Predicting Machine Failure in advance

Project Workflow:

Machine failure prediction refers to the task of using machine learning and data analysis techniques to predict when a machine or equipment is likely to fail or experience a breakdown. By analyzing historical data and identifying patterns and indicators, machine failure prediction models can provide early warnings or alerts, enabling proactive maintenance and minimizing downtime.

Here is an overview of the process of machine failure predictions:

Data Collection: Relevant data is collected from the machines or equipment, such as sensor readings, operational parameters, maintenance records, and historical failure data. This data serves as the basis for training and building the predictive models.

Data Preprocessing: The collected data is cleaned, organized, and preprocessed to remove noise, handle missing values, and normalize the data. Feature engineering techniques may be applied to extract relevant features that capture patterns related to machine failures.

Feature Selection: Selecting the most informative features is crucial for building accurate prediction models. Various techniques, such as statistical analysis, correlation analysis, or domain knowledge, can be employed for feature selection.

Model Development: Machine learning algorithms, such as classification, regression, or time series analysis methods, are applied to train prediction models using the preprocessed data. The choice of algorithms depends on the nature of the data and the specific requirements of the prediction task.

Model Evaluation and Validation: The developed models are evaluated using suitable evaluation metrics to assess their performance and generalization capabilities. Cross-validation techniques may be employed to ensure robustness and reliability of the models.

Prediction and Maintenance Planning: Once the models are trained and validated, they can be used to predict machine failures in real-time. These predictions can help in scheduling preventive maintenance, optimizing resource allocation, and minimizing costly unplanned downtime.

By accurately predicting machine failures in advance, organizations can improve operational efficiency, reduce maintenance costs, enhance safety, and maximize the lifespan of their machines and equipment.

Step1: Importing the necessary Libraries and Dataset.

In [1]:
```
## Importing necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import datetime
import warnings
warnings.filterwarnings('ignore')
sns.set(style = "whitegrid",font_scale = 1.5)
%matplotlib inline
plt.rcParams['figure.figsize']=[12,8]
```

In [2]:
```
#importing Dataset
sensor_df = pd.read_csv('sensor.csv')
```

Step2: Data Wrangling

Data wrangling is the process of transforming and structuring data from one raw form into a desired format with the intent of improving data quality and making it more consumable and useful for analytics or machine learning.

In [3]:
```
print("The dataset has " , sensor_df.shape[0],"rows and", sensor_df.sh
```

```
The dataset has  220320 rows and 55 columns
```

In [4]:
```
#First 10 rows
sensor_df.head(10)
```

Out[4]:

	Unnamed: 0	timestamp	sensor_00	sensor_01	sensor_02	sensor_03	sensor_04	sensor_05
0	0	2018-04-01 00:00:00	2.465394	47.09201	53.2118	46.310760	634.3750	76.45975
1	1	2018-04-01 00:01:00	2.465394	47.09201	53.2118	46.310760	634.3750	76.45975
2	2	2018-04-01 00:02:00	2.444734	47.35243	53.2118	46.397570	638.8889	73.54598
3	3	2018-04-01 00:03:00	2.460474	47.09201	53.1684	46.397568	628.1250	76.98898
4	4	2018-04-01 00:04:00	2.445718	47.13541	53.2118	46.397568	636.4583	76.58897
5	5	2018-04-01 00:05:00	2.453588	47.09201	53.1684	46.397568	637.6157	78.18568
6	6	2018-04-01 00:06:00	2.455556	47.04861	53.1684	46.397568	633.3333	75.81614
7	7	2018-04-01 00:07:00	2.449653	47.13541	53.1684	46.397568	630.6713	75.77331
8	8	2018-04-01 00:08:00	2.463426	47.09201	53.1684	46.397568	631.9444	74.58916
9	9	2018-04-01 00:09:00	2.445718	47.17882	53.1684	46.397568	641.7823	74.57428

10 rows × 55 columns

In [5]:
```
# Last 10 rows
sensor_df.tail(10)
```

Out[5]:

	Unnamed: 0	timestamp	sensor_00	sensor_01	sensor_02	sensor_03	sensor_04	sens
220310	220310	2018-08-31 23:50:00	2.404398	47.61285	50.520830	43.142361	629.976800	65.2
220311	220311	2018-08-31 23:51:00	2.402431	47.69965	50.520832	43.142361	637.963000	64.4
220312	220312	2018-08-31 23:52:00	2.404398	47.61285	50.520832	43.142361	637.036987	65.5
220313	220313	2018-08-31 23:53:00	2.406366	47.69965	50.520830	43.142361	629.976807	63.7
220314	220314	2018-08-31 23:54:00	2.396528	47.61285	50.564240	43.142361	637.962952	62.8
220315	220315	2018-08-31 23:55:00	2.407350	47.69965	50.520830	43.142361	634.722229	64.5
220316	220316	2018-08-31 23:56:00	2.400463	47.69965	50.564240	43.142361	630.902771	65.8
220317	220317	2018-08-31 23:57:00	2.396528	47.69965	50.520830	43.142361	625.925903	67.2
220318	220318	2018-08-31 23:58:00	2.406366	47.69965	50.520832	43.142361	635.648100	65.0
220319	220319	2018-08-31 23:59:00	2.396528	47.69965	50.520832	43.142361	639.814800	65.4

10 rows × 55 columns

We have data for sensor readings from April to August, collected daily every minute.

In [6]:
```
sensor_df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 220320 entries, 0 to 220319
Data columns (total 55 columns):
 #   Column          Non-Null Count   Dtype  
---  ------          --------------   -----  
 0   Unnamed: 0      220320 non-null  int64  
 1   timestamp       220320 non-null  object 
 2   sensor_00       210112 non-null  float64
 3   sensor_01       219951 non-null  float64
 4   sensor_02       220301 non-null  float64
 5   sensor_03       220301 non-null  float64
 6   sensor_04       220301 non-null  float64
 7   sensor_05       220301 non-null  float64
 8   sensor_06       215522 non-null  float64
 9   sensor_07       214869 non-null  float64
 10  sensor_08       215213 non-null  float64
 11  sensor_09       215725 non-null  float64
 12  sensor_10       220301 non-null  float64
 13  sensor_11       220301 non-null  float64
 14  sensor_12       220301 non-null  float64
 15  sensor_13       220301 non-null  float64
 16  sensor_14       220299 non-null  float64
 17  sensor_15       0 non-null       float64
 18  sensor_16       220289 non-null  float64
 19  sensor_17       220274 non-null  float64
 20  sensor_18       220274 non-null  float64
 21  sensor_19       220304 non-null  float64
 22  sensor_20       220304 non-null  float64
 23  sensor_21       220304 non-null  float64
 24  sensor_22       220279 non-null  float64
 25  sensor_23       220304 non-null  float64
 26  sensor_24       220304 non-null  float64
 27  sensor_25       220284 non-null  float64
 28  sensor_26       220300 non-null  float64
 29  sensor_27       220304 non-null  float64
 30  sensor_28       220304 non-null  float64
 31  sensor_29       220248 non-null  float64
 32  sensor_30       220059 non-null  float64
 33  sensor_31       220304 non-null  float64
 34  sensor_32       220252 non-null  float64
 35  sensor_33       220304 non-null  float64
 36  sensor_34       220304 non-null  float64
 37  sensor_35       220304 non-null  float64
 38  sensor_36       220304 non-null  float64
 39  sensor_37       220304 non-null  float64
 40  sensor_38       220293 non-null  float64
 41  sensor_39       220293 non-null  float64
 42  sensor_40       220293 non-null  float64
 43  sensor_41       220293 non-null  float64
 44  sensor_42       220293 non-null  float64
 45  sensor_43       220293 non-null  float64
 46  sensor_44       220293 non-null  float64
 47  sensor_45       220293 non-null  float64
 48  sensor_46       220293 non-null  float64
 49  sensor_47       220293 non-null  float64
 50  sensor_48       220293 non-null  float64
 51  sensor_49       220293 non-null  float64
 52  sensor_50       143303 non-null  float64
 53  sensor_51       204937 non-null  float64
 54  machine_status  220320 non-null  object 
```

```
dtypes: float64(52), int64(1), object(2)
memory usage: 92.5+ MB
```

The data set consists of 51 numerical features ,timestamp and a categorical label.

The label contains string values that represent normal, broken and recovering operational conditions of the machine.

In [7]:
```
#statistical analysis of the dataset:Mean, Standard Deviation
sensor_df.describe().T
```

Out[7]:

	count	mean	std	min	25%	50%	
Unnamed: 0	220320.0	110159.500000	63601.049991	0.000000	55079.750000	110159.500000	16
sensor_00	210112.0	2.372221	0.412227	0.000000	2.438831	2.456539	
sensor_01	219951.0	47.591611	3.296666	0.000000	46.310760	48.133678	
sensor_02	220301.0	50.867392	3.666820	33.159720	50.390620	51.649300	
sensor_03	220301.0	43.752481	2.418887	31.640620	42.838539	44.227428	
sensor_04	220301.0	590.673936	144.023912	2.798032	626.620400	632.638916	
sensor_05	220301.0	73.396414	17.298247	0.000000	69.976260	75.576790	
sensor_06	215522.0	13.501537	2.163736	0.014468	13.346350	13.642940	
sensor_07	214869.0	15.843152	2.201155	0.000000	15.907120	16.167530	
sensor_08	215213.0	15.200721	2.037390	0.028935	15.183740	15.494790	
sensor_09	215725.0	14.799210	2.091963	0.000000	15.053530	15.082470	
sensor_10	220301.0	41.470339	12.093519	0.000000	40.705260	44.291340	
sensor_11	220301.0	41.918319	13.056425	0.000000	38.856420	45.363140	
sensor_12	220301.0	29.136975	10.113935	0.000000	28.686810	32.515830	
sensor_13	220301.0	7.078858	6.901755	0.000000	1.538516	2.929809	
sensor_14	220299.0	376.860041	113.206382	32.409550	418.103250	420.106200	
sensor_15	0.0	NaN	NaN	NaN	NaN	NaN	
sensor_16	220289.0	416.472892	126.072642	0.000000	459.453400	462.856100	
sensor_17	220274.0	421.127517	129.156175	0.000000	454.138825	462.020250	
sensor_18	220274.0	2.303785	0.765883	0.000000	2.447542	2.533704	
sensor_19	220304.0	590.829775	199.345820	0.000000	662.768975	665.672400	
sensor_20	220304.0	360.805165	101.974118	0.000000	398.021500	399.367000	
sensor_21	220304.0	796.225942	226.679317	95.527660	875.464400	879.697600	
sensor_22	220279.0	459.792815	154.528337	0.000000	478.962600	531.855900	
sensor_23	220304.0	922.609264	291.835280	0.000000	950.922400	981.925000	
sensor_24	220304.0	556.235397	182.297979	0.000000	601.151050	625.873500	
sensor_25	220284.0	649.144799	220.865166	0.000000	693.957800	740.203500	
sensor_26	220300.0	786.411781	246.663608	43.154790	790.489575	861.869600	
sensor_27	220304.0	501.506589	169.823173	0.000000	448.297950	494.468450	
sensor_28	220304.0	851.690339	313.074032	4.319347	782.682625	967.279850	
sensor_29	220248.0	576.195305	225.764091	0.636574	518.947225	564.872500	
sensor_30	220059.0	614.596442	195.726872	0.000000	627.777800	668.981400	
sensor_31	220304.0	863.323100	283.544760	23.958330	839.062400	917.708300	
sensor_32	220252.0	804.283915	260.602361	0.240716	760.607475	878.850750	
sensor_33	220304.0	486.405980	150.751836	6.460602	489.761075	512.271750	
sensor_34	220304.0	234.971776	88.376065	54.882370	172.486300	226.356050	
sensor_35	220304.0	427.129817	141.772519	0.000000	353.176625	473.349350	

	count	mean	std	min	25%	50%
sensor_36	220304.0	593.033876	289.385511	2.260970	288.547575	709.668050
sensor_37	220304.0	60.787360	37.604883	0.000000	28.799220	64.295485
sensor_38	220293.0	49.655946	10.540397	24.479166	45.572910	49.479160
sensor_39	220293.0	36.610444	15.613723	19.270830	32.552080	35.416660
sensor_40	220293.0	68.844530	21.371139	23.437500	57.812500	66.406250
sensor_41	220293.0	35.365126	7.898665	20.833330	32.552080	34.895832
sensor_42	220293.0	35.453455	10.259521	22.135416	32.812500	35.156250
sensor_43	220293.0	43.879591	11.044404	24.479166	39.583330	42.968750
sensor_44	220293.0	42.656877	11.576355	25.752316	36.747684	40.509260
sensor_45	220293.0	43.094984	12.837520	26.331018	36.747684	40.219910
sensor_46	220293.0	48.018585	15.641284	26.331018	40.509258	44.849540
sensor_47	220293.0	44.340903	10.442437	27.199070	39.062500	42.534720
sensor_48	220293.0	150.889044	82.244957	26.331018	83.912030	138.020800
sensor_49	220293.0	57.119968	19.143598	26.620370	47.743060	52.662040
sensor_50	143303.0	183.049260	65.258650	27.488426	167.534700	193.865700
sensor_51	204937.0	202.699667	109.588607	27.777779	179.108800	197.338000

In [8]:
```
print('all class labels:',sensor_df['machine_status'].unique())
```

```
all class labels: ['NORMAL' 'BROKEN' 'RECOVERING']
```

The label data has 3 machine status values:

BROKEN represents machine is failed.

RECOVERING represents machine trying to recover from failed status.

NORMAL represents the machine is working in normal status.

In [9]:
```
#Machine status distribution

sensor_df['machine_status'].value_counts()
```

Out[9]:
```
NORMAL        205836
RECOVERING     14477
BROKEN             7
Name: machine_status, dtype: int64
```

Step 3: Data Preprocessing

We can already see that the data requires some cleaning, there are missing values, an empty column and a timestamp with an incorrect data type. So I will apply the following steps to tidy up the data set.

Remove redundant columns

Remove duplicates

Handle missing values

Convert Timestamp column which is of type object to datetime

In [10]:
```
#no values for sensor 15 and unnamed column is unnecessary,so I will d
sensor_df.drop(['sensor_15','Unnamed: 0'],inplace = True,axis=1)
sensor_df.head()
```

Out[10]:

	timestamp	sensor_00	sensor_01	sensor_02	sensor_03	sensor_04	sensor_05	sensor_06
0	2018-04-01 00:00:00	2.465394	47.09201	53.2118	46.310760	634.3750	76.45975	13.41146
1	2018-04-01 00:01:00	2.465394	47.09201	53.2118	46.310760	634.3750	76.45975	13.41146
2	2018-04-01 00:02:00	2.444734	47.35243	53.2118	46.397570	638.8889	73.54598	13.32465
3	2018-04-01 00:03:00	2.460474	47.09201	53.1684	46.397568	628.1250	76.98898	13.31742
4	2018-04-01 00:04:00	2.445718	47.13541	53.2118	46.397568	636.4583	76.58897	13.35359

5 rows × 53 columns

Next, let's handle the missing values and for that let's first see the columns that have missing values and see what percentage of the data is missing.

In [11]:
```
#check percentage of missing values for each column
(sensor_df.isnull().sum().sort_values(ascending=False)/len(sensor_df))
```

Out[11]:
```
sensor_50         34.956881
sensor_51          6.982117
sensor_00          4.633261
sensor_07          2.474129
sensor_08          2.317992
sensor_06          2.177741
sensor_09          2.085603
sensor_01          0.167484
sensor_30          0.118464
sensor_29          0.032680
sensor_32          0.030864
sensor_17          0.020879
sensor_18          0.020879
sensor_22          0.018609
sensor_25          0.016340
sensor_16          0.014070
sensor_49          0.012255
sensor_48          0.012255
sensor_47          0.012255
sensor_46          0.012255
sensor_45          0.012255
sensor_44          0.012255
sensor_43          0.012255
sensor_42          0.012255
sensor_41          0.012255
sensor_40          0.012255
sensor_39          0.012255
sensor_38          0.012255
sensor_14          0.009532
sensor_26          0.009078
sensor_03          0.008624
sensor_10          0.008624
sensor_13          0.008624
sensor_12          0.008624
sensor_11          0.008624
sensor_05          0.008624
sensor_04          0.008624
sensor_02          0.008624
sensor_36          0.007262
sensor_37          0.007262
sensor_28          0.007262
sensor_27          0.007262
sensor_31          0.007262
sensor_35          0.007262
sensor_24          0.007262
sensor_23          0.007262
sensor_34          0.007262
sensor_21          0.007262
sensor_20          0.007262
sensor_19          0.007262
sensor_33          0.007262
timestamp          0.000000
machine_status     0.000000
dtype: float64
```

In [12]:
```
#too many missing values in sensor 50 , so dropping that
sensor_df.drop('sensor_50',inplace = True,axis=1)
sensor_df.head()
```

Out[12]:

	timestamp	sensor_00	sensor_01	sensor_02	sensor_03	sensor_04	sensor_05	sensor_06
0	2018-04-01 00:00:00	2.465394	47.09201	53.2118	46.310760	634.3750	76.45975	13.41146
1	2018-04-01 00:01:00	2.465394	47.09201	53.2118	46.310760	634.3750	76.45975	13.41146
2	2018-04-01 00:02:00	2.444734	47.35243	53.2118	46.397570	638.8889	73.54598	13.32465
3	2018-04-01 00:03:00	2.460474	47.09201	53.1684	46.397568	628.1250	76.98898	13.31742
4	2018-04-01 00:04:00	2.445718	47.13541	53.2118	46.397568	636.4583	76.58897	13.35359

5 rows × 52 columns

I decided to impute some of the missing values with their mean

In [13]:
```
#imputing the remaining missing values with mean
sensor_df.fillna(sensor_df.mean(),inplace= True)
sensor_df.isnull().sum()
```

Out[13]:
```
timestamp         0
sensor_00         0
sensor_01         0
sensor_02         0
sensor_03         0
sensor_04         0
sensor_05         0
sensor_06         0
sensor_07         0
sensor_08         0
sensor_09         0
sensor_10         0
sensor_11         0
sensor_12         0
sensor_13         0
sensor_14         0
sensor_16         0
sensor_17         0
sensor_18         0
sensor_19         0
sensor_20         0
sensor_21         0
sensor_22         0
sensor_23         0
sensor_24         0
sensor_25         0
sensor_26         0
sensor_27         0
sensor_28         0
sensor_29         0
sensor_30         0
sensor_31         0
sensor_32         0
sensor_33         0
sensor_34         0
sensor_35         0
sensor_36         0
sensor_37         0
sensor_38         0
sensor_39         0
sensor_40         0
sensor_41         0
sensor_42         0
sensor_43         0
sensor_44         0
sensor_45         0
sensor_46         0
sensor_47         0
sensor_48         0
sensor_49         0
sensor_51         0
machine_status    0
dtype: int64
```

In [14]:
```
#checking of duplicate rows

sensor_df.duplicated().any()
```

Out[14]:
```
False
```

No duplicate values, hence we don't need to remove any row.

In [15]:
```
# Now, lets make it a time series and set it as index
sensor_df['timestamp'] = pd.to_datetime(sensor_df['timestamp'])
sensor_df = sensor_df.set_index('timestamp')
```

After the data wrangling process,my final dataset includes 52 sensors/features,datetime and machine status column that contains 3 classes that represent the NORMAL,BROKEN AND RECOVERING operating conditions of the pump.

The First 5 rows of the dataset looks as follows.

In [16]:
```
sensor_df.head()
```

Out[16]:

timestamp	sensor_00	sensor_01	sensor_02	sensor_03	sensor_04	sensor_05	sensor_06	ser
2018-04-01 00:00:00	2.465394	47.09201	53.2118	46.310760	634.3750	76.45975	13.41146	1(
2018-04-01 00:01:00	2.465394	47.09201	53.2118	46.310760	634.3750	76.45975	13.41146	1(
2018-04-01 00:02:00	2.444734	47.35243	53.2118	46.397570	638.8889	73.54598	13.32465	1(
2018-04-01 00:03:00	2.460474	47.09201	53.1684	46.397568	628.1250	76.98898	13.31742	1(
2018-04-01 00:04:00	2.445718	47.13541	53.2118	46.397568	636.4583	76.58897	13.35359	1(

5 rows × 51 columns

Step 4: Exploratory Data Analysis(EDA)

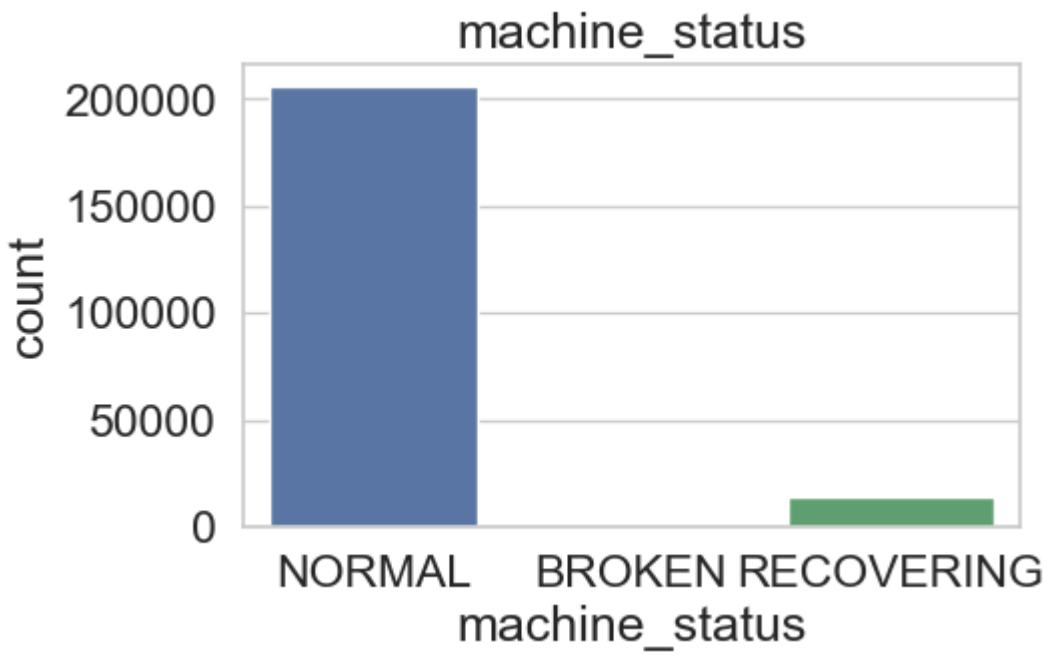
Now that I have cleaned the data, I can start exploring to acquaint with the data set.On top of some quantitative EDA, I performed additional graphical EDA to look for trends and any odd behaviors

In [17]:
```
## Machine status distribution
sensor_df.machine_status.value_counts()
```

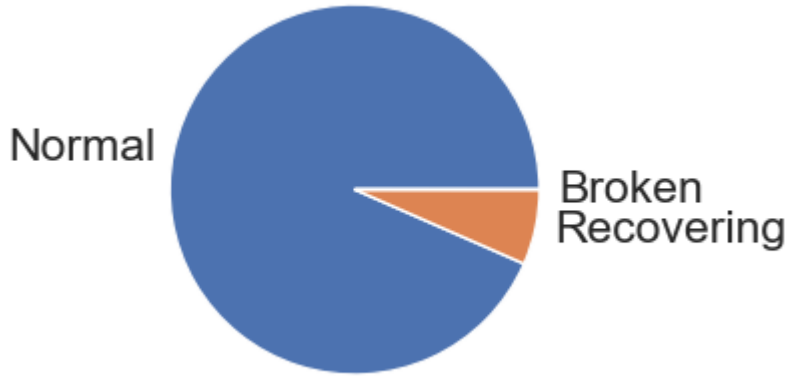
Out[17]:
```
NORMAL        205836
RECOVERING     14477
BROKEN             7
Name: machine_status, dtype: int64
```

In [18]:
```
#Plotting Machine Status Distribution
plt.figure(figsize=(5,3))
plt.title('machine_status')
sns.countplot(x='machine_status',data=sensor_df)
```

Out[18]:
```
<AxesSubplot:title={'center':'machine_status'}, xlabel='machine_stat
us', ylabel='count'>
```



In [19]:
```
#machine status -  pie chart
plt.figure(figsize=(5,3))
stroke_labels = ["Normal","Recovering","Broken"]
sizes = sensor_df.machine_status.value_counts()

plt.pie(x=sizes,labels=stroke_labels)
plt.show()
```



As we can see, majority of the time the sensors register NORMAL. This makes sense because the machine should be working normally most of the time. We then see the RECOVERING as the next group, which also makes sense given that the machine breaks then take a bit to recover. Lastly, only 0.004% of the data is in BROKEN status. This is okay because a machine breaks, and in theory, shouldn't be broken for very long.

