Photo by Will Myers on Unsplash

# Anomaly Detection in Time Series Sensor Data

Bauyrjan Jyenis Sep 26, 2020 · 14 min read

> *Anomaly detection involves identifying the differences, deviations, and exceptions from the norm in a dataset. It's sometimes referred to as outlier detection.*

its use cases include (but not limited to) detecting fraud transactions, fraudulent insurance claims, cyber attacks to detecting abnormal equipment behaviors.

In this article, I will focus on the application of anomaly detection in the Manufacturing industry which I believe is an industry that lagged far behind in the area of effectively taking advantage of Machine Learning techniques compared to other industries.

## The Problem Description

Manufacturing industry is considered a heavy industry in which they tend to utilize various types of heavy machinery such as giant motors, pumps, pipes, furnaces, conveyor belts, haul trucks, dozers, graders, and electric shovels etc. These are often considered as the most critical assets for their operations. Therefore, the integrity and reliability of these equipment is often the core focus of their Asset Management programs.

The prime reason why they care so much about these assets is that the failure of these equipment often results in production loss that could consequently lead to loss of hundreds of thousands of dollars if not millions depending on the size and scale of the operations. So this is a pretty serious deal for a Maintenance Manager of a manufacturing plant to run a robust Asset Management framework with highly skilled Reliability Engineers to ensure the reliability and availability of these critical assets.

Therefore, the ability to detect anomalies in advance and be able to mitigate risks is a very valuable capability which further allows preventing an unplanned downtime, unnecessary maintenance (condition based vs mandatory maintenance) and will also enable more effective way of managing critical components for these assets. The production loss from unplanned downtime, the cost of unnecessary maintenance and having excess or shortage of critical components translate into serious magnitudes in terms of dollar amount.

In this post, I will implement different anomaly detection techniques in Python with Scikit-learn (aka sklearn) and our goal is going to be to search for anomalies in the

## The Data

It is very hard to find a public data from a manufacturing industry for this particular use case but I was able to find one that is not perfect. The data set contains sensor readings from 53 sensors installed on a pump to measure various behaviors of the pump. This data set can be found here.

First, I will download the data using the following code and Kaggle API

```
!kaggle datasets download -d nphantawee/pump-sensor-data
```

Once downloaded, read the CSV file into the pandas DataFrame with the following code and check out the details of the data.

```
df = pd.read_csv('sensor.csv')
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
```

```
 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
```

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 data types to the correct data type

```
# Entire "sensor_15" column is NaN therefore remove it from data
del df['sensor_15']
# Let's convert the data type of timestamp column to datatime format
import warnings
warnings.filterwarnings("ignore")
df_tidy['date'] = pd.to_datetime(df_tidy['timestamp'])
del df_tidy['timestamp']
```

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. To do that, I'll write a function that calculates the percentage of missing values so I can use the same function multiple times throughout the notebook.

```
# Function that calculates the percentage of missing values
def calc_percent_NAs(df):
    nans =
pd.DataFrame(df.isnull().sum().sort_values(ascending=False)/len(df),
columns=['percent'])
    idx = nans['percent'] > 0
    return nans[idx]

# Let's use above function to look at top ten columns with NaNs
calc_percent_NAs(df).head(10)
```

| | percent |
|---|---|
| **sensor_50** | 0.349569 |
| **sensor_51** | 0.069821 |
| **sensor_00** | 0.046333 |
| **sensor_07** | 0.024741 |
| **sensor_08** | 0.023180 |
| **sensor_06** | 0.021777 |
| **sensor_09** | 0.020856 |
| **sensor_01** | 0.001675 |
| **sensor_30** | 0.001185 |
| **sensor_29** | 0.000327 |

Percentage of missing values by each column

is ready for the next step which is Exploratory Data Analysis. The tidy data set has 52 sensors, machine status column that contains three classes (NORMAL, BROKEN, RECOVERING) which represent normal operating, broken and recovering conditions of the pump respectively and then the datetime column which represents the timestamp.

