Big data analytics for IoT
Using available water pump sensor data to predict pump failure in the future

Project Report

Course 055153: Big Data

Instructor: *Assoc. Prof. Dr.Thoai Nam*

Department: Faculty of Computer Science and Engineering

Date: August 22, 2024

	Tran Tho Nhan	1910405
Master Students:	Dinh Thanh Phong	2270243
	Nguyen Tran Thao Van	2170586

Contents

1 Introduction

This project we are trying to apply big data tools to solve the actual problem come from water pump.

Our group has tried to use as many tools as possible for this project. This project is really useful to help us in understanding and applying some famous framework for big data such as Kafka, Spark, Python, Linux...

Big data analytics for IoT-based water pump sensors:
+ A tool generating data streaming like water pump sensors
+ Data collection: using Kafka
+ Data analytics: filtering, sampling, integration – using Spark
+ Analysis and visualization: using Python
+ Functions and performance evaluation

2 Data descriptions, motivations and challenges

The data used in this project come from an open source on Kaggle. Kaggle is a data science competition platform and online community of data scientists and machine learning practitioners under Google LLC.

The data name is: pump-sensor-data
Access link: https://www.kaggle.com/datasets/nphantawee/pump-sensor-data/data
File type: .csv
File size: 124.06 MB

The data come from a small team that taking care of water pump of a small area far from big town, there are 7 system failure in last year. Those failure cause huge problem to many people and also lead to some serious living problem of some family. The team can't see any pattern in the data when the system goes down, so they are not sure where to put more attention.

They believe in using data to solve problem, they provide available sensor data and hope that someone can help.

The data are from all available sensor, all of them are raw value. Total sensor are 52 units.
The data includes 55 columns with 220320 recordings.

The date contain 3 main group of data
- Timestamp data
- Sensor data(52 series): All values are raw values
- Machine status: This is target label that I want to predict when the failure will happen

The parameters are along these lines:
SENSOR00 - Motor Casing Vibration
SENSOR01 - Motor Frequency A
SENSOR02 - Motor Frequency B
SENSOR03 - Motor Frequency C
SENSOR04 - Motor Speed
SENSOR05 - Motor Current
SENSOR06 - Motor Active Power
SENSOR07 - Motor Apparent Power
SENSOR08 - Motor Reactive Power
SENSOR09 - Motor Shaft Power
SENSOR10 - Motor Phase Current A
SENSOR11 - Motor Phase Current B
SENSOR12 - Motor Phase Current C
SENSOR13 - Motor Coupling Vibration
SENSOR14 - Motor Phase Voltage AB
SENSOR16 - Motor Phase Voltage BC
SENSOR17 - Motor Phase Voltage CA
SENSOR18 - Pump Casing Vibration

SENSOR19 - Pump Stage 1 Impeller Speed
SENSOR20 - Pump Stage 1 Impeller Speed
SENSOR21 - Pump Stage 1 Impeller Speed
SENSOR22 - Pump Stage 1 Impeller Speed
SENSOR23 - Pump Stage 1 Impeller Speed
SENSOR24 - Pump Stage 1 Impeller Speed
SENSOR25 - Pump Stage 2 Impeller Speed
SENSOR26 - Pump Stage 2 Impeller Speed
SENSOR27 - Pump Stage 2 Impeller Speed
SENSOR28 - Pump Stage 2 Impeller Speed
SENSOR29 - Pump Stage 2 Impeller Speed
SENSOR30 - Pump Stage 2 Impeller Speed
SENSOR31 - Pump Stage 2 Impeller Speed
SENSOR32 - Pump Stage 2 Impeller Speed
SENSOR33 - Pump Stage 2 Impeller Speed
SENSOR34 - Pump Inlet Flow
SENSOR35 - Pump Discharge Flow
SENSOR36 - Pump UNKNOWN
SENSOR37 - Pump Lube Oil Overhead Reservoir Level
SENSOR38 - Pump Lube Oil Return Temp
SENSOR39 - Pump Lube Oil Supply Temp
SENSOR40 - Pump Thrust Bearing Active Temp
SENSOR41 - Motor Non Drive End Radial Bearing Temp 1
SENSOR42 - Motor Non Drive End Radial Bearing Temp 2
SENSOR43 - Pump Thrust Bearing Inactive Temp
SENSOR44 - Pump Drive End Radial Bearing Temp 1
SENSOR45 - Pump non Drive End Radial Bearing Temp 1
SENSOR46 - Pump Non Drive End Radial Bearing Temp 2
SENSOR47 - Pump Drive End Radial Bearing Temp 2
SENSOR48 - Pump Inlet Pressure
SENSOR49 - Pump Temp Unknown
SENSOR50 - Pump Discharge Pressure 1
SENSOR51 - Pump Discharge Pressure 2
Pump Status

This project is trying to build a simulation model to correlate/ predict whether the pump may be shut down (broken) for a period of time based on the input signal of the sensors.

3 Used tools and applications

In this project, we are trying to use as many tools or systems as possible to understand how to deal with big data problems.
Some tools and system that we used in this project such as Ubuntu Linux, IBA-Group-IT/IoT-data-simulator, Kafka, Spark, Python and also knowledge about statistic to evaluate the result.

3.1 Ubuntu Linux

Ubuntu is a popular free and open-source Linux-based operating system you can use on a computer or virtual private server.
Ubuntu was introduced in 2004 by a British company Canonical. It was based on Debian – a popular distro back then – which was difficult to install. As a result, Ubuntu was proposed as a more user-friendly alternative.

Linux is a family of operating systems based on the Linux kernel – the core of an operating system. It enables the communication between hardware and software components.
Linux is based on Unix and built around the Linux kernel. It was released in 1991 and is available for web servers, gaming consoles, embedded systems, desktops, and personal computers. It comes in many different versions called distributions.

Figure 1: Ubuntu-Linux

3.2 IBA-Group-IT/IoT-data-simulator

An IoT data simulator is a required tool in any IoT project. While using real world sensors and devices is required for final integration testing to make sure the system works end to end without any surprises, it is very impractical to use the real devices during development and initial testing phases where tests tend to be quicker, shorter, failing quicker and more often.

Data simulator solves several problems at once:
1. It is not necessary to run the physical equipment just to perform a unit or early integration test.

2. You can use synthetic generated data if you do not yet have access to the physical devices, or you can replay captured data to get it as real as possible.
3. By imploding the data with some randomization, you can simulate a large number of devices that you may not be able to get hold of during the test and development.
4. You can replay data at various frequencies to run the test quicker (e.g. when the real world data is pushed every 5 minutes and it takes several days to trigger an alert).
5. You can simulate data from different stages of the system, so that the testing of the later stages does not have to wait until all the previous stages are completed to transform raw data into a suitable input. For example, you can simulate raw device data to develop and test initial processing, simulate "cleaned up" data stream to develop and test analytic components, and simulate analytic component results to develop and test visualization and UX.

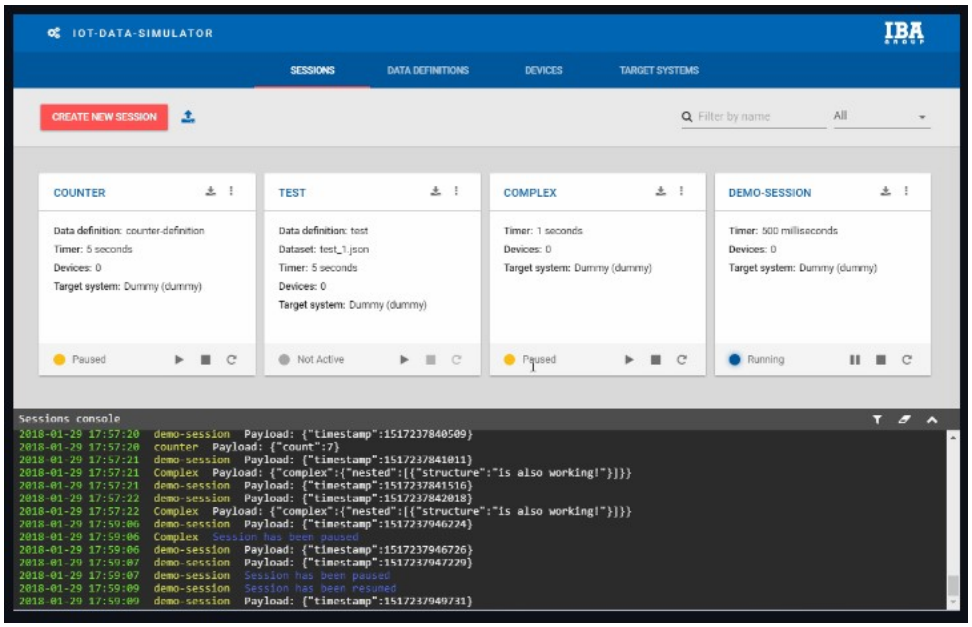


Figure 2: IBA-Group-IT/IoT-data-simulator

IoT-data-simulator is the tool which allows you to simulate IoT devices data with great flexibility. With this tool you won't need to code another new simulator for each IoT project.

