Gabriel Zuany Duarte Vargas

Sensors Data Handler

Brazil, ES

2023

1 Introduction

This document presents the *Sensors Data Handler*. The project presented here aims to build a model that, based on data collected from sensors in an industrial equipment, could assess and predict possible failures and abnormalities in the machine in real-time.

In addition to this introduction, this document is organized as follows: Section 2 presents the methods adopted in reading the provided data; Section 3 outlines the main processes in filtering, cleaning, and adapting the values contained in the reading files; Section 4 covers the exploratory analysis stage, where descriptive statistical techniques were used to understand the relationship and context of the data. Section 5 explains the choice of the applied Machine Learning model, provides theoretical and implementation details about it, and also presents the obtained results. Finally, Section 6 illustrates a real situation of streaming data flow with Apache Kafka.

2 Data Reading

This chapter covers the data reading stage and some techniques that assisted in the development in this regard.

2.1 Data Source

The data was provided in CSV (Comma Separated Values) format, where each sensor monitoring the equipment had a file of records.

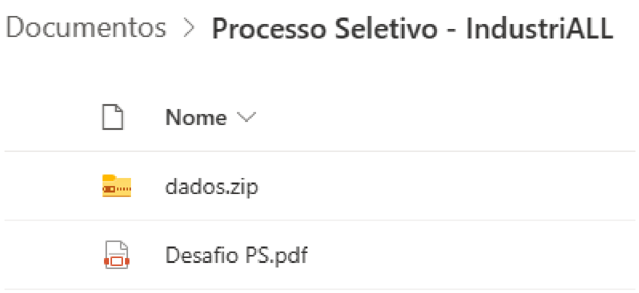

Figura 1 – Data Source

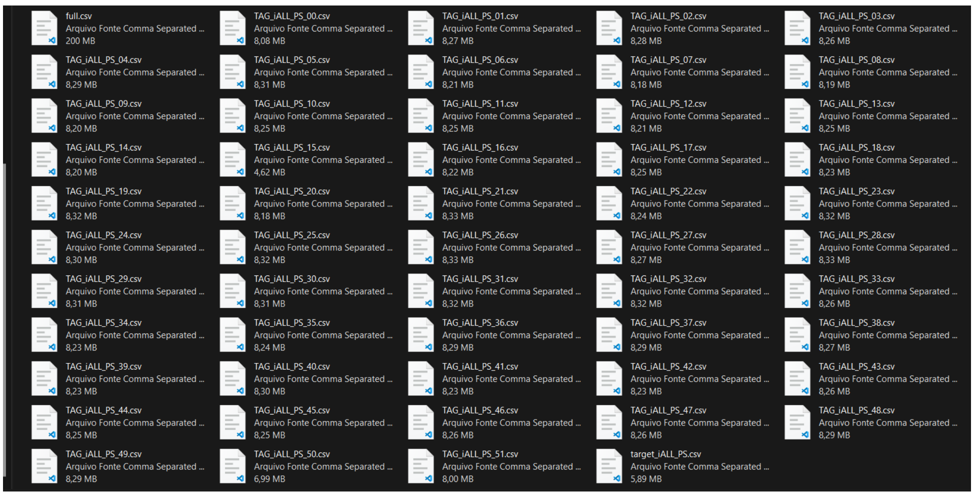

Figura 2 – Files

2.2 Aggregation of Sensor Data

Initially, there were two alternatives for aggregating individual data into a single DataFrame: dynamically aggregate by reading one by one and adding to the general DataFrame, or pre-build this complete dataset and only read it. The first alternative saves storage but, being at runtime, loses overall performance, while the second behaves in the opposite way. Thinking about these two scenarios, a script was created that constructs this complete file if not found.

```python
sensor_summarize_sequential.py > ...
import os
import time
import pandas as pd

DATAFOLDER = 'data/'
BASE_NAME = 'TAG_iALL_PS_'

start_time = time.time()

TAG_iALL_PS_00 = pd.read_csv(DATAFOLDER + 'TAG_iALL_PS_00.csv')
TAG_iALL_PS_00 = TAG_iALL_PS_00.set_index('timestamp')

full_df = pd.DataFrame()
full_df = pd.concat([full_df, TAG_iALL_PS_00], ignore_index=False)
for file in os.listdir(DATAFOLDER):
    if file.endswith('00.csv'):
        continue
    print('Reading file: ' + file)
    df = pd.read_csv(DATAFOLDER + file)
    df = df.set_index('timestamp')
    full_df = pd.concat([full_df, df[file.split('.')[0]]], ignore_index=False, axis=1)
    if file == 'target_iALL_PS.csv':
        full_df = full_df.rename(columns={'target_iALL_PS': 'status'})

full_df.to_csv(DATAFOLDER + 'full.csv')
# print(full_df.head())

end_time = time.time()
print('Elapsed time: ' + str(end_time - start_time) + ' seconds')
```

Figura 3 – Script (Sequential)

```python
src > sensor_summarize_parallel.py > ...
from multiprocessing import cpu_count
import os
import time
import pandas as pd
from concurrent.futures import ProcessPoolExecutor

DATAFOLDER = 'data/'
BASE_NAME = 'TAG_iALL_PS_'

start_time = time.time()

def read(file):
    if file.endswith('00.csv'):
        return None

    print('Reading file: ' + file)
    df = pd.read_csv(DATAFOLDER + file)
    df = df.set_index('timestamp')

    return df[file.split('.')[0]]

if __name__ == '__main__':
    # Get the list of files to process
    files_to_process = [file for file in os.listdir(DATAFOLDER) if not file.endswith('00.csv')]

    # Use ProcessPoolExecutor for parallel processing
    with ProcessPoolExecutor(cpu_count()) as executor:
        # Map the function to process each file in parallel
        dfs = list(executor.map(read, files_to_process))

    # Create the full_df by concatenating the DataFrames
    full_df = pd.concat([pd.read_csv(DATAFOLDER + 'TAG_iALL_PS_00.csv').set_index('timestamp')] + dfs, ignore_index=False, axis=1)

    # Rename the 'target_iALL_PS' column to 'status'
    if 'target_iALL_PS.csv' in files_to_process:
        full_df = full_df.rename(columns={'target_iALL_PS': 'status'})

    # Save the result to a CSV file
    full_df.to_csv(DATAFOLDER + 'full.csv')
    end_time = time.time()
    print('Elapsed time: ' + str(end_time - start_time) + ' seconds')
```

