL of rule like this:

```
SELECT state.reported.* FROM '$aws/things/+/shadow/update/accepted'
```

If you only have one thing, you can replace '+' with your things' name. Then select an action. Since we need to split the TD001.txt and FD002.txt into multiple columns, so we choose **DynamoDBv2**.

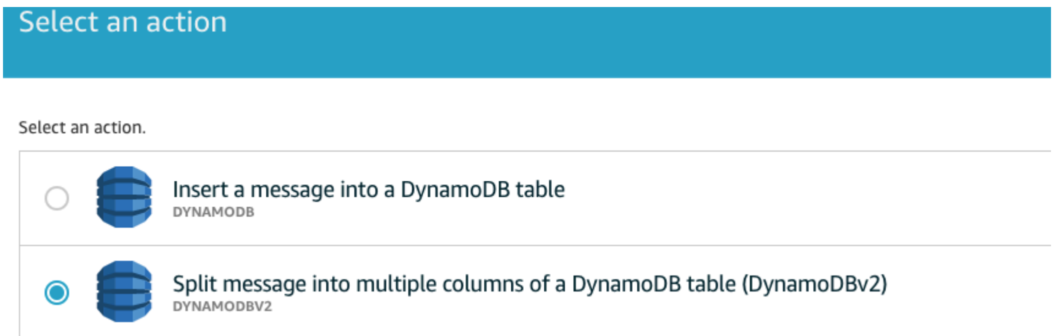


Fig.3.2 select an action

Click **create a new resource**, jump to DynamoDB page. Create a DDB table, set 'id' as the partition key and 'timestamp' as the sort key.

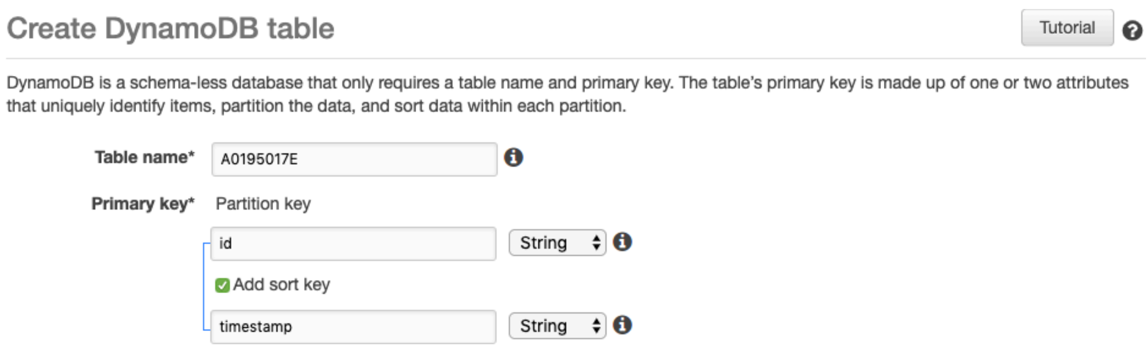


Fig.3.3 create DynamoDB table

Then return to the IoT core, click fresh and select the DDB table named '**A0195017E**' as the resource. Create a role and finish this part.

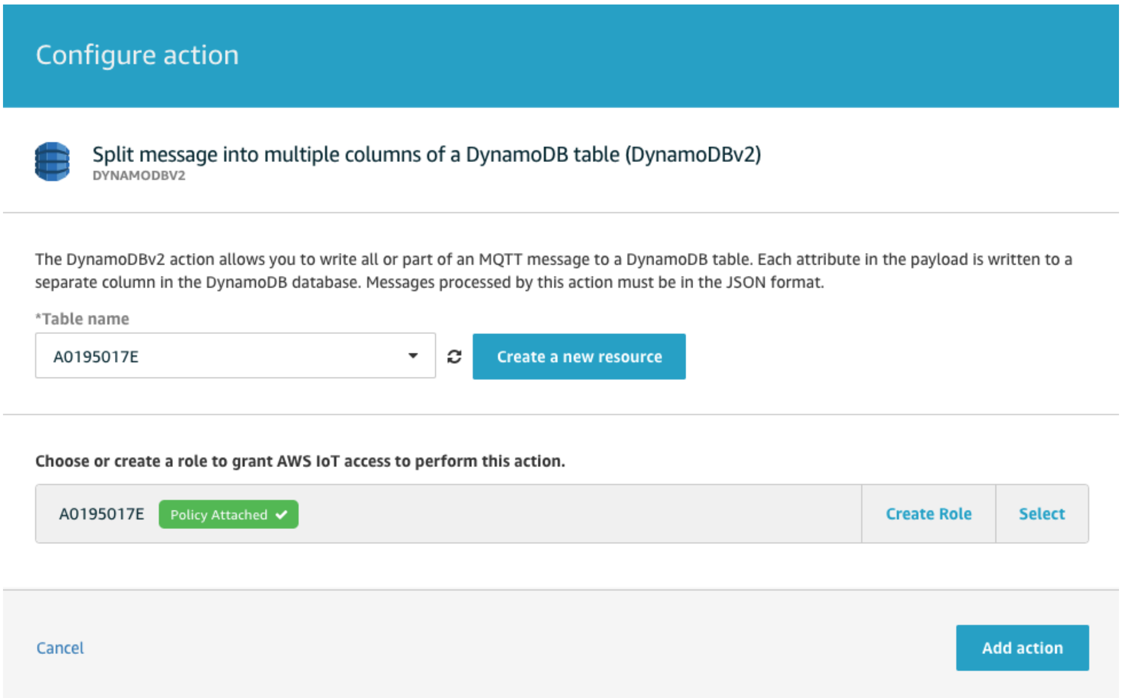
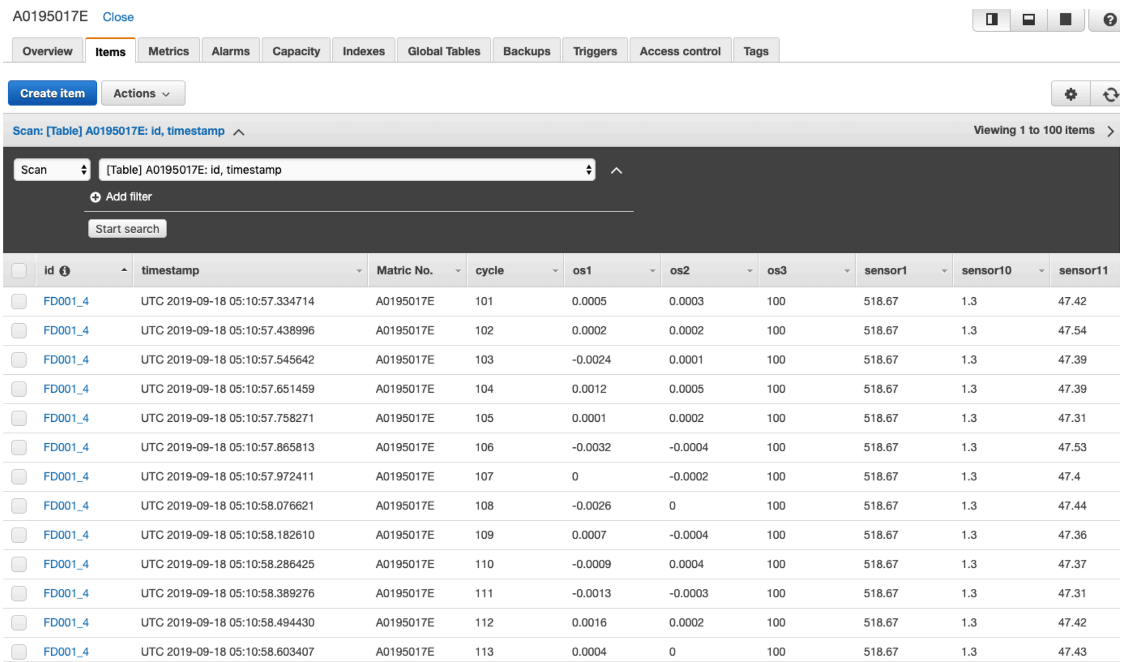