Simulator features that you will like:
1. Replay existing datasets with modified data (such as updated timestamps, generated ids etc);
2. Automatic derivation of dataset structure which allows you to customize your dataset without the need to describe its structure from scratch;
3. Generate datasets of any complexity. Generated data can be described via constructor or JavaScript function. Multiple rules are available in constructor such as "Random integer", "UUID" and others. JS function gives you maximum flexibility to generate data - it supports popular JS libraries lodash and momentjs.
4. Send data to the platforms you use with minimum configuration (see Supported target systems section);
5. Customize frequency with which data will be sent - based on dataset timestamp properties or just constant time interval. The tool also supports relative timestamp properties which depend

on other timestamp or date properties. It means that data can be replayed with the same interval between timestamps as in initial dataset.
Easy installation - all you need is to download 2 docker files and run 2 commands.

3.3 Kafka

Apache Kafka® is an event streaming platform.
Kafka combines three key capabilities so you can implement your use cases for event streaming end-to-end with a single battle-tested solution:

1. To publish (write) and subscribe to (read) streams of events, including continuous import/export of your data from other systems.
2. To store streams of events durably and reliably for as long as you want.
3. To process streams of events as they occur or retrospectively.

And all this functionality is provided in a distributed, highly scalable, elastic, fault-tolerant, and secure manner. Kafka can be deployed on bare-metal hardware, virtual machines, and containers, and on-premises as well as in the cloud. You can choose between self-managing your Kafka environments and using fully managed services offered by a variety of vendors.

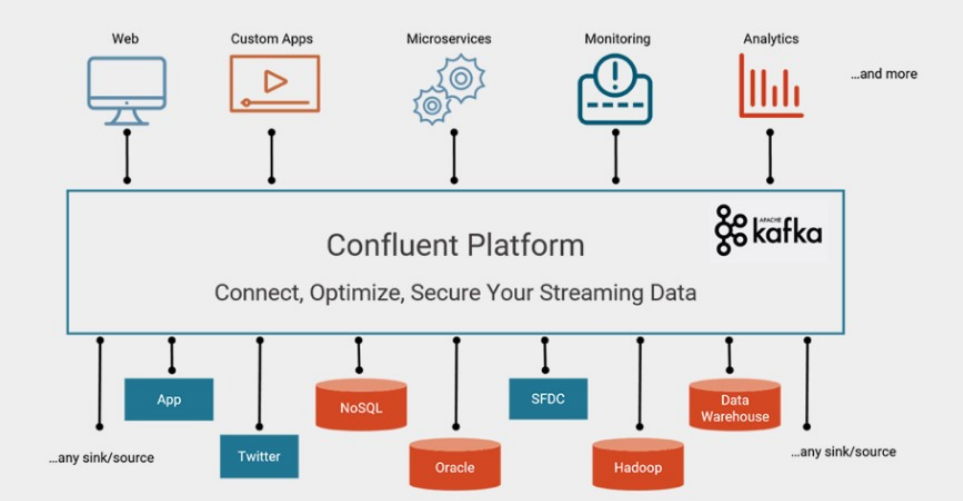


Figure 3: Apache Kafka

Main Concepts and Terminology: An event records the fact that ”something happened” in the world or in your business. It is also called record or message in the documentation. When you read or write data to Kafka, you do this in the form of events. Conceptually, an event has a key, value, timestamp, and optional metadata headers. Here’s an example event:
- Event key: ”Alice”
- Event value: ”Made a payment of 200 Dollar to Bob”
- Event timestamp: ”Jun. 25, 2020 at 2:06 p.m.”
Producers are those client applications that publish (write) events to Kafka, and consumers are those that subscribe to (read and process) these events. In Kafka, producers and consumers are

fully decoupled and agnostic of each other, which is a key design element to achieve the high scalability that Kafka is known for. For example, producers never need to wait for consumers. Kafka provides various guarantees such as the ability to process events exactly-once.

Events are organized and durably stored in topics. Very simplified, a topic is similar to a folder in a filesystem, and the events are the files in that folder. An example topic name could be ”payments”. Topics in Kafka are always multi-producer and multi-subscriber: a topic can have zero, one, or many producers that write events to it, as well as zero, one, or many consumers that subscribe to these events. Events in a topic can be read as often as needed—unlike traditional messaging systems, events are not deleted after consumption. Instead, you define for how long Kafka should retain your events through a per-topic configuration setting, after which old events will be discarded. Kafka’s performance is effectively constant with respect to data size, so storing data for a long time is perfectly fine.

Topics are partitioned, meaning a topic is spread over a number of ”buckets” located on different Kafka brokers. This distributed placement of your data is very important for scalability because it allows client applications to both read and write the data from/to many brokers at the same time. When a new event is published to a topic, it is actually appended to one of the topic’s partitions. Events with the same event key (e.g., a customer or vehicle ID) are written to the same partition, and Kafka guarantees that any consumer of a given topic-partition will always read that partition’s events in exactly the same order as they were written.

To make your data fault-tolerant and highly-available, every topic can be replicated, even across geo-regions or datacenters, so that there are always multiple brokers that have a copy of the data just in case things go wrong, you want to do maintenance on the brokers, and so on. A common production setting is a replication factor of 3, i.e., there will always be three copies of your data. This replication is performed at the level of topic-partitions.

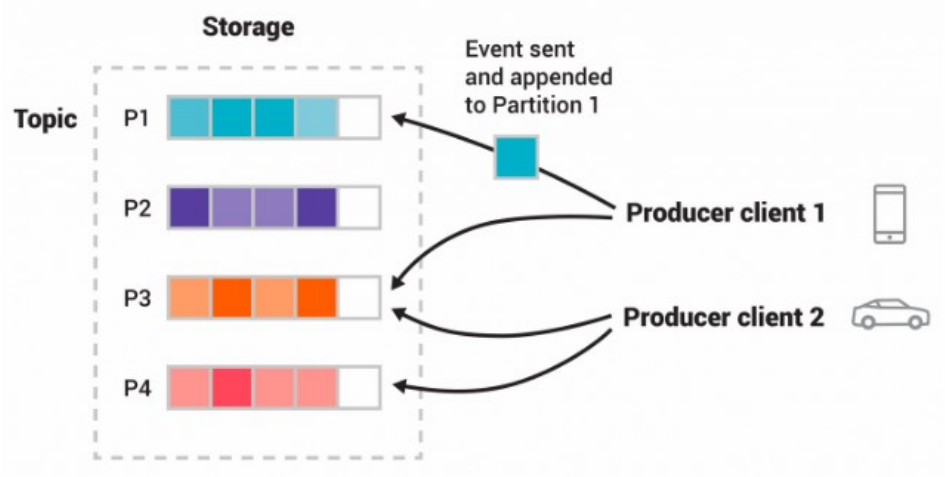


Figure 4: This example Kafka topic has four partitions P1–P4

In addition to command line tooling for management and administration tasks, Kafka has five core APIs for Java and Scala:

- The Admin API to manage and inspect topics, brokers, and other Kafka objects.
- The Producer API to publish (write) a stream of events to one or more Kafka topics.
- The Consumer API to subscribe to (read) one or more topics and to process the stream of events produced to them.
- The Kafka Streams API to implement stream processing applications and microservices. It provides higher-level functions to process event streams, including transformations, stateful operations like aggregations and joins, windowing, processing based on event-time, and more. Input is read from one or more topics in order to generate output to one or more topics, effectively transforming the input streams to output streams.
- The Kafka Connect API to build and run reusable data import/export connectors that consume (read) or produce (write) streams of events from and to external systems and applications so they can integrate with Kafka. For example, a connector to a relational database like PostgreSQL might capture every change to a set of tables. However, in practice, you typically don't need to implement your own connectors because the Kafka community already provides hundreds of ready-to-use connectors.

