Data Mining - SNCB Project

A.Y. 2023/2024

A work by:

Giorgia Barzan, ID: 000588230

Jibril Gharib, ID: 000588311

Màté Meglécz, ID: 000588471

Maxime Pichet, ID: 000588169

Data loading, exploration and preprocessing

The aim of this project is to delve into the provided dataset, identifying the factors contributing to anomalies, and subsequently constructing models to detect and understand these anomalies.

The dataset contains various variables related to train operations, and by analyzing them, we aim to gain insights into the train operations and identify any anomalies or patterns that may exist.

Through data exploration and visualization techniques, we will uncover valuable information that can help improve train performance and maintenance.

We begin by importing useful libraries.

```
# imports
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
# import geopandas as gpd
# from shapely.geometry import Point
# from geopy.geocoders import Nominatim
import seaborn as sns
import os
```

The first step involves the loading of the data. We also rename the column Unnamed : 0 by ID and transform the column 'timestamps_UTC' into a date time variable, to be able to look more generally at the time period.

```
data = pd.read_csv('./ar41_for_ulb.csv',  sep=';')
data = data.rename(columns={'Unnamed: 0' : 'ID'})
```

```
data.sort_values(by="mapped_veh_id", ascending=True)
```

```
                ID  mapped_veh_id      timestamps_UTC        lat       lon  \
2087711    2087711          102.0 2023-07-31 10:22:30  51.013086  3.780829
8226243    8226243          102.0 2023-02-03 12:43:59  51.015870  3.774773
14651287  14651287          102.0 2023-02-03 12:43:56  51.015658  3.775543
5466267    5466267          102.0 2023-02-03 12:43:46  51.015676  3.775517
5423322    5423322          102.0 2023-02-03 12:42:59  51.015883  3.774768
...            ...            ...                 ...        ...       ...
1107266    1107266          197.0 2023-06-29 09:30:39  50.419863  4.535626
12813319  12813319          197.0 2023-02-02 13:09:41  50.419114  4.533986
12657550  12657550          197.0 2023-02-02 13:09:51  50.418820  4.533492
10669886  10669886          197.0 2023-08-16 23:26:31  50.418918  4.533207
12321342  12321342          197.0 2023-03-13 19:35:23  50.417397  4.529748

          RS_E_InAirTemp_PC1  RS_E_InAirTemp_PC2  RS_E_OilPress_PC1  \
2087711                 26.0                26.0                0.0
8226243                 10.0                 9.5               69.0
14651287                 0.0                19.0                0.0
5466267                 20.0                18.0              224.0
5423322                 20.0                20.0              227.0
...                      ...                 ...                ...
1107266                 52.0                51.0                3.0
12813319                25.0                20.0              224.0
```

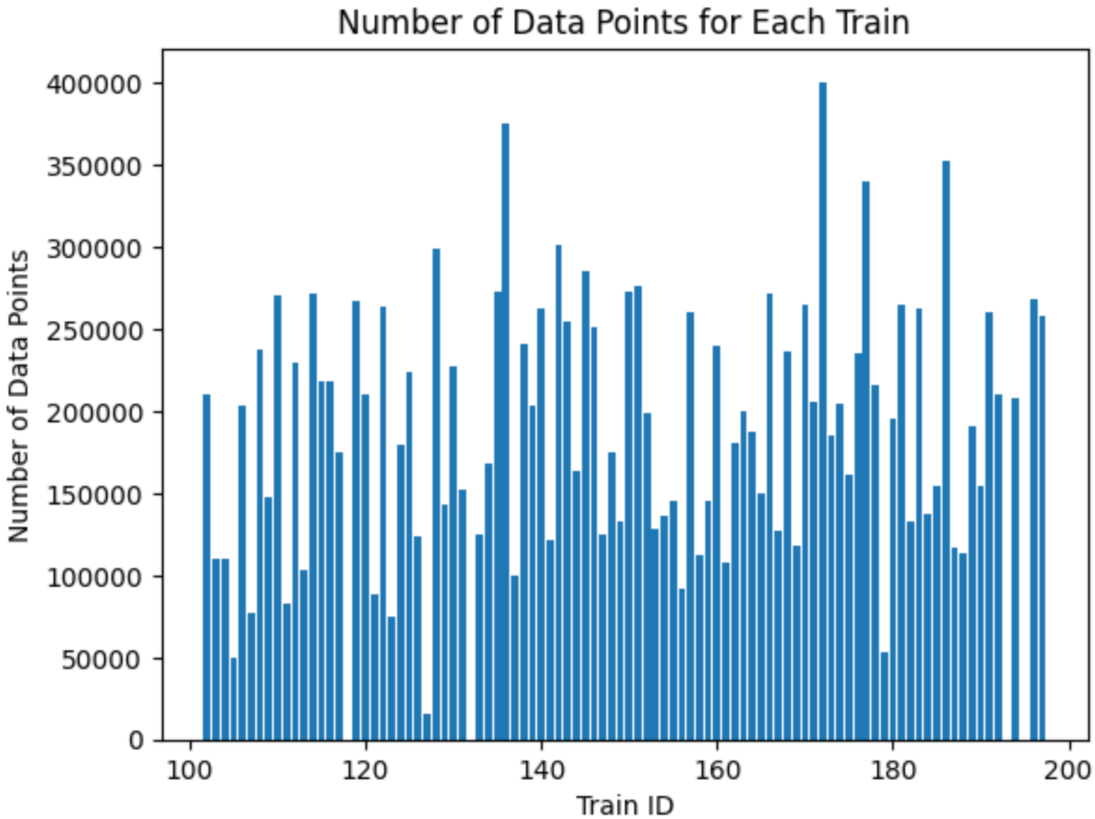
```
12657550                25.0                20.0              224.0
10669886                29.0                30.0              224.0
12321342                30.0                24.0              234.0

          RS_E_OilPress_PC2  RS_E_RPM_PC1  RS_E_RPM_PC2  RS_E_WatTemp_PC1  \
2087711                 0.0           0.0           0.0              27.0
8226243                55.0         164.0         106.5              40.0
14651287              110.0           0.0         147.0              80.0
5466267               231.0         797.0         788.0              80.0
5423322               241.0         797.0         799.0              80.0
...                     ...           ...           ...               ...
1107266                 0.0           0.0           0.0              81.0
12813319              372.0         795.0         797.0              77.0
12657550              379.0         799.0         804.0              77.0
10669886              369.0         802.0         794.0              78.0
12321342              345.0         870.0         880.0              78.0

          RS_E_WatTemp_PC2  RS_T_OilTemp_PC1  RS_T_OilTemp_PC2  month
2087711               31.0              18.0              22.0      7
8226243               39.5              75.5              76.5      2
14651287              79.0              75.0              77.0      2
5466267               79.0              74.0              77.0      2
5423322               78.0              76.0              77.0      2
...                    ...               ...               ...    ...
1107266               81.0              79.0              82.0      6
12813319              50.0              73.0              50.0      2
12657550              50.0              71.0              50.0      2
10669886              56.0              70.0              52.0      8
12321342              67.0              77.0              70.0      3

[17679273 rows x 16 columns]
```

```python
data['timestamps_UTC'] = pd.to_datetime(data['timestamps_UTC'])
data = data.sort_values(by='timestamps_UTC')
data['month'] = data['timestamps_UTC'].dt.month
```

Data exploration

With the following plot we have a visual representation of the distribution of the observations in our trains.

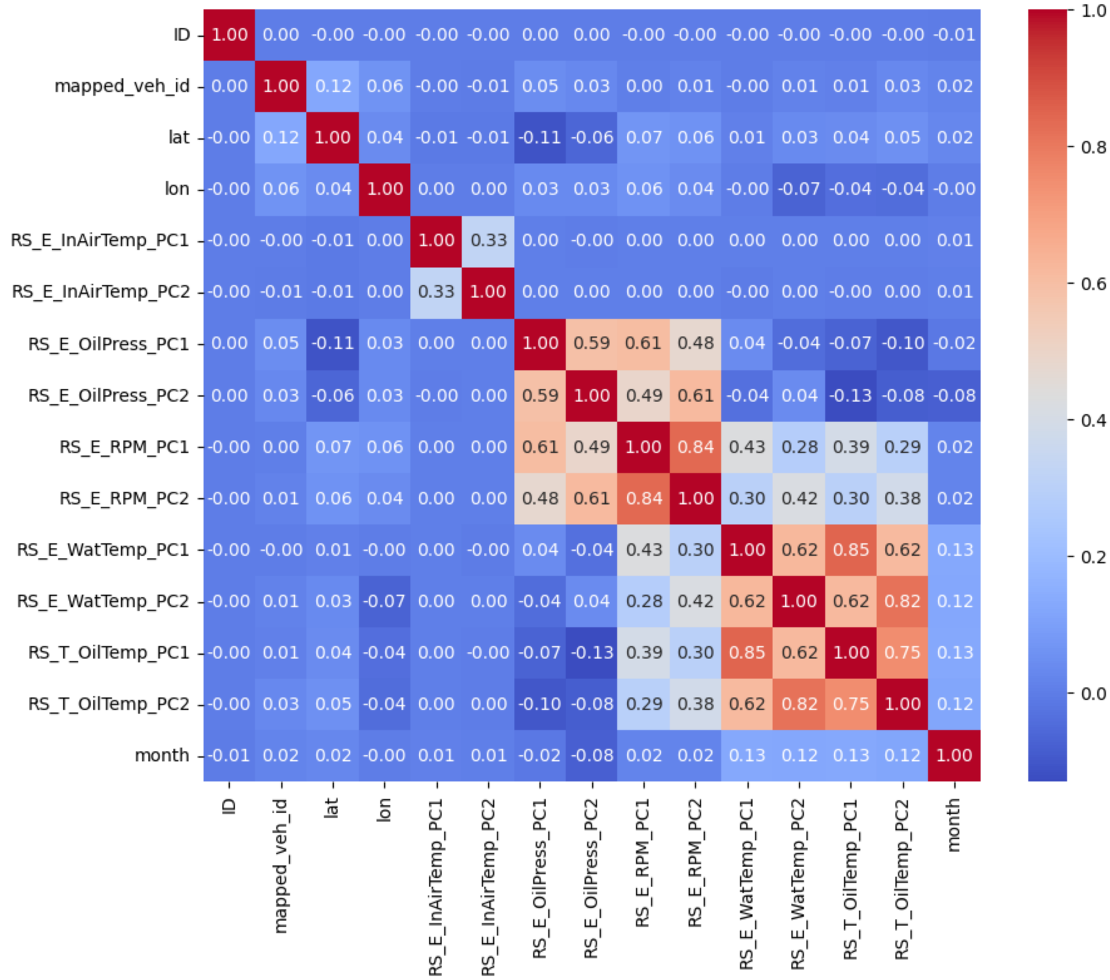
```python
train_counts = data['mapped_veh_id'].value_counts()
plt.bar(train_counts.index, train_counts.values)
plt.xlabel('Train ID')
plt.ylabel('Number of Data Points')
plt.title('Number of Data Points for Each Train')

plt.show()
```



Correlation Matrix

A crucial aspect of data exploration involves analyzing the correlation matrix, which illustrates the relationships between different variables. We achieve this by initially generating the correlation matrix and subsequently visualizing it through plotting.

```
unstamped_data = data.drop(columns=('timestamps_UTC'))

correlation_matrix = unstamped_data.corr()
plt.figure(figsize=(10, 8))
sns.heatmap(correlation_matrix, annot=True, cmap='coolwarm', fmt=".2f")
plt.show()
```



When examining the correlation matrix without timestamps, noticeable correlations emerge among certain components, such as water temperature and oil temperature. This connection is logical: a high water temperature may limit the engine and oil's ability to extract more energy, potentially leading to elevated oil pressure and, consequently, explaining anomalies. While these connections are hypothetical, the correlation matrix introduces intriguing theories.

```
# Find correlated columns
correlated_columns = find_correlated_columns(data)

# Display the results
print(f'Correlated Columns: {correlated_columns}')
```

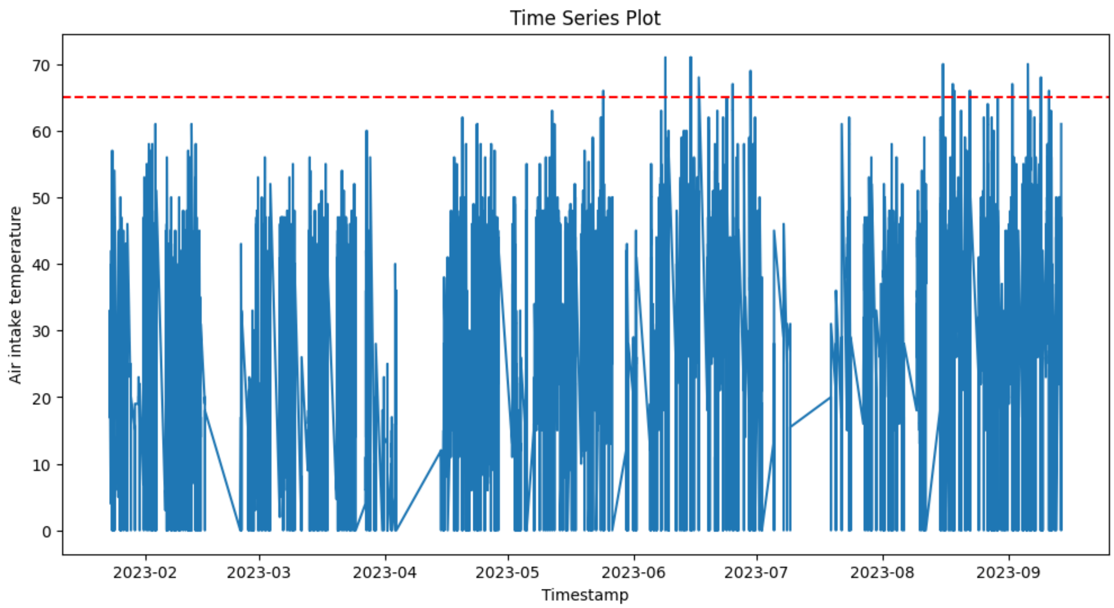
```
Correlated Columns: {'month', 'RS_T_OilTemp_PC1', 'RS_T_OilTemp_PC2', 'RS_E_RPM_PC
2'}
```

Threshold analysis

Beginning the threshold analysis, we will focus on train 102 as an illustrative example. Our objective is to quantify the instances where values surpass the maximum threshold for intake temperature. Additionally, we will explore the threshold for oil and water temperatures.

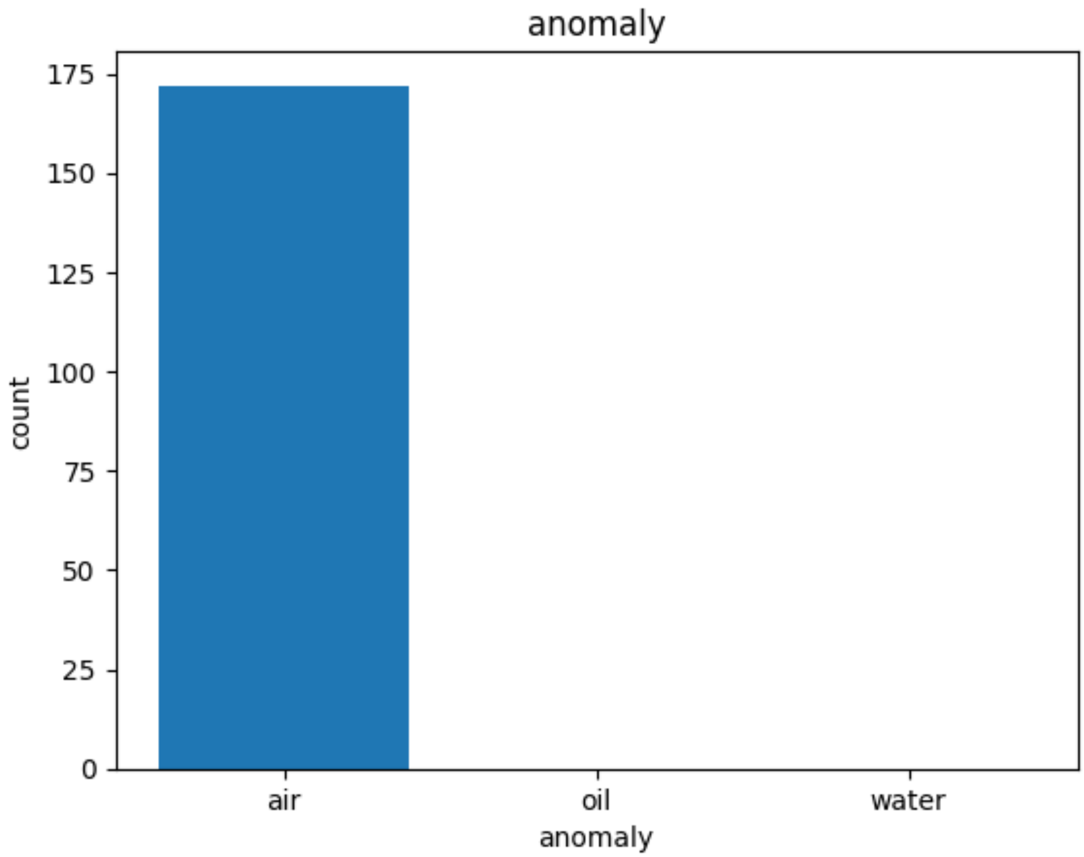
```
train_102_data = data[data['mapped_veh_id']==102]
plt.figure(figsize=(12, 6))
plt.plot(train_102_data['timestamps_UTC'], train_102_data['RS_E_InAirTemp_PC1'])
plt.axhline(y=65, color='r', linestyle='--', label=f'Anormaly treshold')
plt.xlabel('Timestamp')
```

```
plt.ylabel('Air intake temperature')
plt.title('Time Series Plot')

plt.show()
```



With an understanding of the thresholds for oil temperature and water temperature, we delve into examining the behavior of this particular train concerning these variables. Notably, our observation reveals anomalies exclusively within the in-air temperature variable.

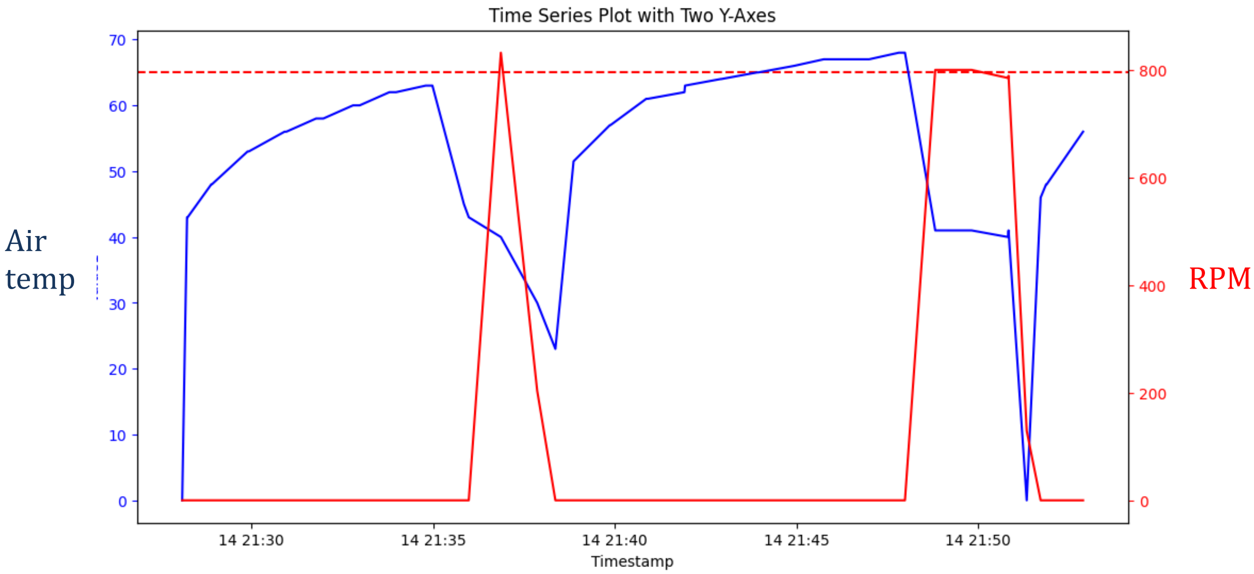
```
train_102_pc1_inair_anomalies = train_102_data[train_102_data['RS_E_InAirTemp_PC1'
]>65]
train_102_pc1_oil_anomalies = train_102_data[train_102_data['RS_T_OilTemp_PC1']>11
5]
train_102_pc1_water_anomalies = train_102_data[train_102_data['RS_E_WatTemp_PC1']>
100]

df = {'anomaly':['air','oil','water'],'count':[len(train_102_pc1_inair_anomalies),
len(train_102_pc1_oil_anomalies),len(train_102_pc1_water_anomalies)]}
df = pd.DataFrame(df)
plt.figure()
plt.bar(df['anomaly'],df['count'])
plt.xlabel('anomaly')
plt.ylabel('count')
plt.title('anomaly')

plt.show()
```



With a timeseries plot we examine the behaviour of 'RS_E_InAirTemp_PC1' and 'RS_E_RPM_PC2' throughout time, and notice that as one increases the other tends to decrease.

```
position = train_102_data.loc[train_102_data.index == indices_over_65[0]].index.to
list()

# print(train_102_data.get(position))
start = indices_over_65[0]
end = indices_over_65[0]
reduced = train_102_data.iloc[135600:135650]
fig, ax1 = plt.subplots(figsize=(12, 6))

plt.axhline(y=65, color='r', linestyle='--', label=f'Anormaly treshold')

ax1.plot(reduced['timestamps_UTC'], reduced['RS_E_InAirTemp_PC1'], color='blue', l
abel='Air temp')
ax1.set_xlabel('Timestamp')
ax1.set_ylabel('Value1', color='blue')
ax1.tick_params('y', colors='blue')


ax2 = ax1.twinx()


ax2.plot(reduced['timestamps_UTC'], reduced['RS_E_RPM_PC2'], color='red', label='R
PM')
ax2.set_ylabel('Value2', color='red')
ax2.tick_params('y', colors='red')

plt.title('Time Series Plot with Two Y-Axes')
```

```
plt.show()
```



We will now delve into the basic statistics of our data to extract valuable insights.

```
#Data exploration


#Min Max for each column
mapped_vehicle_id = data['mapped_veh_id']
print('mapped_veh_id : Min ', mapped_vehicle_id.min(), ' Max ', mapped_vehicle_id.
max())

timestamp = data['timestamps_UTC']
print('TimeStamps_UTC : Min ', timestamp.min(), ' Max ', timestamp.max())


lat = data['lat']
print('Lat : Min ', lat.min(), ' Max ', lat.max())

lon = data['lon']
print('lon : Min ', lon.min(), ' Max ', lon.max())


numeric_columns = data.select_dtypes(include='number').columns.difference(['mapped
_veh_id', 'timestamps_UTC','lon','lat','ID'])

pd.set_option('display.float_format', '{:.2f}'.format)

basic_statistics = data[numeric_columns].describe()
print("Basic Statistics:")
print(basic_statistics)

print(data.info())

print("Is null :")
print(data.isnull().sum())
```

```
mapped_veh_id : Min  102.0  Max  197.0
TimeStamps_UTC : Min  2022-08-22 14:31:20  Max  2023-09-13 21:52:58
Lat : Min  48.2956767  Max  52.8570588
lon : Min  0.1750491  Max  8.0454919
Basic Statistics:
       RS_E_InAirTemp_PC1  RS_E_InAirTemp_PC2  RS_E_OilPress_PC1  \
count         17679273.00         17666547.00        17679273.00
mean                32.02               32.33             263.61
std                328.00              348.00             115.24
min                  0.00                0.00               0.00
25%                 22.00               22.00             203.00
50%                 32.00               33.00             238.00
75%                 40.00               39.00             320.00
max              65535.00            65535.00             690.00

       RS_E_OilPress_PC2  RS_E_RPM_PC1  RS_E_RPM_PC2  RS_E_WatTemp_PC1  \
count        17666547.00   17679273.00   17666547.00       17679273.00
mean              270.69        912.25        907.96             76.93
std               116.12        383.31        388.47             13.65
min                 0.00          0.00          0.00            -15.00
25%               210.00        797.00        797.00             77.00
50%               248.00        801.00        801.00             81.00
75%               331.00        812.00        811.00             84.00
max               690.00       2309.00       9732.00            109.00

       RS_E_WatTemp_PC2  RS_T_OilTemp_PC1  RS_T_OilTemp_PC2       month
count       17666547.00       17679273.00       17666547.00 17679273.00
mean              76.14             76.55             76.16        4.89
std               14.53             14.50             15.35        2.28
min              -17.00           -128.00              0.00        1.00
25%               76.00             74.00             74.00        3.00
50%               81.00             81.00             81.00        5.00
75%               84.00             85.00             85.00        7.00
max              119.00            127.00            117.00       12.00
<class 'pandas.core.frame.DataFrame'>
Index: 17679273 entries, 11759521 to 10041108
Data columns (total 16 columns):
 #   Column              Dtype
---  ------              -----
 0   ID                  int64
 1   mapped_veh_id       float64
 2   timestamps_UTC      datetime64[ns]
 3   lat                 float64
 4   lon                 float64
 5   RS_E_InAirTemp_PC1  float64
 6   RS_E_InAirTemp_PC2  float64
 7   RS_E_OilPress_PC1   float64
 8   RS_E_OilPress_PC2   float64
 9   RS_E_RPM_PC1        float64
 10  RS_E_RPM_PC2        float64
 11  RS_E_WatTemp_PC1    float64
 12  RS_E_WatTemp_PC2    float64
 13  RS_T_OilTemp_PC1    float64
 14  RS_T_OilTemp_PC2    float64
 15  month               int32
dtypes: datetime64[ns](1), float64(13), int32(1), int64(1)
```

```
memory usage: 2.2 GB
None
Is null :
ID                       0
mapped_veh_id            0
timestamps_UTC           0
lat                      0
lon                      0
RS_E_InAirTemp_PC1       0
RS_E_InAirTemp_PC2   12726
RS_E_OilPress_PC1        0
RS_E_OilPress_PC2    12726
RS_E_RPM_PC1             0
RS_E_RPM_PC2         12726
RS_E_WatTemp_PC1         0
RS_E_WatTemp_PC2     12726
RS_T_OilTemp_PC1         0
RS_T_OilTemp_PC2     12726
month                    0
dtype: int64
```

As evident from the data, there are indications of sensor errors. The maximum value for a 16-bit unsigned integer is 65535, suggesting that the recorded maximum value for air intake may be a result of sensor failure, reflecting the highest possible 16-bit integer value. This serves as our initial anomaly detection observation.

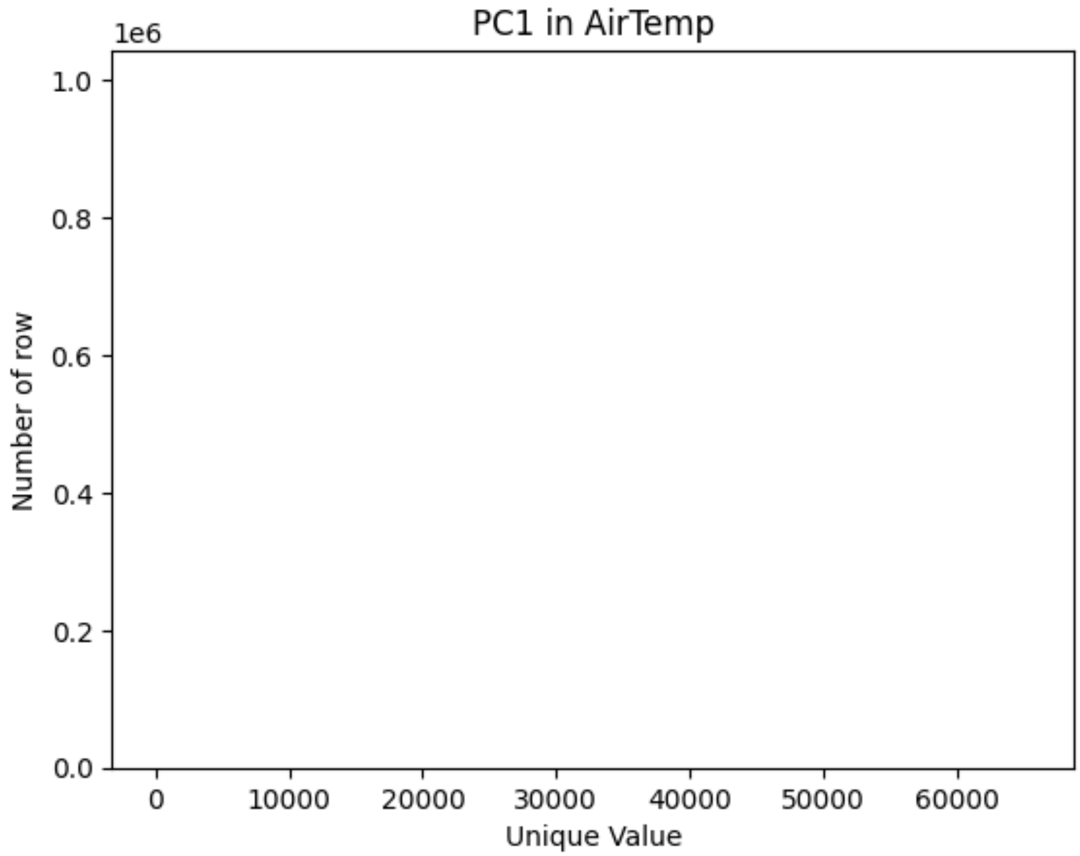
To further identify equipment failures, we plan to implement basic checks to assess the proper functioning of the equipment. While this detailed analysis will be conducted in subsequent stages, we can explore potential equipment malfunctions. For instance, if the train is in motion, the oil pressure, water pressure, and their corresponding temperatures should not be at their minimum or maximum values. An oil temperature reading of -128, for example, lacks practical significance.

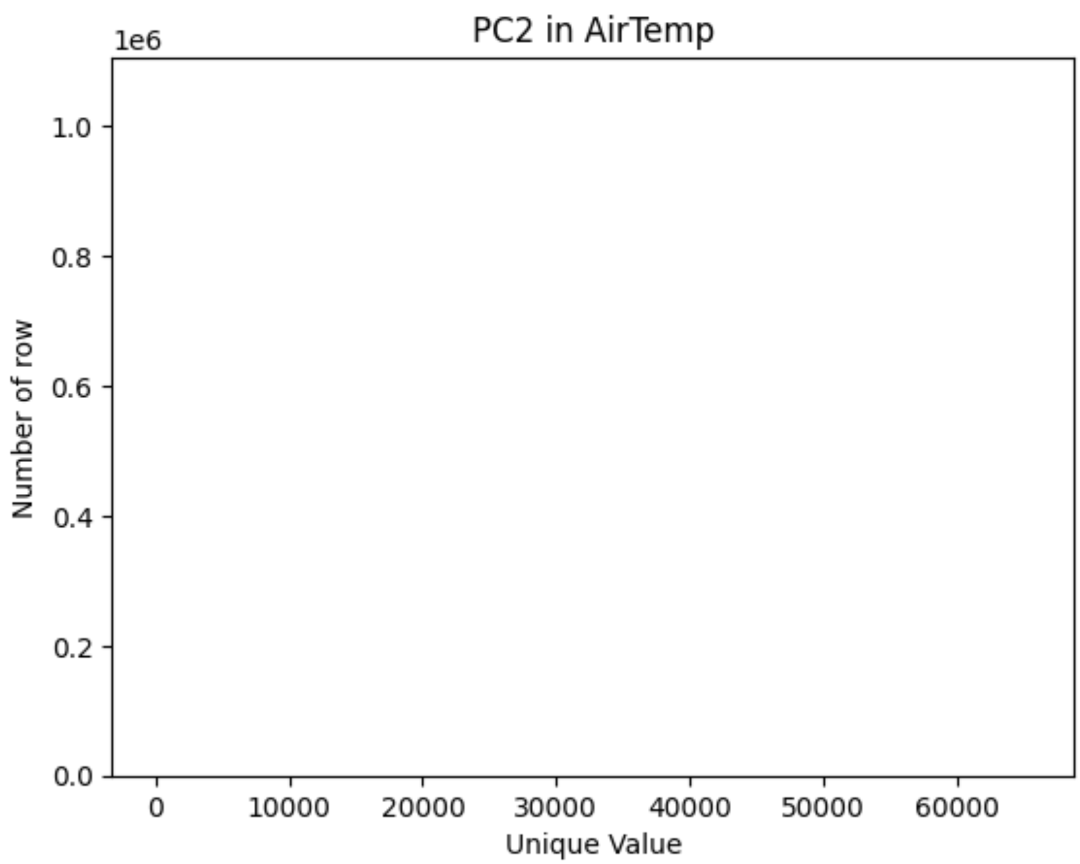
Additionally, the dataset reveals the presence of numerous NaN values and dates extending back to 2022, which are undesirable for our analysis. It's important to note that our analysis spans from January 2023 to September 2023. Further refinement of the data will be undertaken as we progress in our exploration.