Figura 4 – Slightly refined script (Parallel Processing)

- Generated file layout:

```
timestamp : object | TAG_iALL_PS_00 : float64 | TAG_iALL_PS_01 : float64 | TAG_iALL_PS_02 : float64 | TAG_iALL_PS_03 : float64 |
TAG_iALL_PS_04 : float64 | TAG_iALL_PS_05 : float64 | TAG_iALL_PS_06 : float64 | TAG_iALL_PS_07 : float64 | TAG_iALL_PS_08 : float64 |
TAG_iALL_PS_09 : float64 | TAG_iALL_PS_10 : float64 | TAG_iALL_PS_11 : float64 | TAG_iALL_PS_12 : float64 | TAG_iALL_PS_13 : float64 |
TAG_iALL_PS_14 : float64 | TAG_iALL_PS_15 : float64 | TAG_iALL_PS_16 : float64 | TAG_iALL_PS_17 : float64 | TAG_iALL_PS_18 : float64 |
TAG_iALL_PS_19 : float64 | TAG_iALL_PS_20 : float64 | TAG_iALL_PS_21 : float64 | TAG_iALL_PS_22 : float64 | TAG_iALL_PS_23 : float64 |
TAG_iALL_PS_24 : float64 | TAG_iALL_PS_25 : float64 | TAG_iALL_PS_26 : float64 | TAG_iALL_PS_27 : float64 | TAG_iALL_PS_28 : float64 |
TAG_iALL_PS_29 : float64 | TAG_iALL_PS_30 : float64 | TAG_iALL_PS_31 : float64 | TAG_iALL_PS_32 : float64 | TAG_iALL_PS_33 : float64 |
TAG_iALL_PS_34 : float64 | TAG_iALL_PS_35 : float64 | TAG_iALL_PS_36 : float64 | TAG_iALL_PS_37 : float64 | TAG_iALL_PS_38 : float64 |
TAG_iALL_PS_39 : float64 | TAG_iALL_PS_40 : float64 | TAG_iALL_PS_41 : float64 | TAG_iALL_PS_42 : float64 | TAG_iALL_PS_43 : float64 |
TAG_iALL_PS_44 : float64 | TAG_iALL_PS_45 : float64 | TAG_iALL_PS_46 : float64 | TAG_iALL_PS_47 : float64 | TAG_iALL_PS_48 : float64 |
TAG_iALL_PS_49 : float64 | TAG_iALL_PS_50 : float64 | TAG_iALL_PS_51 : float64 | status : object |
```

Figura 5 – File Layout

3 Data Preprocessing, Filtering, and Cleaning

In this stage, methods such as *head()*, *tail()*, *summarize()*, etc., were used to gain insights into the data. Based on this, duplicate rows were removed to avoid interference in subsequent calculations. Subsequently, a cautious removal of rows with all null values occurred, as a null value for one or more sensors can clearly indicate a failure (provided that the status is not null). There was also a reindexing after removing these values, and the promotion of the *timestamp* column as the index column, transforming our DataFrame into a time series.

Furthermore, the DataFrame was sorted based on the index. Although we know the nature of the data and expect it to be ordered, it is not a guarantee. Reordering provides greater confidence in working with the data in the subsequent steps.

```python
# Drop duplicates and NaNs (when all rows/columns are NaN)
full_df = full_df.drop_duplicates()
full_df = full_df.dropna(axis=1, how='all')
full_df = full_df.dropna(axis=0, how='all')
full_df = full_df.reset_index()

# Convert the timestamp column to datetime and set it as the index
full_df['timestamp'] = pd.to_datetime(full_df['timestamp'])
full_df = full_df.sort_values(by='timestamp')
full_df = full_df.reset_index()
full_df = full_df.set_index('timestamp') if 'timestamp' in full_df.columns else full_df

# Create a column with the status as a boolean (might be useful for plotting later)
full_df['status_bool'] = np.where(full_df['status'] == 'NORMAL', 0, 1)

# drop the columns that are not useful
full_df = full_df.drop(columns=['index'])
full_df = full_df.drop(columns=['level_0'])
```
✓ 1.6s

Figura 6 – Initial Processing

4 Exploratory Analysis

4.1 Initial Checks

Initial checks included examining the proportion of normal and abnormal states of the equipment within the recorded values to understand the predominant behavior.

```
status
NORMAL     205836
ANORMAL     14484
Name: count, dtype: int64
status
NORMAL     0.934259
ANORMAL    0.065741
Name: proportion, dtype: float64
```

Figura 7 – Proportion between Status

4.2 Identification of Outliers

Given the quantity of available data, it was not feasible to analyze (or plot) box plots for each sensor. Therefore, the decision was made to apply an analytical outlier identification method. The analytical method does not always identify all outliers (there may be opposite scenarios); however, in the current case, the results were significantly accurate.

- $IQR = Q3 - Q1$
- $Upperbound = Q3 + 1.5 \times IQR$
- $Lowerbound = Q1 - 1.5 \times IQR$

```
Outliers: 84262
Non normal: 14484
Non normal and outlier: 6363
```

Figura 8 – Outliers

Figura 9 – Boxplot of 'TAG iALL PS 00' (example)

4.3 Correlation Between Variables (Sensors)

This is an essential step for the decision regarding the machine learning model that will be chosen, as we will understand the correlation between the variables we have and the target (status). From the heatmap below, we can see that the correlation between variable data is generally low (only the middle portion, between sensor 16 and 36, has a bit more relationship among themselves). We assume, then, that these are predominantly independent sensors.