3.4 Spark

Apache Spark is a lightning-fast cluster computing technology, designed for fast computation. It is based on Hadoop MapReduce and it extends the MapReduce model to efficiently use it for more types of computations, which includes interactive queries and stream processing. The main feature of Spark is its in-memory cluster computing that increases the processing speed of an application.

Spark is designed to cover a wide range of workloads such as batch applications, iterative algorithms, interactive queries and streaming. Apart from supporting all these workload in a respective system, it reduces the management burden of maintaining separate tools.

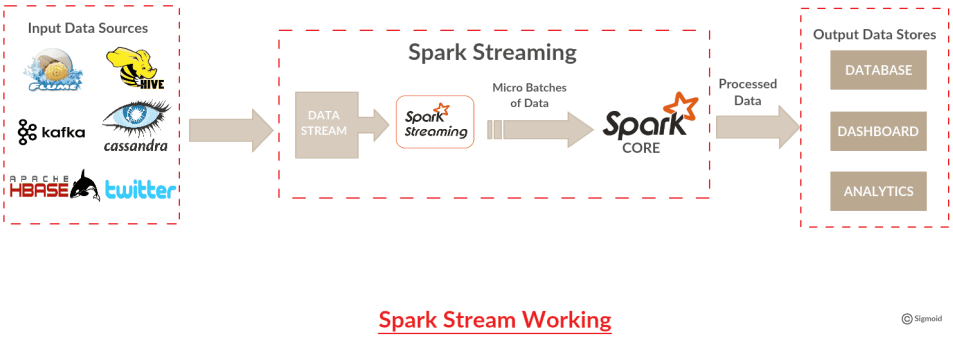


Figure 5: Apache Spark

Apache Spark has following features

- Speed: Spark helps to run an application in Hadoop cluster, up to 100 times faster in memory, and 10 times faster when running on disk. This is possible by reducing number of read/write operations to disk. It stores the intermediate processing data in memory.

- Supports multiple languages: Spark provides built-in API in Java, Scala, or Python. Therefore, you can write applications in different languages. Spark comes up with 80 high-level operators for interactive querying.
- Advanced Analytics: Spark not only supports 'Map' and 'reduce'. It also supports SQL queries, Streaming data, Machine learning (ML), and Graph algorithms.

3.5 Python

Python is an interpreted, object-oriented, high-level programming language with dynamic semantics. Its high-level built in data structures, combined with dynamic typing and dynamic binding, make it very attractive for Rapid Application Development, as well as for use as a scripting or glue language to connect existing components together. Python's simple, easy to learn syntax emphasizes readability and therefore reduces the cost of program maintenance. Python supports modules and packages, which encourages program modularity and code reuse. The Python interpreter and the extensive standard library are available in source or binary form without charge for all major platforms, and can be freely distributed.

Often, programmers fall in love with Python because of the increased productivity it provides. Since there is no compilation step, the edit-test-debug cycle is incredibly fast. Debugging Python programs is easy: a bug or bad input will never cause a segmentation fault. Instead, when the interpreter discovers an error, it raises an exception. When the program doesn't catch the exception, the interpreter prints a stack trace. A source level debugger allows inspection of local and global variables, evaluation of arbitrary expressions, setting breakpoints, stepping through the code a line at a time, and so on. The debugger is written in Python itself, testifying to Python's introspective power. On the other hand, often the quickest way to debug a program is to add a few print statements to the source: the fast edit-test-debug cycle makes this simple approach very effective.

4 Implementation and results

4.1 IBA-Group-IT/IoT-data-simulator generate streaming data then send data to Kafka topic

4.1.1 Install and create Kafka topic

Firstly, we need to start ZooKeeper and Kafka environment.
Once all services have successfully launched, you will have a basic Kafka environment running and ready to use.

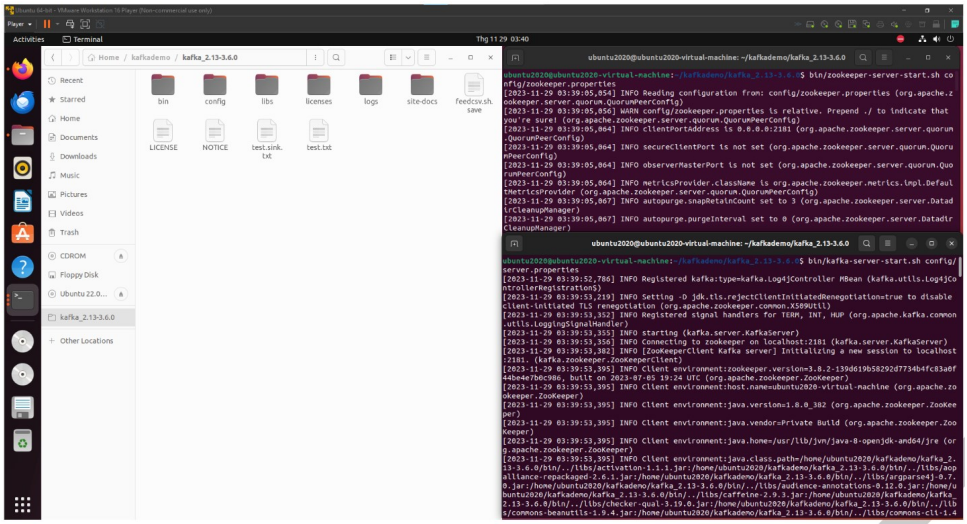


Figure 6: Set up Kafka environment

At the server.properties , we need to assign IP addresses for Kafka topics.
In this project, we chose IP and port as: 172.18.0.1:9092
And we created the topics name: connect-test

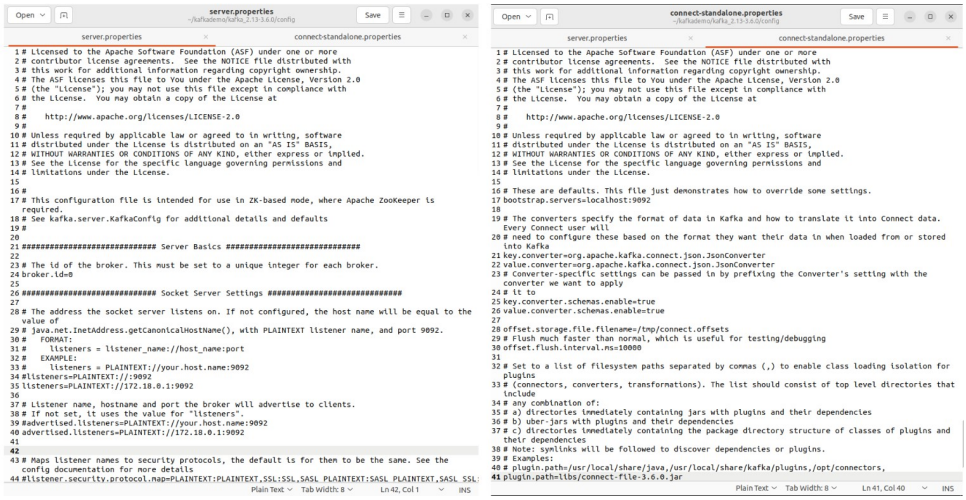


Figure 7: Set up Kafka connect

4.1.2 Generating streaming data by IoT-data-Simulator

In folder IoT-data-simulator-master that we download from the Simulator website.
Prerequisites docker (v. 17.05+) and docker-compose should be installed.
Startup Run the commands “docker-compose up” in the folder with docker-compose.yml and .env files from release folder.