WE are now going to visually inversigate the distribution of the variables and observe the frequency of specific values.

```
uniqueValuePC1 = data['RS_E_InAirTemp_PC1'].value_counts()
uniqueValuePC2 = data['RS_E_InAirTemp_PC2'].value_counts()

plt.bar(uniqueValuePC1.index, uniqueValuePC1.values)
plt.xlabel('Unique Value')
plt.ylabel('Number of row')
plt.title('PC1 in AirTemp')
plt.show()

plt.bar(uniqueValuePC2.index, uniqueValuePC2.values)
plt.xlabel('Unique Value')
plt.ylabel('Number of row')
plt.title('PC2 in AirTemp')
plt.show()
```

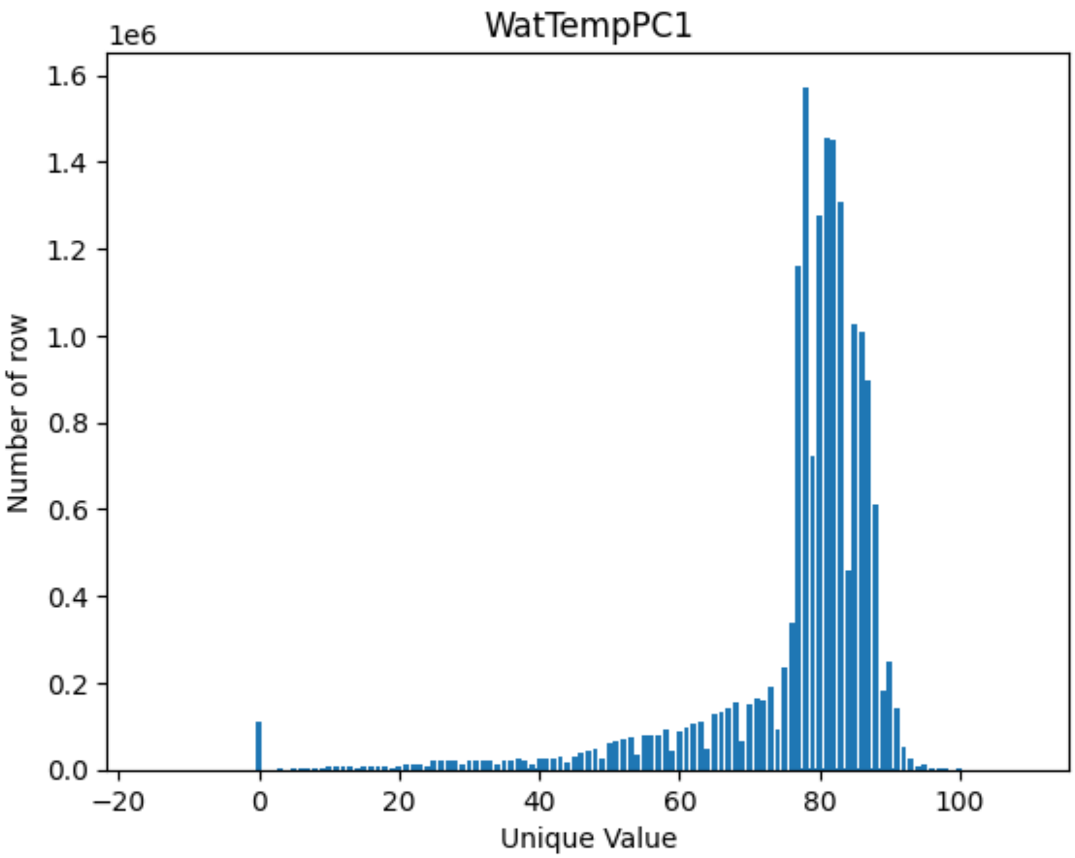
```
print(data['RS_E_InAirTemp_PC1'].value_counts()[0])
print(data['RS_E_InAirTemp_PC2'].value_counts()[0])
```

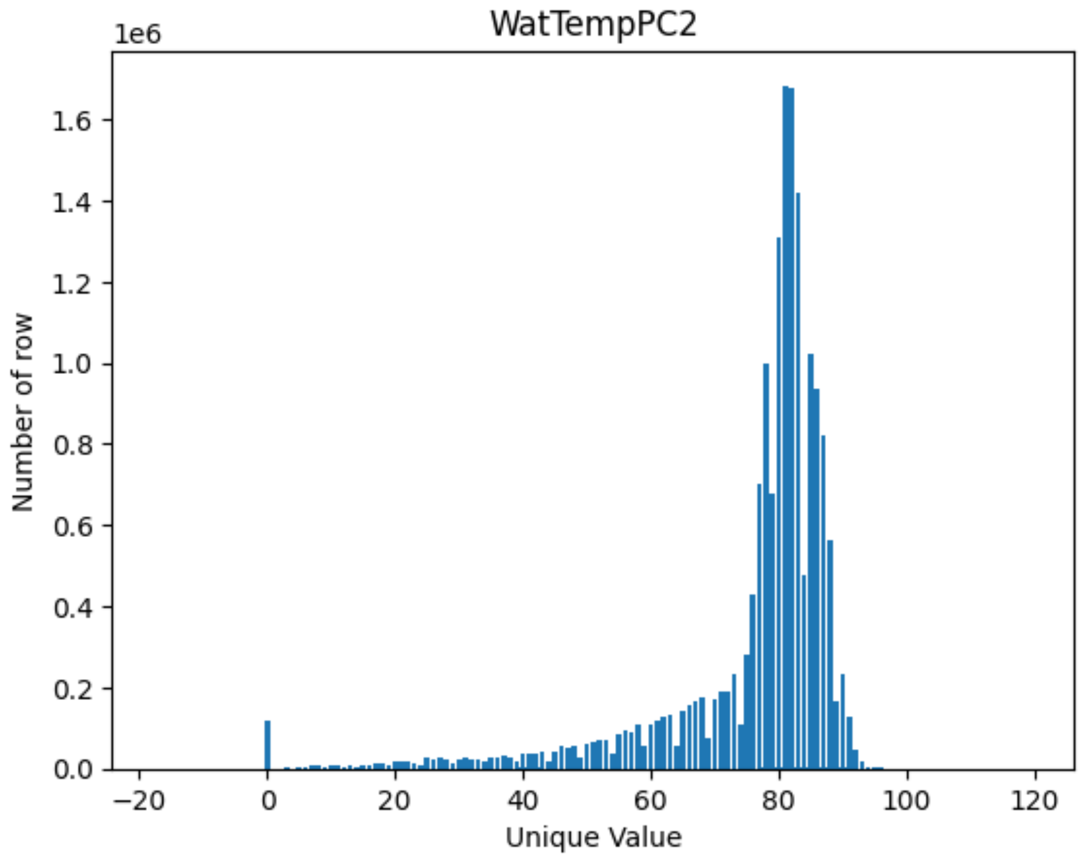




```
114869
122668
```

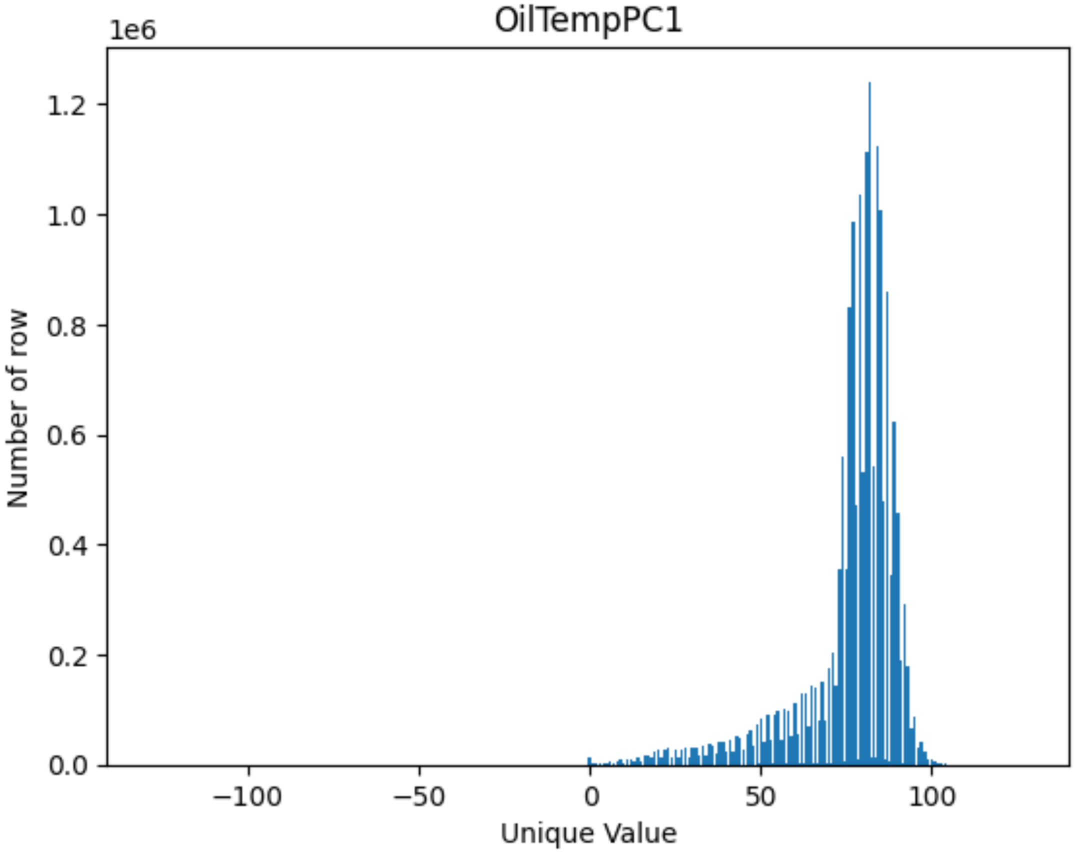
```
WatTempPC1 = data['RS_E_WatTemp_PC1'].value_counts()
WatTempPC2 = data['RS_E_WatTemp_PC2'].value_counts()

plt.bar(WatTempPC1.index, WatTempPC1.values)
plt.xlabel('Unique Value')
plt.ylabel('Number of row')
plt.title('WatTempPC1')
plt.show()

plt.bar(WatTempPC2.index, WatTempPC2.values)
plt.xlabel('Unique Value')
plt.ylabel('Number of row')
plt.title('WatTempPC2')
plt.show()

print(data['RS_E_WatTemp_PC1'].value_counts()[0])
print(data['RS_E_WatTemp_PC2'].value_counts()[0])
```



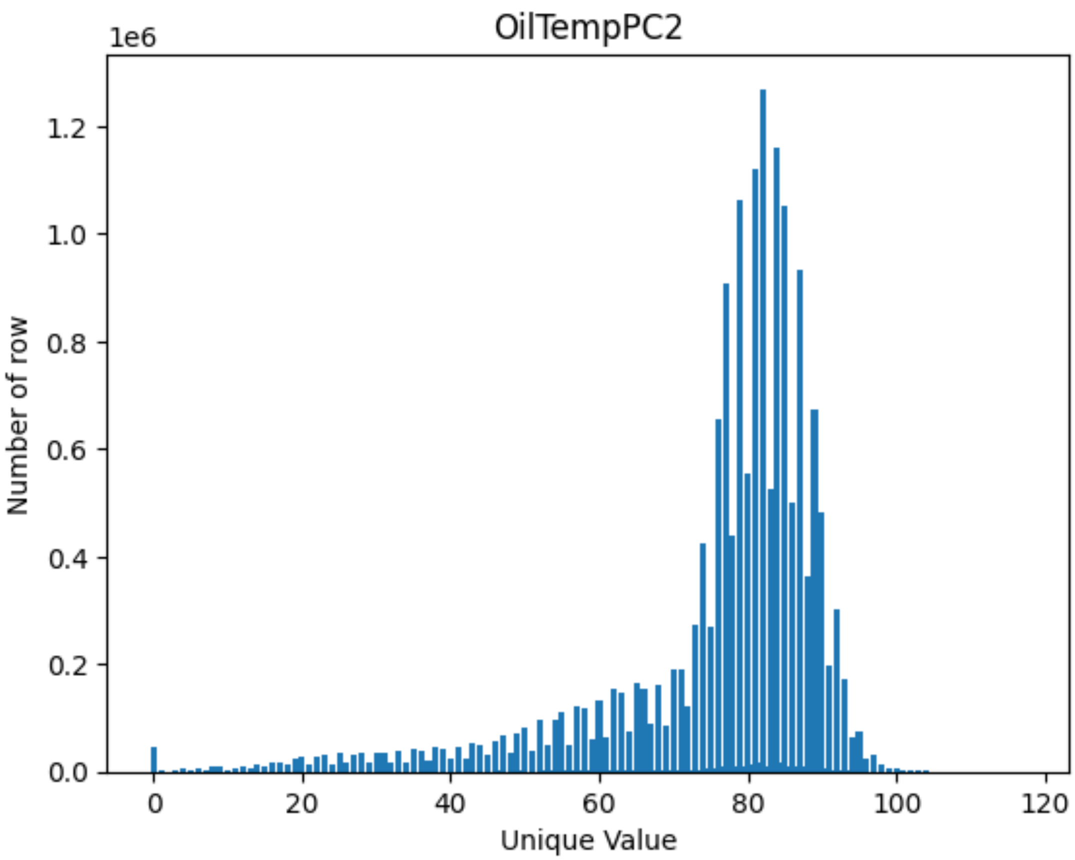



```
111275
119331
```

```
OilTempPC1 = data['RS_T_OilTemp_PC1'].value_counts()
OilTempPC2 = data['RS_T_OilTemp_PC2'].value_counts()

plt.bar(OilTempPC1.index, OilTempPC1.values)
plt.xlabel('Unique Value')
plt.ylabel('Number of row')
plt.title('OilTempPC1')
plt.show()

plt.bar(OilTempPC2.index, OilTempPC2.values)
plt.xlabel('Unique Value')
plt.ylabel('Number of row')
plt.title('OilTempPC2')
plt.show()

print(data['RS_T_OilTemp_PC1'].value_counts()[0])
print(data['RS_T_OilTemp_PC2'].value_counts()[0])
```



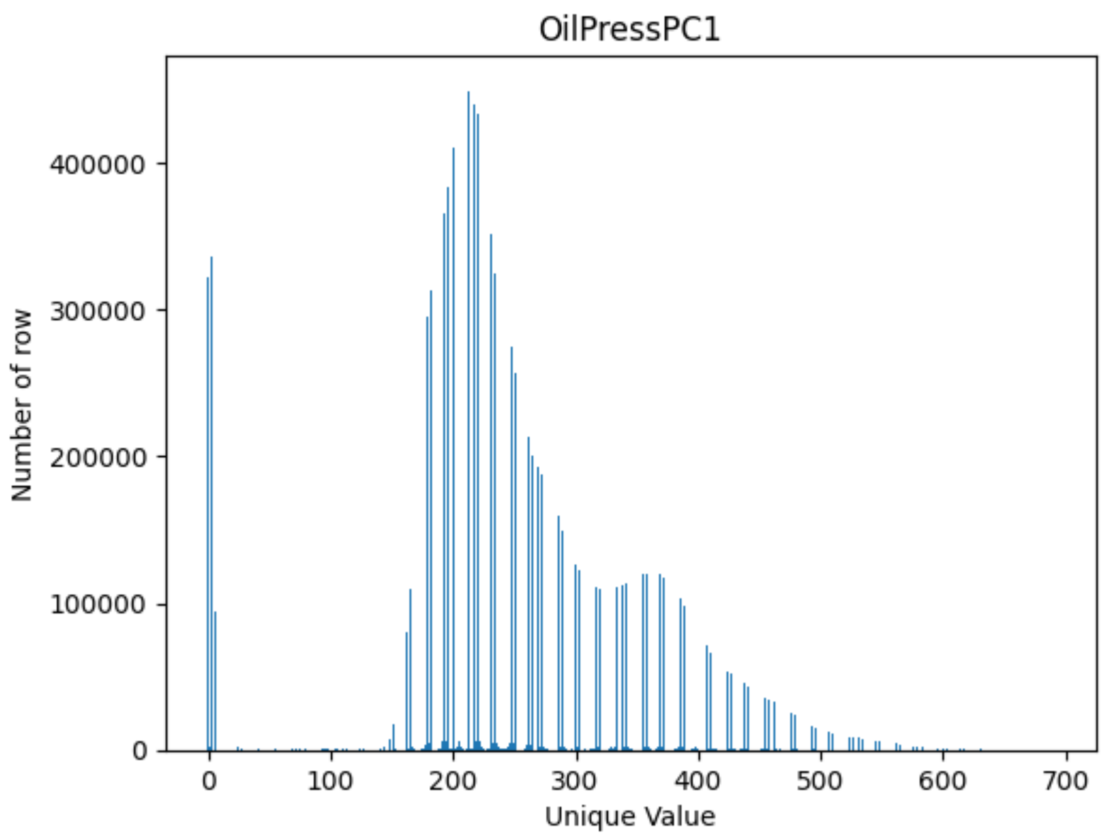
OilTempPC1
1e6
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Number of row
−100
−50
0
50
100
Unique Value

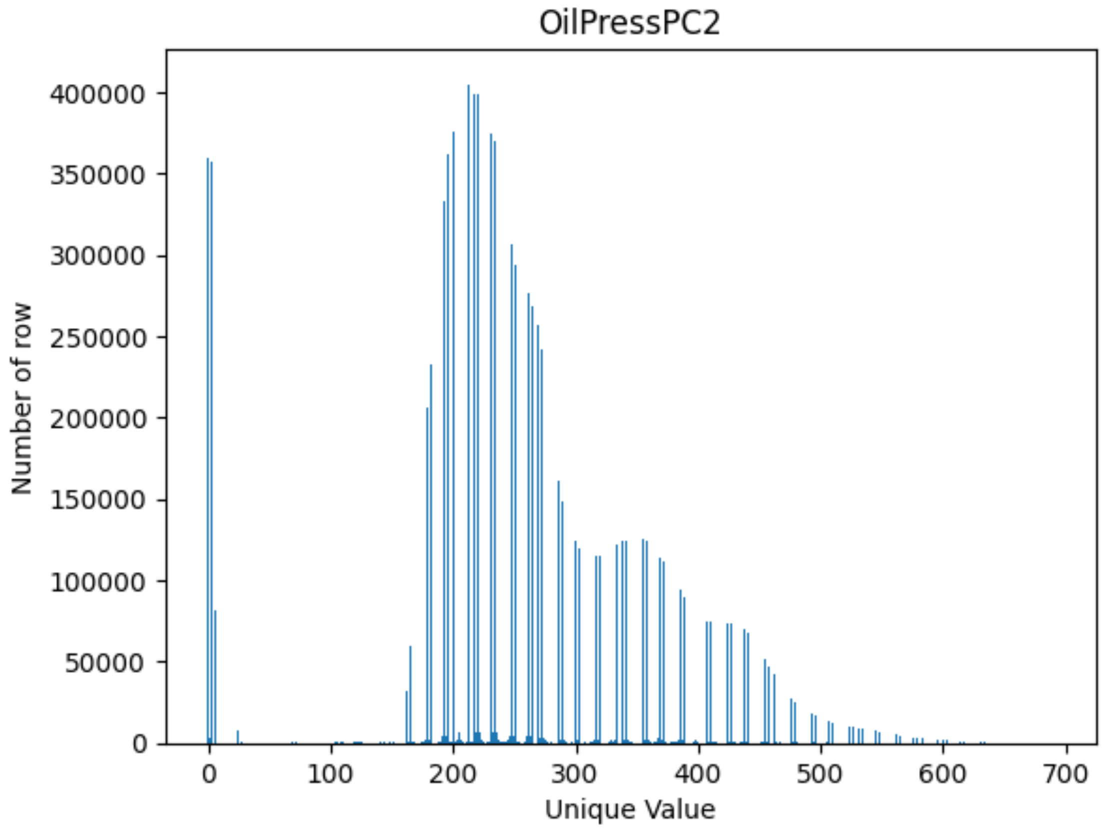


OilTempPC2
1e6
1.2
1.0
0.8
0.6
0.4
0.2
0.0
Number of row
0
20
40
60
80
100
120
Unique Value

13862
44976

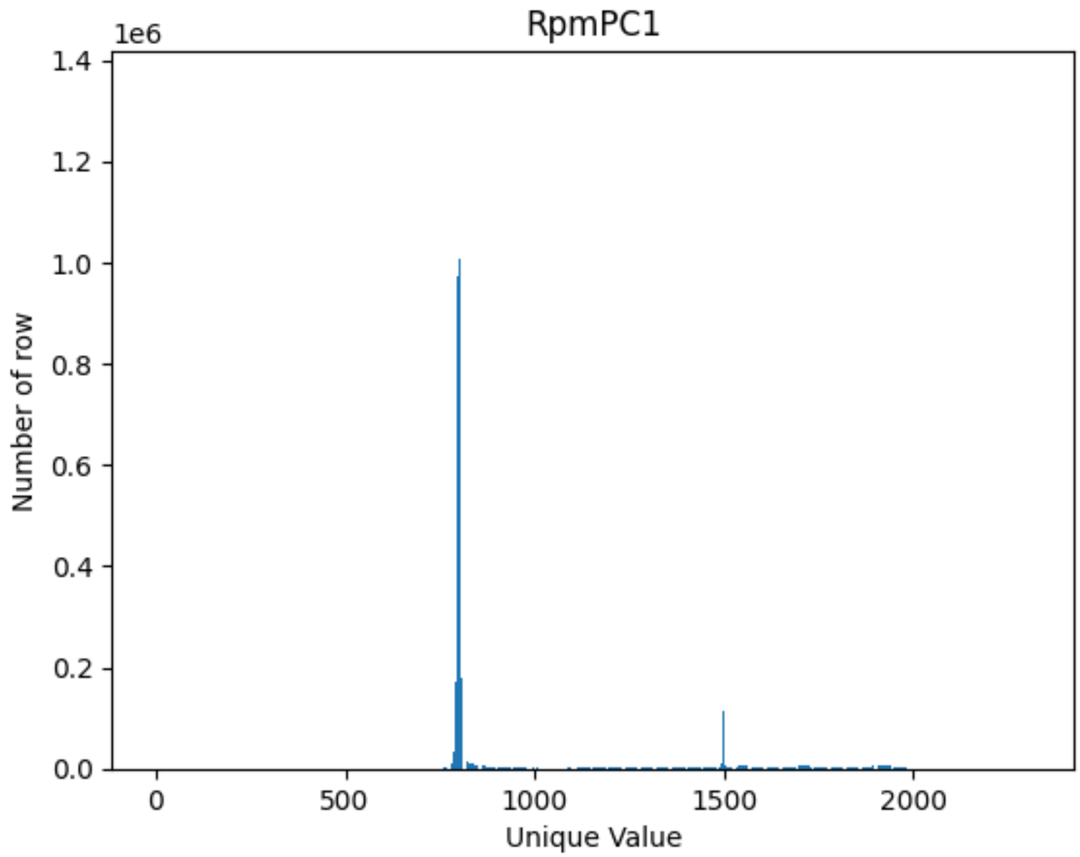
```python
OilPressPC1 = data['RS_E_OilPress_PC1'].value_counts()
OilPressPC2 = data['RS_E_OilPress_PC2'].value_counts()

plt.bar(OilPressPC1.index, OilPressPC1.values)
plt.xlabel('Unique Value')
plt.ylabel('Number of row')
plt.title('OilPressPC1')
plt.show()

plt.bar(OilPressPC2.index, OilPressPC2.values)
plt.xlabel('Unique Value')
plt.ylabel('Number of row')
plt.title('OilPressPC2')
plt.show()

print(data['RS_E_OilPress_PC1'].value_counts()[0])
print(data['RS_E_OilPress_PC2'].value_counts()[0])
```

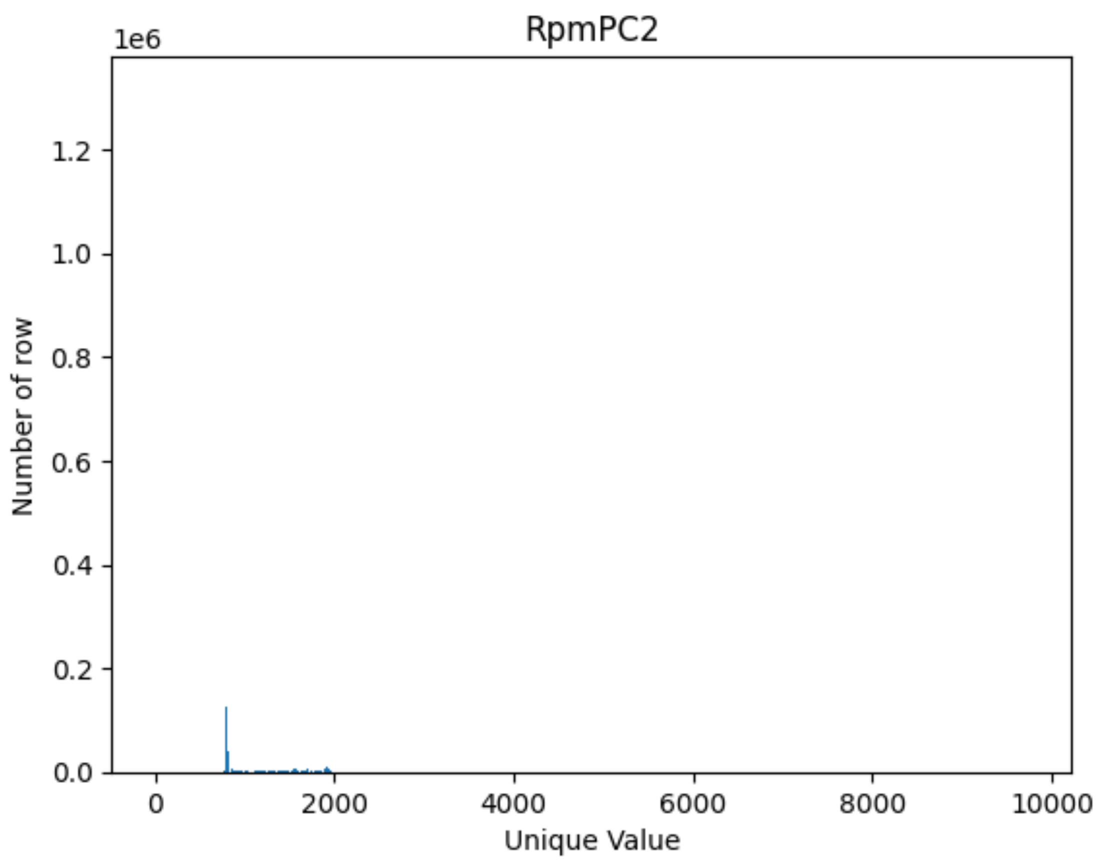





```
321704
360022
```

```python
RpmPC1 = data['RS_E_RPM_PC1'].value_counts()
RpmPC2 = data['RS_E_RPM_PC2'].value_counts()

plt.bar(RpmPC1.index, RpmPC1.values)
plt.xlabel('Unique Value')
plt.ylabel('Number of row')
plt.title('RpmPC1')
plt.show()

plt.bar(RpmPC2.index, RpmPC2.values)
plt.xlabel('Unique Value')
plt.ylabel('Number of row')
plt.title('RpmPC2')
plt.show()

print(data['RS_E_RPM_PC1'].value_counts()[0])
print(data['RS_E_RPM_PC2'].value_counts()[0])
```






```
884204
967780
```

The dataset reveals numerous instances where the water temperature, oil temperature, and oil pressure observations are marked as 0. Additionally, there are observations that surpass predefined thresholds. It is reasonable to conclude that values equal to 0 represent anomalies in sensor readings, as such values are not physically plausible.

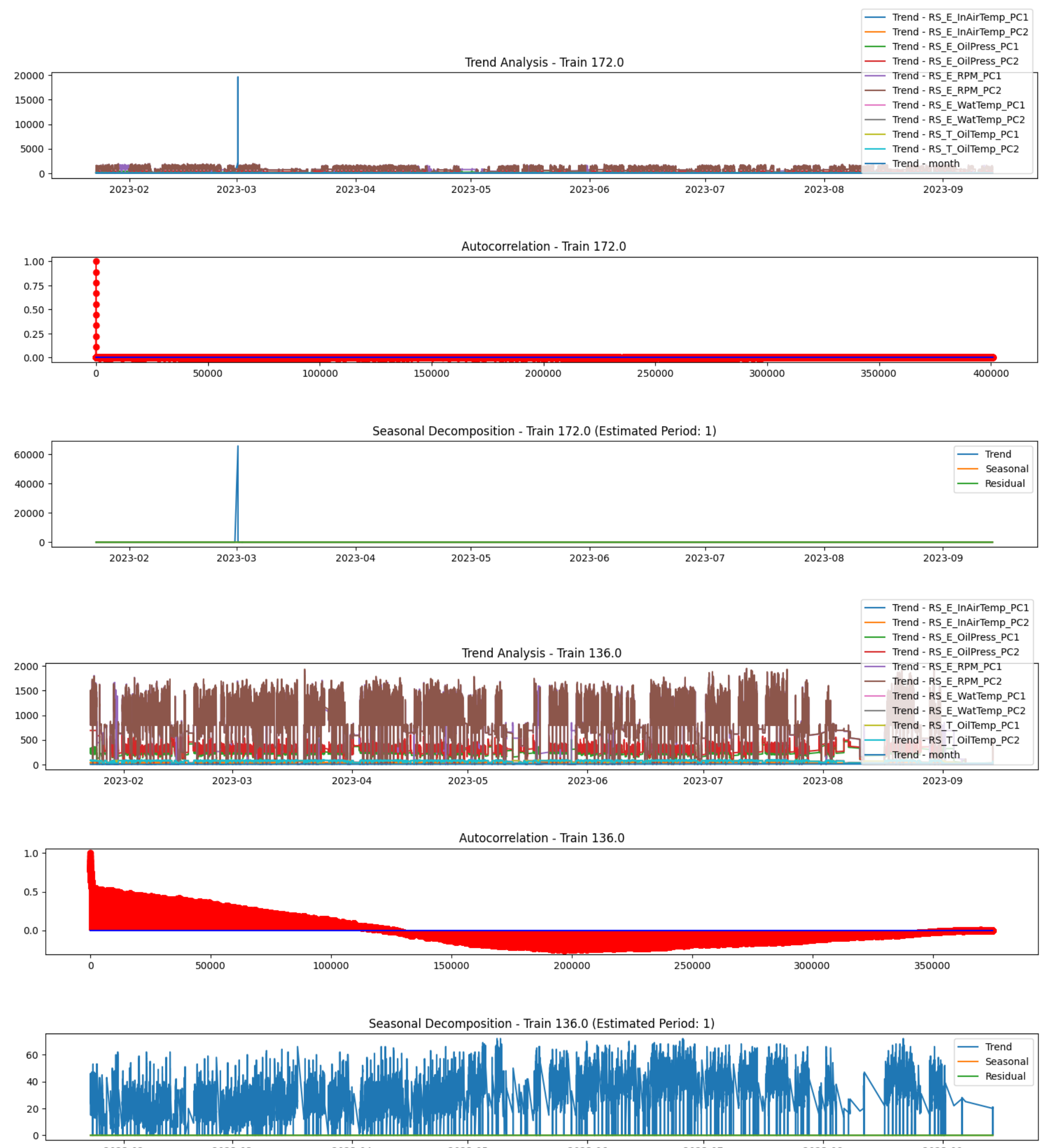
Trends, Season and cycle exploration

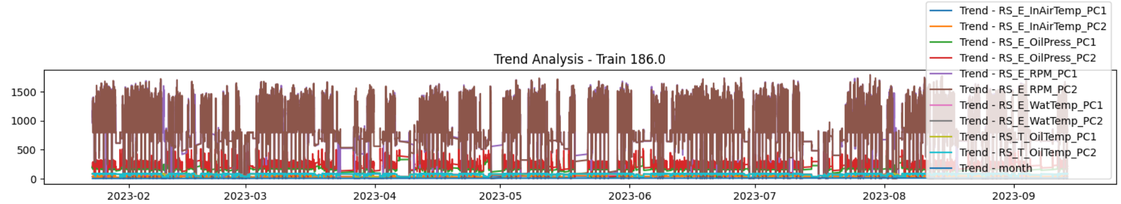
We now examine different aspects of time series data, including trends, seasonal decomposition, and autocorrelation/lag. The analysis will be demonstrated with the first three trains for efficiency, but the number of trains can be customized if desired. The outcomes consistently exhibit similarity.

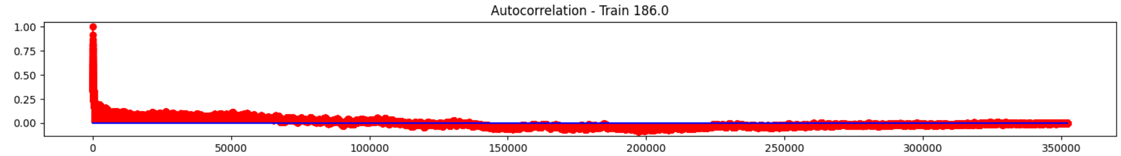
```
import statsmodels.api as sm

top_trains = data['mapped_veh_id'].value_counts().nlargest(3).index

for train_id in top_trains:
    train_data = data[data['mapped_veh_id'] == train_id]
    train_data = train_data.set_index('timestamps_UTC')

    plt.figure(figsize=(15, 10))


    plt.subplot(4, 1, 2)
    for numeric_column in numeric_columns:
        rolling_mean = train_data[numeric_column].rolling(window=30).mean()
        plt.plot(train_data.index, rolling_mean, label=f'Trend - {numeric_column}')
    plt.title(f'Trend Analysis - Train {train_id}')
    plt.legend()


    acf_result = sm.tsa.acf(train_data[numeric_columns[0]], nlags=len(train_data)-1)


    plt.subplot(4, 1, 3)
    lags = np.arange(len(acf_result))
    plt.stem(lags, acf_result, basefmt='b-', linefmt='r-', markerfmt='ro')
    plt.title(f'Autocorrelation - Train {train_id}')

    threshold = 2 / np.sqrt(len(train_data))
    significant_lags = np.where(np.abs(acf_result[1:]) > threshold)[0] + 1
    if len(significant_lags) > 0:
        estimated_period = significant_lags[0]
    else:
        estimated_period = 1

    result = sm.tsa.seasonal_decompose(train_data[numeric_columns[0]], period=estimated_period)

    plt.subplot(4, 1, 4)
    plt.plot(result.trend, label='Trend')
    plt.plot(result.seasonal, label='Seasonal')
    plt.plot(result.resid, label='Residual')
```

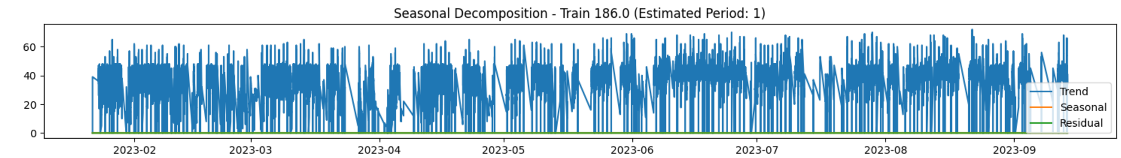
```
    plt.title(f'Seasonal Decomposition - Train {train_id} (Estimated Period: {esti
mated_period})')
    plt.legend()

    plt.tight_layout()
    plt.show()
```

```
/tmp/ipykernel_11368/2682445253.py:44: UserWarning: Creating legend with loc="best
" can be slow with large amounts of data.
  plt.tight_layout()
/home/jibril/.local/lib/python3.8/site-packages/IPython/core/pylabtools.py:152: Us
erWarning: Creating legend with loc="best" can be slow with large amounts of data.
  fig.canvas.print_figure(bytes_io, **kw)
```









In this analysis, we computed trend analysis, autocorrelation, and seasonal decomposition for the three trains with the most extensive data. The results indicate an absence of clear autocorrelation between past and present values. Figure 1, depicting the moving mean, reveals a consistent pattern, suggesting a lack of discernible trends in our data. Additionally, the absence of seasonality or residual patterns in the last graph indicates that the data is relatively stable, lacking significant fluctuations and seasonal patterns. Overall, these observations suggest that the operation of the trains remains somewhat constant, devoid of discernible trends.

This is to be expected since, over time, train operations tend to exhibit similarities. Variables such as air temperature, water temperature, and rpm remain relatively constant with each use of the trains.

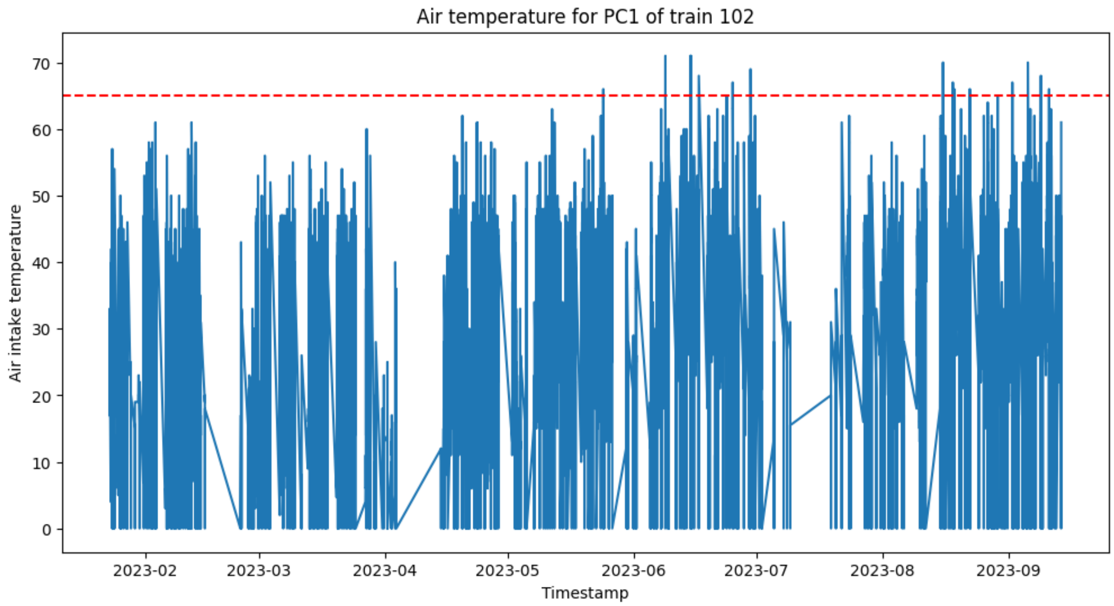
The consistency in these environmental variables is not surprising, as they are expected to remain stable during each operation of the train. However, this does not eliminate the possibility of anomalies occurring sporadically. It simply suggests that these anomalies do not follow a seasonal pattern. Hence, techniques for detecting such anomalies become crucial.

The absence of seasons, trends, or patterns makes it challenging to rely on historical data for future predictions. However, our primary goal is anomaly detection, requiring a different approach.

Taking train 102 as an example, specifically focusing on the intake air temperature for PC1, we observe instances where it surpasses the specified threshold for safe operations. These thresholds are set at 65°C for air, 100°C for water, and 115°C for oil. As evident from the previous data description, it's apparent that certain trains exceed these safety thresholds.

```
train_102_data = data[data['mapped_veh_id']==102]
train_102_data.loc[:,'timestamps_UTC'] = pd.to_datetime(train_102_data['timestamps_UTC'])
train_102_data = train_102_data.sort_values(by='timestamps_UTC')

plt.figure(figsize=(12, 6))
plt.plot(train_102_data['timestamps_UTC'], train_102_data['RS_E_InAirTemp_PC1'])
plt.axhline(y=65, color='r', linestyle='--', label=f'Anormaly treshold')
plt.xlabel('Timestamp')
plt.ylabel('Air intake temperature')
plt.title('Air temperature for PC1 of train 102')
```

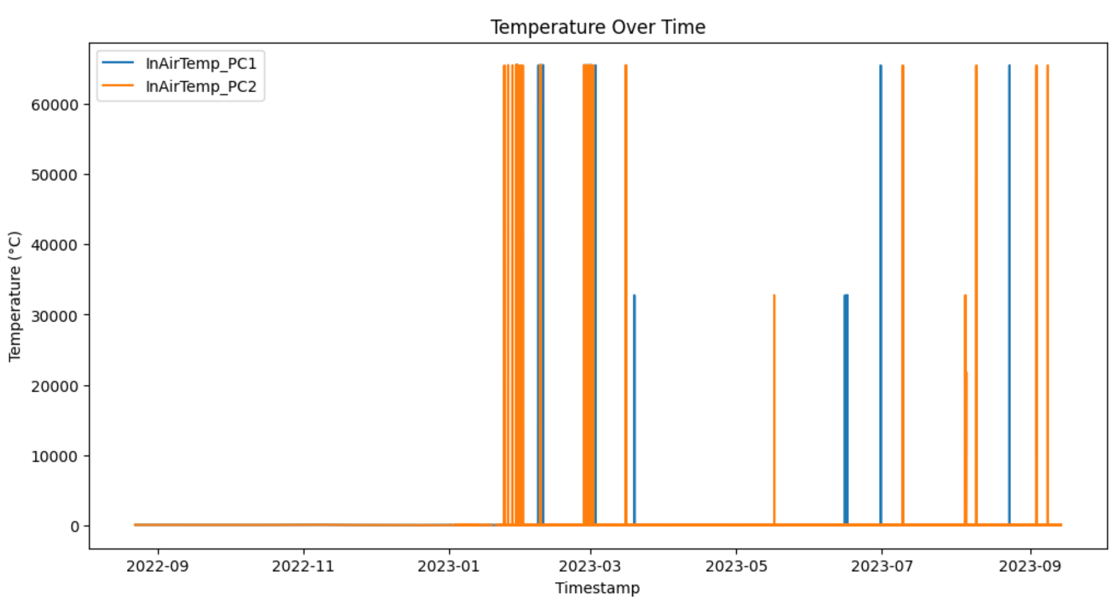
```
plt.show()
```



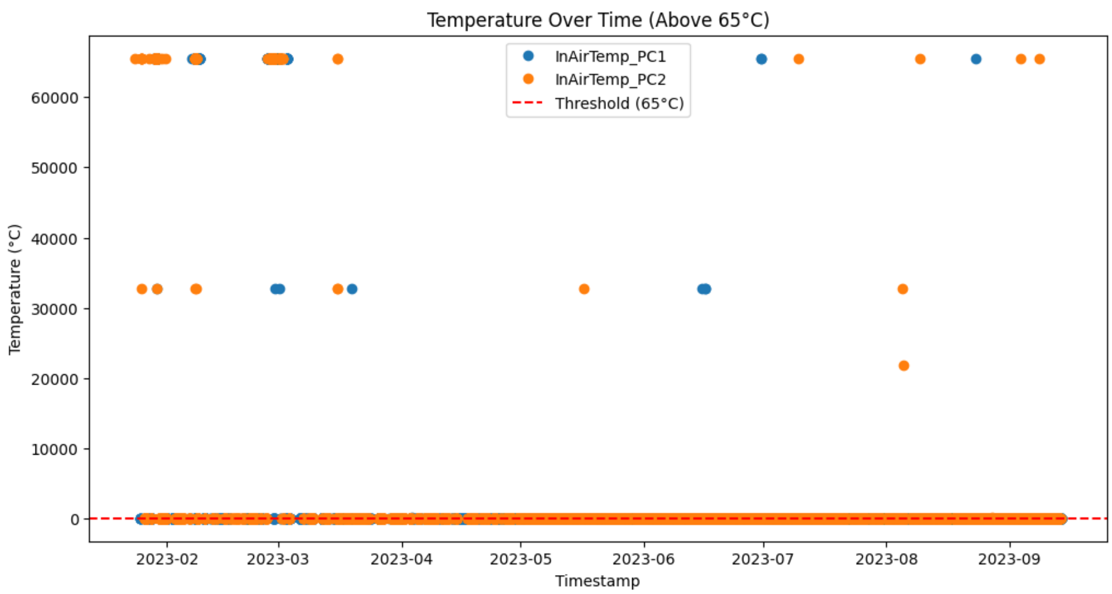
In the graph above, it is noteworthy that instances of the air intake temperature exceeding the threshold predominantly occur during the summer months. Moreover, these occurrences are not isolated but happen repeatedly.

