VSB Power Line Fault Detection

1.Business/Real-world Problem:

1.1 Source:

This was posted as a kaggle challenge by the ENET Centre which researches and develops renewable energy resources with the goal of reducing or eliminating harmful environmental impacts.

Source: https://www.kaggle.com/c/vsb-power-line-fault-detection/overview (https://www.kaggle.com/c/vsb-power-line-fault-detection/overview)

Data: Enet Centre, VSB — T.U. of Ostrava

1.2 What is Partial Discharge?

Partial Discharge (PD) is a localized dielectric breakdown (DB) (which does not completely bridge the space between the two conductors) of a small portion of a solid or fluid electrical insulation (EI) system under high voltage (HV) stress.

- Wikipedia

1.3 Problem Statement

Overhead power line signals run for 100s of kilometers transferring power from one region to another. These distances make it difficult and expensive to manually inspect for any damages caused to the power lines.These damages lead to a phenomenon called Partial Discharge (PD) in the insulators of the power line.

The main objective of this case study is to detect these partial discharge patterns in signals acquired from lines with a new meter. Effective classifiers using this data will make it possible to continuously monitor power lines for faults.

1.4. Real-world/Business objectives and constraints

• Minimize binary-class error:The cost of missing a PD pattern in a signal is very high .

• probability estimates

• There is no strict latency requirement:As There’s no time limitation as partial discharge faults do damage over time and not immediately so limit can be in hours

• There is no strict interpretability requirement

2. Machine Learning Problem

2.1. Data Overview

The data used to detect the presence of the PD pattern is provided by the VŠB. This data is acquired directly from the overhead power lines with the new meter designed at ENET Centre.

Source:https://www.kaggle.com/c/vsb-power-line-fault-detection/data (https://www.kaggle.com/c/vsb-power-line-fault-detection/data)

Signal data

• The signal data measured directly from the new meter designed at the ENET Center. • Each signal has 800,000 floating-point data. • There are 8,712 signals in the training data and 20,337 signals in the test data. • The signal data is provided in the parquet file format, where each column represents a signal. Therefore, the size of the train signal data is 800000x8712.

Metadata Metadata contains the following information about the signals:

• Signal id — A unique integer used to identify each signal. A signal id of '0' corresponds to column '0' in the signal data.

• Phase id — 3 conductors are used to transfer the power from one region to another. Each conductor carries a signal. The phase of the signal in each conductor is different. The phase id mainly refers to the conductor in which the signal is being carried. Each signal id is associated with a unique phase id of 0, 1, or 2. Please refer to this for more information.

• Measurement id — Since the signals are measured using a meter, each signal is associated with a measurement id. Each measurement id is associated with 3 signals corresponding to the 3 phases of the signal.

• Target — This field provides information on whether a given signal has a PD pattern in it or not. A value of 0 corresponds to the PD pattern not present and a value of 1 corresponds to the PD pattern present for the respective signal id . This field is present only for the training data and has to be determined for the test data.

2.2. Mapping the real-world problem to an ML problem:

The objective of the problem is to check if a PD pattern is present in the given signal (x(t)) or not.Therefore, this is a binary classification problem.

2.3 Performance Metric

The key performance metric used by Kaggle to evaluate the model performance is the Matthews Correlation Coefficient (MCC).

The Matthews correlation coefficient (MCC), instead, is a more reliable statistical rate which produces a high score only if the prediction obtained good results in all of the four confusion matrix categories (true positives, false negatives, true negatives, and false positives).

Matthews correlation coefficient (MCC). As a measure unaffected by the unbalanced datasets issue, the Matthews correlation coefficient is a contingency matrix method of calculating the Pearson product-moment correlation coefficient [22] between actual and predicted values. In terms of the entries of M, MCC reads as follows:

MCC=(TP·TN−FP·FN)/sqrt{(TP+FP)·(TP+FN)·(TN+FP)·(TN+FN)}

2.4. Machine Learning Objectives and Constraints

• Objective: Predict the probability of each data-point belonging to each of the 2 classes.

• Constraints: Class probabilities are needed. Penalize the errors in class probabilities => Metric is Matthews's correlation coefficient.

• Some Latency constraints.

3. Exploratory Data Analysis

EDA Reference:https://www.kaggle.com/go1dfish/basic-eda (https://www.kaggle.com/go1dfish/basic-eda)

In [ ]:

```
from google.colab import files
files.upload()
```

Choose Files No file chosen

Upload widget is only available when the cell has been executed in the current browser session.
Please rerun this cell to enable.

```
Saving kaggle.json to kaggle.json
```

Out[81]:

```
{'kaggle.json': b'{"username":"kusalbera","key":"53ccbf04507c36fc0fcded36e13
0e70c"}'}
```

In [ ]:

```
# Let's make sure the kaggle.json file is present.
!ls -lha kaggle.json
```

```
-rw-r--r-- 1 root root 65 May 23 09:07 kaggle.json
```

In [ ]:

```
# Next, install the Kaggle API client.
!pip install -q kaggle
```

In [ ]:

```
# The Kaggle API client expects this file to be in ~/.kaggle,
# so move it there.
!mkdir -p ~/.kaggle
!cp kaggle.json ~/.kaggle/

# This permissions change avoids a warning on Kaggle tool startup.
!chmod 600 ~/.kaggle/kaggle.json
```

In [ ]:

```
# Download data from Kaggle using its API
!kaggle competitions download -c vsb-power-line-fault-detection
```

```
Downloading vsb-power-line-fault-detection.zip to /content
100% 9.98G/10.0G [03:26<00:00, 46.3MB/s]
100% 10.0G/10.0G [03:26<00:00, 51.9MB/s]
```

In [ ]:

```
!unzip /content/vsb-power-line-fault-detection.zip
```

```
Archive:  /content/vsb-power-line-fault-detection.zip
  inflating: metadata_test.csv
  inflating: metadata_train.csv
  inflating: sample_submission.csv
  inflating: test.parquet
  inflating: train.parquet
```

3.1 Load Libraries

In [ ]:

```
! pip install siml
```

```
Collecting siml
  Downloading siml-0.4.0.tar.gz (12 kB)
Requirement already satisfied: numpy in /usr/local/lib/python3.7/dist-packag
es (from siml) (1.21.6)
Requirement already satisfied: scikit-learn in /usr/local/lib/python3.7/dist
-packages (from siml) (1.0.2)
Requirement already satisfied: threadpoolctl>=2.0.0 in /usr/local/lib/python
3.7/dist-packages (from scikit-learn->siml) (3.1.0)
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/python3.7/dist
-packages (from scikit-learn->siml) (1.1.0)
Requirement already satisfied: scipy>=1.1.0 in /usr/local/lib/python3.7/dist
-packages (from scikit-learn->siml) (1.4.1)
Building wheels for collected packages: siml
  Building wheel for siml (setup.py) ... done
  Created wheel for siml: filename=siml-0.4.0-py3-none-any.whl size=13726 sh
a256=62067c77a89cfb68c58deab85620890b28e61af4e8398ff4fdc87f61a47811ac
  Stored in directory: /root/.cache/pip/wheels/78/4a/0b/6d7bfb04dad3de613c87
0498a7c0132aee2a863b5796a4c10c
Successfully built siml
Installing collected packages: siml
Successfully installed siml-0.4.0
```

In [ ]:

```
#data structures
import pandas as pd
import pyarrow.parquet as pq
import numpy as np


#used for plotting
import seaborn as sns
import matplotlib.pyplot as plt
import plotly.graph_objects as go

#used for feature engineering(signal processing tools)
from scipy.fftpack import fft
from scipy.signal import welch
from siml.sk_utils import *
from siml.signal_analysis_utils import *

from tqdm import tqdm
import ast

import warnings
warnings.filterwarnings('ignore')
```

3.2 Load Data

Importing meta_data

In [ ]:

```
df_metadata_train = pd.read_csv("/content/metadata_train.csv")
df_metadata_test = pd.read_csv("/content/metadata_test.csv")
```

In [ ]:

```
df_metadata_train = df_metadata_train.set_index(['id_measurement', 'phase'])
df_metadata_train.head()
```

Out[20]:

		signal_id	target
id_measurement	phase		
0	0	0	0
	1	1	0
	2	2	0
1	0	3	1
	1	4	1

signal_id - can be used as a key to join the target and signal data
id_measurement - different phases belonging to same signal have same id

phase - Each signal consists of 3 phases
target - partial discharge present or not

In [ ]:

```
df_metadata_test = df_metadata_test.set_index(['signal_id'])
df_metadata_test.head()
```

Out[21]:

signal_id	id_measurement	phase
8712	2904	0
8713	2904	1
8714	2904	2
8715	2905	0
8716	2905	1

In [ ]:

```
print("metadata_train shape is {}".format(df_metadata_train.shape))
print("metadata_test shape is {}".format(df_metadata_test.shape))
```

```
metadata_train shape is (8712, 4)
metadata_test shape is (20337, 3)
```

Checking for null values

In [ ]:

```
print("Number of Null values in the train-metadata:\n", df_metadata_train.isnull().sum())
print("\nNumber of NA values in the train-metadata:\n",df_metadata_train.isna().sum())
```

```
Number of Null values in the train-metadata:
 signal_id         0
id_measurement    0
phase             0
target            0
dtype: int64

Number of NA values in the train-metadata:
 signal_id         0
id_measurement    0
phase             0
target            0
dtype: int64
```

In [ ]:

```
print("Number of Null values in the test-metadata:\n", df_metadata_test.isnull().sum())
print("\nNumber of NA values in the test-metadata:\n",df_metadata_test.isna().sum())
```

```
Number of Null values in the test-metadata:
 signal_id         0
id_measurement    0
phase             0
dtype: int64

Number of NA values in the test-metadata:
 signal_id         0
id_measurement    0
phase             0
dtype: int64
```

check target

target: 0 if the power line is undamaged.
target: 1 if there is a fault.

In [ ]:

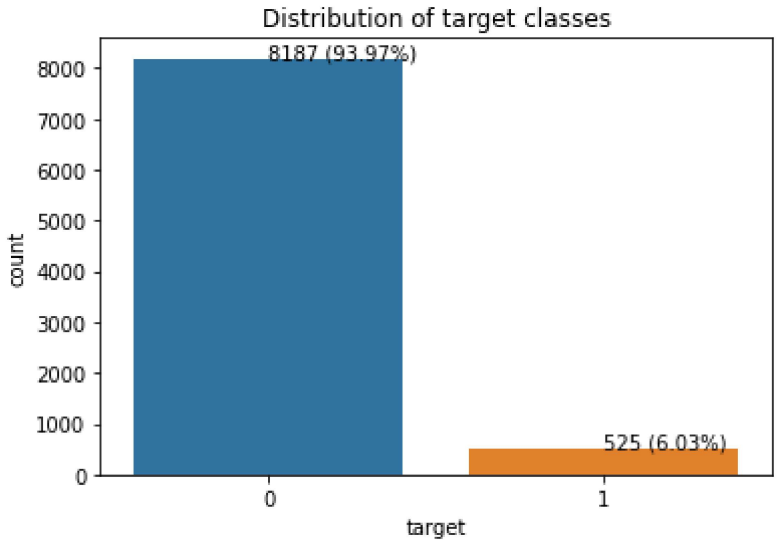
```
print("Number of unique values of target:\n{}".format(df_metadata_train['target'].value_cou
```

```
Number of unique values of target:
0    8187
1     525
Name: target, dtype: int64
```

Below plot for distribution of the target classes

In [ ]:

```
subplot = sns.countplot(x='target', data=df_metadata_train)

# https://github.com/mwaskom/seaborn/issues/1582
for i,j in enumerate(subplot.patches):
    percent = np.round((df_metadata_train[df_metadata_train['target']==i].shape[0]/df_metac
    subplot.annotate(str(df_metadata_train[df_metadata_train['target']==i].shape[0]) + f" (
                (j.get_x()+j.get_width()/2, j.get_height()))
plt.title("Distribution of target classes")
plt.show()
```



Observations:

The target classes are highly unbalanced.

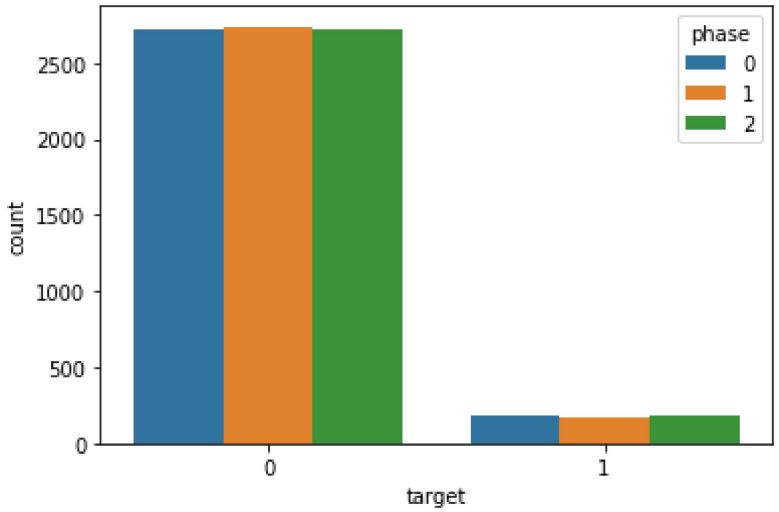
> 93.97% (8187) of the total data (8712) is of class 0.
> 6.03% (525) of the total data (8712) is of class 1.

In [ ]:

```
sns.countplot(x = 'target',hue = 'phase',data =df_metadata_train)
```

Out[24]:

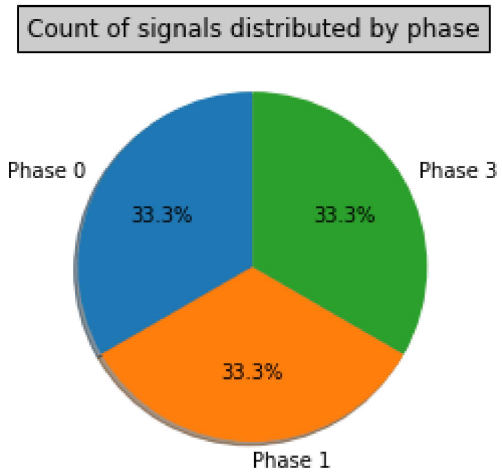
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f5db969f350>
```



Even if we look at the each of the phases, target values are still imbalanced in each phase

In [ ]:

```
# https://www.w3resource.com/graphics/matplotlib/piechart/matplotlib-piechart-exercise-2.ph
data = df_metadata_train['phase'].value_counts()
labels = ['Phase 0', 'Phase 1', 'Phase 3']
#colors = ["#1f77b4", "#ff7f0e", "#2ca02c"]
title = 'Count of signals distributed by phase'
plt.pie(data, labels=labels, shadow=True, startangle=90,autopct='%.1f%%')
plt.title(title, bbox={'facecolor':'0.8', 'pad':5})
plt.show()
```



check id_measurement

In [ ]:

```
print("Signal contains the following phases: {}".format(list(set(df_metadata_train['phase']
print("Count of each phase is as follows:\n{}".format(df_metadata_train['phase'].value_cour
```

```
Signal contains the following phases: [0, 1, 2]
Count of each phase is as follows:
0    2904
1    2904
2    2904
Name: phase, dtype: int64
```

Let's check if id_measurement value is same then is target label same?

In [ ]:

```
#iterating through each signal
for i in range(0,len(df_metadata_train),3):
    check1 = df_metadata_train.loc[i]['target']
    #flag var to check if same target or not
    flag = 0
    #check for other 2 phases
    for j in range(1,3):
        check2 = df_metadata_train.loc[i+j]['target']
        #if different target value
        if check1!=check2:
            print(df_metadata_train.loc[i:i+2])
            flag = 1
    if flag == 1:
        break
```

```
     signal_id  id_measurement  phase  target
201        201              67      0       1
202        202              67      1       1
203        203              67      2       0
```

From above it can be said that same id_measurement does not mean same target value

Importing parquet_data

As the data is large first lets look at the first 1000 signals

In [ ]:

```
df_signal_train =pq.read_pandas('/content/train.parquet',columns=[str(i) for i in range(100
```

In [ ]:

```
df_signal_train.head()
```

Out[10]:

	0	1	2	3	4	5	6	7	8	9	...	990	991	992	993	994	995	996	997	998
0	18	1	-19	-16	-5	19	-15	15	-1	-16	...	-18	10	9	18	-20	1	18	-19	-6
1	18	0	-19	-17	-6	19	-17	16	0	-15	...	-20	8	8	20	-19	2	18	-18	-6
2	17	-1	-20	-17	-6	19	-17	15	-3	-15	...	-20	6	6	17	-22	0	18	-18	-6
3	18	1	-19	-16	-5	20	-16	16	0	-15	...	-21	5	6	18	-19	1	18	-18	-6
4	18	0	-19	-16	-5	20	-17	16	-2	-14	...	-20	5	6	19	-21	1	19	-18	-5

5 rows × 1000 columns

In [ ]:

```
df_signal_train.shape
```

Out[11]:

```
(800000, 1000)
```

It can be observed that each signal consists of 800,000 points and each column represent a signal in meta_data.

Each signal contains 800,000 measurements of a power line's voltage, taken over 20 milliseconds. As the underlying electric grid operates at 50 Hz, this means each signal covers a single complete grid cycle. The grid itself operates on a 3-phase power scheme, and all three phases are measured simultaneously.

Transposing the dataframe as each column represents one data point.

In [ ]:

```
df_signal_train=df_signal_train.T
```

In [ ]:

```
df_signal_train.head()
```

Out[21]:

	0	1	2	3	4	5	6	7	8	9	...	799990	799991	799992	799993	799
0	18	18	17	18	18	18	19	18	18	17	...	18	18	17	17	
1	1	0	-1	1	0	0	1	0	0	0	...	1	0	0	0	
2	-19	-19	-20	-19	-19	-20	-18	-19	-20	-19	...	-19	-20	-21	-18	
3	-16	-17	-17	-16	-16	-15	-16	-17	-18	-17	...	-15	-15	-15	-15	
4	-5	-6	-6	-5	-5	-4	-5	-7	-7	-7	...	-4	-4	-4	-5	

5 rows × 800000 columns

Adding signal id to the main data frame

In [ ]:

```
df_signal_train['signal_id'] = list(df_metadata_train[:1000]['signal_id'])
```

In [ ]:

```
df_signal_train.head()
```

Out[24]:

	0	1	2	3	4	5	6	7	8	9	...	799991	799992	799993	799994	79999
0	18	18	17	18	18	18	19	18	18	17	...	18	17	17	18	19
1	1	0	-1	1	0	0	1	0	0	0	...	0	0	0	0	2
2	-19	-19	-20	-19	-19	-20	-18	-19	-20	-19	...	-20	-21	-18	-19	-18
3	-16	-17	-17	-16	-16	-15	-16	-17	-18	-17	...	-15	-15	-15	-15	-15
4	-5	-6	-6	-5	-5	-4	-5	-7	-7	-7	...	-4	-4	-5	-4	-4

5 rows × 800001 columns

Merging Metadata and Signal Data based on signal_id

In [ ]:

```
df_signal_train = df_signal_train.merge(df_metadata_train[:1000], on='signal_id')
```

In [ ]:

```
df_signal_train.head()
```

Out[26]:

	0	1	2	3	4	5	6	7	8	9	...	799994	799995	799996	799997	799998
0	18	18	17	18	18	18	19	18	18	17	...	18	19	19	17	19
1	1	0	-1	1	0	0	1	0	0	0	...	0	2	1	0	1
2	-19	-19	-20	-19	-19	-20	-18	-19	-20	-19	...	-19	-18	-19	-19	-18
3	-16	-17	-17	-16	-16	-15	-16	-17	-18	-17	...	-15	-15	-15	-15	-14
4	-5	-6	-6	-5	-5	-4	-5	-7	-7	-7	...	-4	-4	-4	-4	-3

5 rows × 800004 columns

Checking for null values in the dataframe

In [ ]:

```
df_signal_train.isnull().sum().sum()
```

Out[27]:

```
0
```

Observation:

There is no null values

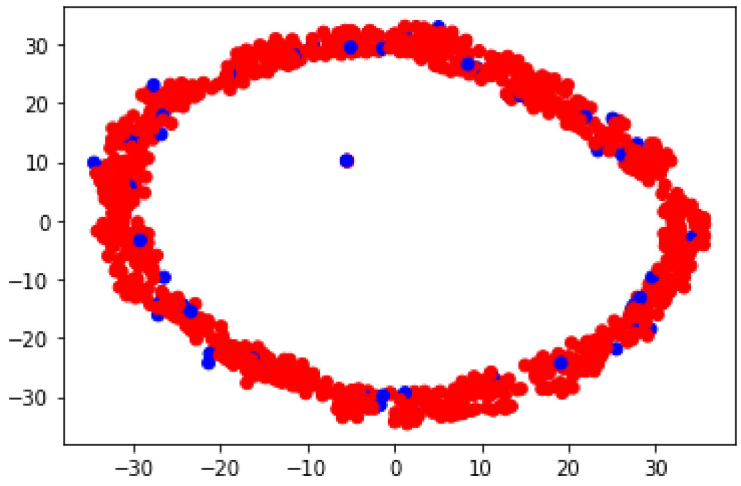
Plotting 2d plots using t-SNE using different values of perplexity and learning rate

Plotting the t-SNE plots only for 1000 points due to computational limitations

1. Using perplexity: 30 and learning rate: 200

In [ ]:

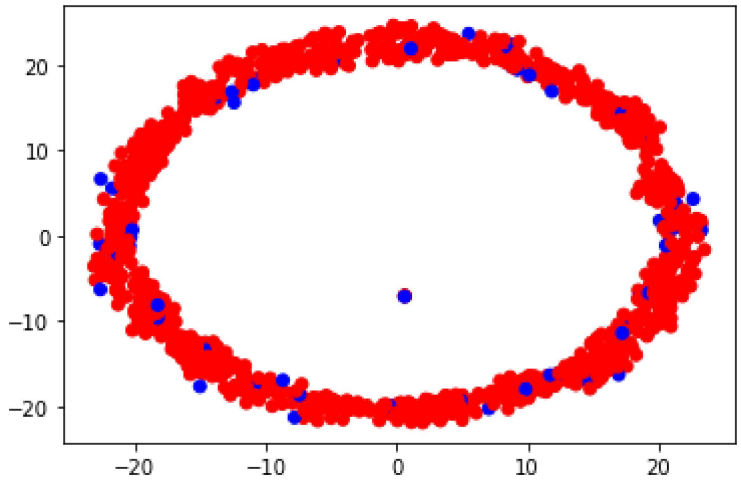
```
from sklearn.manifold import TSNE

tsne = TSNE(n_components=2, perplexity=30, learning_rate=200, random_state=42)

X_embedding = tsne.fit_transform(df_signal_train)
y = np.array(df_signal_train['target'])

tsne = np.hstack((X_embedding, y.reshape(-1,1)))
tsne_to_df = pd.DataFrame(data=tsne, columns=['Dimension_x','Dimension_y','Score'])
colors = {0:'red', 1:'blue', 2:'green'}
plt.scatter(tsne_to_df['Dimension_x'], tsne_to_df['Dimension_y'], c=tsne_to_df['Score'].app
plt.show()

del(tsne)
```



2. Using perplexity: 50 and learning rate:200

In [ ]:

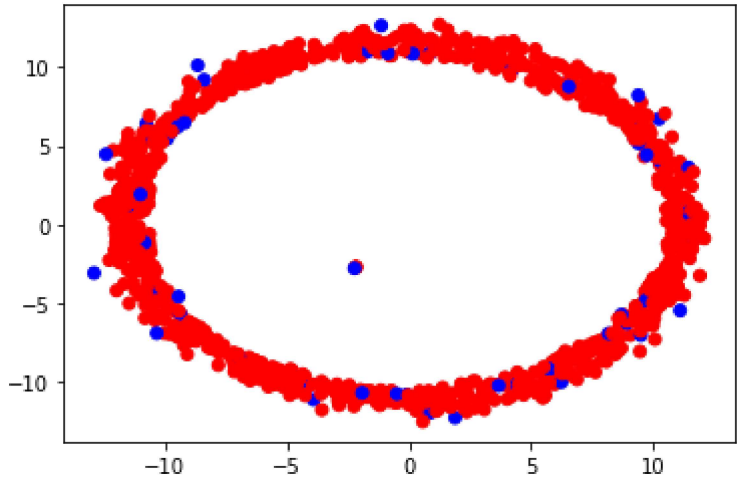
```
from sklearn.manifold import TSNE

tsne = TSNE(n_components=2, perplexity=50, learning_rate=200, random_state=42)

X_embedding = tsne.fit_transform(df_signal_train)
y = np.array(df_signal_train['target'])

tsne = np.hstack((X_embedding, y.reshape(-1,1)))
tsne_to_df = pd.DataFrame(data=tsne, columns=['Dimension_x','Dimension_y','Score'])
colors = {0:'red', 1:'blue', 2:'green'}
plt.scatter(tsne_to_df['Dimension_x'], tsne_to_df['Dimension_y'], c=tsne_to_df['Score'].app
plt.show()

del(tsne)
```



3. Using Perplexity:100 and learning rate: 150

In [ ]:

```
from sklearn.manifold import TSNE

tsne = TSNE(n_components=2, perplexity=100, learning_rate=150, random_state=42)

X_embedding = tsne.fit_transform(df_signal_train)
y = np.array(df_signal_train['target'])

tsne = np.hstack((X_embedding, y.reshape(-1,1)))
tsne_to_df = pd.DataFrame(data=tsne, columns=['Dimension_x','Dimension_y','Score'])
colors = {0:'red', 1:'blue', 2:'green'}
plt.scatter(tsne_to_df['Dimension_x'], tsne_to_df['Dimension_y'], c=tsne_to_df['Score'].app
plt.show()

del(tsne)
```



Observation:

The points are not well seperated in 2-dimensions as observed by these t-sne plots

Plotting signals

Plotting normal Signal

In [ ]:

```
#signal with target 0 (normal signal)
df_signal_train.loc[1]['target']
```

Out[35]:

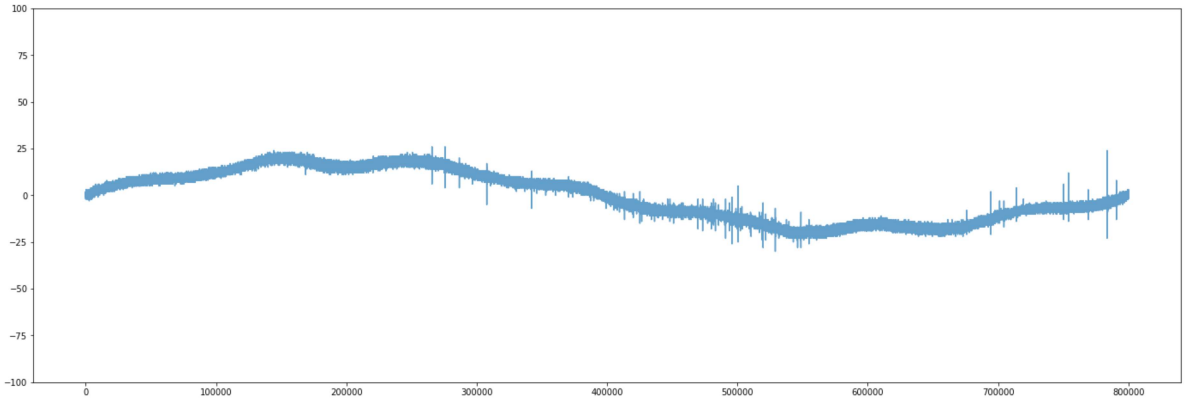
```
0
```

In [ ]:

```
plt.figure(figsize=(24, 8))
plt.plot((df_signal_train.loc[1].values), alpha=0.7);
plt.ylim([-100, 100])
```

Out[37]:

```
(-100.0, 100.0)
```



Plotting Faulty signal

In [ ]:

```
#signal with target 1 (faulty signal)
df_signal_train.loc[3]['target']
```

Out[38]:

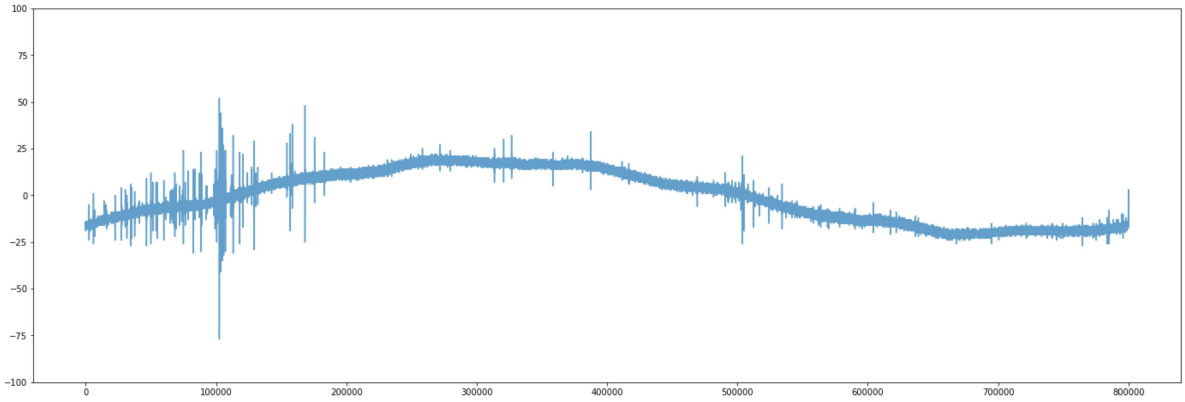
```
1
```

In [ ]:

```
plt.figure(figsize=(24, 8))
plt.plot((df_signal_train.loc[3].values), alpha=0.7);
plt.ylim([-100, 100])
```

Out[39]:

```
(-100.0, 100.0)
```



Observation:

Faulty signal has more noise

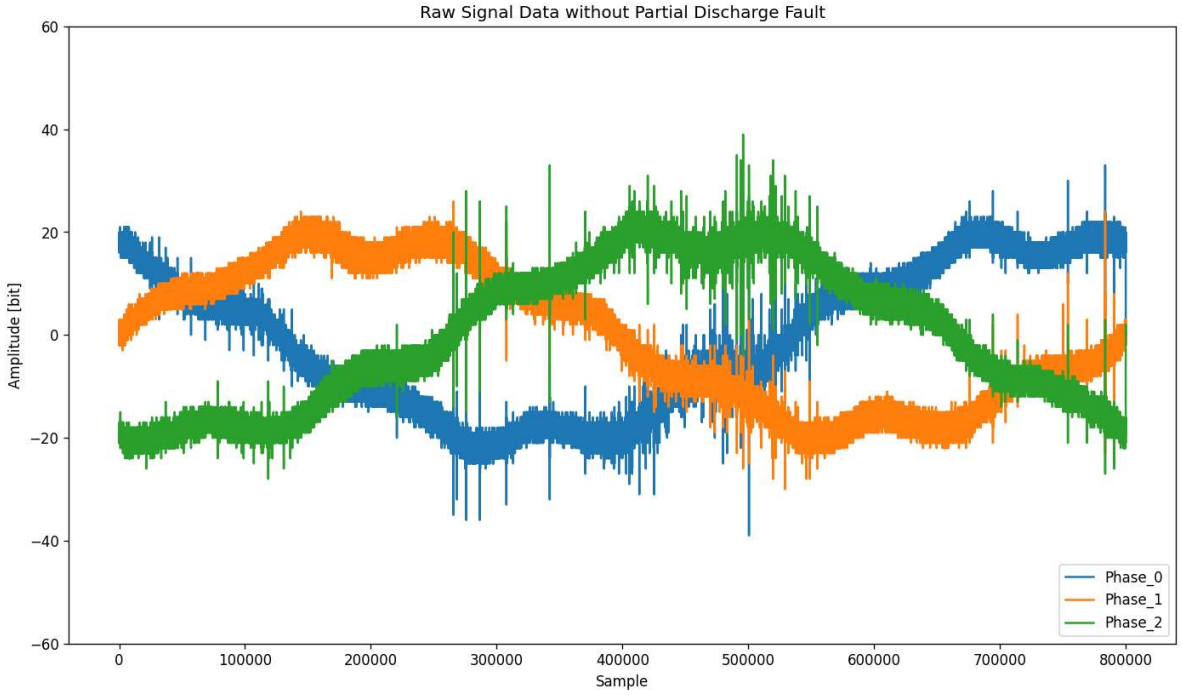
let's look at all 3 phases combined

In [ ]:

```
fig=plt.figure(figsize=(14, 8), dpi= 120, facecolor='w', edgecolor='k')
plot_labels = ['Phase_0', 'Phase_1', 'Phase_2']
plt.plot((df_signal_train.loc[0].values), label=plot_labels[0])
plt.plot((df_signal_train.loc[1].values), label=plot_labels[1])
plt.plot((df_signal_train.loc[2].values), label=plot_labels[2])
plt.ylim((-60, 60))
plt.legend(loc='lower right')
plt.title('Raw Signal Data without Partial Discharge Fault')
plt.xlabel('Sample')
plt.ylabel('Amplitude [bit]')
```

Out[47]:

```
Text(0, 0.5, 'Amplitude [bit]')
```

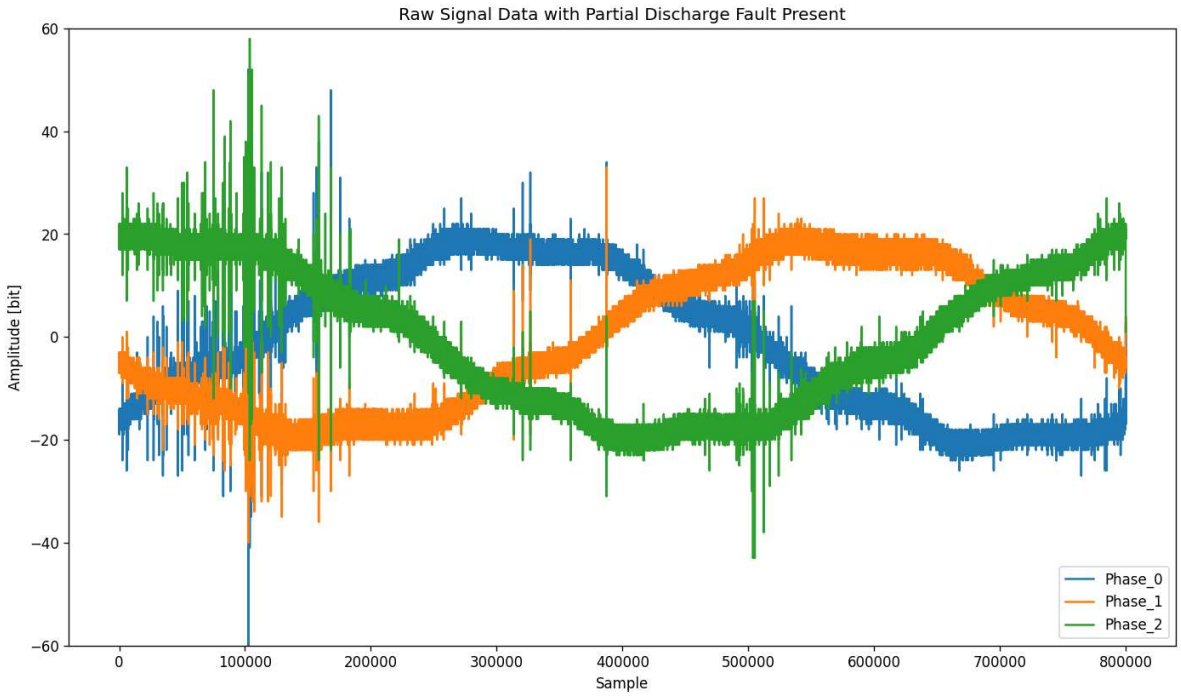


In [ ]:

```
fig=plt.figure(figsize=(14, 8), dpi= 120, facecolor='w', edgecolor='k')
plot_labels = ['Phase_0', 'Phase_1', 'Phase_2']
plt.plot((df_signal_train.loc[3].values), label=plot_labels[0])
plt.plot((df_signal_train.loc[4].values), label=plot_labels[1])
plt.plot((df_signal_train.loc[5].values), label=plot_labels[2])
plt.ylim((-60, 60))
plt.legend(loc='lower right')
plt.title('Raw Signal Data with Partial Discharge Fault Present')
plt.xlabel('Sample')
plt.ylabel('Amplitude [bit]')
```

Out[46]:

```
Text(0, 0.5, 'Amplitude [bit]')
```



Observation:

Faulty signal have more noise than the normal signal. Hence, noise can be a very useful feature for fault detection

Flatiron

reference: https://www.kaggle.com/miklgr500/flatiron (https://www.kaggle.com/miklgr500/flatiron)

The idea of flatiron is similar to High Pass Filter. It allows high frequency to pass. It can be useful for noise extraction

In [ ]:

```
def filtering(x, alpha=50, beta=1):
    x_new = np.zeros_like(x)
    zero = x[0]
    for i in range(1, len(x)):
        zero = zero*(alpha-beta)/alpha + beta*x[i]/alpha
        x_new[i] =  x[i] - zero
    return x_new
```

In [ ]:

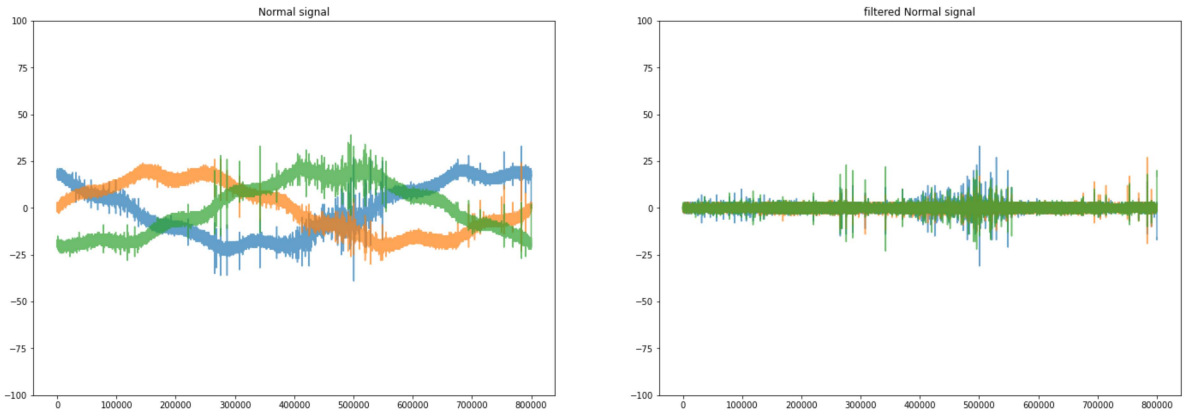
```
#Flattening a Normal signal
normal_signal_filter =  [None] * 3
normal_signal_filter[0] = filtering(df_signal_train.loc[0].values)
normal_signal_filter[1] = filtering(df_signal_train.loc[1].values)
normal_signal_filter[2] = filtering(df_signal_train.loc[2].values)
```

In [ ]:

```
#Code to plot normal signal with flattened faulty signal
f, ax = plt.subplots(1, 2, figsize=(24, 8))

ax[0].plot((df_signal_train.loc[0].values), alpha=0.7);
ax[0].plot((df_signal_train.loc[1].values), alpha=0.7);
ax[0].plot((df_signal_train.loc[2].values), alpha=0.7);
ax[0].set_title('Normal signal')
ax[0].set_ylim([-100, 100])

ax[1].plot((normal_signal_filter)[0], alpha=0.7);
ax[1].plot((normal_signal_filter)[1], alpha=0.7);
ax[1].plot((normal_signal_filter)[2], alpha=0.7);
ax[1].set_title('filtered Normal signal')
ax[1].set_ylim([-100, 100])
```

Out[44]:

```
(-100.0, 100.0)
```



Plotting faulty signal with flattened faulty signal

In [ ]:

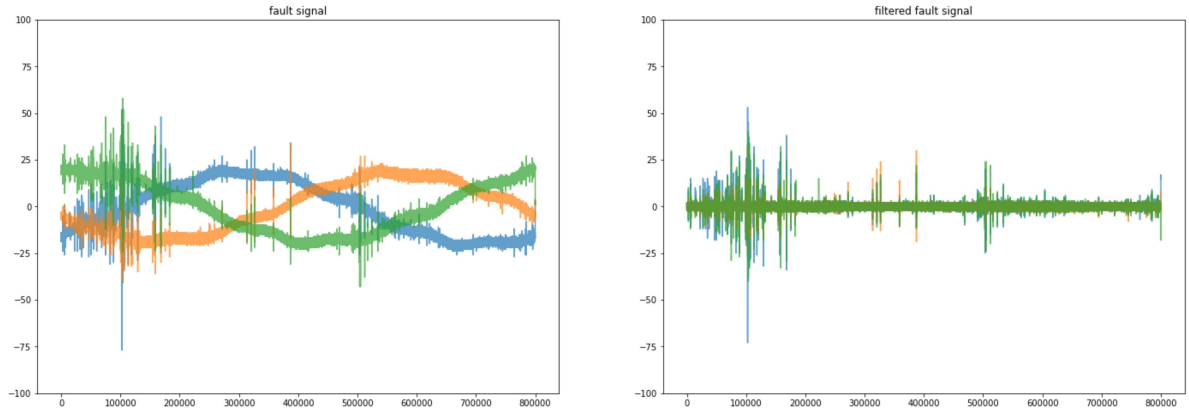
```
#Flattening a faulty signal
faulty_signal_filter =  [None] * 3
faulty_signal_filter[0] = filtering(df_signal_train.loc[3].values)
faulty_signal_filter[1] = filtering(df_signal_train.loc[4].values)
faulty_signal_filter[2] = filtering(df_signal_train.loc[5].values)
```

In [ ]:

```
#Code to plot faulty signal with flattened faulty signal
f, ax = plt.subplots(1, 2, figsize=(24, 8))

ax[0].plot((df_signal_train.loc[3].values), alpha=0.7);
ax[0].plot((df_signal_train.loc[4].values), alpha=0.7);
ax[0].plot((df_signal_train.loc[5].values), alpha=0.7);
ax[0].set_title('fault signal')
ax[0].set_ylim([-100, 100])

ax[1].plot((faulty_signal_filter)[0], alpha=0.7);
ax[1].plot((faulty_signal_filter)[1], alpha=0.7);
ax[1].plot((faulty_signal_filter)[2], alpha=0.7);
ax[1].set_title('filtered fault signal')
ax[1].set_ylim([-100, 100])
```

Out[51]:

```
(-100.0, 100.0)
```



Observation:

Faulty signal has more noise than normal signal

In [ ]:

4. Featurization

Extracting statistical features from our dataset

In [ ]:

```
stat_feature_df = pd.DataFrame(columns=['signal_id','mean','std','min','max','var','skew','
```

In [ ]:

```
import scipy
from statsmodels.robust import mad
import collections
```

In [ ]:

```
for index,row in tqdm(df_metadata_train.iterrows()):

    stat = []
    signal =  pq.read_pandas('/content/train.parquet',columns=[str(row['signal_id'])]).to_p

    #signal_id
    stat.append(row['signal_id'])

    #mean
    stat.append(signal.mean())

    #std
    stat.append(signal.std())

    #min
    stat.append(signal.min())

    #max
    stat.append(signal.max())
    # var
    var = np.nanvar(signal)
    stat.append(var)

    #skewness
    skew = scipy.stats.skew(signal)
    stat.append(skew)
    #kurtosis
    kurt = scipy.stats.kurtosis(signal)
    stat.append(kurt)
    #percentiles of signal
    q0 = np.quantile(signal, 0.10)
    stat.append(q0)
    q1 = np.quantile(signal, 0.25)
    stat.append(q1)
    q2 = np.quantile(signal, 0.75)
    stat.append(q2)
    q3 = np.quantile(signal, 0.90)
    stat.append(q3)
    #bandwidth
    band_width = [1/signal.max(),1/signal.min()]
    stat.append(band_width)
    #entropy
    counter_values = collections.Counter(signal).most_common()
    probabilities = [elem[1]/len(signal) for elem in counter_values]
    entropy=scipy.stats.entropy(probabilities)
    stat.append(entropy)
    #crossings
    zero_crossing_indices = np.nonzero(np.diff(np.array(signal) > 0))[0]
    no_zero_crossings = len(zero_crossing_indices)
    mean_crossing_indices = np.nonzero(np.diff(np.array(signal) > np.nanmean(signal)))[0]
    no_mean_crossings = len(mean_crossing_indices)
    stat.append(no_zero_crossings)
    stat.append(no_mean_crossings)

    #target
    stat.append(row['target'])
```

```
    stat_feature_df.loc[len(stat_feature_df)] = stat
```

```
8712it [49:09,  2.95it/s]
```

In [ ]:

```
for index,row in tqdm(df_metadata_train.iterrows()):

    stat = []
    signal =  pq.read_pandas('/content/train.parquet',columns=[str(row['signal_id'])]).to_p

    #signal_id
    stat.append(row['signal_id'])

    #mean
    stat.append(signal.mean())

    #std
    stat.append(signal.std())

    #min
    stat.append(signal.min())

    #max
    stat.append(signal.max())
    # var
    var = np.nanvar(signal)
    stat.append(var)

    #skewness
    skew = scipy.stats.skew(signal)
    stat.append(skew)
    #kurtosis
    kurt = scipy.stats.kurtosis(signal)
    stat.append(kurt)
    #percentiles of signal
    q0 = np.quantile(signal, 0.10)
    stat.append(q0)
    q1 = np.quantile(signal, 0.25)
    stat.append(q1)
    q2 = np.quantile(signal, 0.75)
    stat.append(q2)
    q3 = np.quantile(signal, 0.90)
    stat.append(q3)
    #bandwidth
    band_width = [1/signal.max(),1/signal.min()]
    stat.append(band_width)
    #entropy
    counter_values = collections.Counter(signal).most_common()
    probabilities = [elem[1]/len(signal) for elem in counter_values]
    entropy=scipy.stats.entropy(probabilities)
    stat.append(entropy)
    #crossings
    zero_crossing_indices = np.nonzero(np.diff(np.array(signal) > 0))[0]
    no_zero_crossings = len(zero_crossing_indices)
    mean_crossing_indices = np.nonzero(np.diff(np.array(signal) > np.nanmean(signal)))[0]
    no_mean_crossings = len(mean_crossing_indices)
    stat.append(no_zero_crossings)
    stat.append(no_mean_crossings)

    #target
    stat.append(row['target'])
```

```
    stat_feature_df.loc[len(stat_feature_df)] = stat
```

```
8712it [41:15,  3.52it/s]
```

In [ ]:

```
stat_feature_df.to_csv('stat_feature_df.csv',index=False)
```

In [ ]:

```
stat_feature_df = pd.read_csv('stat_feature_df.csv',index_col=False)
```

In [ ]:

```
stat_feature_df.head()
```

Out[29]:

	signal_id	mean	std	min	max	var	skew	kurt	q0	q1	
0	0	-0.960271	13.870724	-39	33	192.396993	0.003591	-1.433378	-20.0	-13.0	1
1	1	-0.194125	13.037134	-30	26	169.966873	0.006952	-1.422562	-18.0	-12.0	1
2	2	-0.043555	13.684282	-28	39	187.259580	0.005427	-1.426438	-18.0	-13.0	1
3	3	-0.997401	13.673630	-77	52	186.968157	-0.001616	-1.474346	-19.0	-14.0	1
4	4	-0.175586	12.938372	-40	33	167.401478	-0.000686	-1.478338	-17.0	-13.0	1

In [ ]:

```
3swS
```

In [ ]:

```
stat_feature_df.shape
```

Out[46]:

```
(8712, 17)
```

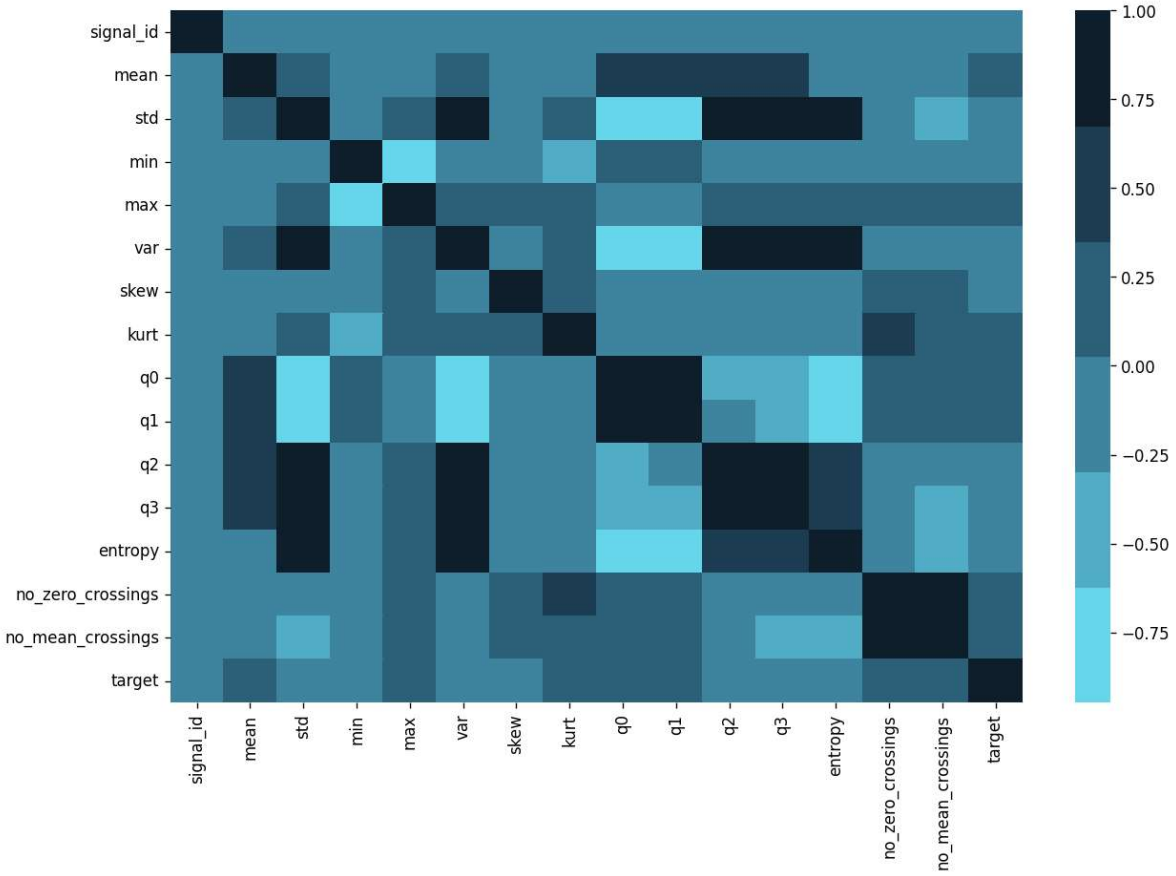
other featurerization

In [ ]:

```
from matplotlib import pyplot as plt
import seaborn as sns
blues = ["#66D7EB", "#51ACC5", "#3E849E", "#2C5F78", "#1C3D52", "#0E1E2B"]
cor = stat_feature_df.corr()
f, ax = plt.subplots(figsize=(12, 8), dpi= 120, facecolor='w', edgecolor='k')
sns.heatmap(cor, cmap=blues)
```

Out[47]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f3a01f20f50>
```



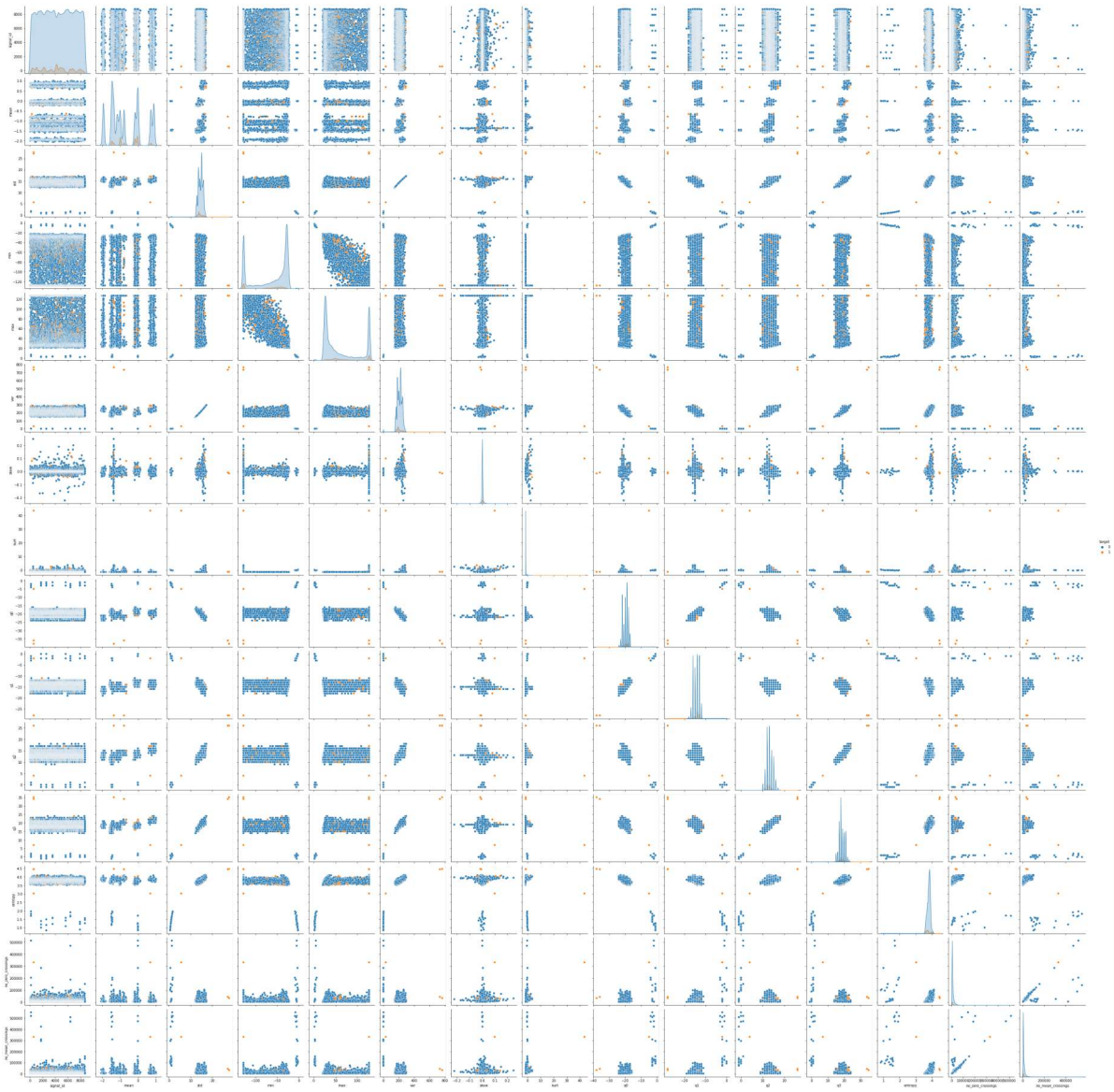
Type *Markdown* and LaTeX: α^2

In [ ]:

```
sns.pairplot(stat_feature_df, hue="target",height=3, diag_kind="kde", diag_kws=dict(shade=T
```