| date | sensor_00 | sensor_01 | sensor_02 | sensor_03 | sensor_04 | sensor_05 | sensor_06 | sensor_07 | sensor_08 | sensor_09 | ... | sensor_47 | sensor_48 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 2018-04-01 00:00:00 | 2.465394 | 47.09201 | 53.2118 | 46.310760 | 634.3750 | 76.45975 | 13.41146 | 16.13136 | 15.56713 | 15.05353 | ... | 38.194440 | 157.9861 |
| 2018-04-01 00:01:00 | 2.465394 | 47.09201 | 53.2118 | 46.310760 | 634.3750 | 76.45975 | 13.41146 | 16.13136 | 15.56713 | 15.05353 | ... | 38.194440 | 157.9861 |
| 2018-04-01 00:02:00 | 2.444734 | 47.35243 | 53.2118 | 46.397570 | 638.8889 | 73.54598 | 13.32465 | 16.03733 | 15.61777 | 15.01013 | ... | 38.194443 | 155.9606 |
| 2018-04-01 00:03:00 | 2.460474 | 47.09201 | 53.1684 | 46.397568 | 628.1250 | 76.98898 | 13.31742 | 16.24711 | 15.69734 | 15.08247 | ... | 38.194440 | 155.9606 |
| 2018-04-01 00:04:00 | 2.445718 | 47.13541 | 53.2118 | 46.397568 | 636.4583 | 76.58897 | 13.35359 | 16.21094 | 15.69734 | 15.08247 | ... | 38.773150 | 158.2755 |

The first 10 rows of the tidy data

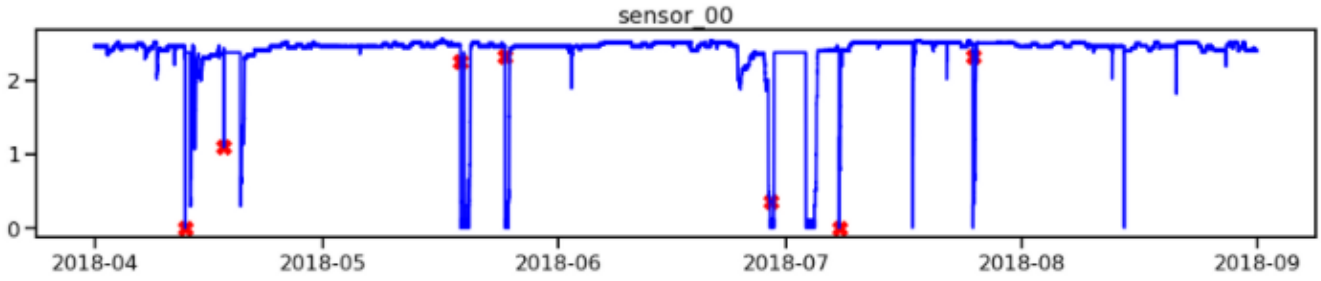
## Exploratory Data Analysis (EDA)

Now that we have cleaned our data, we 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 particular, it is interesting to see 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 pump breaks down and how that reflects in the sensor readings. The following code plots the mentioned graph for each of the sensors, but let’s take a look at that for the sensor_00.

```
# Extract the readings from the BROKEN state of the pump
broken = df[df['machine_status']=='BROKEN']
# Extract the names of the numerical columns
df2 = df.drop(['machine_status'], axis=1)
names=df2.columns
# Plot time series for each sensor with BROKEN state marked with X
in red color
```

```
    _ = plt.plot(df[name], color='blue')
    _ = plt.title(name)
    plt.show()
```

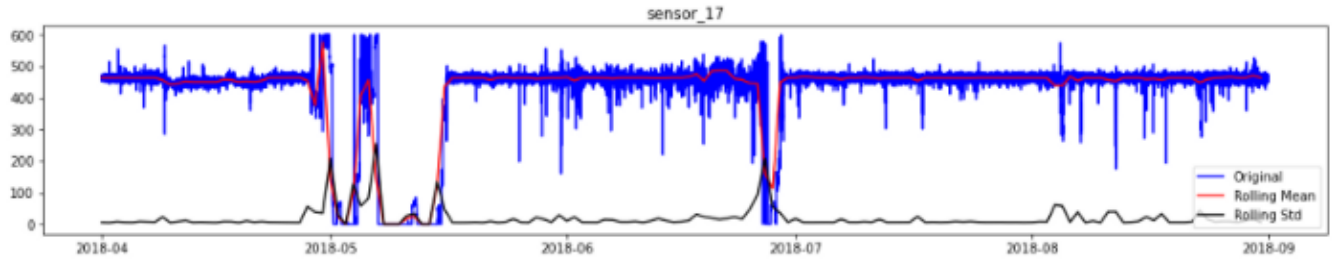



As seen clearly from the above plot, the red marks, which represent the broken state of the pump, perfectly overlaps with the observed disturbances of the sensor reading. Now we have a pretty good intuition about how each of the sensor reading behaves when the pump is broken vs operating normally.