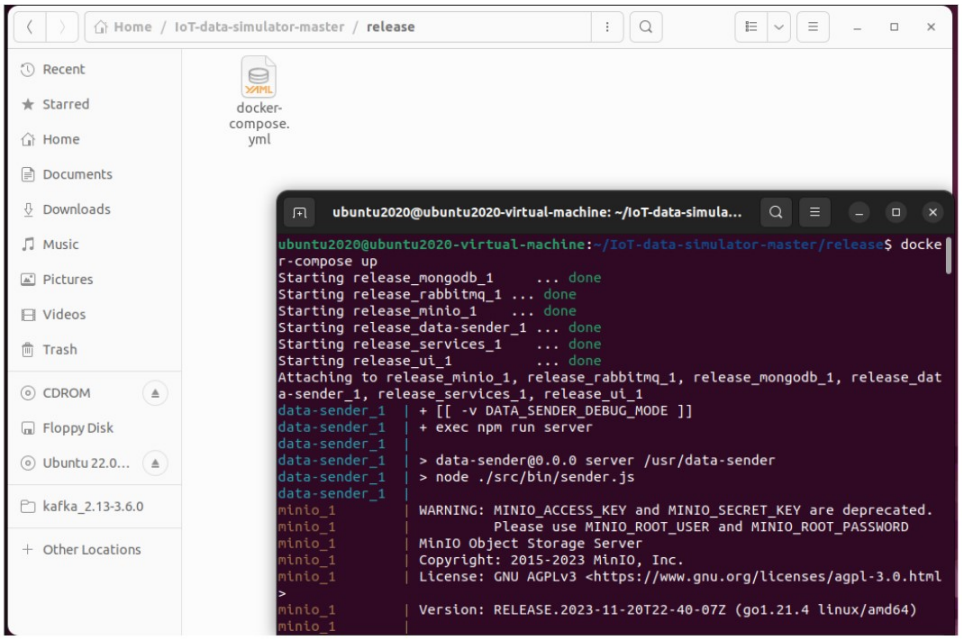


Figure 8: Start up IoT simulator

After a while UI will be available by the following url: http://localhost:8090

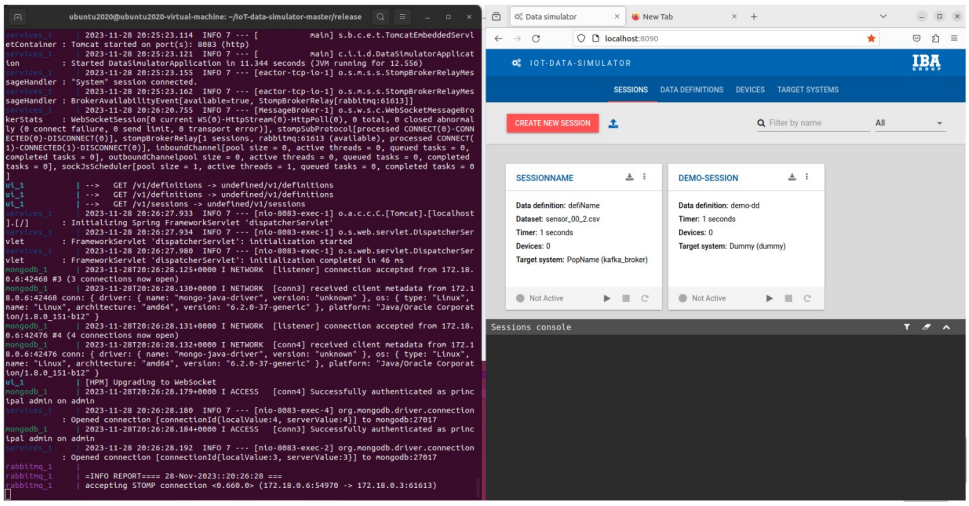


Figure 9: Open UI simulator

We need to upload the csv data file

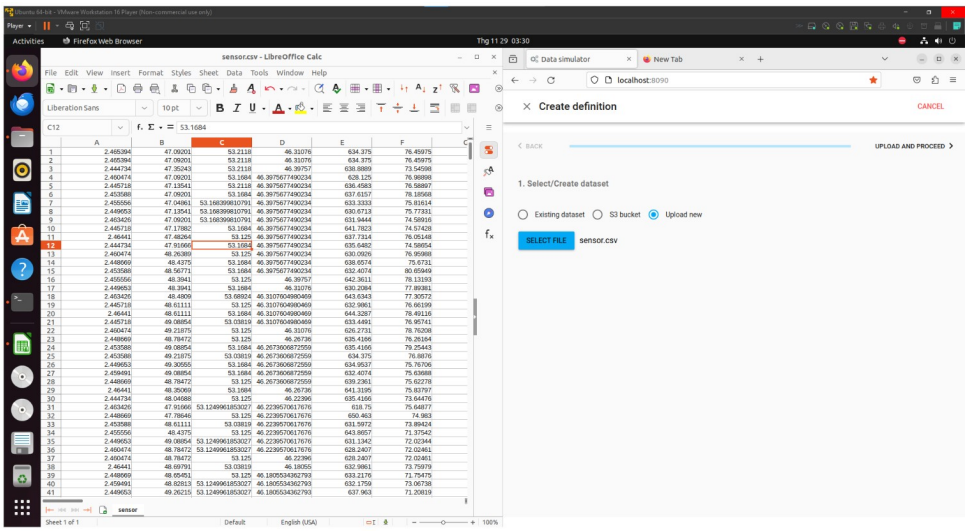


Figure 10: Upload data

The simulator will recognize the data type.
We can set up the interval like seconds.

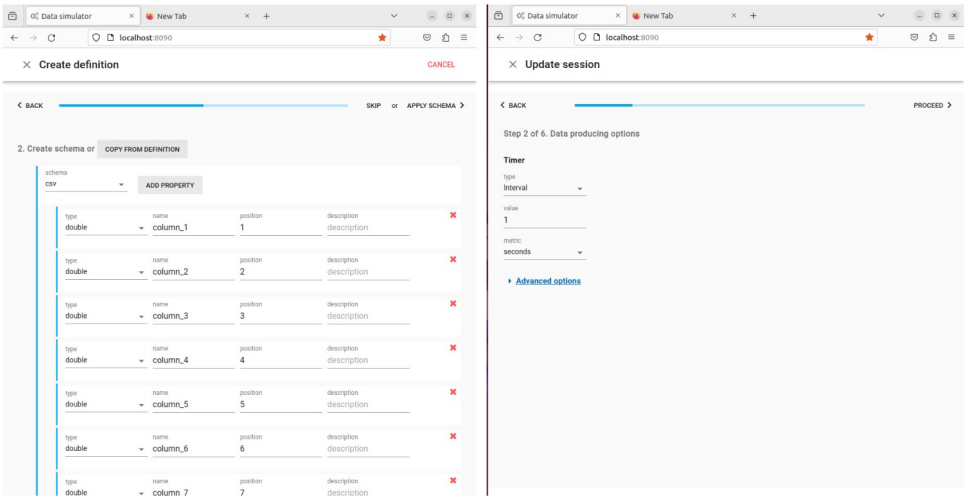


Figure 11: Data type

4.1.3 Auto sending data from simulator to Kafka topic by Kafka Connect

Now we can run the simulator, then the data will send automatically to Kafka topic every second.
We can check the data in Kafka topic by open Kafka-console-consumer.sh for connect-test.

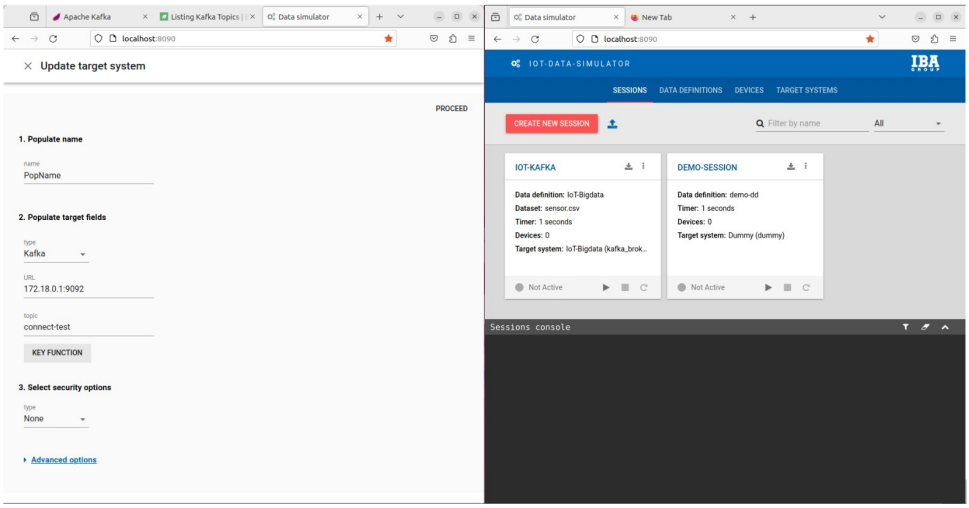


Figure 12: Kafka consumer

Now we finished to collect data from IoT-simulator, then other consumer can connect to Kafka topic to use the data from here.
In this project, Kafka topic is available for Spark to connect to.