Through additional graphical visualizations, we delve deeper into this variable and pinpoint the trains that predominantly exhibit this anomalous behavior.

```
plt.figure(figsize=(12, 6))
plt.plot(data['timestamps_UTC'], data['RS_E_InAirTemp_PC1'], label='InAirTemp_PC1'
)
plt.plot(data['timestamps_UTC'], data['RS_E_InAirTemp_PC2'], label='InAirTemp_PC2'
)
plt.title('Temperature Over Time')
plt.xlabel('Timestamp')
plt.ylabel('Temperature (°C)')
plt.legend()
plt.show()
```



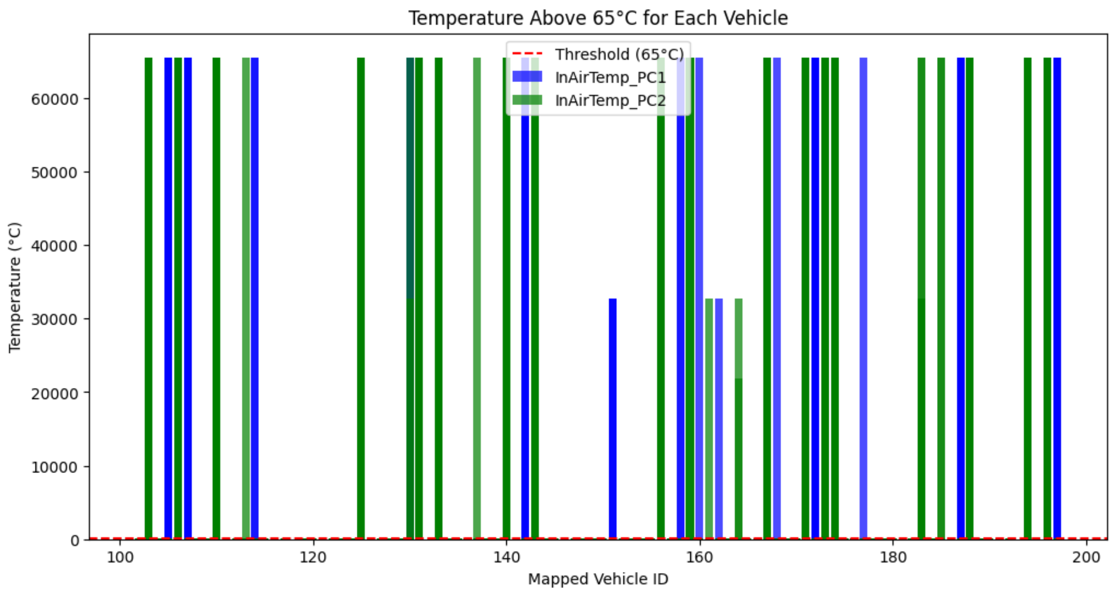
```
plt.figure(figsize=(12, 6))

filtered_data_pc1 = data[data['RS_E_InAirTemp_PC1'] > 65]
filtered_data_pc2 = data[data['RS_E_InAirTemp_PC2'] > 65]

plt.plot(filtered_data_pc1['timestamps_UTC'], filtered_data_pc1['RS_E_InAirTemp_PC
1'], 'o', label='InAirTemp_PC1')
plt.plot(filtered_data_pc2['timestamps_UTC'], filtered_data_pc2['RS_E_InAirTemp_PC
2'], 'o', label='InAirTemp_PC2')

plt.axhline(y=65, color='r', linestyle='--', label='Threshold (65°C)')

plt.title('Temperature Over Time (Above 65°C)')
plt.xlabel('Timestamp')
plt.ylabel('Temperature (°C)')
plt.legend()
plt.show()
```




```
plt.figure(figsize=(12, 6))

filtered_data_pc1 = data[data['RS_E_InAirTemp_PC1'] > 65]
filtered_data_pc2 = data[data['RS_E_InAirTemp_PC2'] > 65]

plt.bar(filtered_data_pc1['mapped_veh_id'], filtered_data_pc1['RS_E_InAirTemp_PC1'
], color='blue', label='InAirTemp_PC1', alpha=0.7)
plt.bar(filtered_data_pc2['mapped_veh_id'], filtered_data_pc2['RS_E_InAirTemp_PC2'
], color='green', label='InAirTemp_PC2', alpha=0.7)

plt.axhline(y=65, color='r', linestyle='--', label='Threshold (65°C)')

plt.title('Temperature Above 65°C for Each Vehicle')
plt.xlabel('Mapped Vehicle ID')
plt.ylabel('Temperature (°C)')
plt.legend()
plt.show()
```

```
/home/jibril/.local/lib/python3.10/site-packages/IPython/core/pylabtools.py:152: U
serWarning: Creating legend with loc="best" can be slow with large amounts of data
```

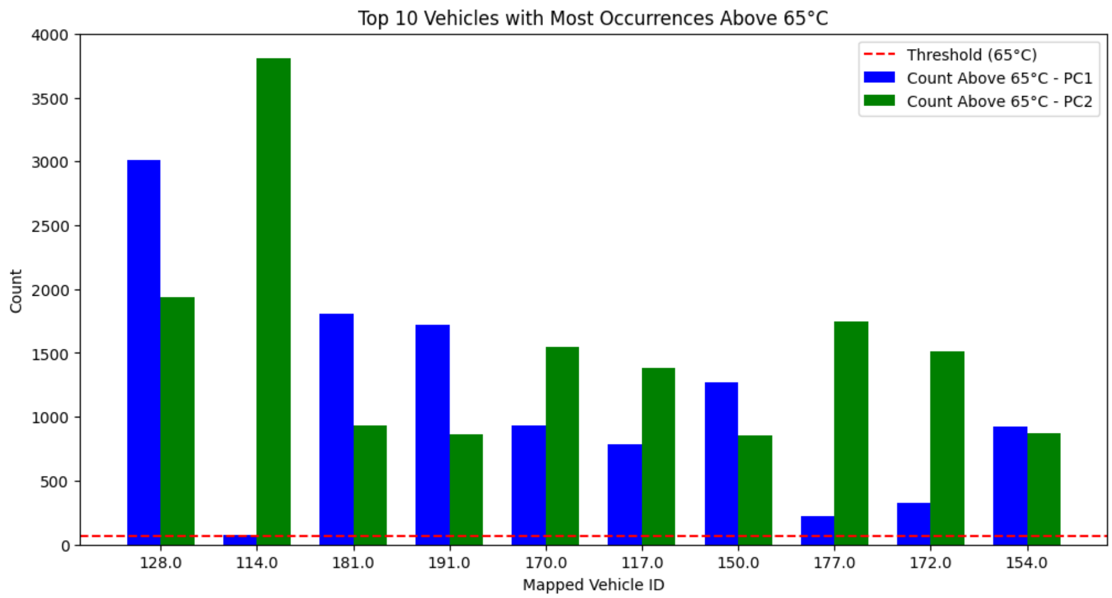
.
```
fig.canvas.print_figure(bytes_io, **kw)
```



We now opt to select the top 10 vehicles that surpass the 65-degree threshold, given the substantial number of vehicles exceeding this limit.

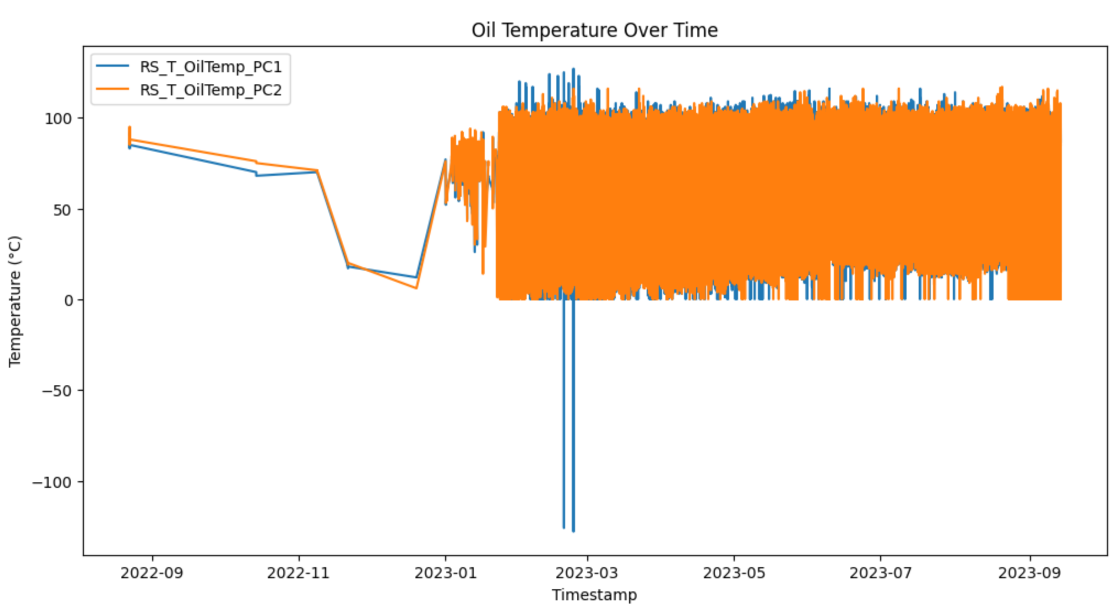
```
plt.figure(figsize=(12, 6))

filtered_data_pc1 = data[data['RS_E_InAirTemp_PC1'] > 65]
filtered_data_pc2 = data[data['RS_E_InAirTemp_PC2'] > 65]

count_pc1 = filtered_data_pc1.groupby('mapped_veh_id').size().to_frame(name='count
_pc1')
count_pc2 = filtered_data_pc2.groupby('mapped_veh_id').size().to_frame(name='count
_pc2')

merged_data = pd.merge(count_pc1, count_pc2, how='outer', left_index=True, right_i
ndex=True)

merged_data['total_count'] = merged_data['count_pc1'] + merged_data['count_pc2']
top_10_vehicles = merged_data.sort_values(by='total_count', ascending=False).head(
10)

bar_width = 0.35
index = np.arange(len(top_10_vehicles))

plt.bar(index, top_10_vehicles['count_pc1'], bar_width, color='blue', label='Count
Above 65°C - PC1')
plt.bar(index + bar_width, top_10_vehicles['count_pc2'], bar_width, color='green',
label='Count Above 65°C - PC2')

plt.axhline(y=65, color='r', linestyle='--', label='Threshold (65°C)')

plt.title('Top 10 Vehicles with Most Occurrences Above 65°C')
plt.xlabel('Mapped Vehicle ID')
plt.ylabel('Count')
plt.xticks(index + bar_width / 2, top_10_vehicles.index)
```

```
plt.legend()
plt.show()
```

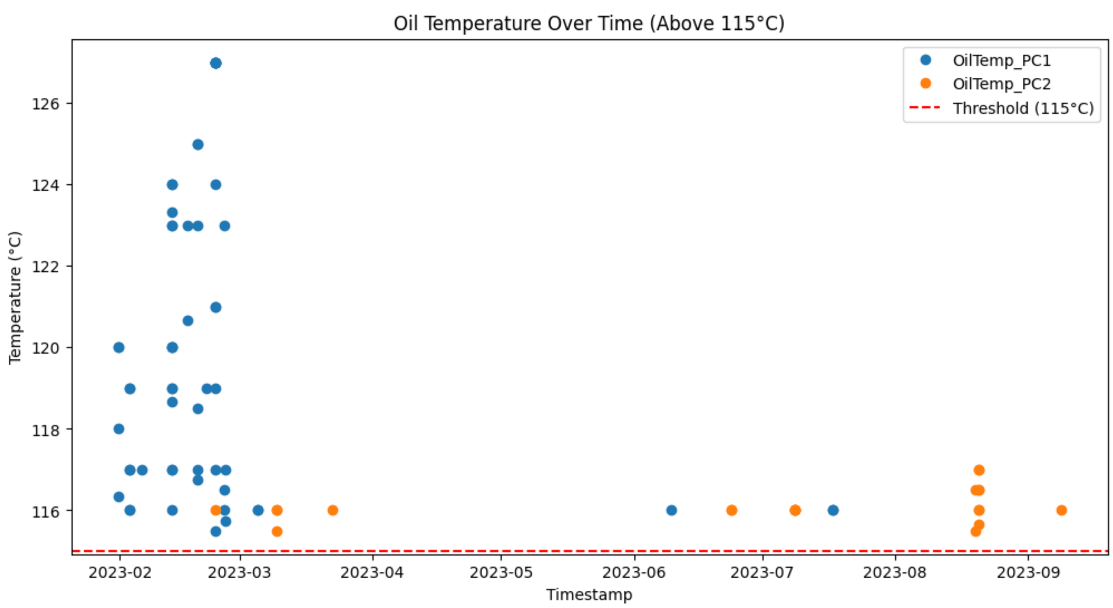


Mirroring the earlier analysis, we also investigate the oil temperature and water temperature. Notably, it is evident that a majority of values surpassing the maximum oil temperature are observed in the first engine. Regarding trains that consistently exceed this threshold, they are evenly distributed between the first and second engine.

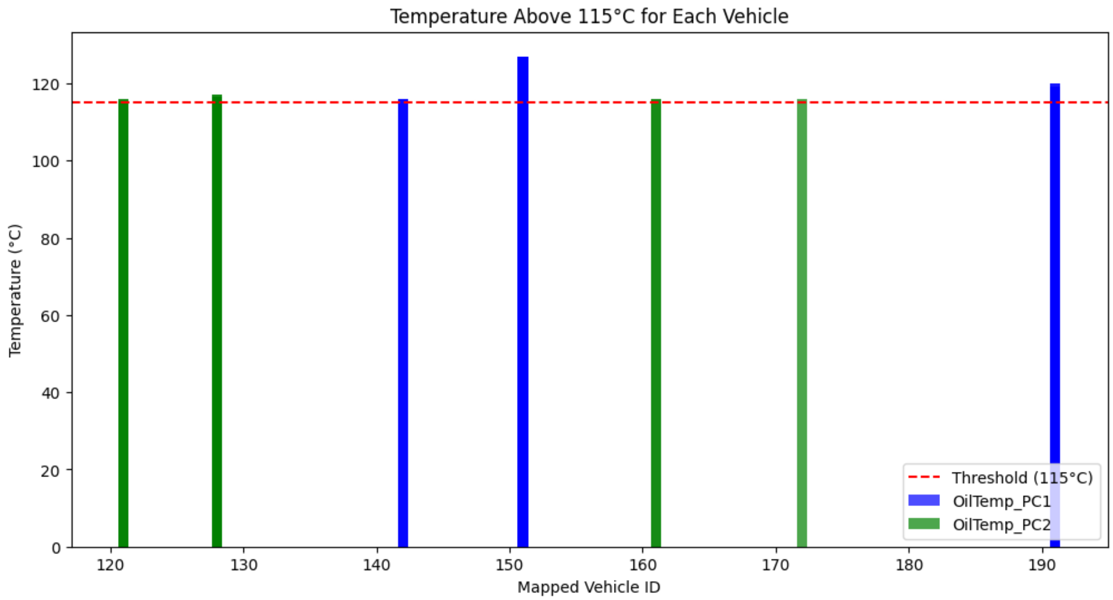
```
plt.figure(figsize=(12, 6))
plt.plot(data['timestamps_UTC'], data['RS_T_OilTemp_PC1'], label='RS_T_OilTemp_PC1
')
plt.plot(data['timestamps_UTC'], data['RS_T_OilTemp_PC2'], label='RS_T_OilTemp_PC2
')
plt.title('Oil Temperature Over Time')
plt.xlabel('Timestamp')
plt.ylabel('Temperature (°C)')
plt.legend()
plt.show()
```



```
plt.figure(figsize=(12, 6))

filtered_data_pc1 = data[data['RS_T_OilTemp_PC1'] > 115]
filtered_data_pc2 = data[data['RS_T_OilTemp_PC2'] > 115]

plt.plot(filtered_data_pc1['timestamps_UTC'], filtered_data_pc1['RS_T_OilTemp_PC1'
], 'o', label='OilTemp_PC1')
plt.plot(filtered_data_pc2['timestamps_UTC'], filtered_data_pc2['RS_T_OilTemp_PC2'
], 'o', label='OilTemp_PC2')

plt.axhline(y=115, color='r', linestyle='--', label='Threshold (115°C)')

plt.title('Oil Temperature Over Time (Above 115°C)')
plt.xlabel('Timestamp')
plt.ylabel('Temperature (°C)')
plt.legend()
plt.show()
```

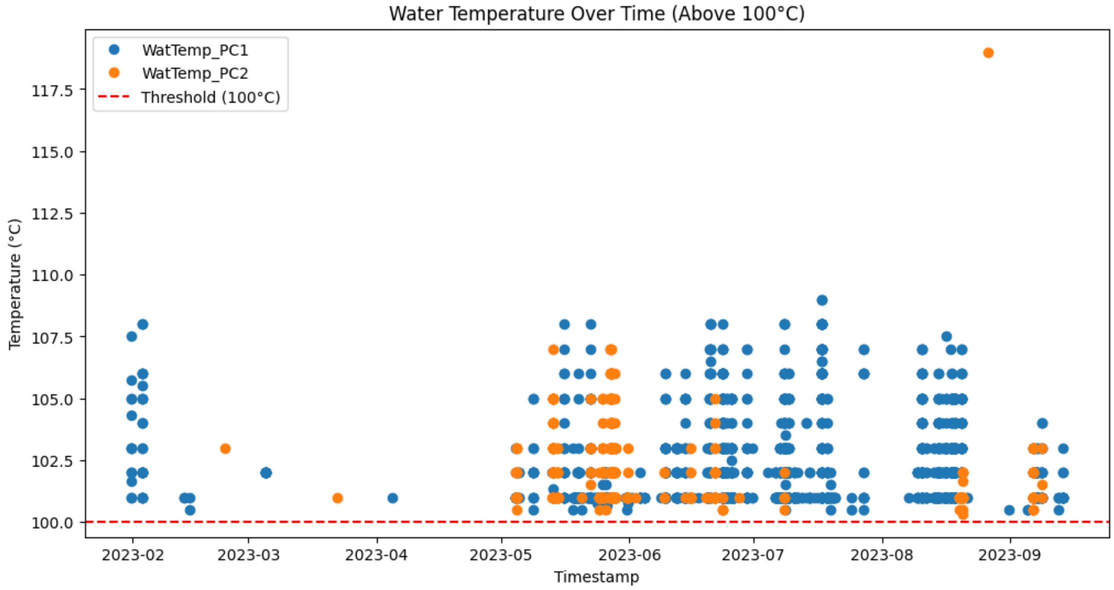



```
plt.figure(figsize=(12, 6))

filtered_data_pc1 = data[data['RS_T_OilTemp_PC1'] > 115]
filtered_data_pc2 = data[data['RS_T_OilTemp_PC2'] > 115]

plt.bar(filtered_data_pc1['mapped_veh_id'], filtered_data_pc1['RS_T_OilTemp_PC1'],
color='blue', label='OilTemp_PC1', alpha=0.7)
plt.bar(filtered_data_pc2['mapped_veh_id'], filtered_data_pc2['RS_T_OilTemp_PC2'],
color='green', label='OilTemp_PC2', alpha=0.7)

plt.axhline(y=115, color='r', linestyle='--', label='Threshold (115°C)')

plt.title('Temperature Above 115°C for Each Vehicle')
plt.xlabel('Mapped Vehicle ID')
plt.ylabel('Temperature (°C)')
plt.legend()
plt.show()
```



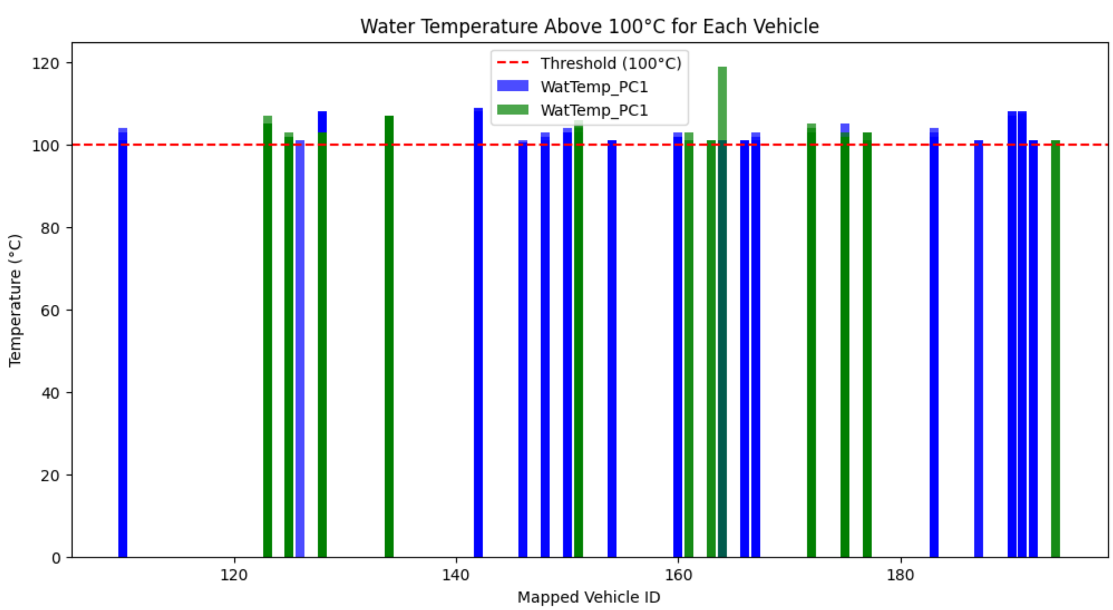
We have performed a similar analysis for water temperature, revealing that observations exceeding the threshold primarily occur during the hottest months of the year.

```
plt.figure(figsize=(12, 6))

# Filtrer les données pour ne prendre que celles au-dessus de 65 degrés
filtered_data_pc1 = data[data['RS_E_WatTemp_PC1'] > 100]
filtered_data_pc2 = data[data['RS_E_WatTemp_PC2'] > 100]

# Tracer les données filtrées
plt.plot(filtered_data_pc1['timestamps_UTC'], filtered_data_pc1['RS_E_WatTemp_PC1'
], 'o', label='WatTemp_PC1')
plt.plot(filtered_data_pc2['timestamps_UTC'], filtered_data_pc2['RS_E_WatTemp_PC2'
], 'o', label='WatTemp_PC2')

plt.axhline(y=100, color='r', linestyle='--', label='Threshold (100°C)')  # Ajoute
r une ligne pour le seuil

plt.title('Water Temperature Over Time (Above 100°C)')
plt.xlabel('Timestamp')
plt.ylabel('Temperature (°C)')
plt.legend()
plt.show()
```

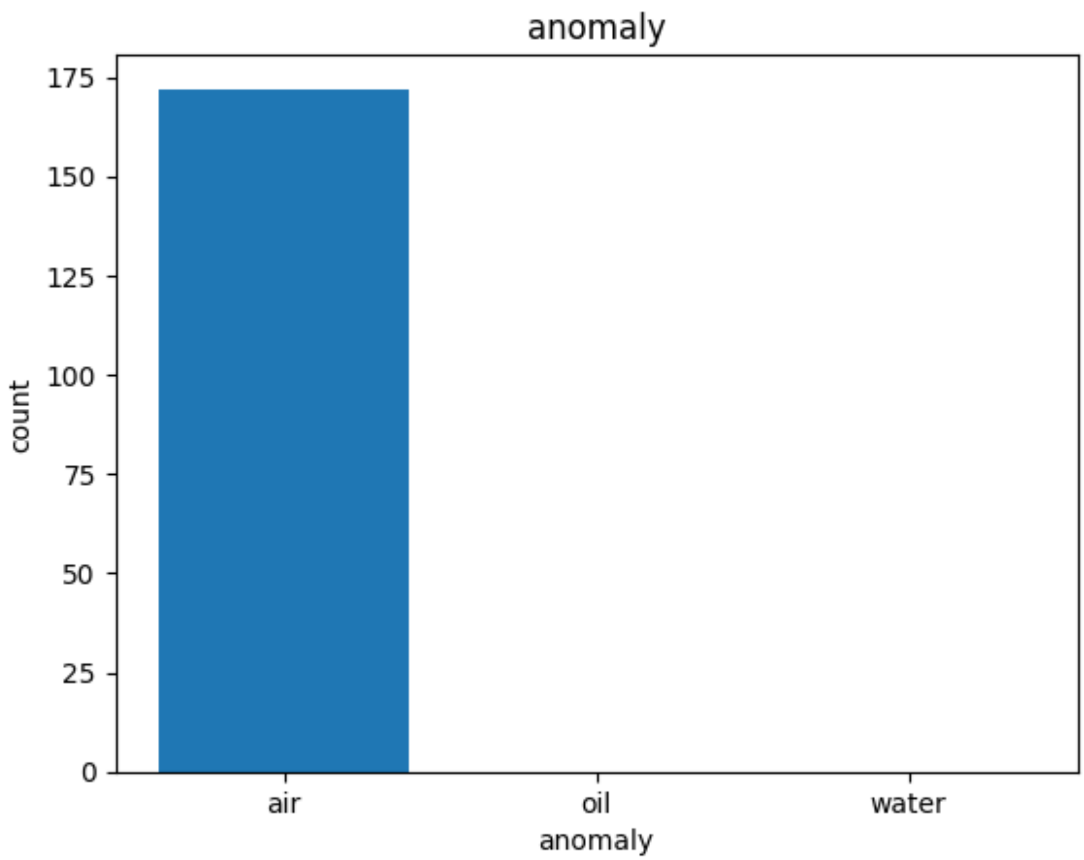


We can see that the water temperature exceeds the threshold of 100 degrees especially during the summer

```
plt.figure(figsize=(12, 6))

filtered_data_pc1 = data[data['RS_E_WatTemp_PC1'] > 100]
filtered_data_pc2 = data[data['RS_E_WatTemp_PC2'] > 100]

plt.bar(filtered_data_pc1['mapped_veh_id'], filtered_data_pc1['RS_E_WatTemp_PC1'],
color='blue', label='WatTemp_PC1', alpha=0.7)
plt.bar(filtered_data_pc2['mapped_veh_id'], filtered_data_pc2['RS_E_WatTemp_PC2'],
color='green', label='WatTemp_PC1', alpha=0.7)

plt.axhline(y=100, color='r', linestyle='--', label='Threshold (100°C)')

plt.title('Water Temperature Above 100°C for Each Vehicle')
plt.xlabel('Mapped Vehicle ID')
plt.ylabel('Temperature (°C)')
plt.legend()
plt.show()
```



```
train_102_pc1_inair_anomalies = train_102_data[train_102_data['RS_E_InAirTemp_PC1'
]>65]
train_102_pc1_oil_anomalies = train_102_data[train_102_data['RS_T_OilTemp_PC1']>11
5]
train_102_pc1_water_anomalies = train_102_data[train_102_data['RS_E_WatTemp_PC1']>
100]

df = {'anomaly':['air','oil','water'],'count':[len(train_102_pc1_inair_anomalies),
len(train_102_pc1_oil_anomalies),len(train_102_pc1_water_anomalies)]}
df = pd.DataFrame(df)
plt.figure()
plt.bar(df['anomaly'],df['count'])
plt.xlabel('anomaly')
plt.ylabel('count')
plt.title('anomaly')

plt.show()
```



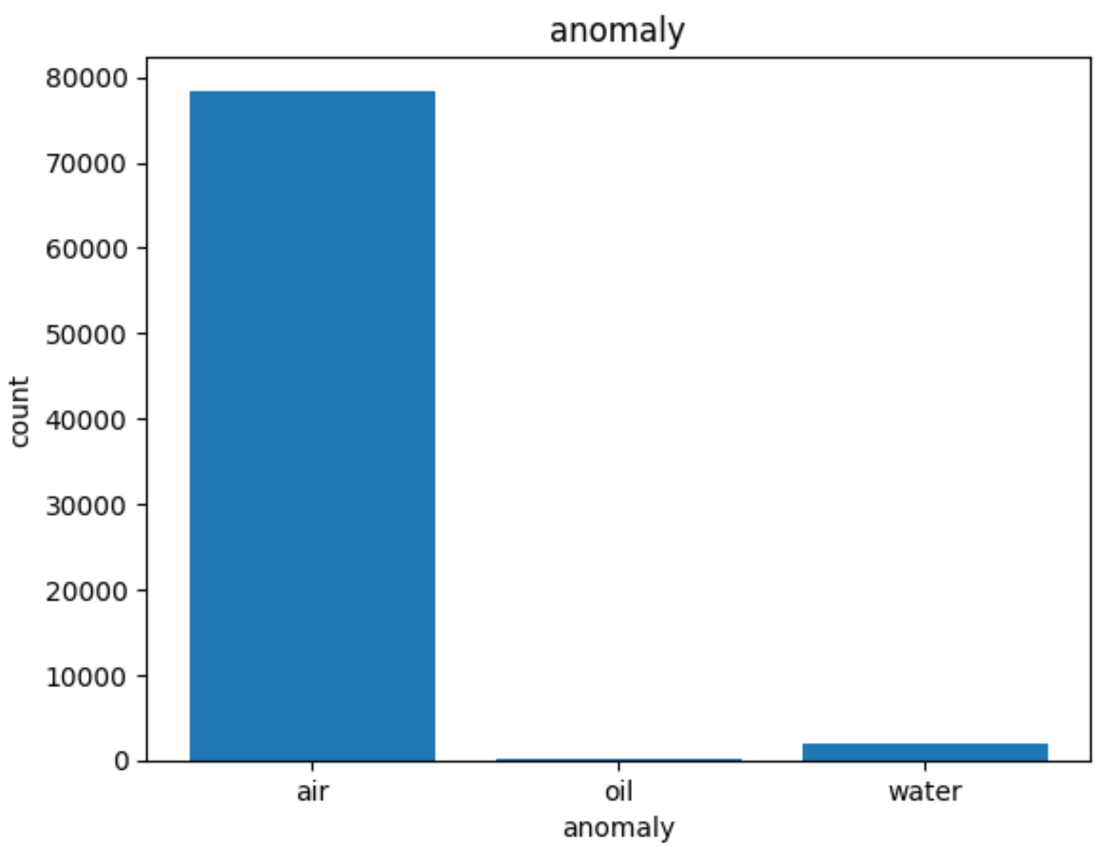
In this instance, we observe that, for train 102, occurrences of the air intake temperature surpassing the threshold are more frequent compared to oil and water temperatures. Consequently, it is crucial to examine whether this pattern holds true for all trains.

Let's examine the data for all trains collectively.

```
all_train_air_threshold = data[(data['RS_E_InAirTemp_PC1'] > 65) | (data['RS_E_InA
irTemp_PC2'] > 65)]
all_train_oil_threshold = data[(data['RS_T_OilTemp_PC1']>115) | (data['RS_T_OilTem
p_PC2']>115)]
all_train_water_threshold = data[(data['RS_E_WatTemp_PC1']>100) | (data['RS_E_WatT
emp_PC2']>100)]
```

```
df = {'anomaly':['air','oil','water'],'count':[len(all_train_air_threshold),len(al
l_train_oil_threshold),len(all_train_water_threshold)]}
df = pd.DataFrame(df)
plt.figure()
plt.bar(df['anomaly'],df['count'])
plt.xlabel('anomaly')
plt.ylabel('count')
plt.title('anomaly')
```

```
Text(0.5, 1.0, 'anomaly')
```



Now, upon analyzing data from all trains and both heads of the train, it becomes apparent that air intake exceeds the threshold more frequently. Water and oil thresholds are less frequently breached, suggesting their potential higher criticality. Additionally, the correlation matrix indicates a somewhat strong correlation between oil and water variables.

```
print('Air temperature anomalies:')
print(all_train_air_threshold.describe())
print('Oil temperature anomalies:')
print(all_train_oil_threshold.describe())
print('Water temperature anomalies:')
print(all_train_water_threshold.describe())
```

```
Air temperature anomalies:
               ID  mapped_veh_id                  timestamps_UTC       lat  \
count    78446.00       78446.00                           78446 78446.00
mean   8821986.96         150.98  2023-06-28 16:11:26.107258624     50.96
min        769.00         102.00             2023-01-24 00:37:18     49.99
25%    4431583.00         128.00  2023-06-04 17:57:15.249999872     50.93
50%    8825215.00         151.00             2023-06-25 08:33:26     51.02
```

```
75%   13246159.00         175.00  2023-08-11 12:28:35.249999872    51.18
max   17679233.00         197.00            2023-09-13 21:51:11    51.25
std    5116800.77          26.56                            NaN     0.28

           lon  RS_E_InAirTemp_PC1  RS_E_InAirTemp_PC2  RS_E_OilPress_PC1  \
count 78446.00            78446.00            78384.00           78446.00
mean      4.37              427.02              477.33             107.79
min       3.58                0.00                0.00               0.00
25%       3.78               41.00               48.00               3.00
50%       4.04               65.00               66.00               6.00
75%       5.11               68.00               69.00             213.00
max       5.54            65535.00            65535.00             690.00
std       0.65             4905.20             5202.68             145.92

       RS_E_OilPress_PC2  RS_E_RPM_PC1  RS_E_RPM_PC2  RS_E_WatTemp_PC1  \
count           78384.00      78446.00      78384.00          78446.00
mean               83.12        380.62        298.36             76.68
min                 0.00          0.00          0.00             -8.50
25%                 3.00          0.00          0.00             74.00
50%                 3.00          0.00          0.00             79.00
75%               193.00        799.00        796.00             83.00
max               690.00       2056.00       2281.00            109.00
std               135.60        532.84        511.69             11.73

       RS_E_WatTemp_PC2  RS_T_OilTemp_PC1  RS_T_OilTemp_PC2    month
count          78384.00          78446.00          78384.00 78446.00
mean              75.28             77.80             77.29     6.42
min              -17.00              0.00              0.00     1.00
25%               71.00             74.00             73.00     6.00
50%               78.00             79.00             78.00     6.00
75%               82.00             82.00             82.00     8.00
max              106.00            109.00            116.00     9.00
std               11.97              9.29             10.73     1.75
Oil temperature anomalies:
               ID  mapped_veh_id                 timestamps_UTC   lat   lon  \
count       76.00          76.00                             76 76.00 76.00
mean   8756755.88         151.78  2023-03-30 09:53:00.065789184 50.96  4.27
min     316002.00         121.00            2023-01-31 17:23:56 50.46  3.58
25%    4259480.00         142.00  2023-02-13 04:41:09.750000128 50.84  3.60
50%   10034938.50         151.00            2023-02-23 05:36:31 51.01  3.81
75%   12845028.75         151.00  2023-04-11 07:21:03.750000128 51.15  5.16
max   16604727.00         191.00            2023-09-08 14:33:26 51.20  5.35
std    4874592.70          20.22                            NaN  0.19  0.69

       RS_E_InAirTemp_PC1  RS_E_InAirTemp_PC2  RS_E_OilPress_PC1  \
count               76.00               76.00              76.00
mean                32.18               29.78             106.85
min                  0.00                0.00               0.00
25%                 21.75               15.00               6.00
50%                 34.50               30.50              10.00
75%                 41.25               41.62             196.00
max                 73.00               67.00             493.00
std                 18.89               14.66             160.42

       RS_E_OilPress_PC2  RS_E_RPM_PC1  RS_E_RPM_PC2  RS_E_WatTemp_PC1  \
count              76.00         76.00         76.00             76.00
```

```
mean              280.06         385.74       1081.01             62.04
min                 0.00           0.00          0.00              0.00
25%               232.12           0.00        799.88             47.00
50%               262.00           0.00        809.00             63.00
75%               342.75         802.50       1499.00             77.75
max               438.00        1897.00       1719.00            108.00
std                96.73         626.90        449.02             31.48

       RS_E_WatTemp_PC2  RS_T_OilTemp_PC1  RS_T_OilTemp_PC2  month
count             76.00             76.00             76.00  76.00
mean              81.72            105.03             92.31   3.38
min                0.00             44.50             46.00   1.00
25%               76.00            100.75             81.00   2.00
50%               86.00            117.00             88.00   2.00
75%               90.00            120.17            115.54   3.75
max              102.00            127.00            117.00   9.00
std               20.77             25.72             17.76   2.41
Water temperature anomalies:
               ID  mapped_veh_id                timestamps_UTC     lat  \
count     1986.00        1986.00                          1986 1986.00
mean   8897577.57         139.62  2023-06-29 02:25:05.297079552   51.06
min       4833.00         110.00           2023-01-31 17:23:35   50.75
25%    4401069.00         128.00  2023-05-27 15:11:51.249999872   50.99
50%    8836625.50         128.00           2023-06-25 13:51:15   51.05
75%   13339411.75         142.00  2023-08-10 13:48:51.249999872   51.16
max   17678918.00         194.00           2023-09-13 15:13:50   51.25
std    5159752.60          18.30                           NaN    0.10

          lon  RS_E_InAirTemp_PC1  RS_E_InAirTemp_PC2  RS_E_OilPress_PC1  \
count 1986.00             1986.00             1986.00            1986.00
mean     5.05               53.24               43.61             189.43
min      3.58               14.33                0.00               0.00
25%      5.11               44.00               40.00              10.00
50%      5.24               49.00               42.00             182.00
75%      5.32               60.38               46.00             310.00
max      5.54               87.00               85.00             503.00
std      0.51               14.48               11.19             140.33

       RS_E_OilPress_PC2  RS_E_RPM_PC1  RS_E_RPM_PC2  RS_E_WatTemp_PC1  \
count            1986.00       1986.00       1986.00           1986.00
mean              228.26        839.28       1094.37             97.84
min                 0.00          0.00          0.00             28.00
25%               175.00          0.00        800.00            101.00
50%               203.00        802.00        852.00            101.00
75%               320.00       1488.00       1500.00            103.00
max               438.00       1972.00       1986.00            109.00
std               100.84        650.70        524.97             10.35

       RS_E_WatTemp_PC2  RS_T_OilTemp_PC1  RS_T_OilTemp_PC2   month
count           1986.00           1986.00           1986.00 1986.00
mean              86.15             98.08             92.50    6.38
min                0.00             38.00             30.00    1.00
25%               82.00             96.00             89.00    5.00
50%               85.00            100.00             93.00    6.00
75%               95.00            103.00             99.00    8.00
```

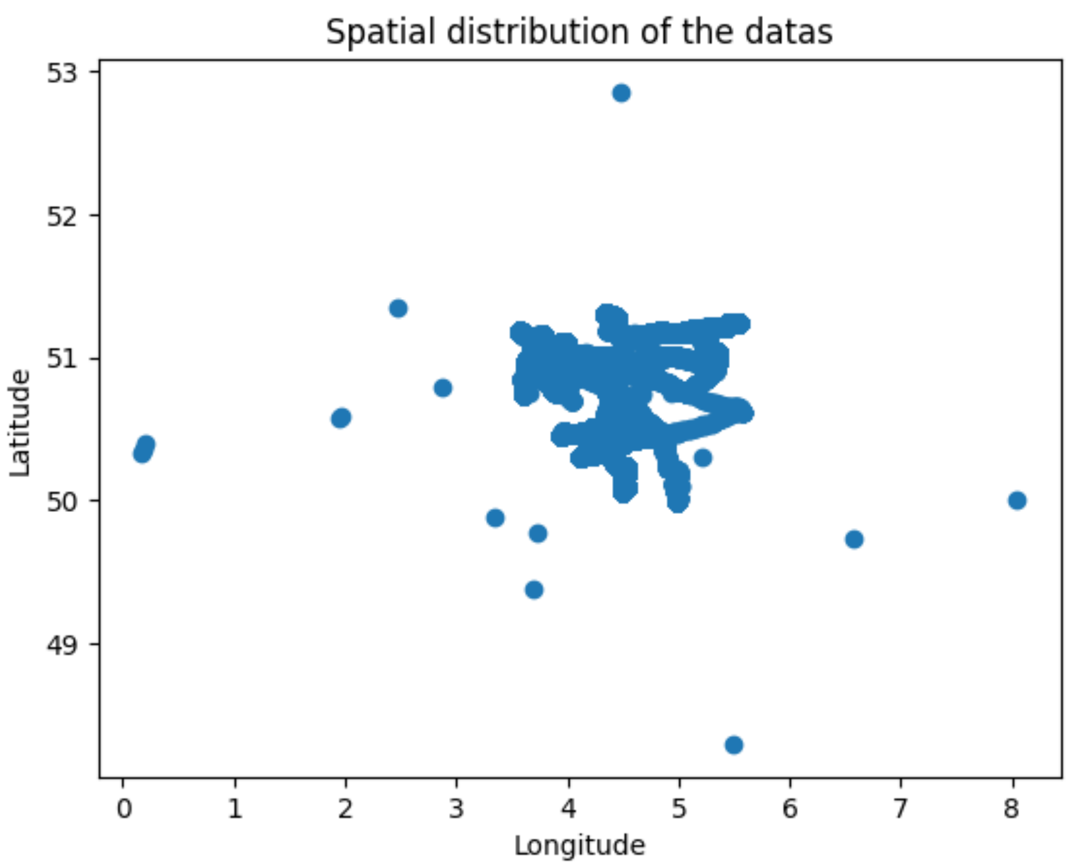
```
max                 119.00              119.00              117.00     9.00
std                  16.15                9.54               12.12     1.43
```

Processing the geolocation

With access to latitude and longitude information for the observations, we aimed to explore the geographical aspects of the data, seeking potential correlations between physical location and anomalies. The graph below illustrates the spatial distribution of the data, and a concentration is noticeable in what we confidently identify as Belgium. The remaining dots may represent anomalies.