Fig.3.4 congfigure action

At this stage, we have set up the DymanoDB. Run the code we have written, we can publish data to the DynamoDB table.



id	timestamp	Matric No.	cycle	os1	os2	os3	sensor1	sensor10	sensor11
FD001_4	UTC 2019-09-18 05:10:57.334714	A0195017E	101	0.0005	0.0003	100	518.67	1.3	47.42
FD001_4	UTC 2019-09-18 05:10:57.438996	A0195017E	102	0.0002	0.0002	100	518.67	1.3	47.54
FD001_4	UTC 2019-09-18 05:10:57.545642	A0195017E	103	-0.0024	0.0001	100	518.67	1.3	47.39
FD001_4	UTC 2019-09-18 05:10:57.651459	A0195017E	104	0.0012	0.0005	100	518.67	1.3	47.39
FD001_4	UTC 2019-09-18 05:10:57.758271	A0195017E	105	0.0001	0.0002	100	518.67	1.3	47.31
FD001_4	UTC 2019-09-18 05:10:57.865813	A0195017E	106	-0.0032	-0.0004	100	518.67	1.3	47.53
FD001_4	UTC 2019-09-18 05:10:57.972411	A0195017E	107	0	-0.0002	100	518.67	1.3	47.4
FD001_4	UTC 2019-09-18 05:10:58.076621	A0195017E	108	-0.0026	0	100	518.67	1.3	47.44
FD001_4	UTC 2019-09-18 05:10:58.182610	A0195017E	109	0.0007	-0.0004	100	518.67	1.3	47.36
FD001_4	UTC 2019-09-18 05:10:58.286425	A0195017E	110	-0.0009	0.0004	100	518.67	1.3	47.37
FD001_4	UTC 2019-09-18 05:10:58.389276	A0195017E	111	-0.0013	-0.0003	100	518.67	1.3	47.31
FD001_4	UTC 2019-09-18 05:10:58.494430	A0195017E	112	0.0016	0.0002	100	518.67	1.3	47.42
FD001_4	UTC 2019-09-18 05:10:58.603407	A0195017E	113	0.0004	0	100	518.67	1.3	47.43

Fig.3.5 check under AWS DynamoDB service

4. Step 2 : Simulating two IoT things.

4.1 Add one more thing under AWS IoT platform

When I set up my own AWS IoT platform, I have created two things. The other one named '**thing2_ A0195017E**'. The configure parameters for thing2 are shown as below. And the data for thing2 is 'train_FD002.txt'. Repeat other steps as thing1.

```
# Configure parameters.
SHADOW_CLIENT = "ShadowClient2_ A0195017E"
HOST_NAME = "a3okd7abcwvnh4-ats.iot.ap-southeast-1.amazonaws.com"
ROOT_CA = "AmazonRootCA1.pem.txt"
PRIVATE_KEY = "3d36242fd5-private.pem.key"
CERT_FILE = "3d36242fd5-certificate.pem.crt "
SHADOW_HANDLER = "thing2_A0195017E"
```

4.2 Run two things simultaneously

Then Modify the rule under AWS IoT platform to be triggered by '$aws/things/+/shadow/update/accepted'.

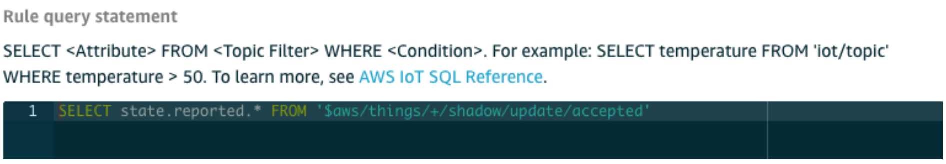
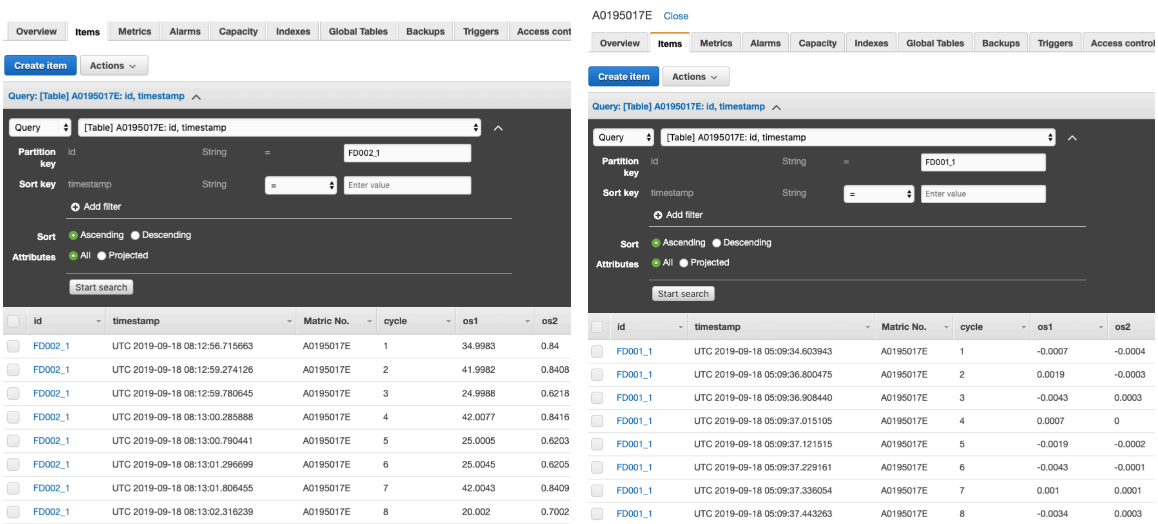


Fig.4.1 modify the rule

Run those two files simultaneously at the JupyterLab and fresh DynamoDB, we can get the results as below. Both of train_FD001.txt and train_FD002.txt are pushed into DynamoDB.



id	timestamp	Matric No.	cycle	os1	os2
FD002_1	UTC 2019-09-18 08:12:56.715663	A0195017E	1	34.9983	0.84
FD002_1	UTC 2019-09-18 08:12:59.274126	A0195017E	2	41.9982	0.8408
FD002_1	UTC 2019-09-18 08:12:59.780645	A0195017E	3	24.9988	0.6218
FD002_1	UTC 2019-09-18 08:13:00.285888	A0195017E	4	42.0077	0.8416
FD002_1	UTC 2019-09-18 08:13:00.790441	A0195017E	5	25.0005	0.6203
FD002_1	UTC 2019-09-18 08:13:01.296699	A0195017E	6	25.0045	0.6205
FD002_1	UTC 2019-09-18 08:13:01.806455	A0195017E	7	42.0043	0.8409
FD002_1	UTC 2019-09-18 08:13:02.316239	A0195017E	8	20.002	0.7002

id	timestamp	Matric No.	cycle	os1	os2
FD001_1	UTC 2019-09-18 05:09:34.603943	A0195017E	1	-0.0007	-0.0004
FD001_1	UTC 2019-09-18 05:09:36.800475	A0195017E	2	0.0019	-0.0003
FD001_1	UTC 2019-09-18 05:09:36.908440	A0195017E	3	-0.0043	0.0003
FD001_1	UTC 2019-09-18 05:09:37.015105	A0195017E	4	0.0007	0
FD001_1	UTC 2019-09-18 05:09:37.121515	A0195017E	5	-0.0019	-0.0002
FD001_1	UTC 2019-09-18 05:09:37.229161	A0195017E	6	-0.0043	-0.0001
FD001_1	UTC 2019-09-18 05:09:37.336054	A0195017E	7	0.001	0.0001
FD001_1	UTC 2019-09-18 05:09:37.443263	A0195017E	8	-0.0034	0.0003

Fig.4.2 stored in table

Finally, we store both files' data in the table.