Heatmap for Correlation

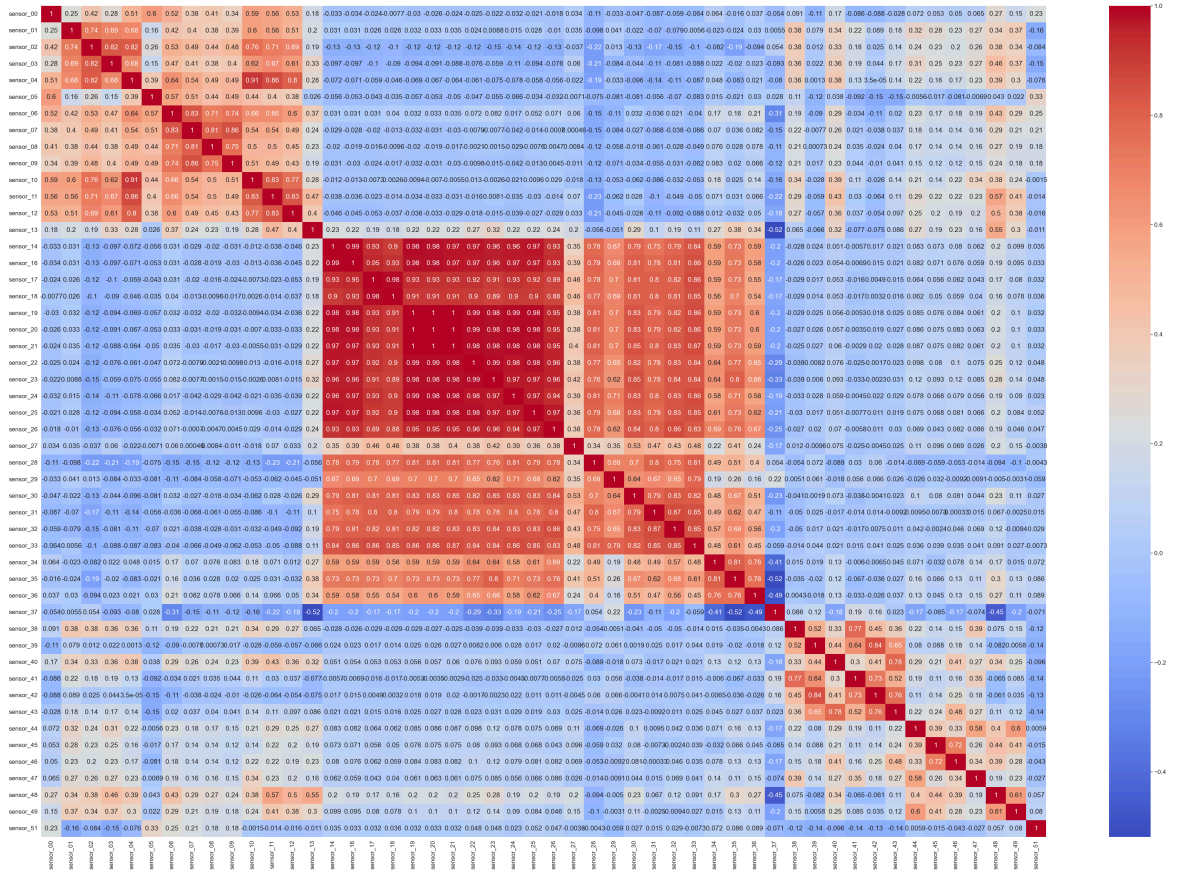
Correlation Anlaysis analysis the correlation between the input parameters (sensors) and determine which parameters are the most important, deciding on the output of the model.

A value of 1 means Positive Correlation

A value of 0 means No Correlation

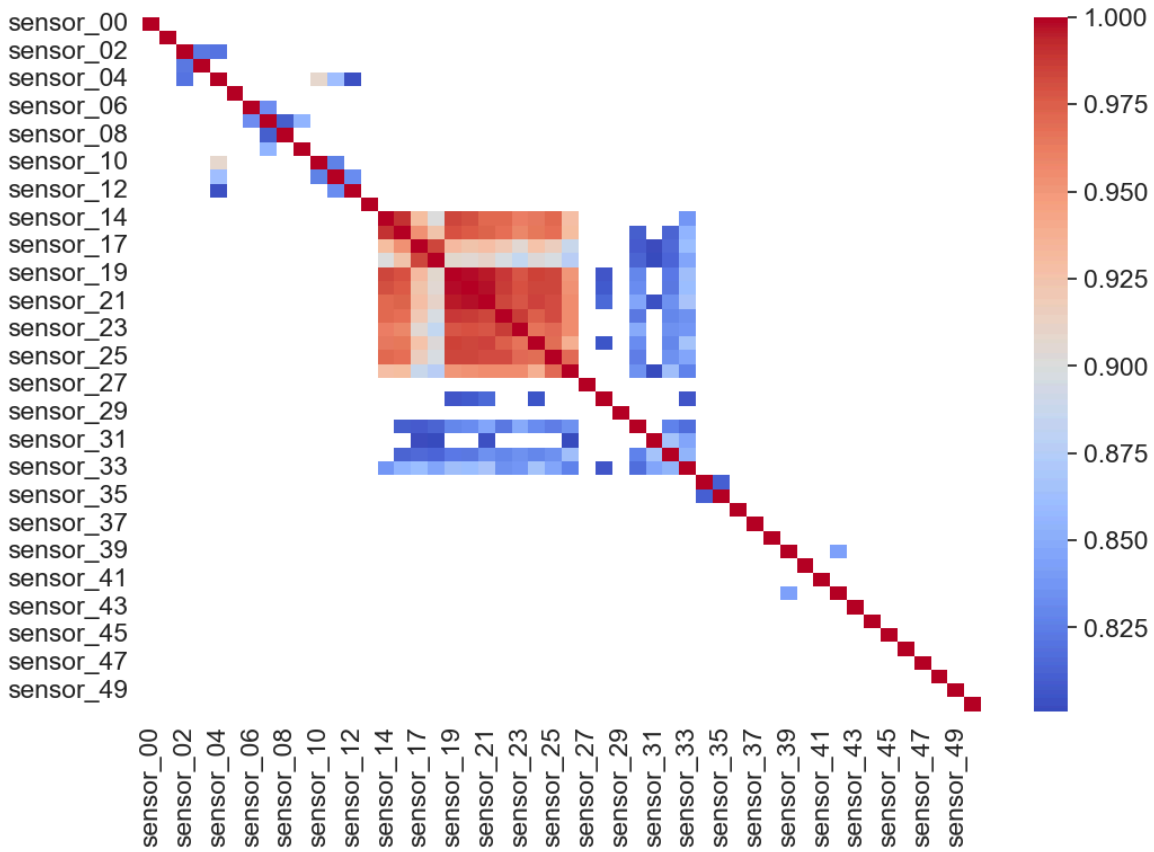
A value of -1 means Negative Correlation

In [20]:
```
plt.figure(figsize=(60,40))
sns.heatmap(sensor_df.corr(),annot=True,cmap='coolwarm');
corr = sensor_df.corr()
```



In [21]:
```
#The HeatMap shows Correlation between features greater than 0.8
corr80 = corr[abs(corr)> 0.8]
sns.heatmap(corr80,cmap='coolwarm')
```

Out[21]: <AxesSubplot:>

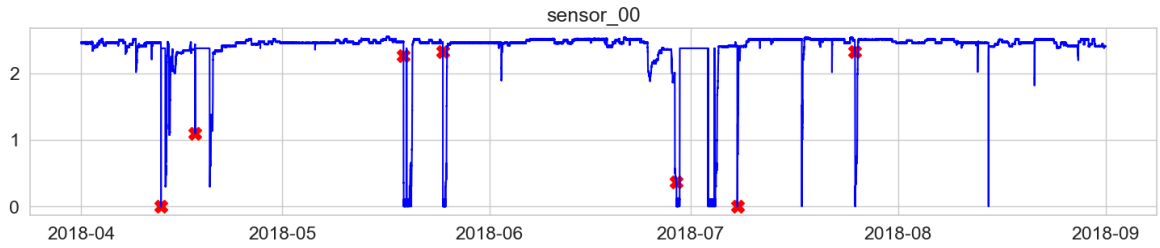


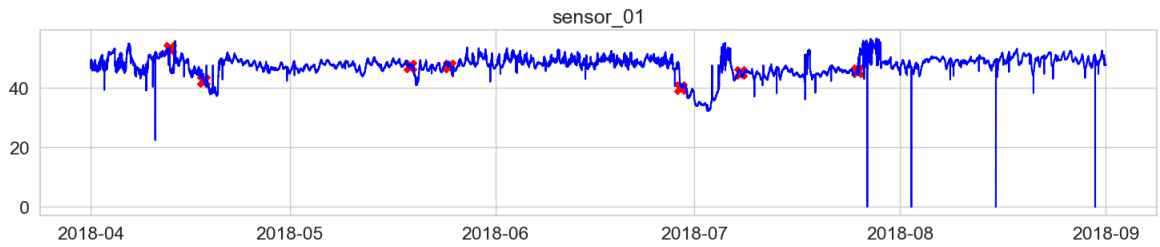
From the heatmap ,we see that sensor 14 to 29 have a high correlation with the target variable, hence will be important in predicting machine failure.

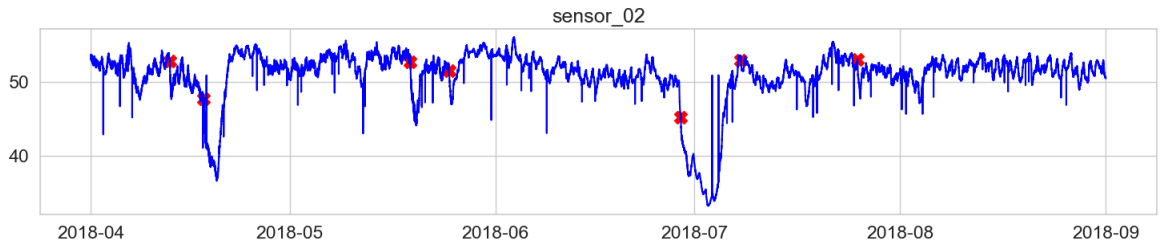
In [22]:
```
#sns.pairplot(sensor_df,hue='machine_status')

#plt.show()
```

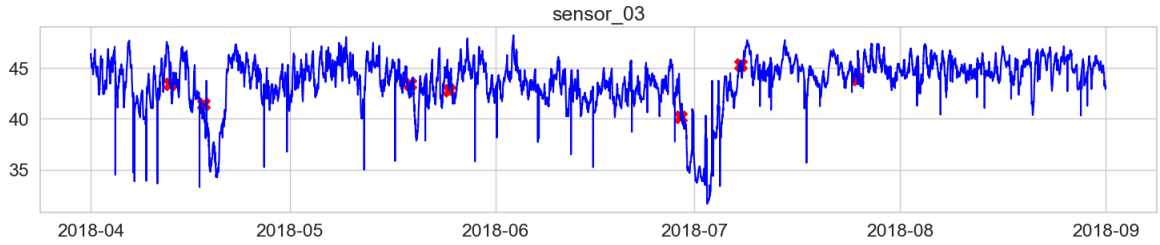
The plots below show the sensor readings plotted over time with the machine status of “BROKEN” marked up on the same graph in red color. That way, we can clearly see when the machine breaks down and how that reflects in the sensor readings. The following code plots the mentioned graph for each of the sensors.

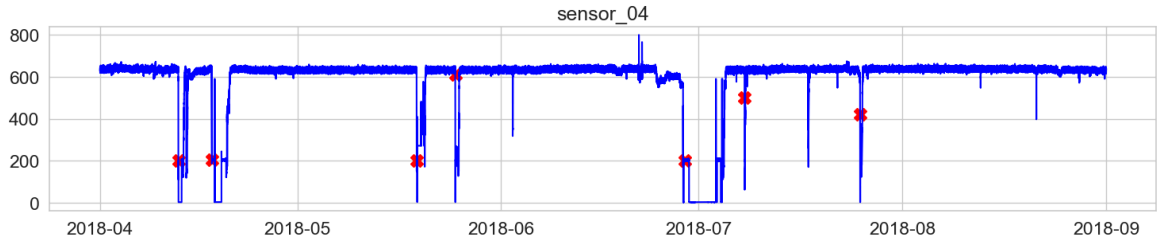
In [23]:
```
#extract the readings from the Broken state of the pump
broken= sensor_df[sensor_df['machine_status']=='BROKEN']

#Extract the name of the numerical columns
sensor_df_2 = sensor_df.drop(['machine_status'],axis=1)
names= sensor_df_2.columns

#plot timeseries for each sensor with Broken state marked with X in re

for name in names:
    _=plt.figure(figsize=(18,3))
    _=plt.plot(broken[name],linestyle='none',marker='X',color='red',ma
    _=plt.plot(sensor_df[name],color='blue')
    _=plt.title(name)
    plt.show()
```