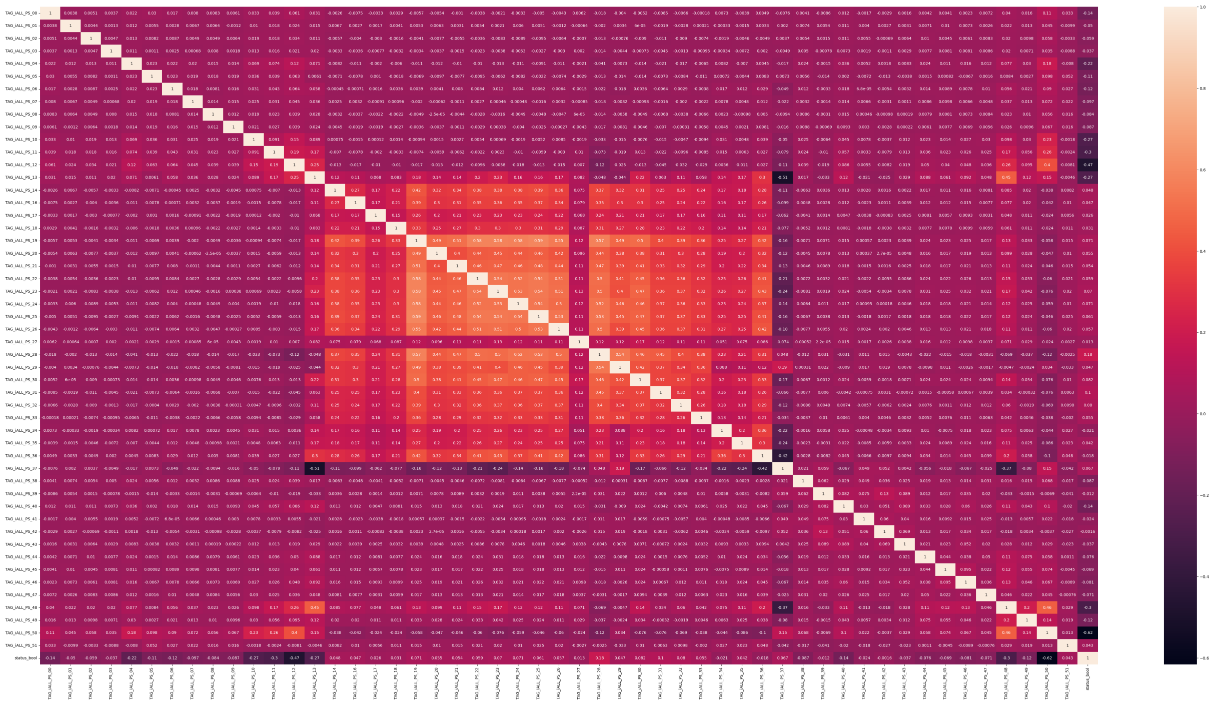

Figura 10 – Correlation between Sensors

Analyzing individually through the heatmap below, we notice that individual sensors have little effect on the equipment status. Therefore, it is a set of sensor states that interfere with the status.

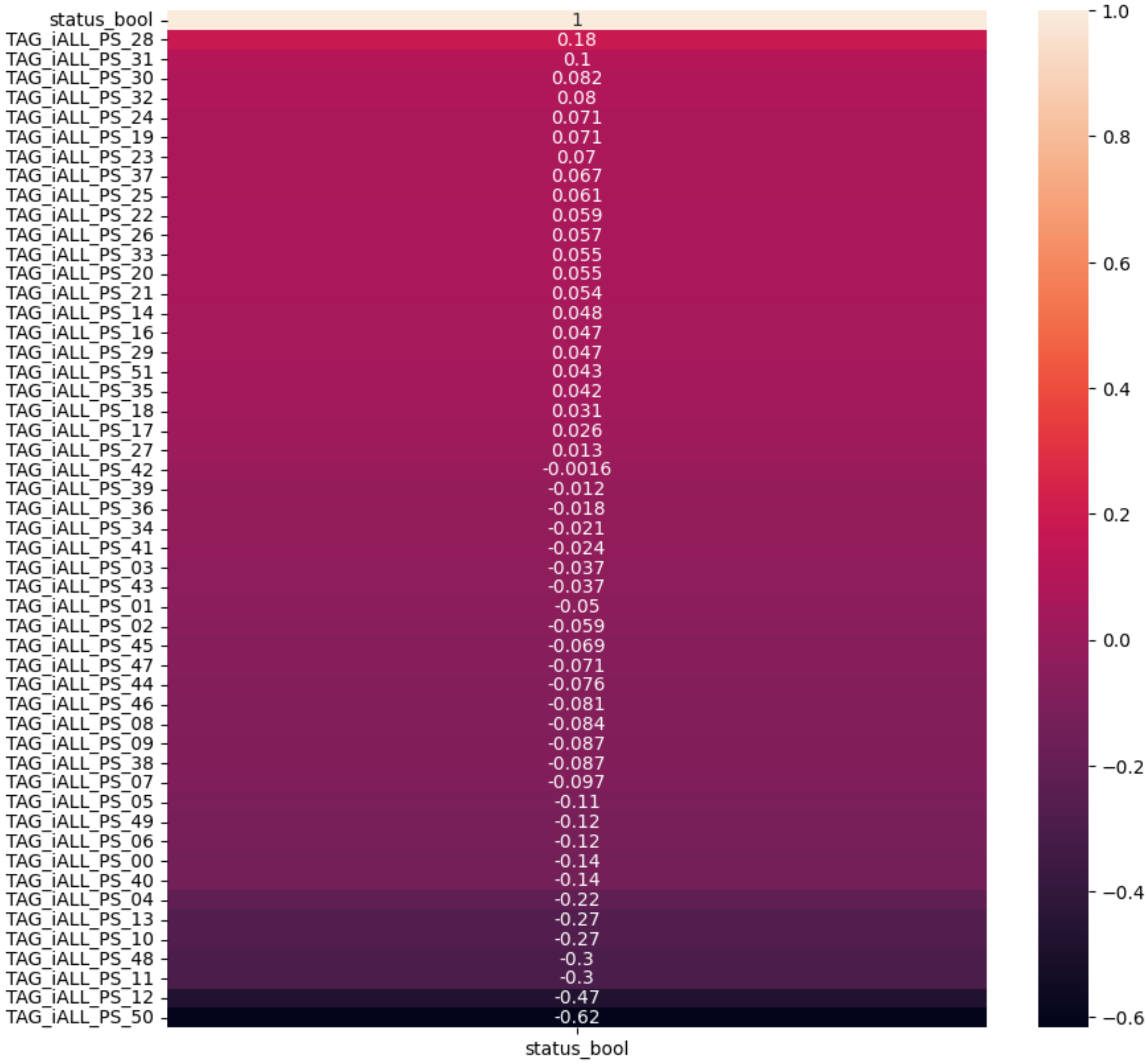

Figura 11 – Correlation between Sensors and Status

With this, we conclude that we have a multivariate analysis with independent features.