Out[30]:

```
<seaborn.axisgrid.PairGrid at 0x7ff730c4d0d0>
```



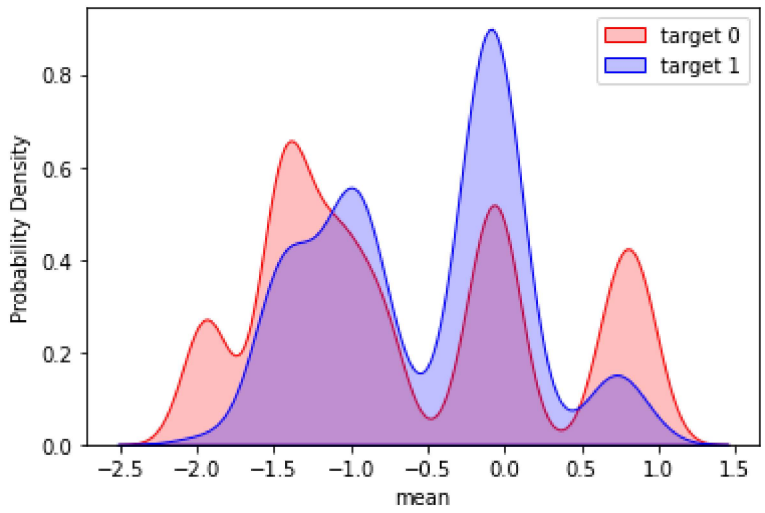
Mean

In [ ]:

```
sns.kdeplot(stat_feature_df.loc[(stat_feature_df['target']==0),
            'mean'], color='r', shade=True, Label='target 0')

sns.kdeplot(stat_feature_df.loc[(stat_feature_df['target']==1),
            'mean'], color='b', shade=True, Label='target 1')
plt.legend()
plt.xlabel('mean')
plt.ylabel('Probability Density')
```

Out[31]:

```
Text(0, 0.5, 'Probability Density')
```

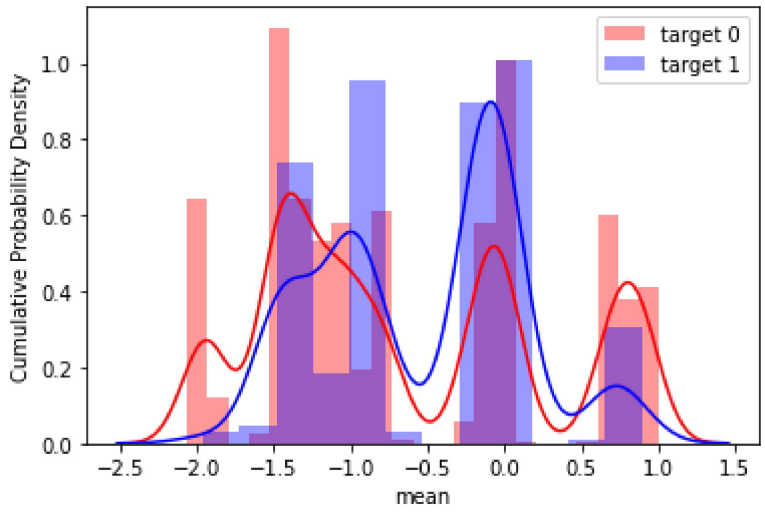


In [ ]:

```
sns.distplot(stat_feature_df.loc[(stat_feature_df['target']==0),
            'mean'], color='r', label='target 0')

sns.distplot(stat_feature_df.loc[(stat_feature_df['target']==1),
            'mean'], color='b', label='target 1')
plt.legend()
plt.xlabel('mean')
plt.ylabel('Cumulative Probability Density')
```

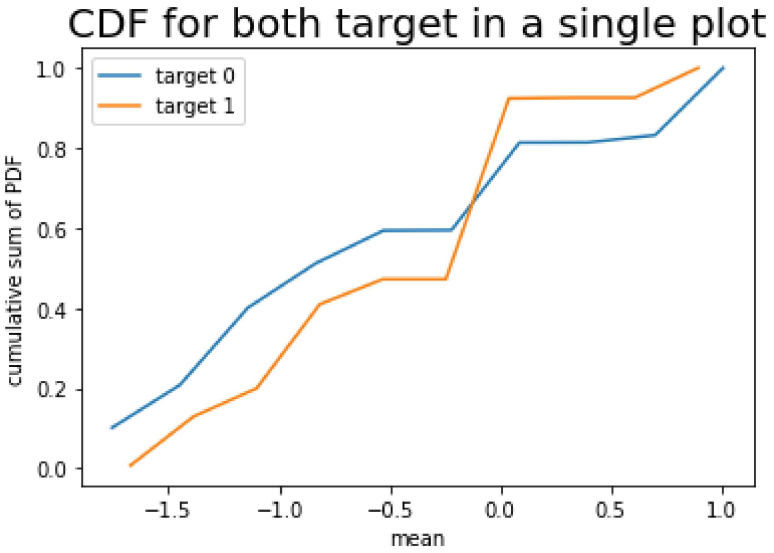
Out[33]:

```
Text(0, 0.5, 'Cumulative Probability Density')
```



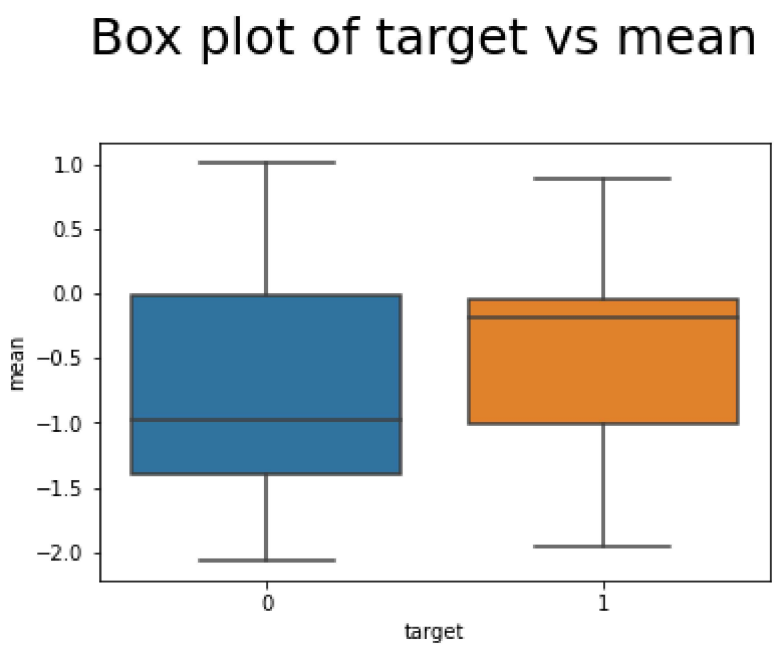
In [ ]:

```
count1, bins_count1 = np.histogram(stat_feature_df.loc[(stat_feature_df['target']==0),
            'mean'], bins=10)
pdf1 = count1 / sum(count1)
cdf1 = np.cumsum(pdf1)

count2, bins_count2 = np.histogram(stat_feature_df.loc[(stat_feature_df['target']==1),
            'mean'], bins=10)
pdf2 = count2 / sum(count2)
cdf2 = np.cumsum(pdf2)
plt.plot(bins_count1[1:], cdf1, label="target 0")
plt.plot(bins_count2[1:], cdf2, label="target 1")

plt.xlabel('mean')
plt.ylabel("cumulative sum of PDF")
plt.legend()
plt.title("CDF for both target in a single plot",fontsize=20)
plt.show()
```



In [ ]:

```
sns.boxplot(x=stat_feature_df['target'], y=stat_feature_df['mean'], data=stat_feature_df)

plt.suptitle("Box plot of target vs mean",fontsize=25,
          y=1.1)
plt.show()
```



If the mean of the signal is between less than -2 or more than 0.5 it is more likely that there is no partial discharge

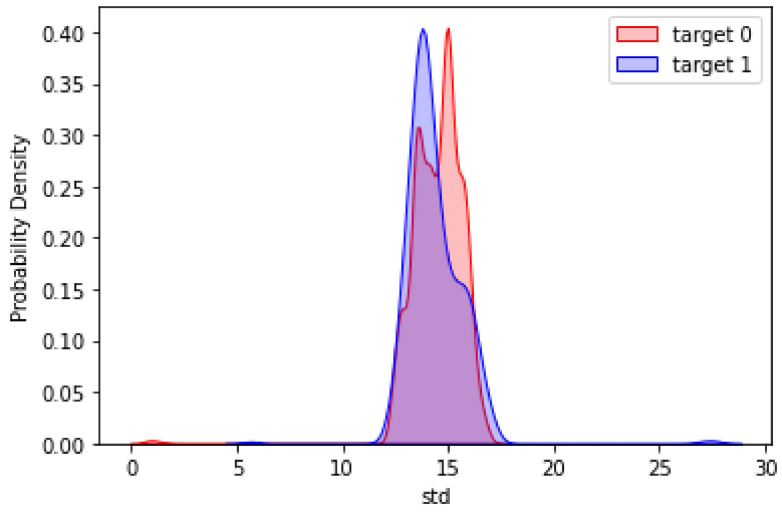
Standard Deviation

In [ ]:

```
sns.kdeplot(stat_feature_df.loc[(stat_feature_df['target']==0),
            'std'], color='r', shade=True, Label='target 0')

sns.kdeplot(stat_feature_df.loc[(stat_feature_df['target']==1),
            'std'], color='b', shade=True, Label='target 1')
plt.legend()
plt.xlabel('std')
plt.ylabel('Probability Density')
```

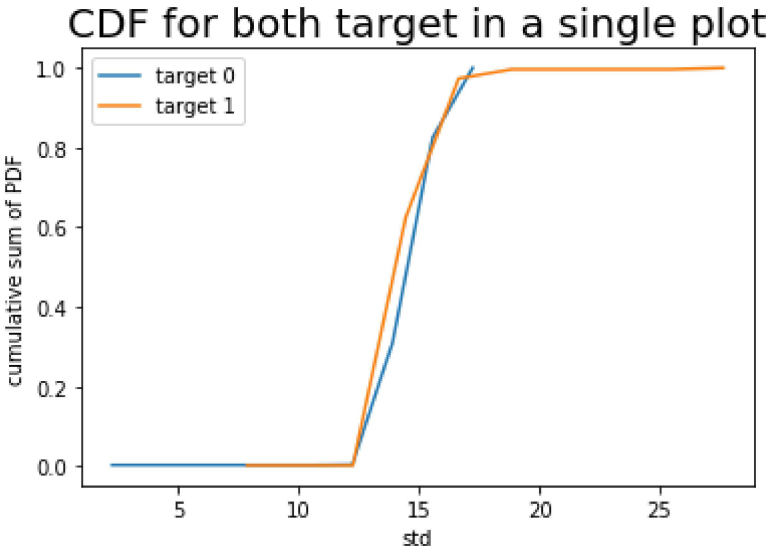
Out[32]:

```
Text(0, 0.5, 'Probability Density')
```



In [ ]:

```
count1, bins_count1 = np.histogram(stat_feature_df.loc[(stat_feature_df['target']==0),
            'std'], bins=10)
pdf1 = count1 / sum(count1)
cdf1 = np.cumsum(pdf1)

count2, bins_count2 = np.histogram(stat_feature_df.loc[(stat_feature_df['target']==1),
            'std'], bins=10)
pdf2 = count2 / sum(count2)
cdf2 = np.cumsum(pdf2)
plt.plot(bins_count1[1:], cdf1, label="target 0")
plt.plot(bins_count2[1:], cdf2, label="target 1")

plt.xlabel('std')
plt.ylabel("cumulative sum of PDF")
plt.legend()
plt.title("CDF for both target in a single plot",fontsize=20)
plt.show()
```



It is more probable that if std is greater than 15 then there is partial discharge

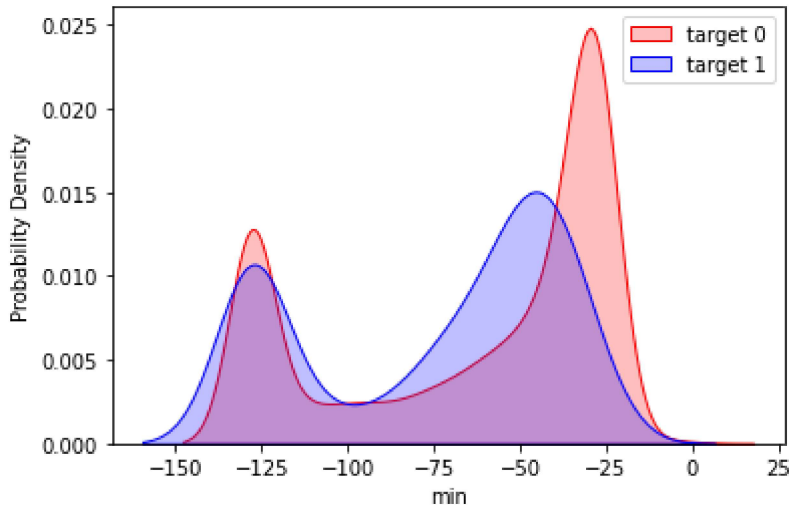
Minimum of signal

In [ ]:

```
sns.kdeplot(stat_feature_df.loc[(stat_feature_df['target']==0),
            'min'], color='r', shade=True, Label='target 0')

sns.kdeplot(stat_feature_df.loc[(stat_feature_df['target']==1),
            'min'], color='b', shade=True, Label='target 1')
plt.legend()
plt.xlabel('min')
plt.ylabel('Probability Density')
```

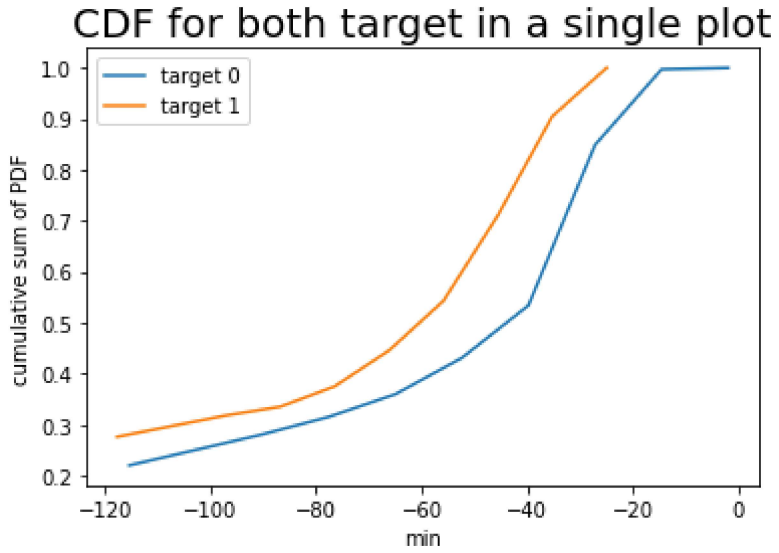
Out[33]:

```
Text(0, 0.5, 'Probability Density')
```



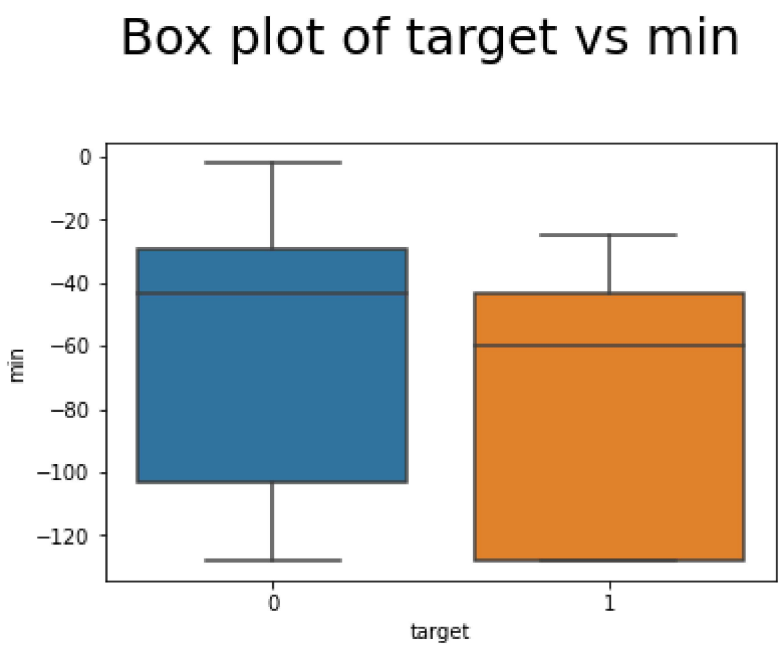
In [ ]:

```
count1, bins_count1 = np.histogram(stat_feature_df.loc[(stat_feature_df['target']==0),
            'min'], bins=10)
pdf1 = count1 / sum(count1)
cdf1 = np.cumsum(pdf1)

count2, bins_count2 = np.histogram(stat_feature_df.loc[(stat_feature_df['target']==1),
            'min'], bins=10)
pdf2 = count2 / sum(count2)
cdf2 = np.cumsum(pdf2)
plt.plot(bins_count1[1:], cdf1, label="target 0")
plt.plot(bins_count2[1:], cdf2, label="target 1")

plt.xlabel('min')
plt.ylabel("cumulative sum of PDF")
plt.legend()
plt.title("CDF for both target in a single plot",fontsize=20)
plt.show()
```



In [ ]:

```
sns.boxplot(x=stat_feature_df['target'], y=stat_feature_df['min'], data=stat_feature_df)

plt.suptitle("Box plot of target vs min",fontsize=25,
          y=1.1)
plt.show()
```



if the minimum of signal is greater than -30 there's more probability of no partial discharge

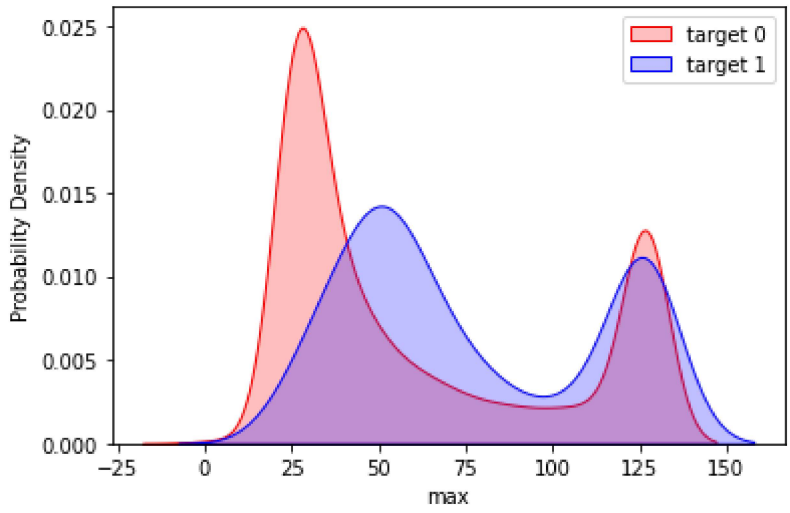
Maximum of signal

In [ ]:

```
sns.kdeplot(stat_feature_df.loc[(stat_feature_df['target']==0),
            'max'], color='r', shade=True, Label='target 0')

sns.kdeplot(stat_feature_df.loc[(stat_feature_df['target']==1),
            'max'], color='b', shade=True, Label='target 1')
plt.legend()
plt.xlabel('max')
plt.ylabel('Probability Density')
```