Fig.4.3 total size of table

5. Step 3 : Visualise the data

5.1 Query data from AWS DynamoDB

In the following visualization, we will use the data querying from AWS DynamoDB we pushed just now. To connect AWS DynamoDB, we need to use IAM to get access key, which allows you to sign programmatic requests to AWS services.

Access Key ID

AKIAIAE4ZHJYNQSTJS3A

Fig.5.1 access key ID

In order to interact with DynamoDB by using Python, we can use boto3, which is easy to integrate your Python application, library, or script with AWS services including Amazon S3, Amazon EC2, Amazon DynamoDB, and more.

```
    #parameters config
    AWS_ACCESS_ID='AKIAIAE4ZHJYNQSTJS3A'
    AWS_ACCESS_KEY='4miq8EXXmAe5GNkxJf+jeyhT3r1Ftum5w5pSR9LY'
    TABLE_NAME='A0195017E'
    REGION_NAME='ap-southeast-1'

    #create service
    client=boto3.client('dynamodb',region_name=REGION_NAME,aws_access_key_id=AW
S_ACCESS_ID,aws_secret_access_key=AWS_ACCESS_KEY)
    dynamodb=boto3.resource('dynamodb',region_name=REGION_NAME,aws_access_key_i
d=AWS_ACCESS_ID,aws_secret_access_key=AWS_ACCESS_KEY)

    #scan table
    class DDB2JSON(json.JSONEncoder):
        def default(self, o):
            if isinstance(o,decimal.Decimal):
                if o%1>0:
                    return float(o)
                else:
                    return int(o)
            return super(DDB2JSON,self).default(o)
    table=dynamodb.Table(TABLE_NAME)
    response=table.scan()
    item=[]
    for i in response['Items']:
        item.append(json.dumps(i,cls=DDB2JSON))
    while 'LastEvaluatedKey' in response:
        response=table.scan(ExclusiveStartKey=response['LastEvaluatedKey'])
        for j in response['Items']:
            item.append(json.dumps(j,cls=DDB2JSON))

    #display
    m=1
    msg=pd.DataFrame()
    for n in item:
        msg=msg.append(pd.DataFrame(json.loads(n),index=[m]))
        m+=1
    msg.head(100)
```

Finally, we can get the data pulled from DynamoDB.

[1]:

	sensor21	sensor20	sensor18	id	sensor19	sensor16	sensor17	sensor14	sensor15	sensor12	...	sensor6	sensor5	Matric No.	sensor4	sensor3	sensor9	sensor8	sensor7	sensor2	sensor1
1	23.4190	39.06	2388	FD001_1	100	0.03	392	8138.62	8.4195	521.66	...	21.61	14.62	A0195017E	1400.60	1589.70	9046.19	2388.06	554.36	641.82	518.67
2	23.4236	39.00	2388	FD001_1	100	0.03	392	8131.49	8.4318	522.28	...	21.61	14.62	A0195017E	1403.14	1591.82	9044.07	2388.04	553.75	642.15	518.67
3	23.3442	38.95	2388	FD001_1	100	0.03	390	8133.23	8.4178	522.42	...	21.61	14.62	A0195017E	1404.20	1587.99	9052.94	2388.08	554.26	642.35	518.67
4	23.3739	38.88	2388	FD001_1	100	0.03	392	8133.83	8.3682	522.86	...	21.61	14.62	A0195017E	1401.87	1582.79	9049.48	2388.11	554.45	642.35	518.67
5	23.4044	38.90	2388	FD001_1	100	0.03	393	8133.80	8.4294	522.19	...	21.61	14.62	A0195017E	1406.22	1582.85	9055.15	2388.06	554.00	642.37	518.67
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
96	23.3255	38.88	2388	FD001_1	100	0.03	392	8130.69	8.4311	521.66	...	21.61	14.62	A0195017E	1395.16	1584.07	9048.71	2388.07	553.34	642.19	518.67
97	23.2963	39.01	2388	FD001_1	100	0.03	392	8128.74	8.4105	521.67	...	21.61	14.62	A0195017E	1407.81	1595.77	9046.10	2388.09	553.40	642.07	518.67
98	23.2554	38.96	2388	FD001_1	100	0.03	391	8127.89	8.4012	522.31	...	21.61	14.62	A0195017E	1404.56	1591.11	9045.49	2388.06	552.75	642.00	518.67
99	23.2323	38.82	2388	FD001_1	100	0.03	393	8131.77	8.4481	521.42	...	21.61	14.62	A0195017E	1406.13	1592.73	9045.14	2388.08	553.76	642.46	518.67
100	23.4090	38.93	2388	FD001_1	100	0.03	392	8132.49	8.4241	521.55	...	21.61	14.62	A0195017E	1411.35	1589.63	9045.72	2388.07	554.22	642.22	518.67

Fig.5.2 data pulled from DynamoDB

5.2 Visualization

After we pulling data from DynamoDB to the JuptyerLab workspace, we can use them to do some researches. According to 'CMAPSS_Data_Readme', we can know the function of each sensor.

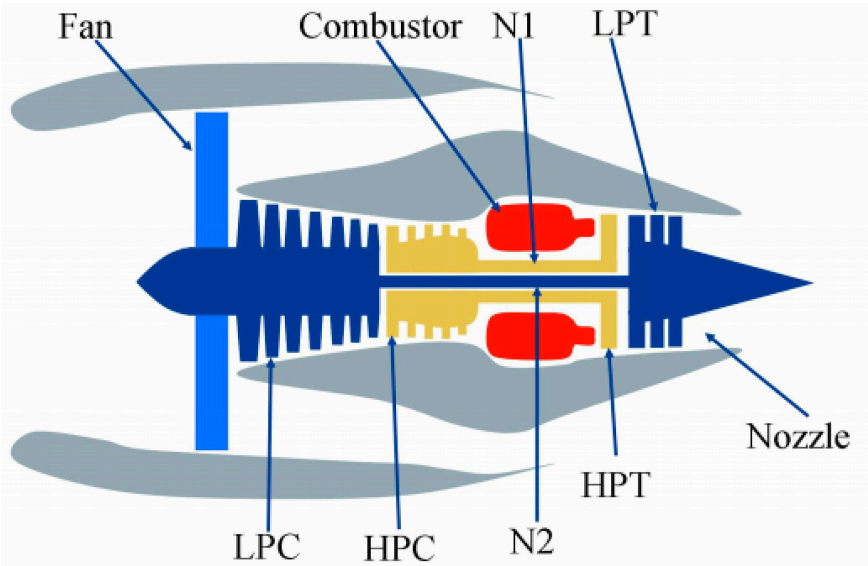


Fig.5.3 engine

Measurement Temperatures		
T48 (s1)	Total temperature at HPT outlet	R
T2 (s2)	Total temperature at fan inlet	R
T24 (s3)	Total temperature at LPC outlet	R
T30 (s4)	Total temperature at HPC outlet	R
T50 (...)	Total temperature at LPT outlet	R
Pressure Measurements		
P2	Pressure at fan inlet	psia
P15	Total pressure in bypass-duct	psia
P30	Total pressure at HPC outlet	psia
Other Measurements		
Nf	Physical fan speed	rpm
Nc	Physical core speed	rpm

epr	Engine pressure ratio (P50/P2)	--
phi	Ratio of fuel flow to Ps30	pps/psiu
Ps30	Static pressure at HPC outlet	psia
NfR	Corrected fan speed	rpm
NcR	Corrected core speed	rpm
BPR	Bypass ratio	--
farB	Burner fuel-air ratio	--
htBleed	Bleed enthalpy	--
PCNfRdmd	Percent corrected fan speed	pct
W31	HPT cooland bleed	lbm/s
W32	HPT cooland bleed	lbm/s