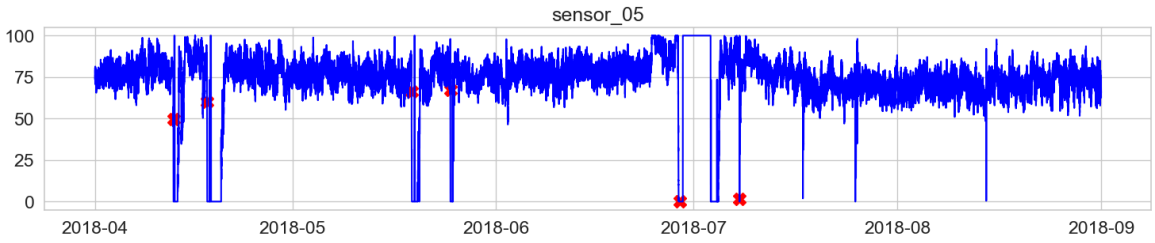


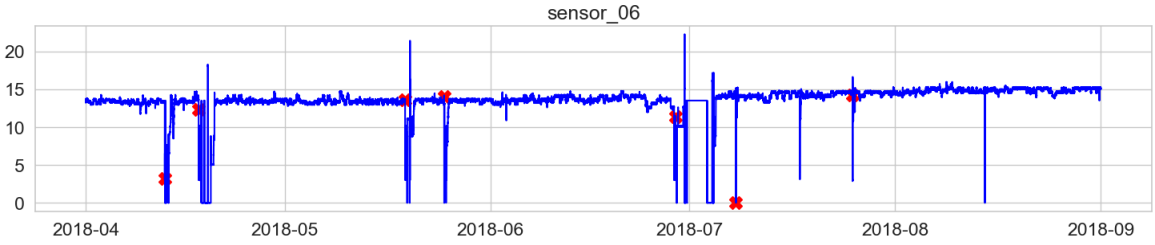


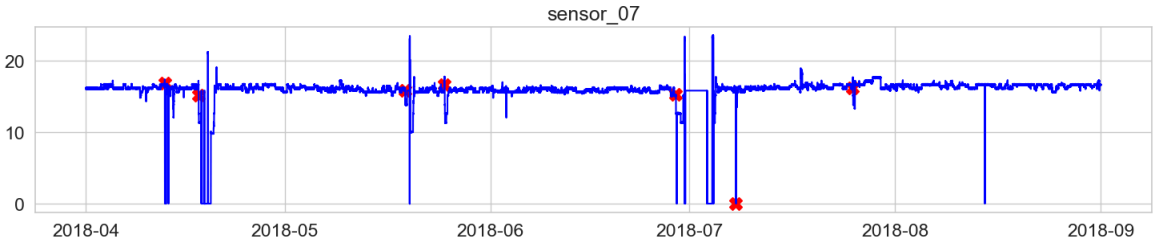


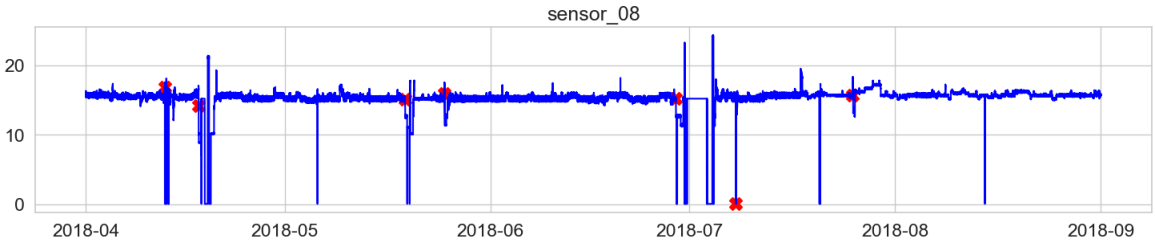


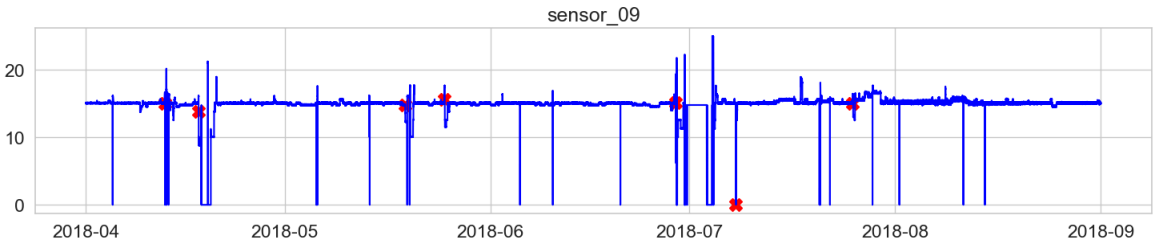


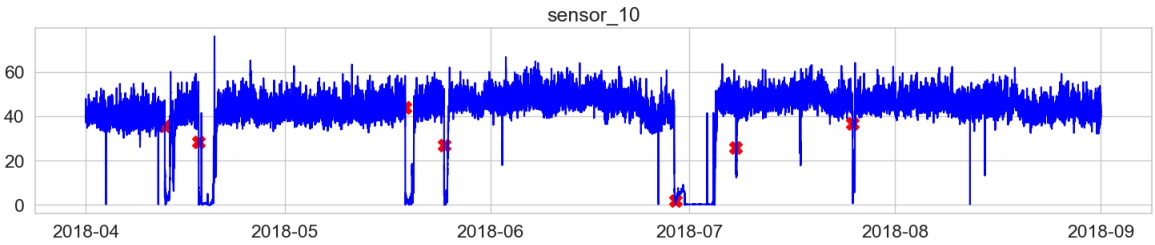

sensor_05
100
75
50
25
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09

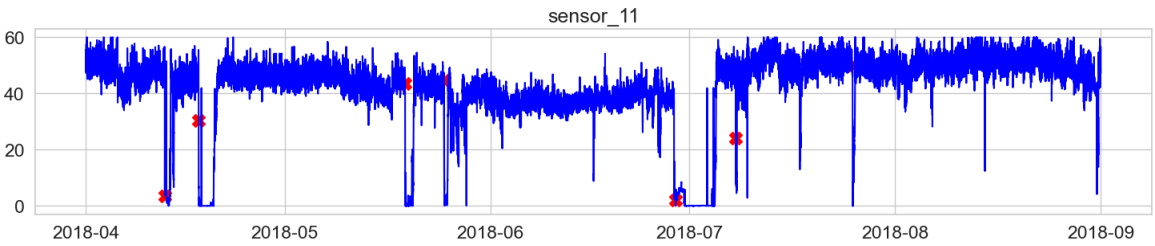

sensor_06
20
15
10
5
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09

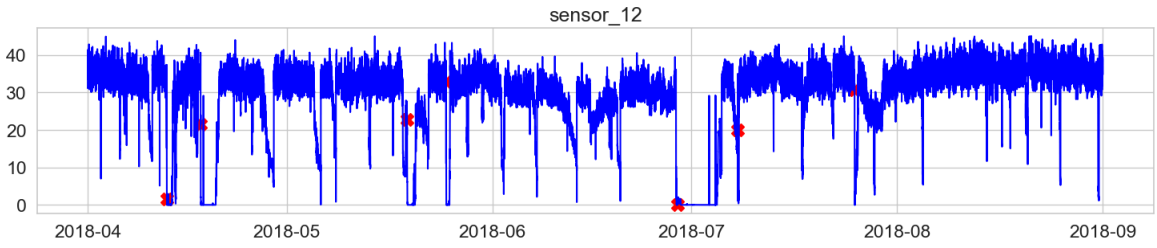

sensor_07
20
10
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09

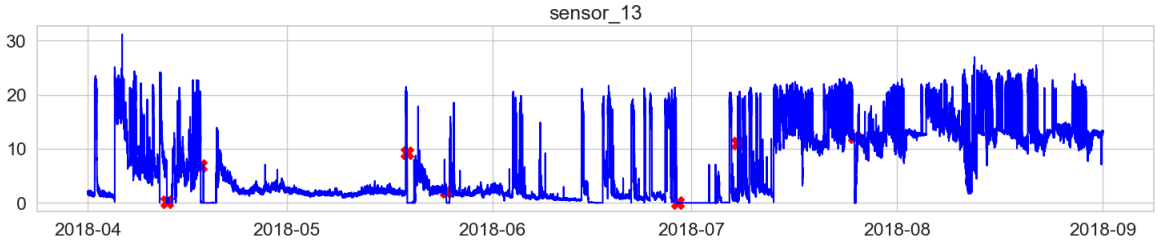

sensor_08
20
10
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09

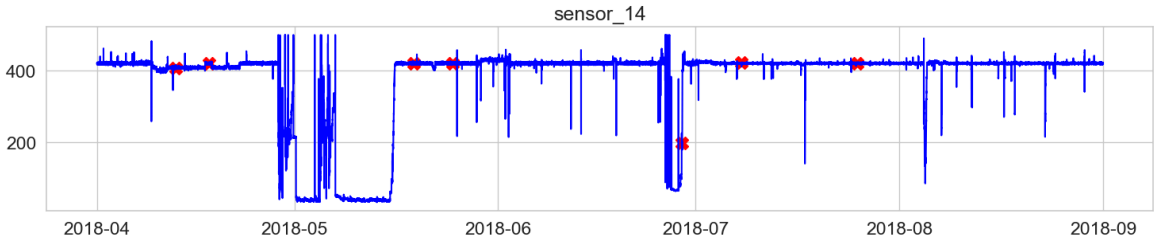

sensor_09
20
10
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09

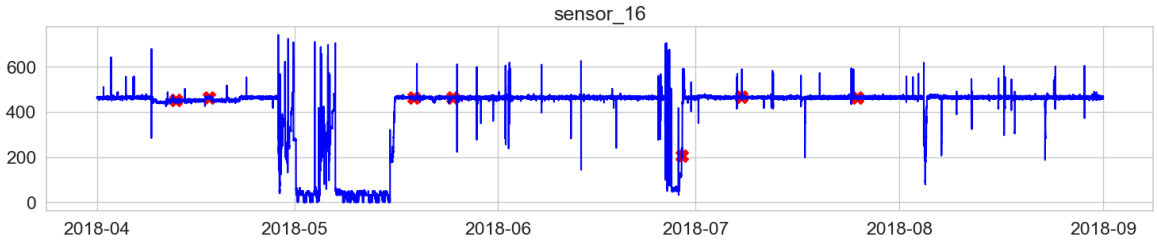

sensor_10
60
40
20
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09

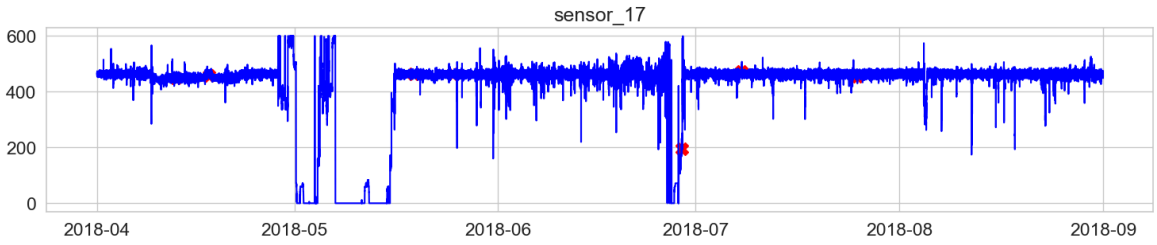

sensor_11
60
40
20
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09

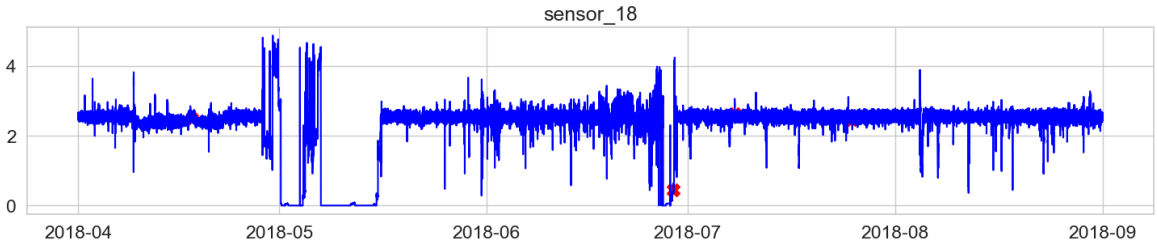

sensor_12
40
30
20
10
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09

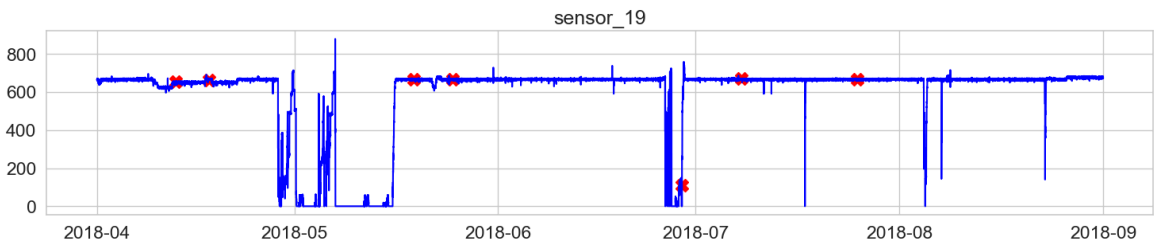

sensor_13
30
20
10
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09

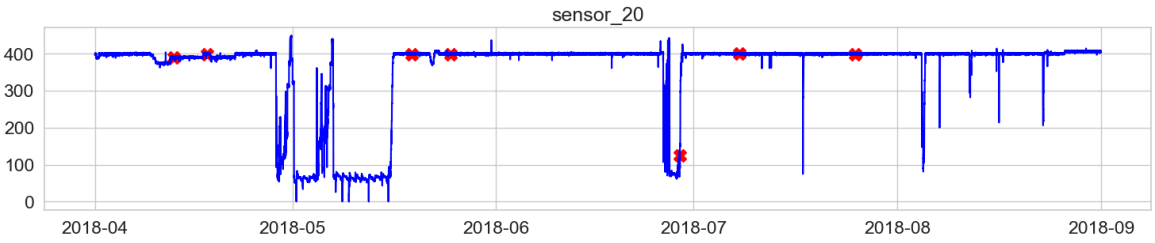

sensor_14
400
200
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09

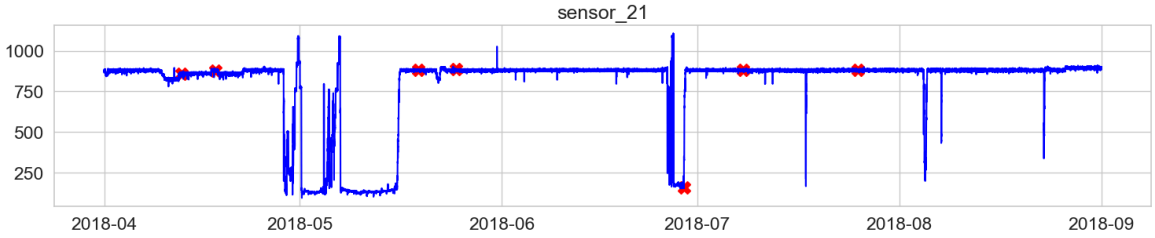

sensor_16
600
400
200
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09

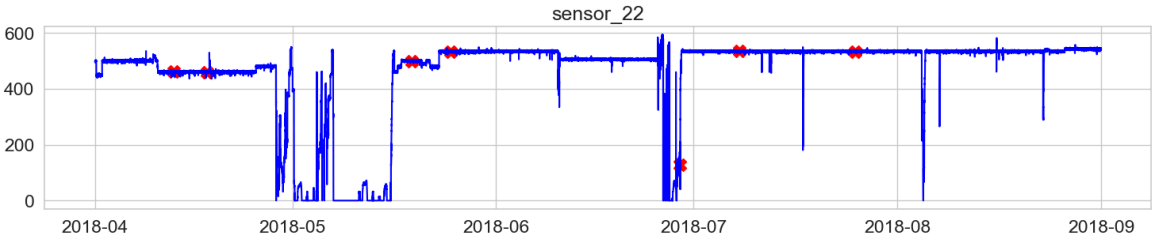

sensor_17
600
400
200
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09

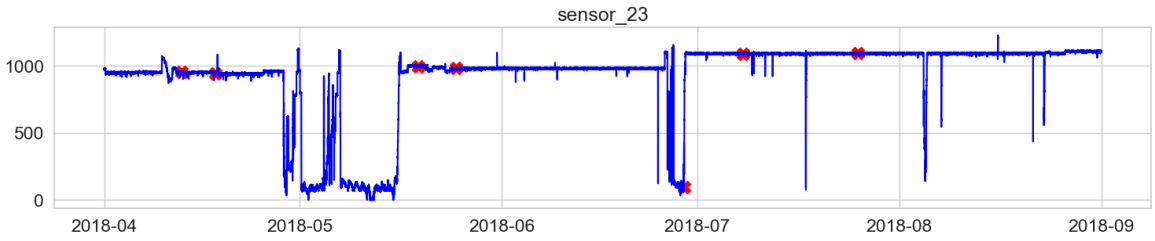

sensor_18
4
2
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09

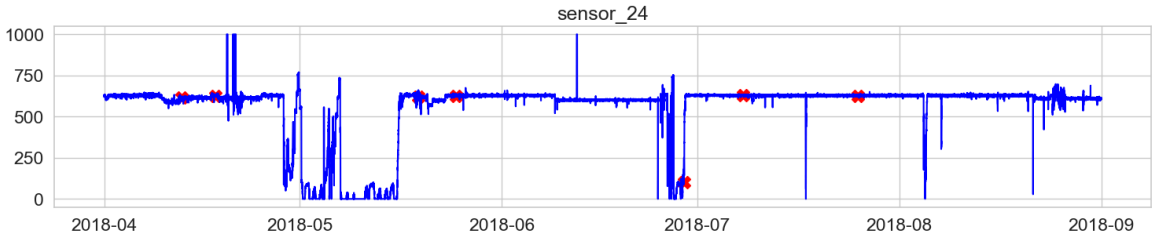

sensor_19
800
600
400
200
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09

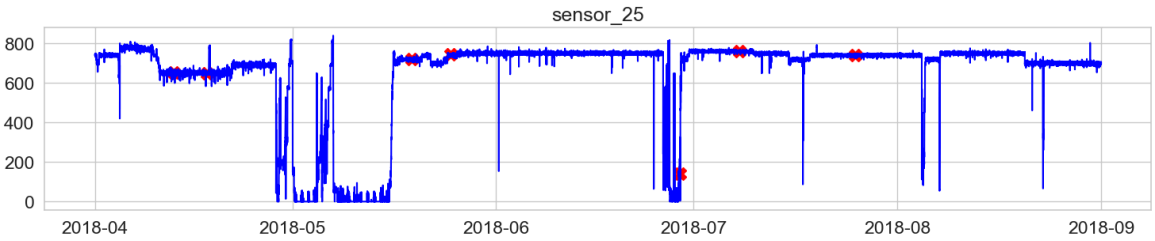

sensor_20
400
300
200
100
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09

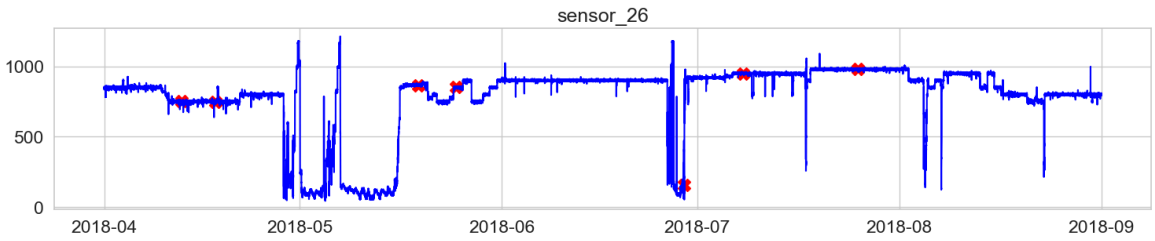

sensor_21
1000
750
500
250
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09

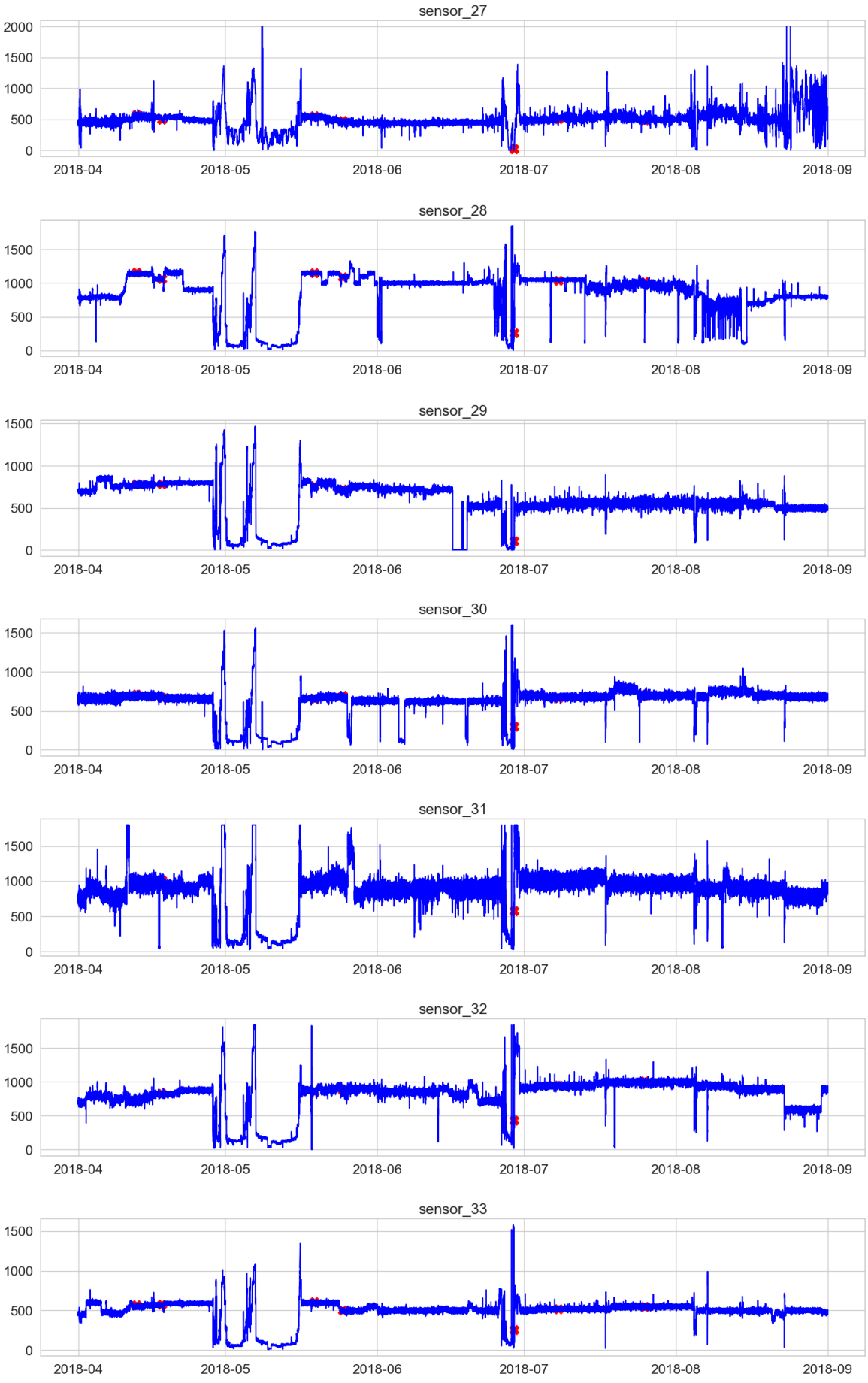

sensor_22
600
400
200
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09

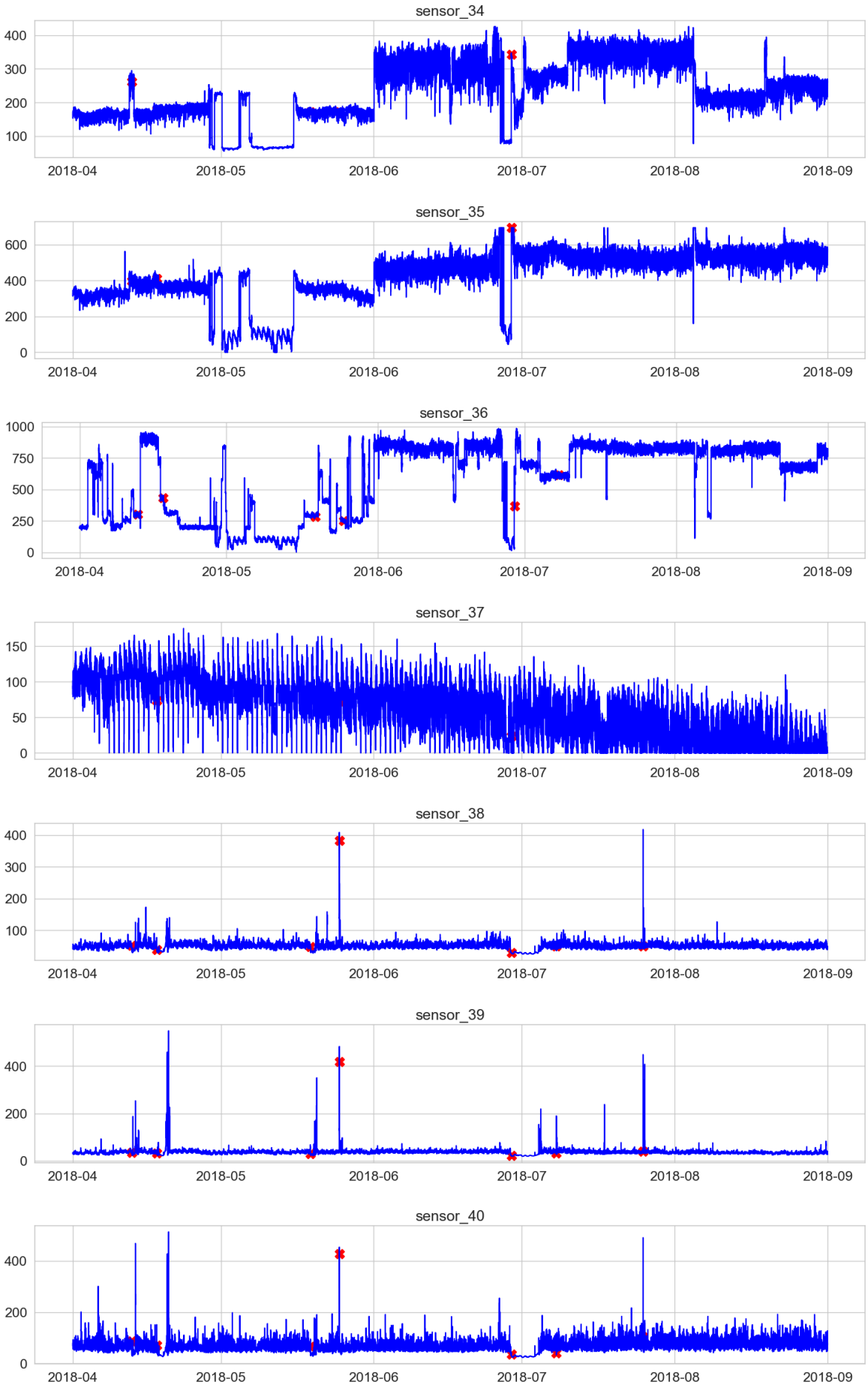

sensor_23
1000
500
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09

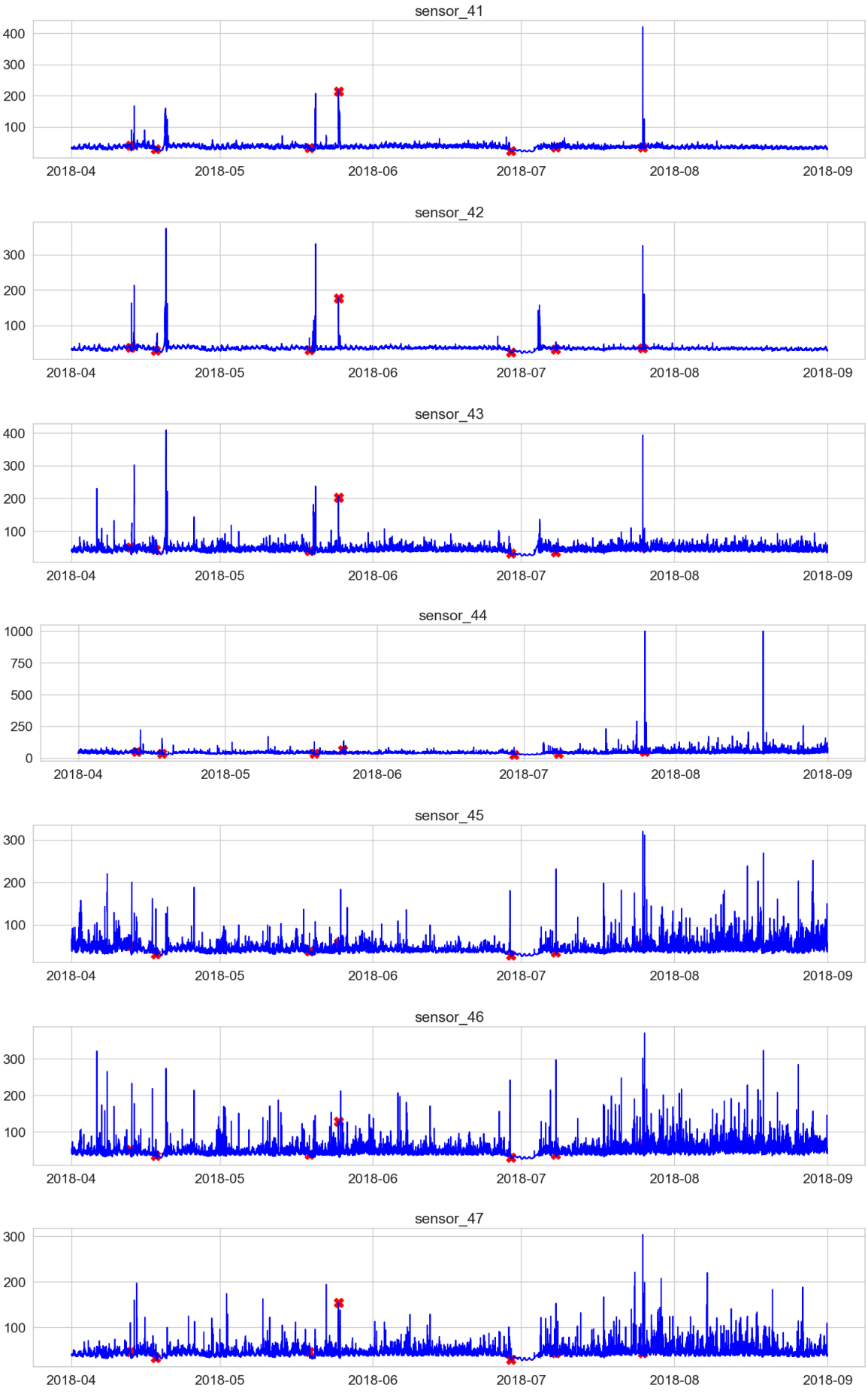

sensor_24
1000
750
500
250
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09

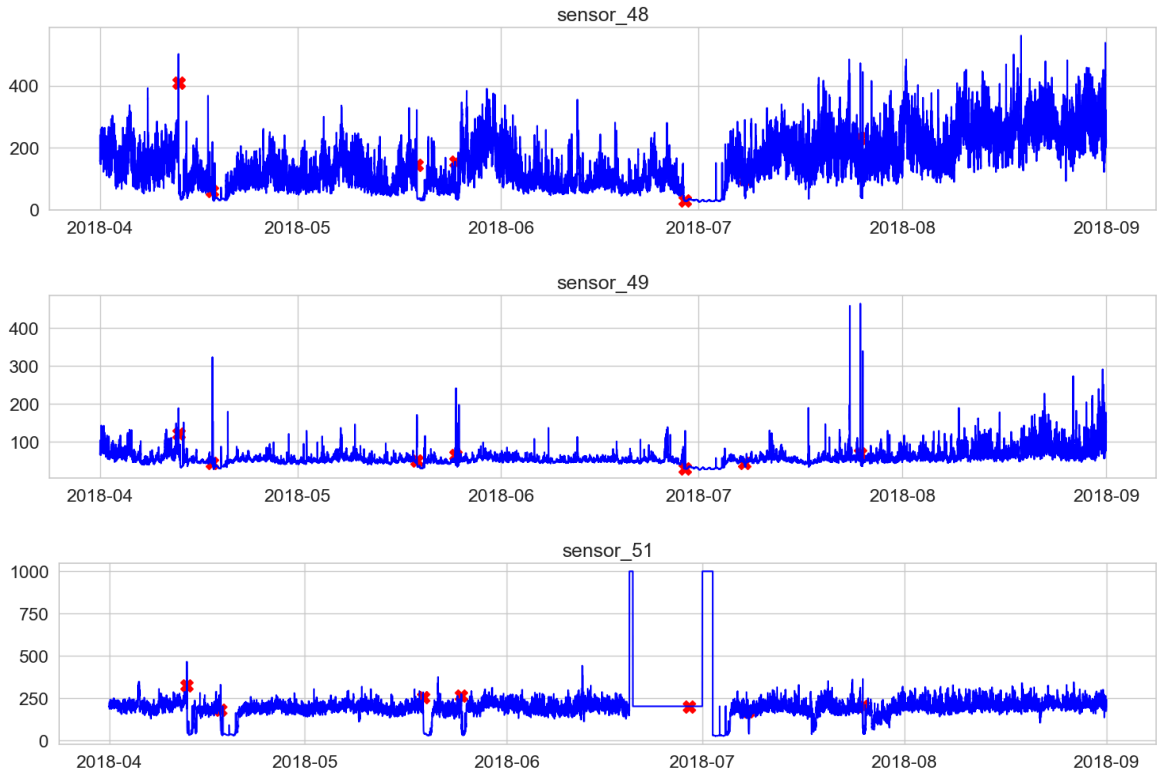

sensor_25
800
600
400
200
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09


sensor_26
1000
500
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09


sensor_27
2000
1500
1000
500
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09
sensor_28
1500
1000
500
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09
sensor_29
1500
1000
500
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09
sensor_30
1500
1000
500
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09
sensor_31
1500
1000
500
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09
sensor_32
1500
1000
500
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09
sensor_33
1500
1000
500
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09


sensor_34
400
300
200
100
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09
sensor_35
600
400
200
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09
sensor_36
1000
750
500
250
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09
sensor_37
150
100
50
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09
sensor_38
400
300
200
100
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09
sensor_39
400
200
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09
sensor_40
400
200
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09


sensor_41
400
300
200
100
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09
sensor_42
300
200
100
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09
sensor_43
400
300
200
100
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09
sensor_44
1000
750
500
250
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09
sensor_45
300
200
100
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09
sensor_46
300
200
100
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09
sensor_47
300
200
100
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09



As seen clearly from the above plots, the red marks, which represent the broken state of the machine(huge spikes), perfectly overlaps with the observed disturbances of the sensor reading. Now we have a pretty good intuition about how each of the sensor reading behaves when the machine is broken vs operating normally.

This could be an indictator for predicting failure.

In [24]:
```
#Converting the machine status column to numeric
MS_class_dict = {"BROKEN": 0, "NORMAL": 1, "RECOVERING": 2}
sensor_df['machine_status'] = sensor_df['machine_status'].map(MS_class
sensor_df.head()
```

Out[24]:

timestamp	sensor_00	sensor_01	sensor_02	sensor_03	sensor_04	sensor_05	sensor_06	ser
2018-04-01 00:00:00	2.465394	47.09201	53.2118	46.310760	634.3750	76.45975	13.41146	1(
2018-04-01 00:01:00	2.465394	47.09201	53.2118	46.310760	634.3750	76.45975	13.41146	1(
2018-04-01 00:02:00	2.444734	47.35243	53.2118	46.397570	638.8889	73.54598	13.32465	1(
2018-04-01 00:03:00	2.460474	47.09201	53.1684	46.397568	628.1250	76.98898	13.31742	1(
2018-04-01 00:04:00	2.445718	47.13541	53.2118	46.397568	636.4583	76.58897	13.35359	1(

5 rows × 51 columns

Step 5: Stationarity and Autocorrelation

In time series analysis, it is important that the data is stationary and have no autocorrelation.

Autocorrelation refers to the behavior of the data where the data is correlated with itself in a different time period.

Rolling mean or avergae is a calculation to analyze data points by creating a series of averages of different subsets of the full data set. It is also called a moving mean.