5 Machine Learning Model

5.1 Naive Bayes

Given that we have independent variables, the Naive Bayes algorithm is a good choice for our problem.

5.1.1 General Formulation

The conditional probability $P(y|X)$, representing the probability of the class (event) y given the feature vector X, follows Bayes' Theorem:

$$P(y|X) = \frac{P(X|y) \cdot P(y)}{P(X)}$$

5.1.2 Independence

The distinctive feature of Naive Bayes is the assumption of independence between features, given the class. Thus, the formula simplifies to:

$$P(y|X) = \frac{P(x_1|y) \cdot P(x_2|y) \cdot \ldots \cdot P(x_n|y) \cdot P(y)}{P(X)}$$

5.1.3 Class Selection

To determine the most probable class, we further simplify, eliminating $P(X)$ since it does not depend on the class:

$$y = \text{argmax}_y \left(P(x_1|y) \cdot P(x_2|y) \cdot \ldots \cdot P(x_n|y) \cdot P(y)\right)$$

5.1.4 Logarithmic Transformation

Given that probabilities are values between 0 and 1, the expression can be rewritten as a sum of logarithms to facilitate calculations:

$$y = \text{argmax}_y \left(\log(P(x_1|y)) + \log(P(x_2|y)) + \ldots + \log(P(x_n|y)) + \log(P(y))\right)$$

5.1.5 Prior and Class-Conditional with Gaussian Distribution

$P(y)$ represents the a priori probability (probability of an event occurring based on prior knowledge or information), reflecting the frequency of each class. $P(x_i|y)$ represents the class-conditional probability, often modeled with the Gaussian distribution formula:

$$P(x_i|y) = \frac{1}{\sqrt{2\pi\sigma_y^2}} \exp\left(-\frac{(x_i - \mu_y)^2}{2\sigma_y^2}\right)$$

5.1.6 Main Steps

- Training: Calculate the mean, variance, and frequency of each class.
- Predictions: Calculate the future probability for each class using the simplified formula.
- Class Choice: Select the class with the highest a priori probability as the final prediction.

5.2 Model Application

The model implementation was done through the scikit-learn module. It is important to note that there are several different implementations of the Naive Bayes probability distribution function, and in this case, two distinct forms were used:

- Gaussian
- Bernoulli

```python
from sklearn.naive_bayes import GaussianNB, BernoulliNB
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score

# set nan values to 0. The missing values might not affect the model
full_df = full_df.fillna(0)

# create a list of features
features = full_df.columns.tolist()
features.remove('status')
features.remove('status_bool')

# create a list of target
target = 'status'

# split the data into train and test
X_train, X_test, y_train, y_test = train_test_split(full_df[features], full_df[target], test_size=0.2, random_state=42)
```

Figura 12 – Split of Samples

```python
# create a Gaussian and Bernoulli Naive Bayes classifier
gnb = GaussianNB()
bnb = BernoulliNB()

# train the model using the training sets
gnb.fit(X_train, y_train)
bnb.fit(X_train, y_train)

# predict the response for test dataset
y_pred_gnb = gnb.predict(X_test)
y_pred_bnb = bnb.predict(X_test)
```

Figura 13 – Training and Prediction

```
# print the first 10 anormal predictions
def print_result_sample(y_pred, y_test):
    first_10_anormal_predictions = []
    first_10_anormal_actual = []
    for i in range(len(y_pred)):
        if y_pred[i] == 'ANORMAL':
            first_10_anormal_predictions.append(y_pred[i])
            first_10_anormal_actual.append(y_test[i])
            if len(first_10_anormal_predictions) == 10:
                break

    print(first_10_anormal_predictions)
    print(first_10_anormal_actual)

print(f"\nGNB accuracy: {accuracy_score(y_test, y_pred_gnb)}")
print_result_sample(y_pred_gnb, y_test)
print(f"\nBNB accuracy: {accuracy_score(y_test, y_pred_bnb)}")
print_result_sample(y_pred_bnb, y_test)

# check how many predictions match between the two models
print(f"\nNumber of matching predictions: {np.sum(y_pred_gnb == y_pred_bnb)} / {len(y_pred_gnb)}")
```

✓ 1.2s

```
GNB accuracy: 0.9864969135802469
['ANORMAL', 'ANORMAL', 'ANORMAL', 'ANORMAL', 'ANORMAL', 'ANORMAL', 'ANORMAL', 'ANORMAL', 'ANORMAL', 'ANORMAL']
['NORMAL', 'ANORMAL', 'ANORMAL', 'ANORMAL', 'ANORMAL', 'ANORMAL', 'ANORMAL', 'ANORMAL', 'ANORMAL', 'ANORMAL']

BNB accuracy: 0.9746505083514887
['ANORMAL', 'ANORMAL', 'ANORMAL', 'ANORMAL', 'ANORMAL', 'ANORMAL', 'ANORMAL', 'ANORMAL', 'ANORMAL', 'ANORMAL']
['ANORMAL', 'ANORMAL', 'ANORMAL', 'ANORMAL', 'ANORMAL', 'ANORMAL', 'NORMAL', 'ANORMAL', 'ANORMAL', 'ANORMAL']

Number of matching predictions: 42604 / 44064
```

Figura 14 – ML Result

6 Simulating Real Scenario with Apache Kafka

Apache Kafka is an open-source stream processing platform developed by the Apache Software Foundation, written in Scala and Java. The project aims to provide a unified, high-throughput, and low-latency platform for real-time data processing. Its storage layer is a "scalable publisher/subscriber message queue designed as a distributed transaction log,"making it highly valuable for corporate infrastructures processing data streams. With this in mind, the additional idea for this project was to simulate an environment in which sensor data is not in a pre-provided static file but is instead generated and consumed in real-time. Thus, the Naive Bayes model runs each time new data is generated, increasing its accuracy.