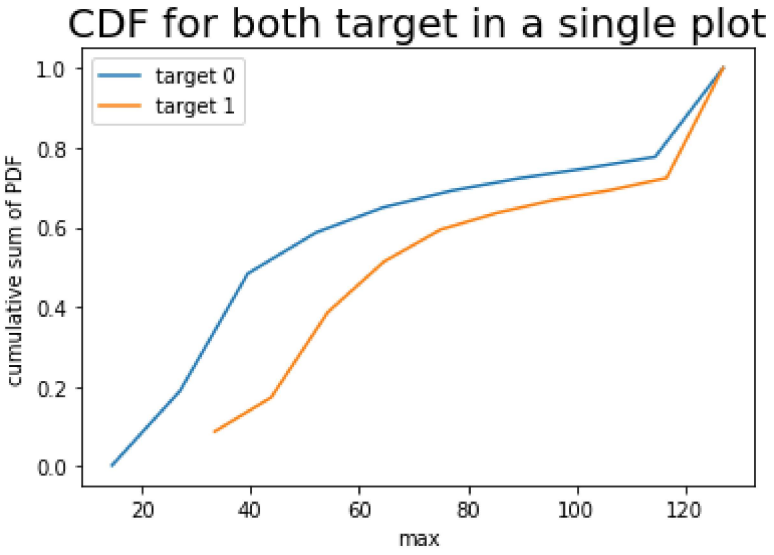
Out[34]:

```
Text(0, 0.5, 'Probability Density')
```



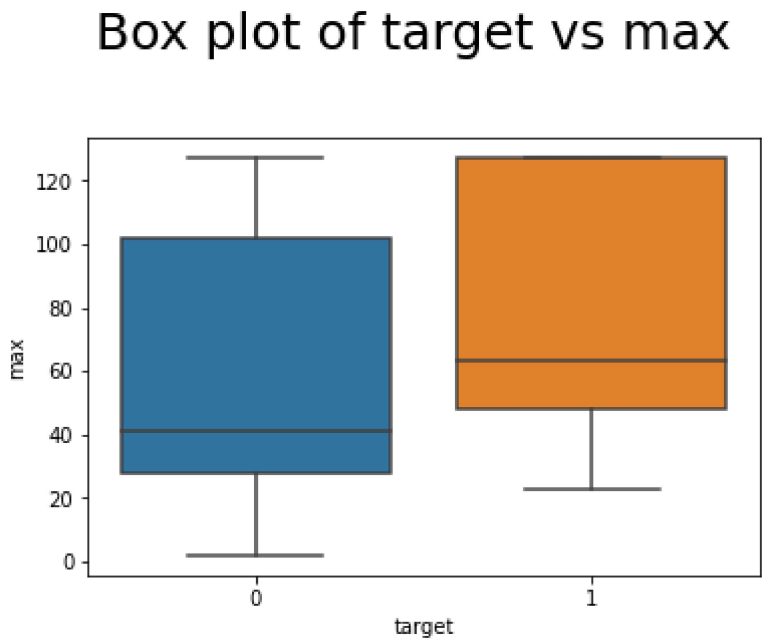
In [ ]:

```
count1, bins_count1 = np.histogram(stat_feature_df.loc[(stat_feature_df['target']==0),
            'max'], bins=10)
pdf1 = count1 / sum(count1)
cdf1 = np.cumsum(pdf1)

count2, bins_count2 = np.histogram(stat_feature_df.loc[(stat_feature_df['target']==1),
            'max'], bins=10)
pdf2 = count2 / sum(count2)
cdf2 = np.cumsum(pdf2)
plt.plot(bins_count1[1:], cdf1, label="target 0")
plt.plot(bins_count2[1:], cdf2, label="target 1")

plt.xlabel('max')
plt.ylabel("cumulative sum of PDF")
plt.legend()
plt.title("CDF for both target in a single plot",fontsize=20)
plt.show()
```



In [ ]:

```
sns.boxplot(x=stat_feature_df['target'], y=stat_feature_df['max'], data=stat_feature_df)

plt.suptitle("Box plot of target vs max",fontsize=25,
          y=1.1)
plt.show()
```



if the maximum of signal is less than 50 there's more probability of no partial discharge

In [ ]:

```

```

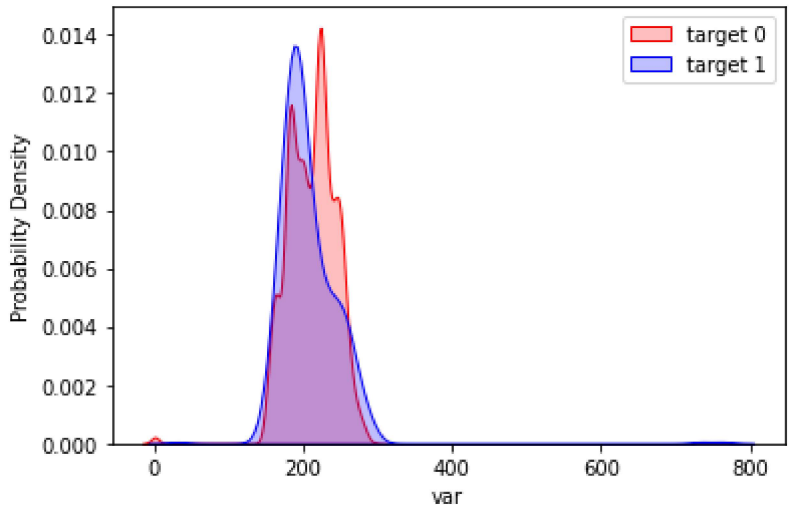
Variance

In [ ]:

```
sns.kdeplot(stat_feature_df.loc[(stat_feature_df['target']==0),
            'var'], color='r', shade=True, Label='target 0')

sns.kdeplot(stat_feature_df.loc[(stat_feature_df['target']==1),
            'var'], color='b', shade=True, Label='target 1')
plt.legend()
plt.xlabel('var')
plt.ylabel('Probability Density')
```

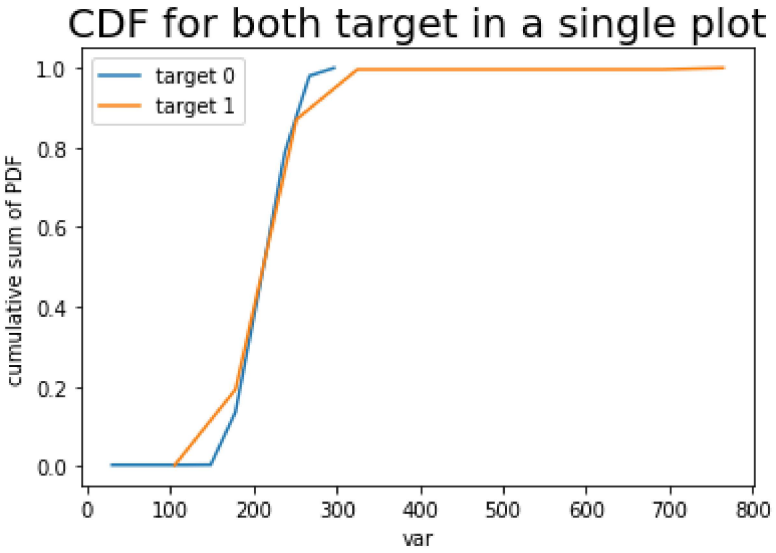
Out[35]:

```
Text(0, 0.5, 'Probability Density')
```



In [ ]:

```
count1, bins_count1 = np.histogram(stat_feature_df.loc[(stat_feature_df['target']==0),
            'var'], bins=10)
pdf1 = count1 / sum(count1)
cdf1 = np.cumsum(pdf1)

count2, bins_count2 = np.histogram(stat_feature_df.loc[(stat_feature_df['target']==1),
            'var'], bins=10)
pdf2 = count2 / sum(count2)
cdf2 = np.cumsum(pdf2)
plt.plot(bins_count1[1:], cdf1, label="target 0")
plt.plot(bins_count2[1:], cdf2, label="target 1")

plt.xlabel('var')
plt.ylabel("cumulative sum of PDF")
plt.legend()
plt.title("CDF for both target in a single plot",fontsize=20)
plt.show()
```



if the variance of signal is greater than 200 there's more probability of no partial discharge

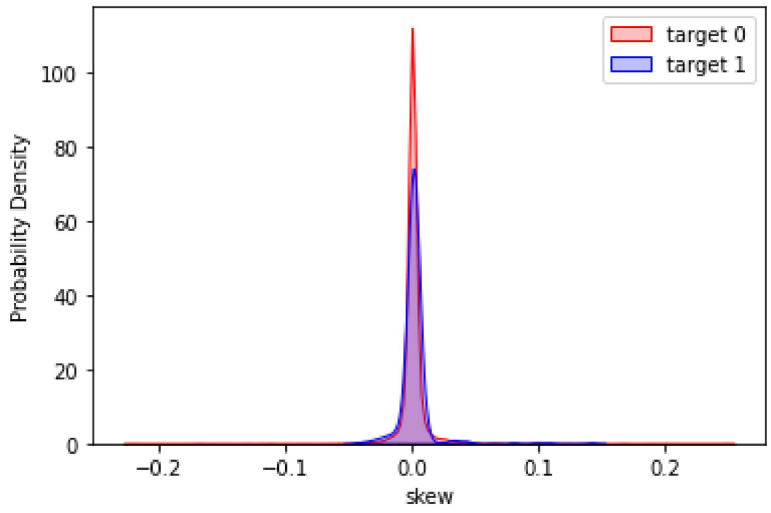
Skewness

In [ ]:

```
sns.kdeplot(stat_feature_df.loc[(stat_feature_df['target']==0),
            'skew'], color='r', shade=True, Label='target 0')

sns.kdeplot(stat_feature_df.loc[(stat_feature_df['target']==1),
            'skew'], color='b', shade=True, Label='target 1')
plt.legend()
plt.xlabel('skew')
plt.ylabel('Probability Density')
```

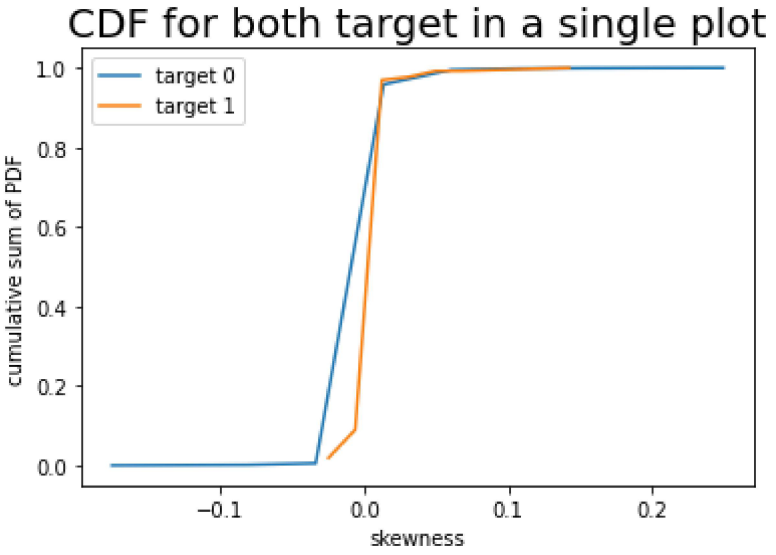
Out[36]:

```
Text(0, 0.5, 'Probability Density')
```



In [ ]:

```
count1, bins_count1 = np.histogram(stat_feature_df.loc[(stat_feature_df['target']==0),
            'skew'], bins=10)
pdf1 = count1 / sum(count1)
cdf1 = np.cumsum(pdf1)

count2, bins_count2 = np.histogram(stat_feature_df.loc[(stat_feature_df['target']==1),
            'skew'], bins=10)
pdf2 = count2 / sum(count2)
cdf2 = np.cumsum(pdf2)
plt.plot(bins_count1[1:], cdf1, label="target 0")
plt.plot(bins_count2[1:], cdf2, label="target 1")

plt.xlabel('skewness')
plt.ylabel("cumulative sum of PDF")
plt.legend()
plt.title("CDF for both target in a single plot",fontsize=20)
plt.show()
```



if the skewness of signal is greater than 0.1 and less than 0.0 there's more probability of no partial discharge

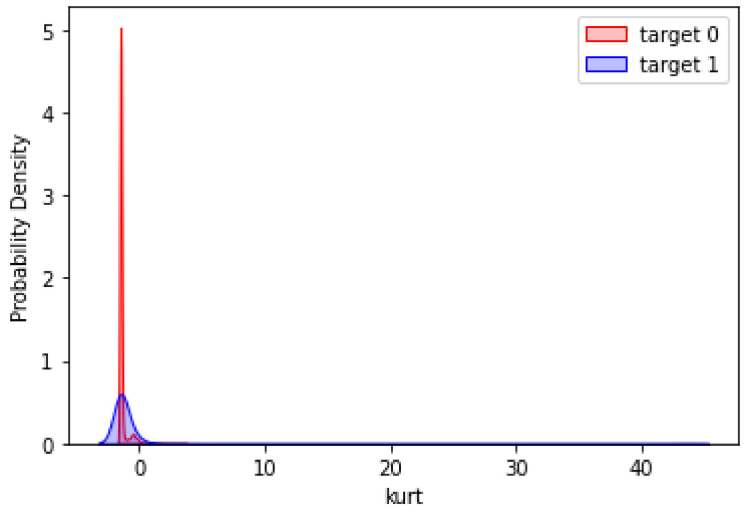
Kurtosis

In [ ]:

```
sns.kdeplot(stat_feature_df.loc[(stat_feature_df['target']==0),
            'kurt'], color='r', shade=True, Label='target 0')

sns.kdeplot(stat_feature_df.loc[(stat_feature_df['target']==1),
            'kurt'], color='b', shade=True, Label='target 1')
plt.legend()
plt.xlabel('kurt')
plt.ylabel('Probability Density')
```

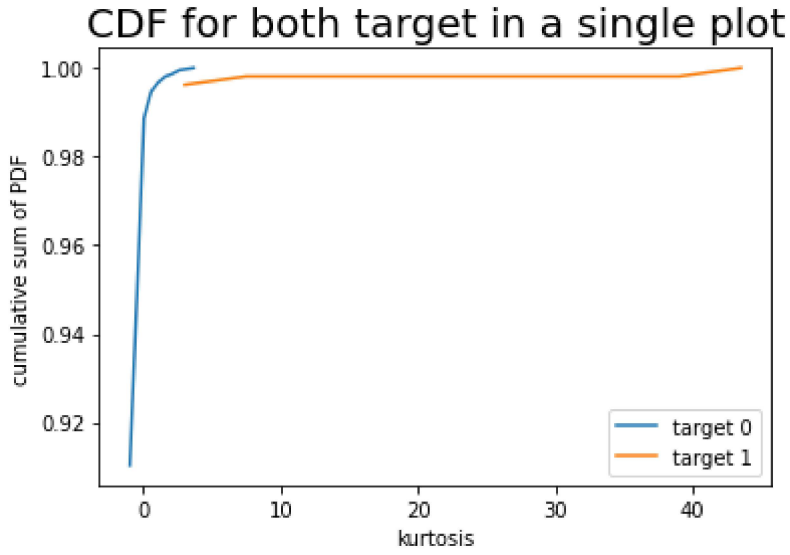
Out[37]:

```
Text(0, 0.5, 'Probability Density')
```



In [ ]:

```
count1, bins_count1 = np.histogram(stat_feature_df.loc[(stat_feature_df['target']==0),
            'kurt'], bins=10)
pdf1 = count1 / sum(count1)
cdf1 = np.cumsum(pdf1)

count2, bins_count2 = np.histogram(stat_feature_df.loc[(stat_feature_df['target']==1),
            'kurt'], bins=10)
pdf2 = count2 / sum(count2)
cdf2 = np.cumsum(pdf2)
plt.plot(bins_count1[1:], cdf1, label="target 0")
plt.plot(bins_count2[1:], cdf2, label="target 1")

plt.xlabel('kurtosis')
plt.ylabel("cumulative sum of PDF")
plt.legend()
plt.title("CDF for both target in a single plot",fontsize=20)
plt.show()
```



if the kurtosis of signal is greater than 5 than there's more probability of partial discharge

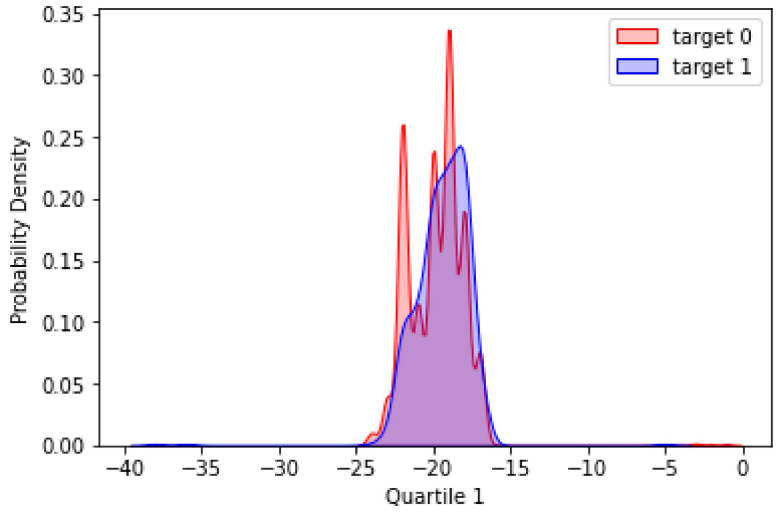
Quartile 1

In [ ]:

```
sns.kdeplot(stat_feature_df.loc[(stat_feature_df['target']==0),
            'q0'], color='r', shade=True, Label='target 0')

sns.kdeplot(stat_feature_df.loc[(stat_feature_df['target']==1),
            'q0'], color='b', shade=True, Label='target 1')
plt.legend()
plt.xlabel('Quartile 1')
plt.ylabel('Probability Density')
```

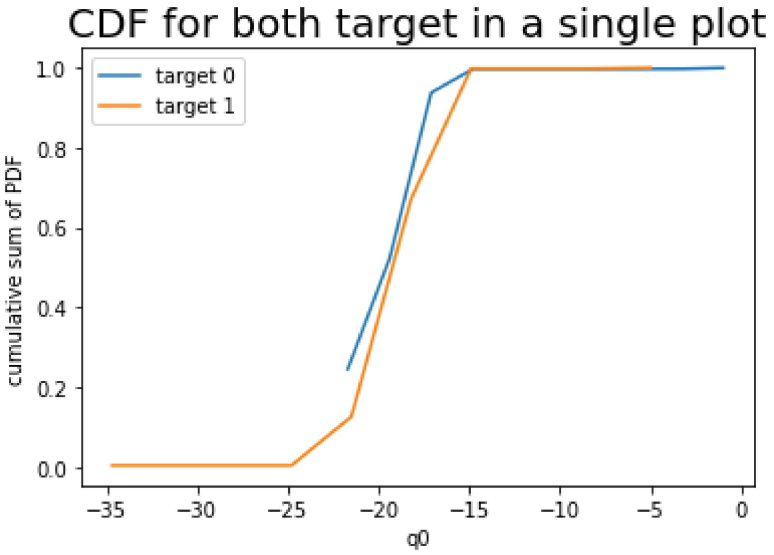
Out[38]:

```
Text(0, 0.5, 'Probability Density')
```