Table.5.1 Parameter descriptions

5.2.1 Temperature analysis

According to table.5.1, we know that sensor 1-5 regard to Measurement Temperatures. Then I visualize the 5 sensors' data as below. I find that the data from sensor 1 & 5 don't have too much change during the flight. I guess that this is because the location of sensor 1 (HPT) and sensor 5 (LPT) are close to Nozzle, where the temperature is steady.

```
sensor='sensor5'
sensor_all_times = train.ix[:,sensor]
sensor_1st_time = train.ix[train['id']==1,sensor]
plt.figure(1)
plt.plot(sensor_all_times)
plt.title('Total temperature Signal from sensor5 in all times')
plt.figure(2)
plt.plot(sensor_1st_time)
plt.title('Total temperature Signal from sensor5 in 1st time')
```

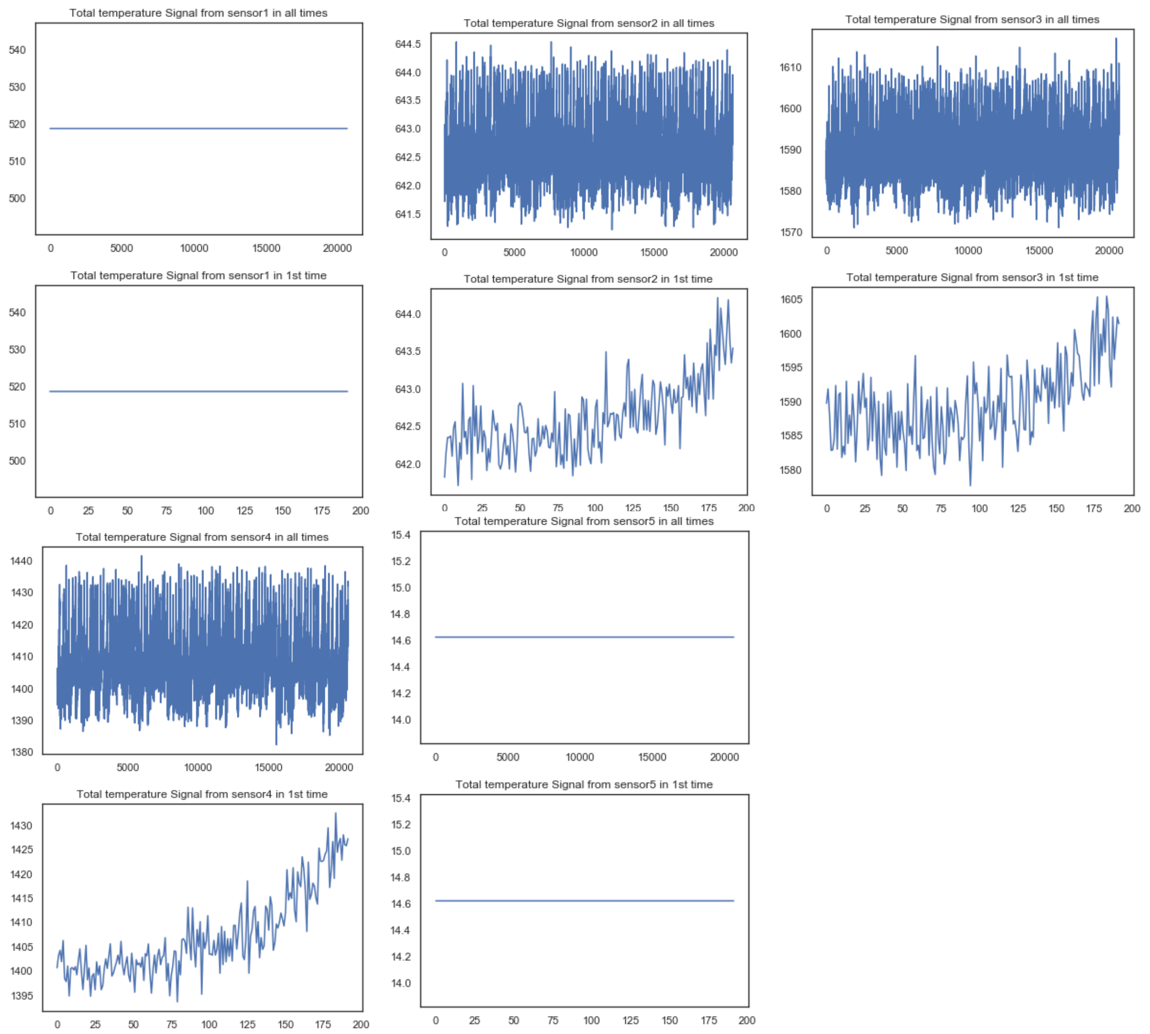


Fig.5.4 the temperature signal from sensor 1-5

5.2.2 Pressure analysis

The sensor 6-8 regard to Pressure Measurements. Then I visualize the 3 sensors' data as below. I find that the data from sensor 6 doesn't have too much change during the flight. The data from sensor 7 shows a decline trend, but the data from sensor 8 shows an increase trend.

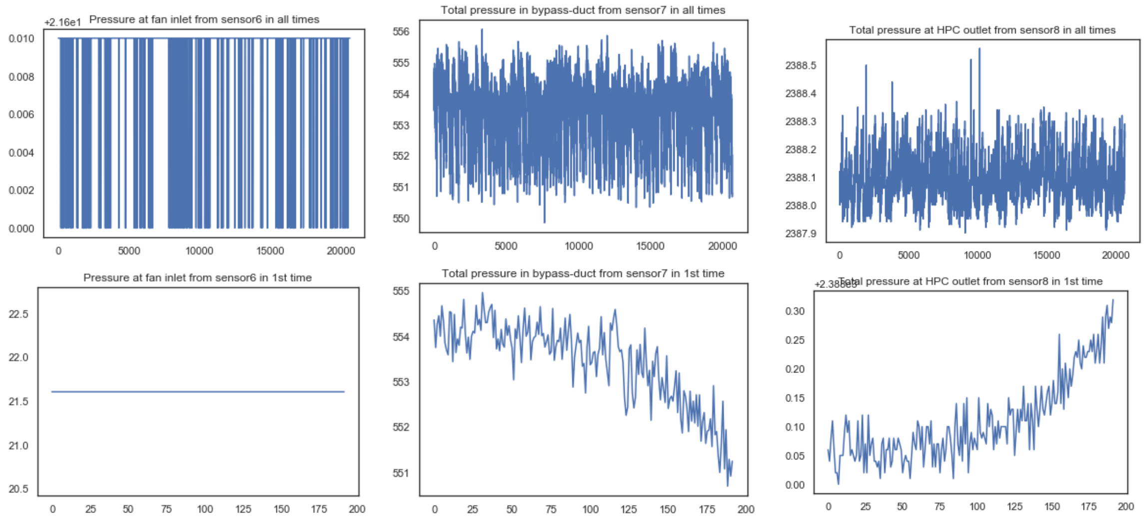
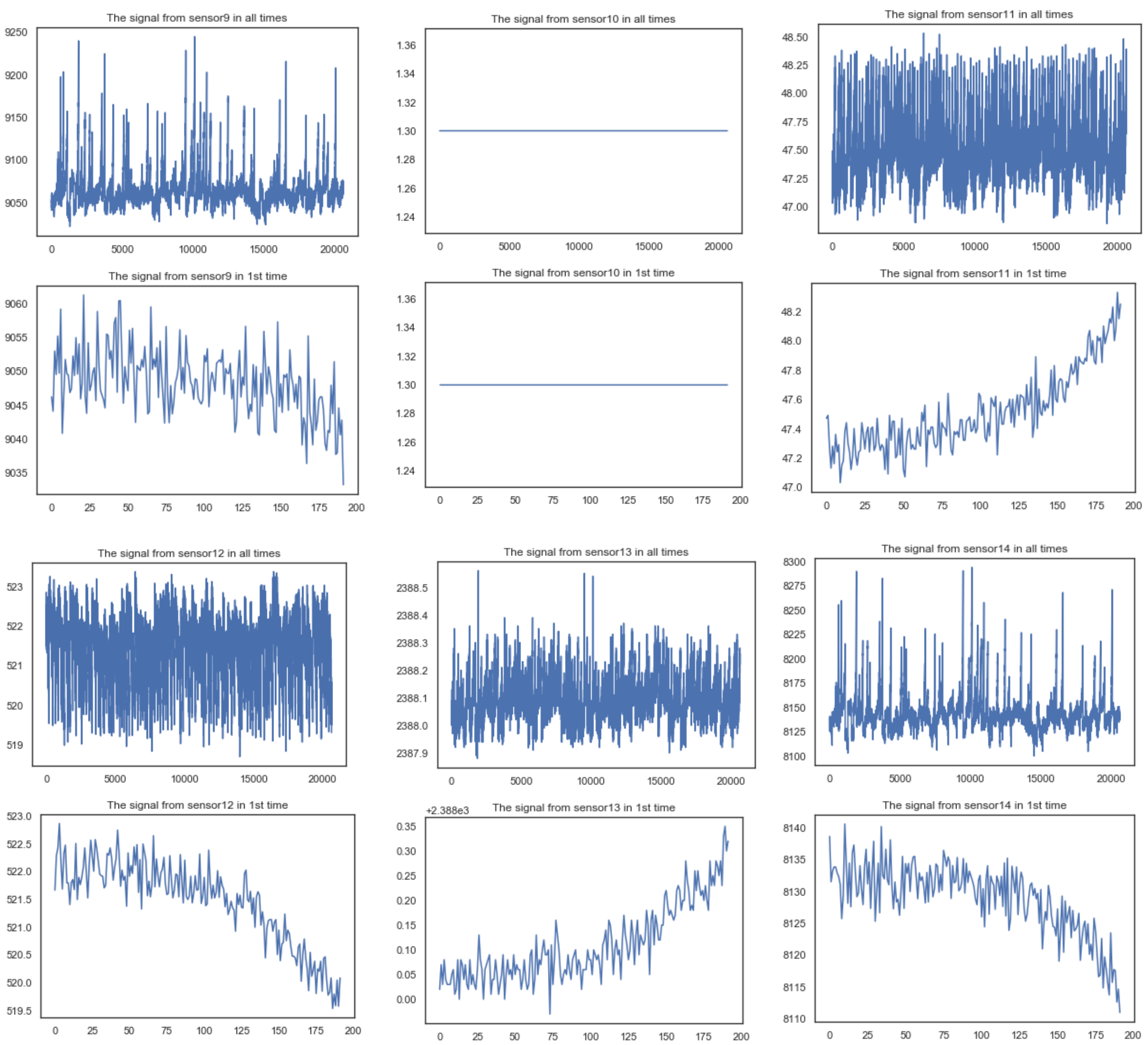