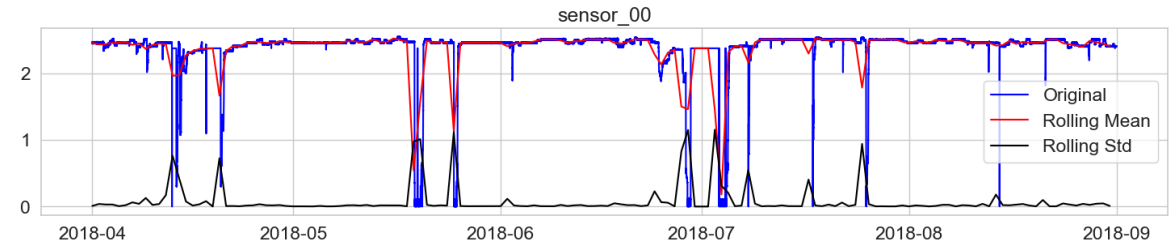
As the next step, I will visually inspect the stationarity of each feature in the data set and to do that,I resampled the data by daily average.

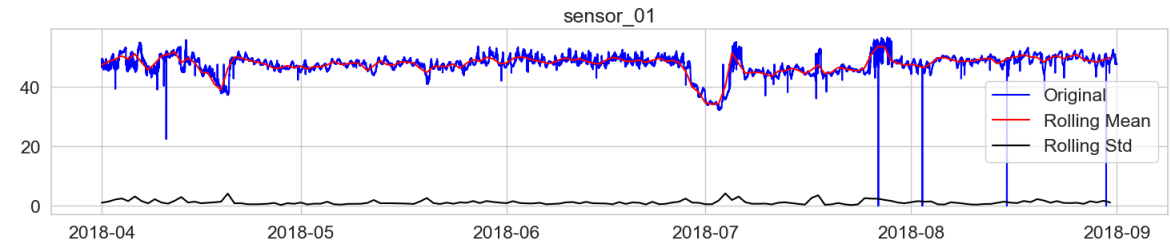
Later, I will also perform the Dickey Fuller test to quantitatively verify the observed stationarity.

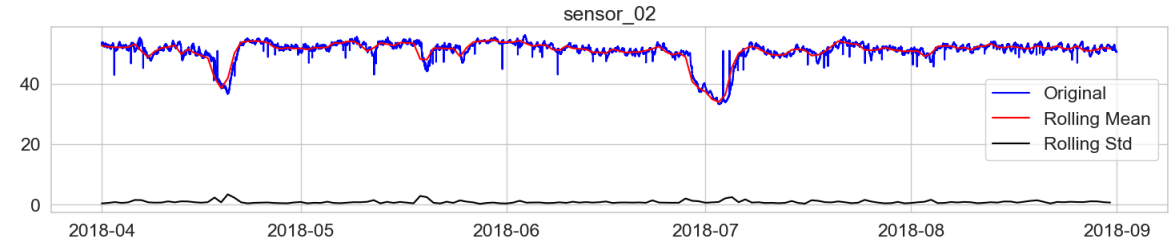
In addition, I will inspect the autocorrelation of the features before feeding them into the clustering algorithms to detect anomalies.

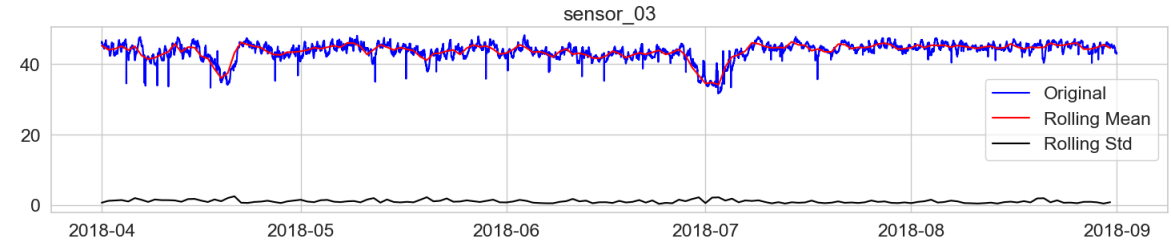
In [25]:
```
# Resample the entire dataset by daily average
rollmean = sensor_df.resample(rule='D').mean()
rollstd = sensor_df.resample(rule='D').std()
```

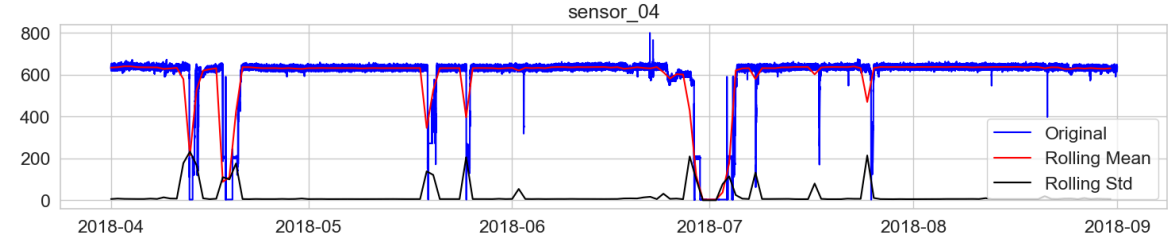
In [26]:
```
# Plot time series for each sensor with its mean and standard deviatio
for name in names:
    _ = plt.figure(figsize=(18,3))
    _ = plt.plot(sensor_df[name], color='blue', label='Original')
    _ = plt.plot(rollmean[name], color='red', label='Rolling Mean')
    _ = plt.plot(rollstd[name], color='black', label='Rolling Std' )
    _ = plt.legend(loc='best')
    _ = plt.title(name)
    plt.show()
```

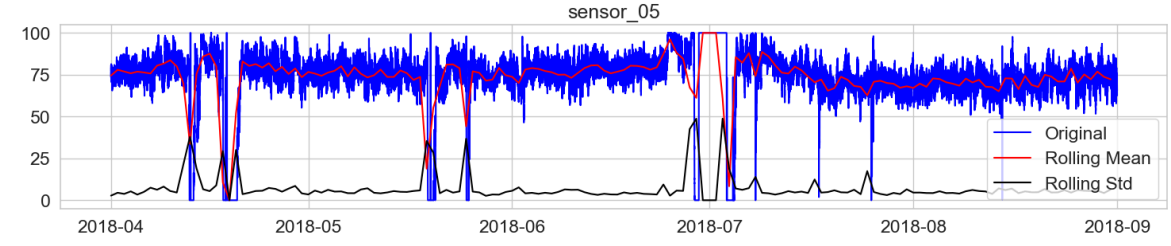


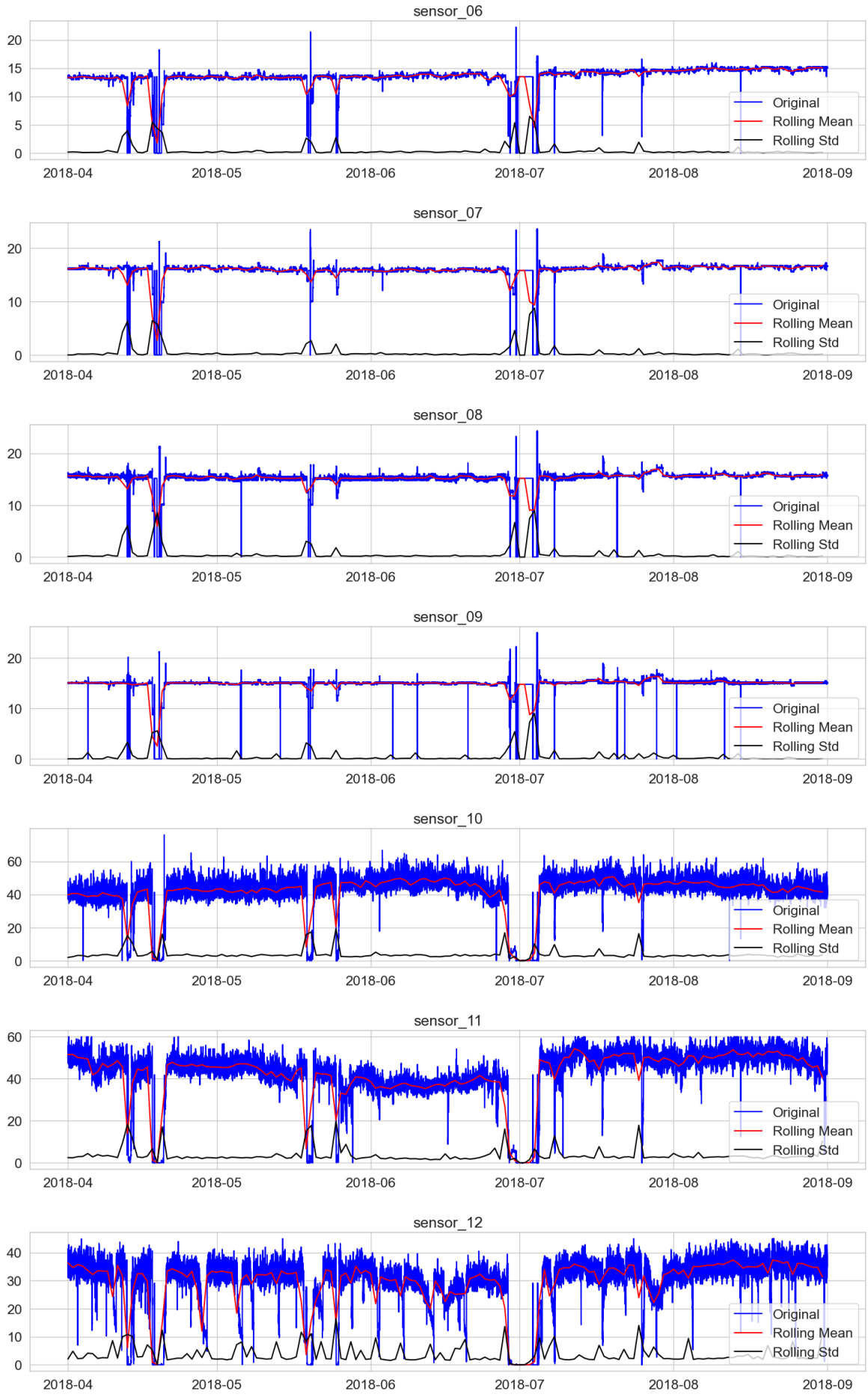


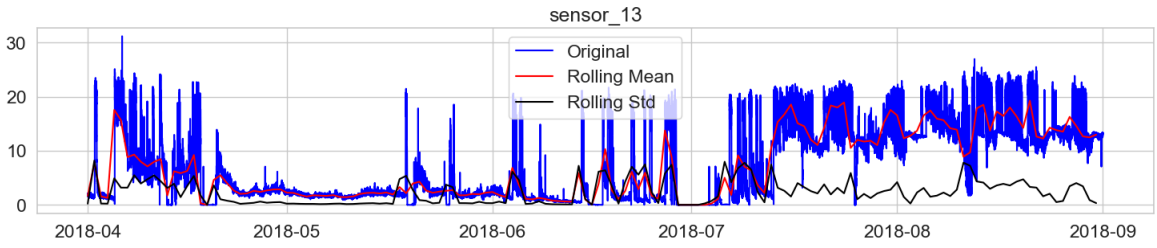


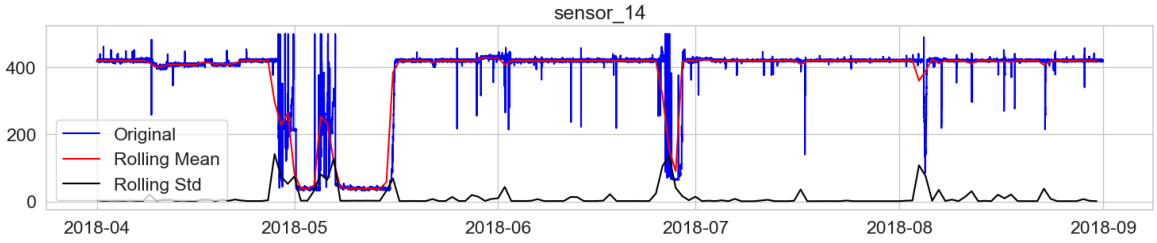


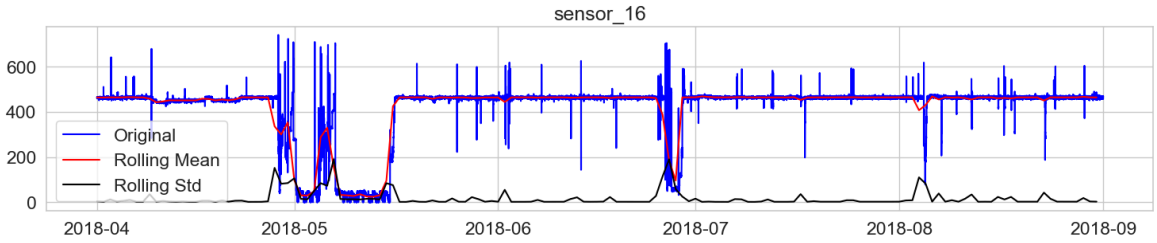


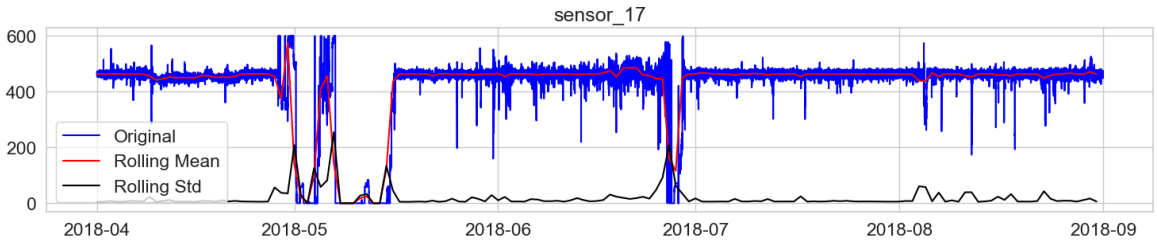


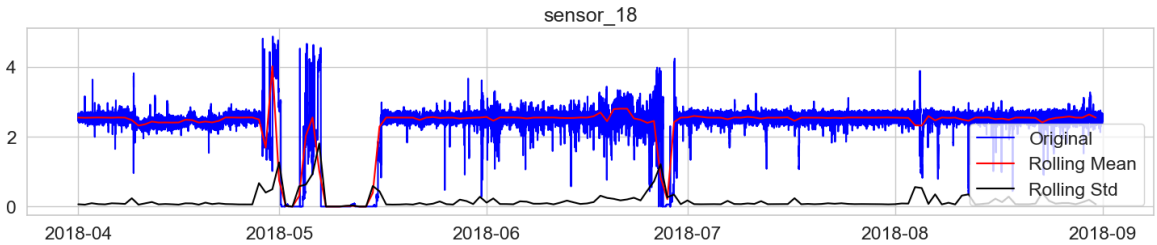

sensor_06
sensor_07
sensor_08
sensor_09
sensor_10
sensor_11
sensor_12
Original
Rolling Mean
Rolling Std
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09

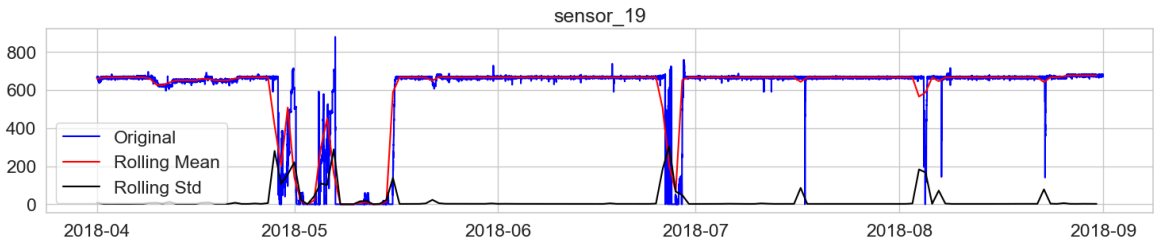

sensor_13
Original
Rolling Mean
Rolling Std
30
20
10
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09

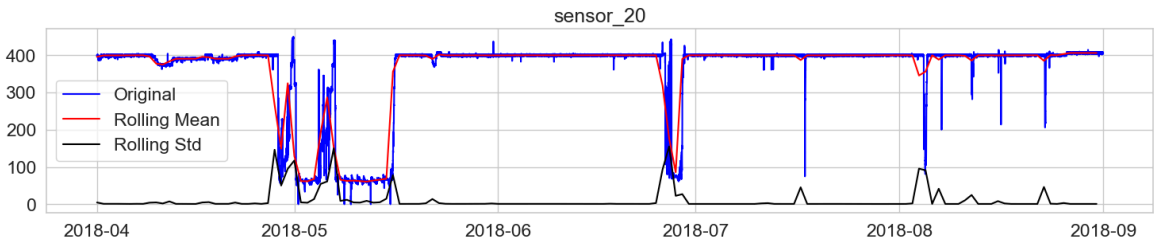

sensor_14
Original
Rolling Mean
Rolling Std
400
200
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09

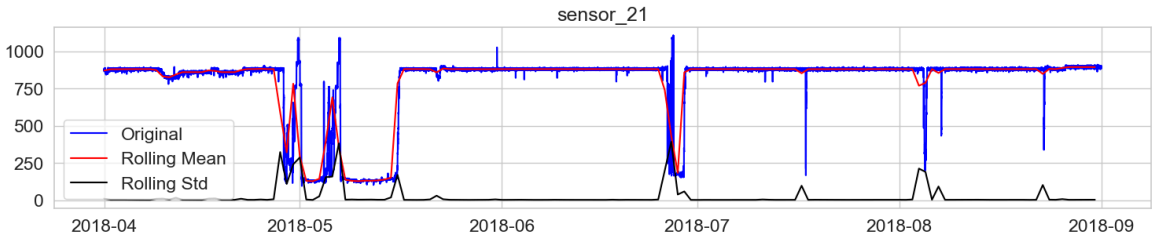

sensor_16
Original
Rolling Mean
Rolling Std
600
400
200
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09

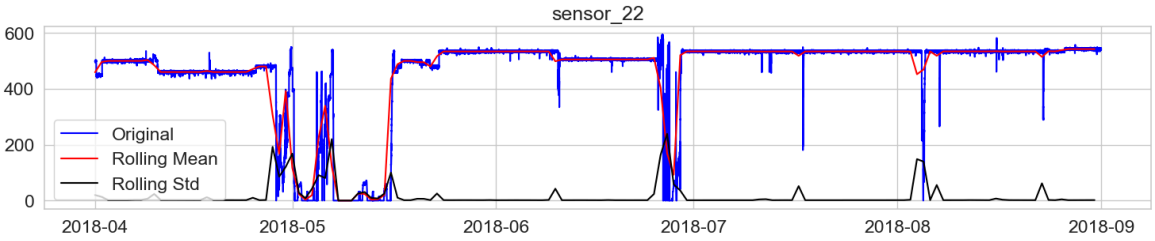

sensor_17
Original
Rolling Mean
Rolling Std
600
400
200
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09

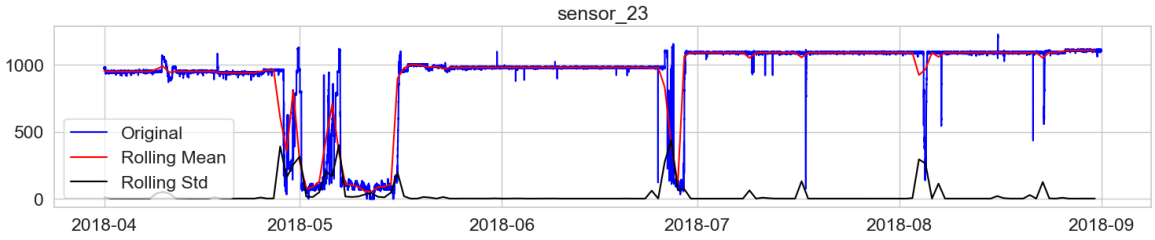

sensor_18
Original
Rolling Mean
Rolling Std
4
2
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09

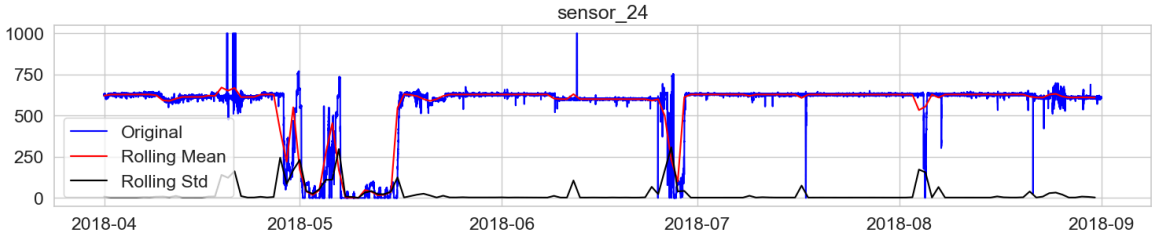

sensor_19
Original
Rolling Mean
Rolling Std
800
600
400
200
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09

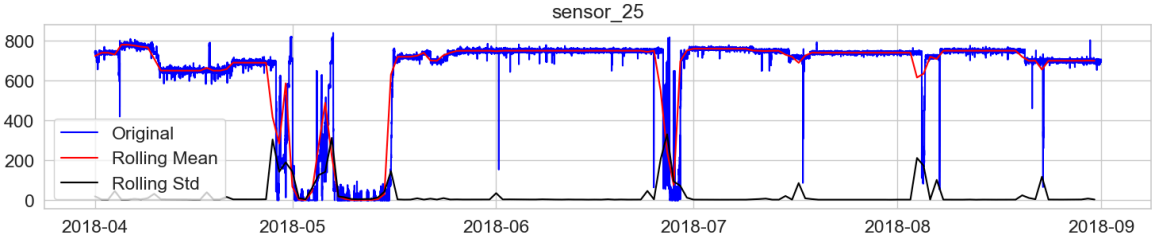

sensor_20
Original
Rolling Mean
Rolling Std
400
300
200
100
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09

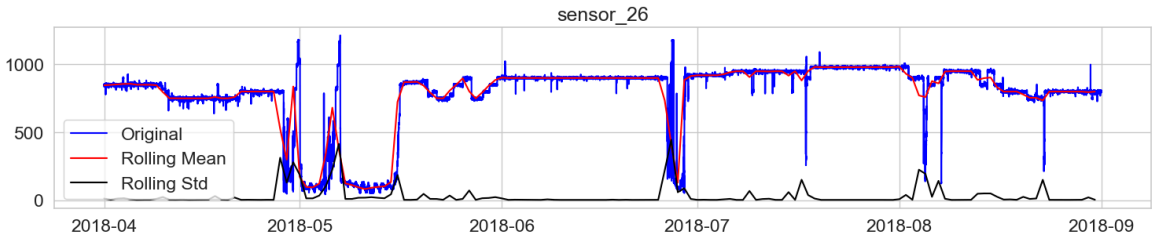

sensor_21
Original
Rolling Mean
Rolling Std
1000
750
500
250
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09

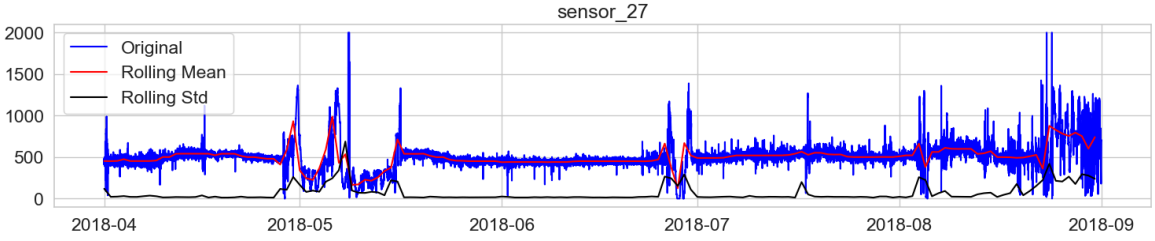

sensor_22
Original
Rolling Mean
Rolling Std
600
400
200
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09