6.1 Pipeline Architecture

The main idea is that, at each timestamp (second, in this case), sensor data is received, formatted, and written to the Kafka queue (*producer*). Then, the *consumer* listens to this queue, and for each new entry, it reads, processes, and writes the new records to the database. The database (PostgreSQL) is running in a Docker container. After the data is stored in the database, the script with the Machine Learning model is triggered and starts consuming from the table, generating real-time updated analyses.

6.2 Overview of Kafka

6.2.1 Kafka Topic

Kafka "topics"are mechanisms that store the sequence of received events. These topics can be replicated across different partitions of the broker and even replicated to other nodes (distributed systems) to be processed/consumed in parallel. For this project, only one topic (*SensorDataStream*) with two partitions was created.

```python
from kafka.admin import KafkaAdminClient,NewTopic

TOPIC="SensorsDataStream"

try:
    admin_client = KafkaAdminClient(bootstrap_servers="localhost:9092", client_id='IndustriALL')
except Exception as e:
    print("Exception while connecting Kafka")
    print(str(e))
    exit(1)


if TOPIC in admin_client.list_topics():
    print("Topic already exists")
    exit(0)

try:
    topic_list = []
    new_topic = NewTopic(name=TOPIC, num_partitions= 2, replication_factor=1)
    topic_list.append(new_topic)
    admin_client.create_topics(new_topics=topic_list)
    print("Topic created successfully")
except Exception as e:
    print("Exception while creating topic")
    print(str(e))
    exit(1)
```

Figura 15 – Creating Topic

6.2.2 Enqueuing Data

Sensor data is read from the local file *full.csv* created by the *sensor-summarize.py* referenced in the previous sections. From the DataFrame, it inserts row by row into the queue with a 1-second delay to simulate the real condition of the problem.

```python
from time import sleep
from kafka import KafkaProducer
import json
import pandas as pd

TOPIC = "SensorsDataStream"
INPUT_FILE = "data/full.csv"

df = pd.read_csv(INPUT_FILE)

producer = KafkaProducer(
    bootstrap_servers=['localhost:9092'],
    value_serializer=lambda v: json.dumps(v).encode('utf-8'))

for idx in range(len(df)):
    print(df.iloc[idx].to_dict())
    producer.send(TOPIC, df.iloc[idx].to_dict())
    producer.flush()
    sleep(1)

producer.close()
```

Figura 16 – Producer Script

Terminal

File Edit View Search Terminal Help


```
S_03': 105.48293379659302, 'TAG_iALL_PS_04': 1503.486461674161, 'TAG_iALL_PS_05'
: 103.56996973689056, 'TAG_iALL_PS_06': 6.15986302516886, 'TAG_iALL_PS_07': 21.9
04177644678622, 'TAG_iALL_PS_08': 59.7320062495748, 'TAG_iALL_PS_09': 45.9429485
4836127, 'TAG_iALL_PS_10': 64.33404502805527, 'TAG_iALL_PS_11': 118.349041764501
24, 'TAG_iALL_PS_12': 45.03880008057119, 'TAG_iALL_PS_13': 8.692714802377052, 'T
AG_iALL_PS_14': 613.0778349475679, 'TAG_iALL_PS_15': nan, 'TAG_iALL_PS_16': 991.
6075771141888, 'TAG_iALL_PS_17': 1055.8311680489542, 'TAG_iALL_PS_18': 6.5254563
45849921, 'TAG_iALL_PS_19': 1253.2468332910053, 'TAG_iALL_PS_20': 669.7390292204
593, 'TAG_iALL_PS_21': 1783.7534353500623, 'TAG_iALL_PS_22': 847.2446087787572,
'TAG_iALL_PS_23': 1639.2966319234, 'TAG_iALL_PS_24': 1297.3009680784469, 'TAG_iA
LL_PS_25': 1320.664384482156, 'TAG_iALL_PS_26': 1515.4614805877416, 'TAG_iALL_PS
_27': 1420.7179319120382, 'TAG_iALL_PS_28': 1829.5200883571624, 'TAG_iALL_PS_29'
: 1336.3692894301744, 'TAG_iALL_PS_30': 1413.1754242979794, 'TAG_iALL_PS_31': 14
67.607568439488, 'TAG_iALL_PS_32': 897.4581533064713, 'TAG_iALL_PS_33': 933.0318
156574274, 'TAG_iALL_PS_34': 234.13302911824985, 'TAG_iALL_PS_35': 948.737609614
3008, 'TAG_iALL_PS_36': 720.9681562510398, 'TAG_iALL_PS_37': 156.32603014522303,
 'TAG_iALL_PS_38': 77.57448057722208, 'TAG_iALL_PS_39': 25.453747450381464, 'TAG
_iALL_PS_40': 104.38526679656216, 'TAG_iALL_PS_41': 55.45920457244837, 'TAG_iALL
_PS_42': 79.09089918361585, 'TAG_iALL_PS_43': 89.16191290015365, 'TAG_iALL_PS_44
': 61.45769705526574, 'TAG_iALL_PS_45': 84.80172275039179, 'TAG_iALL_PS_46': 97.
76902151777884, 'TAG_iALL_PS_47': 124.71275231368692, 'TAG_iALL_PS_48': 344.5765
2480342466, 'TAG_iALL_PS_49': 72.50202605297736, 'TAG_iALL_PS_50': 351.836411581
881, 'TAG_iALL_PS_51': 384.8748167580968, 'status': 'NORMAL'}
```