For transparency, the code attempting to extract town information from the data is provided below. However, this attempt faced limitations due to API constraints.

```
import matplotlib.pyplot as plt

plt.scatter(data['lon'], data['lat'])
plt.title('Spatial distribution of the data')
plt.xlabel('Longitude')
plt.ylabel('Latitude')
plt.show()
```




```
from geopy.geocoders import options

options.default_user_agent = "dataMining-Project-ULB-2023"


# Specify your custom user agent
custom_user_agent = "dataMining-Project-ULB-2023"
```

```python
# Initialize the geolocator
geolocator = Nominatim(user_agent=custom_user_agent)

import pandas as pd
from geopy.exc import GeocoderTimedOut
from time import sleep
from opencage.geocoder import OpenCageGeocode


data['lat'] = data['lat'].round(4)
data['lon'] = data['lon'].round(4)

data['city'] = ""
unique_pairs = data[['lat', 'lon']].round(5).drop_duplicates()

print("\nUnique pairs of lat lon :")
print(len(unique_pairs))

opencage_api_key = "f7788d09d2d44156bf25d4f057808a9b"
geocoder = OpenCageGeocode(opencage_api_key)

# Set the batch size and delay between requests
batch_size = 100
delay_seconds = 1
results = []
# Function to geocode a batch of locations
def geocode_batch(batch):
    for index, row in batch.iterrows():
        location = f"{row['lat']}, {row['lon']}"
        try:
            result = geocoder.reverse_geocode(row['lat'], row['lon'])
            if(result[0]["components"]["town"]):
                results.append(result[0]["components"]["town"])
            elif(result[0]["components"]["city"]):
                results.append(row['lat'],row['lon'],result[0]["components"]["city
"])
        except (GeocoderTimedOut, Exception) as e:
            print(f"Error geocoding {location}: {e}")
            results.append(None)
        sleep(delay_seconds)  # Throttle requests
    return results

# Split the DataFrame into batches
for i in range(0, len(unique_pairs), batch_size):
    batch = unique_pairs.iloc[i:i + batch_size]
    # Geocode the batch
    batch_results = geocode_batch(batch)

df = pd.DataFrame(results, columns=['lat', 'lon', 'city'])

csv_file_path = 'cities.csv'

df.to_csv(csv_file_path, index=False)

print(f"Le tableau a été sauvegardé au format CSV dans : {csv_file_path}")
```

```python
from geocode.geocode import Geocode

gc = Geocode()
gc.load()  # load geonames data

mydata = ['Tel Aviv', 'Mangalore 🇮🇳']

for input_text in mydata:
    locations = gc.decode(50.9922,3.6413)
    print(locations)

# Create the 'city' column with initial values
data['city'] = ""


from geopy.geocoders import options

options.default_user_agent = "dataMining-Project-ULB-2023"


# Specify your custom user agent
custom_user_agent = "dataMining-Project-ULB-2023"
# Initialize the geolocator
geolocator = Nominatim(user_agent=custom_user_agent)

# Define the function to update the 'city' column for a batch of rows
def update_city_batch(batch):
    for index, row in batch.iterrows():
        location = geolocator.reverse((row['lat'], row['lon']), language="en")
        if location and 'address' in location.raw:
            data.at[index, 'city'] = location.raw['address'].get('city', '')

# Define the batch size
batch_size = 1000

# Process the DataFrame in batches
for i in range(0, len(data), batch_size):
    batch = data.iloc[i:i+batch_size]
    update_city_batch(batch)
```

Preprocessing

Preprocessing is a crucial step in refining raw data before further analysis or modeling. Common tasks include handling missing values, cleaning data inconsistencies, normalizing numerical features, encoding categorical variables, and other transformations. These steps are essential to ensure that subsequent analyses and models are built on accurate and well-structured data.

```python
import pandas as pd
import matplotlib.pyplot as plt
import numpy as np
import csv
```

```
data = pd.read_csv('../ar41_for_ulb.csv', sep=';')
data = data.rename(columns={'Unnamed: 0' : 'ID'})
```

We begin with renaming and sorting by trains ascending and chronological order.

```
# data = data.rename(columns={'Unnamed: 0': 'ID'})
data['timestamps_UTC'] = pd.to_datetime(data['timestamps_UTC'])
data = data.sort_values(by='timestamps_UTC')
data['month'] = data['timestamps_UTC'].dt.month
data.sort_values(by="mapped_veh_id", ascending=True)
```

```
          Unnamed: 0  mapped_veh_id      timestamps_UTC        lat       lon  \
2087711      2087711          102.0 2023-07-31 10:22:30  51.013086  3.780829
8226243      8226243          102.0 2023-02-03 12:43:59  51.015870  3.774773
14651287    14651287          102.0 2023-02-03 12:43:56  51.015658  3.775543
5466267      5466267          102.0 2023-02-03 12:43:46  51.015676  3.775517
5423322      5423322          102.0 2023-02-03 12:42:59  51.015883  3.774768
...              ...            ...                 ...        ...       ...
1107266      1107266          197.0 2023-06-29 09:30:39  50.419863  4.535626
12813319    12813319          197.0 2023-02-02 13:09:41  50.419114  4.533986
12657550    12657550          197.0 2023-02-02 13:09:51  50.418820  4.533492
10669886    10669886          197.0 2023-08-16 23:26:31  50.418918  4.533207
12321342    12321342          197.0 2023-03-13 19:35:23  50.417397  4.529748

          RS_E_InAirTemp_PC1  RS_E_InAirTemp_PC2  RS_E_OilPress_PC1  \
2087711                 26.0                26.0                0.0
8226243                 10.0                 9.5               69.0
14651287                 0.0                19.0                0.0
5466267                 20.0                18.0              224.0
5423322                 20.0                20.0              227.0
...                      ...                 ...                ...
1107266                 52.0                51.0                3.0
12813319                25.0                20.0              224.0
12657550                25.0                20.0              224.0
10669886                29.0                30.0              224.0
12321342                30.0                24.0              234.0

          RS_E_OilPress_PC2  RS_E_RPM_PC1  RS_E_RPM_PC2  RS_E_WatTemp_PC1  \
2087711                 0.0           0.0           0.0              27.0
8226243                55.0         164.0         106.5              40.0
14651287              110.0           0.0         147.0              80.0
5466267               231.0         797.0         788.0              80.0
5423322               241.0         797.0         799.0              80.0
...                     ...           ...           ...               ...
1107266                 0.0           0.0           0.0              81.0
12813319              372.0         795.0         797.0              77.0
12657550              379.0         799.0         804.0              77.0
10669886              369.0         802.0         794.0              78.0
12321342              345.0         870.0         880.0              78.0

          RS_E_WatTemp_PC2  RS_T_OilTemp_PC1  RS_T_OilTemp_PC2  month
2087711               31.0              18.0              22.0      7
8226243               39.5              75.5              76.5      2
14651287              79.0              75.0              77.0      2
5466267               79.0              74.0              77.0      2
5423322               78.0              76.0              77.0      2
...                    ...               ...               ...    ...
```

```
1107266                81.0              79.0              82.0      6
12813319               50.0              73.0              50.0      2
12657550               50.0              71.0              50.0      2
10669886               56.0              70.0              52.0      8
12321342               67.0              77.0              70.0      3

[17679273 rows x 16 columns]
```

Duplicate entries can potentially compromise the accuracy of predictions. Through this analysis, we can affirm that there are no duplicate records in the dataset.

```
print(f"Len before duplicates : {len(data)}")

# Drop duplicates
data = data.drop_duplicates()

print(f"Len after duplicates : {len(data)}")
```

```
Len before duplicates : 17679273

Len after duplicates : 17679273
```

NaNs treatment

NaNs (Not a Number) treatment is a crucial step in data preprocessing. It involves handling missing or undefined values in the dataset. The goal is to ensure the quality and reliability of the data, as missing values can introduce bias and affect the performance of machine learning models.

```
print(f'Number of nan in dataset : {data.isna().sum()}')
```

```
Number of nan in dataset : ID                          0
mapped_veh_id           0
timestamps_UTC          0
lat                     0
lon                     0
RS_E_InAirTemp_PC1      0
RS_E_InAirTemp_PC2  12726
RS_E_OilPress_PC1       0
RS_E_OilPress_PC2   12726
RS_E_RPM_PC1            0
RS_E_RPM_PC2        12726
RS_E_WatTemp_PC1        0
RS_E_WatTemp_PC2    12726
RS_T_OilTemp_PC1        0
RS_T_OilTemp_PC2    12726
month                   0
dtype: int64
```

While the occurrence of NaN values is not extensive, addressing them is essential to prevent potential issues and ensure the robustness of the analysis.

We decide now to focus on a particular train, Train 129. We will examine its NaN values to determine whether they are isolated incidents or indicative of equipment anomalies.

```
train_number = '129'
data_129 = pd.read_csv(f'sncb_with_weather_moving/train_data_{train_number}.0.csv'
, delimiter=';')
```

```
data_129 = data_129.drop(columns=['temp','feels_like','pressure','humidity','cloud
s','wind_speed','wind_deg','weather','moving'])
data_129.loc[:,'timestamps_UTC'] = pd.to_datetime(data_129['timestamps_UTC'])
data_129 = data_129.sort_values(by='timestamps_UTC')
nan_counts = data_129.isna().sum()
print("Number of NaN values in each column:")
print(nan_counts)
```

```
Number of NaN values in each column:
Unnamed: 0              0
mapped_veh_id           0
timestamps_UTC          0
lat                     0
lon                     0
RS_E_InAirTemp_PC1      0
RS_E_InAirTemp_PC2   1585
RS_E_OilPress_PC1       0
RS_E_OilPress_PC2    1585
RS_E_RPM_PC1            0
RS_E_RPM_PC2         1585
RS_E_WatTemp_PC1        0
RS_E_WatTemp_PC2     1585
RS_T_OilTemp_PC1        0
RS_T_OilTemp_PC2     1585
dtype: int64
```

Train 129 exhibits a notable presence of NaN values, constituting 12.5% of all NaN values in our dataset.

With the following function we examine all NaN values in the dataset.

As observed earlier, all NaN occurrences are associated with engine PC2. If the subsequent data point is within a 30-minute interval, it is considered a minor issue, possibly a small glitch or a delay in equipment startup. In such cases, we replace the NaN value with the corresponding value from PC1. However, if there is no data point with a value within a 30-minute window following the NaN value, it is identified as an anomaly.

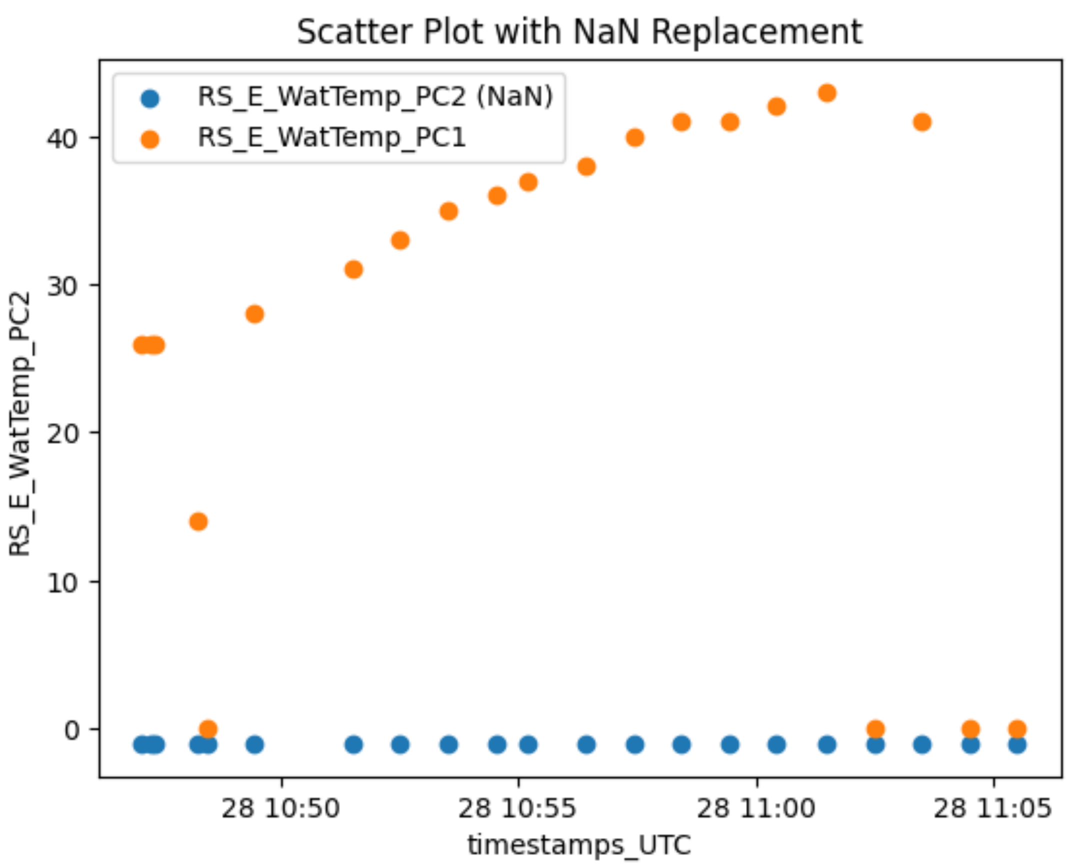
```
from datetime import timedelta
import json

def process_dataframe(df):
    df = df.reset_index(drop=True)
    new_data = pd.DataFrame(columns=df.columns)
    anomaly = []

    pc2_column = 'RS_E_InAirTemp_PC2'
    pc1_column = 'RS_E_InAirTemp_PC1'
    for index, value in df[pc2_column].items():
        if pd.isna(value):
            next_index = index + 1
            while next_index < len(df) and pd.isna(df[pc2_column][next_index]):
                next_index += 1

            if next_index < len(df):
                time_difference = df['timestamps_UTC'][next_index] - df['timestamp
s_UTC'][index]
                if time_difference <= timedelta(minutes=30):
```

```
                    df.at[index, pc2_column] = df.at[next_index, pc1_column]
                    df.at[index, 'RS_T_OilTemp_PC2'] = df.at[next_index, 'RS_T_Oil
Temp_PC1']
                    df.at[index, 'RS_E_WatTemp_PC2'] = df.at[next_index, 'RS_E_Wat
Temp_PC1']
                    df.at[index, 'RS_E_RPM_PC2'] = df.at[next_index, 'RS_E_RPM_PC1
']
                    df.at[index, 'RS_E_OilPress_PC2'] = df.at[next_index, 'RS_E_Oi
lPress_PC1']
                else:
                    anomaly.append(df.iloc[max(index-10,0):index+11])
                    new_data = pd.concat([new_data, df.loc[[index]]])
                    df.at[index, pc2_column] = None
                    df.at[index, 'RS_T_OilTemp_PC2'] = None
                    df.at[index, 'RS_E_WatTemp_PC2'] = None
                    df.at[index, 'RS_E_RPM_PC2'] =None
                    df.at[index, 'RS_E_OilPress_PC2'] = None
            else:
                anomaly.append(df.iloc[max(index-10,0):index+11])
                new_data = pd.concat([new_data, df.loc[[index]]])
                df.at[index, pc2_column] = None
                df.at[index, 'RS_T_OilTemp_PC2'] = None
                df.at[index, 'RS_E_WatTemp_PC2'] = None
                df.at[index, 'RS_E_RPM_PC2'] =None
                df.at[index, 'RS_E_OilPress_PC2'] = None
    return df, anomaly, new_data

data_processed, json_data, anomaly_df = process_dataframe(data_129)
nan_counts = data_processed.isna().sum()
print("Number of NaN values in each column:")
print(nan_counts)
```

```
6.0

C:\Users\maxim\AppData\Local\Temp\ipykernel_20340\3568256177.py:28: FutureWarning:
The behavior of DataFrame concatenation with empty or all-NA entries is deprecated
. In a future version, this will no longer exclude empty or all-NA columns when de
termining the result dtypes. To retain the old behavior, exclude the relevant entr
ies before the concat operation.
  new_data = pd.concat([new_data, df.loc[[index]]])

Number of NaN values in each column:
Unnamed: 0              0
mapped_veh_id           0
timestamps_UTC          0
lat                     0
lon                     0
RS_E_InAirTemp_PC1      0
RS_E_InAirTemp_PC2   1563
RS_E_OilPress_PC1       0
RS_E_OilPress_PC2    1563
RS_E_RPM_PC1            0
RS_E_RPM_PC2         1563
RS_E_WatTemp_PC1        0
RS_E_WatTemp_PC2     1563
RS_T_OilTemp_PC1        0
RS_T_OilTemp_PC2     1563
dtype: int64
```

Here, the algorithm successfully addressed a small number of NaN values. Now, let's examine a specific instance where an anomaly was detected.

```
#The eleventh anomaly for instance. This has 10 points before and 10 after
selected_anomaly = json_data[10]

selected_anomaly = selected_anomaly.fillna(-1)
plt.scatter(selected_anomaly['timestamps_UTC'],selected_anomaly['RS_E_WatTemp_PC2'
], label='RS_E_WatTemp_PC2 (NaN)')
plt.scatter(selected_anomaly['timestamps_UTC'],selected_anomaly['RS_E_WatTemp_PC1'
],label='RS_E_WatTemp_PC1 ')

plt.title('Scatter Plot with NaN Replacement')
plt.xlabel('timestamps_UTC')
plt.ylabel('RS_E_WatTemp_PC2')
plt.legend()
plt.show()
```



In the graph above, the blue dots represent PC2, while the orange dots represent PC1. Although we observe values for PC1, PC2 remains NaN for an extended period. To visualize the NaN values, they have been set to -1. The continuous occurrence of NaN values over an extended time period from the sensors implies an anomaly.

We can now apply this to all the datasets.

```
data['mapped_veh_id'] = data['mapped_veh_id'].astype(int)
result_dfs = []
anomalies = []
```

```python
removed_data = pd.DataFrame(columns=data.columns)

for train_number, group_df in data.groupby('mapped_veh_id'):
    result_df, anomaly, new_data = process_dataframe(group_df)
    print(len(anomaly))
    result_dfs.append(result_df)
    anomalies.extend(anomaly)
    removed_data = pd.concat([removed_data, new_data])
    anomaly_as_list_of_dicts = [slice_df.to_dict(orient='records') for slice_df in
anomaly]


    with open(f'instrument_anomalies/instrument_nan_anomaly_{train_number}.json',
'w') as file:
        json.dump({"content":anomaly_as_list_of_dicts}, file, default=str)


removed_data.to_excel("removed_data_malfunction.xlsx", index=False)
merged_df = pd.concat(result_dfs, ignore_index=True)

nan_counts = data.isna().sum()
print("Number of NaN values in each column before processing:")
print(nan_counts)
nan_counts = merged_df.isna().sum()
print("Number of NaN values in each column:")
print(nan_counts)
all_data_processed = merged_df.dropna()
nan_counts = all_data_processed.isna().sum()
print("Number of NaN values in each column:")
print(nan_counts)
```

```
18.0
8.0
17.0
6.0

41.0
29.0
40.0

40.0
40.0
26.0


28.0
0.0

21.0
8.0
28.0

38.0
53.0
52.0
36.0
```

31.0
36.0
10.0
36.0

23.0
7.0
6.0

3.0
40.0
8.0
20.0

41.0
35.0

26.0
36.0
41.0
40.0
46.0
30.0

38.0
6.0
27.0
13.0

17.0
38.0
16.0
23.0

25.0
9.0

13.0
32.0
26.0
35.0

6.0

54.0
26.0
0.0

17.0
15.0
0.0
10.0

17.0
8.0
17.0
8.0

7.0
0.0

```
26.0
52.0

43.0
42.0
22.0
11.0

15.0
17.0
29.0

44.0
11.0
9.0
17.0

61.0
6.0
0.0

18.0

41.0
0.0
17.0
20.0

20.0
28.0
0.0
36.0
```

Look at the unwanted data

In this section, we are concerned with removing the data that we do not need for model construction. We observed that some lines in the .csv file were dated 2022, which is outside our working scope, as the subject specifies a timeframe from January to September 2023. We also eliminated duplicate entries. Additionally, we converted the 'temp' and 'feels_like' columns to degrees to ensure consistency with the 'InAirTemp' columns, all in the same unit.

The next part focuses on generating a JSON file for each type of anomaly and for each train, resulting in three JSON files per train. Here, we are specifically addressing anomalies related to the thresholds specified in the subject.

```python
# Check the temporal extent of the data
print("Min Timestamp:", data['timestamps_UTC'].min())
print("Max Timestamp:", data['timestamps_UTC'].max())
```

```
Running cells with 'c:\Users\jibri\AppData\Local\Microsoft\WindowsApps\python3.11.
exe' requires the ipykernel package.

Run the following command to install 'ipykernel' into the Python environment.

Command: 'c:/Users/jibri/AppData/Local/Microsoft/WindowsApps/python3.11.exe -m pip
install ipykernel -U --user --force-reinstall'
```

```
data = data[data['timestamps_UTC'] >= '2023-01-01']
len(data)

#Automatisation of the process
import os
import json
csv_folder_path = '../sncb_data_v4/'
for filename in os.listdir(csv_folder_path):
    if filename.endswith(".csv"):
        file_id = filename.split('_')[2].split('.')[0]
        data = pd.read_csv(os.path.join(csv_folder_path, filename), sep=';')
        data = data.rename(columns={'Unnamed: 0': 'ID'})

        data['timestamps_UTC'] = pd.to_datetime(data['timestamps_UTC'])
        data = data.sort_values(by='timestamps_UTC')
        data['month'] = data['timestamps_UTC'].dt.month
        #data.sort_values(by="mapped_veh_id", ascending=True)
        data = data.sort_values(by='timestamps_UTC')
        print(f"Len before duplicates : {len(data)}")
        # Drop duplicates
        data = data.drop_duplicates(subset=data.columns.difference(['ID']))
        data = data.reset_index(drop=True)
        print(f"Len after duplicates : {len(data)}")

        # Check the temporal extent of the data
        print("Min Timestamp:", data['timestamps_UTC'].min())
        print("Max Timestamp:", data['timestamps_UTC'].max())

        data = data[data['timestamps_UTC'] >= '2023-01-01']
        len(data)

        result_df, anomaly, new_data = process_dataframe(data)
        data = result_df.dropna()
        data = data.reset_index(drop=True)
        data['temp'] = round(data['temp'] - 273.15, 2)
        data['feels_like'] = round(data['feels_like'] - 273.15, 2)

        data = data.loc[:, ~data.columns.duplicated()]
        new_data = pd.DataFrame(columns=data.columns)
        data = data.reset_index(drop=True)

        #inAir
        filtered_data_InAirAnomaly = data[(data['RS_E_InAirTemp_PC1'] > 65) | (dat
a['RS_E_InAirTemp_PC2'] > 65)]
        array_json = []
        anomaly_as_list_of_dicts = []

        for index, row in filtered_data_InAirAnomaly.iterrows():
            current_id = row['ID']
            current_index = data[data['ID'] == current_id].index[0]
            indices_to_include = data.iloc[max(current_index-10,0):current_index+1
1]
            anomaly_as_list_of_dicts = indices_to_include.to_dict(orient='records'
)
            array_json.append(anomaly_as_list_of_dicts)
```

```
        for record_list in array_json:
            for record in record_list:
                record['timestamps_UTC'] = record['timestamps_UTC'].strftime('%Y-%
m-%d %H:%M:%S')


        json_data_InAirAnomaly = json.dumps({"content": array_json}, indent=2)

        with open(f'../JSON/InAirTempAnomaly/InAirTempAnomaly_{file_id}.json', 'w'
) as json_file:
             json_file.write(json_data_InAirAnomaly)



        #OilTemp
        filtered_data_OilTempAnomaly = data[(data['RS_T_OilTemp_PC1'] > 115) | (da
ta['RS_T_OilTemp_PC2'] > 115) | ((data['RS_T_OilTemp_PC2'] < data['temp'] - 3) & (
data['RS_T_OilTemp_PC1'] < data['temp'] - 3))]
        array_json = []
        anomaly_as_list_of_dicts = []
        for index in filtered_data_OilTempAnomaly.index:
            indices_to_include = data.iloc[max(index-10,0):index+11]
            anomaly_as_list_of_dicts = indices_to_include.to_dict(orient='records'
)
            array_json.append(anomaly_as_list_of_dicts)

        for record_list in array_json:
            for record in record_list:
                record['timestamps_UTC'] = record['timestamps_UTC'].strftime('%Y-%
m-%d %H:%M:%S')




        json_data_OilTempAnomaly = json.dumps({"content": array_json}, indent=2)

        with open(f'../JSON/OilTempAnomaly/OilTempAnomaly_{file_id}.json', 'w') as
json_file:
            json_file.write(json_data_OilTempAnomaly)


        #WatTemp
        filtered_data_WatTempAnomaly = data[(data['RS_E_WatTemp_PC1'] > 100) | (da
ta['RS_E_WatTemp_PC2'] > 100)]
        array_json = []
        anomaly_as_list_of_dicts = []
        for index in filtered_data_WatTempAnomaly.index:
            indices_to_include = data.iloc[max(index-10,0):index+11]
            anomaly_as_list_of_dicts = indices_to_include.to_dict(orient='records'
)
            array_json.append(anomaly_as_list_of_dicts)

        for record_list in array_json:
            for record in record_list:
                record['timestamps_UTC'] = record['timestamps_UTC'].strftime('%Y-%
```

```
m-%d %H:%M:%S')



        json_data_WatTempAnomaly = json.dumps({"content": array_json}, indent=2)

        with open(f'../JSON/WatTempAnomaly/WatTempAnomaly_{file_id}.json', 'w') as
json_file:
            json_file.write(json_data_WatTempAnomaly)

        #now we preprocess we save the CSV
        data.to_csv(f"../sncb_data_v4/train_data_{file_id}.0.csv")
```

Weather data

From the OpenWeather API, we can get the current weather data for a given coordinate. The data is returned in JSON format. We can use the requests library to make a GET request to the API. The API requires an API key, which we can get by signing up for a free account at https://openweathermap.org/api. The API key is passed in the query string.

First, we converted the timestamps into unix time format because the API requires the timestamp to be in unix time format. Then we rounded the latitude and longitude to 2 decimal places, because the API only accepts 2 decimal places for the latitude and longitude. 17 million requests would be too costly, so we grouped the data by latitude and longitude, and only made one request per group. We got the minimum and the maximum of the timestamps in each group, so we could query the API for the weather data for the whole time period.

Then we created the dataframe from the received json data, and finally merged it with the train data. We also saved the dataframes into csv files, and each csv contained only one train's data.

Movement data

From the latitude, longitude and time data we can try to predict if the train is moving or not. If the difference in either the latitude or the longitude is greater than 0.00001, then we can say that the train is moving. We also created a new column, called "moving", which is 1 if the train is moving, and 0 if it is not moving. This difference translates to about a few meters, the time difference is usually around a minute. So, if the train doesn't move more than a few meters in a minute, then we can say that it is not moving.

Anomaly detection methods

In this section, we explore various anomaly detection methods to discern their efficacy. We meticulously evaluate the Isolation Forest, Support Vector Machine (SVM), k-Nearest Neighbors (KNN), DBSCAN, and Variational Autoencoder (VAE) models. This comparative analysis aims to identify the most effective model and assess their respective performances in detecting anomalies within our dataset.

Training, validation and testing sets

In the pursuit of developing robust models capable of identifying anomalies in train data, a crucial preliminary step involves the division of the dataset into training, validation, and test sets. Each of these subsets serves a distinct purpose in enhancing both the training process and the subsequent evaluation of the model's anomaly detection capabilities.

By strategically dividing the data, we create designated sets that contribute to different aspects of model development:

- **Training Set:** Comprising 18 anomaly-infused trains and 41 normal trains, this subset forms the backbone of the model's learning process. Enriched with a mix of anomaly-laden and normal instances, the training set allows the model to grasp the intricacies of both typical and atypical patterns, fostering a comprehensive understanding.
- **Validation Set:** To validate the model's performance effectively, we isolate 4 anomaly-laden trains and 10 normal trains. This subset serves as a safeguard against overfitting. By assessing the model's performance on a subset that it hasn't seen during training, we can fine-tune parameters and ensure the model's adaptability to unseen data, enhancing its generalization capabilities.
- **Testing Set:** We earmark 10 trains with anomalies, creating a robust ground for evaluating the model's anomaly detection performance. Additionally, 9 trains without anomalies are included to gauge the model's ability to correctly identify normal instances. The testing set provides an unbiased assessment of the model's effectiveness. By evaluating its performance on a set of entirely new instances, including both anomalies and normal instances, we gain insights into its real-world applicability.

The deliberate distribution of 22 anomaly-laden trains for training/validation purposes and the reservation of 10 for testing involve random allocation, aiming to strike a balance. This ensures the model encounters diverse anomaly scenarios during training while maintaining fairness and an unbiased evaluation on the testing set.

```
import random
import pandas as pd
import os

# Set the seed for reproducibility
random.seed(42)

# Set the working directory
directory = '/mnt/c/Users/jibri/OneDrive - INSA Lyon/Bureau/ULB/DataMingin/sncb_da
ta_v4/'
os.chdir(directory)

# IDs of trains with the most anomalies
numbers = [128, 114, 181, 191, 170, 117, 150, 177, 172, 154, 142, 151, 121, 161, 1
10, 126, 134, 146, 148, 160, 164, 166, 167, 175, 183, 187, 190, 192, 123, 125, 163
, 194]

# Complete number IDs of the trains
full_range = set(range(102, 198)) - {118, 132, 193, 195}

# Calculate the remaining numbers
remaining_numbers = full_range - set(numbers)

# Take 9 random numbers without replacement from the remaining numbers
testing_numbers = random.sample(list(remaining_numbers), 9)

filenames_test = [f'train_data_{num}.0.csv' for num in testing_numbers]

# Read dataframes from files
```

```
dataframes_test = [pd.read_csv(filename, sep=',', low_memory=False) for filename i
n filenames_test]

remaining_numbers = remaining_numbers - set(testing_numbers)

# Take 41 random numbers without replacement from the remaining numbers
training_numbers = random.sample(list(remaining_numbers), 41)

filenames_train = [f'train_data_{num}.0.csv' for num in training_numbers]

# Read dataframes from files
dataframes_train = [pd.read_csv(filename, sep=',', low_memory=False) for filename
in filenames_train]

remaining_numbers = remaining_numbers - set(training_numbers)

# Remaining numbers are now used for validation
filenames_val = [f'train_data_{num}.0.csv' for num in remaining_numbers]

# Read dataframes from files
dataframes_val = [pd.read_csv(filename, sep=',', low_memory=False) for filename in
filenames_val]

# Use the same seed for random.sample to get identical samples
random.seed(42)

test_numbers_anomalies = random.sample(list(numbers), 10)

filenames_test = [f'train_data_{num}.0.csv' for num in test_numbers_anomalies]

# Read dataframes from files
dataframes_test_anomalies = [pd.read_csv(filename, sep=',', low_memory=False) for
filename in filenames_test]

numbers = set(numbers) - set(test_numbers_anomalies)

train_numbers_anomalies = random.sample(list(numbers), 18)

filenames_train = [f'train_data_{num}.0.csv' for num in train_numbers_anomalies]

# Read dataframes from files
dataframes_train_anomalies = [pd.read_csv(filename, sep=',', low_memory=False) for
filename in filenames_train]

numbers = numbers - set(train_numbers_anomalies)

filenames_val = [f'train_data_{num}.0.csv' for num in numbers]

# Read dataframes from files
dataframes_val_anomalies = [pd.read_csv(filename, sep=',', low_memory=False) for f
ilename in filenames_val]
```

Having initially recorded datasets separately, the next step involves their concatenation.

```python
# Concatenate the dataframes

df_train = pd.concat(dataframes_train, ignore_index=True)
df_test = pd.concat(dataframes_test, ignore_index=True)
df_val = pd.concat(dataframes_val, ignore_index=True)
df_train_an = pd.concat(dataframes_train_anomalies, ignore_index=True)
df_test_an = pd.concat(dataframes_test_anomalies, ignore_index=True)
df_val_an = pd.concat(dataframes_val_anomalies, ignore_index=True)

# Create the final train, validation and test dataframes

df_train = pd.concat([df_train, df_train_an], ignore_index=True)
df_test = pd.concat([df_test, df_test_an], ignore_index=True)
df_val = pd.concat([df_val, df_val_an], ignore_index=True)

df_train.shape, df_test.shape, df_val.shape
```

```
((1015737, 26), (4254036, 26), (296350, 26))
```

```python
del dataframes_train, dataframes_test, dataframes_val, dataframes_train_anomalies,
dataframes_test_anomalies, dataframes_val_anomalies, df_train_an, df_test_an, df_v
al_an
```

Data types transormation

To enhance our analysis and facilitate time-based operations, we transform the 'timestamps' variable into the 'datetime' datatype. This adjustment will enable us to leverage temporal information effectively, ensuring accurate insights and a more comprehensive understanding of the dataset. We do this operation for all the datasets.

```python
# Dataframe info
print(df_train.info())
print(df_test.info())
print(df_val.info())
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 11359352 entries, 0 to 11359351
Data columns (total 26 columns):
 #   Column              Dtype
---  ------              -----
 0   Unnamed: 0          int64
 1   ID                  int64
 2   mapped_veh_id       float64
 3   timestamps_UTC      object
 4   lat                 float64
 5   lon                 float64
 6   RS_E_InAirTemp_PC1  float64
 7   RS_E_InAirTemp_PC2  float64
 8   RS_E_OilPress_PC1   float64
 9   RS_E_OilPress_PC2   float64
 10  RS_E_RPM_PC1        float64
 11  RS_E_RPM_PC2        float64
 12  RS_E_WatTemp_PC1    float64
 13  RS_E_WatTemp_PC2    float64
 14  RS_T_OilTemp_PC1    float64
 15  RS_T_OilTemp_PC2    float64
 16  temp                float64
 17  feels_like          float64
```

```
 18  pressure            float64
 19  humidity            float64
 20  clouds              float64
 21  wind_speed          float64
 22  wind_deg            float64
 23  weather             object
 24  moving              int64
 25  month               int64
dtypes: float64(20), int64(4), object(2)

[8 rows x 24 columns]
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3413055 entries, 0 to 3413054
Data columns (total 26 columns):
 #   Column              Dtype
---  ------              -----
 0   Unnamed: 0          int64
 1   ID                  int64
 2   mapped_veh_id       float64
 3   timestamps_UTC      object
 4   lat                 float64
 5   lon                 float64
 6   RS_E_InAirTemp_PC1  float64
 7   RS_E_InAirTemp_PC2  float64
 8   RS_E_OilPress_PC1   float64
 9   RS_E_OilPress_PC2   float64
 10  RS_E_RPM_PC1        float64
 11  RS_E_RPM_PC2        float64
 12  RS_E_WatTemp_PC1    float64
 13  RS_E_WatTemp_PC2    float64
 14  RS_T_OilTemp_PC1    float64
 15  RS_T_OilTemp_PC2    float64
 16  temp                float64
 17  feels_like          float64
 18  pressure            float64
 19  humidity            float64
 20  clouds              float64
 21  wind_speed          float64
 22  wind_deg            float64
 23  weather             object
 24  moving              int64
 25  month               int64
dtypes: float64(20), int64(4), object(2)


[8 rows x 24 columns]
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2900728 entries, 0 to 2900727
Data columns (total 26 columns):
 #   Column              Dtype
---  ------              -----
 0   Unnamed: 0          int64
 1   ID                  int64
 2   mapped_veh_id       float64
 3   timestamps_UTC      object
 4   lat                 float64
 5   lon                 float64
```

```
 6   RS_E_InAirTemp_PC1  float64
 7   RS_E_InAirTemp_PC2  float64
 8   RS_E_OilPress_PC1   float64
 9   RS_E_OilPress_PC2   float64
 10  RS_E_RPM_PC1        float64
 11  RS_E_RPM_PC2        float64
 12  RS_E_WatTemp_PC1    float64
 13  RS_E_WatTemp_PC2    float64
 14  RS_T_OilTemp_PC1    float64
 15  RS_T_OilTemp_PC2    float64
 16  temp                float64
 17  feels_like          float64
 18  pressure            float64
 19  humidity            float64
 20  clouds              float64
 21  wind_speed          float64
 22  wind_deg            float64
 23  weather             object
 24  moving              int64
 25  month               int64
dtypes: float64(20), int64(4), object(2)
memory usage: 575.4+ MB
None
[8 rows x 24 columns]
```

```
df_test['timestamps_UTC'] = pd.to_datetime(df_test['timestamps_UTC'])
df_val['timestamps_UTC'] = pd.to_datetime(df_val['timestamps_UTC'])
df_train['timestamps_UTC'] = pd.to_datetime(df_train['timestamps_UTC'])
```

One-hot encoding for categorical data:

Upon scrutinizing the data types in our datasets, it becomes evident that the 'weather' variable, despite its categorical nature, is currently labeled as an object. The variable 'month' is also categorical, but encoded as integer.