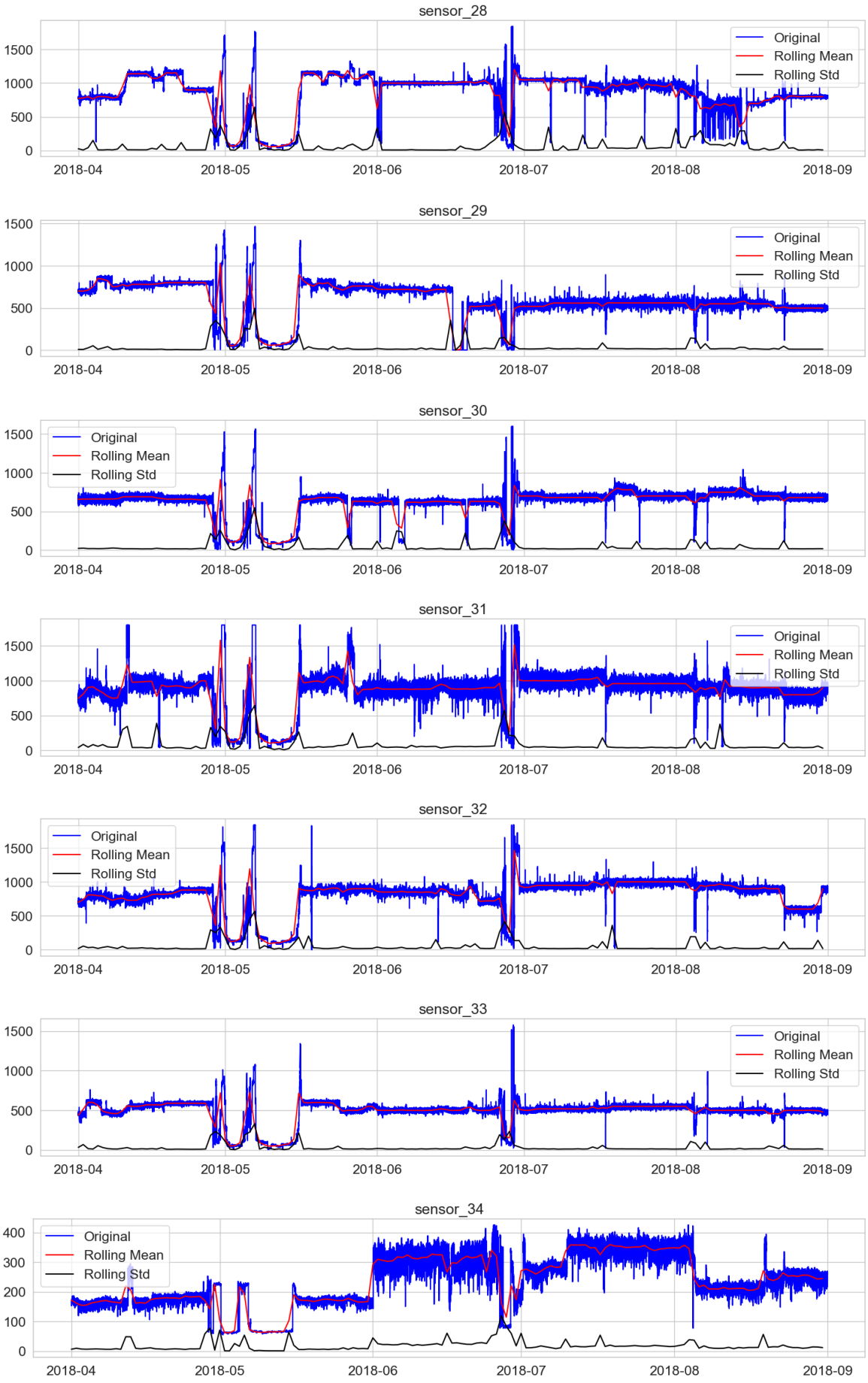

sensor_23
Original
Rolling Mean
Rolling Std
1000
500
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09

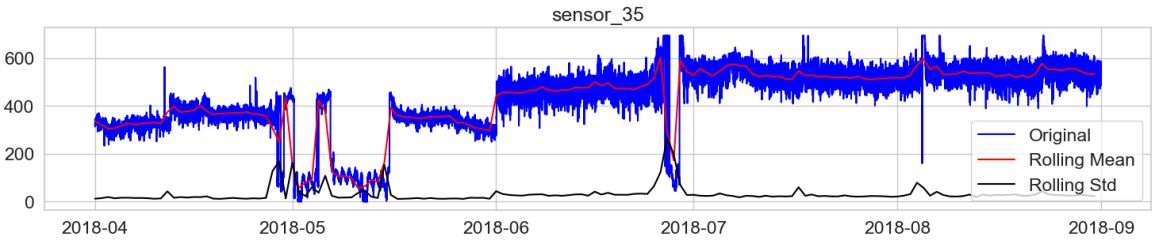

sensor_24
Original
Rolling Mean
Rolling Std
1000
750
500
250
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09

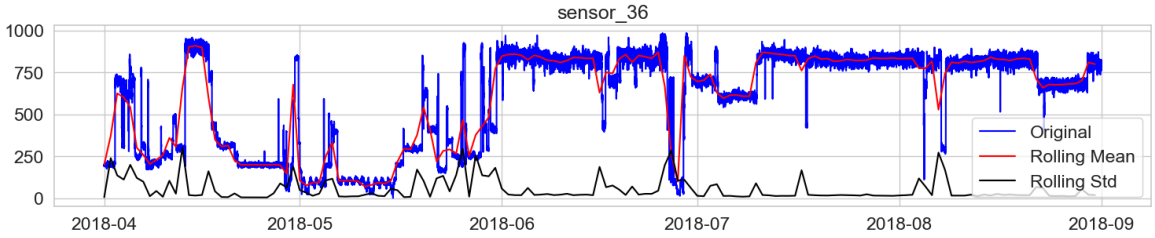

sensor_25
Original
Rolling Mean
Rolling Std
800
600
400
200
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09

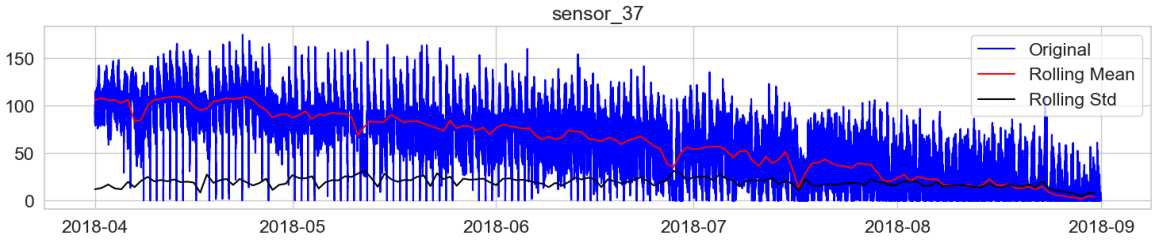

sensor_26
Original
Rolling Mean
Rolling Std
1000
500
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09

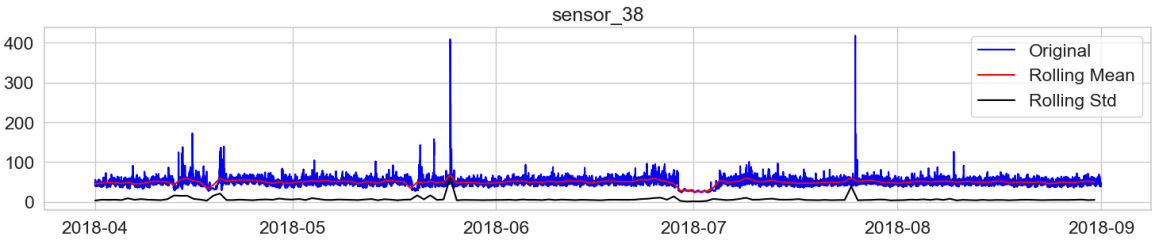

sensor_27
Original
Rolling Mean
Rolling Std
2000
1500
1000
500
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09

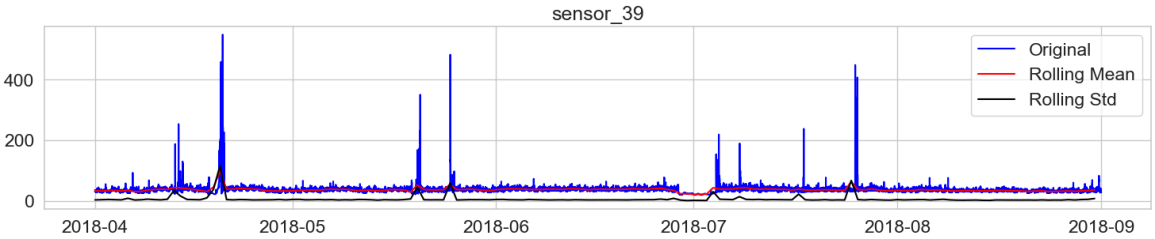

sensor_28
Original
Rolling Mean
Rolling Std
sensor_29
Original
Rolling Mean
Rolling Std
sensor_30
Original
Rolling Mean
Rolling Std
sensor_31
Original
Rolling Mean
Rolling Std
sensor_32
Original
Rolling Mean
Rolling Std
sensor_33
Original
Rolling Mean
Rolling Std
sensor_34
Original
Rolling Mean
Rolling Std
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09

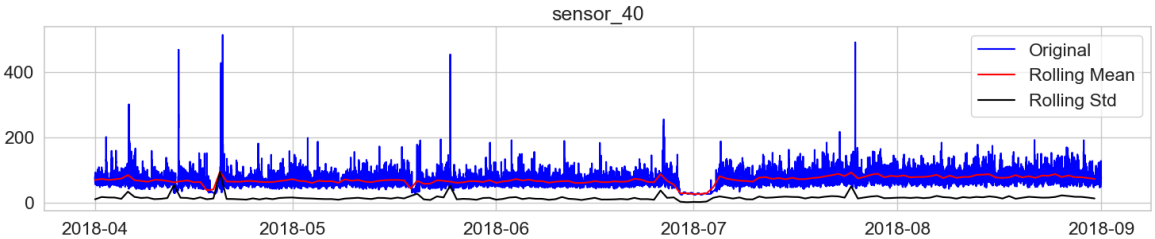

sensor_35
600
400
200
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09
Original
Rolling Mean
Rolling Std

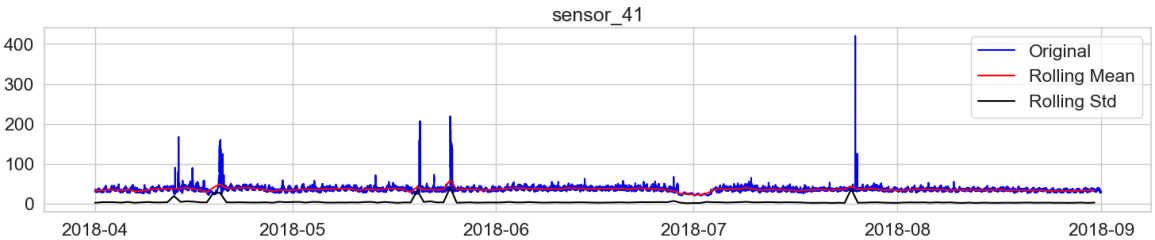

sensor_36
1000
750
500
250
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09
Original
Rolling Mean
Rolling Std

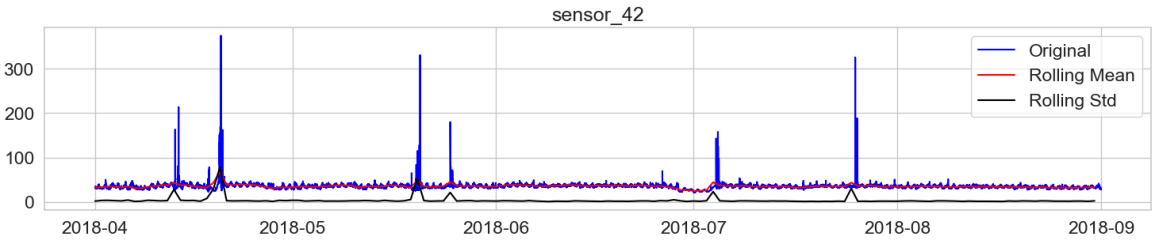

sensor_37
150
100
50
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09
Original
Rolling Mean
Rolling Std

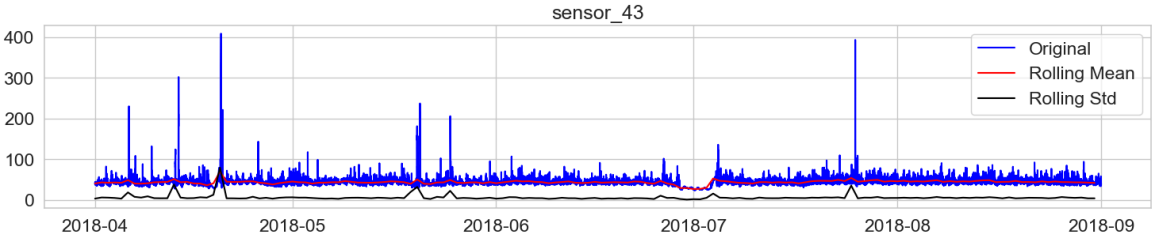

sensor_38
400
300
200
100
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09
Original
Rolling Mean
Rolling Std

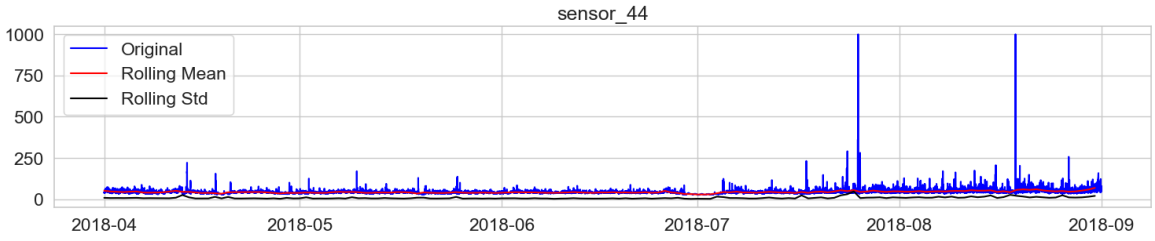

sensor_39
400
200
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09
Original
Rolling Mean
Rolling Std

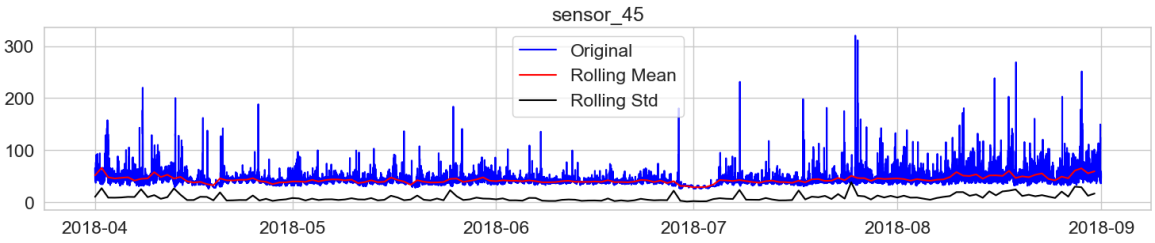

sensor_40
400
200
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09
Original
Rolling Mean
Rolling Std

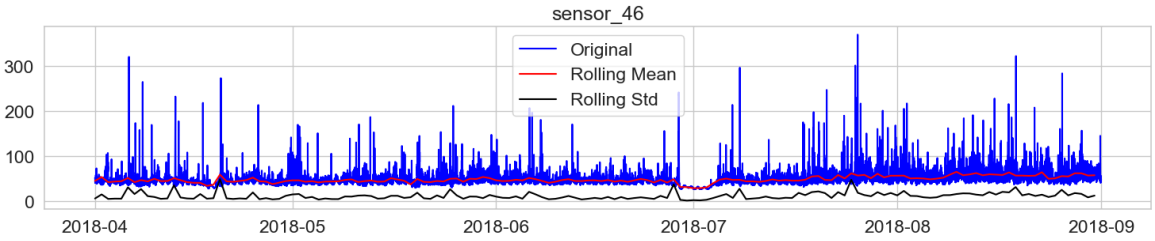

sensor_41
400
300
200
100
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09
Original
Rolling Mean
Rolling Std

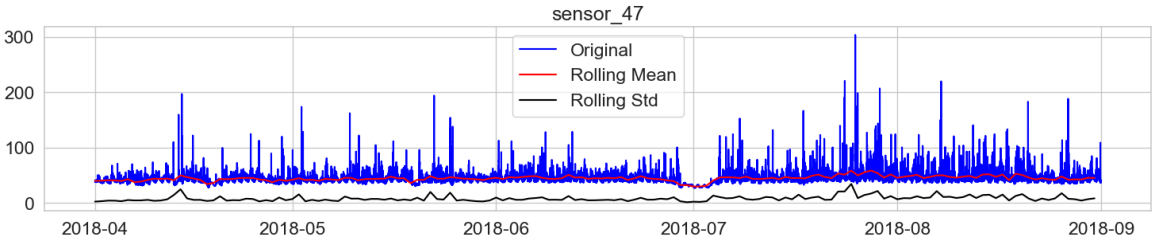

sensor_42
Original
Rolling Mean
Rolling Std
300
200
100
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09

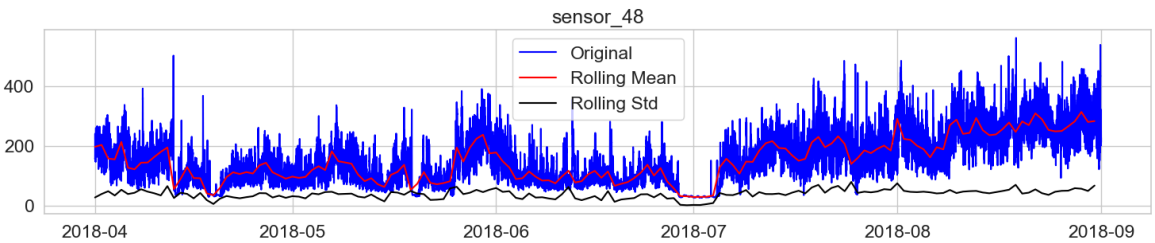

sensor_43
Original
Rolling Mean
Rolling Std
400
300
200
100
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09

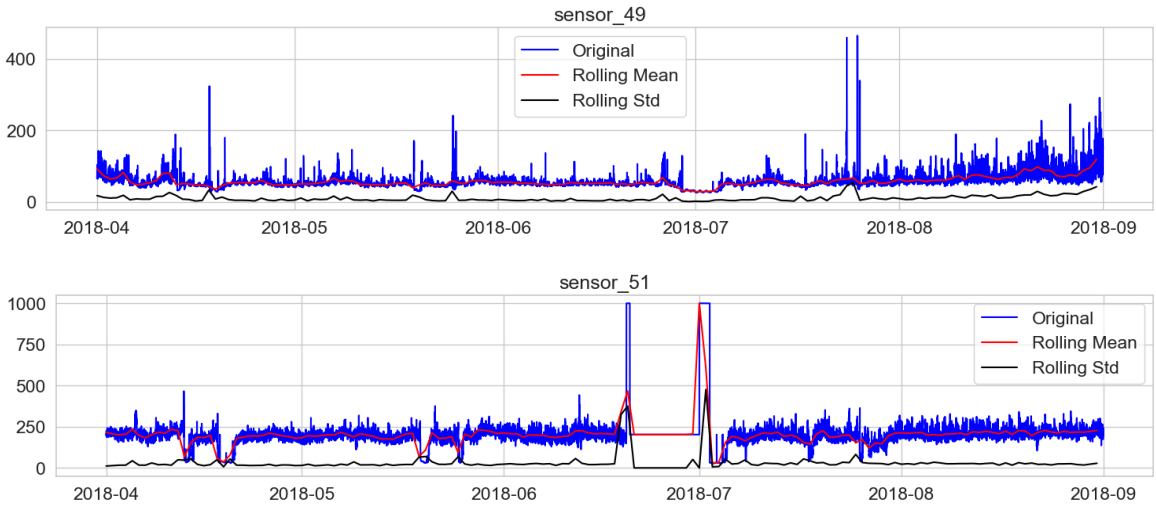

sensor_44
Original
Rolling Mean
Rolling Std
1000
750
500
250
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09


sensor_45
Original
Rolling Mean
Rolling Std
300
200
100
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09


sensor_46
Original
Rolling Mean
Rolling Std
300
200
100
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09


sensor_47
Original
Rolling Mean
Rolling Std
300
200
100
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09


sensor_48
Original
Rolling Mean
Rolling Std
400
200
0
2018-04
2018-05
2018-06
2018-07
2018-08
2018-09



Looking at the above ,plots, the data actually looks pretty stationary where the rolling mean and standard deviation don't seem to change over time except during the downtime of the pump which is expected. This was the case for most of the sensors in this data set but it may not always be the case in which situations various transformation methods must be applied to make the data stationary before training the data.

Step 6: Pre-processing and Feature Engineering

It is pretty computationally expensive to train models with all of the 52 sensors/features and it is not efficient. Therefore, I will employ Principal Component Analysis (PCA) technique to extract new features to be used for the modeling. In order to properly apply PCA, the data must be scaled and standardized. This is because PCA and most of the learning algorithms are distance based algorithms. If noticed from the first 10 rows of the data, the magnitude of the values from each feature is not consistent. Some are very small while some others are really large values. I will perform the following steps using the Pipeline library.

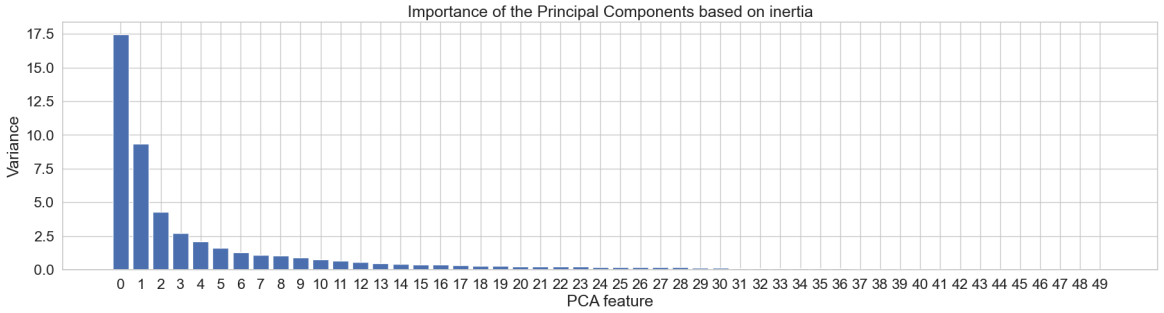
1.Scale the data

2.Perform PCA and look at the most important principal components based on inertia

In [27]:
```
# Standardize/scale the dataset and apply PCA
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
from sklearn.pipeline import make_pipeline

# Extract the names of the numerical columns
df2 = sensor_df.drop(['machine_status'], axis=1)
names=df2.columns
x = sensor_df[names]
scaler = StandardScaler()
pca = PCA()
pipeline = make_pipeline(scaler, pca)
pipeline.fit(x)
```

Out[27]:
```
Pipeline(steps=[('standardscaler', StandardScaler()), ('pca', PCA
())])
```

In [28]:
```
features = range(pca.n_components_)
_ = plt.figure(figsize=(22, 5))
_ = plt.bar(features, pca.explained_variance_)
_ = plt.xlabel('PCA feature')
_ = plt.ylabel('Variance')
_ = plt.xticks(features)
_ = plt.title("Importance of the Principal Components based on inertia
plt.show()
```



It appears that the first two principal components are the most important as per the features extracted by the PCA in above importance plot. So as the next step, I will perform PCA with 2 components which will be my features to be used in the training of the models.

In [29]:
```
# Calculate PCA with 2 components
pca = PCA(n_components=2)
principalComponents = pca.fit_transform(x)
principalDf = pd.DataFrame(data = principalComponents, columns = ['pc1
principalDf.head()
```

Out[29]:

	pc1	pc2
0	69.523134	265.704571
1	69.523134	265.704571
2	27.841569	283.462197
3	24.548981	290.213914
4	29.627090	294.619639

In [30]:
```
sensor_df['pc1']=pd.Series(principalDf['pc1'].values, index=sensor_df.
sensor_df['pc2']=pd.Series(principalDf['pc2'].values, index=sensor_df.
```

In [31]:
```
sensor_df['pc1'],sensor_df['pc2']
```

Out[31]:
```
(timestamp
 2018-04-01 00:00:00     69.523134
 2018-04-01 00:01:00     69.523134
 2018-04-01 00:02:00     27.841569
 2018-04-01 00:03:00     24.548981
 2018-04-01 00:04:00     29.627090
                            ...
 2018-08-31 23:55:00   -308.531521
 2018-08-31 23:56:00   -294.603672
 2018-08-31 23:57:00   -300.207654
 2018-08-31 23:58:00   -285.141390
 2018-08-31 23:59:00   -298.182526
 Name: pc1, Length: 220320, dtype: float64,
 timestamp
 2018-04-01 00:00:00    265.704571
 2018-04-01 00:01:00    265.704571
 2018-04-01 00:02:00    283.462197
 2018-04-01 00:03:00    290.213914
 2018-04-01 00:04:00    294.619639
                            ...
 2018-08-31 23:55:00   -274.597310
 2018-08-31 23:56:00   -256.328332
 2018-08-31 23:57:00   -256.932727
 2018-08-31 23:58:00   -263.099471
 2018-08-31 23:59:00   -264.545953
 Name: pc2, Length: 220320, dtype: float64)
```

Check stationarity with Dickey-Fuller Test

A time series has stationarity if a shift in time doesn't cause a change in the shape of the distribution. Basic properties of the distribution like the mean , variance and covariance are constant over time. Stationarity can be defined in precise mathematical terms, but for our purpose we mean a flat looking series, without trend, constant variance over time, a constant autocorrelation structure over time and no periodic fluctuations (seasonality).