## Stationarity and Autocorrelation

In time series analysis, it is important that the data is stationary and have no autocorrelation. Stationarity refers to the behavior where the mean and standard deviation of the data changes over time, the data with such behavior is considered not stationary. On the other hand, autocorrelation refers to the behavior of the data where the data is correlated with itself in a different time period. As the next step, I will visually inspect the stationarity of each feature in the data set and the following code will do just that. Later, we will also perform the Dickey Fuller test to quantitatively verify the observed stationarity. In addition, we will inspect the autocorrelation of the features before feeding them into the clustering algorithms to detect anomalies.

```
# Resample the entire dataset by daily average
rollmean = df.resample(rule='D').mean()
rollstd = df.resample(rule='D').std()
# Plot time series for each sensor with its mean and standard
deviation
for name in names:
    _ = plt.figure(figsize=(18,3))
    _ = plt.plot(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')
```




Time series looks pretty stationary

Looking at the readings from one of the sensors, sensor_17 in this case, notice that 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.

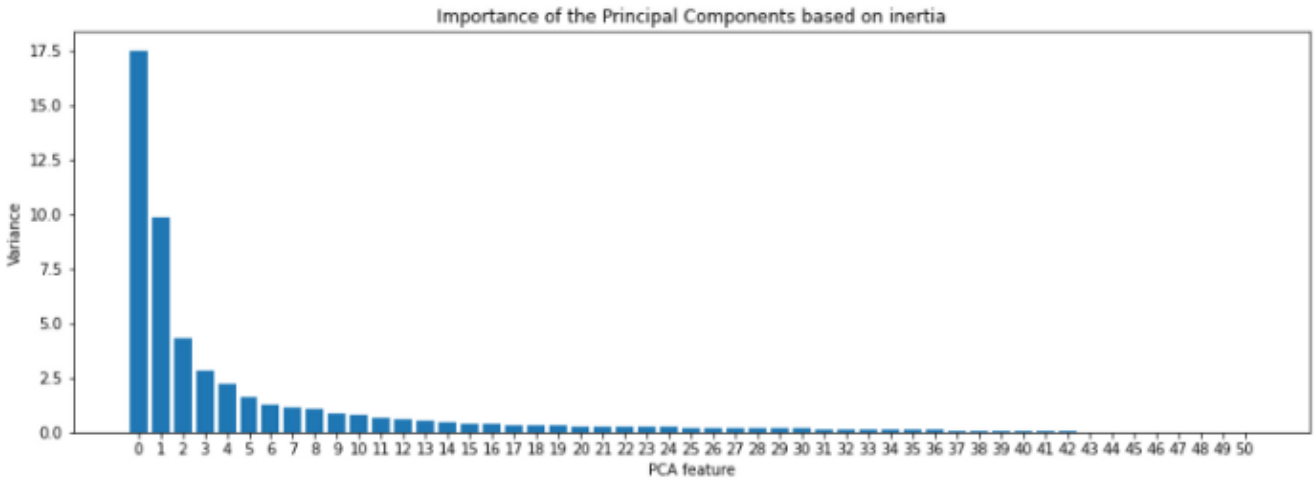
## Pre-Processing and Dimensionality Reduction

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 tidy 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

```
# Standardize/scale the dataset and apply PCA
from sklearn.preprocessing import StandardScaler
from sklearn.decomposition import PCA
```

```
names=df2.columns
x = df[names]
scaler = StandardScaler()
pca = PCA()
pipeline = make_pipeline(scaler, pca)
pipeline.fit(x)

# Plot the principal components against their inertia

features = range(pca.n_components_)
_ = plt.figure(figsize=(15, 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()
```




As per the graph, the first two components are the most important

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.