In [ ]:

```
count1, bins_count1 = np.histogram(stat_feature_df.loc[(stat_feature_df['target']==0),
            'q0'], bins=10)
pdf1 = count1 / sum(count1)
cdf1 = np.cumsum(pdf1)

count2, bins_count2 = np.histogram(stat_feature_df.loc[(stat_feature_df['target']==1),
            'q0'], bins=10)
pdf2 = count2 / sum(count2)
cdf2 = np.cumsum(pdf2)
plt.plot(bins_count1[1:], cdf1, label="target 0")
plt.plot(bins_count2[1:], cdf2, label="target 1")

plt.xlabel('q0')
plt.ylabel("cumulative sum of PDF")
plt.legend()
plt.title("CDF for both target in a single plot",fontsize=20)
plt.show()
```



if the quatile 1 of signal is less than -20 there's more probability of partial discharge

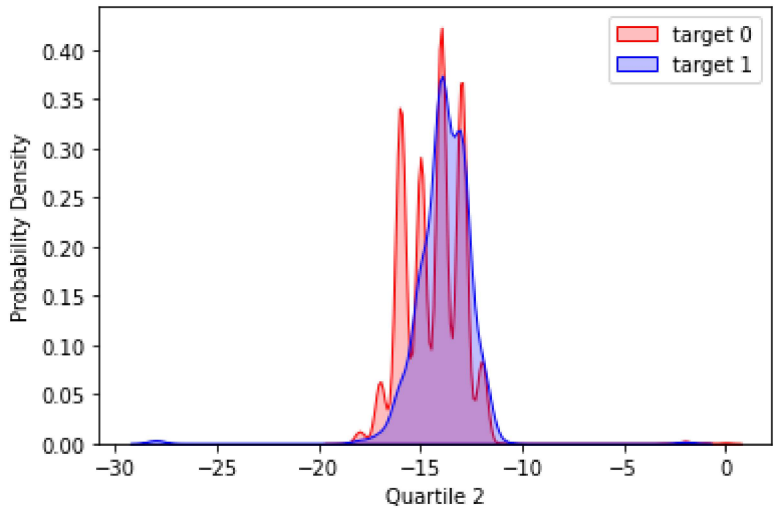
Quartile 2

In [ ]:

```
sns.kdeplot(stat_feature_df.loc[(stat_feature_df['target']==0),
            'q1'], color='r', shade=True, Label='target 0')

sns.kdeplot(stat_feature_df.loc[(stat_feature_df['target']==1),
            'q1'], color='b', shade=True, Label='target 1')
plt.legend()
plt.xlabel('Quartile 2')
plt.ylabel('Probability Density')
```

Out[39]:

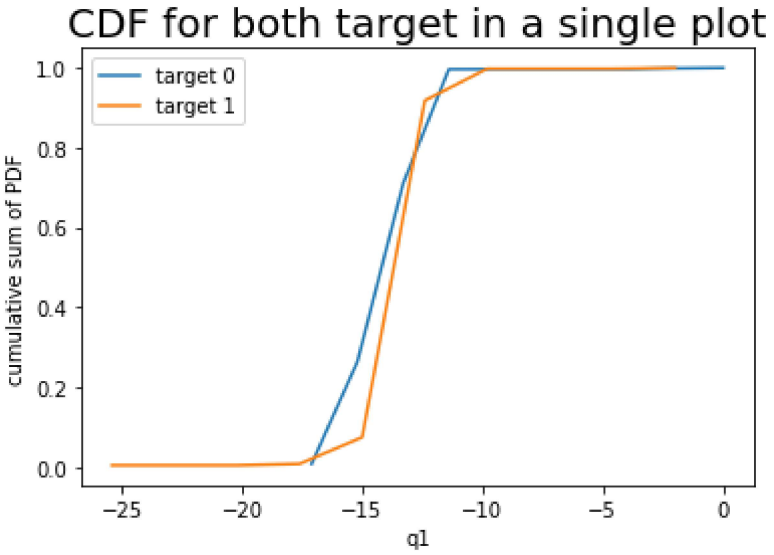
```
Text(0, 0.5, 'Probability Density')
```



Type *Markdown* and LaTeX: α^2

In [ ]:

```
count1, bins_count1 = np.histogram(stat_feature_df.loc[(stat_feature_df['target']==0),
            'q1'], bins=10)
pdf1 = count1 / sum(count1)
cdf1 = np.cumsum(pdf1)

count2, bins_count2 = np.histogram(stat_feature_df.loc[(stat_feature_df['target']==1),
            'q1'], bins=10)
pdf2 = count2 / sum(count2)
cdf2 = np.cumsum(pdf2)
plt.plot(bins_count1[1:], cdf1, label="target 0")
plt.plot(bins_count2[1:], cdf2, label="target 1")

plt.xlabel('q1')
plt.ylabel("cumulative sum of PDF")
plt.legend()
plt.title("CDF for both target in a single plot",fontsize=20)
plt.show()
```



if the quatile 2 of signal is less than -15 there's more probability of partial discharge

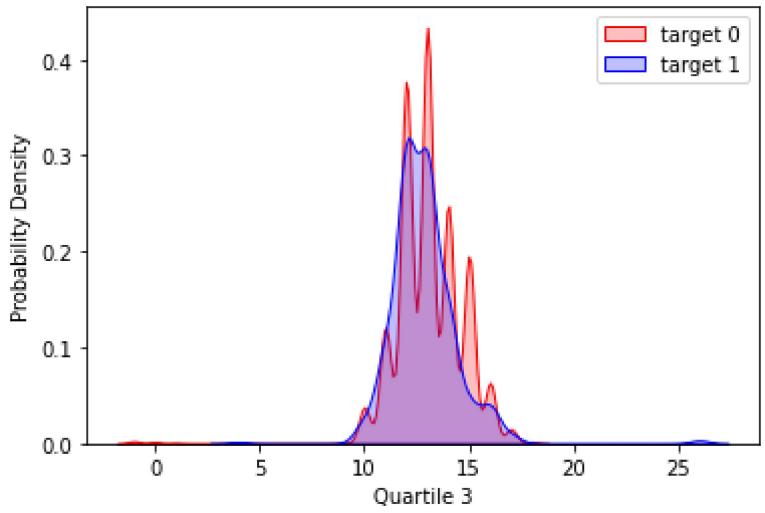
Quartile 3

In [ ]:

```
sns.kdeplot(stat_feature_df.loc[(stat_feature_df['target']==0),
            'q2'], color='r', shade=True, Label='target 0')

sns.kdeplot(stat_feature_df.loc[(stat_feature_df['target']==1),
            'q2'], color='b', shade=True, Label='target 1')
plt.legend()
plt.xlabel('Quartile 3')
plt.ylabel('Probability Density')
```

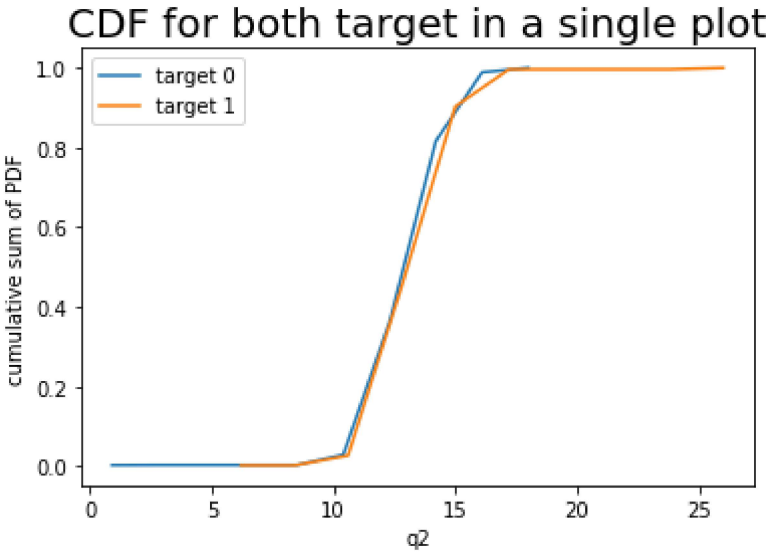
Out[40]:

```
Text(0, 0.5, 'Probability Density')
```



In [ ]:

```
count1, bins_count1 = np.histogram(stat_feature_df.loc[(stat_feature_df['target']==0),
            'q2'], bins=10)
pdf1 = count1 / sum(count1)
cdf1 = np.cumsum(pdf1)

count2, bins_count2 = np.histogram(stat_feature_df.loc[(stat_feature_df['target']==1),
            'q2'], bins=10)
pdf2 = count2 / sum(count2)
cdf2 = np.cumsum(pdf2)
plt.plot(bins_count1[1:], cdf1, label="target 0")
plt.plot(bins_count2[1:], cdf2, label="target 1")

plt.xlabel('q2')
plt.ylabel("cumulative sum of PDF")
plt.legend()
plt.title("CDF for both target in a single plot",fontsize=20)
plt.show()
```



if the quatile 3 of signal is greater than 15 there's more probability of partial discharge. if the quatile 1 of signal is less than 5 there's more probability of no partial discharge.

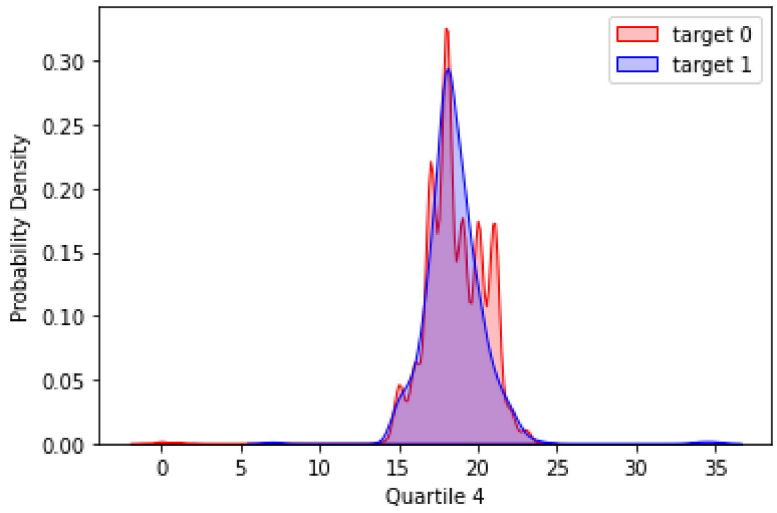
Quartile 4

In [ ]:

```
sns.kdeplot(stat_feature_df.loc[(stat_feature_df['target']==0),
            'q3'], color='r', shade=True, Label='target 0')

sns.kdeplot(stat_feature_df.loc[(stat_feature_df['target']==1),
            'q3'], color='b', shade=True, Label='target 1')
plt.legend()
plt.xlabel('Quartile 4')
plt.ylabel('Probability Density')
```

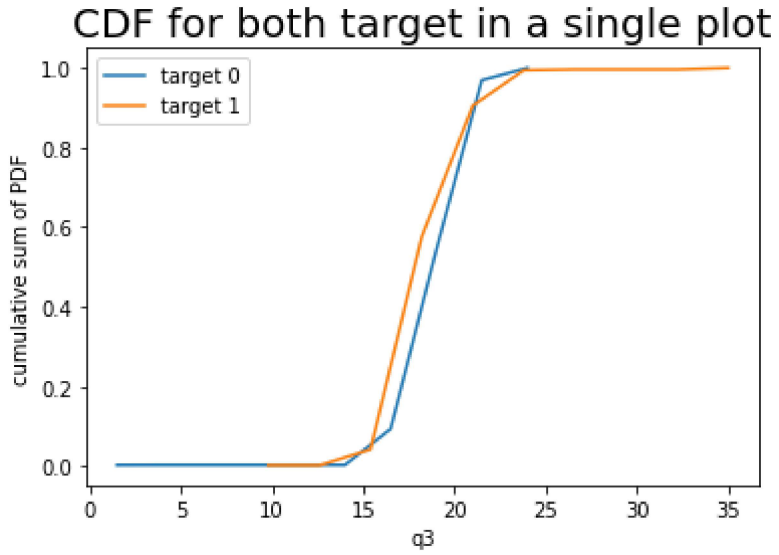
Out[41]:

```
Text(0, 0.5, 'Probability Density')
```



In [ ]:

```
count1, bins_count1 = np.histogram(stat_feature_df.loc[(stat_feature_df['target']==0),
            'q3'], bins=10)
pdf1 = count1 / sum(count1)
cdf1 = np.cumsum(pdf1)

count2, bins_count2 = np.histogram(stat_feature_df.loc[(stat_feature_df['target']==1),
            'q3'], bins=10)
pdf2 = count2 / sum(count2)
cdf2 = np.cumsum(pdf2)
plt.plot(bins_count1[1:], cdf1, label="target 0")
plt.plot(bins_count2[1:], cdf2, label="target 1")

plt.xlabel('q3')
plt.ylabel("cumulative sum of PDF")
plt.legend()
plt.title("CDF for both target in a single plot",fontsize=20)
plt.show()
```



if the quatile 4 of signal is greater than 20 there's more probability of partial discharge. if the quatile 1 of signal is less than 10 there's more probability of no partial discharge.

Bandwidth of signal

In [ ]:

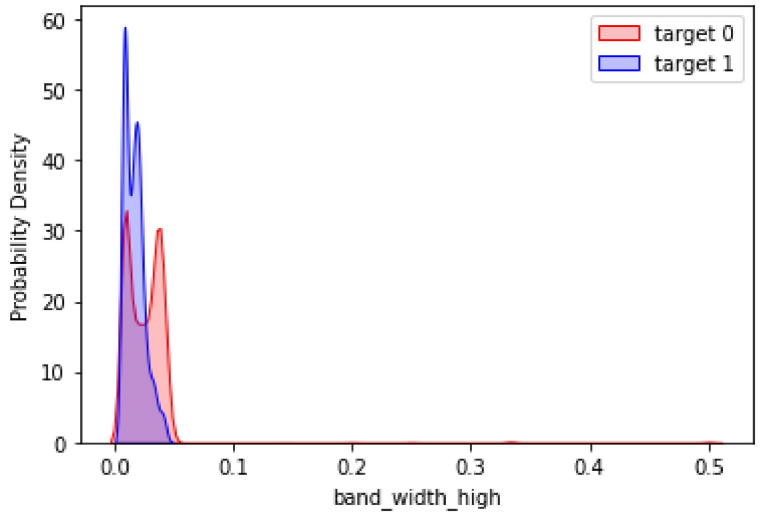
```
bw1 = []
bw2 = []
#extracting bandwiths from stats_df
for index,row in stat_feature_df.iterrows():
    bw = ast.literal_eval(row['band_width'])
    bw1.append(bw[0])
    bw2.append(bw[1])
stat_feature_df['band_width_high'] = bw1
stat_feature_df['band_width_low'] = bw2
```

In [ ]:

```
sns.kdeplot(stat_feature_df.loc[(stat_feature_df['target']==0),
            'band_width_high'], color='r', shade=True, Label='target 0')

sns.kdeplot(stat_feature_df.loc[(stat_feature_df['target']==1),
            'band_width_high'], color='b', shade=True, Label='target 1')
plt.legend()
plt.xlabel('band_width_high')
plt.ylabel('Probability Density')
```

Out[12]:

```
Text(0, 0.5, 'Probability Density')
```



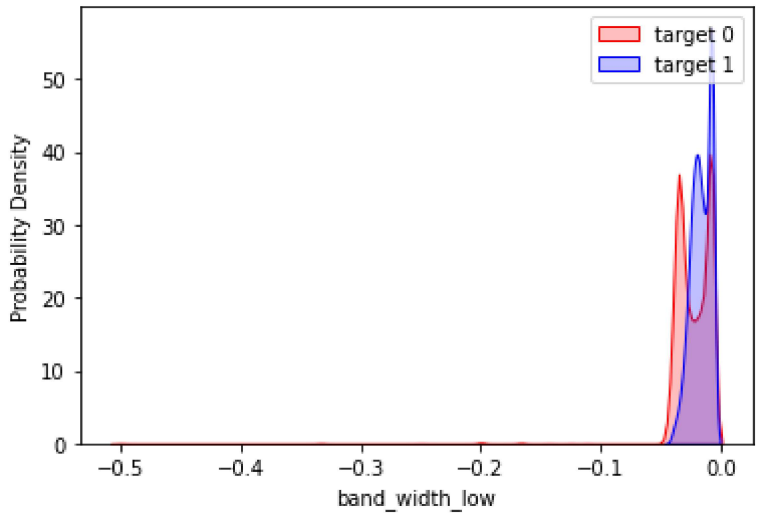
the bandwidth is almost overlapping but if value lies between 15 and 18 then most likely there is no partial discharge

In [ ]:

```
sns.kdeplot(stat_feature_df.loc[(stat_feature_df['target']==0),
            'band_width_low'], color='r', shade=True, Label='target 0')

sns.kdeplot(stat_feature_df.loc[(stat_feature_df['target']==1),
            'band_width_low'], color='b', shade=True, Label='target 1')
plt.legend()
plt.xlabel('band_width_low')
plt.ylabel('Probability Density')
```

Out[13]:

```
Text(0, 0.5, 'Probability Density')
```



If the band width is less than -15 then there is more probability of there not being partial discharge

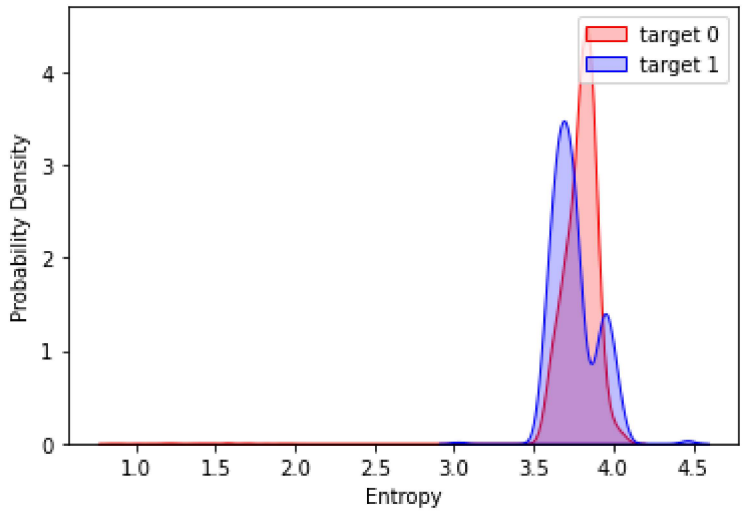
Entropy

In [ ]:

```
sns.kdeplot(stat_feature_df.loc[(stat_feature_df['target']==0),
            'entropy'], color='r', shade=True, Label='target 0')

sns.kdeplot(stat_feature_df.loc[(stat_feature_df['target']==1),
            'entropy'], color='b', shade=True, Label='target 1')
plt.legend()
plt.xlabel('Entropy')
plt.ylabel('Probability Density')
```