Fig.5.5 the pressure signal from sensor 6-8

5.2.3 Other Measurements

I also show the rest of sensors' data as below:



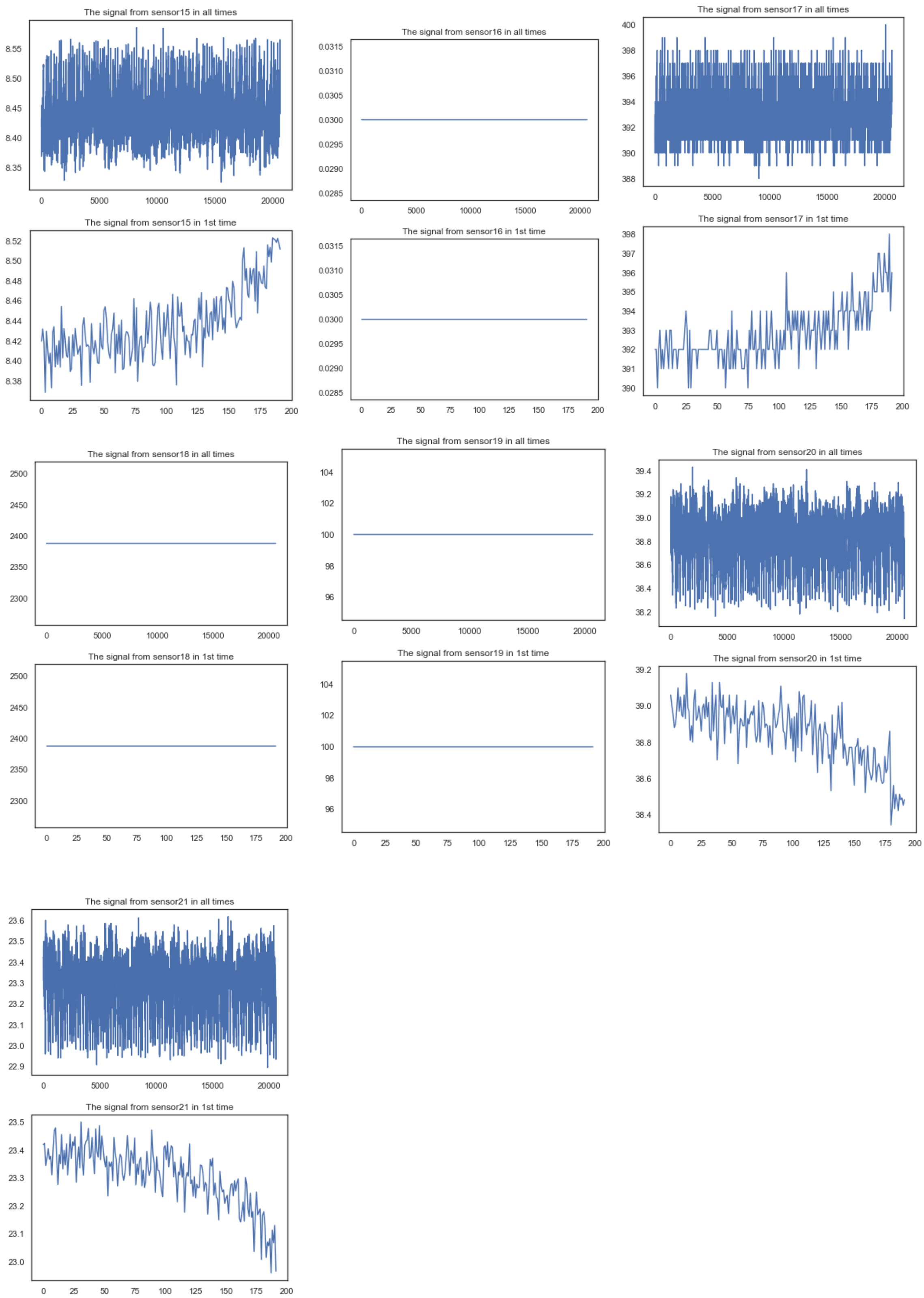


Fig.5.6 the temperature signal from sensor 1-5

5.2.4 Predict Remaining Useful Life (RUL)

In this part, I won't display data simply, and I am going to estimate the remaining useful life by using the Lasso and Random Forest. Those sensors' data can be used to predict RUL. We have two options to construct a target signal for our model to predict: A. Remaining time, which decreases linearly; B. Decelerating health index, which decreases exponentially. I will use FD_002.txt to do this part visualization.

- **Lasso**

Lasso regression is a type of linear regression that uses shrinkage. Shrinkage is where data values are shrunk towards a central point, like the mean. The lasso procedure encourages simple, sparse models (i.e. models with fewer parameters). This particular type of regression is well-suited for models showing high levels of multicollinearity or when you want to automate certain parts of model selection, like variable selection / parameter elimination.

The acronym 'LASSO' stands for Least Absolute Shrinkage and Selection Operator.