Figura 17 – Producer Terminal

6.2.3 Processing the Queue

While messages are enqueued by the *producer*, our *consumer* processes them one by one and inserts them into the database as it reads, processes, and formats the entries.

```python
from time import sleep
from kafka import KafkaConsumer
import json
import pandas as pd
import psycopg2 as pg

TOPIC="SensorsDataStream"
TABLE_NAME=TOPIC

try:
    conn = pg.connect("dbname='postgres' user='postgres' host='localhost' password='example'")
    print("Connected to database")
    cursor = conn.cursor()
except Exception as e:
    print("Exception while connecting to database")
    print(str(e))
    exit(1)

try:
    consumer = KafkaConsumer(TOPIC,
                             group_id=None,
                              bootstrap_servers=['localhost:9092'],
                              auto_offset_reset = 'earliest')
except Exception as e:
    print("Exception while connecting Kafka")
    print(str(e))
    exit(1)

print("Consumer started")
print(consumer.subscription())
check = True
cursor.execute(f"ROLLBACK;")
```

Figura 18 – Connecting to Queue and Database

```python
for msg in consumer:
    dictionary = json.loads(msg.value.decode('utf-8'))
    df = pd.DataFrame(dictionary, index=[0])
    print(df)

    columns = ",".join(df.columns)

    # create table if not exists
    if check:
        try:
            cursor.execute(f"CREATE TABLE {TABLE_NAME} (id SERIAL PRIMARY KEY);")
            for column in df.columns:
                if column == "timestamp":
                    cursor.execute(f"ALTER TABLE {TABLE_NAME} ADD COLUMN {column} TIMESTAMP;")
                elif column == "status":
                    cursor.execute(f"ALTER TABLE {TABLE_NAME} ADD COLUMN {column} VARCHAR;")
                else:
                    cursor.execute(f"ALTER TABLE {TABLE_NAME} ADD COLUMN {column} FLOAT;")
            conn.commit()
            check = False
        except pg.errors.DuplicateTable:
            print("Table already exists")
            check = False

    # insert data
    values = df.values.tolist()[0]
    values_str = ""
    is_timestamp = True
    for value in values:
        if is_timestamp:
            values_str+=f"'{value}',"
            is_timestamp = False
            continue
        if value == "NORMAL" or value == "ANORMAL":
            values_str+=f"'{value}',"
            break
        values_str+=str(value)+","
    values_str = values_str[:-1]
    values_str = values_str.replace("nan", "NULL")

    cursor.execute(f"INSERT INTO {TABLE_NAME}({columns}) VALUES ({values_str})", values)
    conn.commit()
    sleep(1)

cursor.close()
conn.close()
```

Figura 19 – Processing Queue and Writing to Database

```
Terminal
File  Edit  View  Search  Terminal  Help

[1 rows x 54 columns]
             timestamp  TAG_iALL_PS_00  ...  TAG_iALL_PS_51  status
0  2018-04-01 00:25:00        6.009467  ...      383.953027  NORMAL

[1 rows x 54 columns]
             timestamp  TAG_iALL_PS_00  ...  TAG_iALL_PS_51  status
0  2018-04-01 00:27:00        2.013368  ...      666.257182  NORMAL

[1 rows x 54 columns]
             timestamp  TAG_iALL_PS_00  ...  TAG_iALL_PS_51  status
0  2018-04-01 00:28:00        -0.87427  ...      594.525997  NORMAL

[1 rows x 54 columns]
             timestamp  TAG_iALL_PS_00  ...  TAG_iALL_PS_51  status
0  2018-04-01 00:29:00        3.461133  ...      657.742314  NORMAL

[1 rows x 54 columns]
             timestamp  TAG_iALL_PS_00  ...  TAG_iALL_PS_51  status
0  2018-04-01 00:30:00         6.91672  ...       229.23811  NORMAL

[1 rows x 54 columns]
```

Figura 20 – Consumer Terminal

6.3 Database

The chosen database was PostgreSQL running in Docker. It is a simple implementation since we are only interested in the values emitted by the sensors. If the sensor type and additional information were provided, we would model the database with PK and FK to allow joins and other filters, but it was not relevant in this context.

```yaml
services:

  db:
    image: postgres
    restart: always
    environment:
      POSTGRES_PASSWORD: example
    volumes:
      - pgdata:/var/lib/postgresql/data
    ports:
      - 5432:5432
volumes:
  pgdata:

# user: postgres
# host: localhost
# port: 5432
# pwd: example
# db: postgres
```