We therefore employ the one-hot encoding technique first on the training dataset, then, if necessary, on the other datasets. This transformation is essential because machine learning models work better with numerical data. By executing one-hot encoding on the 'weather' and 'month' variables, we ensure a seamless conversion of categorical values into a numerical format.

This process generates binary columns, each representing a distinct weather category. Consequently, each data point is characterized by a combination of 1s and 0s, conveying the presence or absence of specific weather conditions. This representation simplifies the incorporation of 'weather' details into our analytical and modeling pursuits. The same result is obtained for the variable 'month', where the combination of 1s and 0s indicates whether or not the observation was made in a particular month.

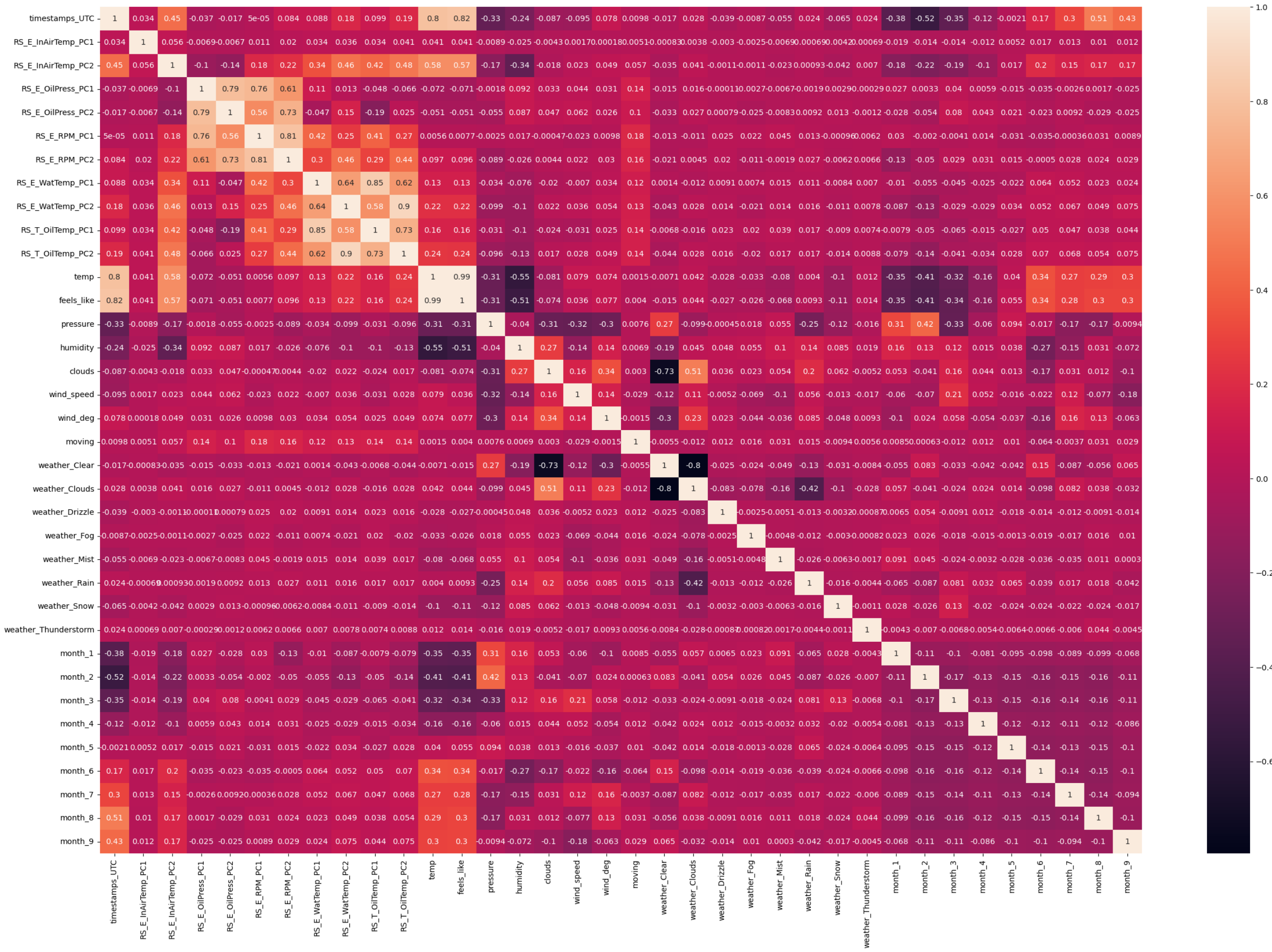
```
# One hot encoding

df_train = pd.get_dummies(df_train, columns=['weather'], prefix='weather')


# One hot encoding of month

df_train = pd.get_dummies(df_train, columns=['month'], prefix='month')
```

Correlation matrix

The correlation matrix serves as a valuable tool for investigating the relationships between variables, particularly for identifying weather features influencing train-related attributes.

This analysis is initially conducted on the training set. Subsequently, the same variables identified as influential are removed from the validation and test sets. This not only ensures consistency in feature selection across datasets but also holds significance in the context of large datasets. Eliminating redundant variables from the training set is crucial for expediting the model training process and focusing the model's attention on the truly impactful variables that influence train-related attributes.

```python
import matplotlib.pyplot as plt
import seaborn as sns

df_train_cor = df_train.drop(['ID', 'Unnamed: 0', 'mapped_veh_id', 'lat', 'lon'],
axis=1)

# create a correlation matrix
corr_matrix = df_train_cor.corr()

# set the size of the figure
plt.figure(figsize=(30, 20))

# visualize the correlation matrix
sns.heatmap(corr_matrix, annot=True)

# show the plot
plt.show()

del df_train_cor, corr_matrix
```



The correlation plot reveals that the weather features exhibit a relatively modest impact on the train-related attributes, suggesting that a substantial portion of these weather-related variables may not significantly contribute to the model. The same can be said for the variable months. Therefore, several of these weather and month features can be deemed less influential and subsequently removed. This elimination process aims to streamline the dataset, ensuring that the model focuses on the most pertinent variables, a step particularly crucial in optimizing the model's efficiency.

```
# Remove variables that may not affect the model

df_train = df_train.drop(['ID', 'mapped_veh_id', 'lat', 'lon', 'clouds', 'wind_speed',
                                           'wind_deg', 'weather_Clear', 'weather_Clouds', 'weather_Rain',
                                           'weather_Drizzle', 'weather_Fog', 'weather_Haze', 'weather_Mist', 'weather_Snow', 'weather_Thunderstorm',
                                           'month_1' , 'month_2',    'month_3',  'month_4', 'month_5',  'month_6',  'month_7',  'month_8',  'month_9'], axis=1)

df_test = df_test.drop(['ID', 'lat', 'lon', 'clouds', 'wind_speed',
                                           'wind_deg', 'weather', 'month'], axis=1)

df_val = df_val.drop(['ID',  'lat', 'lon', 'clouds', 'wind_speed',
                                           'wind_deg', 'weather', 'month'], axis=1)
```

Isolation forest

The Isolation Forest quickly detects anomalies by isolating them in a simplified decision tree. It does this by randomly selecting features and creating partitions, making anomalies stand out as they need fewer splits to be isolated. This method is efficient for high-dimensional datasets with large amounts of data.

We are currently generating a column identifying potential anomalies based on the available information. It is established that certain thresholds must not be exceeded or deviated from. Specifically, the InAirTemp is expected to remain below 65 degrees, the water temperature should not exceed 100 degrees and must not be zero, the oil temperature is constrained below 115 degrees and cannot be zero. Additionally, the oil pressure must not register as zero, and the RPMs must not be null. Furthermore, a potential sensor defect is indicated if the oil pressure is recorded as 690. We are applying this procedure for all the datasets, which is crucial for evaluating model accuracy in anomaly detection.

```
import numpy as np

X = df_train.iloc[:, 2:]
X_val = df_val.iloc[:, 3:]
X_test =df_test.iloc[:, 3:]

# Create a column 'TrueAnomaly' that identifies the observations that we assume to
be anomalies

X['TrueAnomaly'] = np.where(
    (X['RS_E_InAirTemp_PC1'] > 65) | (X['RS_E_InAirTemp_PC2'] > 65) | (X['RS_E_Wat
Temp_PC1'] > 100) |
    (X['RS_E_WatTemp_PC2'] > 100) | (X['RS_T_OilTemp_PC1'] > 115) | (X['RS_T_OilTe
mp_PC2'] > 115) |
    (X['RS_E_WatTemp_PC1'] == 0) | (X['RS_E_WatTemp_PC2'] == 0) | (X['RS_T_OilTemp
_PC1'] == 0) |
    (X['RS_T_OilTemp_PC2'] == 0) | (X['RS_E_OilPress_PC1'] == 0) | (X['RS_E_OilPre
ss_PC2'] == 0) |
    (X['RS_E_OilPress_PC1'] == 690) | (X['RS_E_OilPress_PC2'] == 690) |
    ((X['RS_E_RPM_PC1'] == 0) & (X['RS_E_RPM_PC2'] != 0)) | ((X['RS_E_RPM_PC2'] ==
0) & (X['RS_E_RPM_PC1'] != 0)),
    1, 0
)

# remove the column 'TrueAnomaly' from the train set

y_train = X['TrueAnomaly']
X_train = X.drop(['TrueAnomaly'], axis=1)

X_val['TrueAnomaly'] = np.where(
    (X_val['RS_E_InAirTemp_PC1'] > 65) | (X_val['RS_E_InAirTemp_PC2'] > 65) | (X_v
al['RS_E_WatTemp_PC1'] > 100) |
    (X_val['RS_E_WatTemp_PC2'] > 100) | (X_val['RS_T_OilTemp_PC1'] > 115) | (X_val
['RS_T_OilTemp_PC2'] > 115) |
    (X_val['RS_E_WatTemp_PC1'] == 0) | (X_val['RS_E_WatTemp_PC2'] == 0) | (X_val['
RS_T_OilTemp_PC1'] == 0) |
    (X_val['RS_T_OilTemp_PC2'] == 0) | (X_val['RS_E_OilPress_PC1'] == 0) | (X_val[
'RS_E_OilPress_PC2'] == 0) |
    (X_val['RS_E_OilPress_PC1'] == 690) | (X_val['RS_E_OilPress_PC2'] == 690) |
    ((X_val['RS_E_RPM_PC1'] == 0) & (X_val['RS_E_RPM_PC2'] != 0)) | ((X_val['RS_E_
```

```
RPM_PC2'] == 0) & (X_val['RS_E_RPM_PC1'] != 0)),
    1, 0
)

# remove the column 'TrueAnomaly' from the validation set

y_val = X_val['TrueAnomaly']
X_val = X_val.drop(['TrueAnomaly'], axis=1)

# remove the column 'TrueAnomaly' from the test set

X_test['TrueAnomaly'] = np.where(
    (X_test['RS_E_InAirTemp_PC1'] > 65) | (X_test['RS_E_InAirTemp_PC2'] > 65) | (X
_test['RS_E_WatTemp_PC1'] > 100) |
    (X_test['RS_E_WatTemp_PC2'] > 100) | (X_test['RS_T_OilTemp_PC1'] > 115) | (X_t
est['RS_T_OilTemp_PC2'] > 115) |
    (X_test['RS_E_WatTemp_PC1'] == 0) | (X_test['RS_E_WatTemp_PC2'] == 0) | (X_tes
t['RS_T_OilTemp_PC1'] == 0) |
    (X_test['RS_T_OilTemp_PC2'] == 0) | (X_test['RS_E_OilPress_PC1'] == 0) | (X_te
st['RS_E_OilPress_PC2'] == 0) |
    (X_test['RS_E_OilPress_PC1'] == 690) | (X_test['RS_E_OilPress_PC2'] == 690) |
    ((X_test['RS_E_RPM_PC1'] == 0) & (X_test['RS_E_RPM_PC2'] != 0)) | ((X_test['RS
_E_RPM_PC2'] == 0) & (X_test['RS_E_RPM_PC1'] != 0)),
    1, 0
)
y_test = X_test['TrueAnomaly']
X_test = X_test.drop(['TrueAnomaly'], axis=1)
```

Grid search is a crucial step in fine-tuning machine learning models to identify the optimal set of hyperparameters for enhanced performance. It involves systematically testing different combinations of hyperparameters to find the configuration that yields the best results.

In the subsequent code, we initiate a grid search for the best parameters using the Isolation Forest algorithm. The parameter under consideration is "contamination," representing the expected proportion of anomalies in the dataset. The grid search spans different contamination values, and for each value, the Isolation Forest model is trained and evaluated on the validation set.

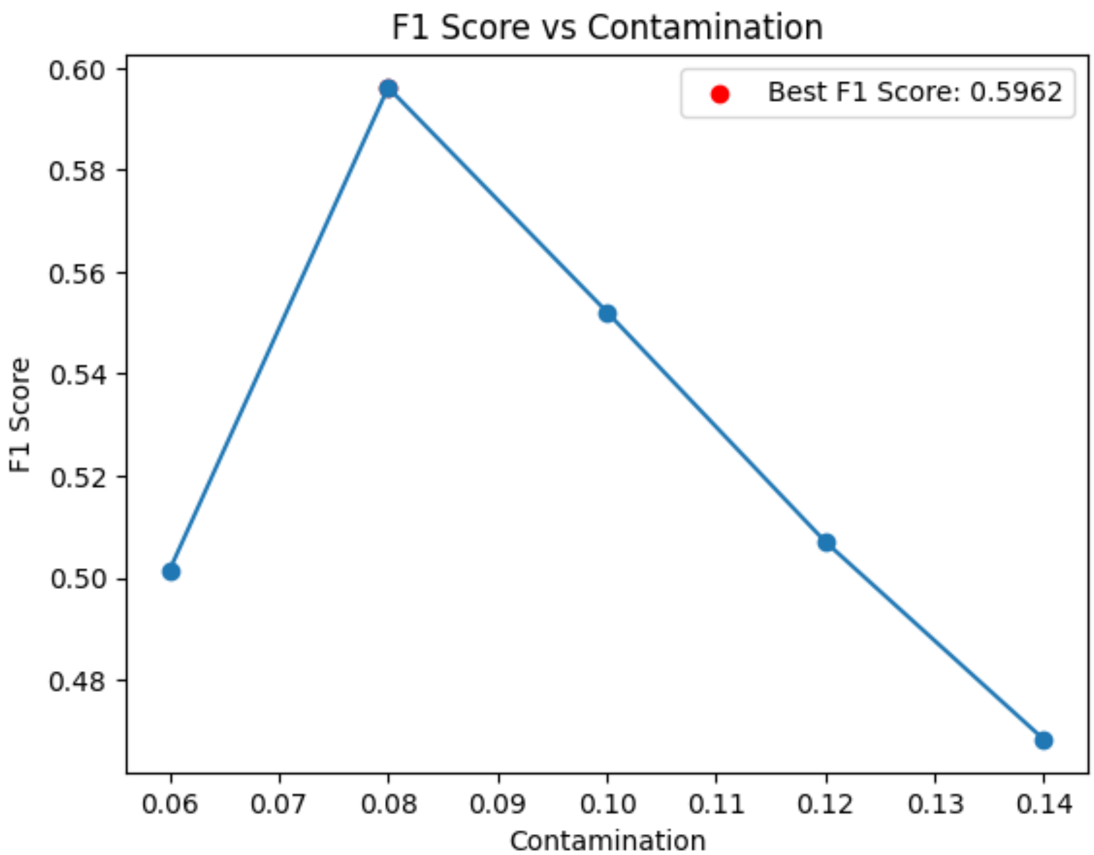
The goal is to find the contamination value that maximizes the F1 score, a metric that balances precision and recall, providing a comprehensive measure of the model's performance. The final step includes visualizing the F1 scores across contamination values to identify the optimal setting.

```
# Grid search for the best parameters
import numpy as np
from sklearn.ensemble import IsolationForest
from sklearn.metrics import precision_score, recall_score, f1_score
import matplotlib.pyplot as plt

contamination_values = [0.06, 0.08, 0.1, 0.12, 0.14]
f1_scores = []

for contamination_value in contamination_values:
    # Train the model with the current contamination value
    isolation_forest = IsolationForest(contamination=contamination_value, random_s
tate=42)
```

```python
    isolation_forest.fit(X_train)

    # Predict anomalies on the validation set
    anomaly_scores_val = isolation_forest.decision_function(X_val)
    predictions_val = isolation_forest.predict(X_val)

    # Compute evaluation metrics
    precision_val = precision_score(y_val, np.where(predictions_val == -1, 1, 0))
    recall_val = recall_score(y_val, np.where(predictions_val == -1, 1, 0))
    f1_val = f1_score(y_val, np.where(predictions_val == -1, 1, 0))
    f1_scores.append(f1_val)

    # Print metrics for the current contamination value
    print(f'Contamination: {contamination_value}')
    print(f'Precision (Validation): {precision_val:.4f}')
    print(f'Recall (Validation): {recall_val:.4f}')
    print(f'F1 Score (Validation): {f1_val:.4f}')
    print('\n')

# Identify the maximum F1 score
best_contamination = contamination_values[np.argmax(f1_scores)]
best_f1_score = max(f1_scores)

# Create the plot
plt.plot(contamination_values, f1_scores, marker='o')
plt.scatter(best_contamination, best_f1_score, color='red', label=f'Best F1 Score:
{best_f1_score:.4f}')
plt.xlabel('Contamination')
plt.ylabel('F1 Score')
plt.title('F1 Score vs Contamination')
plt.legend()
plt.show()
```

```
Contamination: 0.06
Precision (Validation): 0.4915
Recall (Validation): 0.5119
F1 Score (Validation): 0.5015


Contamination: 0.08
Precision (Validation): 0.4809
Recall (Validation): 0.7843
F1 Score (Validation): 0.5962


Contamination: 0.1
Precision (Validation): 0.4052
Recall (Validation): 0.8666
F1 Score (Validation): 0.5522


Contamination: 0.12
Precision (Validation): 0.3488
Recall (Validation): 0.9284
F1 Score (Validation): 0.5071
```

```
Contamination: 0.14
Precision (Validation): 0.3086
Recall (Validation): 0.9700
F1 Score (Validation): 0.4682
```



The previous plot reveals an increasing trend where the F1 score ascends slightly as the contamination parameter increases, until it reaches a certain point when it starts to decrease. As a result, the optimal contamination parameter identified through the grid search is situated at that value, which in this case is 0.08.

Testing

The optimal contamination parameter is now applied to predict anomalies on the test set. However, the outcomes reveal a comparatively lower performance compared to the validation set.

These findings, in conjunction with the earlier results, suggest that the Isolation Forest tends to exhibit higher precision than recall. Higher precision means that when the model flags an instance as an anomaly, it is more likely to be a genuine anomaly. However, this may come at the cost of missing some anomalies, leading to a lower recall rate.

```python
import numpy as np
from sklearn.ensemble import IsolationForest
from sklearn.metrics import precision_score, recall_score, f1_score

# Train the training test with the best parameter
best_model = IsolationForest(contamination=best_contamination)
```

```python
best_model.fit(X_train)

# Predict anomalies on the test set
anomaly_scores_test = best_model.decision_function(X_test)
predictions_test = best_model.predict(X_test)

# Add the anomaly scores and the predictions to the test set
X_test['AnomalyScore'] = anomaly_scores_test
X_test['IsAnomaly'] = np.where(predictions_test == -1, 1, 0)

# Evaluate the performance on the test set
precision_test = precision_score(y_test, X_test['IsAnomaly'])
recall_test = recall_score(y_test, X_test['IsAnomaly'])
f1_test = f1_score(y_test, X_test['IsAnomaly'])

# Print the metrics for the test set
print(f'Precision (Test): {precision_test:.4f}')
print(f'Recall (Test): {recall_test:.4f}')
print(f'F1 Score (Test): {f1_test:.4f}')
```

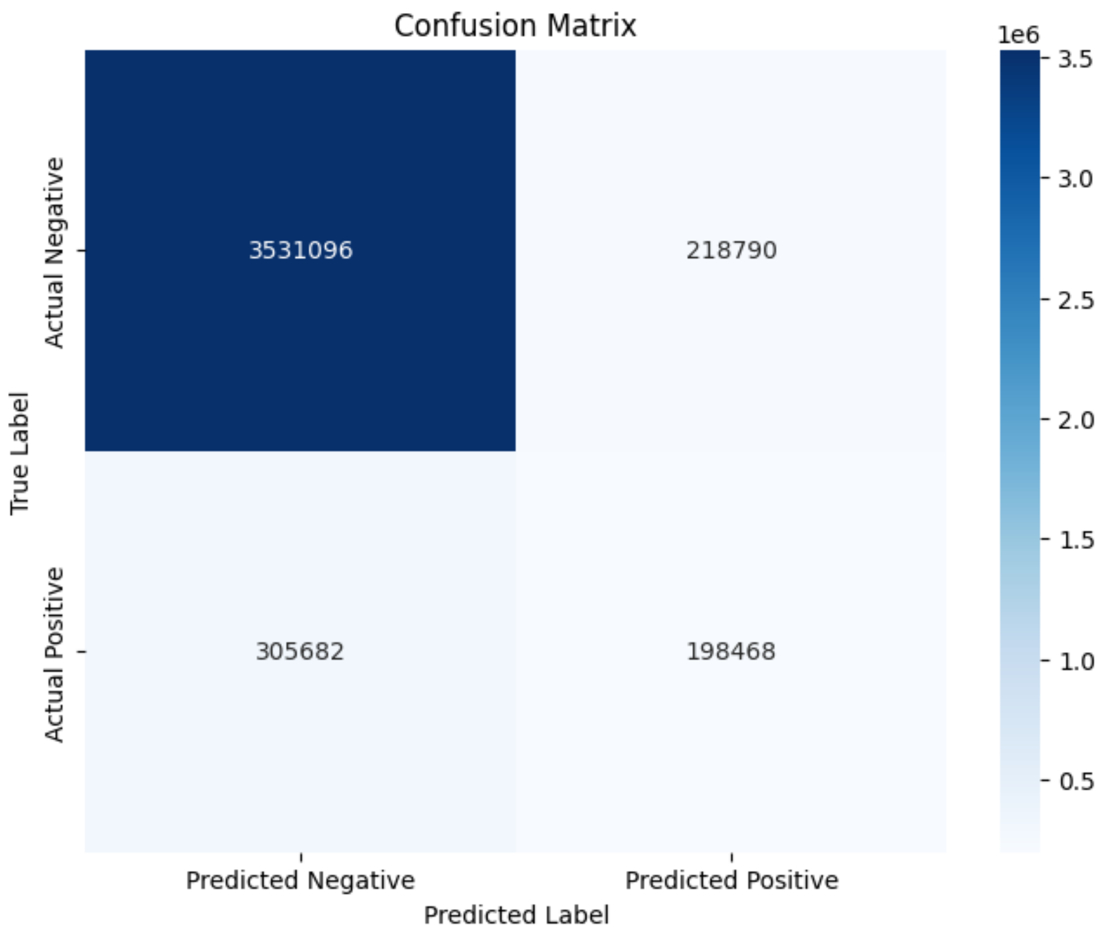
```
Precision (Test): 0.4756
Recall (Test): 0.3937
F1 Score (Test): 0.4308
```

We will now delve into the visualization of the confusion matrix to assess the model's performance on the testing data. This matrix provides a comprehensive view, detailing the counts of True Positives (correctly identified positives), True Negatives (correctly identified negatives), False Positives (incorrectly labeled as positives), and False Negatives (incorrectly labeled as negatives). Examining these metrics provides a comprehensive evaluation of the model's accuracy, precision, recall, and F1-score. This analysis offers valuable insights into the model's effectiveness, highlighting its strengths and areas for improvement in correctly identifying positive and negative instances.

```python
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix

# Count anomalies found by the model
count_an = X_test['IsAnomaly'].value_counts()

# Count anomalies in the test set
count_an_true = y_test.value_counts()

# See if there's correspondence between the anomalies found by the model and the a
nomalies in the test set
print('Anomalies found:', count_an)
print('Number of true anomalies', count_an_true)

# Compute la matrice di confusione
conf_matrix = confusion_matrix(y_test, X_test['IsAnomaly'])

# Extract TP, TN, FP, FN
TN, FP, FN, TP = conf_matrix.ravel()

print(f'TN: {TN}, FP: {FP}, FN: {FN}, TP: {TP}')
```

```python
# Create heatmap of confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
            xticklabels=['Predicted Negative', 'Predicted Positive'],
            yticklabels=['Actual Negative', 'Actual Positive'])
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.show()
```

```
Anomalies found: IsAnomaly
0    3836778
1     417258
Name: count, dtype: int64
Number of true anomalies TrueAnomaly
0    3749886
1     504150
Name: count, dtype: int64
TN: 3531096, FP: 218790, FN: 305682, TP: 198468
```

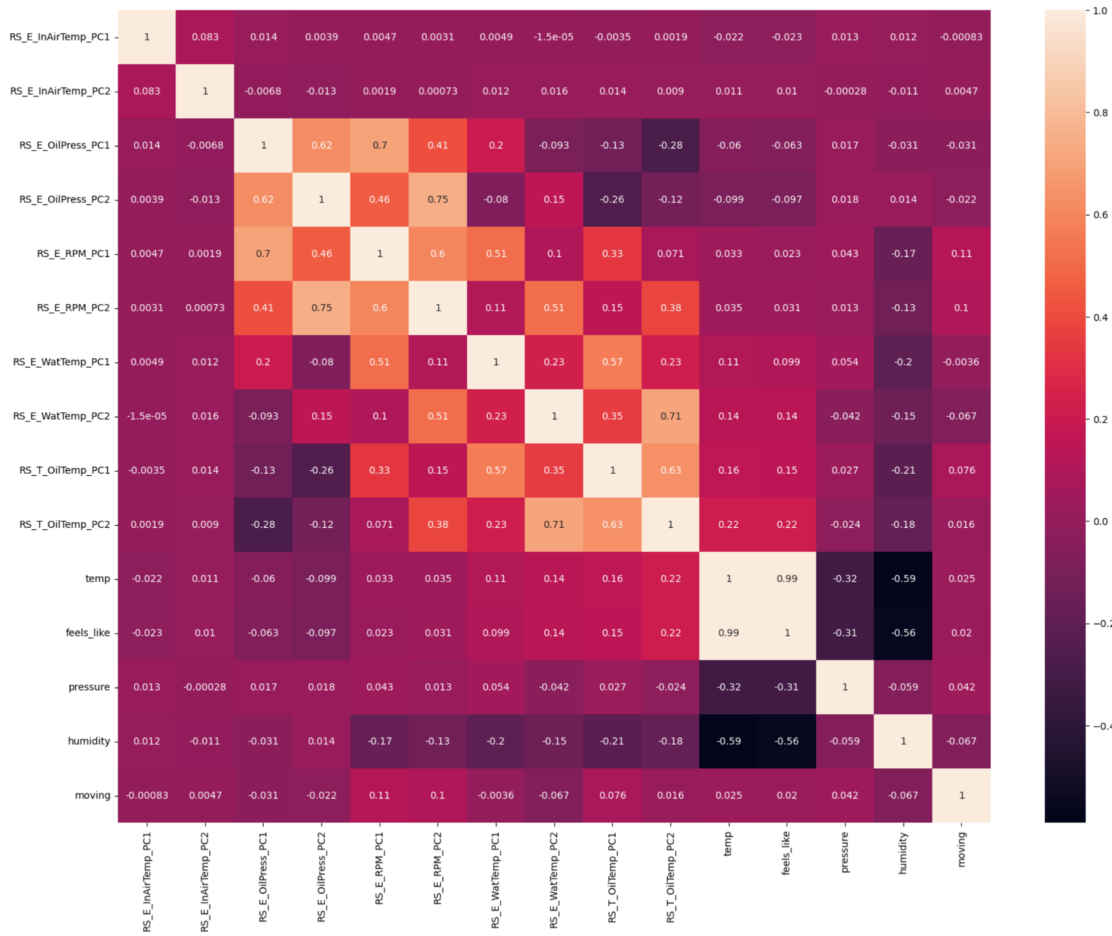


As expected, the confusion matrix doesn't exhibit favorable results, given the relatively low F1 score on the testing data. Consequently, we will explore alternative anomaly detection methods in pursuit of improved performance.

Influence of weather variables on the predicted anomalies

Despite the less promising outcomes in the predictions, we will continue our exploration by focusing on the impact of weather variables. This involves isolating the predicted anomalies and calculating the correlation matrix.

```python
import seaborn as sns
import matplotlib.pyplot as plt

X_test_anomalies = X_test[X_test['IsAnomaly'] == 1]

X_test_corr = X_test_anomalies.drop(['IsAnomaly', 'AnomalyScore'], axis=1)

# Correlation matrix for the anomalies
corr_matrix = X_test_corr.corr()

plt.figure(figsize=(20, 15))

sns.heatmap(corr_matrix, annot=True)

# show the plot
plt.show()

del X_test_corr, corr_matrix
```



By selecting a cutoff that we arbitrarely set at 0.15, we visualize more clearly the variables that influence the train variables.

```python
# List of PC1 motor variables
var_pc1 = [
    'RS_E_InAirTemp_PC1', 'RS_E_OilPress_PC1', 'RS_E_RPM_PC1', 'RS_E_WatTemp_PC1',
'RS_T_OilTemp_PC1'
]
```

```
# List of PC2 motor variables
var_pc2 = [
    'RS_E_InAirTemp_PC2', 'RS_E_OilPress_PC2', 'RS_E_RPM_PC2', 'RS_E_WatTemp_PC2',
'RS_T_OilTemp_PC2'
]

# List of all variables (motor and meteorological)
all_variables = var_pc1 + var_pc2 + [
    'temp', 'feels_like', 'pressure', 'humidity', 'moving'
]

corr_matrix = X_test_anomalies[all_variables].corr()

# Specify the cutoff for the absolute correlation value (e.g., 0.15)
correlation_cutoff = 0.15

meteorological_correlations_pc1 = corr_matrix[corr_matrix.index.isin(var_pc1)][['t
emp','feels_like', 'pressure', 'humidity', 'moving']]

meteorological_correlations_pc2 = corr_matrix[corr_matrix.index.isin(var_pc2)][['t
emp','feels_like', 'pressure', 'humidity', 'moving']]

# Print meteorological variables strongly correlated with engine PC1
if not meteorological_correlations_pc1.empty:
    print("Variables strongly correlated with engine PC1:")
    print(meteorological_correlations_pc1[
        (meteorological_correlations_pc1.abs() >= correlation_cutoff)
    ])
else:
    print("No strong correlations found for engine PC1.")

# Print meteorological variables strongly correlated with engine PC2
if not meteorological_correlations_pc2.empty:
    print("\nVariables strongly correlated with engine PC2:")
    print(meteorological_correlations_pc2[
        (meteorological_correlations_pc2.abs() >= correlation_cutoff)
    ])
else:
    print("No strong correlations found for engine PC2.")

# Print the names of meteorological variables strongly correlated with engine PC1
if not meteorological_correlations_pc1.empty:
    print("Variables strongly correlated with engine PC1:")
    correlated_variables_pc1 = meteorological_correlations_pc1.columns[
        (meteorological_correlations_pc1.abs() >= correlation_cutoff).any(axis=0)
    ]
    print(correlated_variables_pc1)
else:
    print("No strong correlations found for engine PC1.")

# Print the names of meteorological variables strongly correlated with engine PC2
if not meteorological_correlations_pc2.empty:
    print("\nVariables strongly correlated with engine PC2:")
    correlated_variables_pc2 = meteorological_correlations_pc2.columns[
        (meteorological_correlations_pc2.abs() >= correlation_cutoff).any(axis=0)
    ]
```

```
    print(correlated_variables_pc2)
else:
    print("No strong correlations found for engine PC2.")
```

```
Variables strongly correlated with engine PC1:
                        temp  feels_like  pressure  humidity  moving
RS_E_InAirTemp_PC1       NaN         NaN       NaN       NaN     NaN
RS_E_OilPress_PC1        NaN         NaN       NaN       NaN     NaN
RS_E_RPM_PC1             NaN         NaN       NaN -0.167098     NaN
RS_E_WatTemp_PC1         NaN         NaN       NaN -0.200300     NaN
RS_T_OilTemp_PC1    0.158492         NaN       NaN -0.212756     NaN

Variables strongly correlated with engine PC2:
                        temp  feels_like  pressure  humidity  moving
RS_E_InAirTemp_PC2       NaN         NaN       NaN       NaN     NaN
RS_E_OilPress_PC2        NaN         NaN       NaN       NaN     NaN
RS_E_RPM_PC2             NaN         NaN       NaN       NaN     NaN
RS_E_WatTemp_PC2         NaN         NaN       NaN       NaN     NaN
RS_T_OilTemp_PC2    0.216623    0.216695       NaN -0.180341     NaN
Variables strongly correlated with engine PC1:
Index(['temp', 'humidity'], dtype='object')

Variables strongly correlated with engine PC2:
Index(['temp', 'feels_like', 'humidity'], dtype='object')
```

The correlation matrix of anomalies reveals a general influence of various meteorological factors on PC2 and a predominant impact of humidity on PC1. However, it's essential to note that these observations might lack reliability due to the model's poor performance.

SVM

We now delve into the efficacy of a one-class Support Vector Machine (SVM) for anomaly detection, leveraging its ability to create a boundary around normal instances in high-dimensional space. SVMs detect outliers by identifying data points that fall outside this boundary, making them well-suited for anomaly detection tasks.

The SVM faces limitations when dealing with large datasets. Its time complexity is O(n^3), and the space complexity is O(n^2), where n represents the size of the training dataset. Consequently, training the model on an exceedingly large dataset may become infeasible due to computational constraints. One potential solution involves working with a smaller dataset. Instead of random instance sampling, to achieve more balanced classes, we can address the issue by strategically partitioning the dataset. In this approach, fewer trains are allocated to the training and validation sets.