4.2 Filtering, sampling, integration – using Spark

4.2.1 Read CSV file

Use Spark to read data in CSV file format

```
import pyspark
from pyspark.sql import SparkSession,Row
from pyspark.sql.functions import col,isnan,when,count,monotonically_incr
spark = SparkSession.builder.appName("watermetering").getOrCreate()
path = "sensor.csv"

df = spark.read.csv(path,header=True, inferSchema=True)
# Print Schema
df.printSchema()
# Print Dataframe
df.show()
```

4.2.2 Data Cleaning

After reading the data, check for columns with a large number of NULL data and remove those columns.

```
df2 = df.select([count(when(col(c).contains('None') | \
                            col(c).contains('NULL') | \
                            (col(c) == '' ) | \
                            col(c).isNull() | \
```

```
                isnan(c), c
               )).alias(c)
        for c in df.columns])
df2.show()

#Null data
df = df.drop('_c0', 'timestamp','sensor_00','sensor_15','sensor_50','sens
df.show()
```

Based on the analysis of sensor performance affecting the operating state of the pump, it is important for modeling. To optimize the model, we select the model's input parameters according to the following hypotheses.

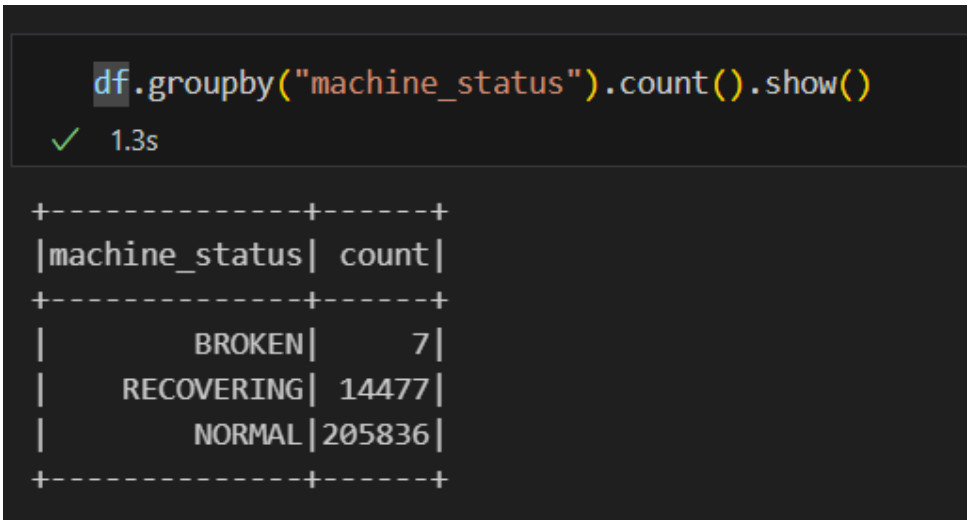


Figure 13: Machine status

The database has 7 BROKEN states, which are then RECOVERED and returned to a NORMAL operating state. For the sack of simplicity, we can assume that 25% of the data could be used to train the model (covering 2 BROKEN states), the remaining 75% of the data is used to test the predictability of the model based on input parameters (covers 5 BROKEN points).
For graphical illustration purpose, we assume the BROKEN state transitions have a value of 0, the RECOVERING state and NORMAL operation value 0.5 and 1, respectively and converted it into a new column named: "Operation".

```
import numpy as np

choices = [1, 0, 0.5]
df = df.withColumn(
     'Operation',
     when(df['machine_status'] == "NORMAL", choices[0])
     .when(df['machine_status'] == "BROKEN", choices[1])
     .when(df['machine_status'] == "RECOVERING", choices[2])
     )
df.show()
```

4.3 Analysis and visualization: using Python

4.3.1 LSTM model

Long Short Term Memory networks – usually just called "LSTMs" – are a special kind of RNN, capable of learning long-term dependencies. They were introduced by Hochreiter and Schmidhuber (1997), and were refined and popularized by many people in following work.1 They work tremendously well on a large variety of problems, and are now widely used.
LSTMs are explicitly designed to avoid the long-term dependency problem. Remembering information for long periods of time is practically their default behavior, not something they struggle to learn!
We divide the dataset into smaller dataset randomly before training the model.
Sensors numbers 4, 6, 7, 8, 9 will be included in the dataset 1. Sensors 1, 4, 10, 14, 19, 25, 34, 38 will be included in the dataset 2. Sensors 2, 5, 11, 16, 20, 26, 39 will be included in the dataset 3 and sensors 3, 6, 12, 17, 21, 28, 40 will be included in the dataset 4.

4.3.2 Traing the model and implement the prediction

- Training set:
We choose 50,000 data points with 2 broken points to train the model,
Testing set:
The remaining 160,000 points with 5 broken states will be used to test the predictivity of the model.
Proceed to train the model with dataset 2.

```
import pandas as pd
def series_to_supervised(data, n_in=1, n_out=1, dropnan=True):
    n_vars = 1 if type(data) is list else data.shape[1]
    dff = pd.DataFrame(data)
    cols, names = list(), list()
    for i in range(n_in, 0, -1):
        cols.append(dff.shift(-i))
        names += [('var%d(t-%d)' % (j+1, i)) for j in range(n_vars)]
    for i in range(0, n_out):
        cols.append(dff.shift(-i))
        if i==0:
            names += [('var%d(t)' % (j+1)) for j in range(n_vars)]
        else:
            names += [('var%d(t+%d)' % (j+1)) for j in range(n_vars)]
        agg = pd.concat(cols, axis=1)
        agg.columns = names
        if dropnan:
            agg.dropna(inplace=True)
        return agg

from sklearn.preprocessing import MinMaxScaler

values = data.values
```

```
scaler = MinMaxScaler(feature_range=(0, 1))
scaled = scaler.fit_transform(values)
reframed = series_to_supervised(scaled, 1, 1)
r = list(range(data.shape[1]+1, 2*data.shape[1]))
reframed.drop(reframed.columns[r], axis=1, inplace=True)
reframed.head()

# Data spliting into train and test data series.
values = reframed.values
n_train_time = 50000
train = values[:n_train_time, :]
test = values[n_train_time:, :]
train_x, train_y = train[:, :-1], train[:, -1]
test_x, test_y = test[:, :-1], test[:, -1]
train_x = train_x.reshape((train_x.shape[0], 1, train_x.shape[1]))
test_x = test_x.reshape((test_x.shape[0], 1, test_x.shape[1]))

    from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dropout
from keras.layers import Dense
from sklearn.metrics import mean_squared_error,r2_score
import matplotlib.pyplot as plt
import numpy as np

model = Sequential()
model.add(LSTM(100, input_shape=(train_x.shape[1], train_x.shape[2])))
model.add(Dropout(0.2))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')

# Network fitting
history = model.fit(train_x, train_y, epochs=50, batch_size=70, validatio

# Loss history plot
plt.plot(history.history['loss'])
plt.plot(history.history['val_loss'])
plt.title('model loss')
plt.ylabel('loss')
plt.xlabel('epoch')
plt.legend(['train', 'test'], loc='upper right')
plt.show()

size = data.shape[1]

# Prediction test
```

```
yhat = model.predict(test_x)
test_x = test_x.reshape((test_x.shape[0], size))

# invert scaling for prediction
inv_yhat = np.concatenate((yhat, test_x[:, 1-size:]), axis=1)
inv_yhat = scaler.inverse_transform(inv_yhat)
inv_yhat = inv_yhat[:,0]

# invert scaling for actual
test_y = test_y.reshape((len(test_y), 1))
inv_y = np.concatenate((test_y, test_x[:, 1-size:]), axis=1)
inv_y = scaler.inverse_transform(inv_y)
inv_y = inv_y[:,0]

# calculate RMSE
rmse = np.sqrt(mean_squared_error(inv_y, inv_yhat))
print('Test RMSE: %.3f' % rmse)
```