A. Linear target

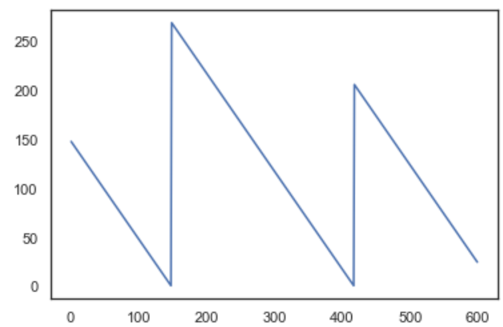


Fig.5.7 the linear target

```
    # prepare the data
    # then split them into train and test sets
    y = train['rul']
```

```
    features = train.columns.drop(['id', 'te', 'rul'])
    X = pd.DataFrame(normalize(train[features], axis=0))
    X.columns = features
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=12)

    # try Lasso
    ls = LassoCV(random_state=12)
    ls = ls.fit(X_train, y_train)

    #print('List of tried parameter values: ', ls.alphas_)
    print('Optimal value: ', ls.alpha_)
    print(ls.coef_)
    print('Useful sensors to predict RUL: ', X_train.columns[abs(ls.coef_) >
1e-6])

    # compare predict RUL and real RUL
    plt.figure(1)
    plt.plot(np.log(ls.predict(X_test)), np.log(y_test), 'ro')
    plt.xlabel('Predicted RUL')
    plt.ylabel('Real RUL')
    plt.plot(range(6), range(6))

    # compare predict RUL and real RUL
    plt.figure(2)
    plt.plot(ls.predict(X_test), y_test, 'ro')
    plt.xlabel('Predicted RUL')
    plt.ylabel('Real RUL')
    plt.title('Comparison of Predicted RUL and Real RUL')
    plt.axis([-100, 400, 0, 400])
    plt.plot(range(300), range(300))  # plot the line y = x of perfect
prediction
```

Use LassoCV function to estimate, and obtain the result in Fig.5.7. We can use alpha_() function to get the optimal value, and output its coefficient to know which sensor can be used in estimating (Fig.5.8).

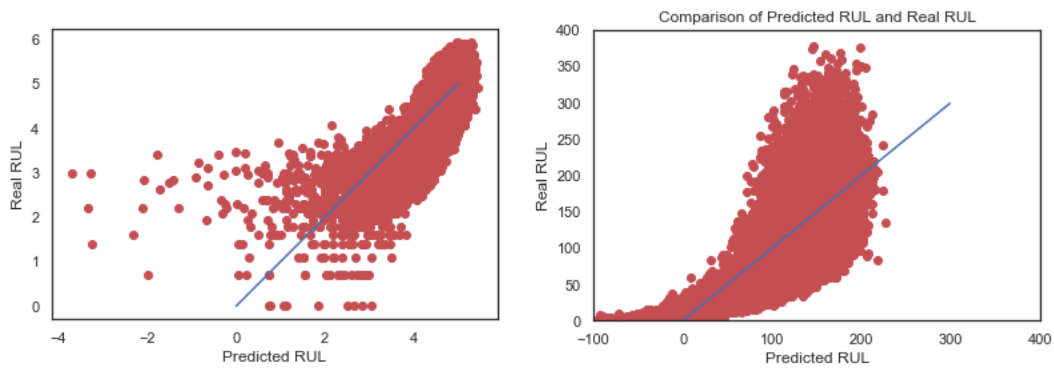


Fig.5.8 Comparison of Predicted RUL and Real RUL

```
Optimal value:  4.4302205772772525e-06
[ 1.90109755e+04 -6.20733614e+04 -7.36232016e+03  1.57382354e+06
 -3.25539089e+05 -3.93507505e+05 -4.57621062e+05 -6.87938514e+04
  3.22857959e+03  2.17841147e+04  1.13866064e+06 -4.56482021e+04
 -1.91098994e+04 -7.61664273e+05 -1.00133763e+03  2.51902190e+05
 -1.08411069e+06 -7.65798976e+05 -2.96750304e+03 -2.65420392e+05
  2.77331340e+05 -6.71506851e+05 -3.39633814e+04 -6.24922954e+03]
Useful sensors to predict RUL:  Index(['os1', 'os2', 'os3', 's1', 's2', 's3', 's4', 's5', 's6', 's7', 's8',
       's9', 's10', 's11', 's12', 's13', 's14', 's15', 's16', 's17', 's18',
       's19', 's20', 's21'],
      dtype='object')
```

Fig.5.9 Optimal value, coefficient and useful sensors

B. Exponential target

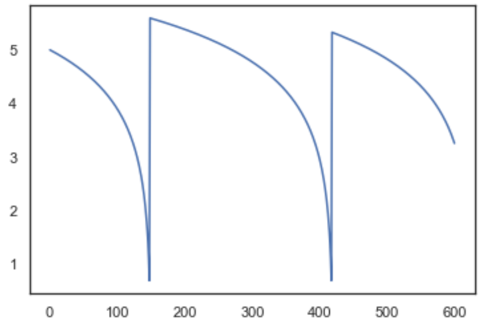


Fig.5.10 Exponential target

```
    # Now let's try do log transformation to create exponential-degenerating
RUL

    plt.figure(1)
    train['rul'] = train[['id', 'te']].groupby('id').transform(f1)
    plt.plot(train.rul[1:600])

    y = train['rul']
    features = train.columns.drop(['id', 'te', 'rul'])
    X = train[features]
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=12)

    # try Lasso
    from sklearn.linear_model import LassoCV

    ls = LassoCV(random_state=12)
    ls = ls.fit(X_train, y_train)

    #print('List of tried parameter values: ', ls.alphas_)
    print('Optimal value: ', ls.alpha_)
    print(ls.coef_)
    print('Useful sensors to predict RUL: ', X_train.columns[abs(ls.coef_) >
1e-6])

    # compare predict RUL and real RUL
    plt.figure(2)
    plt.plot(ls.predict(X_test), y_test, 'ro')
    plt.xlabel('Predicted RUL')
    plt.ylabel('Real RUL')
    plt.plot(range(6), range(6))
    plt.axis([-2, 6, 0, 6])
```

```
plt.figure(3)
plt.plot(np.exp(ls.predict(X_test)), np.exp(y_test), 'ro')
plt.xlabel('Predicted RUL')
plt.ylabel('Real RUL')
plt.title('Comparison of Predicted RUL and Real RUL')
plt.axis([-100, 400, 0, 400])
plt.plot(range(300), range(300))
```

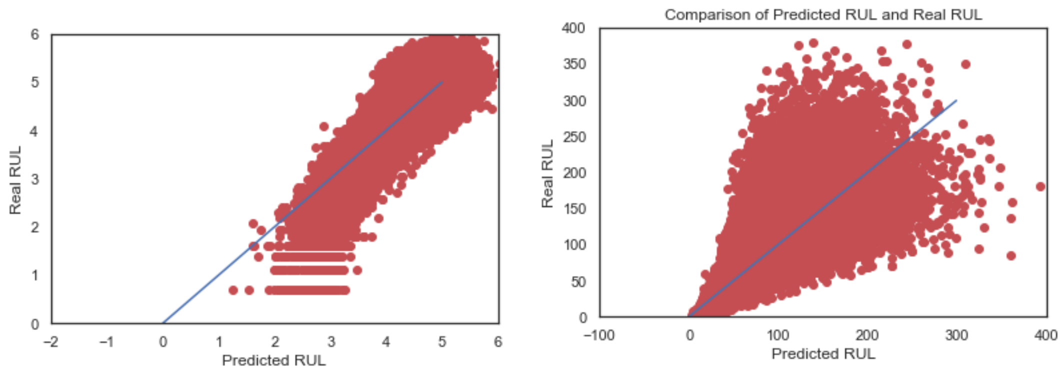


Fig.5.11 Comparison of Predicted RUL and Real RUL

```
Optimal value:  0.00975280076258
[ 0.          0.          0.          0.          -0.          -0.04008465
 -0.06005507  0.          0.          0.02878443 -0.          -0.011902    0.
 -0.          0.         -0.          0.00314782 -0.           0.
 -0.02156106  0.          0.          0.           0.        ]
Useful sensors to predict RUL:  Index(['s3', 's4', 's7', 's9', 's14', 's17'], dtype='object')
```

Fig.5.12 Optimal value, coefficient and useful sensors

We can find that Fig.5.8 & Fig.5.11 and Fig.5.9 &Fig.5.12 exist something different because of the different target signal.