ADF (Augmented Dickey Fuller) Test The Dickey Fuller test is one of the most popular statistical tests. It can be used to determine the presence of unit root in the series, and hence help us understand if the series is stationary or not. The null and alternate hypothesis of this test are:

Null Hypothesis: The series has a unit root (value of a =1)

Alternate Hypothesis: The series has no unit root.

If we fail to reject the null hypothesis, we can say that the series is non-stationary. This means that the series can be linear or difference stationary (we will understand more about difference stationary in the next section).

Now, I will check again the stationarity and autocorrelation of these two principal components just to be sure they are stationary and not autocorrelated

In [32]:
```
from statsmodels.tsa.stattools import adfuller
# Run Augmented Dickey Fuller Test
result = adfuller(principalDf['pc1'])
# Print p-value
print(result[1])
```

```
2.462944436954928e-05
```

In [33]:
```
# Run Augmented Dickey Fuller Test
result = adfuller(principalDf['pc2'])
# Print p-value
print(result[1])
```

```
1.7777868189432241e-06
```

Running the Dickey Fuller test on the 1st principal component, I got a p-value of 2.4629444369550607e-05 which is very small number (much smaller than 0.05). Thus, I will reject the Null Hypothesis and say the data is stationary. I performed the same on the 2nd component and got a similar result.

So both of the principal components are stationary which is what I wanted.

Check for Autocorrelation

Autocorrelation represents the degree of similarity between a given time series and a lagged version of itself over successive time intervals . Autocorrelation measures the relationship between a variable's current value and its past values.

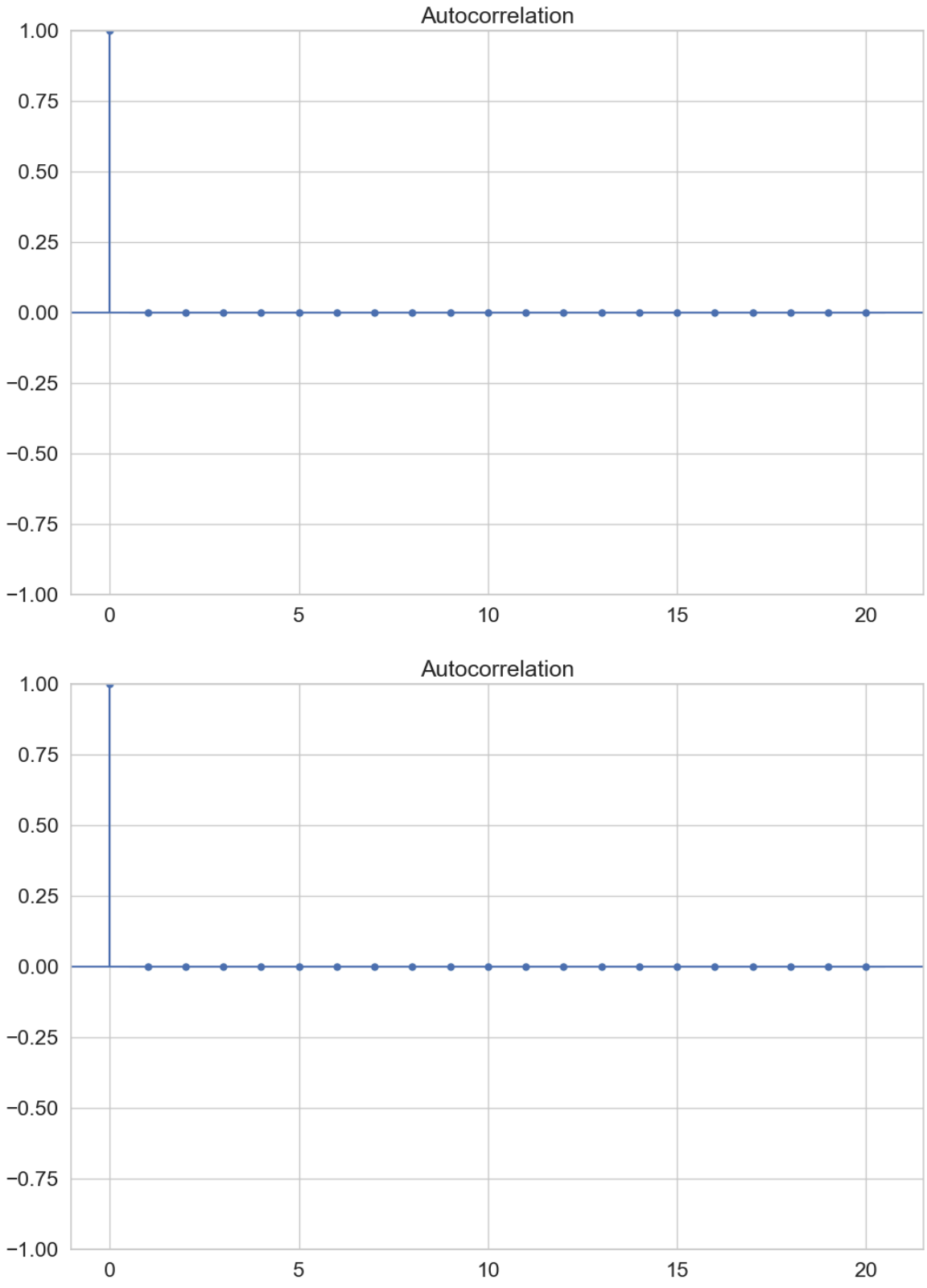
Now, let's check for autocorrelation in both of these principal components. It can be done one of the two ways; either with the pandas autocorr() method or ACF plot. I will use the latter in this case to quickly visually verify that there is no autocorrelation. The following code does just that.

In [34]:
```
# Compute change in daily mean
pca1 = principalDf['pc1'].pct_change()
# Compute autocorrelation
autocorrelation = pca1.dropna().autocorr()
print('Autocorrelation is: ', autocorrelation)
```

```
Autocorrelation is:  -6.773604518564134e-06
```

In [35]:
```
# Plot ACF
from statsmodels.graphics.tsaplots import plot_acf
plot_acf(pca1.dropna(), lags=20, alpha=0.05)
```

Out[35]:

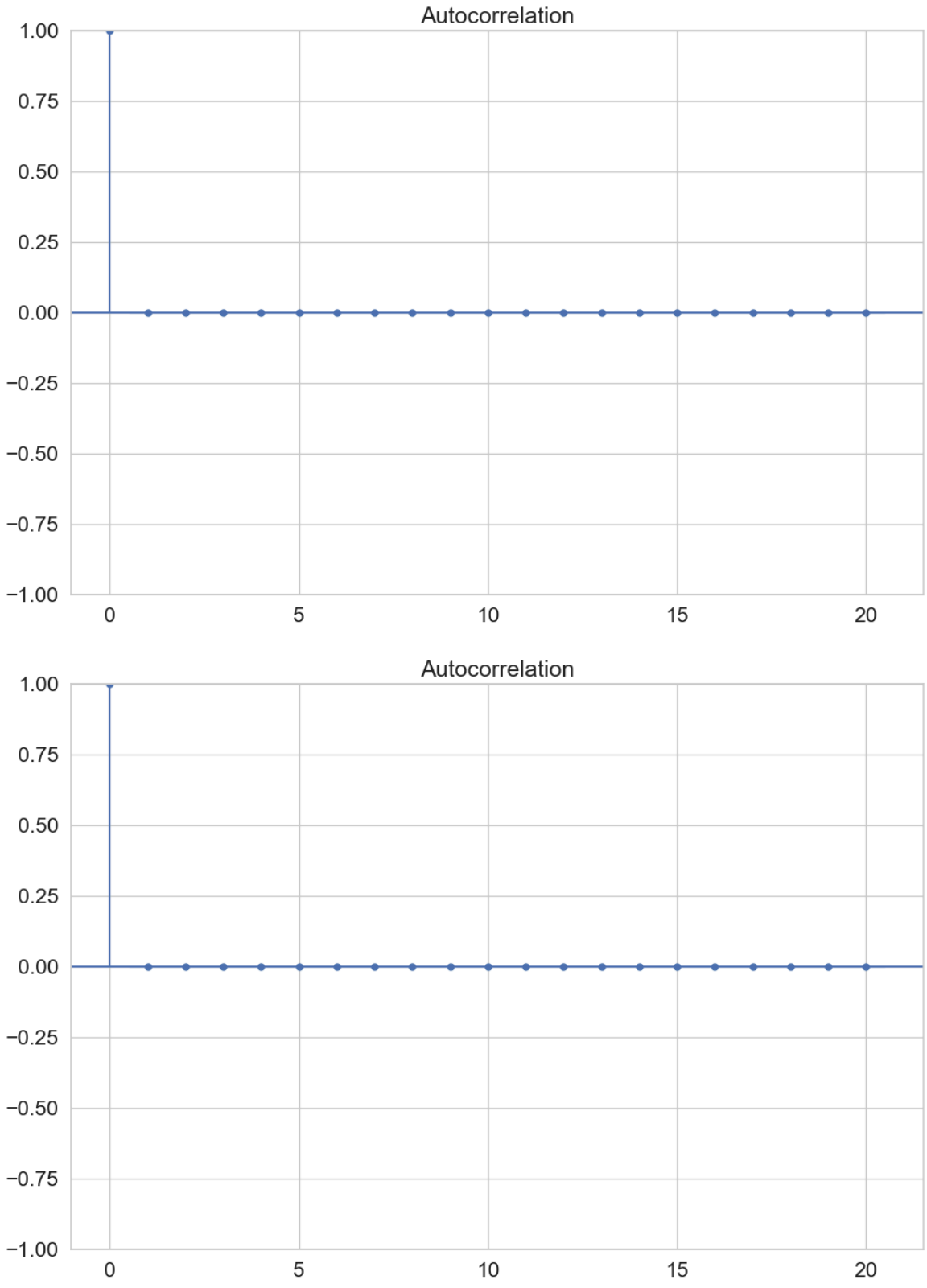


In [36]:
```
# Compute change in daily mean
pca2 = principalDf['pc2'].pct_change()
# Compute autocorrelation
autocorrelation = pca2.autocorr()
print('Autocorrelation is: ', autocorrelation)
```

```
Autocorrelation is:  -1.4768185278992946e-06
```

In [37]:
```
# Plot ACF
from statsmodels.graphics.tsaplots import plot_acf
plot_acf(pca2.dropna(), lags=20, alpha=0.05)
```

Out[37]:



As seen from above, both features "pca1" and "pca2" are stationary and not autocorrelated, I am ready for modeling using these new features(principal components).

Step 7: Modelling

In this step, I will perform the following unsupervised learning algorithms to detect anomalies.

1) Benchmark model: Interquartile Range (IQR)

2) K-Means clustering

3) Isolation Forest

Let's start training with these algorithms.

Base Model 1: Detect Outliers Using the Interquartile Range (IQR)

Anomalies are defined as rare events that could be represented by the outliers in the data set. As an initial step, I want to apply a basic statistics technique to get the feel of the outliers present in this data set. Later, I will compare the results of the other models to the results from the Base Model for further model evaluation.

0: normal

1: anomaly

Strategy:

1)Calculate IQR which is the difference between 75th (Q3)and 25th (Q1) percentiles.

2)Calculate upper and lower bounds for the outlier.

3)Filter the data points that fall outside the upper and lower bounds and flag them as outliers.

Finally, plot the outliers on top of the time series data

In [38]:
```
#Calculate IQR for the 1st principal component (pc1)
q1_pc1, q3_pc1 = sensor_df['pc1'].quantile([0.25, 0.75])
iqr_pc1 = q3_pc1 - q1_pc1
iqr_pc1
```

Out[38]:
```
240.64393663534756
```

In [39]:
```
# Calculate upper and lower bounds for outlier for pc1
lower_pc1 = q1_pc1 - (1.5*iqr_pc1)
upper_pc1 = q3_pc1 + (1.5*iqr_pc1)
lower_pc1,upper_pc1
```

Out[39]:
```
(-740.3139494894435, 222.26179705194662)
```

In [40]:
```
# Filter out the outliers from the pc1
sensor_df['anomaly_pc1'] = ((sensor_df['pc1']>upper_pc1) | (sensor_df|
sensor_df['anomaly_pc1']
```

Out[40]:
```
timestamp
2018-04-01 00:00:00    0
2018-04-01 00:01:00    0
2018-04-01 00:02:00    0
2018-04-01 00:03:00    0
2018-04-01 00:04:00    0
                      ..
2018-08-31 23:55:00    0
2018-08-31 23:56:00    0
2018-08-31 23:57:00    0
2018-08-31 23:58:00    0
2018-08-31 23:59:00    0
Name: anomaly_pc1, Length: 220320, dtype: int64
```

In [41]:
```
# Calculate IQR for the 2nd principal component (pc2)
q1_pc2, q3_pc2 = sensor_df['pc2'].quantile([0.25, 0.75])
iqr_pc2 = q3_pc2 - q1_pc2
iqr_pc2
```

Out[41]:
```
352.9994254729783
```

In [42]:
```
# Calculate upper and lower bounds for outlier for pc2
lower_pc2 = q1_pc2 - (1.5*iqr_pc2)
upper_pc2 = q3_pc2 + (1.5*iqr_pc2)
lower_pc2,upper_pc2
```

Out[42]:
```
(-734.080484268977, 677.9172176229362)
```

In [43]:
```
# Filter out the outliers from the pc2
sensor_df['anomaly_pc2'] = ((sensor_df['pc2']>upper_pc2) | (sensor_df|
sensor_df['anomaly_pc2']
```

Out[43]:
```
timestamp
2018-04-01 00:00:00    0
2018-04-01 00:01:00    0
2018-04-01 00:02:00    0
2018-04-01 00:03:00    0
2018-04-01 00:04:00    0
                      ..
2018-08-31 23:55:00    0
2018-08-31 23:56:00    0
2018-08-31 23:57:00    0
2018-08-31 23:58:00    0
2018-08-31 23:59:00    0
Name: anomaly_pc2, Length: 220320, dtype: int64
```

In [44]:
```
sensor_df['anomaly_pc1'].value_counts()
```

Out[44]:
```
0    190149
1     30171
Name: anomaly_pc1, dtype: int64
```

In [45]:
```
sensor_df['anomaly_pc2'].value_counts()
```

Out[45]:
```
0    218696
1      1624
Name: anomaly_pc2, dtype: int64
```

Now I want to select the most important 20 features in the data set to detect anomalies in them. To find out these features, I will use Univariate feature selection technique.

In [46]:
```
# Apply SelectKBest class to extract the best 20 features - Univariate
from sklearn.feature_selection import SelectKBest
from sklearn.feature_selection import chi2
from sklearn.preprocessing import MinMaxScaler

x = sensor_df.drop(['machine_status', 'pc1', 'pc2', 'anomaly_pc1', 'ar
y = sensor_df['machine_status']
scaler = MinMaxScaler()
x_scaled = scaler.fit_transform(x)
bestfeatures = SelectKBest(score_func=chi2, k=15)
fit = bestfeatures.fit(x_scaled, y)
dfscores = pd.DataFrame(fit.scores_)
dfcolumns = pd.DataFrame(x.columns)
featureScores = pd.concat([dfcolumns, dfscores], axis=1)
featureScores.columns = ['Feature', 'Score']
print(featureScores.nlargest(15, 'Score'))
```

```
      Feature         Score
11  sensor_11  10123.423188
12  sensor_12   9893.118661
4   sensor_04   8183.512414
10  sensor_10   7770.145936
2   sensor_02   4579.464879
13  sensor_13   3460.233573
47  sensor_48   3004.248623
3   sensor_03   2680.377467
5   sensor_05   1694.808230
6   sensor_06   1379.585913
0   sensor_00   1205.839338
7   sensor_07    872.728602
9   sensor_09    699.247388
8   sensor_08    653.005739
39  sensor_40    639.106075
```

The above sensors have a high feature score and will be used for further analysis.

In [47]:
```
sensor_df.head(20)
```

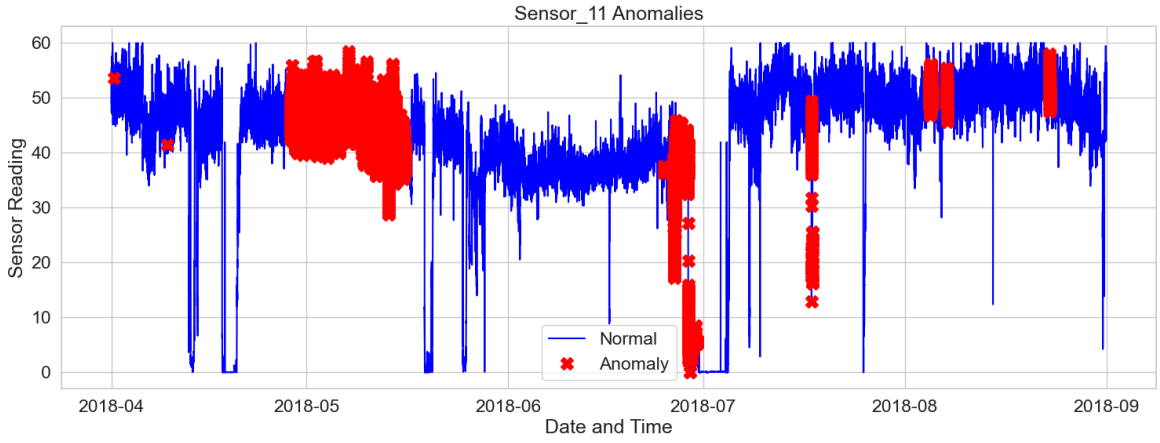
Out[47]:

timestamp	sensor_00	sensor_01	sensor_02	sensor_03	sensor_04	sensor_05	sensor_06	ser
2018-04-01 00:00:00	2.465394	47.09201	53.21180	46.310760	634.3750	76.45975	13.41146	1(
2018-04-01 00:01:00	2.465394	47.09201	53.21180	46.310760	634.3750	76.45975	13.41146	1(
2018-04-01 00:02:00	2.444734	47.35243	53.21180	46.397570	638.8889	73.54598	13.32465	1(
2018-04-01 00:03:00	2.460474	47.09201	53.16840	46.397568	628.1250	76.98898	13.31742	1(
2018-04-01 00:04:00	2.445718	47.13541	53.21180	46.397568	636.4583	76.58897	13.35359	1(
2018-04-01 00:05:00	2.453588	47.09201	53.16840	46.397568	637.6157	78.18568	13.41146	1(
2018-04-01 00:06:00	2.455556	47.04861	53.16840	46.397568	633.3333	75.81614	13.43316	1(
2018-04-01 00:07:00	2.449653	47.13541	53.16840	46.397568	630.6713	75.77331	13.25231	1(
2018-04-01 00:08:00	2.463426	47.09201	53.16840	46.397568	631.9444	74.58916	13.28848	1(
2018-04-01 00:09:00	2.445718	47.17882	53.16840	46.397568	641.7823	74.57428	13.38252	1(
2018-04-01 00:10:00	2.464410	47.48264	53.12500	46.397568	637.7314	76.05148	13.41146	1(
2018-04-01 00:11:00	2.444734	47.91666	53.16840	46.397568	635.6482	74.58654	13.41146	1(
2018-04-01 00:12:00	2.460474	48.26389	53.12500	46.397568	630.0926	76.95988	13.34635	1(
2018-04-01 00:13:00	2.448669	48.43750	53.16840	46.397568	638.6574	75.67310	13.31742	1(
2018-04-01 00:14:00	2.453588	48.56771	53.16840	46.397568	632.4074	80.65949	13.38976	1(
2018-04-01 00:15:00	2.455556	48.39410	53.12500	46.397570	642.3611	78.13193	13.35359	1(
2018-04-01 00:16:00	2.449653	48.39410	53.16840	46.310760	630.2084	77.89381	13.30295	1(

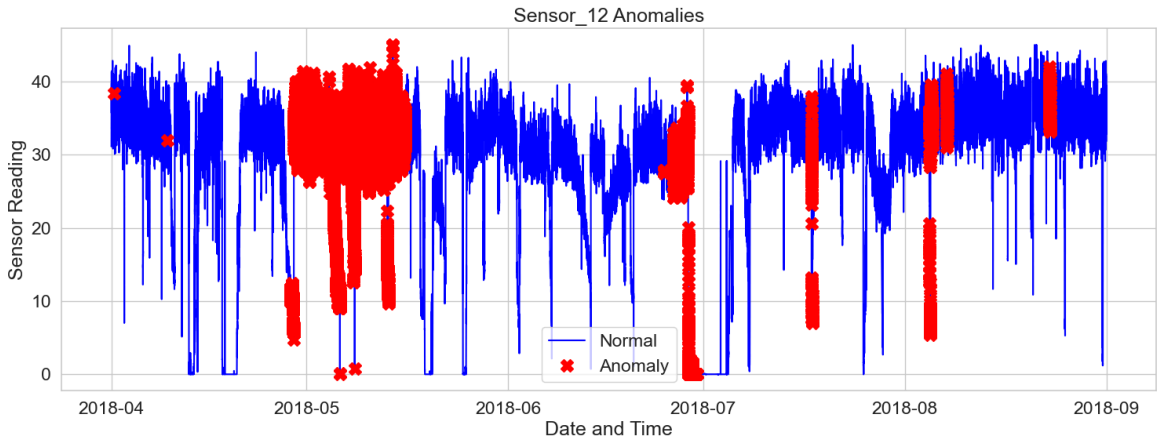
timestamp	sensor_00	sensor_01	sensor_02	sensor_03	sensor_04	sensor_05	sensor_06	ser
2018-04-01 00:17:00	2.463426	48.48090	53.68924	46.310760	643.6343	77.30572	13.34635	1(
2018-04-01 00:18:00	2.445718	48.61111	53.12500	46.310760	632.9861	76.66199	13.34635	1(
2018-04-01 00:19:00	2.464410	48.61111	53.16840	46.310760	644.3287	78.49116	13.34635	1(

20 rows × 55 columns

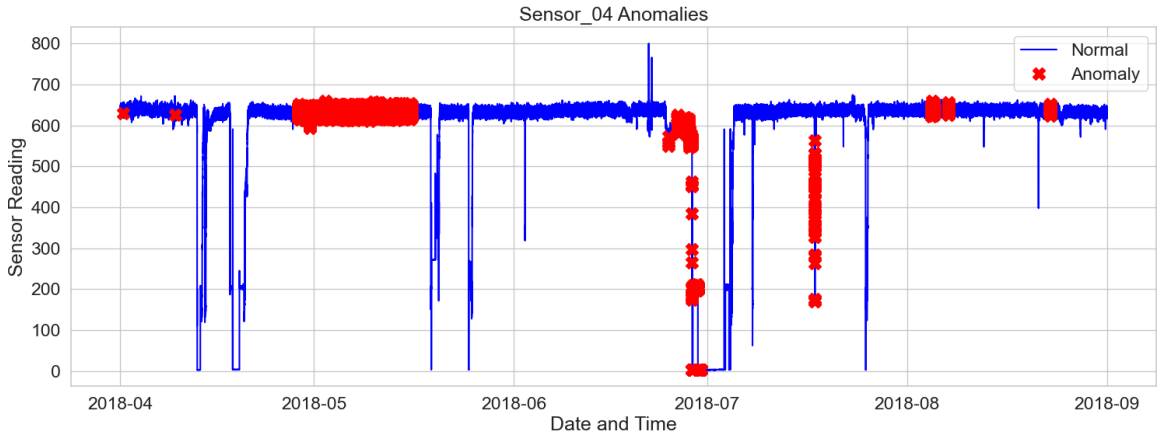
In [48]:
```
# Let's plot the outliers from pc1 on top of the sensor_11 see where t
a = sensor_df[sensor_df['anomaly_pc1'] == 1] #anomaly
_ = plt.figure(figsize=(18,6))
_ = plt.plot(sensor_df['sensor_11'], color='blue', label='Normal')
_ = plt.plot(a['sensor_11'], linestyle='none', marker='X', color='red'
_ = plt.xlabel('Date and Time')
_ = plt.ylabel('Sensor Reading')
_ = plt.title('Sensor_11 Anomalies')
_ = plt.legend(loc='best')
plt.show();
```



In [49]:
```
# Let's plot the outliers from pc1 on top of the sensor_12 see where t
a = sensor_df[sensor_df['anomaly_pc1'] == 1] #anomaly
_ = plt.figure(figsize=(18,6))
_ = plt.plot(sensor_df['sensor_12'], color='blue', label='Normal')
_ = plt.plot(a['sensor_12'], linestyle='none', marker='X', color='red'
_ = plt.xlabel('Date and Time')
_ = plt.ylabel('Sensor Reading')
_ = plt.title('Sensor_12 Anomalies')
_ = plt.legend(loc='best')
plt.show();
```