After training the model with the dataset including sensors 4, 6, 7, 8, 9. We do the same with the dataset 2

4.3.3 Visualization with python

```
    # Observing the overall distribution
color = [
    'lightgreen',
    'lightgray',
    'black'
]

fig, (ax1, ax2) = plt.subplots(ncols=2, figsize=(20,10))

df.machine_status.value_counts().plot(
    kind='bar',
    ax=ax1,
    color=color,
    edgecolor = 'black',
    linewidth=4
)

df.machine_status.value_counts().plot(
    kind='pie',
    ax=ax2,
    autopct='%.4f%%',
    colors=color
)
```

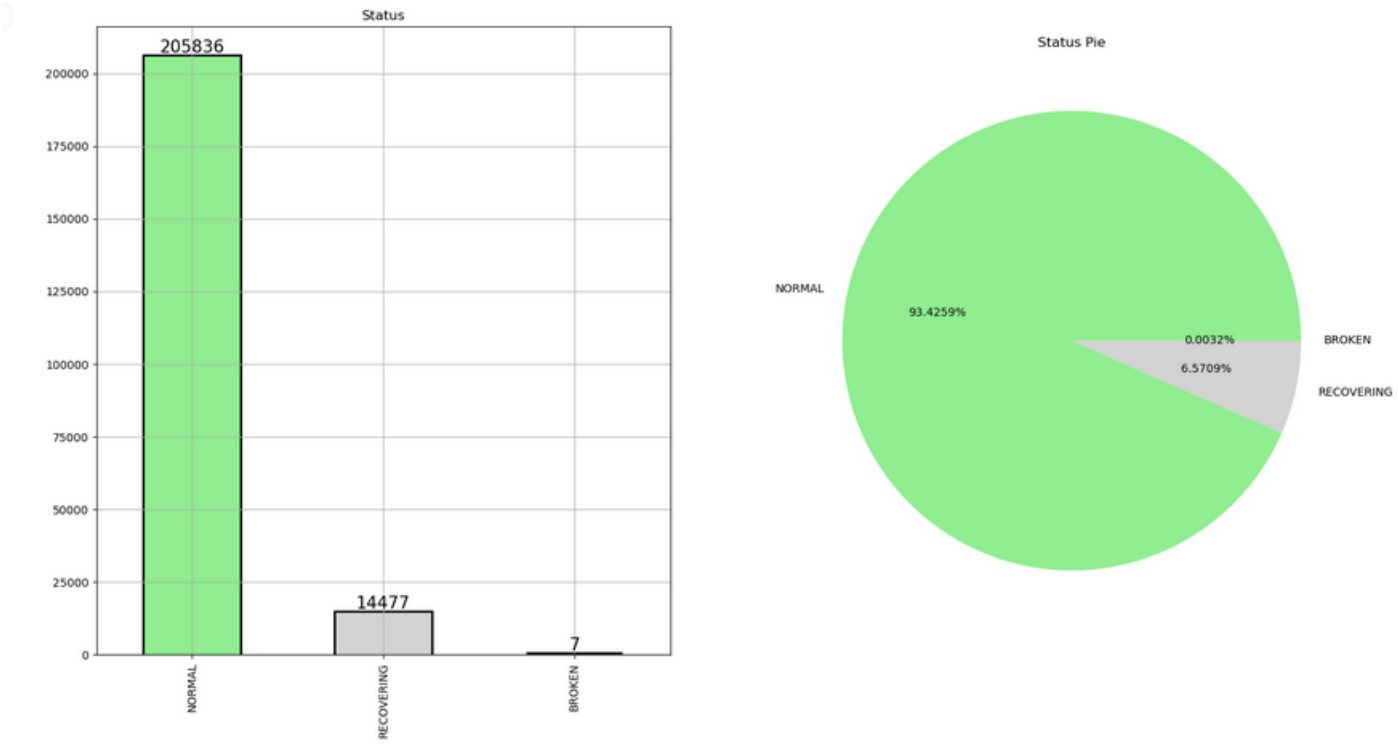


Figure 14: Pump malfunctions and normal functioning.

```
for bar in df.machine_status.value_counts().plot(kind='bar',ax=ax1,color=
    ax1.annotate(format(bar.get_height()),(bar.get_x() + bar.get_width()
                 ha='center',
                 va='center',
                 size=15,
                 xytext=(0, 8),
                 textcoords='offset points'
                )

ax1.grid()
ax1.set_title('Status')
ax2.set_title('Status Pie')
ax2.set_ylabel('')
plt.show()
```

On these graphs 14, we can visualize the level of unsteadiness in our dataset regarding pump malfunctions and normal functioning.
On these charts 15, we see a pattern of the sensors consistently dropping during a malfunction.
We can check outlier in figure 16

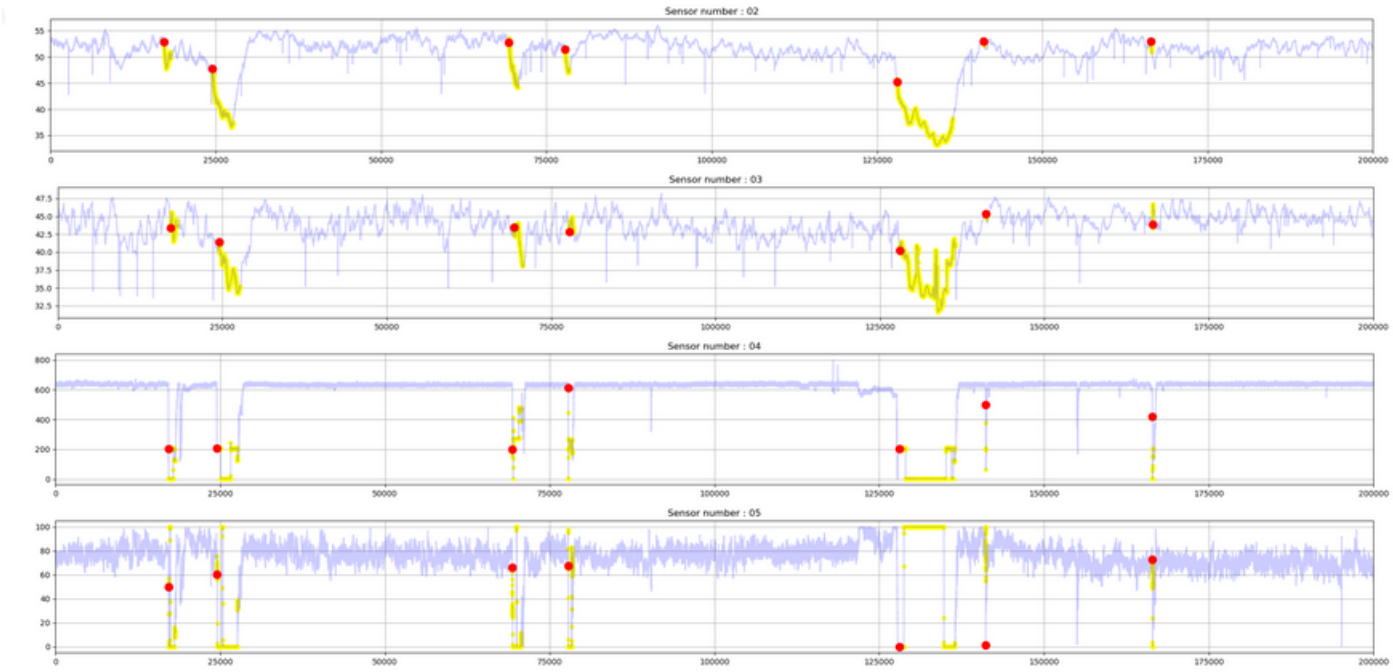


Figure 15: Observing the sensors at the time of failure

4.4 Functions and performance evaluation

4.4.1 Functions

Building the predictive model involves using two models, LSTM and LSTM with MinMax Scaler.