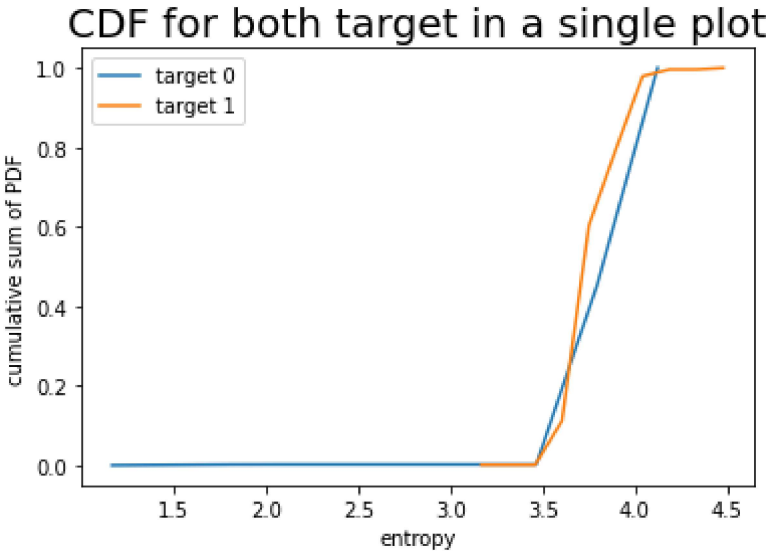
Out[45]:

```
Text(0, 0.5, 'Probability Density')
```



In [ ]:

```
count1, bins_count1 = np.histogram(stat_feature_df.loc[(stat_feature_df['target']==0),
            'entropy'], bins=10)
pdf1 = count1 / sum(count1)
cdf1 = np.cumsum(pdf1)

count2, bins_count2 = np.histogram(stat_feature_df.loc[(stat_feature_df['target']==1),
            'entropy'], bins=10)
pdf2 = count2 / sum(count2)
cdf2 = np.cumsum(pdf2)
plt.plot(bins_count1[1:], cdf1, label="target 0")
plt.plot(bins_count2[1:], cdf2, label="target 1")

plt.xlabel('entropy')
plt.ylabel("cumulative sum of PDF")
plt.legend()
plt.title("CDF for both target in a single plot",fontsize=20)
plt.show()
```



if the entropy 1 of signal is less than 3 there's more probability of no partial discharge.

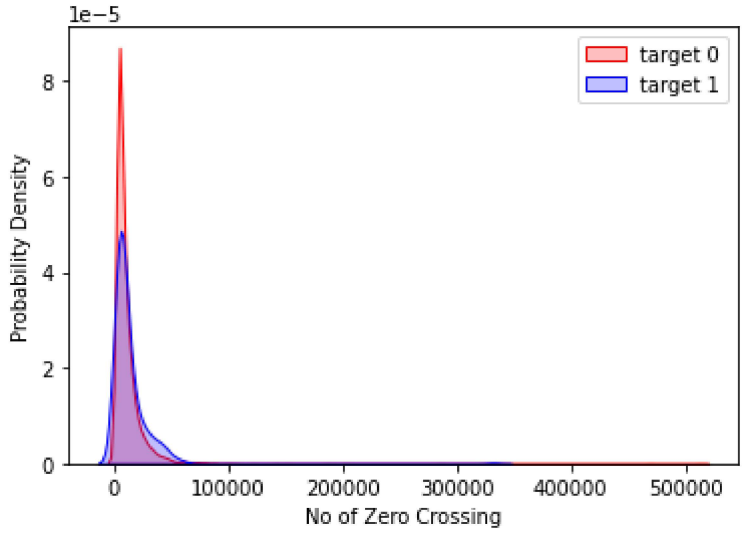
No of Zero Crossing

In [ ]:

```
sns.kdeplot(stat_feature_df.loc[(stat_feature_df['target']==0),
            'no_zero_crossings'], color='r', shade=True, Label='target 0')

sns.kdeplot(stat_feature_df.loc[(stat_feature_df['target']==1),
            'no_zero_crossings'], color='b', shade=True, Label='target 1')
plt.legend()
plt.xlabel('No of Zero Crossing')
plt.ylabel('Probability Density')
```

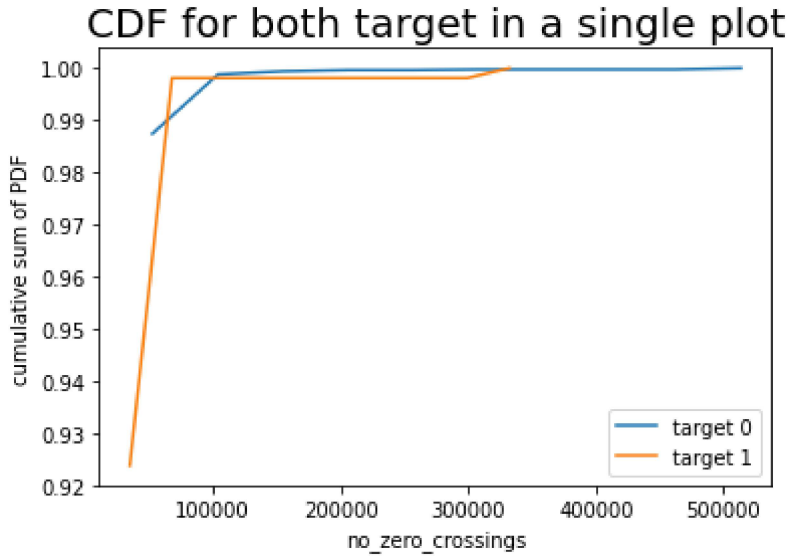
Out[46]:

```
Text(0, 0.5, 'Probability Density')
```



In [ ]:

```
count1, bins_count1 = np.histogram(stat_feature_df.loc[(stat_feature_df['target']==0),
            'no_zero_crossings'], bins=10)
pdf1 = count1 / sum(count1)
cdf1 = np.cumsum(pdf1)

count2, bins_count2 = np.histogram(stat_feature_df.loc[(stat_feature_df['target']==1),
            'no_zero_crossings'], bins=10)
pdf2 = count2 / sum(count2)
cdf2 = np.cumsum(pdf2)
plt.plot(bins_count1[1:], cdf1, label="target 0")
plt.plot(bins_count2[1:], cdf2, label="target 1")

plt.xlabel('no_zero_crossings')
plt.ylabel("cumulative sum of PDF")
plt.legend()
plt.title("CDF for both target in a single plot",fontsize=20)
plt.show()
```



if the no of zero crossing is greater than 300000 than there is more no partial discharge.

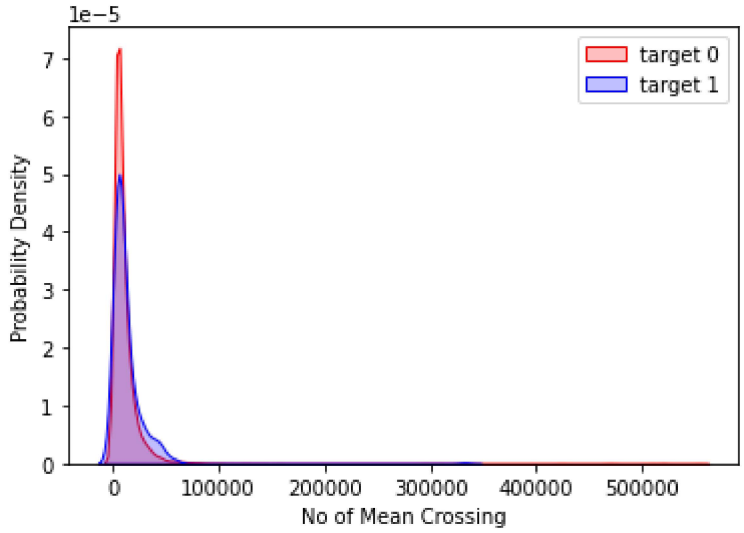
No of Mean Crossing

In [ ]:

```
sns.kdeplot(stat_feature_df.loc[(stat_feature_df['target']==0),
            'no_mean_crossings'], color='r', shade=True, Label='target 0')

sns.kdeplot(stat_feature_df.loc[(stat_feature_df['target']==1),
            'no_mean_crossings'], color='b', shade=True, Label='target 1')
plt.legend()
plt.xlabel('No of Mean Crossing')
plt.ylabel('Probability Density')
```

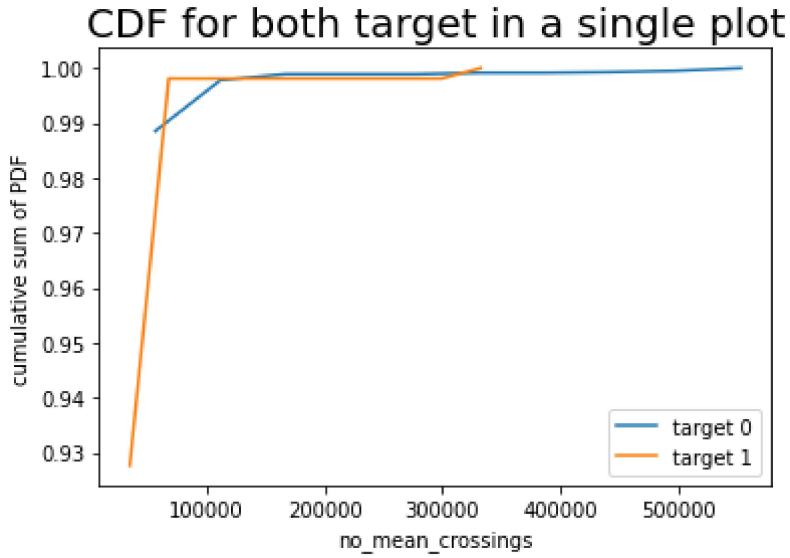
Out[47]:

```
Text(0, 0.5, 'Probability Density')
```



In [ ]:

```
count1, bins_count1 = np.histogram(stat_feature_df.loc[(stat_feature_df['target']==0),
            'no_mean_crossings'], bins=10)
pdf1 = count1 / sum(count1)
cdf1 = np.cumsum(pdf1)

count2, bins_count2 = np.histogram(stat_feature_df.loc[(stat_feature_df['target']==1),
            'no_mean_crossings'], bins=10)
pdf2 = count2 / sum(count2)
cdf2 = np.cumsum(pdf2)
plt.plot(bins_count1[1:], cdf1, label="target 0")
plt.plot(bins_count2[1:], cdf2, label="target 1")

plt.xlabel('no_mean_crossings')
plt.ylabel("cumulative sum of PDF")
plt.legend()
plt.title("CDF for both target in a single plot",fontsize=20)
plt.show()
```



if the no of mean crossing is greater than 300000 than there is more no partial discharge.

Feature Engineering

Refernce:[https://www.kaggle.com/junkoda/handmade-features (https://www.kaggle.com/junkoda/handmade-features)](https://www.kaggle.com/junkoda/handmade-features)

Spectra of a signal

we initally calculate the mean and percentile for every chunk of 1000 values

In [ ]:

```
# Load training data
import numpy as np
import pandas as pd
import pyarrow.parquet as pq
from tqdm import tqdm_notebook as tqdm
```

In [ ]:

```
def compute_spectra(signals, *, m = 1000):
    means = []
    percentiles = []
    percentile_values = (100,99,95,0,1,5)
    for raw_signal in tqdm(signals):

        #normalizing the signal values
        signal = raw_signal.astype('float32').reshape(-1, m) / 128.0

        #mean of signal
        mean = np.mean(signal,axis=1)

        #percentiles of signal
        percentile = np.abs(np.percentile(signal,percentile_values,axis=1)-mean)

        #calaculating the baseline of percentiles
        baseline = np.percentile(percentile,5.0)

        #subtracting the baseline
        percentile = np.maximum(0.0,percentile-baseline)

        means.append(mean)

        percentiles.append(percentile.T)

    Dict= {}
    Dict['mean'] = np.array(means)
    Dict['percentile'] = np.array(percentiles)

    return Dict
```

Peak interval

Then, within the 800 chunks of the spectra,the peak interval of width=150 that contains the maximum deviation in the max - mean spectrum.

a top hat filter selects a band of signal of desired frequency by the specification of a lower and upper bounding frequencies

code snippet taken from:https://www.kaggle.com/junkoda/handmade-features (https://www.kaggle.com/junkoda/handmade-features)

In [ ]:

```
import tensorflow as tf
tf.compat.v1.disable_eager_execution()
def max_windowed(spec, *, width=150, stride=10):
    """
    Smooth the spectrum with a tophat window function and find the
    peak inteval that maximises the smoothed spectrum.

    Returns: d(dict)
      d['w'] (array): smoothed max - mean spectrum
      d['ibegin'] (array): the left edge index of the peak interval
    """
    n = spec.shape[0]
    length = spec.shape[1] # 800
    nspec = spec.shape[2] # 6 spectra

    n_triplet = n // 3

    # Reorganize the max spectrum from 8712 data to 2904 triplets with 3 phases
    max_spec = np.empty((n_triplet, length, 3))
    for i in range(n_triplet):
        max_spec[i, :, 0] = spec[3*i, :, 0] # phase 0
        max_spec[i, :, 1] = spec[3*i + 1, :, 0] # phase 1
        max_spec[i, :, 2] = spec[3*i + 2, :, 0] # phase 2

    x = tf.compat.v1.placeholder(tf.float32, [None, length, 3]) # input spectra before smoo
    # 800 -> 80: static convolaution
    # convolution but not CNN, the kernel is static
    # smoothing/convolution kernel
    # tophat window function
    # shape (3, 1) adds up 3 phases to one output
    K = np.ones((width, 3, 1), dtype='float32') / width

    W_conv1 = tf.constant(K)
    h_conv1 = tf.nn.conv1d(x, W_conv1, stride=stride, padding='VALID')

    with tf.compat.v1.Session() as sess:
        w = sess.run(h_conv1, feed_dict={x:max_spec})

    imax = np.argmax(w[:, :, 0], axis=1) # index of maximum smoothed spectrum

    Dict = {}
    Dict['w'] = w # smoothed max spectrum
    Dict['ibegin'] = imax*stride

    return Dict
```

1.From the peak interval calculated above, we extract features like mean and max

2.Instead of considering each phase independently we combine all the 3 phases of a signal

In [ ]:

```
def compute_spectra_features(spectra,peaks):


    percentiles = spectra['percentile']

    n = percentiles.shape[0]
    length = percentiles.shape[1]
    nspec = percentiles.shape[2]

    no_signals = n//3

    phase_percentiles = np.empty((no_signals,length,nspec,3))

    #creating an array which combines all 3 phases
    for i in range(no_signals):
        phase_percentiles[i, :, :, 0] = percentiles[3*i, :, :] # phase 0
        phase_percentiles[i, :, :, 1] = percentiles[3*i + 1, :, :] # phase 1
        phase_percentiles[i, :, :, 2] = percentiles[3*i + 2, :, :] # phase 2


    width = 150

    no_perc_features = 3

    #array to store final features
    spectra_features = np.empty((no_signals,no_perc_features*nspec*3 + 3))

    #array to store percentile features
    perc_phase_features = np.empty((no_signals,no_perc_features,nspec,3))


    for i in range(no_signals):

        #max of the total percentile features
        perc_phase_features[i,0,:,:] = np.max(phase_percentiles[i,:,:,:],axis = 0)

        peak_start = peaks['ibegin'][i]
        peak_end = peak_start +width
        peak_mid = peak_start+width//2

        #mean of the percentle features in peak interval
        perc_phase_features[i,1,:,:] = np.mean(phase_percentiles[i,peak_start:peak_end,:,:]

        #max of the percentle features in peak interval
        perc_phase_features[i,2,:,:] = np.max(phase_percentiles[i,peak_start:peak_end,:,:],

        #storing the mean value at the mid index of peak interval of each phase
        spectra_features[i,0] = spectra['mean'][3*i,peak_mid]
        spectra_features[i,1] = spectra['mean'][3*i+1,peak_mid]
        spectra_features[i,2] = spectra['mean'][3*i+2,peak_mid]

    #storing all the features
    shape = perc_phase_features.shape
    spectra_features[:,3:] = perc_phase_features.reshape(shape[0], shape[1]*shape[2]*shape[

    return spectra_features
```

In [ ]:

```
```

calculating spectra features of train dataset

In [ ]:

```
X_train = []
for i in range(8):

    begin = 3*int(8712//3 *(i/8))

    end = 3*int(8712//3 *((i+1)/8))

    print('{}-{}'.format(begin,end))

    raw_signal_data_train= pq.read_pandas('/content/train.parquet',columns=[str(i) for i in ir

    #print(raw_signal_data_test.shape)
    spec_train = compute_spectra(raw_signal_data_train)
    train_peaks = max_windowed(spec_train['percentile'])

    train_spectra_features = compute_spectra_features(spec_train,train_peaks)

    X_train.append(train_spectra_features)

    del raw_signal_data_train
```

```
0-1089
100%|██████████| 1089/1089 [00:23<00:00, 45.47it/s]
1089-2178
100%|██████████| 1089/1089 [00:26<00:00, 40.69it/s]
2178-3267
100%|██████████| 1089/1089 [00:29<00:00, 36.75it/s]
3267-4356
100%|██████████| 1089/1089 [00:27<00:00, 39.64it/s]
4356-5445
100%|██████████| 1089/1089 [00:23<00:00, 46.04it/s]
5445-6534
100%|██████████| 1089/1089 [00:26<00:00, 41.45it/s]
6534-7623
100%|██████████| 1089/1089 [00:31<00:00, 34.07it/s]
7623-8712
100%|██████████| 1089/1089 [00:28<00:00, 38.84it/s]
```

In [ ]:

```
#concatenate all 8 parts
X_train_spectra = np.concatenate(X_train, axis=0)
```