Figura 21 – Docker Compose Pgsql

	id	timestamp	tag_iall_ps_00	tag_iall_ps_01	tag_iall_ps_02	tag_iall_ps_03	tag_iall_ps_04	tag_iall_ps_05	tag_iall_ps_06	tag_iall_ps_07	tag_iall_ps_08	tag_iall_ps_09
93	93	2018-04-01 01:31:00.000	3.0252057121	56.1409784023	108.6835224763	93.7372616011	1,295.6620761135	75.2307978293	26.4968999056	36.8189435555	38.356635131	17.5297496588
94	94	2018-04-01 01:33:00.000	5.0208757939	59.5634779848	75.3622583963	32.337486835	842.6515194271	133.6080388679	31.1880917758	30.3699468933	54.7586004925	41.0185634083
95	95	2018-04-01 01:35:00.000	4.2171450275	5.8194111799	83.941072112	85.8390998634	1,136.1624560074	149.7886200006	34.7688630467	31.0466407104	34.2811268165	21.8190974891
96	96	2018-04-01 01:37:00.000	6.5756074188	111.9914626976	99.3230684024	90.5433683004	1,081.5248876355	114.2816341298	45.4758761086	64.202993596	21.9170128389	15.8343607281
97	97	2018-04-01 01:34:00.000	1.8570044679	-3.7822791008	110.4372120137	123.699423227	593.9688608603	287.1981179429	23.794715362	12.5169777844	10.2148268689	46.7861228048
98	98	2018-04-01 01:36:00.000	3.7127314443	88.7734386002	189.3710626858	62.9926672489	1,308.8691608862	121.7108756618	39.2812625895	20.8397765587	28.9433141366	48.1426635792
99	99	2018-04-01 01:38:00.000	6.6147005977	147.0936164358	71.785063745	64.1066135114	470.4902289419	188.9519299712	23.3145484974	52.1147737961	57.9783337823	24.9152014496
100	100	2018-04-01 01:40:00.000	4.2124190349	13.5834036736	153.9285204469	73.3211459819	1,920.6947813092	236.5882190947	14.6727044868	21.6971986104	2.1957581746	26.0715189119
101	101	2018-04-01 01:41:00.000	3.3655628953	193.2138795652	62.8719436781	101.9762806184	1,005.1448700225	160.0694009479	25.1799551763	47.7667440312	55.8749170623	13.7710786048
102	102	2018-04-01 01:39:00.000	3.4136414042	64.2622986908	158.3612928758	85.3192014136	737.4667875573	221.9977359401	16.1982904566	43.959174233	21.9735588056	25.1153993074
103	103	2018-04-01 01:44:00.000	7.7493374016	5.7690958504	109.6741773815	133.9681468607	1,505.0457651357	214.3366297745	27.757524941	34.5394671033	49.2282118435	48.4031898552
104	104	2018-04-01 01:42:00.000	3.8040125738	134.3833520083	115.9327238826	58.6275997857	1,965.8761852588	140.7520771892	21.5337677686	6.304387079	33.1107655562	22.1213457011
105	105	2018-04-01 01:43:00.000	1.4563793568	133.6007544352	-4.8408971776	97.7505631694	589.4179634592	206.1191858926	29.4973336445	27.295086139	22.104965747	15.3603935261
106	106	2018-04-01 01:45:00.000	10.5799661044	28.7474493399	116.4269452797	104.844090305	992.737987047	166.0121437768	28.7953004489	26.546994237	37.3586474941	12.9140564634
107	107	2018-04-01 01:47:00.000	5.749633574	83.660775984	175.4314774732	89.2782653222	835.1093606233	104.4220270101	25.8623482881	46.2879661882	-1.9766939731	10.6437142668
108	108	2018-04-01 01:49:00.000	2.9805601543	95.0489893555	81.0441927511	117.5459286153	1,233.2270969195	152.6047823169	53.0904464355	28.033143728	40.9120684996	32.203230792
109	109	2018-04-01 01:46:00.000	5.3201159463	151.0492128127	64.5035264644	111.7292500626	1,109.0151009819	97.7126889757	-1.8860818136	23.5866380573	25.6853752886	46.3154057225
110	110	2018-04-01 01:48:00.000	4.4527760091	33.0422493976	101.2603931632	141.5026530891	1,380.9196813976	85.0321795125	18.1262909614	11.1773578154	33.3824777835	12.9424837022
111	111	2018-04-01 01:50:00.000	6.3342167527	83.0986427181	111.5888549278	68.8582609026	735.7672975556	257.5089178418	30.3633398142	26.4667444213	33.234675508	10.4917428783
112	112	2018-04-01 01:51:00.000	2.0962776062	43.2378685985	71.5940993764	90.3939990309	668.399602682	45.8761176726	26.2521550652	36.7756144434	22.8039581929	17.5744397277
113	113	2018-04-01 01:52:00.000	2.6316512216	85.0110539187	134.3339987347	62.8700621327	537.4567626277	315.8116091077	16.6816853405	33.2192429516	43.0414136169	64.6720409289
114	114	2018-04-01 01:53:00.000	7.6971711468	161.9832809811	192.032673737	45.9019218172	975.4732114603	176.3266602216	30.7682088356	64.5193469852	20.6545004236	2.1077786872
115	115	2018-04-01 01:57:00.000	3.5729115308	97.643431261	145.7570698519	92.5274723913	303.3207685853	166.5186354761	44.8256158652	30.2802364083	36.9756223356	52.042658735
116	116	2018-04-01 01:54:00.000	4.7523561965	42.6012830298	157.8416391899	73.7814984803	762.4870154217	127.4959736155	37.1627210383	15.0555834934	44.8210268062	16.1733298723
117	117	2018-04-01 01:55:00.000	7.137111335	200.461816649	106.0553666758	113.9834830486	936.4031735155	150.2688786135	37.6308713928	35.3569890867	35.002704258	3.6203688616
118	118	2018-04-01 01:56:00.000	0.7543127442	109.1142326807	181.9147803054	59.4792524317	1,374.1707588742	334.4779628068	39.6377552175	9.0218023927	59.8999696217	29.7766686246
119	119	2018-04-01 01:58:00.000	1.5807190175	205.4908064148	102.7406355166	74.6113154523	2,018.4008932442	148.2853780498	42.5759542555	-5.3725543076	68.830417069	33.382664625
120	120	2018-04-01 02:01:00.000	0.9906876667	184.7040868965	-73.5796245803	37.1424134263	2,447.2107488634	111.4929232037	21.9573759148	22.4185825563	33.1529914799	20.7289141054
121	121	2018-04-01 01:59:00.000	5.3729405933	128.0880048529	120.9134766449	74.7322929886	1,533.4857296684	162.6362846706	39.4944138133	34.0445778521	36.2682946985	17.2068040096
122	122	2018-04-01 02:00:00.000	6.8991116084	-35.0155383147	89.9106978899	35.3203210934	749.5018587391	101.7783249266	28.4119902178	20.9315254397	61.4845849928	28.0949206235
123	123	2018-04-01 02:05:00.000	4.0412581009	80.6814925077	79.5509465399	49.2348630326	17.1504431965	71.8414002669	-5.4831670219	46.9609010379	12.7945076627	42.3761386215
124	124	2018-04-01 02:02:00.000	4.6607391717	49.3096718864	64.854987629	76.5665545861	2,016.3586817412	151.889453991	45.7840468071	68.6556488684	43.744255233	24.4903673666
125	125	2018-04-01 02:03:00.000	4.6740815699	78.0053574713	45.8411482917	89.8380662941	1,329.8873454329	159.5700096528	15.225235257	25.83953246	51.111048612	26.0281331084

Refresh Save Cancel Export data 50000 125 125 row(s) fetched - 28ms (3ms fetch), on 2023-11-29 at 17:54:18

Figura 22 – Records being Written

6.4 Real-time Algorithm

Since values are being generated in real-time, the Naive Bayes model is executed each time new data is inserted into the database, thereby increasing its accuracy.