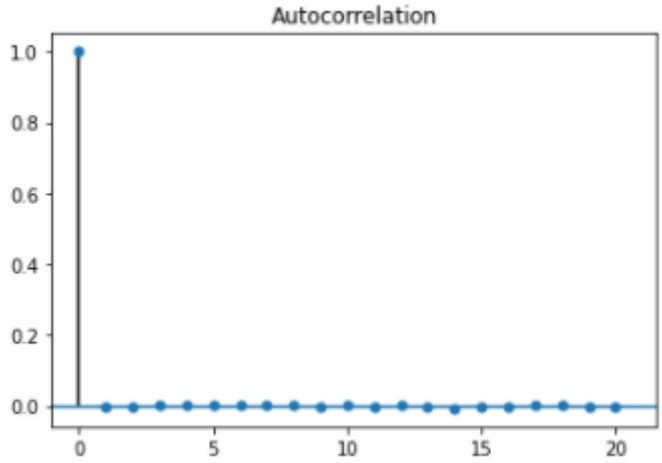
```
# Calculate PCA with 2 components
pca = PCA(n_components=2)
principalComponents = pca.fit_transform(x)
principalDf = pd.DataFrame(data = principalComponents, columns =
['pc1', 'pc2'])
```

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

Running the Dickey Fuller test on the 1st principal component, I got a p-value of 5.4536849418486247e-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.

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.

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




Given my new features from PCA are stationary and not autocorrelated, I am ready for modeling.

## Modeling

In this step, I will perform the following 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.

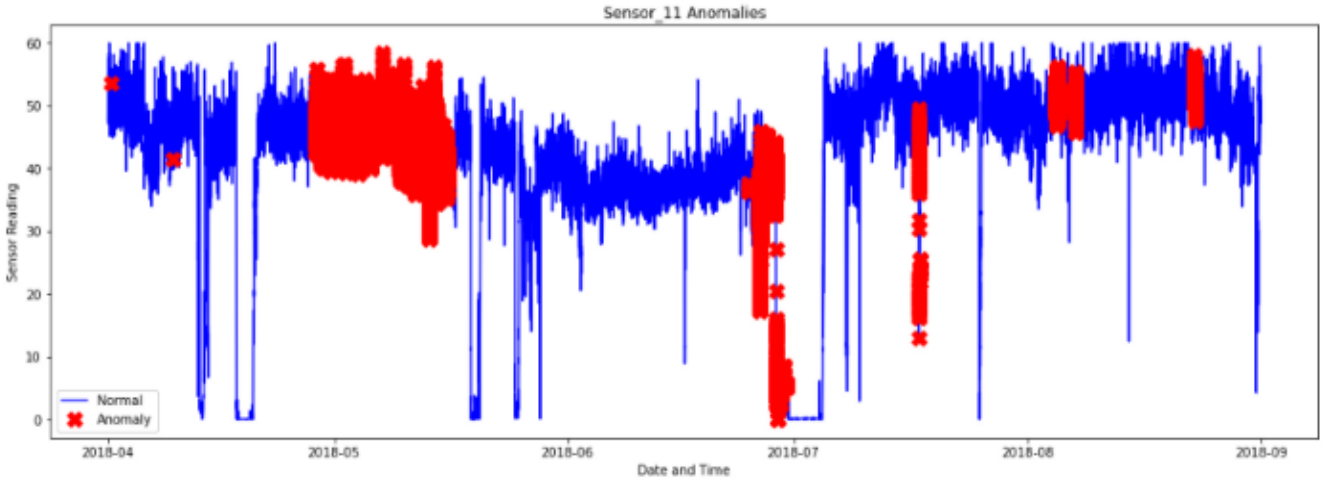
### Interquartile Range

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.
4. Finally, plot the outliers on top of the time series data (the readings from sensor_11 in this case)

```
# Calculate IQR for the 1st principal component (pc1)

q1_pc1, q3_pc1 = df['pc1'].quantile([0.25, 0.75])
iqr_pc1 = q3_pc1 - q1_pc1

# 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)

# Filter out the outliers from the pc1

df['anomaly_pc1'] = ((df['pc1']>upper_pc1) | (df['pc1']
<lower_pc1)).astype('int')

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

```
upper_pc2 = q3_pc2 + (1.5*iqr_pc2)

# Filter out the outliers from the pc2

df['anomaly_pc2'] = ((df['pc2']>upper_pc2) | (df['pc2']
<lower_pc2)).astype('int')