- **Random Forest**

Random forest is an ensemble learning method used for classification, regression and other tasks. Random Forest builds a set of decision trees. Each tree is developed from a bootstrap sample from the training data. When developing individual trees, an arbitrary subset of attributes is drawn (hence the term “Random”), from which the best attribute for the split is selected. The final model is based on the majority vote from individually developed trees in the forest.

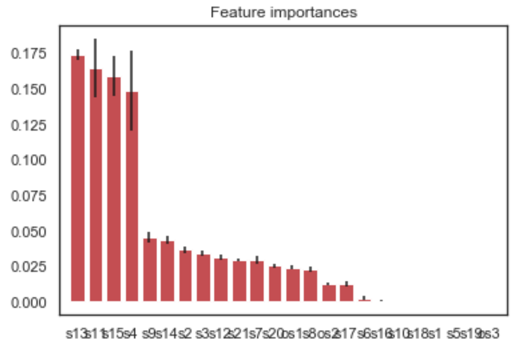


Fig.5.13 the feature importances of the forest

A. Linear target

```
    def f1(col):
        # Option 1: Reverse the time evolution, where remaining time of a
machine is 1 at the failure.
        return col[::-1]

    plt.figure(1)
    train['rul'] = train[['id', 'te']].groupby('id').transform(f1)
    plt.plot(train.rul[1:600])

    y = train['rul']
    features = train.columns.drop(['id', 'te', 'rul'])
    X = train[features]
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=12)

    # try Random Forest
    rf = RandomForestRegressor(random_state=12)
    rf = rf.fit(X_train, y_train)

    # Feature importance
    importances = rf.feature_importances_
    std = np.std([tree.feature_importances_ for tree in rf.estimators_],
                 axis=0)
    indices = np.argsort(importances)[::-1]

    # Plot the feature importances of the forest
    plt.figure()
    plt.title("Feature importances")
    plt.bar(range(X_train.shape[1]), importances[indices],
           color="r", yerr=std[indices], align="center")
    plt.xticks(range(X_train.shape[1]), X_train.columns[indices])
    plt.xlim([-1, X_train.shape[1]])
    plt.show()


    # compare predict RUL and real RUL
    plt.figure(2)
    plt.plot(np.log(rf.predict(X_test)), np.log(y_test), 'ro')
    plt.xlabel('Predicted RUL')
    plt.ylabel('Real RUL')
    plt.plot(range(6), range(6))

    plt.figure(3)
```

```
    plt.plot(rf.predict(X_test), y_test, 'ro')
    plt.xlabel('Predicted RUL')
    plt.ylabel('Real RUL')
    plt.title('Comparison of Predicted RUL and Real RUL')
    plt.plot(range(300), range(300))
```

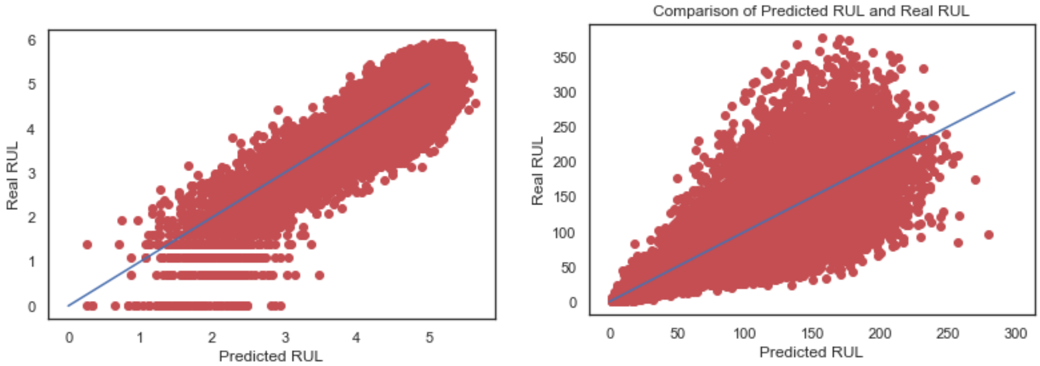


Fig.5.14 Comparison of Predicted RUL and Real RUL

B. Exponential target

```
    # Now let's try do log transformation to create exponential-degenerating
RUL

    def f1(col):
        # Option 2: transform time evolution into exponential-degenerating
remaining health index
        return np.log(col[::-1] + 1)

    plt.figure(1)
    train['rul'] = train[['id', 'te']].groupby('id').transform(f1)
    plt.plot(train.rul[1:600])

    y = train['rul']
    features = train.columns.drop(['id', 'te', 'rul'])
    X = train[features]
    X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3,
random_state=12)

    # try Random Forest
    from sklearn.ensemble import RandomForestRegressor

    rf = RandomForestRegressor(random_state=12)
    rf = rf.fit(X_train, y_train)

    # compare predict RUL and real RUL
    plt.figure(2)
    plt.plot(rf.predict(X_test), y_test, 'ro')
    plt.xlabel('Predicted RUL')
    plt.ylabel('Real RUL')
    plt.plot(range(6), range(6))
    #plt.axis([-2, 6, 0, 6])

    plt.figure(3)
    plt.plot(np.exp(rf.predict(X_test)), np.exp(y_test), 'ro')
    plt.xlabel('Predicted RUL')
    plt.ylabel('Real RUL')
```

```
    plt.title('Comparison of Predicted RUL and Real RUL')
    plt.plot(range(300), range(300))
```

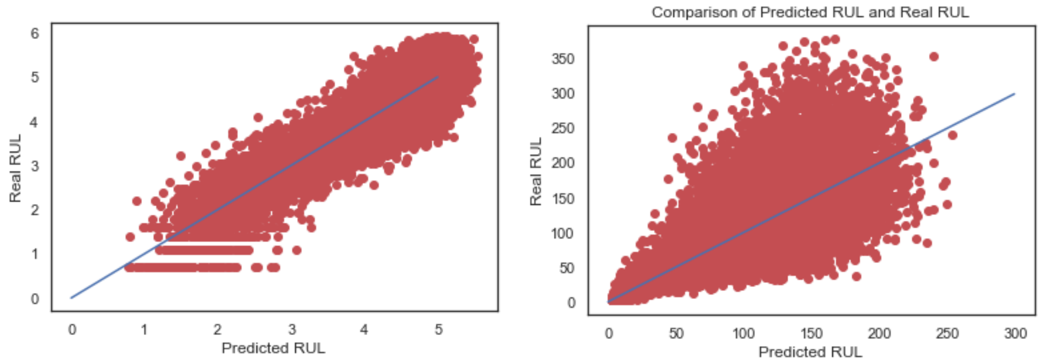


Fig.5.15 Comparison of Predicted RUL and Real RUL

6. Step 4 : Use other data sources

I will use other data to repeat this experiment again.

6.1 Publish data to DynamoDB

Create a new DynamoDB table named 'MotorVehiclePopulation', and use year (Number) as primary key (Fig.6.1). Then set an action and choose 'MotorVehiclePopulation' as the source (Fig.6.2). Finally, connect the DynamoDB and publish data (Fig.6.3).