This approach aims to mitigate the computational challenges associated with the SVM's inefficiency on exceptionally large datasets.

```
### division with less data

import random
import pandas as pd
import os

# Set the seed for reproducibility
random.seed(42)
```

```
# Set the working directory
directory = '/mnt/c/Users/jibri/OneDrive - INSA Lyon/Bureau/ULB/DataMingin/sncb_da
ta_v4/'
os.chdir(directory)

# IDs of trains with the most anomalies
numbers = [128, 114, 181, 191, 170, 117, 150, 177, 172, 154, 142, 151, 121, 161, 1
10, 126, 134, 146, 148, 160, 164, 166, 167, 175, 183, 187, 190, 192, 123, 125, 163
, 194]

# Complete number IDs of the trains
full_range = set(range(102, 198)) - {118, 132, 193, 195}

# Calculate the remaining numbers
remaining_numbers = full_range - set(numbers)

# Take 9 random numbers without replacement from the remaining numbers
testing_numbers = random.sample(list(remaining_numbers), 9)

filenames_test = [f'train_data_{num}.0.csv' for num in testing_numbers]

# Read dataframes from files
dataframes_test = [pd.read_csv(filename, sep=',', low_memory=False) for filename i
n filenames_test]

remaining_numbers = remaining_numbers - set(testing_numbers)

# Take 2 random numbers without replacement from the remaining numbers
training_numbers = random.sample(list(remaining_numbers), 2)

filenames_train = [f'train_data_{num}.0.csv' for num in training_numbers]

# Read dataframes from files
dataframes_train = [pd.read_csv(filename, sep=',', low_memory=False) for filename
in filenames_train]

remaining_numbers = remaining_numbers - set(training_numbers)

val_numbers = random.sample(list(remaining_numbers), 1)
# Remaining numbers are now used for validation
filenames_val = [f'train_data_{num}.0.csv' for num in val_numbers]

# Read dataframes from files
dataframes_val = [pd.read_csv(filename, sep=',', low_memory=False) for filename in
filenames_val]

# Use the same seed for random.sample to get identical samples
random.seed(42)

test_numbers_anomalies = random.sample(list(numbers), 10)

filenames_test = [f'train_data_{num}.0.csv' for num in test_numbers_anomalies]

# Read dataframes from files
dataframes_test_anomalies = [pd.read_csv(filename, sep=',', low_memory=False) for
```

```
filename in filenames_test]

numbers = set(numbers) - set(test_numbers_anomalies)

train_numbers_anomalies = random.sample(list(numbers), 2)

filenames_train = [f'train_data_{num}.0.csv' for num in train_numbers_anomalies]

# Read dataframes from files
dataframes_train_anomalies = [pd.read_csv(filename, sep=',', low_memory=False) for
filename in filenames_train]

numbers = numbers - set(train_numbers_anomalies)

val_numbers_anomalies = random.sample(list(numbers), 1)

filenames_val = [f'train_data_{num}.0.csv' for num in val_numbers_anomalies]

# Read dataframes from files
dataframes_val_anomalies = [pd.read_csv(filename, sep=',', low_memory=False) for f
ilename in filenames_val]
```

As we did before, the next step involves their concatenation.

```
# Concatenate the dataframes

df_train_sm = pd.concat(dataframes_train, ignore_index=True)
df_test = pd.concat(dataframes_test, ignore_index=True)
df_val_sm = pd.concat(dataframes_val, ignore_index=True)
df_train_an = pd.concat(dataframes_train_anomalies, ignore_index=True)
df_test_an = pd.concat(dataframes_test_anomalies, ignore_index=True)
df_val_an = pd.concat(dataframes_val_anomalies, ignore_index=True)

# Create the final train, validation and test dataframes

df_train_sm = pd.concat([df_train_sm, df_train_an], ignore_index=True)
df_test = pd.concat([df_test, df_test_an], ignore_index=True)
df_val_sm = pd.concat([df_val_sm, df_val_an], ignore_index=True)

del dataframes_train, dataframes_test, dataframes_val, dataframes_train_anomalies,
dataframes_test_anomalies, dataframes_val_anomalies, df_train_an, df_test_an, df_v
al_an
```

As observed earlier, the larger dataset showed that weather and month data are not informative, so we will omit the one-hot encoding step for these features and remove them.

```
# Remove variables that may not affect the model

df_train_sm = df_train_sm.drop(['ID', 'mapped_veh_id', 'lat', 'lon', 'clouds', 'wi
nd_speed',
                                        'wind_deg', 'weather', 'month'], axis=1)

df_test = df_test.drop(['ID', 'lat', 'lon', 'clouds', 'wind_speed',
                                        'wind_deg', 'weather', 'month'], axis=1)

df_val_sm = df_val_sm.drop(['ID',  'lat', 'lon', 'clouds', 'wind_speed',
                                        'wind_deg', 'weather', 'month'], axis=1)
```

Mirroring the methodology used for the Isolation Forest, before delving into the grid search, we first generate a column that identifies the true anomalies.

```
import numpy as np

X = df_train_sm.iloc[:, 2:]
X_val = df_val_sm.iloc[:, 3:]
X_test =df_test.iloc[:, 3:]

# Create a column 'TrueAnomaly' that identifies the observations that we assume to
be anomalies

X['TrueAnomaly'] = np.where(
    (X['RS_E_InAirTemp_PC1'] > 65) | (X['RS_E_InAirTemp_PC2'] > 65) | (X['RS_E_Wat
Temp_PC1'] > 100) |
    (X['RS_E_WatTemp_PC2'] > 100) | (X['RS_T_OilTemp_PC1'] > 115) | (X['RS_T_OilTe
mp_PC2'] > 115) |
    (X['RS_E_WatTemp_PC1'] == 0) | (X['RS_E_WatTemp_PC2'] == 0) | (X['RS_T_OilTemp
_PC1'] == 0) |
    (X['RS_T_OilTemp_PC2'] == 0) | (X['RS_E_OilPress_PC1'] == 0) | (X['RS_E_OilPre
ss_PC2'] == 0) |
    (X['RS_E_OilPress_PC1'] == 690) | (X['RS_E_OilPress_PC2'] == 690) |
    ((X['RS_E_RPM_PC1'] == 0) & (X['RS_E_RPM_PC2'] != 0)) | ((X['RS_E_RPM_PC2'] ==
0) & (X['RS_E_RPM_PC1'] != 0)),
    1, 0
)

# remove the column 'TrueAnomaly' from the train set

y_train = X['TrueAnomaly']
X_train = X.drop(['TrueAnomaly'], axis=1)

X_val['TrueAnomaly'] = np.where(
    (X_val['RS_E_InAirTemp_PC1'] > 65) | (X_val['RS_E_InAirTemp_PC2'] > 65) | (X_v
al['RS_E_WatTemp_PC1'] > 100) |
    (X_val['RS_E_WatTemp_PC2'] > 100) | (X_val['RS_T_OilTemp_PC1'] > 115) | (X_val
['RS_T_OilTemp_PC2'] > 115) |
    (X_val['RS_E_WatTemp_PC1'] == 0) | (X_val['RS_E_WatTemp_PC2'] == 0) | (X_val['
RS_T_OilTemp_PC1'] == 0) |
    (X_val['RS_T_OilTemp_PC2'] == 0) | (X_val['RS_E_OilPress_PC1'] == 0) | (X_val[
'RS_E_OilPress_PC2'] == 0) |
    (X_val['RS_E_OilPress_PC1'] == 690) | (X_val['RS_E_OilPress_PC2'] == 690) |
    ((X_val['RS_E_RPM_PC1'] == 0) & (X_val['RS_E_RPM_PC2'] != 0)) | ((X_val['RS_E_
RPM_PC2'] == 0) & (X_val['RS_E_RPM_PC1'] != 0)),
    1, 0
)

# remove the column 'TrueAnomaly' from the validation set

y_val = X_val['TrueAnomaly']
X_val = X_val.drop(['TrueAnomaly'], axis=1)

# remove the column 'TrueAnomaly' from the test set

X_test['TrueAnomaly'] = np.where(
    (X_test['RS_E_InAirTemp_PC1'] > 65) | (X_test['RS_E_InAirTemp_PC2'] > 65) | (X
```

```
_test['RS_E_WatTemp_PC1'] > 100) |
    (X_test['RS_E_WatTemp_PC2'] > 100) | (X_test['RS_T_OilTemp_PC1'] > 115) | (X_t
est['RS_T_OilTemp_PC2'] > 115) |
    (X_test['RS_E_WatTemp_PC1'] == 0) | (X_test['RS_E_WatTemp_PC2'] == 0) | (X_tes
t['RS_T_OilTemp_PC1'] == 0) |
    (X_test['RS_T_OilTemp_PC2'] == 0) | (X_test['RS_E_OilPress_PC1'] == 0) | (X_te
st['RS_E_OilPress_PC2'] == 0) |
    (X_test['RS_E_OilPress_PC1'] == 690) | (X_test['RS_E_OilPress_PC2'] == 690) |
    ((X_test['RS_E_RPM_PC1'] == 0) & (X_test['RS_E_RPM_PC2'] != 0)) | ((X_test['RS
_E_RPM_PC2'] == 0) & (X_test['RS_E_RPM_PC1'] != 0)),
    1, 0
)
y_test = X_test['TrueAnomaly']
X_test = X_test.drop(['TrueAnomaly'], axis=1)
```

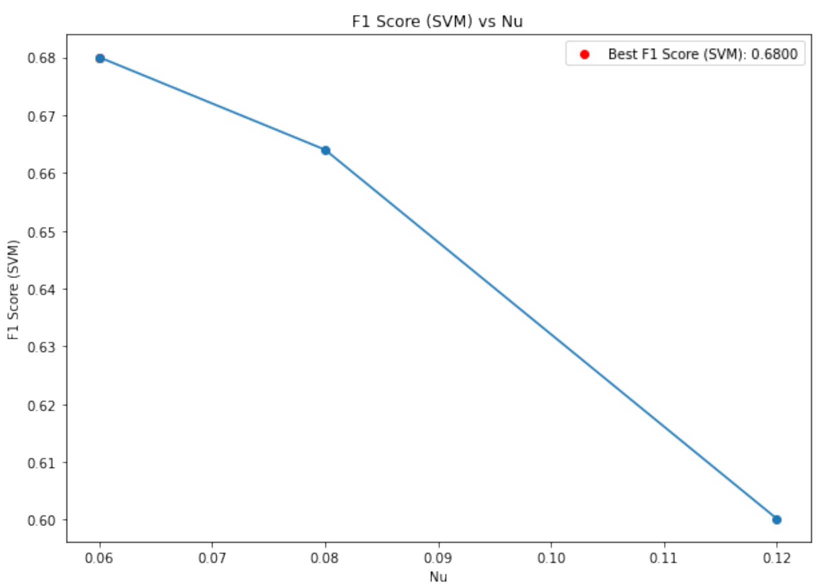
The analysis begins with a grid search involving some nu values, utilizing the OneClassSVM from the scikit-learn library. The model was trained for each nu value on the provided training set (X_train).

The **nu** parameter in the SVM represents the upper bound on the fraction of margin errors and plays a crucial role in determining the trade-off between precision and recall. A higher nu value results in a stricter model, potentially reducing false positives but increasing the chance of false negatives. Conversely, a lower nu value allows for a more lenient model that may capture more anomalies but might also increase false positives.

```
import numpy as np
from sklearn.svm import OneClassSVM
from sklearn.metrics import precision_score, recall_score, f1_score
import matplotlib.pyplot as plt

nu_values = [0.06, 0.08, 0.12]
f1_scores_svm = []

for nu_value in nu_values:
    # Train the model with the current nu value
    svm_model = OneClassSVM(nu=nu_value)
    svm_model.fit(X_train)

    # Predict anomalies on the validation set
    predictions_val_svm = svm_model.predict(X_val)

    # Calculate evaluation metrics
    precision_val_svm = precision_score(y_val, np.where(predictions_val_svm == -1,
1, 0))
    recall_val_svm = recall_score(y_val, np.where(predictions_val_svm == -1, 1, 0)
)
    f1_val_svm = f1_score(y_val, np.where(predictions_val_svm == -1, 1, 0))
    f1_scores_svm.append(f1_val_svm)

    # Print metrics for the current nu value
    print(f'Nu: {nu_value}')
    print(f'Precision (SVM): {precision_val_svm:.4f}')
    print(f'Recall (SVM): {recall_val_svm:.4f}')
    print(f'F1 Score (SVM): {f1_val_svm:.4f}')
    print('\n')
```

```python
# Find the maximum F1 score
best_nu_svm = nu_values[np.argmax(f1_scores_svm)]
best_f1_score_svm = max(f1_scores_svm)

# Create the plot
plt.plot(nu_values, f1_scores_svm, marker='o')
plt.scatter(best_nu_svm, best_f1_score_svm, color='red', label=f'Best F1 Score (SV
M): {best_f1_score_svm:.4f}')
plt.xlabel('Nu')
plt.ylabel('F1 Score (SVM)')
plt.title('F1 Score (SVM) vs Nu')
plt.legend()
plt.show()
```

```
Nu: 0.06
Precision (SVM): 0.6073
Recall (SVM): 0.7725
F1 Score (SVM): 0.6800

Nu: 0.08
Precision (SVM): 0.5451
Recall (SVM): 0.8494
F1 Score (SVM): 0.6640

Nu: 0.12
Precision (SVM): 0.4398
Recall (SVM): 0.9447
F1 Score (SVM): 0.6002
```



After attempting the grid search with small training and validation sets, it proved to be time-consuming. The results obtained indicate that the SVM model's performance, as evaluated on the validation set, falls below 70% in terms of F1 score. Subsequent attempts to perform testing did not conclude in a reasonable and feasible amount of time. It's noteworthy that the execution time of an SVM is also influenced by the size of the testing set, contributing to the extended processing time observed.

Considering that testing results are not expected to surpass or even match those on the validation set, we can assert that the SVM model is not well-suited for this type of data, particularly due to its inefficiency when dealing with larger datasets.

KNN

While Isolation Forest and SVM are well-suited for anomaly detection, we also explore two unsupervised learning models covered in our course: K-Nearest Neighbors (KNN) and DB-Scan.

KNN, or K-Nearest Neighbors, is an algorithm that classifies a point based on the majority of its nearest neighbors in the dataset, with the number of neighbors, known as 'neighbors', which is a key parameter in its configuration. As with other models, we will conduct a grid search to explore the optimal behavior of the model across different values for the number of neighbors.

Our initial step consists creating a column in the datasets that includes the anomalies. Similar to the SVM, this model faces challenges when dealing with large datasets. For this reason, we will utilize the smaller datasets defined earlier.

```
import numpy as np

X = df_train_sm.iloc[:, 2:]
X_val = df_val_sm.iloc[:, 3:]
X_test =df_test.iloc[:, 3:]

# Create a column 'TrueAnomaly' that identifies the observations that we assume to be anomalies

X['TrueAnomaly'] = np.where(
    (X['RS_E_InAirTemp_PC1'] > 65) | (X['RS_E_InAirTemp_PC2'] > 65) | (X['RS_E_WatTemp_PC1'] > 100) |
    (X['RS_E_WatTemp_PC2'] > 100) | (X['RS_T_OilTemp_PC1'] > 115) | (X['RS_T_OilTemp_PC2'] > 115) |
    (X['RS_E_WatTemp_PC1'] == 0) | (X['RS_E_WatTemp_PC2'] == 0) | (X['RS_T_OilTemp_PC1'] == 0) |
    (X['RS_T_OilTemp_PC2'] == 0) | (X['RS_E_OilPress_PC1'] == 0) | (X['RS_E_OilPress_PC2'] == 0) |
    (X['RS_E_OilPress_PC1'] == 690) | (X['RS_E_OilPress_PC2'] == 690) |
    ((X['RS_E_RPM_PC1'] == 0) & (X['RS_E_RPM_PC2'] != 0)) | ((X['RS_E_RPM_PC2'] == 0) & (X['RS_E_RPM_PC1'] != 0)),
    1, 0
)

# remove the column 'TrueAnomaly' from the train set

y_train = X['TrueAnomaly']
X_train = X.drop(['TrueAnomaly'], axis=1)

X_val['TrueAnomaly'] = np.where(
    (X_val['RS_E_InAirTemp_PC1'] > 65) | (X_val['RS_E_InAirTemp_PC2'] > 65) | (X_val['RS_E_WatTemp_PC1'] > 100) |
    (X_val['RS_E_WatTemp_PC2'] > 100) | (X_val['RS_T_OilTemp_PC1'] > 115) | (X_val['RS_T_OilTemp_PC2'] > 115) |
    (X_val['RS_E_WatTemp_PC1'] == 0) | (X_val['RS_E_WatTemp_PC2'] == 0) | (X_val['RS_T_OilTemp_PC1'] == 0) |
    (X_val['RS_T_OilTemp_PC2'] == 0) | (X_val['RS_E_OilPress_PC1'] == 0) | (X_val['RS_E_OilPress_PC2'] == 0) |
```

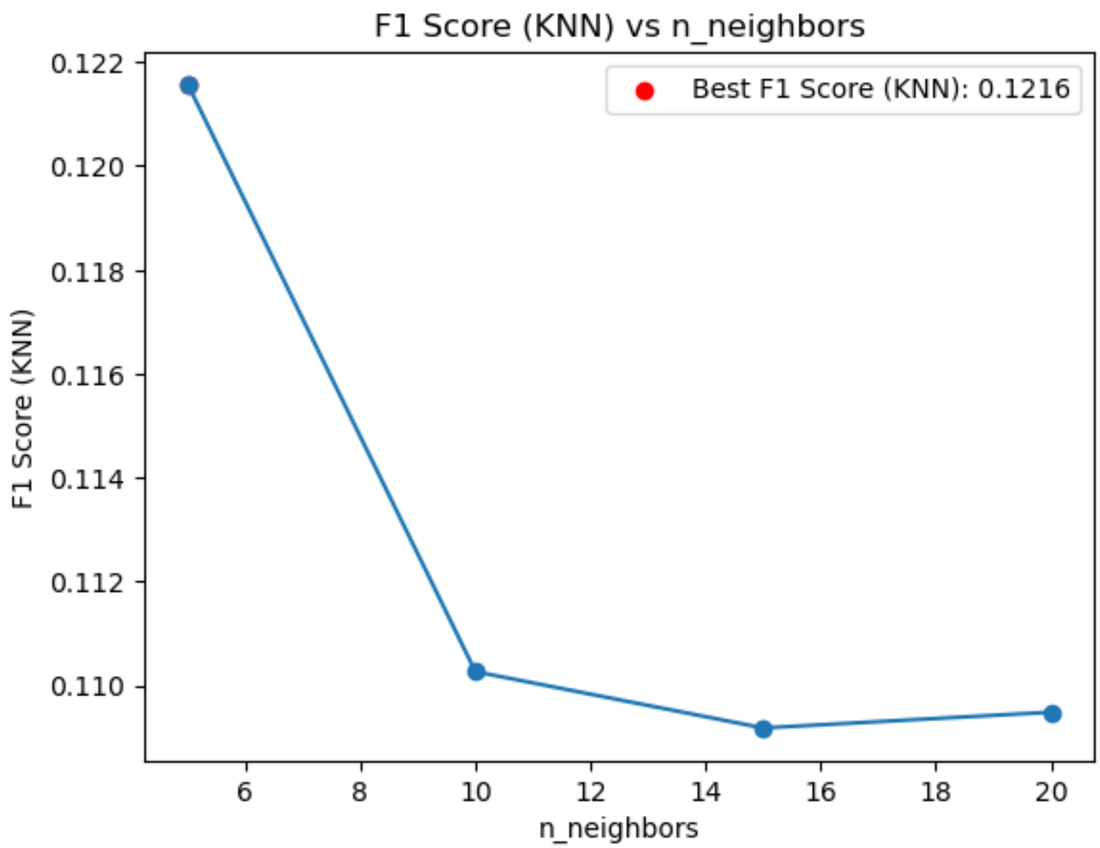
```
    (X_val['RS_E_OilPress_PC1'] == 690) | (X_val['RS_E_OilPress_PC2'] == 690) |
    ((X_val['RS_E_RPM_PC1'] == 0) & (X_val['RS_E_RPM_PC2'] != 0)) | ((X_val['RS_E_
RPM_PC2'] == 0) & (X_val['RS_E_RPM_PC1'] != 0)),
    1, 0
)

# remove the column 'TrueAnomaly' from the validation set

y_val = X_val['TrueAnomaly']
X_val = X_val.drop(['TrueAnomaly'], axis=1)

# remove the column 'TrueAnomaly' from the test set

X_test['TrueAnomaly'] = np.where(
    (X_test['RS_E_InAirTemp_PC1'] > 65) | (X_test['RS_E_InAirTemp_PC2'] > 65) | (X
_test['RS_E_WatTemp_PC1'] > 100) |
    (X_test['RS_E_WatTemp_PC2'] > 100) | (X_test['RS_T_OilTemp_PC1'] > 115) | (X_t
est['RS_T_OilTemp_PC2'] > 115) |
    (X_test['RS_E_WatTemp_PC1'] == 0) | (X_test['RS_E_WatTemp_PC2'] == 0) | (X_tes
t['RS_T_OilTemp_PC1'] == 0) |
    (X_test['RS_T_OilTemp_PC2'] == 0) | (X_test['RS_E_OilPress_PC1'] == 0) | (X_te
st['RS_E_OilPress_PC2'] == 0) |
    (X_test['RS_E_OilPress_PC1'] == 690) | (X_test['RS_E_OilPress_PC2'] == 690) |
    ((X_test['RS_E_RPM_PC1'] == 0) & (X_test['RS_E_RPM_PC2'] != 0)) | ((X_test['RS
_E_RPM_PC2'] == 0) & (X_test['RS_E_RPM_PC1'] != 0)),
    1, 0
)
y_test = X_test['TrueAnomaly']
X_test = X_test.drop(['TrueAnomaly'], axis=1)
```

The grid search conducted on the validation set explores four distinct neighbor values and subsequently identifies the one yielding superior results in terms of the F1 score.

```
import numpy as np
from sklearn.neighbors import NearestNeighbors
from sklearn.metrics import precision_score, recall_score, f1_score
import matplotlib.pyplot as plt

n_neighbors_values = [5, 10, 15, 20]  # You can extend this list with other n_neig
hbors values
f1_scores_knn = []

for n_neighbors_value in n_neighbors_values:
    # Training the model with the current n_neighbors value
    knn = NearestNeighbors(n_neighbors=n_neighbors_value)
    knn.fit(X_train)

    # Calculate the average distances of the k-nearest neighbors for each observat
ion in the validation set
    distances_val, _ = knn.kneighbors(X_val)

    # Calculate the average distance for each observation in the validation set
    average_distances_val = np.mean(distances_val, axis=1)

    # Define a threshold to identify anomalies
    threshold = np.percentile(average_distances_val, 90)  # For example, the 90th
```

```
percentile

    # Identify anomalies in the validation set
    anomalies_val = (average_distances_val > threshold).astype(int)

    # Calculation of evaluation metrics
    precision_val_knn = precision_score(y_val, anomalies_val)
    recall_val_knn = recall_score(y_val, anomalies_val)
    f1_val_knn = f1_score(y_val, anomalies_val)
    f1_scores_knn.append(f1_val_knn)

    # Print metrics for the current n_neighbors value
    print(f'n_neighbors: {n_neighbors_value}')
    print(f'Precision (Validation): {precision_val_knn:.4f}')
    print(f'Recall (Validation): {recall_val_knn:.4f}')
    print(f'F1 Score (Validation): {f1_val_knn:.4f}')
    print('\n')

# Identify the maximum F1 score
best_n_neighbors_knn = n_neighbors_values[np.argmax(f1_scores_knn)]
best_f1_score_knn = max(f1_scores_knn)

# Create the plot
plt.plot(n_neighbors_values, f1_scores_knn, marker='o')
plt.scatter(best_n_neighbors_knn, best_f1_score_knn, color='red', label=f'Best F1 
Score (KNN): {best_f1_score_knn:.4f}')
plt.xlabel('n_neighbors')
plt.ylabel('F1 Score (KNN)')
plt.title('F1 Score (KNN) vs n_neighbors')
plt.legend()
plt.show()
```

```
n_neighbors: 5
Precision (Validation): 0.0947
Recall (Validation): 0.1699
F1 Score (Validation): 0.1216


n_neighbors: 10
Precision (Validation): 0.0858
Recall (Validation): 0.1541
F1 Score (Validation): 0.1103


n_neighbors: 15
Precision (Validation): 0.0850
Recall (Validation): 0.1526
F1 Score (Validation): 0.1092


n_neighbors: 20
Precision (Validation): 0.0852
Recall (Validation): 0.1530
F1 Score (Validation): 0.1095
```



Testing

Due to the substantial time required for the grid search and testing, for the testing, only the code is provided below without including the output. This decision is based on the model's poor performance on the validation set, leading us to expect even worse results on the test set. Hence, further exploration of the model's behavior on these data is deemed unnecessary.

```
# Train the best model on the entire training set
best_model = NearestNeighbors(n_neighbors=best_n_neighbors_knn)
best_model.fit(X_train)

# Predict of anomalies on the test set
distances_test, _ = best_model.kneighbors(X_test)

# Calculate average distances for test set
average_distances_test = np.mean(distances_test, axis=1)

# Set a threshold for anomaly detection
threshold = np.percentile(average_distances_test, 95)

# Generate binary labels for anomalies in the test set
anomalies_test = (average_distances_test > threshold).astype(int)
X_test['IsAnomaly'] = anomalies_test

# Evaluate performance on the test set
precision_test = precision_score(y_test, X_test['IsAnomaly'])
recall_test = recall_score(y_test, X_test['IsAnomaly'])
```

```
f1_test = f1_score(y_test, X_test['IsAnomaly'])

# Print metrics for the test set
print(f'Precision (Test): {precision_test:.4f}')
print(f'Recall (Test): {recall_test:.4f}')
print(f'F1 Score (Test): {f1_test:.4f}')
```

DBSCAN

DBSCAN, or Density-Based Spatial Clustering of Applications with Noise, is a clustering algorithm utilized for identifying clusters of various shapes in a dataset. It can be employed for anomaly detection, distinguishing outliers as noise while forming clusters based on the density of data points. This makes DBSCAN valuable in scenarios where the objective includes both cluster identification and anomaly detection within the data.

This model, although, encounters a time complexity issue, specifically O(n^2). This leads to prolonged training times and can become impractical when dealing with extensive datasets, such as ours. To address this, we can opt for a smaller dataset for training and validation purposes. We will then use the training and validation datasets defined earlier for the SVM.

```
import numpy as np

X = df_train_sm.iloc[:, 2:]
X_val = df_val_sm.iloc[:, 3:]
X_test =df_test.iloc[:, 3:]

# Create a column 'TrueAnomaly' that identifies the observations that we assume to be anomalies

X['TrueAnomaly'] = np.where(
    (X['RS_E_InAirTemp_PC1'] > 65) | (X['RS_E_InAirTemp_PC2'] > 65) | (X['RS_E_WatTemp_PC1'] > 100) |
    (X['RS_E_WatTemp_PC2'] > 100) | (X['RS_T_OilTemp_PC1'] > 115) | (X['RS_T_OilTemp_PC2'] > 115) |
    (X['RS_E_WatTemp_PC1'] == 0) | (X['RS_E_WatTemp_PC2'] == 0) | (X['RS_T_OilTemp_PC1'] == 0) |
    (X['RS_T_OilTemp_PC2'] == 0) | (X['RS_E_OilPress_PC1'] == 0) | (X['RS_E_OilPress_PC2'] == 0) |
    (X['RS_E_OilPress_PC1'] == 690) | (X['RS_E_OilPress_PC2'] == 690) |
    ((X['RS_E_RPM_PC1'] == 0) & (X['RS_E_RPM_PC2'] != 0)) | ((X['RS_E_RPM_PC2'] == 0) & (X['RS_E_RPM_PC1'] != 0)),
    1, 0
)

# remove the column 'TrueAnomaly' from the train set

y_train = X['TrueAnomaly']
X_train = X.drop(['TrueAnomaly'], axis=1)

X_val['TrueAnomaly'] = np.where(
    (X_val['RS_E_InAirTemp_PC1'] > 65) | (X_val['RS_E_InAirTemp_PC2'] > 65) | (X_val['RS_E_WatTemp_PC1'] > 100) |
    (X_val['RS_E_WatTemp_PC2'] > 100) | (X_val['RS_T_OilTemp_PC1'] > 115) | (X_val['RS_T_OilTemp_PC2'] > 115) |
    (X_val['RS_E_WatTemp_PC1'] == 0) | (X_val['RS_E_WatTemp_PC2'] == 0) | (X_val['RS_T_OilTemp_PC1'] == 0) |
```

```
    (X_val['RS_T_OilTemp_PC2'] == 0) | (X_val['RS_E_OilPress_PC1'] == 0) | (X_val[
'RS_E_OilPress_PC2'] == 0) |
    (X_val['RS_E_OilPress_PC1'] == 690) | (X_val['RS_E_OilPress_PC2'] == 690) |
    ((X_val['RS_E_RPM_PC1'] == 0) & (X_val['RS_E_RPM_PC2'] != 0)) | ((X_val['RS_E_
RPM_PC2'] == 0) & (X_val['RS_E_RPM_PC1'] != 0)),
    1, 0
)

# remove the column 'TrueAnomaly' from the validation set

y_val = X_val['TrueAnomaly']
X_val = X_val.drop(['TrueAnomaly'], axis=1)

# remove the column 'TrueAnomaly' from the test set

X_test['TrueAnomaly'] = np.where(
    (X_test['RS_E_InAirTemp_PC1'] > 65) | (X_test['RS_E_InAirTemp_PC2'] > 65) | (X
_test['RS_E_WatTemp_PC1'] > 100) |
    (X_test['RS_E_WatTemp_PC2'] > 100) | (X_test['RS_T_OilTemp_PC1'] > 115) | (X_t
est['RS_T_OilTemp_PC2'] > 115) |
    (X_test['RS_E_WatTemp_PC1'] == 0) | (X_test['RS_E_WatTemp_PC2'] == 0) | (X_tes
t['RS_T_OilTemp_PC1'] == 0) |
    (X_test['RS_T_OilTemp_PC2'] == 0) | (X_test['RS_E_OilPress_PC1'] == 0) | (X_te
st['RS_E_OilPress_PC2'] == 0) |
    (X_test['RS_E_OilPress_PC1'] == 690) | (X_test['RS_E_OilPress_PC2'] == 690) |
    ((X_test['RS_E_RPM_PC1'] == 0) & (X_test['RS_E_RPM_PC2'] != 0)) | ((X_test['RS
_E_RPM_PC2'] == 0) & (X_test['RS_E_RPM_PC1'] != 0)),
    1, 0
)
y_test = X_test['TrueAnomaly']
X_test = X_test.drop(['TrueAnomaly'], axis=1)

del df_train, df_val, df_test

X_test.head()
```

```
   RS_E_InAirTemp_PC1  RS_E_InAirTemp_PC2  RS_E_OilPress_PC1  \
0                 0.0                 0.0                0.0
1                25.0                37.0               20.0
2                25.0                37.0               20.0
3                28.0                37.0               20.0
4                28.0                37.0               20.0

   RS_E_OilPress_PC2  RS_E_RPM_PC1  RS_E_RPM_PC2  RS_E_WatTemp_PC1  \
0                0.0           0.0           0.0               0.0
1                3.0           0.0           0.0              56.0
2                3.0           0.0           0.0              56.0
3                3.0           0.0           0.0              56.0
4                3.0           0.0           0.0              56.0

   RS_E_WatTemp_PC2  RS_T_OilTemp_PC1  RS_T_OilTemp_PC2  temp  feels_like  \
0               0.0              49.0              30.0  2.44        -0.3
1              34.0              50.0              29.0  2.44        -0.3
2              34.0              50.0              29.0  2.44        -0.3
3              34.0              49.0              29.0  2.44        -0.3
4              34.0              49.0              29.0  2.44        -0.3
```

```
   pressure  humidity  moving
0     987.0      75.0       0
1     987.0      75.0       0
2     987.0      75.0       1
3     987.0      75.0       1
4     987.0      75.0       1
```

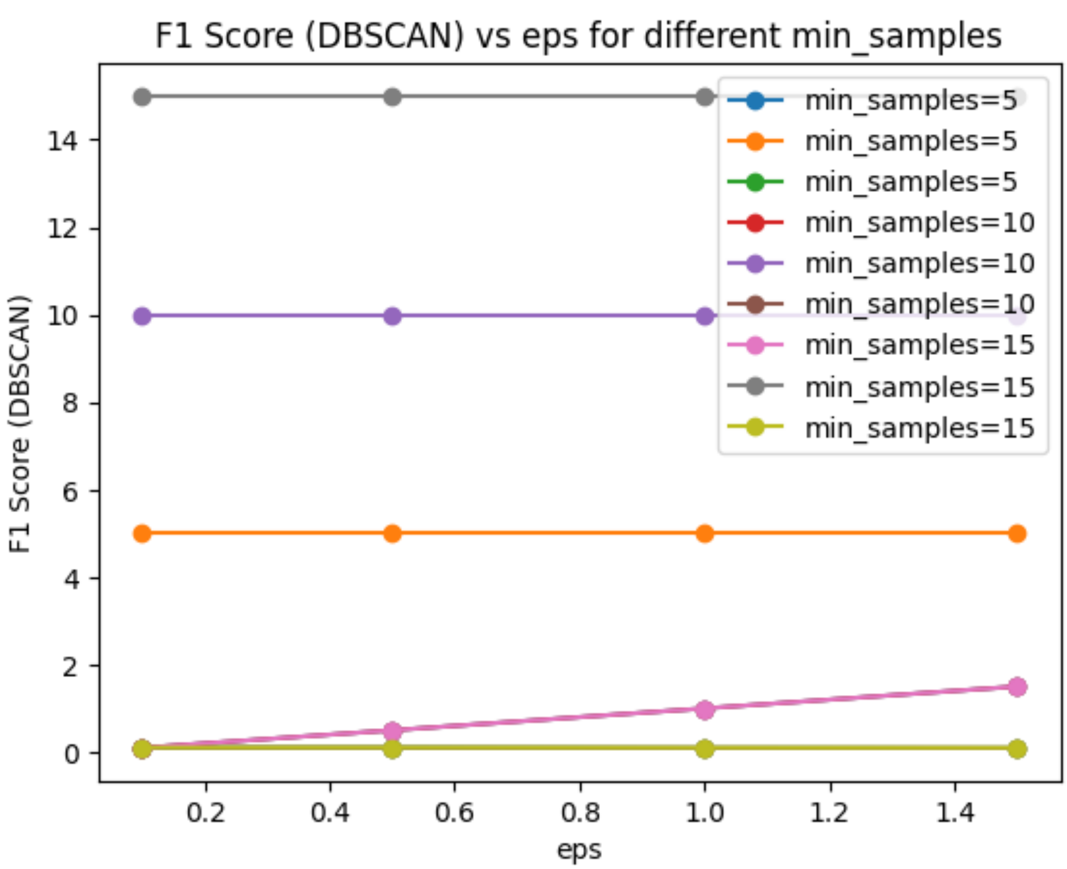
We will now perform a grid search by varying the epsilon (**eps**) value in DBSCAN for anomaly detection. The epsilon value defines the maximum distance for points to be considered neighbors. This search aims to quickly find the optimal epsilon, crucial for balancing cluster formation and noise detection in the data.

```
import numpy as np
from sklearn.cluster import DBSCAN
from sklearn.metrics import precision_score, recall_score, f1_score
import matplotlib.pyplot as plt

eps_values = [0.1, 0.5, 1.0, 1.5]
min_samples_values = [5, 10, 15]

f1_scores_dbscan = []

for min_samples_value in min_samples_values:
    for eps_value in eps_values:
        dbscan = DBSCAN(eps=eps_value, min_samples=min_samples_value)
        labels_train = dbscan.fit_predict(X_train)
        labels_val = dbscan.fit_predict(X_val)

        precision_val_dbscan = precision_score(y_val, np.where(labels_val == -1, 1
, 0), pos_label=1, zero_division=0)
        recall_val_dbscan = recall_score(y_val, np.where(labels_val == -1, 1, 0),
pos_label=1, zero_division=0)
        f1_val_dbscan = f1_score(y_val, np.where(labels_val == -1, 1, 0), pos_labe
l=1, zero_division=0)
        f1_scores_dbscan.append((eps_value, min_samples_value, f1_val_dbscan))

        print(f'min_samples: {min_samples_value}, eps: {eps_value}')
        print(f'Precision (Validation): {precision_val_dbscan:.4f}')
        print(f'Recall (Validation): {recall_val_dbscan:.4f}')
        print(f'F1 Score (Validation): {f1_val_dbscan:.4f}')
        print('\n')

# Find the maximum F1 score
best_params_dbscan = max(f1_scores_dbscan, key=lambda x: x[2])
best_eps_dbscan, best_min_samples_dbscan, best_f1_score_dbscan = best_params_dbsca
n

# Create the plot
plt.plot(eps_values, f1_scores_dbscan[:len(eps_values)], marker='o', label=f'min_s
amples={min_samples_values[0]}')
plt.plot(eps_values, f1_scores_dbscan[len(eps_values):2*len(eps_values)], marker='
o', label=f'min_samples={min_samples_values[1]}')
plt.plot(eps_values, f1_scores_dbscan[2*len(eps_values):], marker='o', label=f'min
_samples={min_samples_values[2]}')

plt.xlabel('eps')
```

```
plt.ylabel('F1 Score (DBSCAN)')
plt.title('F1 Score (DBSCAN) vs eps for different min_samples')
plt.legend()
plt.show()
```

```
min_samples: 5, eps: 0.1
Precision (Validation): 0.0547
Recall (Validation): 0.9801
F1 Score (Validation): 0.1036


min_samples: 5, eps: 0.5
Precision (Validation): 0.0545
Recall (Validation): 0.9766
F1 Score (Validation): 0.1032


min_samples: 5, eps: 1.0
Precision (Validation): 0.0512
Recall (Validation): 0.9133
F1 Score (Validation): 0.0969


min_samples: 5, eps: 1.5
Precision (Validation): 0.0499
Recall (Validation): 0.8874
F1 Score (Validation): 0.0945


min_samples: 10, eps: 0.1
Precision (Validation): 0.0556
Recall (Validation): 0.9971
F1 Score (Validation): 0.1052


min_samples: 10, eps: 0.5
Precision (Validation): 0.0554
Recall (Validation): 0.9937
F1 Score (Validation): 0.1049


min_samples: 10, eps: 1.0
Precision (Validation): 0.0529
Recall (Validation): 0.9465
F1 Score (Validation): 0.1002


min_samples: 10, eps: 1.5
Precision (Validation): 0.0517
Recall (Validation): 0.9237
F1 Score (Validation): 0.0980


min_samples: 15, eps: 0.1
Precision (Validation): 0.0557
```

```
Recall (Validation): 0.9991
F1 Score (Validation): 0.1054


min_samples: 15, eps: 0.5
Precision (Validation): 0.0556
Recall (Validation): 0.9977
F1 Score (Validation): 0.1053


min_samples: 15, eps: 1.0
Precision (Validation): 0.0538
Recall (Validation): 0.9640
F1 Score (Validation): 0.1019


min_samples: 15, eps: 1.5
Precision (Validation): 0.0525
Recall (Validation): 0.9394
F1 Score (Validation): 0.0995
```



Testing

The optimal epsilon value is now applied to predict anomalies on the test set. As for the KNN, the results on the validation set are not promising, so it is not surprising that the performance on the test set is also disappointing. Furthermore, although the recall is considerably high, the precision and F1 score are low, indicating that the model lacks reliability.