Note: The images of the algorithm results show accuracy values of 1.0 and only 'NORMAL' status because they represent the initial records. Only around the 170kth line do 'ANORMAL' statuses begin to appear.

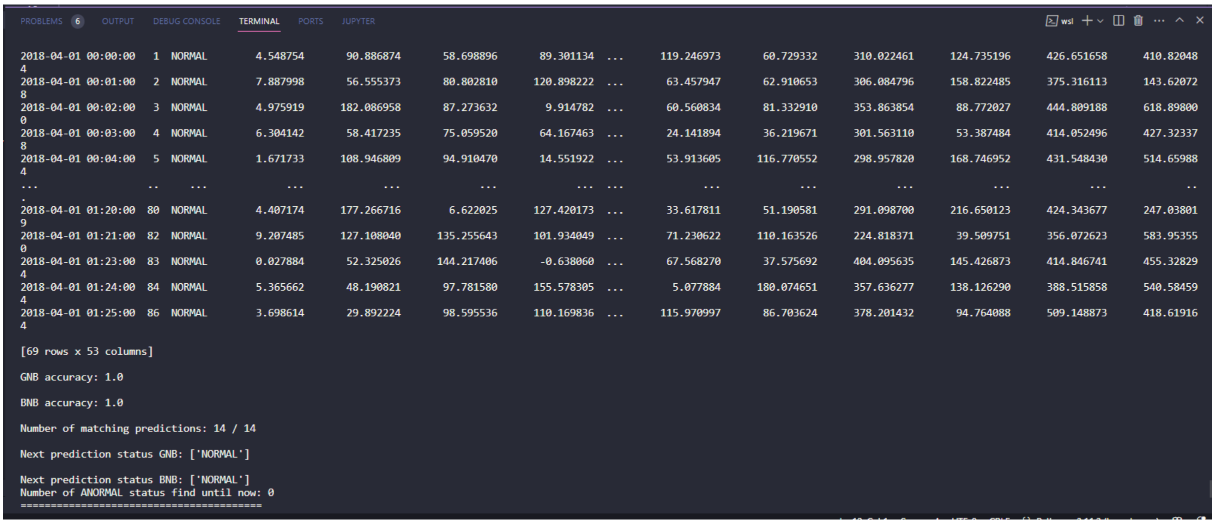

Figura 23 – Result

6.5 Running on Your Machine

All the results above were using Windows 11 with WSL. Only Docker was running on Windows itself; the other services ran in WSL. Enter the project folder and follow the next instructions.

Note: It is necessary to create a 'data' folder containing the full.csv or have, in this folder, the source files to run summirize.py and build full.csv, see Figure ??

6.5.1 Creating the Virtual Environment and Installing Dependencies

- > wsl
- > python3 -m venv venv
- > source venv/bin/activate
- > pip install -r requirements.txt

6.5.2 Installing Dependencies in WSL

- > sudo apt update
- > sudo apt upgrade
- > sudo apt install dos2unix
- > sudo apt install gnome-terminal

6.5.3 Running setup.sh

- > chmod +x kafka/setup.sh
- > chmod +x setup.sh
- > ./setup.sh
- > *(if you need to convert some script, use: > dos2unix <script.sh>)*

  You should see 4 extra terminals for Kafka:

  1. Zookeeper
  2. Server
  3. Producer
  4. Consumer

  And in the same terminal, the execution of Naive Bayes.

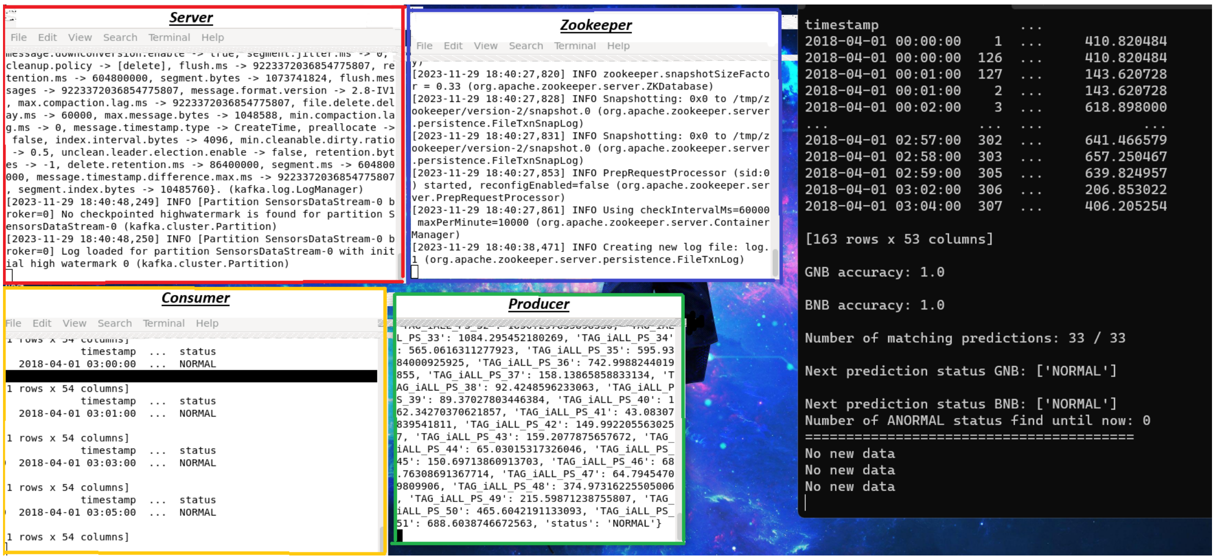

Figura 24 – Terminals