Table name	MotorVehiclePopulation
Primary partition key	year (Number)
Primary sort key	-
Point-in-time recovery	DISABLED **Enable**
Encryption Type	DEFAULT **Manage Encryption**
KMS Master Key ARN	Not Applicable
Time to live attribute	DISABLED **Manage TTL**
Table status	Active
Creation date	September 20, 2019 at 3:43:14 PM UTC+8
Read/write capacity mode	Provisioned
Last change to on-demand mode	-
Provisioned read capacity units	5 (Auto Scaling Disabled)
Provisioned write capacity units	5 (Auto Scaling Disabled)
Last decrease time	-
Last increase time	-
Storage size (in bytes)	0 bytes
Item count	0 **Manage live count**
Region	Asia Pacific (Singapore)
Amazon Resource Name (ARN)	arn:aws:dynamodb:ap-southeast-1:894491098965:table/MotorVehiclePopulation

Fig.6.1 MotorVehiclePopulation table

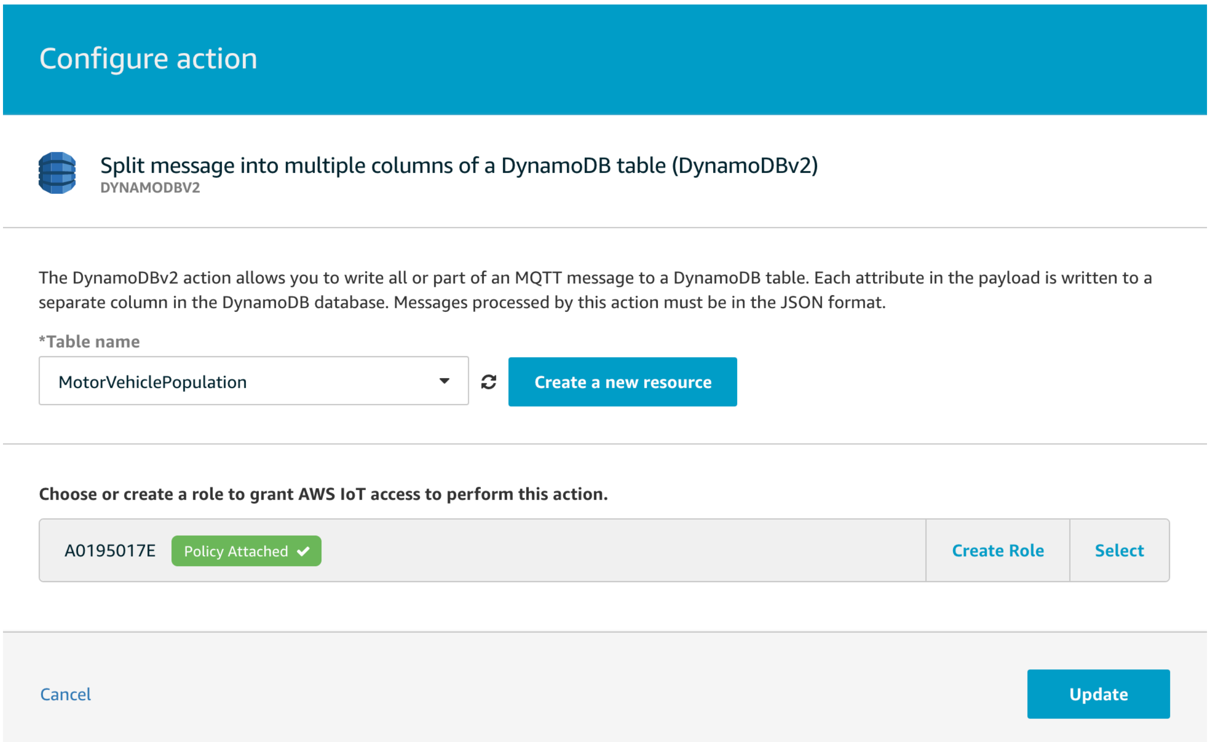


Fig.6.2 configure action

year	category	number	type
2005	Tax Exempted Vehicles	10905	Goods & Other Vehicles
2006	Tax Exempted Vehicles	11625	Goods & Other Vehicles
2007	Tax Exempted Vehicles	12375	Goods & Other Vehicles
2008	Tax Exempted Vehicles	13123	Goods & Other Vehicles
2009	Tax Exempted Vehicles	13405	Goods & Other Vehicles
2010	Tax Exempted Vehicles	13928	Goods & Other Vehicles
2011	Tax Exempted Vehicles	14610	Goods & Other Vehicles
2012	Tax Exempted Vehicles	15371	Goods & Other Vehicles

Fig.6.3 data in DynamoDB

6.2 Visualization

Next, pull data from DynamoDB to visualise.

```
    #visualization
    year=list(range(2005,2017))

    category1=mvp.loc[(mvp['category']=='Cars & Station-
wagons')&(mvp['type']=='Private cars'),'number']
```

```
    category2=mvp.loc[(mvp['category']=='Cars & Station-
wagons')&(mvp['type']=='Company cars'),'number']
    category3=mvp.loc[(mvp['category']=='Cars & Station-
wagons')&(mvp['type']=='Tuition cars'),'number']
    category4=mvp.loc[(mvp['category']=='Cars & Station-
wagons')&(mvp['type']=='Rental cars'),'number']
    category5=mvp.loc[(mvp['category']=='Cars & Station-
wagons')&(mvp['type']=='Off peak cars'),'number']

    plt.figure(1)
    plt.xlabel('Years')
    plt.ylabel('Number')
    plt.bar(year,category1.values,label='Private cars')
    plt.legend()

    plt.figure(2)
    plt.xlabel('Years')
    plt.ylabel('Number')
    width=0.2
    plt.bar(year, category2, width=width,label='Company cars')
    for i in range(len(year)):
        year[i] = year[i] + width
    plt.bar(year, category4, width=width,label='Rental cars')
    for i in range(len(year)):
        year[i] = year[i] + width
    plt.bar(year, category5, width=width,label='Off peak cars')
    plt.legend()

    plt.figure(3)
    plt.xlabel('Years')
    plt.ylabel('Number')
    plt.plot(year,category3.values,label='Tuition cars')
    plt.legend()
```

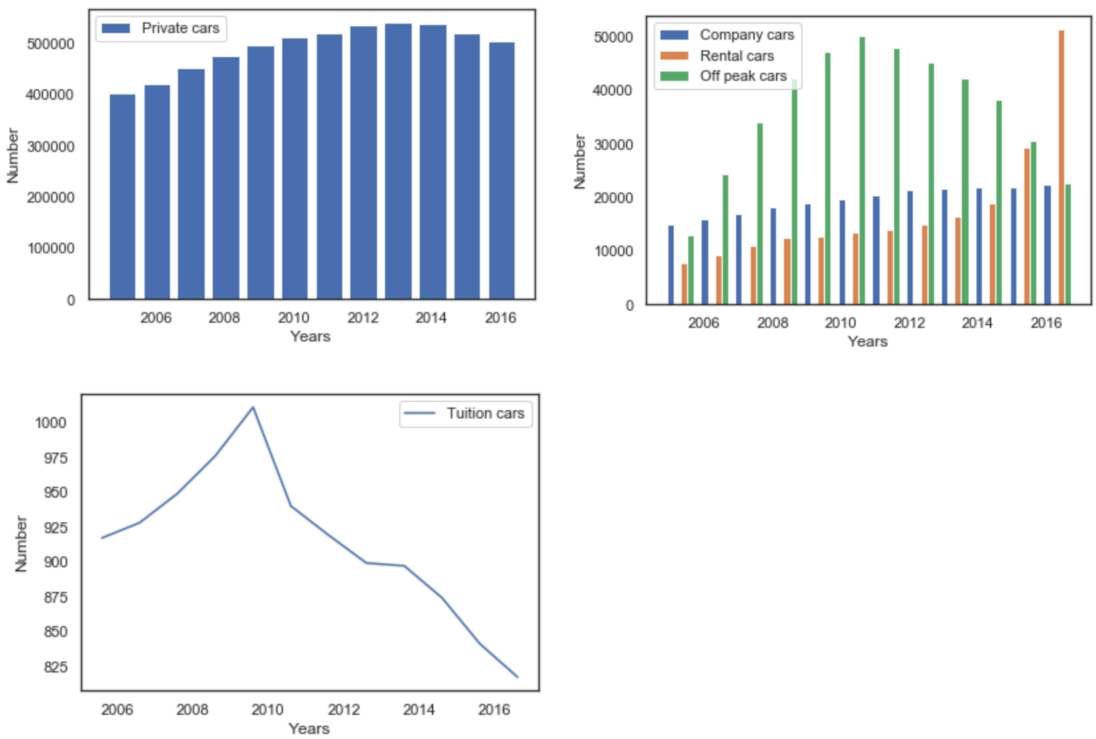


Fig.6.4 the visualization of other source