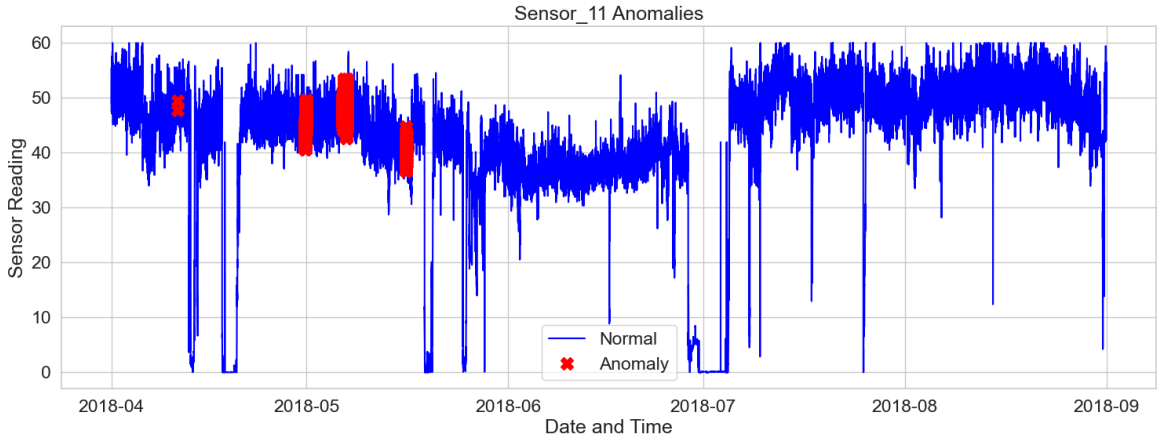


In [50]:
```
# Let's plot the outliers from pc1 on top of the sensor_04 see where t
a = sensor_df[sensor_df['anomaly_pc1'] == 1] #anomaly
_ = plt.figure(figsize=(18,6))
_ = plt.plot(sensor_df['sensor_04'], color='blue', label='Normal')
_ = plt.plot(a['sensor_04'], linestyle='none', marker='X', color='red'
_ = plt.xlabel('Date and Time')
_ = plt.ylabel('Sensor Reading')
_ = plt.title('Sensor_04 Anomalies')
_ = plt.legend(loc='best')
plt.show();
```

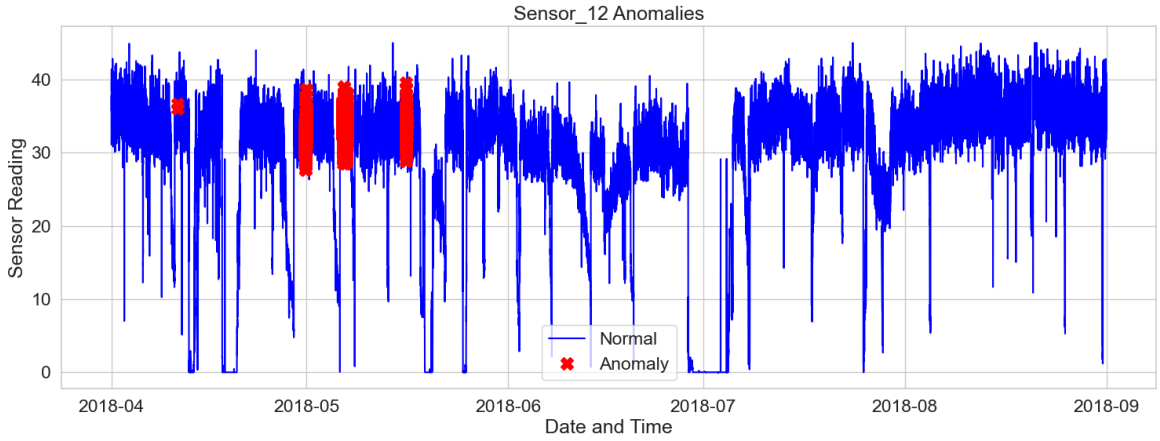


In [51]:
```
#Now lets check for pc2
```

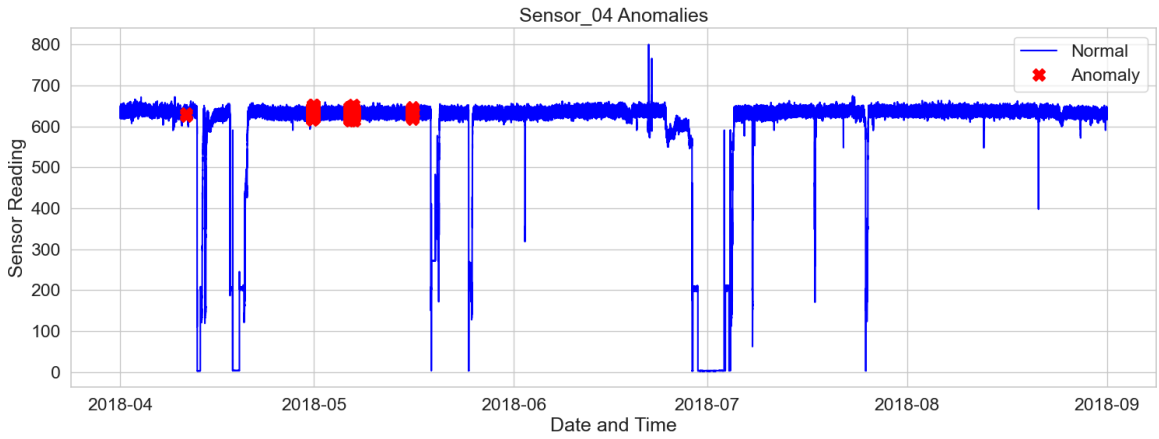
In [52]:
```
# Let's plot the outliers from pc2 on top of the sensor_11 see where t
a = sensor_df[sensor_df['anomaly_pc2'] == 1] #anomaly
_ = plt.figure(figsize=(18,6))
_ = plt.plot(sensor_df['sensor_11'], color='blue', label='Normal')
_ = plt.plot(a['sensor_11'], linestyle='none', marker='X', color='red'
_ = plt.xlabel('Date and Time')
_ = plt.ylabel('Sensor Reading')
_ = plt.title('Sensor_11 Anomalies')
_ = plt.legend(loc='best')
plt.show();
```



In [53]:
```
# Let's plot the outliers from pc2 on top of the sensor_12 see where t
a = sensor_df[sensor_df['anomaly_pc2'] == 1] #anomaly
_ = plt.figure(figsize=(18,6))
_ = plt.plot(sensor_df['sensor_12'], color='blue', label='Normal')
_ = plt.plot(a['sensor_12'], linestyle='none', marker='X', color='red'
_ = plt.xlabel('Date and Time')
_ = plt.ylabel('Sensor Reading')
_ = plt.title('Sensor_12 Anomalies')
_ = plt.legend(loc='best')
plt.show();
```



In [54]:
```
# Let's plot the outliers from pc2 on top of the sensor_04 see where t
a = sensor_df[sensor_df['anomaly_pc2'] == 1] #anomaly
_ = plt.figure(figsize=(18,6))
_ = plt.plot(sensor_df['sensor_04'], color='blue', label='Normal')
_ = plt.plot(a['sensor_04'], linestyle='none', marker='X', color='red'
_ = plt.xlabel('Date and Time')
_ = plt.ylabel('Sensor Reading')
_ = plt.title('Sensor_04 Anomalies')
_ = plt.legend(loc='best')
plt.show();
```



As seen above, the anomalies are detected right before the machine breaks down. This could be a very valuable information for an operator to see and be able to shut down the machine properly before it actually goes down hard.

Also, the anamolies should be investigated and compared to ay failures which have occured.

Let's see if we detect similar pattern in anomalies from the next algorithms.

As seen from above, there are a lot more outliers in pc1 (1st principal component) than that from pc2. Also the outliers in pc1 seem to better explain the failures in the sensor readings from one of the sensors.

Model 2: K-Means Clustering

Strategy:

The underline assumption in the clustering based anomaly detection is that if we cluster the data, normal data will belong to clusters while anomalies will not belong to any clusters or belong to small clusters. We use the following steps to find and visualize anomalies.

Calculate the distance between each point and its nearest centroid. The biggest distances are considered as anomaly.

We use outliers_fraction to provide information to the algorithm about the proportion of the outliers present in our data set.

Situations may vary from data set to data set. However, as a starting figure, I estimate outliers_fraction=0.13 (13% of df are outliers as depicted).

Calculate number_of_outliers using outliers_fraction.

Set threshold as the minimum distance of these outliers.

The anomaly result of anomaly1 contains the above method Cluster (0:normal, 1:anomaly).

Visualize anomalies with Time Series view

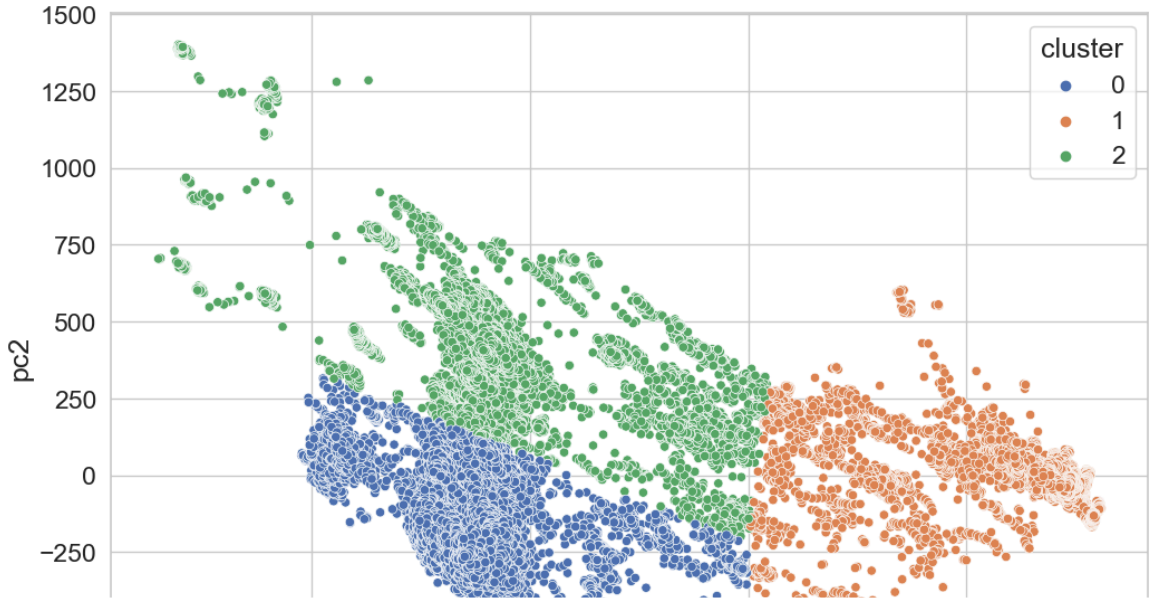
In [55]:
```
#pip install --upgrade scikit-learn
```

In [56]:
```
#pip install threadpoolctl==3.1.0
```

In [66]:
```
# Import necessary libraries
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs
```

In [67]:
```
# I will start k-means clustering with k=3 as I already know that ther
kmeans =KMeans(n_clusters=3).fit(principalDf.values)
principalDf['cluster'] = pd.Categorical(kmeans.labels_)
sns.scatterplot(x="pc1",y="pc2",hue="cluster",data=principalDf)
```

Out[67]: <AxesSubplot:xlabel='pc1', ylabel='pc2'>



In [68]:
```
# Write a function that calculates distance between each point and the
def getDistanceByPoint(data, model):
    """ Function that calculates the distance between a point and cent
            returns the distances in pandas series"""
    distance = []
    for i in range(0,len(data)):
        Xa = np.array(data.loc[i])
        Xb = model.cluster_centers_[model.labels_[i]-1]
        distance.append(np.linalg.norm(Xa-Xb))
    return pd.Series(distance, index=data.index)
```

In [69]:
```
# Assume that 13% of the entire data set are anomalies
outliers_fraction = 0.13
```

In [70]:
```
# get the distance between each point and its nearest centroid. The bi
distance = getDistanceByPoint(principalDf,kmeans)
```

In [71]:
```
# number of observations that equate to the 13% of the entire data set
number_of_outliers = int(outliers_fraction*len(distance))
```

In [72]:
```
# Take the minimum of the largest 13% of the distances as the threshol
threshold = distance.nlargest(number_of_outliers).min()
```

In [73]:
```
# anomaly1 contain the anomaly result of the above method Cluster (0:r
principalDf['anomaly1'] = (distance >= threshold).astype(int)
principalDf['anomaly1']
```

Out[73]:
```
0         0
1         0
2         0
3         0
4         0
         ..
220315    0
220316    0
220317    0
220318    0
220319    0
Name: anomaly1, Length: 220320, dtype: int64
```

In [74]:
```
principalDf.head()
```

Out[74]:

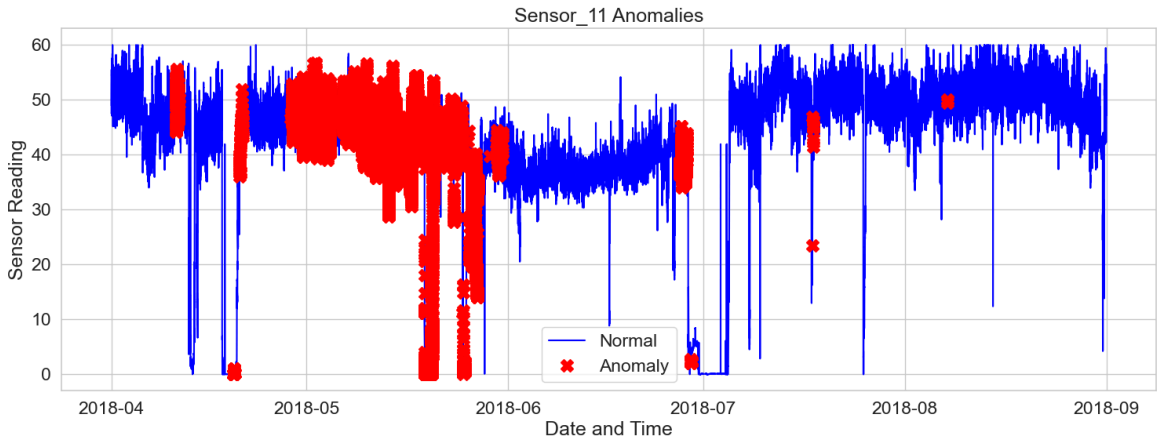
	pc1	pc2	cluster	anomaly1
0	69.523134	265.704571	2	0
1	69.523134	265.704571	2	0
2	27.841569	283.462197	2	0
3	24.548981	290.213914	2	0
4	29.627090	294.619639	2	0

In [75]:
```
principalDf['anomaly1'].value_counts()
```

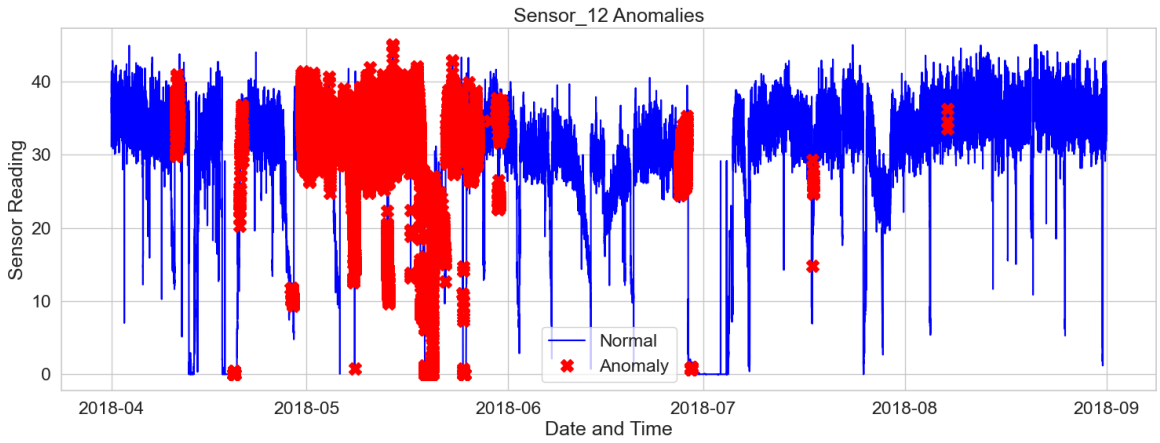
Out[75]:
```
0    191679
1     28641
Name: anomaly1, dtype: int64
```

In [78]:
```
sensor_df['anomaly1'] = pd.Series(principalDf['anomaly1'].values, inde
```

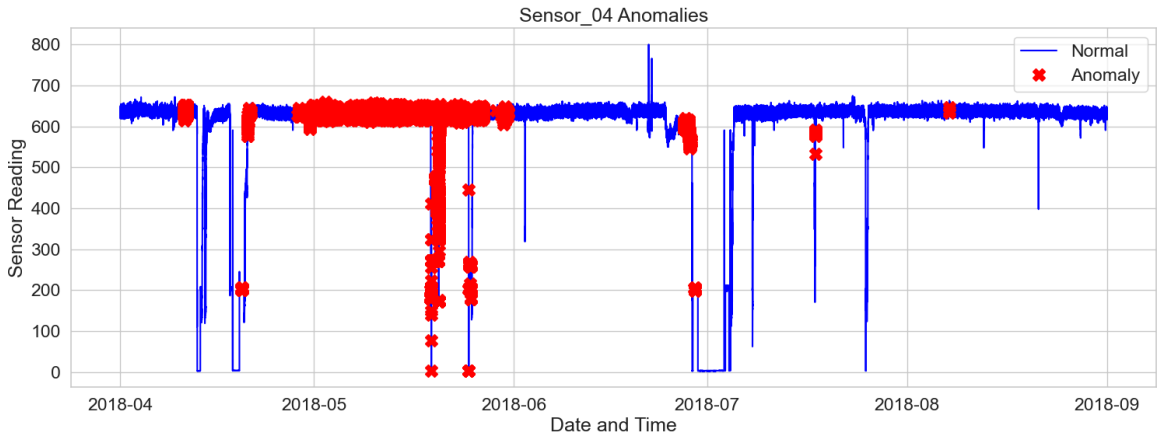
In [79]:
```
# Let's plot the annomaly1 on top of the sensor_11 see where they occu
a = sensor_df[sensor_df['anomaly1'] == 1] #anomaly
_ = plt.figure(figsize=(18,6))
_ = plt.plot(sensor_df['sensor_11'], color='blue', label='Normal')
_ = plt.plot(a['sensor_11'], linestyle='none', marker='X', color='red'
_ = plt.xlabel('Date and Time')
_ = plt.ylabel('Sensor Reading')
_ = plt.title('Sensor_11 Anomalies')
_ = plt.legend(loc='best')
plt.show();
```



In [80]:
```
# Let's plot the outliers from pc1 on top of the sensor_12 see where t

a = sensor_df[sensor_df['anomaly1'] == 1] #anomaly
_ = plt.figure(figsize=(18,6))
_ = plt.plot(sensor_df['sensor_12'], color='blue', label='Normal')
_ = plt.plot(a['sensor_12'], linestyle='none', marker='X', color='red'
_ = plt.xlabel('Date and Time')
_ = plt.ylabel('Sensor Reading')
_ = plt.title('Sensor_12 Anomalies')
_ = plt.legend(loc='best')
plt.show();
```



In [81]:
```
# Let's plot anomaly1 on top of the sensor_04 see where they occured i
a = sensor_df[sensor_df['anomaly1'] == 1] #anomaly
_ = plt.figure(figsize=(18,6))
_ = plt.plot(sensor_df['sensor_04'], color='blue', label='Normal')
_ = plt.plot(a['sensor_04'], linestyle='none', marker='X', color='red'
_ = plt.xlabel('Date and Time')
_ = plt.ylabel('Sensor Reading')
_ = plt.title('Sensor_04 Anomalies')
_ = plt.legend(loc='best')
plt.show();
```



In [82]:
```
sensor_df[sensor_df['anomaly1']==1]['machine_status'].value_counts()
```

Out[82]:
```
1    27310
2     1330
0        1
Name: machine_status, dtype: int64
```

In [84]:
```
sensor_df['machine_status'].value_counts()
```

Out[84]:
```
1    205836
2     14477
0         7
Name: machine_status, dtype: int64
```

Model 3: Isolation Forest

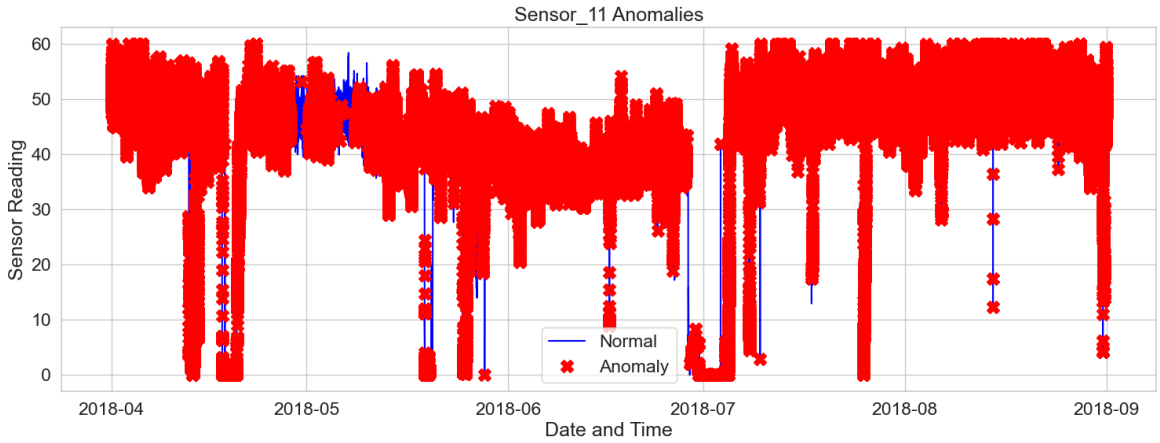
Isolation Forest is a technique for identifying outliers in data.The approach employs binary trees to detect anomalies, resulting in a linear time complexity and low memory usage that is well-suited for processing large datasets.It is an unsupervised model, built on decision Trees.