We can therefore exclude the use of DB-Scan for the detection of anomalies.

```
best_model = DBSCAN(eps=best_eps_dbscan, min_samples=best_min_samples_dbscan)

# Train on X
best_model.fit(X_train)

# Predict anomalies on X_test
dbscan_test = best_model.fit_predict(X_test)

# Label anomalies on X_test
X_test['IsAnomaly'] = np.where(best_model.labels_ == -1, 1, 0)

# Evaluate performance on the test set
precision_test = precision_score(y_test, X_test['IsAnomaly'])
recall_test = recall_score(y_test, X_test['IsAnomaly'])
f1_test = f1_score(y_test, X_test['IsAnomaly'])

# Print metrics for the test set
print(f'Precision (Test): {precision_test:.4f}')
print(f'Recall (Test): {recall_test:.4f}')
print(f'F1 Score (Test): {f1_test:.4f}')
```

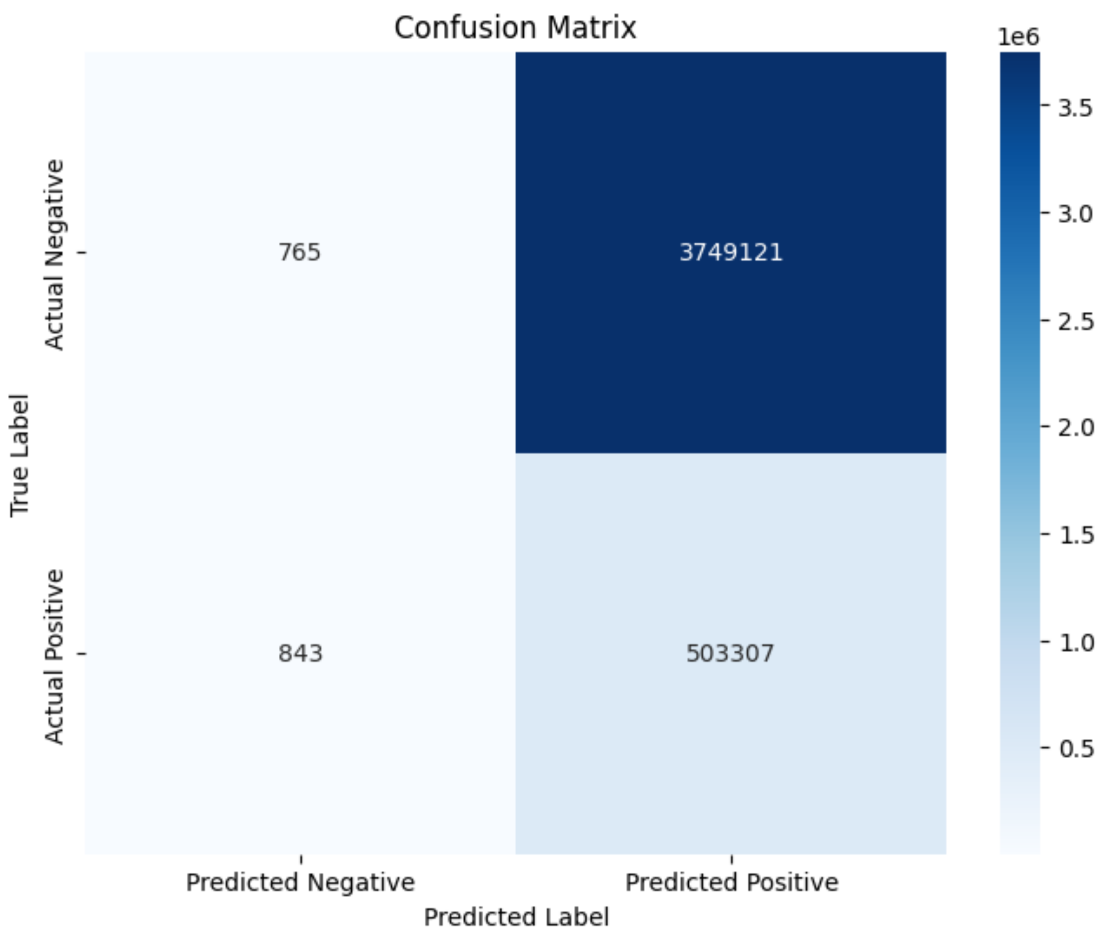
```
Precision (Test): 0.1184
Recall (Test): 0.9983
F1 Score (Test): 0.2116
```

We will now delve into the visualization of the confusion matrix to assess the model's performance on the testing data.

```
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.metrics import confusion_matrix

# Count anomalies found by the model
count_an = X_test['IsAnomaly'].value_counts()

# Count anomalies in the test set
count_an_true = y_test.value_counts()

# See if there's correspondence between the anomalies found by the model and the a
nomalies in the test set
print('Anomalies found:', count_an)
print('Number of true anomalies', count_an_true)

# Compute confusion matrix
conf_matrix = confusion_matrix(y_test, X_test['IsAnomaly'])

# Extract TP, TN, FP, FN
TN, FP, FN, TP = conf_matrix.ravel()

print(f'TN: {TN}, FP: {FP}, FN: {FN}, TP: {TP}')

# Create heatmap of confusion matrix
plt.figure(figsize=(8, 6))
sns.heatmap(conf_matrix, annot=True, fmt='d', cmap='Blues',
```

```
            xticklabels=['Predicted Negative', 'Predicted Positive'],
            yticklabels=['Actual Negative', 'Actual Positive'])
plt.xlabel('Predicted Label')
plt.ylabel('True Label')
plt.title('Confusion Matrix')
plt.show()
```

```
Anomalies found: IsAnomaly
1    4252428
0       1608
Name: count, dtype: int64
Number of true anomalies TrueAnomaly
0    3749886
1     504150
Name: count, dtype: int64
TN: 765, FP: 3749121, FN: 843, TP: 503307
```



It's evident that the number of True Negatives is very low compared to the number of False Positives, and the number of True Positives is significantly higher than the number of False Negatives. This reflects a high recall, albeit at the expense of precision.

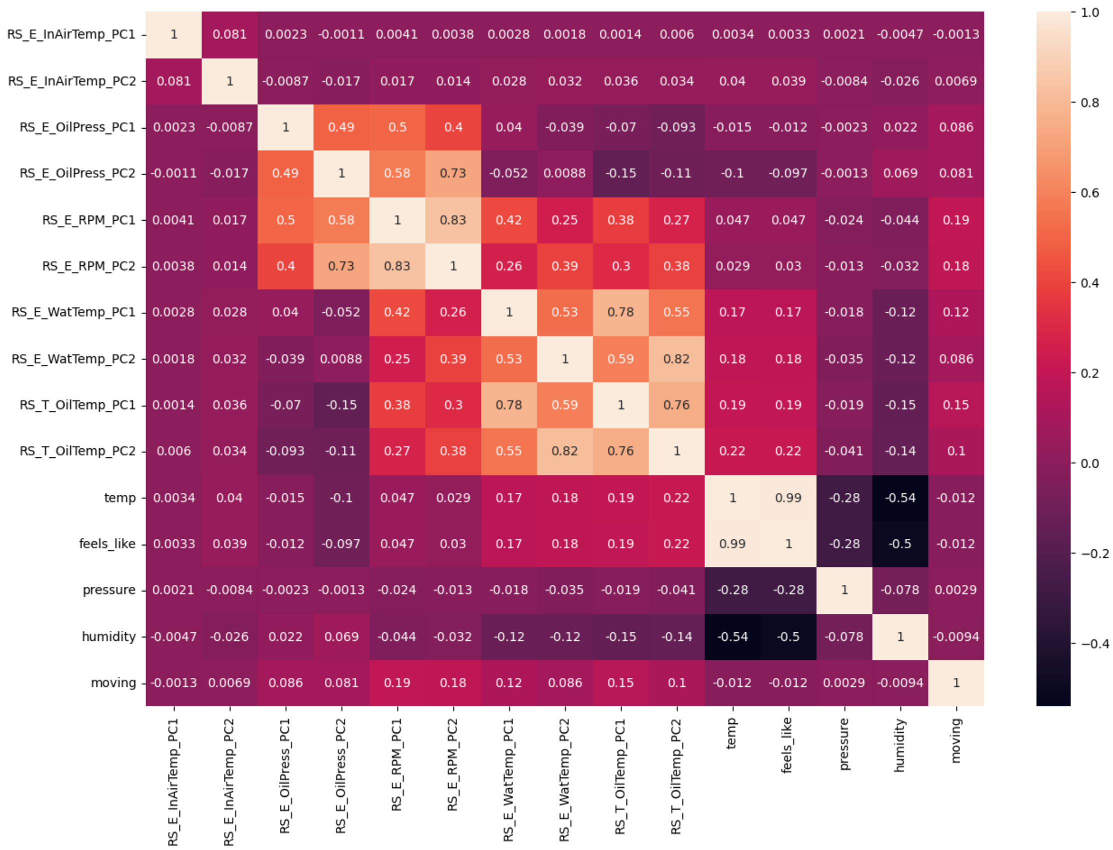
Influence of weather variables on the predicted anomalies

For consistency, we explore the influence of weather variables on the predicted anomalies.

```
X_test_anomalies = X_test[X_test['IsAnomaly'] == 1]
```

```
import seaborn as sns
import matplotlib.pyplot as plt
```

```python
X_test_corr = X_test_anomalies.drop(['IsAnomaly'], axis=1)

# Correlation matrix for the anomalies
corr_matrix = X_test_corr.corr()

plt.figure(figsize=(15, 10))

sns.heatmap(corr_matrix, annot=True)

# show the plot
plt.show()
```



By selecting a cutoff that we arbitrarely set at 0.2, we visualize more clearly the variables that influence the train variables.

```python
# List of PC1 motor variables
var_pc1 = [
    'RS_E_InAirTemp_PC1', 'RS_E_OilPress_PC1', 'RS_E_RPM_PC1', 'RS_E_WatTemp_PC1',
'RS_T_OilTemp_PC1'
]

# List of PC2 motor variables
var_pc2 = [
    'RS_E_InAirTemp_PC2', 'RS_E_OilPress_PC2', 'RS_E_RPM_PC2', 'RS_E_WatTemp_PC2',
'RS_T_OilTemp_PC2'
]

# List of all variables (motor and meteorological)
all_variables = var_pc1 + var_pc2 + [
    'temp', 'feels_like', 'pressure', 'humidity', 'moving'
```

```
]

corr_matrix = X_test_anomalies[all_variables].corr()

# Specify the cutoff for the absolute correlation value (e.g., 0.2)
correlation_cutoff = 0.2

meteorological_correlations_pc1 = corr_matrix[corr_matrix.index.isin(var_pc1)][['t
emp', 'feels_like', 'pressure', 'humidity', 'moving']]

meteorological_correlations_pc2 = corr_matrix[corr_matrix.index.isin(var_pc2)][['t
emp', 'feels_like', 'pressure', 'humidity', 'moving']]

# Print meteorological variables strongly correlated with engine PC1
if not meteorological_correlations_pc1.empty:
    print("Variables strongly correlated with engine PC1:")
    print(meteorological_correlations_pc1[
        (meteorological_correlations_pc1.abs() >= correlation_cutoff)
    ])
else:
    print("No strong correlations found for engine PC1.")

# Print meteorological variables strongly correlated with engine PC2
if not meteorological_correlations_pc2.empty:
    print("\nVariables strongly correlated with engine PC2:")
    print(meteorological_correlations_pc2[
        (meteorological_correlations_pc2.abs() >= correlation_cutoff)
    ])
else:
    print("No strong correlations found for engine PC2.")

# Print names of meteorological variables strongly correlated with engine PC1
if not meteorological_correlations_pc1.empty:
    print("Variables strongly correlated with engine PC1:")
    correlated_variables_pc1 = meteorological_correlations_pc1.columns[
        (meteorological_correlations_pc1.abs() >= correlation_cutoff).any(axis=0)
    ]
    print(correlated_variables_pc1)
else:
    print("No strong correlations found for engine PC1.")

# Print names of meteorological variables strongly correlated with engine PC2
if not meteorological_correlations_pc2.empty:
    print("\nVariables strongly correlated with engine PC2:")
    correlated_variables_pc2 = meteorological_correlations_pc2.columns[
        (meteorological_correlations_pc2.abs() >= correlation_cutoff).any(axis=0)
    ]
    print(correlated_variables_pc2)
else:
    print("No strong correlations found for engine PC2.")
```

```
Variables strongly correlated with engine PC1:
                    temp  feels_like  pressure  humidity  moving
RS_E_InAirTemp_PC1   NaN         NaN       NaN       NaN     NaN
RS_E_OilPress_PC1    NaN         NaN       NaN       NaN     NaN
RS_E_RPM_PC1         NaN         NaN       NaN       NaN     NaN
```

```
RS_E_WatTemp_PC1     NaN          NaN        NaN        NaN      NaN
RS_T_OilTemp_PC1     NaN          NaN        NaN        NaN      NaN

Variables strongly correlated with engine PC2:
                        temp  feels_like  pressure  humidity  moving
RS_E_InAirTemp_PC2       NaN         NaN       NaN       NaN     NaN
RS_E_OilPress_PC2        NaN         NaN       NaN       NaN     NaN
RS_E_RPM_PC2             NaN         NaN       NaN       NaN     NaN
RS_E_WatTemp_PC2         NaN         NaN       NaN       NaN     NaN
RS_T_OilTemp_PC2    0.216387    0.215658       NaN       NaN     NaN
Variables strongly correlated with engine PC1:
Index([], dtype='object')

Variables strongly correlated with engine PC2:
Index(['temp', 'feels_like'], dtype='object')
```

lthough these results cannot be considered reliable for the low performance metrics, only the PC2 is influenced by meteorological factors.

Variational autoencoder

The variational autoencoder serves as a generative model, acquiring a latent representation of the data. It learns this latent representation and endeavors to faithfully reproduce the original data from it.

In the context of anomaly detection, we leverage the reconstruction error to identify anomalies. Elevated reconstruction errors signify anomalous data. During training, the model is fine-tuned to minimize the reconstruction error solely on "correct" data. As anomalies deviate from the norm, the model tends to exhibit higher reconstruction errors for such instances. In testing, the reconstruction error becomes a key metric for detecting anomalies in previously unseen data.

```python
import torch
from torchvision import datasets
from torchvision import transforms
import matplotlib.pyplot as plt
import pandas as pd
import numpy as np
import random


if torch.cuda.is_available():
    DEVICE = "cuda"
else:
    DEVICE = "cpu"
print("Selected device is",DEVICE)

random.seed(42)
```

```
Selected device is cuda
```

Data Preparation

Initially, the selected training data files are loaded. Unnecessary columns, such as id, time, coordinates, etc., are dropped. Additionally, rows containing missing values are removed. The remaining columns undergo scaling to achieve zero mean and unit variance. The scaled data is employed for model training. Categorical columns undergo one-hot encoding, and the resulting columns are used in the training process. In the case of the training dataset, anomalies are excluded from the dataset and are solely utilized during evaluation.

```
# Create custom dataset class
class TrainDataset(torch.utils.data.Dataset):
    def __init__(self, data_path, files, train=True, transform=None, anomaly_per_c
lass=False):
        # Load csv files from data_path
        self.data_path = data_path
        self.files = files

        for file in files:
            data = pd.read_csv(data_path + file, sep=',')
            if file == files[0]:
                self.data = data
            else:
                self.data = pd.concat([self.data, data], axis=0)

        # Print number of nan in temp in rows
        # Print("Number of NaN rows:", self.data['temp'].isnull().sum().sum())

        # Print("Number of NaN rows:", self.data.isnull().sum().sum())
        self.data = self.data.dropna()

        self.data = self.data.drop(['Unnamed: 0', 'timestamps_UTC', 'ID', 'mapped_
veh_id', 'lat', 'lon', 'clouds', 'wind_speed',
                                    'wind_deg', 'weather', 'feels_like'], axis
=1)


        # Print header
        # Print(self.data.head())

        # & self.data['moving'] == 1

        # If train remove anomalies
        if not anomaly_per_class:
            self.data['TrueAnomaly'] = np.where(
                (self.data['RS_E_InAirTemp_PC1'] > 65) | (self.data['RS_E_InAirTem
p_PC2'] > 65) | (self.data['RS_E_WatTemp_PC1'] > 100) |
                (self.data['RS_E_WatTemp_PC2'] > 100) | (self.data['RS_T_OilTemp_P
C1'] > 115) | (self.data['RS_T_OilTemp_PC2'] > 115) |
                (self.data['RS_E_WatTemp_PC1'] == 0) | (self.data['RS_E_WatTemp_PC
2'] == 0) | (self.data['RS_T_OilTemp_PC1'] == 0) |
                (self.data['RS_T_OilTemp_PC2'] == 0) | (self.data['RS_E_OilPress_P
C1'] == 0) | (self.data['RS_E_OilPress_PC2'] == 0) |
                ((self.data['RS_E_RPM_PC1'] == 0) & (self.data['RS_E_RPM_PC2'] !=
0)) | ((self.data['RS_E_RPM_PC2'] == 0) & (self.data['RS_E_RPM_PC1'] != 0)),
                1, 0
            )
```

```
        else:
            # Each type of anomlay gets different class label
            self.data['TrueAnomaly'] = np.where(
                (self.data['RS_E_InAirTemp_PC1'] > 65), 1, 0
            )
            self.data['TrueAnomaly'] = np.where(
                (self.data['RS_E_InAirTemp_PC2'] > 65), 2, self.data['TrueAnomaly'
]
            )
            self.data['TrueAnomaly'] = np.where(
                (self.data['RS_E_WatTemp_PC1'] > 100), 3, self.data['TrueAnomaly']
            )
            self.data['TrueAnomaly'] = np.where(
                (self.data['RS_E_WatTemp_PC2'] > 100), 4, self.data['TrueAnomaly']
            )
            self.data['TrueAnomaly'] = np.where(
                (self.data['RS_T_OilTemp_PC1'] > 115), 5, self.data['TrueAnomaly']
            )
            self.data['TrueAnomaly'] = np.where(
                (self.data['RS_T_OilTemp_PC2'] > 115), 6, self.data['TrueAnomaly']
            )
            self.data['TrueAnomaly'] = np.where(
                (self.data['RS_E_WatTemp_PC1'] == 0), 7, self.data['TrueAnomaly']
            )
            self.data['TrueAnomaly'] = np.where(
                (self.data['RS_E_WatTemp_PC2'] == 0), 8, self.data['TrueAnomaly']
            )
            self.data['TrueAnomaly'] = np.where(
                (self.data['RS_T_OilTemp_PC1'] == 0), 9, self.data['TrueAnomaly']
            )
            self.data['TrueAnomaly'] = np.where(
                (self.data['RS_T_OilTemp_PC2'] == 0), 10, self.data['TrueAnomaly']
            )
            self.data['TrueAnomaly'] = np.where(
                (self.data['RS_E_OilPress_PC1'] == 0), 11, self.data['TrueAnomaly'
]
            )
            self.data['TrueAnomaly'] = np.where(
                (self.data['RS_E_OilPress_PC2'] == 0), 12, self.data['TrueAnomaly'
]
            )
            self.data['TrueAnomaly'] = np.where(
                ((self.data['RS_E_RPM_PC1'] == 0) & (self.data['RS_E_RPM_PC2'] !=
0)), 13, self.data['TrueAnomaly']
            )
            self.data['TrueAnomaly'] = np.where(
                ((self.data['RS_E_RPM_PC2'] == 0) & (self.data['RS_E_RPM_PC1'] !=
0)), 14, self.data['TrueAnomaly']
            )


        # Get the means and stds of the columns
        # Print(self.data.mean())
        # Print(self.data.std())
```

```
        # Standardize data in RS_E_InAirTemp_PC1,RS_E_InAirTemp_PC2,RS_E_OilPress_
PC1,RS_E_OilPress_PC2,RS_E_RPM_PC1,RS_E_RPM_PC2,RS_E_WatTemp_PC1,RS_E_WatTemp_PC2,
RS_T_OilTemp_PC1,RS_T_OilTemp_PC2 columns
        self.data['RS_E_InAirTemp_PC1'] = (self.data['RS_E_InAirTemp_PC1'] - self.
data['RS_E_InAirTemp_PC1'].mean()) / self.data['RS_E_InAirTemp_PC1'].std()
        self.data['RS_E_InAirTemp_PC2'] = (self.data['RS_E_InAirTemp_PC2'] - self.
data['RS_E_InAirTemp_PC2'].mean()) / self.data['RS_E_InAirTemp_PC2'].std()
        self.data['RS_E_OilPress_PC1'] = (self.data['RS_E_OilPress_PC1'] - self.da
ta['RS_E_OilPress_PC1'].mean()) / self.data['RS_E_OilPress_PC1'].std()
        self.data['RS_E_OilPress_PC2'] = (self.data['RS_E_OilPress_PC2'] - self.da
ta['RS_E_OilPress_PC2'].mean()) / self.data['RS_E_OilPress_PC2'].std()
        self.data['RS_E_RPM_PC1'] = (self.data['RS_E_RPM_PC1'] - self.data['RS_E_R
PM_PC1'].mean()) / self.data['RS_E_RPM_PC1'].std()
        self.data['RS_E_RPM_PC2'] = (self.data['RS_E_RPM_PC2'] - self.data['RS_E_R
PM_PC2'].mean()) / self.data['RS_E_RPM_PC2'].std()
        self.data['RS_E_WatTemp_PC1'] = (self.data['RS_E_WatTemp_PC1'] - self.data
['RS_E_WatTemp_PC1'].mean()) / self.data['RS_E_WatTemp_PC1'].std()
        self.data['RS_E_WatTemp_PC2'] = (self.data['RS_E_WatTemp_PC2'] - self.data
['RS_E_WatTemp_PC2'].mean()) / self.data['RS_E_WatTemp_PC2'].std()
        self.data['RS_T_OilTemp_PC1'] = (self.data['RS_T_OilTemp_PC1'] - self.data
['RS_T_OilTemp_PC1'].mean()) / self.data['RS_T_OilTemp_PC1'].std()
        self.data['RS_T_OilTemp_PC2'] = (self.data['RS_T_OilTemp_PC2'] - self.data
['RS_T_OilTemp_PC2'].mean()) / self.data['RS_T_OilTemp_PC2'].std()


        # If temp is < -50 then we add 273.15 twice to convert to kelvin
        self.data['temp'] = np.where(self.data['temp'] < -100, self.data['temp'] +
273.15 + 273.15, self.data['temp'] + 273.15)

        # Standardize data in temp column
        self.data['temp'] = (self.data['temp'] - self.data['temp'].mean()) / self.
data['temp'].std()

        # Standardize data in pressure column
        self.data['pressure'] = (self.data['pressure'] - self.data['pressure'].mea
n()) / self.data['pressure'].std()

        # Standardize data in humidity column
        self.data['humidity'] = (self.data['humidity'] - self.data['humidity'].mea
n()) / self.data['humidity'].std()

        # One hot encode categorical data month
        self.data = pd.get_dummies(self.data, columns=['month'])

        # Standardize data in month columns
        # Self.data['month'] = (self.data['month'] - self.data['month'].mean()) /
self.data['month'].std()

        # Print(self.data.head())

        # Print the number of anomalies
        #print("Number of anomalies:", len(self.data[self.data['TrueAnomaly'] >= 1
]))
        #print("Number of rows:", len(self.data))
        if train:
            self.data = self.data[self.data['TrueAnomaly'] == 0]
```

```python
        if not train:
            # Keep only every 10th row where true anomaly is 0
            self.data = self.data[(self.data.index % 5 == 0 & (self.data['TrueAnom
aly'] == 0)) | (self.data['TrueAnomaly'] != 0)]
            print("Number of rows:", len(self.data))


        self.targets = self.data['TrueAnomaly']
        self.targets = self.targets.to_numpy()
        self.data = self.data.drop(['TrueAnomaly'], axis=1)

        # Transform data to numpy array
        self.data = self.data.to_numpy()


        # Print nan values
        # print("Number of NaN values:", np.count_nonzero(np.isnan(self.data)))

        # Transform data to np.float32
        self.data = self.data.astype(np.float32)

        self.transform = transform

    def __getitem__(self, index):
        x = self.data[index]
        y = self.targets[index]
        if self.transform:
            x = self.transform(x)
        return x, y

    def __len__(self):
        return len(self.data)


# IDs of trains with the most anomalies
numbers = [128, 114, 181, 191, 170, 117, 150, 177, 172, 154, 142, 151, 121, 161, 1
10, 126, 134, 146, 148, 160, 164, 166, 167, 175, 183, 187, 190, 192, 123, 125, 163
, 194]

# Complete number IDs of the trains
full_range = set(range(102, 198)) - {118, 132, 193, 195}

# Calculate the remaining numbers
remaining_numbers = full_range - set(numbers)

# Take 9 random numbers without replacement from the remaining numbers
testing_numbers = random.sample(list(remaining_numbers), 9)

filenames_test = [f'train_data_{num}.0.csv' for num in testing_numbers]


remaining_numbers = remaining_numbers - set(testing_numbers)

# Take 41 random numbers without replacement from the remaining numbers
```

```
training_numbers = random.sample(list(remaining_numbers), 41)

filenames_train = [f'train_data_{num}.0.csv' for num in training_numbers]


remaining_numbers = remaining_numbers - set(training_numbers)

# Remaining numbers are now used for validation
filenames_val = [f'train_data_{num}.0.csv' for num in remaining_numbers]

test_numbers_anomalies = random.sample(list(numbers), 10)

filenames_test.extend([f'train_data_{num}.0.csv' for num in test_numbers_anomalies
])


numbers = set(numbers) - set(test_numbers_anomalies)

train_numbers_anomalies = random.sample(list(numbers), 18)

filenames_train.extend([f'train_data_{num}.0.csv' for num in train_numbers_anomali
es])

numbers = numbers - set(train_numbers_anomalies)

filenames_val.extend([f'train_data_{num}.0.csv' for num in numbers])



# Create train dataset
dataset = TrainDataset(data_path='../train_datas/sncb_data_v4_preprocess2/', files
=filenames_train, train=True)

# DataLoader is used to load the dataset
# For training
loader = torch.utils.data.DataLoader(dataset = dataset,
                                     batch_size = 256,
                                     shuffle = True)




# Create test dataset
dataset_val = TrainDataset(data_path='../train_datas/sncb_data_v4_preprocess2/', f
iles=filenames_val, train=False)

# DataLoader is used to load the dataset
# For testing
loader_val = torch.utils.data.DataLoader(dataset = dataset_val,
                                     batch_size = 1,
                                     shuffle = False)

print(len(dataset))
print(len(dataset_val))
```

```
Number of rows: 588445
10903169
588445
```

Definition of the Model

The model consists in two components: the encoder and the decoder. The encoder processes the input data to generate a latent representation, while the decoder reconstructs the original data from the latent representation. Both the encoder and the decoder are trained jointly, aiming to minimize the mean squared error between the original and reconstructed data. The latent vector size is set to 3, representing a compressed form of the input data vector, which is subsequently decompressed to reconstruct the original data vector.

```python
# Creating a PyTorch class
class AE(torch.nn.Module):
    def __init__(self):
        super().__init__()

        self.encoder = torch.nn.Sequential(
            torch.nn.Linear(23, 256),
            torch.nn.ReLU(),
            torch.nn.Linear(256, 64),
            torch.nn.ReLU(),
            torch.nn.Linear(64, 32),
            torch.nn.ReLU(),
            torch.nn.Linear(32, 16),
            torch.nn.ReLU(),
            torch.nn.Linear(16, 3)
        )

        self.decoder = torch.nn.Sequential(
            torch.nn.Linear(3, 16),
            torch.nn.ReLU(),
            torch.nn.Linear(16, 32),
            torch.nn.ReLU(),
            torch.nn.Linear(32, 64),
            torch.nn.ReLU(),
            torch.nn.Linear(64, 256),
            torch.nn.ReLU(),
            torch.nn.Linear(256, 23)
        )

    def forward(self, x):
        encoded = self.encoder(x)
        decoded = self.decoder(encoded)
        return decoded

# Model Initialization
model = AE()
model.cuda()

# Validation using MSE Loss function
loss_function = torch.nn.MSELoss(reduction='mean')

# Using an Adam Optimizer with lr = 0.1
optimizer = torch.optim.Adam(model.parameters(),
```

```
                                  lr = 1e-3,
                                  weight_decay = 1e-8)
```

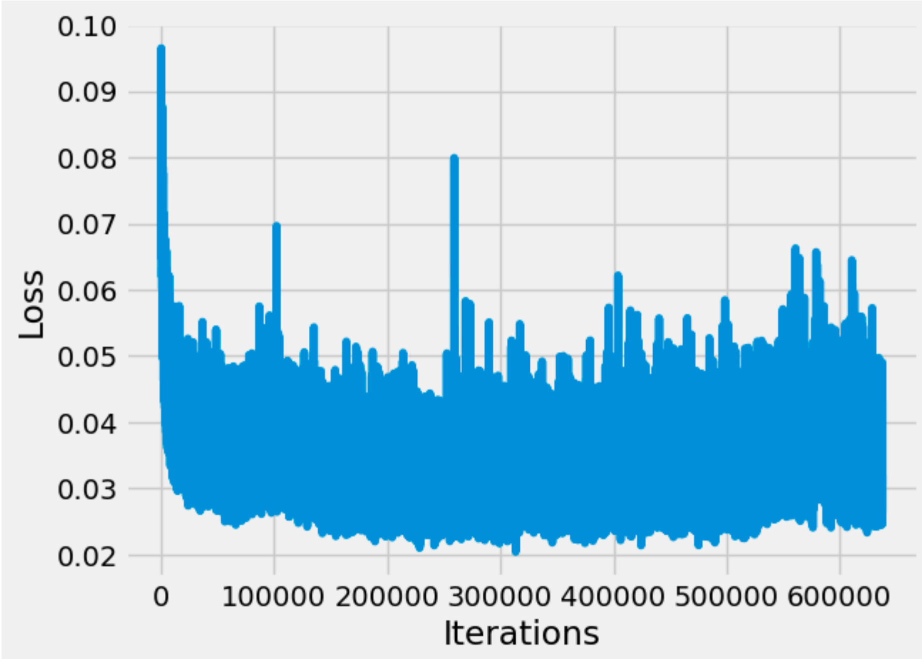
Model Training

Multiple experiments are conducted to optimize training epochs, learning rate, batch size, and model architecture. The finalized model is showcased later.

```
epochs = 15
outputs = []
losses = []
for epoch in range(epochs):
    for (d, _) in loader:
      # Reshaping the image to (-1, 784)
      d = d.reshape(-1, 23)

      # Output of Autoencoder
      reconstructed = model(d.cuda())

      # Calculating the loss function
      loss = loss_function(reconstructed.cuda(), d.cuda())

      # The gradients are set to zero,
      # the gradient is computed and stored.
      # .step() performs parameter update
      optimizer.zero_grad()
      loss.backward()
      optimizer.step()
      print('Epoch [{}/{}], Loss: {:.4f}'.format(epoch+1, epochs, loss.item()))

      # Storing the losses in a list for plotting
      losses.append(loss.cpu().detach().numpy())
    outputs.append((epochs, d.cpu().detach(), reconstructed.cpu().detach()))

# Defining the Plot Style
plt.style.use('fivethirtyeight')
plt.xlabel('Iterations')
plt.ylabel('Loss')

# Plotting the last 100 values
plt.plot(losses[1000:])
```

```
Epoch [1/15], Loss: 0.5223

Epoch [1/15], Loss: 0.4762

Epoch [1/15], Loss: 0.4494

Epoch [1/15], Loss: 0.3998

Epoch [1/15], Loss: 0.4570

Epoch [1/15], Loss: 0.4745

Epoch [1/15], Loss: 0.4282
```

```
Epoch [1/15], Loss: 0.4229

Epoch [1/15], Loss: 0.4296

Epoch [1/15], Loss: 0.4402

Epoch [1/15], Loss: 0.4904

Epoch [1/15], Loss: 0.4160

Epoch [1/15], Loss: 0.4437

Epoch [1/15], Loss: 0.4911

Epoch [1/15], Loss: 0.4304

Epoch [1/15], Loss: 0.3889

Epoch [1/15], Loss: 0.4485

Epoch [1/15], Loss: 0.4025

Epoch [1/15], Loss: 0.4326

Epoch [1/15], Loss: 0.4357

Epoch [1/15], Loss: 0.4238

Epoch [1/15], Loss: 0.4760

Epoch [1/15], Loss: 0.4145

Epoch [1/15], Loss: 0.4023

Epoch [1/15], Loss: 0.4411

...

Epoch [15/15], Loss: 0.0327

Epoch [15/15], Loss: 0.0314

Epoch [15/15], Loss: 0.0373

Epoch [15/15], Loss: 0.0321

Output is truncated. View as a scrollable element or open in a text editor. Adjust
cell output settings...

[<matplotlib.lines.Line2D at 0x7f592850a090>]
```




```
# visualize the losses for each data point and visualize the y
model.eval()
losses_train = []
for (d, y) in loader:
    # Reshaping the image to (-1, 784)
    d = d.reshape(-1, 23)

    # Output of Autoencoder
    reconstructed = model(d.cuda())

    # Calculating the loss function
    loss = loss_function(reconstructed.cuda(), d.cuda())

    # Storing the losses in a list for plotting
    losses_train.append((y[0], loss.item()))

losses_train_og = losses_train.copy()
```