# Let's plot the outliers from pc1 on top of the sensor_11 and see
where they occured in the time series

a = df[df['anomaly_pc1'] == 1] #anomaly
_ = plt.figure(figsize=(18,6))
_ = plt.plot(df['sensor_11'], color='blue', label='Normal')
_ = plt.plot(a['sensor_11'], linestyle='none', marker='X',
color='red', markersize=12, label='Anomaly')
_ = plt.xlabel('Date and Time')
_ = plt.ylabel('Sensor Reading')
_ = plt.title('Sensor_11 Anomalies')
_ = plt.legend(loc='best')
plt.show();
```




Anomalies marked in red color

As seen above, the anomalies are detected right before the pump breaks down. This could be a very valuable information for an operator to see and be able to shut down the pump properly before it actually goes down hard. Let's see if we detect similar pattern in anomalies from the next two algorithms.

## K-Means Clustering

Strategy:

2. 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).

3. Calculate number_of_outliers using outliers_fraction.

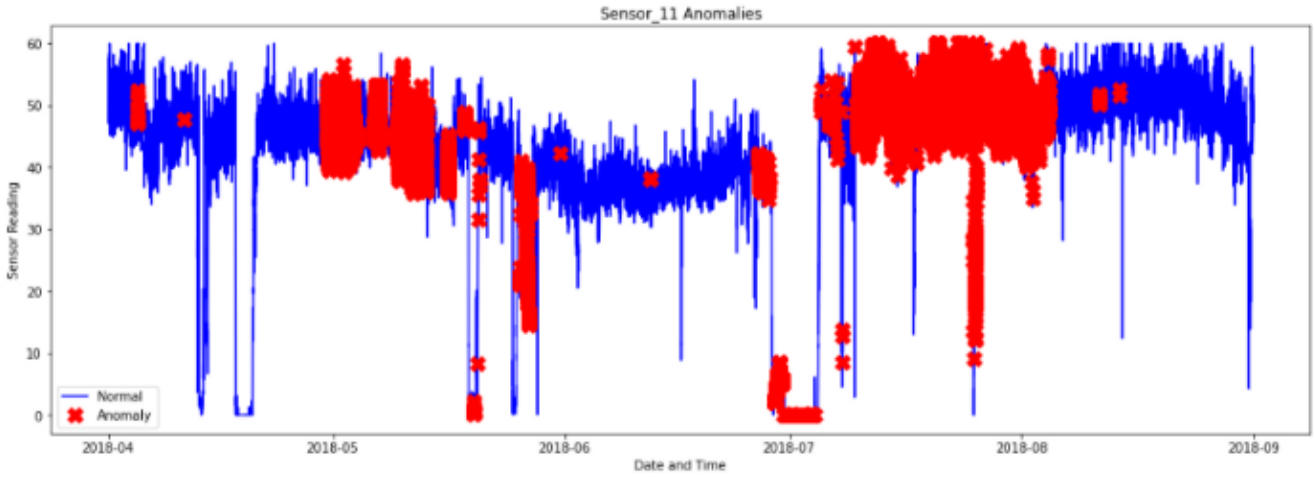
4. Set threshold as the minimum distance of these outliers.

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

6. Visualize anomalies with Time Series view.

```
# Import necessary libraries
from sklearn.cluster import KMeans
# I will start k-means clustering with k=2 as I already know that
there are 3 classes of "NORMAL" vs
# "NOT NORMAL" which are combination of BROKEN" and"RECOVERING"

kmeans = KMeans(n_clusters=2, random_state=42)
kmeans.fit(principalDf.values)
labels = kmeans.predict(principalDf.values)
unique_elements, counts_elements = np.unique(labels,
return_counts=True)
clusters = np.asarray((unique_elements, counts_elements))

# Write a function that calculates distance between each point and
the centroid of the closest cluster

def getDistanceByPoint(data, model):
    """ Function that calculates the distance between a point and
centroid of a cluster,
            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)

# Assume that 13% of the entire data set are anomalies

outliers_fraction = 0.13
```

```
distance = getDistanceByPoint(principalDf, kmeans)

# number of observations that equate to the 13% of the entire data
set

number_of_outliers = int(outliers_fraction*len(distance))

# Take the minimum of the largest 13% of the distances as the
threshold

threshold = distance.nlargest(number_of_outliers).min()