- Code for LSTM model

```
from keras.models import Sequential
from keras.layers import LSTM
from keras.layers import Dropout
from keras.layers import Dense
from sklearn.metrics import mean_squared_error,r2_score
import matplotlib.pyplot as plt
import numpy as np

model = Sequential()
model.add(LSTM(100, input_shape=(train_x.shape[1], train_x.shape[2])))
model.add(Dropout(0.2))
model.add(Dense(1))
model.compile(loss='mean_squared_error', optimizer='adam')
```

Code for LSTM with MinMax Scaler

```
from sklearn.preprocessing import MinMaxScaler
```

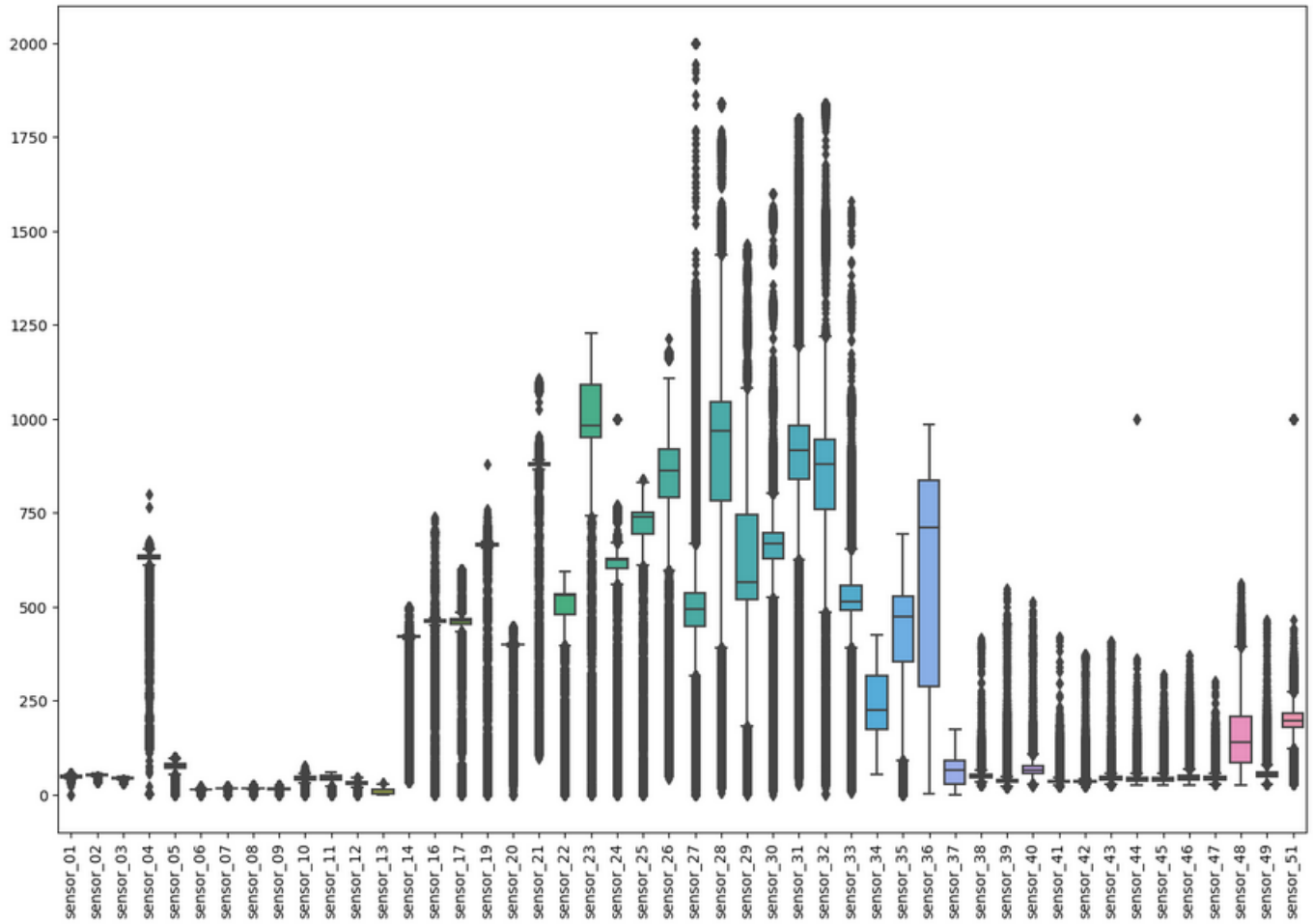


Figure 16: Visualization of Outlier

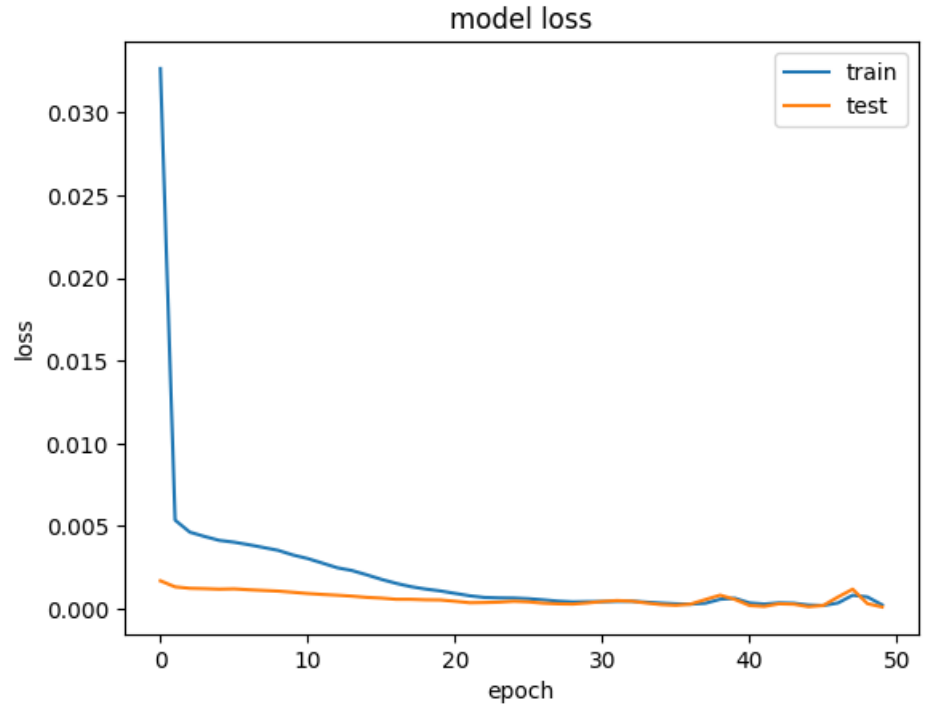
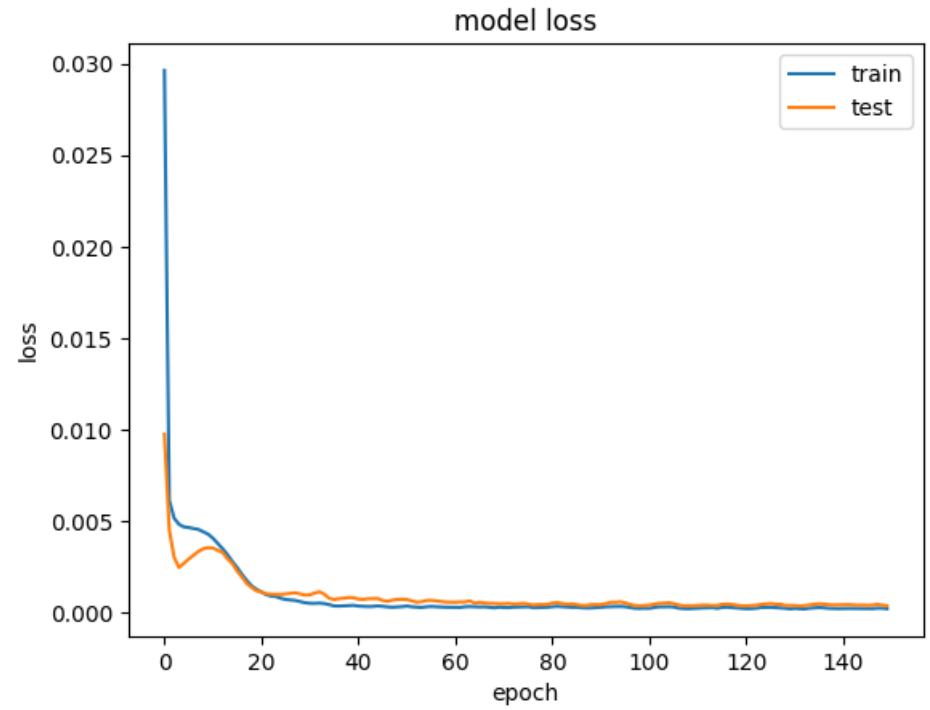