In [85]:
```
# Import IsolationForest
from sklearn.ensemble import IsolationForest
```

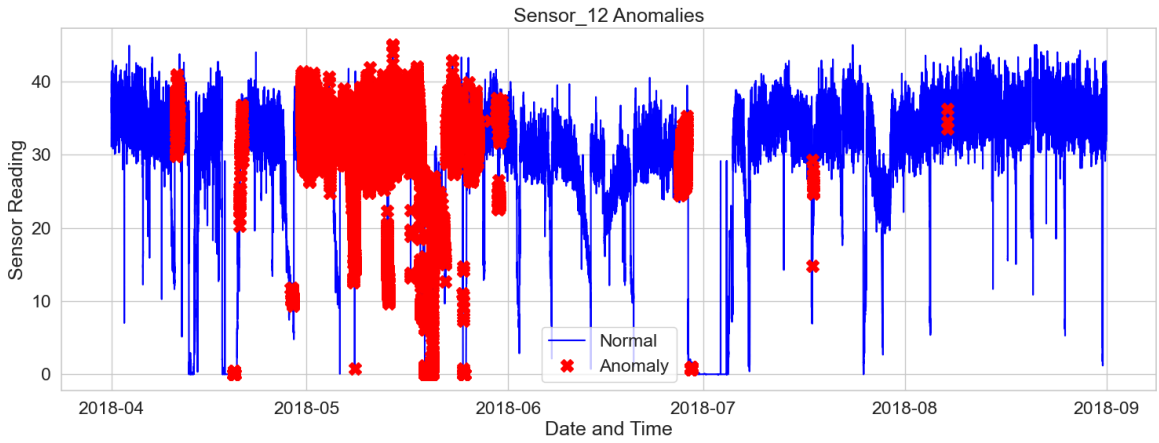
In [86]:
```
# Assume that 13% of the entire data set are anomalies

outliers_fraction = 0.13
model =  IsolationForest(contamination=outliers_fraction)
model.fit(principalDf.values)
principalDf['anomaly2'] = pd.Series(model.predict(principalDf.values))
```

In [87]:
```
# visualization
sensor_df['anomaly2'] = pd.Series(principalDf['anomaly2'].values, inde
```

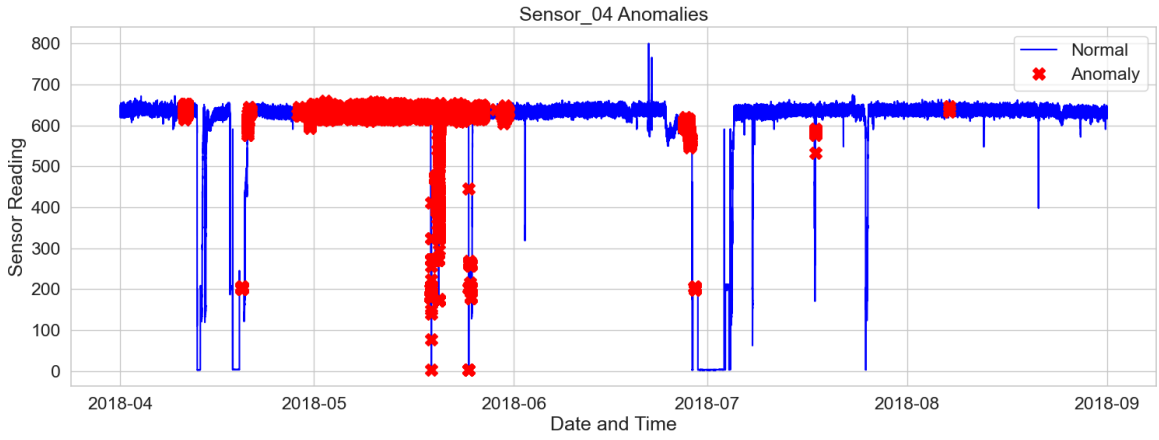
In [88]:
```
a = sensor_df.loc[sensor_df['anomaly2'] == 1] #anomaly
_ = plt.figure(figsize=(18,6))
_ = plt.plot(sensor_df['sensor_11'], color='blue', label='Normal')
_ = plt.plot(a['sensor_11'], linestyle='none', marker='X', color='red'
_ = plt.xlabel('Date and Time')
_ = plt.ylabel('Sensor Reading')
_ = plt.title('Sensor_11 Anomalies')
_ = plt.legend(loc='best')
plt.show();
```



In [89]:
```
# visualization
a = sensor_df[sensor_df['anomaly1'] == 1]
_ = plt.figure(figsize=(18,6))
_ = plt.plot(sensor_df['sensor_12'], color='blue', label='Normal')
_ = plt.plot(a['sensor_12'], linestyle='none', marker='X', color='red'
_ = plt.xlabel('Date and Time')
_ = plt.ylabel('Sensor Reading')
_ = plt.title('Sensor_12 Anomalies')
_ = plt.legend(loc='best')
plt.show();
```



In [90]:
```
# visualization
a = sensor_df[sensor_df['anomaly1'] == 1]
_ = plt.figure(figsize=(18,6))
_ = plt.plot(sensor_df['sensor_04'], color='blue', label='Normal')
_ = plt.plot(a['sensor_04'], linestyle='none', marker='X', color='red'
_ = plt.xlabel('Date and Time')
_ = plt.ylabel('Sensor Reading')
_ = plt.title('Sensor_04 Anomalies')
_ = plt.legend(loc='best')
plt.show();
```



In [91]:
```
sensor_df[sensor_df['anomaly1']==1]['machine_status'].value_counts()
```

Out[91]:
```
1    27310
2     1330
0        1
Name: machine_status, dtype: int64
```

In [92]:
```
sensor_df['anomaly2'].value_counts()
```

Out[92]:
```
 1    191678
-1     28642
Name: anomaly2, dtype: int64
```

In [93]:
```
sensor_df[sensor_df['anomaly2']==1]['machine_status'].value_counts()
```

Out[93]:
```
1    179239
2     12433
0         6
Name: machine_status, dtype: int64
```

In [94]:
```
sensor_df['machine_status'].value_counts()
```

Out[94]:
```
1    205836
2     14477
0         7
Name: machine_status, dtype: int64
```

Model Evaluation

It is interesting to see that all three models detected a lot of the similar anomalies. From the above analysis, we see that IQR is detecting far more anomalies than that of K-Means and Isolation Forest.

Conclusion

So far, we have done anomaly detection with three different methods. In doing so, we went through most of the steps of the commonly applied Data Science Process which includes the following steps:

1)Problem identification

2)Data wrangling

3)Exploratory data analysis

4)Pre-processing and training data development

5)Modeling

6) Model Evaluation

One of the challenges I faced during this project is that training anomaly detection models with unsupervised learning algorithms with such a large data set can be computationally very expensive.

I suggest the following next steps :

1)Feature selection with Feature engineering technique like Chi Square

2)Use features with high Importance and create a new dataframe

3)Implement other classification algorithms such as SVM, Random Forest, XGBBoost.

4)Predict the machine status with the best model given a test set

Other Classification Models

Chi-square Test for Feature Selection:

We calculate Chi-square between each feature and the target and select the desired number of features with best Chi-square scores. Features that show significant dependencies with the target variable are considered important for prediction and can be selected for further analysis.

I have created a new data Frame with features that have high importance from Chi Square Test and I will use this data for further analysis.

In [95]:
```
#DataFrame with the features selected from Chi Square

selected_columns=['sensor_00','sensor_01','sensor_02','sensor_03','ser
df2 = sensor_df[selected_columns]
df2
```

Out[95]:

timestamp	sensor_00	sensor_01	sensor_02	sensor_03	sensor_04	sensor_05	sensor_06	se
2018-04-01 00:00:00	2.465394	47.09201	53.211800	46.310760	634.375000	76.45975	13.41146	
2018-04-01 00:01:00	2.465394	47.09201	53.211800	46.310760	634.375000	76.45975	13.41146	
2018-04-01 00:02:00	2.444734	47.35243	53.211800	46.397570	638.888900	73.54598	13.32465	
2018-04-01 00:03:00	2.460474	47.09201	53.168400	46.397568	628.125000	76.98898	13.31742	
2018-04-01 00:04:00	2.445718	47.13541	53.211800	46.397568	636.458300	76.58897	13.35359	
...	...	...	...	...	...	...	...	
2018-08-31 23:55:00	2.407350	47.69965	50.520830	43.142361	634.722229	64.59095	15.11863	
2018-08-31 23:56:00	2.400463	47.69965	50.564240	43.142361	630.902771	65.83363	15.15480	
2018-08-31 23:57:00	2.396528	47.69965	50.520830	43.142361	625.925903	67.29445	15.08970	
2018-08-31 23:58:00	2.406366	47.69965	50.520832	43.142361	635.648100	65.09175	15.11863	
2018-08-31 23:59:00	2.396528	47.69965	50.520832	43.142361	639.814800	65.45634	15.11863	

220320 rows × 16 columns

Train Test Split

In [96]:
```
# Let's extract the selected features (sensors) from data. Divide the

X = df2.drop(columns=['machine_status'])
y = df2['machine_status']
```

In [97]:
```
# 75% of dataset is used for training and the remaining 25% is used fo

from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X,y,test_size=0.25
```

Scaling the data

#Data is scaled using Min-Max Scaler ,so all the data is transformed within the given range without changing the shape of the original distribution.

In [98]:
```
from sklearn.preprocessing import MinMaxScaler

scaler = MinMaxScaler()

scaler.fit(X_train)
```

Out[98]:
```
MinMaxScaler()
```

In [99]:
```
X_train = scaler.transform(X_train)
X_test = scaler.transform(X_test)
```

In [100]:
```
X_train
```

Out[100]:
```
array([[0.96256281, 0.8156683 , 0.77798846, ..., 0.16648649, 0.07072
043,
        0.19047623],
       [0.00154381, 0.83026119, 0.63757084, ..., 0.01621621, 0.01850
628,
        0.01279762],
       [0.94442326, 0.8156683 , 0.70398461, ..., 0.07999999, 0.05287
508,
        0.19226193],
       ...,
       [0.93014285, 0.85023052, 0.73624278, ..., 0.52324317, 0.15135
492,
        0.23511911],
       [0.96758004, 0.8540707 , 0.6925997 , ..., 0.33783775, 0.08790
481,
        0.16488095],
       [0.96101908, 0.80721973, 0.77039837, ..., 0.48648646, 0.09451
423,
        0.2142857 ]])
```

In [101]:
```
X_test
```

Out[101]:
```
array([[0.97105393, 0.79877116, 0.81593887, ..., 0.26432435, 0.06146
729,
        0.18154761],
       [0.98494792, 0.89247309, 0.77229578, ..., 0.12324323, 0.06212
822,
        0.16488095],
       [0.93064179, 0.67972361, 0.24288394, ..., 0.00972973, 0.00660
939,
        0.00416666],
       ...,
       [0.9648786 , 0.86712759, 0.85578713, ..., 0.38648646, 0.06477
198,
        0.18333331],
       [0.94712469, 0.88786491, 0.79696388, ..., 0.33783775, 0.08790
481,
        0.18988099],
       [0.95677352, 0.8172043 , 0.78937336, ..., 0.09351351, 0.03569
068,
        0.14999999]])
```

The above shows the scaled data. It is now ready for Modelling.

Modelling:

I have used the following Classification models for predicting Machine Failure.

1. Random Forest Classifier
2. KNN(K Nearest Neighbour)
3. Support Vector Machines(SVM)
4. XGBoost

In [102]:
```
from sklearn.metrics import confusion_matrix
from sklearn.metrics import roc_auc_score
```

Random Forest Classifier

Random forests are for supervised machine learning, where there is a labeled target variable.

Random forests can be used for solving regression (numeric target variable) and classification (categorical target variable) problems.

Random forests are an ensemble method, meaning they combine predictions from other models.

In [103]:
```
from sklearn.ensemble import RandomForestClassifier
rfc = RandomForestClassifier(n_estimators=100, criterion='gini',random
rfc.fit(X_train,y_train)
```

Out[103]:
```
RandomForestClassifier(max_depth=6, random_state=23)
```

In [104]:
```
predictions = rfc.predict(X_test)
```

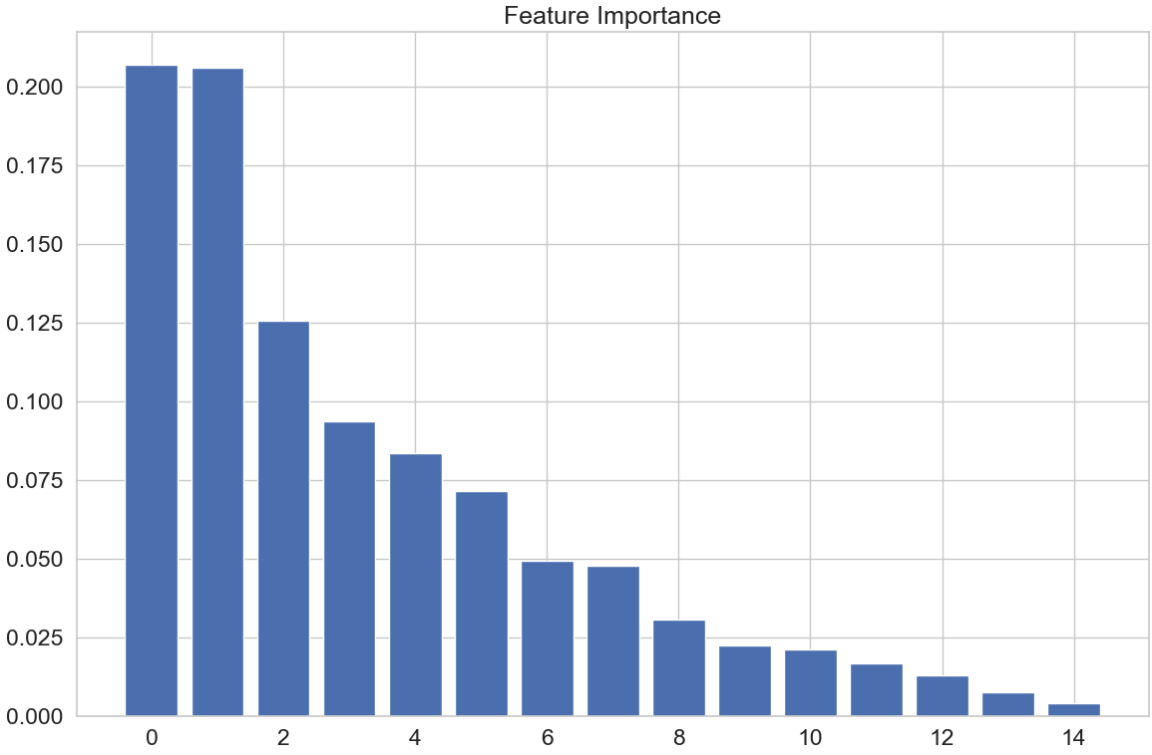
In [105]:
```
rfc.feature_importances_
```

Out[105]:
```
array([0.03054672, 0.12562862, 0.0715204 , 0.04934806, 0.20599328,
       0.08361915, 0.02112161, 0.00758089, 0.01293088, 0.00404673,
       0.04781037, 0.02246334, 0.20699165, 0.0936242 , 0.01677409])
```

In [106]:
```
import numpy as np

importances = rfc.feature_importances_
#
# Sort the feature importance in descending order
#
sorted_indices = np.argsort(importances)[::-1]

feat_labels = df2.columns[1:]

for f in range(X_train.shape[1]):
    print("%2d) %-*s %f" % (f + 1, 30,
                            feat_labels[sorted_indices[f]],
                            importances[sorted_indices[f]]))
```

```
 1) sensor_49                      0.206992
 2) sensor_05                      0.205993
 3) sensor_02                      0.125629
 4) sensor_51                      0.093624
 5) sensor_06                      0.083619
 6) sensor_03                      0.071520
 7) sensor_04                      0.049348
 8) sensor_47                      0.047810
 9) sensor_01                      0.030547
10) sensor_48                      0.022463
11) sensor_07                      0.021122
12) machine_status                 0.016774
13) sensor_09                      0.012931
14) sensor_08                      0.007581
15) sensor_46                      0.004047
```

In [107]:
```
#Plot showing Feature Importance

plt.title('Feature Importance')
plt.bar(range(X_train.shape[1]), importances[sorted_indices], align='c
#plt.xticks(range(X_train.shape[1]),X_train.columns[sorted_indices], r
plt.tight_layout()
plt.show()
```



From the above Feature importance plot, we ca find the most important features that will help in predicting the output .

In [108]:
```
from sklearn.metrics import classification_report,confusion_matrix

print(classification_report(y_test,predictions))
```

```
              precision    recall  f1-score   support

           0       0.00      0.00      0.00         4
           1       1.00      1.00      1.00     51477
           2       1.00      0.99      0.99      3599

    accuracy                           1.00     55080
   macro avg       0.67      0.66      0.66     55080
weighted avg       1.00      1.00      1.00     55080
```

In [109]:
```
import sklearn.metrics as metrics
print("Accuracy score:",metrics.accuracy_score(y_test, predictions))
print("Precision score:",metrics.precision_score(y_test, predictions,a
print("Recall score:",metrics.recall_score(y_test, predictions,average
print("F1 Score :",metrics.f1_score(y_test, predictions,average='macro
```

```
Accuracy score: 0.9992919389978213
Precision score: 0.6662403464818729
Recall score: 0.6635111684061132
F1 Score : 0.6648691030006689
```

A classification report is a performance evaluation metric in machine learning. It is used to show the precision, recall, F1 Score.

Confusion matrix

A confusion matrix is a tool for summarizing the performance of a classification algorithm. A confusion matrix will give us a clear picture of classification model performance and the types of errors produced by the model. It gives us a summary of correct and incorrect predictions broken down by each category. The summary is represented in a tabular form.

Four types of outcomes are possible while evaluating a classification model performance. These four outcomes are described below:-

True Positives (TP) – True Positives occur when we predict an observation belongs to a certain class and the observation actually belongs to that class.

True Negatives (TN) – True Negatives occur when we predict an observation does not belong to a certain class and the observation actually does not belong to that class.

False Positives (FP) – False Positives occur when we predict an observation belongs to a certain class but the observation actually does not belong to that class. This type of error is called Type I error.

False Negatives (FN) – False Negatives occur when we predict an observation does not belong to a certain class but the observation actually belongs to that class. This is a very serious error and it is called Type II error.

These four outcomes are summarized in a confusion matrix given below.

In [110]:
```
print(confusion_matrix(y_test,predictions))
```

```
[[    0     3     1]
 [    0 51476     1]
 [    0    34  3565]]
```

KNN(K Nearest Neighbour)

The K-nearest neighbors algorithm, also called KNN or k-NN, is a non-parametric, supervised learning classifier that uses the concept of proximity to make classifications or predictions about the grouping of a single data point.It assumes that similar points can be found near each other.

Here, I have used k=3 since we have 3 label classes.

In [111]:
```
from sklearn.neighbors import KNeighborsClassifier
```

In [112]:
```
knn = KNeighborsClassifier(n_neighbors=3)
```

In [113]:
```
knn.fit(X_train,y_train)
```

Out[113]:
```
KNeighborsClassifier(n_neighbors=3)
```

In [114]:
```
pred = knn.predict(X_test)
```

In [115]:
```
from sklearn.metrics import classification_report,confusion_matrix
```

In [116]:
```
print(confusion_matrix(y_test,pred))
```

```
[[    0     3     1]
 [    0 51475     2]
 [    0     7  3592]]
```

In [117]:
```
print(classification_report(y_test,pred))
```

```
              precision    recall  f1-score   support

           0       0.00      0.00      0.00         4
           1       1.00      1.00      1.00     51477
           2       1.00      1.00      1.00      3599

    accuracy                           1.00     55080
   macro avg       0.67      0.67      0.67     55080
weighted avg       1.00      1.00      1.00     55080
```

In [118]:
```
import sklearn.metrics as metrics
print("Accuracy score:",metrics.accuracy_score(y_test, pred))
print("Precision score:",metrics.precision_score(y_test, pred,average=
print("Recall score:",metrics.recall_score(y_test, pred,average='macro
print("F1 Score :",metrics.f1_score(y_test, pred,average='macro'))
```

```
Accuracy score: 0.9997639796659404
Precision score: 0.6663237587733333
Recall score: 0.6660053876596842
F1 Score : 0.6661644682952624
```

SVM (Support Vector Machines)

A support vector machine (SVM) is a machine learning algorithm that uses supervised learning models to solve complex classification, regression, and outlier detection problems by performing optimal data transformations that determine boundaries between data points based on predefined classes, labels, or outputs.

The primary objective of the SVM algorithm is to identify a hyperplane that distinguishably segregates the data points of different classes. The hyperplane is localized in such a manner that the largest margin separates the classes under consideration.

In this project, I have used the Radial basis kernel(rbf). Radial Basis Function, to measure the similarity between pairs of data points in the feature space.

In [119]:
```
from sklearn.svm import SVC

model = SVC(kernel='rbf')
```

In [120]:
```
model.fit(X_train,y_train)
```

Out[120]:
```
SVC()
```

In [121]:
```
predictions = model.predict(X_test)
```

In [122]:
```
from sklearn.metrics import classification_report,confusion_matrix
```

In [123]:
```
print(confusion_matrix(y_test,predictions))
```

```
[[    0     2     2]
 [    0 51451    26]
 [    0    34  3565]]
```

In [124]:
```
print(classification_report(y_test,predictions))
```

```
              precision    recall  f1-score   support

           0       0.00      0.00      0.00         4
           1       1.00      1.00      1.00     51477
           2       0.99      0.99      0.99      3599

    accuracy                           1.00     55080
   macro avg       0.66      0.66      0.66     55080
weighted avg       1.00      1.00      1.00     55080
```

In [125]:
```
print("Accuracy score:",metrics.accuracy_score(y_test, predictions))
print("Precision score:",metrics.precision_score(y_test, predictions,a
print("Recall score:",metrics.recall_score(y_test, predictions,average
print("F1 Score :",metrics.f1_score(y_test, predictions,average='macro
```

```
Accuracy score: 0.9988380537400146
Precision score: 0.6638359545588995
Recall score: 0.6633492838104038
F1 Score : 0.6635923860454296
```

XGBoost

XGBoost classifier is a machine learning algorithm that is applied for structured and tabular data . XGBoost is an implementation ofgradient boostedDecision trees designed for speed and performance. XGBoost is an extreme gradient boost algorithm.

In [126]:
```
#!pip install xgboost
```

In [127]:
```
import xgboost as xgb
from xgboost import XGBClassifier
```

In [128]:
```
# convet the target column
df2['machine_status'].replace([2,1,0],['Recovering','Broken','Normal']
```

In [129]:
```

model = XGBClassifier()
model.fit(X_train, y_train)
```

Out[129]:
```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=None, device=None, early_stopping_rou
nds=None,
              enable_categorical=False, eval_metric=None, feature_ty
pes=None,
              gamma=None, grow_policy=None, importance_type=None,
              interaction_constraints=None, learning_rate=None, max_
bin=None,
              max_cat_threshold=None, max_cat_to_onehot=None,
              max_delta_step=None, max_depth=None, max_leaves=None,
              min_child_weight=None, missing=nan, monotone_constrain
ts=None,
              multi_strategy=None, n_estimators=None, n_jobs=None,
              num_parallel_tree=None, objective='multi:softprob',
...)
```

In [130]:
```
y_pred = model.predict(X_test)
```

In [131]:
```
from sklearn.metrics import classification_report,confusion_matrix
```

In [132]:
```
print(confusion_matrix(y_test,predictions))
```

```
[[    0     2     2]
 [    0 51451    26]
 [    0    34  3565]]
```

In [133]:
```
print(classification_report(y_test,predictions))
```

```
              precision    recall  f1-score   support

           0       0.00      0.00      0.00         4
           1       1.00      1.00      1.00     51477
           2       0.99      0.99      0.99      3599

    accuracy                           1.00     55080
   macro avg       0.66      0.66      0.66     55080
weighted avg       1.00      1.00      1.00     55080
```

In [134]:
```
print("Accuracy score:",metrics.accuracy_score(y_test, predictions))
print("Precision score:",metrics.precision_score(y_test, predictions,a
print("Recall score:",metrics.recall_score(y_test, predictions,average
print("F1 Score :",metrics.f1_score(y_test, predictions,average='macro
```

```
Accuracy score: 0.9988380537400146
Precision score: 0.6638359545588995
Recall score: 0.6633492838104038
F1 Score : 0.6635923860454296
```

K- Nearest Neighbour gave the best result from all the above supervised classification models .

Evaluating Models: Accuracy Is Not Enough

As mentioned in the previous section, in anomaly detection applications it is expected that the distribution between the normal and abnormal class(es) may be highly skewed. This is commonly referred to as the class imbalance problem. A model that learns from such skewed data may not be robust; it may be accurate when classifying examples within the normal class, but perform poorly when classifying anomalous examples.

Such a model may also misclassify normal examples as anomalous (false positives, FP), or misclassify anomalous examples as normal ones (false negatives, FN). As we consider both of these types of errors, it becomes obvious that the traditional accuracy metric (total number of correct classifications divided by total classifications) is insufficient in evaluating the skill of an anomaly detection model.

Two important metrics provide a better measure of model skill: precision and recall. Precision is defined as the number of true positives (TP) divided by the number of true positives plus the number of false positives (FP), while recall is the number of true positives divided by the number of true positives plus the number of false negatives (FN). Depending on the use case or application, it may be desirable to optimize for either precision or recall.

Optimizing for precision may be useful when the cost of failure is low, or to reduce human workload. Optimizing for high recall may be more appropriate when the cost of a false negative is very high. While there are several ways to optimize for precision or recall, the manner in which a threshold is set can be used to reflect the precision and recall preferences for each specific use case.

Conclusion and Future Suggestions

Let's review the project:

Data: The data set consists of 51 numerical features ,datetime and a categorical label.The label contains string values that represent normal, broken and recovering operational conditions of the machine. The data set represents 219,521 readings from 51 sensors.

Solution Approach: I will solve this problem by using unsupervised models that detects anomalies from the sensor readings and accordingly predicts the potential failure of the machine. To do that, I will first apply an appropriate unsupervised learning techniques to undertake dimensionality reduction for the effective visualization of the data and EDA. Then I will train classification models and will evaluate their performance to select the best model.

Workflow:

1.First I explored the data given .

2.Then I visualized the data to get a better sense of the patterns.

3.I did data preprocessing by dealing with missing data with the mean of the column.

4.I used PCA to find the most useful features (columns) to use when training the models.

5.Overall I used the following models: Unsupervised models - PCA , KMeans Clusters, Isolation Forest Supervised classification algorithms - KNN,SVM,Random Forest Classifier andXGBoost.

6.Finally, I compared the models performance.

From the unsupervised models ,IQR gave best results and from the supervised models KNN gave the best result.

However, by understanding the data and investigating further, I know that is probably not the case. The data was very unbalanced; almost all the data was labeled NORMAL, then we had some RECOVERING, and very few BROKEN instances. What this means for our models is that it had a hard time identifying anything other than NORMAL. This is important in real-life situations to be aware of. By just looking at the performance test results, I might have thought I got a fantastic model. But the truth is that we were not able to find a single BROKEN instance either the KNN nor the KMeans models.

Contraints: Data set is limited to the sensor readings from a single machine hence may not be the best representation of all the machines. Computing power might become a constraint for effectively visualizing all 51 features at the same time.

Proposed Next Steps:

Every project should recommend ways to enhance their work.

1.I would recommend using more data with more variety. As said, this data was very unbalanced. Ways to do this would be to find more data, create more data, or even artificially create more data.

2.Having more machines to compare to this machine may have helped the algorithms learn better.

3.Feature selection with advanced feature Engineering techniques.

4.Advanced hyperparameter tuning