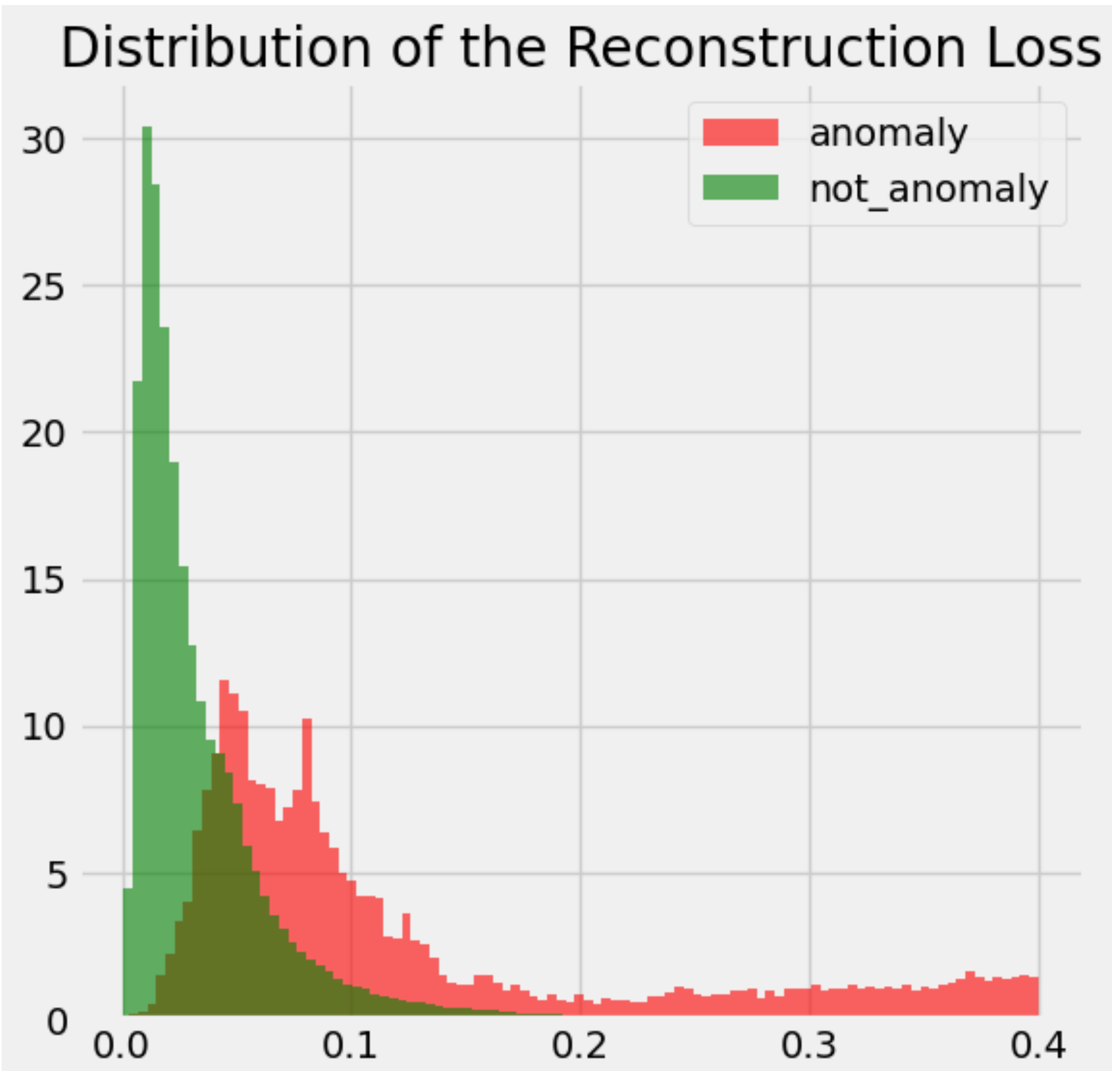
Validation Evaluation

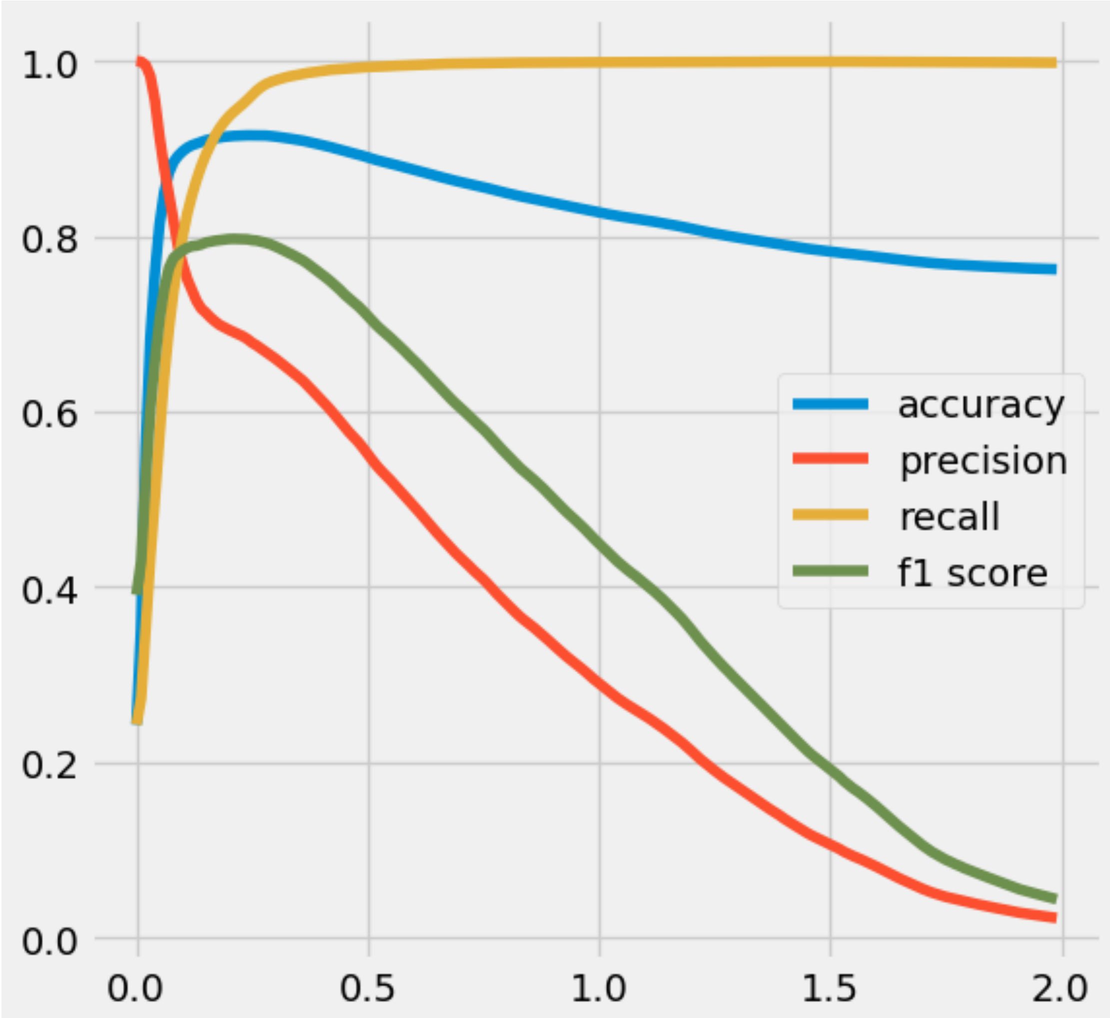
The validation set includes anomalies, and it is used to determine the optimal threshold for anomaly detection. This section displays the histogram of loss values for data points in the validation set. Additionally, the Precision-Recall (PR) curve for the validation data is presented, helping identify the threshold that maximizes the F1 score.

```
# visualize the losses for each data point and visualize the y
model.eval()
losses_val = []
for (d, y) in loader_val:

    d = d.reshape(-1, 23)

    # Output of Autoencoder
    reconstructed = model(d.cuda())

    # Calculating the loss function
```

```
        loss = loss_function(reconstructed.cuda(), d.cuda())

        # Storing the losses in a list for plotting
        losses_val.append((y[0], loss.item()))

losses_val_og = losses_val.copy()

# Find the optimal treshold
losses_val = losses_val_og.copy()
losses_val = np.array(losses_val)

step_size = 0.01
treshold = 0
accuracies = []
precisions = []
recalls = []
f1s = []
for i in range(200):
    y_pred = []
    for j in range(len(losses_val)):
        if losses_val[j][1] < treshold:
            y_pred.append(0)
        else:
            y_pred.append(1)
    y_pred = np.array(y_pred)
    y_true = losses_val[:, 0]
    # calculate the accuracy
    accuracy = (y_pred == y_true).sum() / len(y_true)
    accuracies.append(accuracy)
    # calculate the precision
    precision = (y_pred[y_true == 1] == y_true[y_true == 1]).sum() / len(y_true[y_
true == 1])
    precisions.append(precision)
    # calculate the recall
    recall = (y_pred[y_pred == 1] == y_true[y_pred == 1]).sum() / len(y_true[y_pre
d == 1])
    recalls.append(recall)
    # calculate the f1 score
    f1 = 2 * (precision * recall) / (precision + recall)
    f1s.append(f1)


    treshold += step_size

losses_val = losses_val[losses_val[:, 1] < 0.4]

not_anomaly = np.where(losses_val == 0)[0]
anomaly = np.where(losses_val == 1)[0]
# plot the losses for each data point
fig, ax = plt.subplots(figsize=(6,6))

ax.hist(losses_val[anomaly][:, 1], bins=100, density=True, label="anomaly", alpha=
.6, color="red")
ax.hist(losses_val[not_anomaly][:, 1], bins=100, density=True, label="not_anomaly"
, alpha=.6, color="green")
# draw a vertical line at the threshold
```

```python
#ax.axvline(x=0.2, color='black', linestyle='--', label="threshold")


plt.title("Distribution of the Reconstruction Loss")
plt.legend()
plt.show()

# plot the accuracies, precisions, recalls and f1 scores
fig, ax = plt.subplots(figsize=(6,6))
# the x axis is the treshold
x = np.arange(0, 2, step_size)
ax.plot(x, accuracies, label="accuracy")
ax.plot(x, precisions, label="precision")
ax.plot(x, recalls, label="recall")
ax.plot(x, f1s, label="f1 score")
#ax.axvline(x=0.4, color='black', linestyle='--', label="threshold")

ax.legend()
plt.show()
```

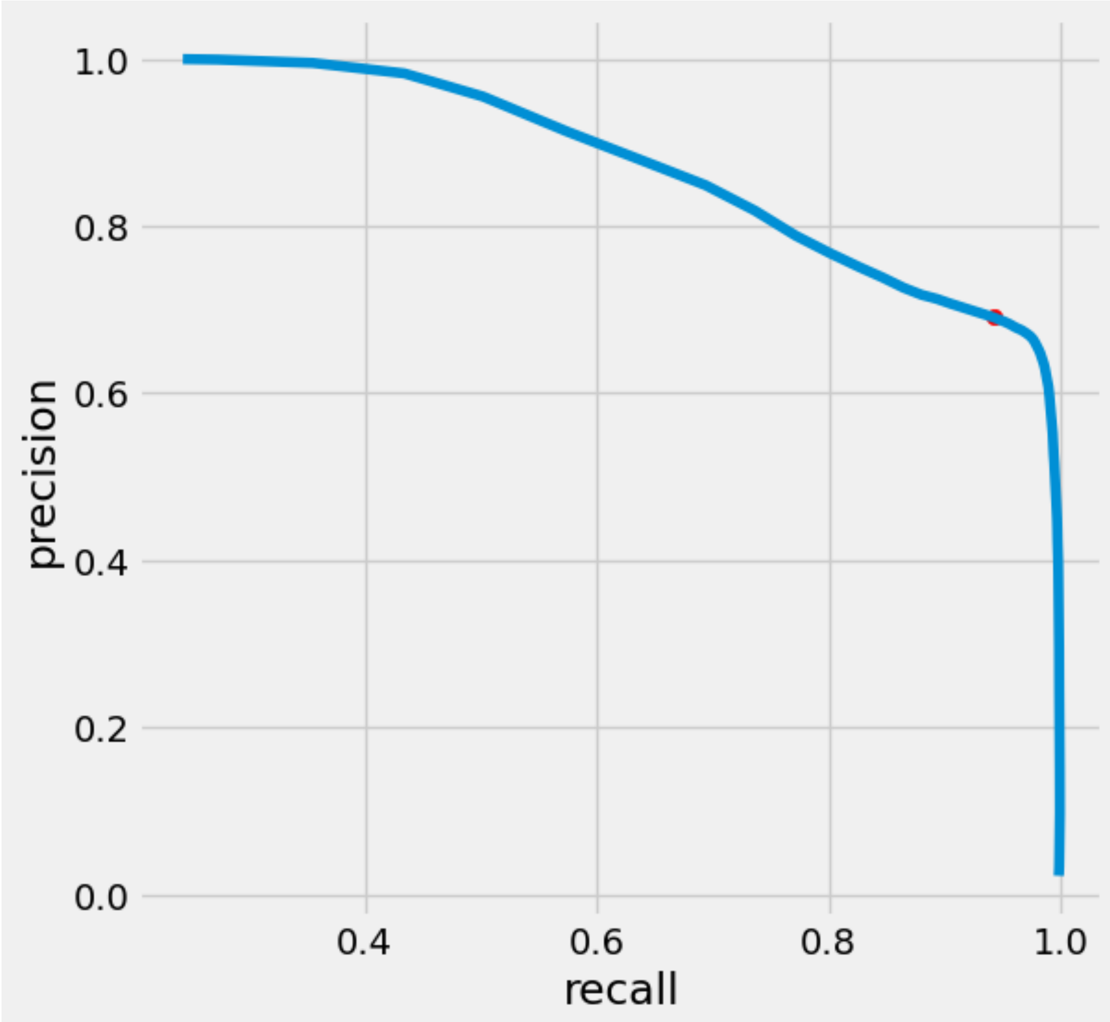




It is noticeable that there is a slight decrease in accuracy, whereas precision and F1 score experience more significant declines. Conversely, recall shows an increase.

```
# create a precison-recall curve
fig, ax = plt.subplots(figsize=(6,6))
ax.plot(recalls, precisions)
plt.xlabel("recall")
plt.ylabel("precision")
# mark the optimal treshold
ax.scatter(recalls[np.argmax(f1s)], precisions[np.argmax(f1s)], color="red")
plt.show()
```

```
optimal_treshold = np.argmax(f1s) * step_size
print("Optimal treshold:", np.argmax(f1s) * step_size)
```




```
Optimal treshold: 0.21
```

After evaluating these metrics we can then select the optimal threshold, which is set to 0.21.

Test Evaluation

The model is assessed on the test data using the threshold determined in the validation set. The histogram of loss values for data points in the test set is presented. This evaluation ensures the model's performance on previously unseen data, with a threshold unaffected by the test set.

```
#ids_test = [180, 181, 182, 183, 184, 185, 186, 187, 188, 189]


#filenames_test = [f'train_data_{num}.0.csv' for num in ids_test]

# create test dataset
dataset_test = TrainDataset(data_path='../train_datas/sncb_data_v4_preprocess2/',
files=filenames_test, train=False)

# DataLoader is used to load the dataset
# for testing
loader_test = torch.utils.data.DataLoader(dataset = dataset_test,
                                          batch_size = 1,
                                          shuffle = False)

print(len(dataset_test))
```

```
Number of rows: 882002
882002
```

```python
# visualize the losses for each data point and visualize the y
model.eval()
losses_test = []
for (d, y) in loader_test:
      # Reshaping the image to (-1, 784)
      d = d.reshape(-1, 23)

      # Output of Autoencoder
      reconstructed = model(d.cuda())

      # Calculating the loss function
      loss = loss_function(reconstructed.cuda(), d.cuda())

      # Storing the losses in a list for plotting
      losses_test.append((y[0], loss.item()))

losses_test_og = losses_test.copy()


losses_test = losses_test_og.copy()
#print((losses_test))
# convert losses to numpy

# normalize the losses
losses_test = np.array(losses_test)
losses_test = losses_test[losses_test[:, 1] <1.9]
#losses_test[:, 1] = (losses_test[:, 1] - losses_test[:, 1].min()) / (losses_test[
:, 1].max() - losses_test[:, 1].min())


not_anomaly = np.where(losses_test == 0)[0]
anomaly = np.where(losses_test == 1)[0]
# plot the losses for each data point
fig, ax = plt.subplots(figsize=(6,6))

ax.hist(losses_test[anomaly][:, 1], bins=100, density=True, label="anomaly", alpha
=.6, color="red")
ax.hist(losses_test[not_anomaly][:, 1], bins=100, density=True, label="not_anomaly
", alpha=.6, color="green")
# draw a vertical line at the threshold
ax.axvline(x=optimal_treshold, color='black', linestyle='--', label="threshold")


plt.title("Distribution of the Reconstruction Loss")
plt.legend()
plt.show()
```


```python
# classify the data points as anomaly or not anomaly with a loss treshold of 0.9
losses_test = losses_test_og.copy()
losses_test = np.array(losses_test)
#losses_test = losses_test[losses_test[:, 1] < 150]
#losses_test[:, 1] = (losses_test[:, 1] - losses_test[:, 1].min()) / (losses_test[
:, 1].max() - losses_test[:, 1].min())
t = optimal_treshold
y_pred = []
for i in range(len(losses_test)):
    if losses_test[i][1] > t:
        y_pred.append(1)
    else:
        y_pred.append(0)

# calculate the accuracy, precision, recall and f1 score
y_true = losses_test[:, 0]
y_pred = np.array(y_pred)
print("Accuracy:", np.sum(y_true == y_pred) / len(y_true))
print("Precision:", np.sum((y_true == y_pred) & (y_pred == 1)) / np.sum(y_pred ==
1))
print("Recall:", np.sum((y_true == y_pred) & (y_pred == 1)) / np.sum(y_true == 1))
print("F1 score:", 2 * np.sum((y_true == y_pred) & (y_pred == 1)) / (np.sum(y_true
== 1) + np.sum(y_pred == 1)))
```

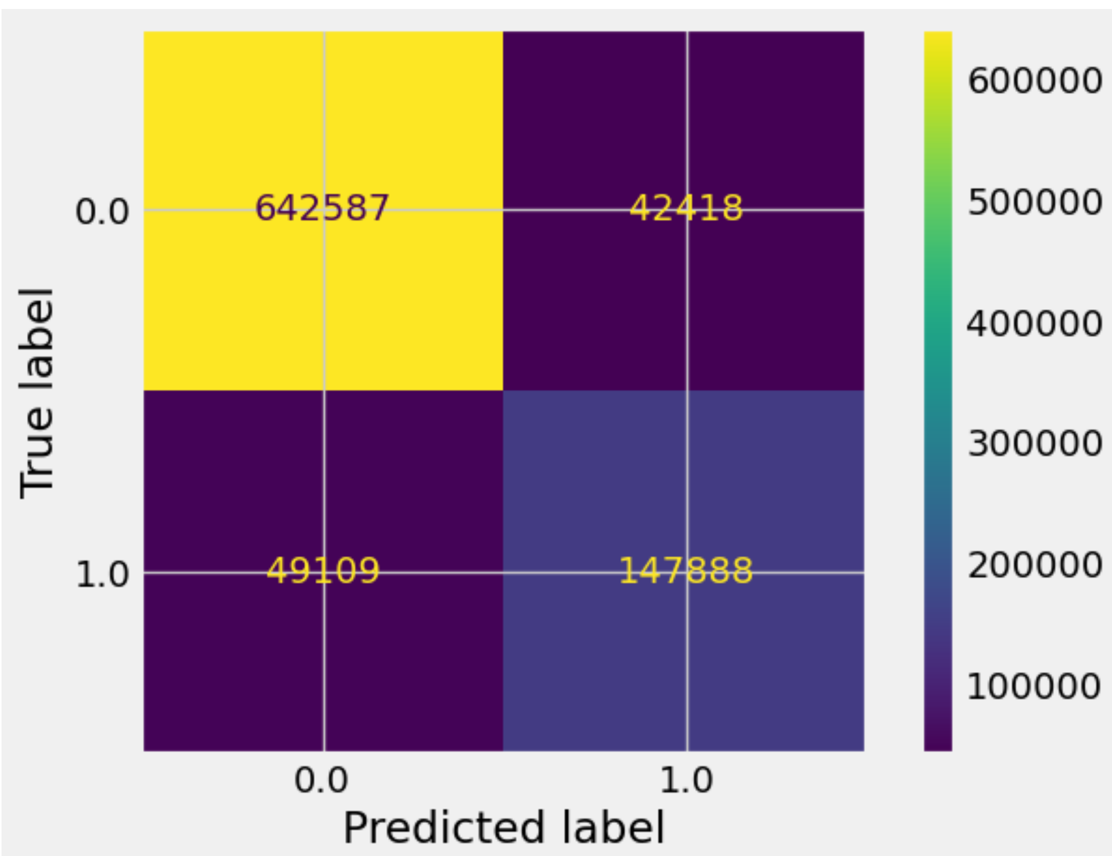
```
Accuracy: 0.8962281264668334
Precision: 0.7771063445188275
Recall: 0.7507119397757326
F1 score: 0.7636811488679406
```

The VAE demonstrates superior anomaly detection performance with an accuracy of 89.6%, precision at 77.7%, recall at 75.1%, and an F1 score of 76.4%.

A visual examination of the confusion matrix provides insights into the model's performance.

```
# make confusion matrix
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay
import seaborn as sns
import matplotlib.pyplot as plt

cm = confusion_matrix(y_true, y_pred)
print(cm)


ConfusionMatrixDisplay.from_predictions(y_true, y_pred)
```

```
[[642587  42418]
 [ 49109 147888]]

<sklearn.metrics._plot.confusion_matrix.ConfusionMatrixDisplay at 0x7f58647a9250>
```



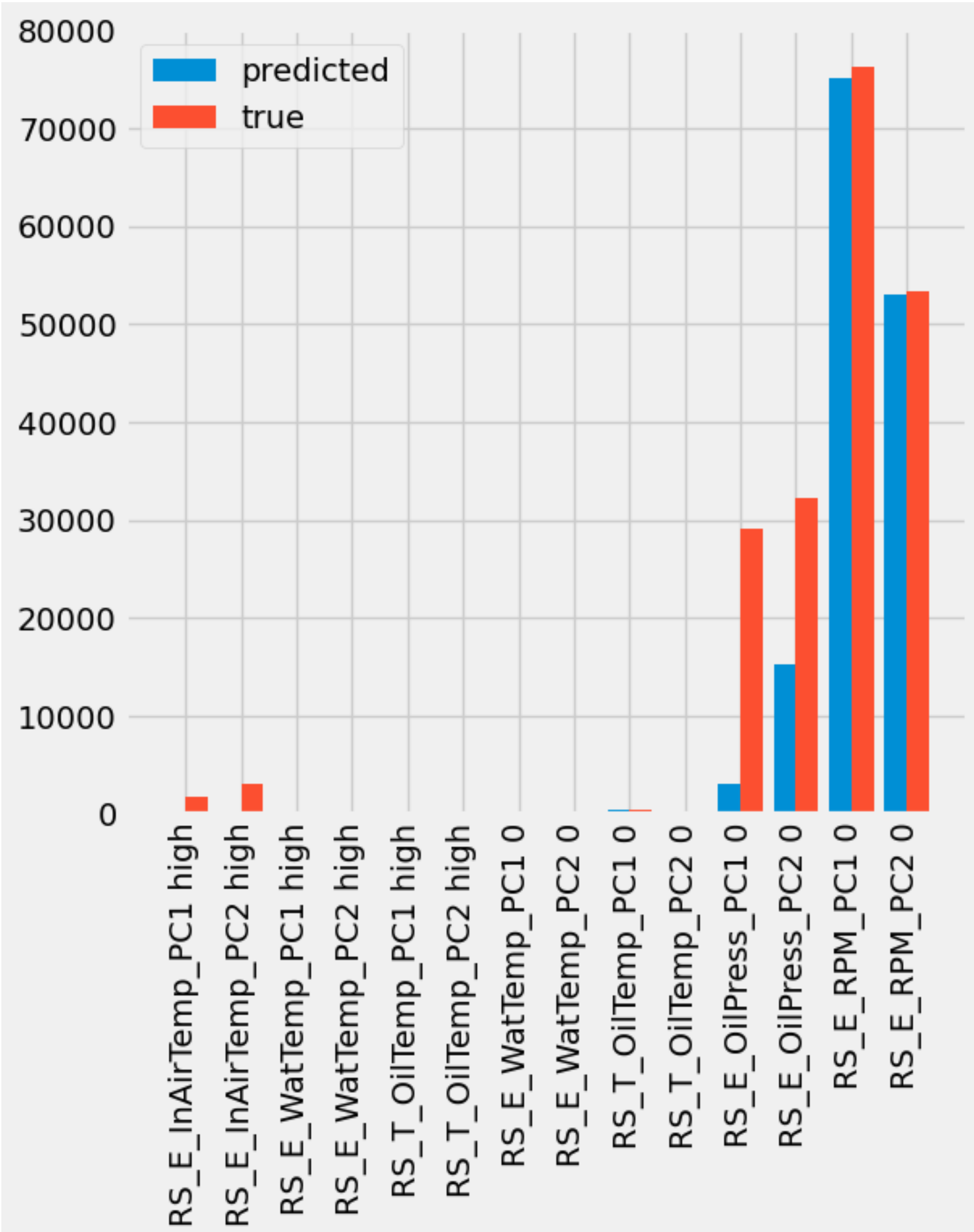
Considering VAEs as the optimal model in our exploration, we can visualize the predicted anomalies versus the true anomalies. The percentage of anomalies identified by the model aligns closely with the actual anomalies, except for the oil pressure. Detailed percentages are provided in the following output.

```
# save model
# get time
import datetime
del dataset
now = datetime.datetime.now()
# save model
torch.save(model.state_dict(), f"model_{now}.pt")

dataset_test = TrainDataset(data_path='../train_datas/sncb_data_v4_preprocess2/',
files=filenames_test, train=False, anomaly_per_class=True)

# DataLoader is used to load the dataset
# for testing
loader_test = torch.utils.data.DataLoader(dataset = dataset_test,
                                     batch_size = 1,
                                     shuffle = False)

print(len(dataset_test))
```

```
Number of rows: 882002
882002
```

```
y_true_labels = []
for (d, y) in loader_test:

      # Storing the losses in a list for plotting
      y_true_labels.append(y[0])

# get unique values in y_true
unique, counts_unique = np.unique(y_true, return_counts=True)
# create a bar plot to visualize the different type of anomalies that was found
counts = [0 for i in range(14)]

# get unique values in y_pred
unique, counts_unique = np.unique(y_true_labels, return_counts=True)

# if y_pred is 1 then we take the corresponding y_true and add 1 to the correspond
ing index in counts
for idx, i in enumerate(y_pred):
    if i == 1 and y_true_labels[idx] != 0:
        counts[int(y_true_labels[idx]) - 1 ] += 1

true_counts = [0 for i in range(14)]

# count the number of true anomalies
for i in range(1, 15):
    true_counts[i - 1] = y_true_labels.count(i)


labels = ['RS_E_InAirTemp_PC1 high', 'RS_E_InAirTemp_PC2 high', 'RS_E_WatTemp_PC1
high', 'RS_E_WatTemp_PC2 high', 'RS_T_OilTemp_PC1 high', 'RS_T_OilTemp_PC2 high',
                    'RS_E_WatTemp_PC1 0', 'RS_E_WatTemp_PC2 0', 'RS_T_OilTemp_PC1
0', 'RS_T_OilTemp_PC2 0', 'RS_E_OilPress_PC1 0', 'RS_E_OilPress_PC2 0', 'RS_E_RPM_
PC1 0', 'RS_E_RPM_PC2 0']
# plot the bar plot
fig, ax = plt.subplots(figsize=(6,6))
x = np.arange(14)
```

```
ax.bar(x-0.2, counts, 0.4, label="predicted")
ax.bar(x+0.2, true_counts, 0.4, label="true")
# add labels to the x axis
ax.set_xticks(x)
ax.set_xticklabels(labels, rotation=90)

ax.legend()
plt.show()

# calculate percentage of anomalies found
print("Percentage of anomalies found:", sum(counts) / sum(true_counts))

# percentage of anomalies found per type of anomaly, use the anomaly labels
print("Percentage of anomalies found per type of anomaly:")
for i in range(14):
    # solve division by 0
    if true_counts[i] == 0:
        print(labels[i], 0, '%')
    else:
        print(labels[i], counts[i] / true_counts[i] * 100, '%')
```



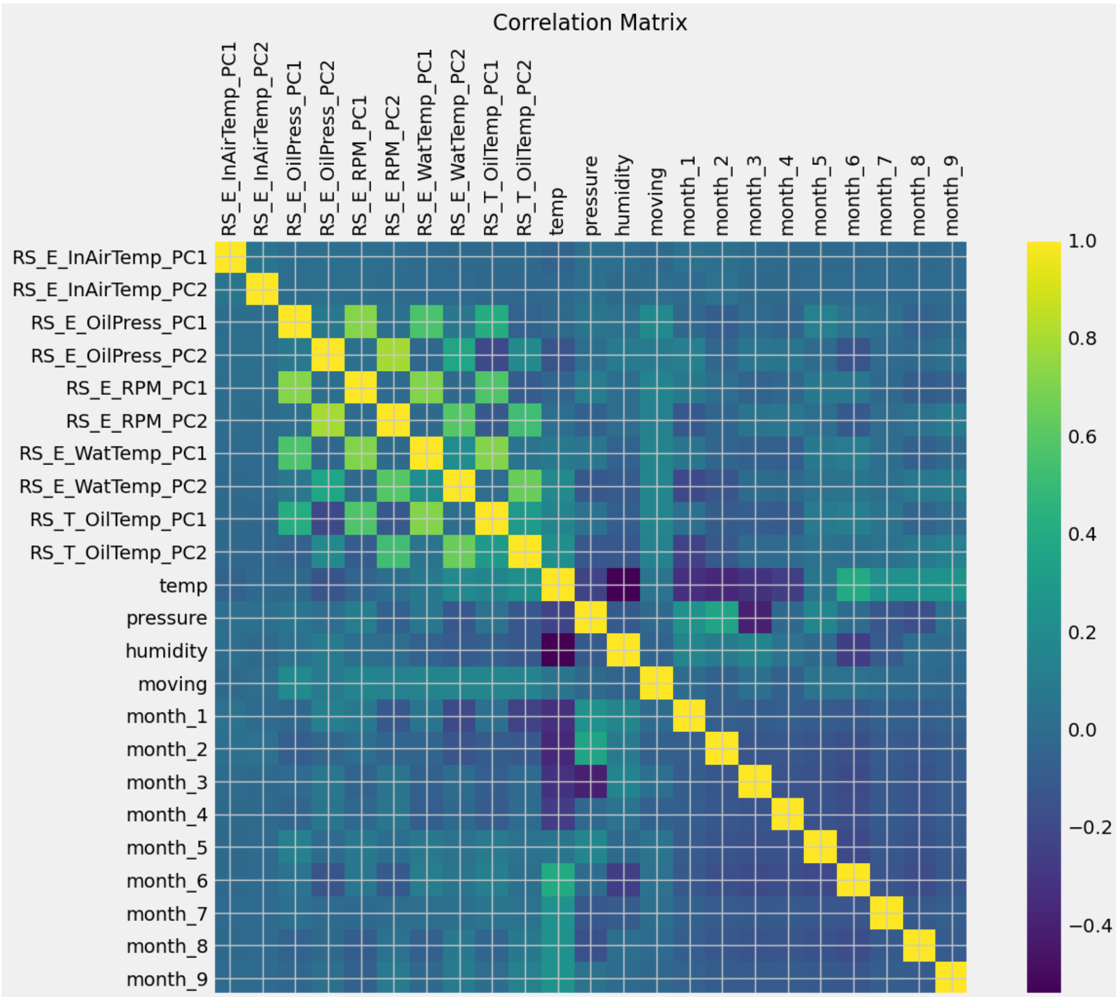

```
Percentage of anomalies found: 0.7507119397757326
Percentage of anomalies found per type of anomaly:
RS_E_InAirTemp_PC1 high 11.372549019607844 %
RS_E_InAirTemp_PC2 high 7.710280373831775 %
RS_E_WatTemp_PC1 high 37.62376237623762 %
RS_E_WatTemp_PC2 high 6.122448979591836 %
RS_T_OilTemp_PC1 high 100.0 %
RS_T_OilTemp_PC2 high 0 %
RS_E_WatTemp_PC1 0 96.62921348314607 %
RS_E_WatTemp_PC2 0 96.0 %
RS_T_OilTemp_PC1 0 93.82022471910112 %
RS_T_OilTemp_PC2 0 92.3780487804878 %
RS_E_OilPress_PC1 0 10.55774841256221 %
RS_E_OilPress_PC2 0 47.23413512337622 %
```

```
RS_E_RPM_PC1 0 98.51792640738023 %
RS_E_RPM_PC2 0 99.3786837857116 %
```

We can now examine the correlation matrix to identify the variables that impact the train-related parameters.

```
all_data, labels = dataset_test[:]

# filter the dataset where the y_pred is 1
anomalies_found = []
for i in range(len(all_data)):
       if y_pred[i] == 1:
              anomalies_found.append(all_data[i])

X_test_corr = anomalies_found
import seaborn as sns

# convert list to dataframe
X_test_corr = pd.DataFrame(np.vstack(X_test_corr))

# set the column names
X_test_corr.columns = ['RS_E_InAirTemp_PC1', 'RS_E_InAirTemp_PC2', 'RS_E_OilPress_
PC1',
       'RS_E_OilPress_PC2', 'RS_E_RPM_PC1', 'RS_E_RPM_PC2', 'RS_E_WatTemp_PC1',
       'RS_E_WatTemp_PC2', 'RS_T_OilTemp_PC1', 'RS_T_OilTemp_PC2', 'temp',
       'pressure', 'humidity', 'moving', 'month_1', 'month_2',
       'month_3', 'month_4', 'month_5', 'month_6', 'month_7', 'month_8',
       'month_9']

# correlation matrix for the anomalies validation set
corr_matrix = X_test_corr.corr()

# set the size of the figure
'''
plt.figure(figsize=(20, 15))

sns.heatmap(corr_matrix, annot=False)

# show the plot
plt.show()
'''


f = plt.figure(figsize=(15, 10))
# display the correlation matrix with numbers
plt.matshow(corr_matrix, fignum=f.number)


# set the x and y axis labels as the column names
plt.xticks(range(X_test_corr.shape[1]), X_test_corr.columns, fontsize=14, rotation
=90)
plt.yticks(range(X_test_corr.shape[1]), X_test_corr.columns, fontsize=14)
# display the values in the heatmap
```

```
#plt.xticks(range(df.select_dtypes(['number']).shape[1]), df.select_dtypes(['numbe
r']).columns, fontsize=14, rotation=45)
#plt.yticks(range(df.select_dtypes(['number']).shape[1]), df.select_dtypes(['numbe
r']).columns, fontsize=14)
cb = plt.colorbar()
cb.ax.tick_params(labelsize=14)
plt.title('Correlation Matrix', fontsize=16);
```



By selecting a cutoff that we arbitrarely set at 0.15, we visualize more clearly the variables that influence the train variables.

```
# List of PC1 motor variables
var_pc1 = [
    'RS_E_InAirTemp_PC1', 'RS_E_OilPress_PC1', 'RS_E_RPM_PC1', 'RS_E_WatTemp_PC1',
'RS_T_OilTemp_PC1'
]

# List of PC2 motor variables
var_pc2 = [
    'RS_E_InAirTemp_PC2', 'RS_E_OilPress_PC2', 'RS_E_RPM_PC2', 'RS_E_WatTemp_PC2',
'RS_T_OilTemp_PC2'
]

# List of all variables (motor and meteorological)
all_variables = var_pc1 + var_pc2 + [
    'temp', 'pressure', 'humidity', 'moving'
```

```
]

corr_matrix = X_test_corr[all_variables].corr()

# Specify the cutoff for the absolute correlation value (e.g., 0.15)
correlation_cutoff = 0.15

meteorological_correlations_pc1 = corr_matrix[corr_matrix.index.isin(var_pc1)][['t
emp', 'pressure', 'humidity', 'moving']]

meteorological_correlations_pc2 = corr_matrix[corr_matrix.index.isin(var_pc2)][['t
emp', 'pressure', 'humidity', 'moving']]

# Print meteorological variables strongly correlated with engine PC1
if not meteorological_correlations_pc1.empty:
    print("Variables strongly correlated with engine PC1:")
    print(meteorological_correlations_pc1[
        (meteorological_correlations_pc1.abs() >= correlation_cutoff)
    ])
else:
    print("No strong correlations found for engine PC1.")

# Print meteorological variables strongly correlated with engine PC2
if not meteorological_correlations_pc2.empty:
    print("\nVariables strongly correlated with engine PC2:")
    print(meteorological_correlations_pc2[
        (meteorological_correlations_pc2.abs() >= correlation_cutoff)
    ])
else:
    print("No strong correlations found for engine PC2.")

# Print the names of meteorological variables strongly correlated with engine PC1
if not meteorological_correlations_pc1.empty:
    print("Variables strongly correlated with engine PC1:")
    correlated_variables_pc1 = meteorological_correlations_pc1.columns[
        (meteorological_correlations_pc1.abs() >= correlation_cutoff).any(axis=0)
    ]
    print(correlated_variables_pc1)
else:
    print("No strong correlations found for engine PC1.")

# Print the names of meteorological variables strongly correlated with engine PC2
if not meteorological_correlations_pc2.empty:
    print("\nVariables strongly correlated with engine PC2:")
    correlated_variables_pc2 = meteorological_correlations_pc2.columns[
        (meteorological_correlations_pc2.abs() >= correlation_cutoff).any(axis=0)
    ]
    print(correlated_variables_pc2)
else:
    print("No strong correlations found for engine PC2.")
```

```
Variables strongly correlated with engine PC1:
                        temp  pressure  humidity    moving
RS_E_InAirTemp_PC1       NaN       NaN       NaN       NaN
RS_E_OilPress_PC1        NaN       NaN       NaN  0.189223
RS_E_RPM_PC1             NaN       NaN       NaN  0.155173
```

```
RS_E_WatTemp_PC1          NaN       NaN       NaN       NaN
RS_T_OilTemp_PC1     0.154523       NaN       NaN  0.159791

Variables strongly correlated with engine PC2:
                         temp  pressure  humidity    moving
RS_E_InAirTemp_PC2        NaN       NaN       NaN       NaN
RS_E_OilPress_PC2         NaN       NaN       NaN       NaN
RS_E_RPM_PC2              NaN       NaN       NaN       NaN
RS_E_WatTemp_PC2     0.185860       NaN       NaN  0.159329
RS_T_OilTemp_PC2     0.228228       NaN       NaN       NaN
Variables strongly correlated with engine PC1:
Index(['temp', 'moving'], dtype='object')

Variables strongly correlated with engine PC2:
Index(['temp', 'moving'], dtype='object')
```

While the correlations are not exceptionally strong, it can be inferred that the movement status of the train significantly influences PC1, whereas the variable 'temp' exerts a more pronounced influence on PC2.

Models conclusions

After a comprehensive analysis, it becomes evident that the Variational Autoencoder (VAE) stands out as the most effective model for anomaly detection on this dataset. The VAE exhibits an impressive overall performance, achieving an 81% F1 score, accompanied by notably high precision and slightly lower recall. This implies that the VAE provides a well-balanced trade-off between precision and recall, making it adept at minimizing both false positives and false negatives.

Contrastingly, the Isolation Forest showed a poor performance on both the validation and test sets. While the Support Vector Machine (SVM) displayed improvement, its drawback lies in escalating complexity when handling large datasets. Similar challenges in scalability and performance were observed with k-means and DBSCAN, both of which exhibited inferior results. Consequently, these latter models are deemed less suitable for consideration in the context of this anomaly detection task.

Streaming mode

This code attempts to simulate a data stream using Isolation Forest for anomaly detection. The idea is to incrementally train the model on an initial batch of data and then simulate real-time streaming of new instances. The `simulate_data_stream` function takes a DataFrame as input, initializes the Isolation Forest model, trains it on an initial batch, and then simulates the arrival of new instances. However, it seems that this code crashes the kernel, possibly due to memory issues or other runtime problems. Despite the challenges, this approach was considered appropriate to mimic a streaming scenario.

```
import pandas as pd
import numpy as np
from sklearn.ensemble import IsolationForest

def simulate_data_stream(df, model, initial_batch_size=100, num_instances=500, int
erval=50):
```

```python
    # Generator to simulate streaming
    def data_stream_generator():
        for _, row in df.iterrows():
            yield row.values  # Each row of the DataFrame becomes an instance in t
he stream

    # Initial training
    X_init = np.array([next(data_stream_generator()) for _ in range(initial_batch_
size)])
    y_init = np.zeros(initial_batch_size)  # 0 indicates normal data
    model.partial_fit(X_init, y_init)

    # List to save anomaly predictions
    anomaly_predictions = []

    # Simulate the real-time stream
    for i in range(num_instances):
        X = np.array([next(data_stream_generator())])  # Simulate receiving a new
data point

        # Adapt the model
        model.partial_fit(X, np.array([0]))  # Assume the data is normal (0)

        # Detect anomalies
        y_pred = model.predict(X)

        # Print intermediate results
        if (i + 1) % interval == 0:
            print(f"Instance {i+1}/{num_instances} - Anomaly Prediction: {y_pred}"
)

        # Save anomaly predictions
        anomaly_predictions.append(y_pred)

    return anomaly_predictions

# Initialize the model
model = IsolationForest()

# Simulate the data stream
simulate_data_stream(X_test, model)
```

```
The Kernel crashed while executing code in the the current cell or a previous cell
. Please review the code in the cell(s) to identify a possible cause of the failur
e. Click <a href='https://aka.ms/vscodeJupyterKernelCrash'>here</a> for more info.
View Jupyter <a href='command:jupyter.viewOutput'>log</a> for further details.
```