Figure 17: RMSE of LSTM models

```
values = data.values
scaler = MinMaxScaler(feature_range=(0, 1))
scaled = scaler.fit_transform(values)
reframed = series_to_supervised(scaled, 1, 1)
r = list(range(data.shape[1]+1, 2*data.shape[1]))
reframed.drop(reframed.columns[r], axis=1, inplace=True)
reframed.head()
```

4.4.2 Evaluation

Training the two models with the given dataset allows us to compare the RMSE results between the LSTM model and the LSTM with MinMax Scaler. The results show that the LSTM model performs better than the latter with $RMSE = 0.01$ in figure 20.
We can compare the real values with the predicted values of the two models through the graph below.




```
5310/5310 [==============================] - 7s 1ms/step
Test RMSE: 0.020
```

Figure 18: RMSE of LSTM models with Minmax Scaler

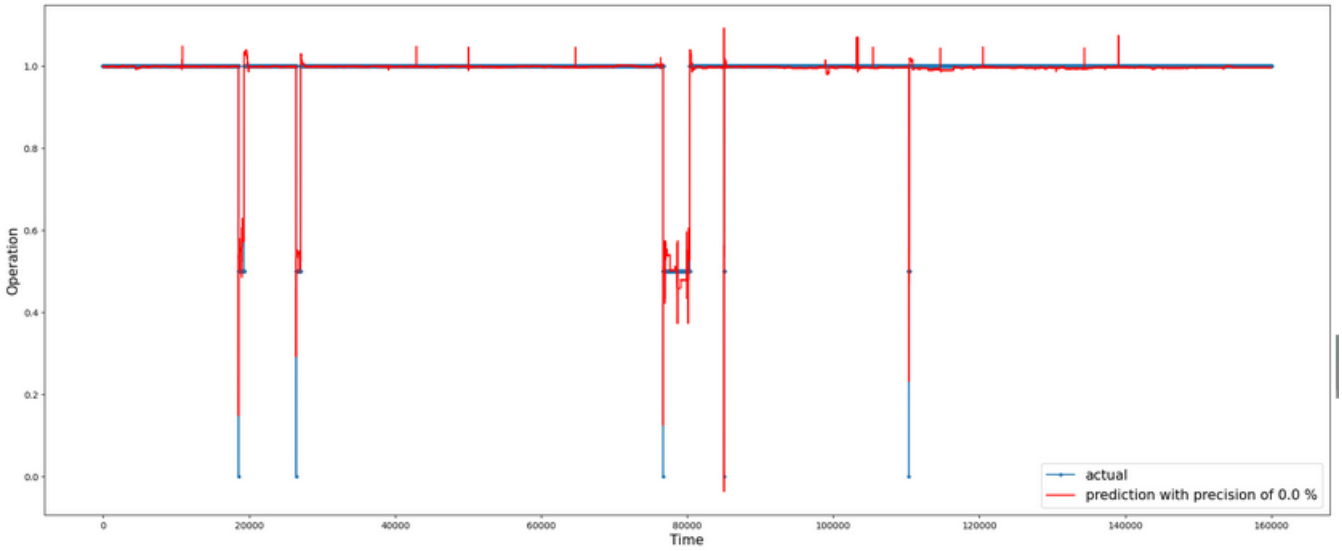


Figure 19: LSTM model

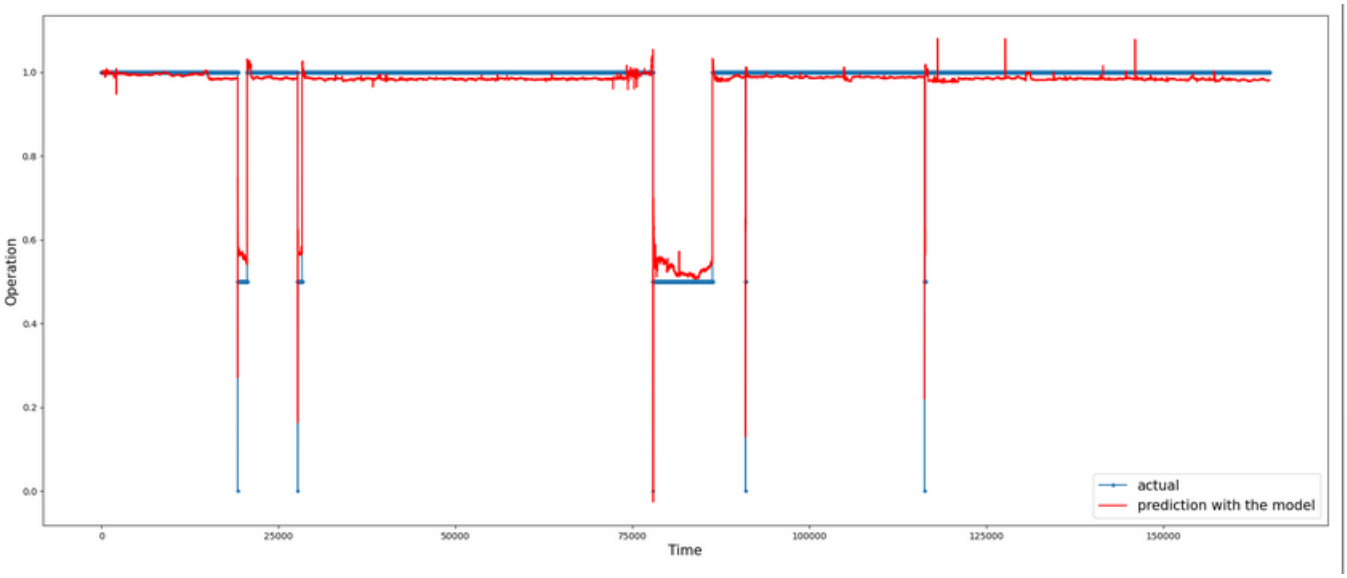


Figure 20: LSTM model with Minmax Scaler

5 Conclusion and future work

- Our primary focus is on scrutinizing the raw data and devising a rational approach. The development of a maintenance prediction plan for pump system operation is not solely reliant on network data but also requires the specification of operational details such as pressure, temperature, flow, vibration, etc., within the pump system. This specific data proves invaluable for analysis, enabling the application of industrial equipment operating knowledge to construct predictive maintenance models through data analysis.

- Additionally, the notebook delves into the analysis and presentation of sensor signals sharing analogous characteristics or reflecting similar operational parameters within the pump system. Based on this analysis, it becomes imperative to cherry-pick a pertinent, straightforward, yet pertinent data series that precisely mirrors the inherent operational characteristics of the pump system. Opting for the correct data not only mitigates computational expenses but also facilitates the creation of a predictive model characterized by high accuracy and reliability.

- The Water Pump Maintenance Break Prediction project uses machine learning and sensor data to predict maintenance needs, improving pump efficiency and reducing downtime. It highlights the importance of predictive maintenance and data-driven decision-making in optimizing industrial operations and ensuring water supply management.

- In conclusion, the project demonstrates a comprehensive approach to handling and analyzing big data generated by IoT-based water pump sensors. The integration of Kafka for data collection, Spark for analytics, and Python for visualization creates a robust and scalable solution. The evaluation of functionality and performance ensures that the system meets the intended requirements and operates efficiently under various conditions. In the future, the use of Tableau for visualization may be researched and applied to large datasets.

References

1. Apache Kafka website: https://kafka.apache.org/
2. Confluent website: https://docs.confluent.io/home/overview.html
3. IBM website: https://www.ibm.com/topics/apache-kafka
4. AWS website: https://aws.amazon.com/what-is/apache-kafka/
5. Pump-sensor-data: https://www.kaggle.com/datasets/nphantawee/pump-sensor-data
6. IBA-Group-IT/IoT-data-simulator: https://github.com/IBA-Group-IT/IoT-data-simulator