# anomaly1 contain the anomaly result of the above method Cluster
(0:normal, 1:anomaly)

principalDf['anomaly1'] = (distance >= threshold).astype(int)
```

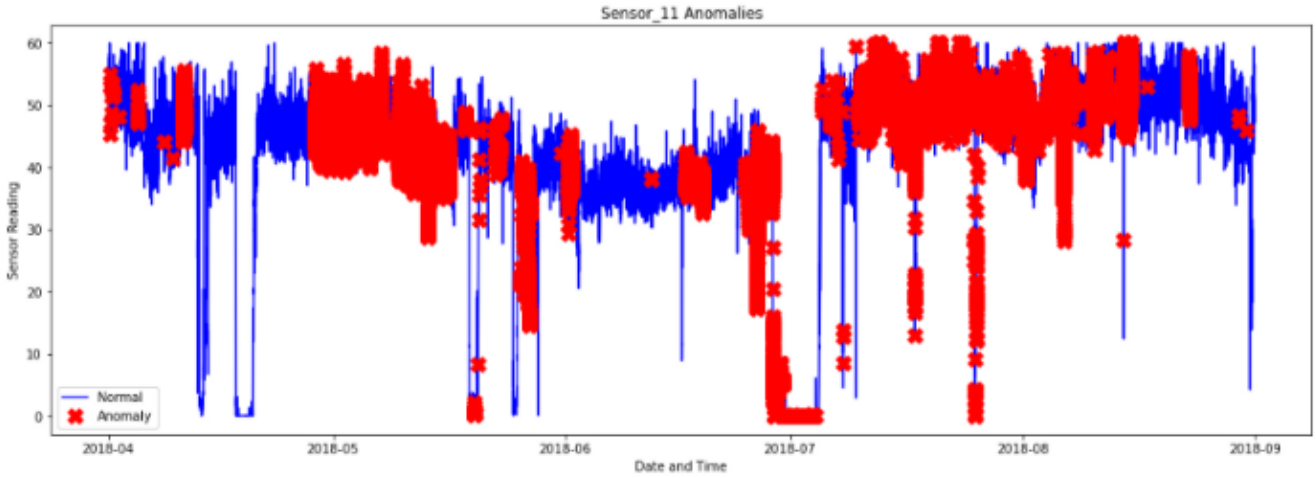



Anomalies are marked in red color

## Isolation Forest

```
# Import IsolationForest

from sklearn.ensemble import IsolationForest

# 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))
```

```
a = df.loc[df['anomaly2'] == -1] #anomaly
_ = plt.figure(figsize=(18,6))
_ = plt.plot(df['sensor_11'], color='blue', label='Normal')
_ = plt.plot(a['sensor_11'], linestyle='none', marker='X',
color='red', markersize=12, label='Anomaly')
_ = plt.xlabel('Date and Time')
_ = plt.ylabel('Sensor Reading')
_ = plt.title('Sensor_11 Anomalies')
_ = plt.legend(loc='best')
plt.show();
```




Anomalies marked in red color

## Model Evaluation

It is interesting to see that all three models detected a lot of the similar anomalies. Just by visually looking at the above graphs, one could easily conclude that the Isolation Forest might be detecting a lot more anomalies than the other two. However, the following tables show, on the contrary, that IQR is detecting far more anomalies than that of K-Means and Isolation Forest.

Anomalies detected by IQR:

| | |
|---|---|
| 0 (NORMAL) | 189,644 |
| 1 (ANOMALY) | 29,877 |

Anomalies detected by K-Means Clustering:

| | |
|---|---|
| 1 (ANOMALY) | 28,537 |

Anomalies detected by Isolation Forest:

| | |
|---|---|
| 0 (NORMAL) | 190,983 |
| 1 (ANOMALY) | 28,538 |

The number of anomalies detected by each of the model

Why do you think that is? Is IQR mostly detecting the anomalies that are closer together while the other two models are detecting the anomalies spread across different time periods? Is IQR more scientific approach than the other two? How do we define accuracy? For now, let me leave you with these questions to think about. I will write more about the model evaluation in more detail in my future posts.

## 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. Documentation

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. For example, I couldn't properly train SVM on this data as it was taking a very long time to train the model with no success.

1. Feature selection with advanced Feature engineering technique
2. Advanced hyperparameter tuning
3. Implement other learning algorithms such as SVM, DBSCAN etc.
4. Predict the machine status with the best model given a test set — BAM!
5. **Deploy the best model into production — DOUBLE BAM!**

I will continue improving the model besides implementing the above mentioned steps and I will plan to share the outcomes in another post in the future.

Jupyter notebook can be found on Github for details. Enjoy detecting anomalies and let's connect on LinkedIn.

## Online References and Useful Materials

- Detecting stationarity in time series data
- A Gentle Introduction to Autocorrelation and Partial Autocorrelation
- Principal Component Analysis
- A One-Stop Shop for Principal Component Analysis
- K-Means Clustering
- Sklearn K-Means Documentation
- Sklearn Isolation Forest Documentatio