In [ ]:

```
np.save('X_train_spectra',X_train_spectra)
```

In [ ]:

```
X_train_spectra.shape
```

Out[16]:

```
(2904, 57)
```

calculating spectra features of test dataset

In [ ]:

```
X_tests = []

# we divide the total dataset into 4 parts in order to fit it in the RAM
#and then we calculate the features
for i in range(16):

    begin = 8712 + 3*int(20337//3 *(i/16))

    end = 8712 + 3*int(20337//3 *((i+1)/16))

    print('{}-{}'.format(begin,end))

    raw_signal_data_test = pq.read_pandas('/content/test.parquet',columns=[str(i) for i in

    #print(raw_signal_data_test.shape)
    test_spectra = compute_spectra(raw_signal_data_test)

    test_peaks = max_windowed(test_spectra['percentile'])

    test_spectra_features = compute_spectra_features(test_spectra,test_peaks)

    X_tests.append(test_spectra_features)

    del raw_signal_data_test
```

```
8712-9981
100%|██████████| 1269/1269 [00:28<00:00, 44.63it/s]
9981-11253
100%|██████████| 1272/1272 [00:35<00:00, 36.21it/s]
11253-12525
100%|██████████| 1272/1272 [00:30<00:00, 41.54it/s]
12525-13794
100%|██████████| 1269/1269 [00:28<00:00, 45.23it/s]
13794-15066
100%|██████████| 1272/1272 [00:27<00:00, 45.86it/s]
15066-16338
100%|██████████| 1272/1272 [00:28<00:00, 45.18it/s]
16338-17607
100%|██████████| 1269/1269 [00:27<00:00, 45.92it/s]
17607-18879
100%|██████████| 1272/1272 [00:27<00:00, 46.07it/s]
18879-20151
100%|██████████| 1272/1272 [00:28<00:00, 45.02it/s]
```

```
20151-21420
100%|██████████| 1269/1269 [00:28<00:00, 45.09it/s]
21420-22692
100%|██████████| 1272/1272 [00:27<00:00, 45.52it/s]
22692-23964
100%|██████████| 1272/1272 [00:27<00:00, 45.83it/s]
23964-25233
100%|██████████| 1269/1269 [00:27<00:00, 46.38it/s]
25233-26505
100%|██████████| 1272/1272 [00:27<00:00, 45.77it/s]
26505-27777
100%|██████████| 1272/1272 [00:27<00:00, 46.01it/s]
27777-29049
100%|██████████| 1272/1272 [00:27<00:00, 45.96it/s]
```

In [ ]:

```
#concatenate all 16 parts
X_test_spectra = np.concatenate(X_tests, axis=0)
```

In [ ]:

```
np.save('X_test_spectra',X_test_spectra)
```

In [ ]:

```
X_test_spectra.shape
```

Out[15]:

```
(6779, 57)
```

Target

we find the target by taking the value which occured more among the 3 phases

In [ ]:

```
y_train = []

for i in range(0,len(df_metadata_train),3):
    y_train.append(df_metadata_train.loc[i]['target'])
```

In [ ]:

```
y_train=np.array(y_train)
```

In [ ]:

```
y_train.shape
```

Out[28]:

```
(2904,)
```

In [ ]:

```
#loading the spectra features
X_train = np.load('X_train_spectra.npy')
```

In [ ]:

```
#loading y_train
y_train=np.load('y.npy')
```

In [ ]:

```
#data structures
import pandas as pd
import numpy as np

from sklearn.model_selection import train_test_split,GridSearchCV,RandomizedSearchCV
from sklearn.metrics import matthews_corrcoef,make_scorer
from sklearn.ensemble import GradientBoostingClassifier
import xgboost as xgb
from sklearn import tree
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier,StackingClassifier,AdaBoostClassifier
from sklearn import svm
from tqdm import tqdm
import ast
import pickle

import warnings
warnings.filterwarnings('ignore')
```

Logistic Regression

In [ ]:

```
# Parameters to tune for LR model
params = {'C': [10**x for x in range(-5,6)]}

# Create a custom (MCC) metric for evaluation of the model performance while
# hyperparameter tuning the XGBoost model
mcc = make_scorer(matthews_corrcoef, greater_is_better=True)

# Create an XGBoost classifier object with log-loss as the loss function to minimize
log_clf = LogisticRegression(random_state=42, class_weight='balanced')


# Perform stratified 5-fold cross validation
grid_clf = GridSearchCV(log_clf, params, scoring=mcc, cv=5, return_train_score=True)

# Fit the model
grid_clf .fit(X_train, y_train)
print(f"Best CV score: {grid_clf.best_score_}")
```

```
Best CV score: 0.5582797151965972
```

In [ ]:

```
grid_clf.best_params_
```

Out[15]:

```
{'C': 1000}
```

Linear Support Vector Machines

In [ ]:

```
# hyperparameter tuning the XGBoost model
params ={'C':[10 ** x for x in range(-5, 3)]}
# Create a custom (MCC) metric for evaluation of the model performance while

mcc = make_scorer(matthews_corrcoef, greater_is_better=True)

# Create an XGBoost classifier object with log-loss as the loss function to minimize
svm_clf = svm.SVC(random_state=42, class_weight='balanced')


# Perform stratified 5-fold cross validation
grid_clf = GridSearchCV(svm_clf, params, scoring=mcc, cv=5, return_train_score=True)

# Fit the model
grid_clf .fit(X_train, y_train)
print(f"Best CV score: {grid_clf.best_score_}")
```

```
Best CV score: 0.5449976219659411
```

In [ ]:

```
grid_clf.best_params_
```

Out[17]:

```
{'C': 100}
```

Kernel SVM

In [ ]:

```
# hyperparameter tuning the XGBoost model
params ={'C':[10 ** x for x in range(-5, 3)],'gamma':[10 ** x for x in range(-5, 3)]}
# Create a custom (MCC) metric for evaluation of the model performance while

mcc = make_scorer(matthews_corrcoef, greater_is_better=True)

# Create an XGBoost classifier object with log-loss as the loss function to minimize
svm_clf = svm.SVC(random_state=42,kernel='rbf',class_weight='balanced')


# Perform stratified 5-fold cross validation
grid_clf = GridSearchCV(svm_clf, params, scoring=mcc, cv=5, return_train_score=True)

# Fit the model
grid_clf .fit(X_train, y_train)
print(f"Best CV score: {grid_clf.best_score_}")
```

```
Best CV score: 0.5599262239333939
```

In [ ]:

```
grid_clf.best_params_
```

Out[19]:

```
{'C': 100, 'gamma': 0.1}
```

Decision Tree

In [ ]:

```
# hyperparameter tuning the DecisionTree model
params ={'max_depth':[1, 5, 10, 50],'min_samples_split':[5, 10, 100, 500]}
# Create a custom (MCC) metric for evaluation of the model performance while

mcc = make_scorer(matthews_corrcoef, greater_is_better=True)

# Create an XGBoost classifier object with log-loss as the loss function to minimize
dt_clf = tree.DecisionTreeClassifier(random_state=42, class_weight='balanced')


# Perform stratified 5-fold cross validation
grid_clf = GridSearchCV(dt_clf, params, scoring=mcc, cv=5, return_train_score=True)

# Fit the model
grid_clf .fit(X_train, y_train)
print(f"Best CV score: {grid_clf.best_score_}")
```

```
Best CV score: 0.5956996400792118
```

In [ ]:

```
grid_clf.best_params_
```

Out[21]:

```
{'max_depth': 10, 'min_samples_split': 5}
```

Random Forest Classifier

In [ ]:

```
# Parameters to tune for random forest model
params = {'n_estimators': [10, 50, 100, 500, 1000],
          'max_depth': [2, 3, 4, 5, 6],
          'min_samples_split': [0.02, 0.04, 0.08, 0.16, 0.32, 0.50]}

# Create a custom (MCC) metric for evaluation of the model performance while
# hyperparameter tuning the XGBoost model
mcc = make_scorer(matthews_corrcoef, greater_is_better=True)

# Create an XGBoost classifier object with log-loss as the loss function to minimize
rf_clf = RandomForestClassifier(random_state=42,
                                class_weight='balanced')




# Perform stratified 5-fold cross validation
rand_clf = RandomizedSearchCV(rf_clf, param_distributions=params, scoring=mcc,
                              cv=5, random_state=42, return_train_score=True,
                              n_jobs=-1)

# Fit the model
rand_clf.fit(X_train, y_train)
print(f"Best CV score: {rand_clf.best_score_}")
```

```
Best CV score: 0.6058456380913314
```

In [ ]:

```
rand_clf.best_params_
```

Out[23]:

```
{'max_depth': 4, 'min_samples_split': 0.02, 'n_estimators': 1000}
```

Gradient Boosting Classifier

In [ ]:

```
# Parameters to tune for XGBoost model
params = {'n_estimators': [10, 50, 100, 500, 1000],
          'max_depth': [2, 3, 4, 5, 6],
          'learning_rate': [0.0001,0.005,0.001,0.05, 0.1]}

# Create a custom (MCC) metric for evaluation of the model performance while
# hyperparameter tuning the XGBoost model
mcc = make_scorer(matthews_corrcoef, greater_is_better=True)

# Create an XGBoost classifier object with log-loss as the loss function to minimize
gbdt_clf = GradientBoostingClassifier(random_state=10)



# Perform stratified 5-fold cross validation
rand_clf = RandomizedSearchCV(gbdt_clf, param_distributions=params, scoring=mcc,
                              cv=5, random_state=42, return_train_score=True,
                              n_jobs=-1)

# Fit the model
rand_clf.fit(X_train, y_train)
print(f"Best CV score: {rand_clf.best_score_}")
```

```
Best CV score: 0.6726505805756288
```

In [ ]:

```
rand_clf.best_params_
```

Out[42]:

```
{'learning_rate': 0.1, 'max_depth': 6, 'n_estimators': 500}
```

AdaBoostClassifier

In [ ]:

```
# Parameters to tune for XGBoost model
params = {'n_estimators': [10, 50, 100, 500, 1000],
          'learning_rate': [0.0001,0.005,0.001,0.05, 0.1]}

# Create a custom (MCC) metric for evaluation of the model performance while
# hyperparameter tuning the XGBoost model
mcc = make_scorer(matthews_corrcoef, greater_is_better=True)

# Create an XGBoost classifier object with log-loss as the loss function to minimize
ada_clf = AdaBoostClassifier(random_state=10)



# Perform stratified 5-fold cross validation
rand_clf = RandomizedSearchCV(ada_clf, param_distributions=params, scoring=mcc,
                              cv=5, random_state=42, return_train_score=True,
                              n_jobs=-1)

# Fit the model
rand_clf.fit(X_train, y_train)
print(f"Best CV score: {rand_clf.best_score_}")
```

```
Best CV score: 0.6539008854750454
```

In [ ]:

```
rand_clf.best_params_
```

Out[40]:

```
{'learning_rate': 0.1, 'n_estimators': 500}
```

XGBoost Classifier

In [ ]:

```
# Parameters to tune for XGBoost model
params = {'n_estimators': [10, 50, 100, 500, 1000],
          'max_depth': [2, 3, 4, 5, 6],
          'learning_rate': [0.0001,0.005,0.001,0.05, 0.1],
          'reg_alpha': [1e-3, 1e-2, 1e-1, 1e0, 1e1, 1e2],
          'reg_lambda': [1e-3, 1e-2, 1e-1, 1e0, 1e1, 1e2]}

# Create a custom (MCC) metric for evaluation of the model performance while
# hyperparameter tuning the XGBoost model
mcc = make_scorer(matthews_corrcoef, greater_is_better=True)

# Create an XGBoost classifier object with log-loss as the loss function to minimize
xgb_clf = xgb.XGBClassifier(random_state=42)



# Perform stratified 5-fold cross validation
rand_clf = RandomizedSearchCV(xgb_clf, param_distributions=params, scoring=mcc,
                              cv=5, random_state=42, return_train_score=True,
                              n_jobs=-1)

# Fit the model
rand_clf.fit(X_train, y_train)
print(f"Best CV score: {rand_clf.best_score_}")
```

```
Best CV score: 0.6856760456886946
```

In [ ]:

```
rand_clf.best_params_
```

Out[27]:

```
{'learning_rate': 0.05,
 'max_depth': 4,
 'n_estimators': 500,
 'reg_alpha': 0.001,
 'reg_lambda': 1.0}
```

Stacking Classifier

In [ ]:

```
import six
import sys
sys.modules['sklearn.externals.six'] = six
from mlxtend.classifier import StackingClassifier
```

In [ ]:

```
clf1=LogisticRegression(random_state=42, C=1000,class_weight='balanced')
#clf1.fit(X_train,y_train)
clf2=svm.SVC(random_state=42, C=100,class_weight='balanced')
#clf2.fit(X_train,y_train)
clf3=svm.SVC(random_state=42, C=100,gamma= 0.1,kernel='rbf',class_weight='balanced')
#clf3.fit(X_train,y_train)
clf4=tree.DecisionTreeClassifier(random_state=42,max_depth=10, min_samples_split=5,class_we
#clf4.fit(X_train,y_train)
clf5=RandomForestClassifier(random_state=42,max_depth=4, min_samples_split=0.02, n_estimato
#clf5.fit(X_train,y_train)
clf6=GradientBoostingClassifier(random_state=10,learning_rate=0.1, max_depth=5, n_estimator
#clf6.fit(X_train,y_train)
clf7=xgb.XGBClassifier(random_state=42,learning_rate=0.05,max_depth=4,n_estimators=500,reg_
#clf7.fit(X_train,y_train)
# Parameters to tune for LR model
```

In [ ]:

```
from sklearn.model_selection import StratifiedKFold,train_test_split,RandomizedSearchCV
```

In [ ]:

```
# split into train and CV data
n_splits = 5
splits = list(StratifiedKFold(n_splits=n_splits, shuffle=True).split(X_train,y_train))

models = []
scores = np.zeros(n_splits)

print('Training...')
print('MCC training & cv') # MCC = Matthews Correlation Coefficient
for i, (idx_train, idx_cv) in enumerate(splits):

    #train and cv split
    X_tr = X_train[idx_train, :]
    y_tr = y_train[idx_train]

    X_cv = X_train[idx_cv, :]
    y_cv = y_train[idx_cv]

    #initalizing and fitting the model
    lr = LogisticRegression()  # defining meta-classifier

    model =StackingClassifier(classifiers =[clf1,clf2,clf3,clf3,clf4,clf5,clf6,clf7], meta_
    model.fit(X_tr, y_tr.astype(float))

    #prediction
    y_predict_train = model.predict(X_tr)
    y_predict_cv = model.predict(X_cv)

    #calculating mcc metric
    score_train = matthews_corrcoef(y_tr, y_predict_train)
    score_cv = matthews_corrcoef(y_cv, y_predict_cv)

    #storing the models
    models.append(model)
    scores[i] = score_cv

    #printing the train and cross validation mcc score
    print('%d %.3f %.3f' % (i, score_train, score_cv))

#average of all the scores
print('CV scores %.3f ± %.3f' % (np.mean(scores), np.std(scores)))
```

```
Training...
MCC training & cv
0 1.000 0.565
1 1.000 0.680
2 1.000 0.709
3 1.000 0.709
4 1.000 0.754
CV scores 0.683 ± 0.063
```

In [ ]:

```
X_test = np.load('X_test_spectra.npy')
```

In [ ]:

```
y_test_probas = np.empty((X_test.shape[0], 5))
model= xgb.XGBClassifier(learning_rate=0.5,max_depth=4,n_estimators=10,reg_alpha=0.01,reg_l
model=model.fit(X_train,y_train)
for i in range(5):

    y_test_probas[:,i] = model.predict_proba(X_test)[:,1]

#taking mean of all the predicted
y_test_proba = np.mean(y_test_probas, axis=1)

# Converting to 0 1 with a threshold 0.25, then replicating 3 copies for 3 phases
y_predict = np.repeat(y_test_proba > 0.25, 3)
```

In [ ]:

```
results_df = pd.DataFrame()
```

In [ ]:

```
signal_id = list(range(len(y_predict)))
signal_id = [i+8712 for i in signal_id]
```

In [ ]:

```
results_df['signal_id'] = signal_id
results_df['target'] = y_predict
```

In [ ]:

```
results_df
```

Out[63]:

	signal_id	target
0	8712	False
1	8713	False
2	8714	False
3	8715	False
4	8716	False
...	...	...
20332	29044	False
20333	29045	False
20334	29046	False
20335	29047	False
20336	29048	False

20337 rows × 2 columns

In [ ]:

```
results_df.to_csv('submission.csv',index=False)
```

After submitting submission file in kaggle I got score of 0.589

In [ ]:

```
from keras.layers import *
from keras.models import *
from keras import backend as K # The backend gives us access to tensorflow operations and a
from keras import optimizers # Allows to access Adam class and modify some parameters
from keras.callbacks import * # This object helps the model to train in a smarter way, avoi
from keras import activations
from keras import regularizers
from keras import initializers
from keras import constraints
from tensorflow.keras.layers import Attention,LSTM,GRU
from keras.utils.vis_utils import plot_model
import tensorboard
%load_ext tensorboard
import tensorflow.compat.v1 as tf

# https://stackoverflow.com/a/56569206/4699076
tf.disable_eager_execution()

from sklearn.model_selection import GridSearchCV, StratifiedKFold

import concurrent.futures
import multiprocessing
from sklearn.model_selection import train_test_split
```

Performance Metric

In [ ]:

```
# Matthews correlation coefficient calculation used inside Keras model
def matthews_correlation(y_true, y_pred):
  """
  Calculate Matthews Correlation Coefficient.

  References
  ----------
  .. [1] https://en.wikipedia.org/wiki/Matthews_correlation_coefficient
  .. [2] https://www.kaggle.com/tarunpaparaju/vsb-competition-attention-bilstm-with-feature
  """
  y_pred_positive = K.round(K.clip(y_pred, 0, 1))
  y_pred_negative = 1 - y_pred_positive

  y_positive = K.round(K.clip(y_true, 0, 1))
  y_negative= 1 - y_positive

  tp = K.sum(y_positive * y_pred_positive)
  tn = K.sum(y_negative * y_pred_negative)

  fp = K.sum(y_negative * y_pred_positive)
  fn = K.sum(y_positive * y_pred_negative)

  numerator = (tp * tn - fp * fn)
  denominator = K.sqrt((tp + fp) * (tp + fn) * (tn + fp) * (tn + fn))

  return numerator / (denominator + K.epsilon())
```

In [ ]:

```
max_num = 127
min_num = -128
```

In [ ]:

```
# This function standardize the data from (-128 to 127) to (-1 to 1)
# Theoretically it helps in the NN Model training, but I didn't tested without it
def standardize_data(signal, min_data, max_data, range_needed=(-1,1)):
    if min_data < 0:
        signal_std = (signal + abs(min_data)) / (max_data + abs(min_data))
    else:
        signal_std = (signal - min_data) / (max_data - min_data)
    if range_needed[0] < 0:
        return signal_std * (range_needed[1] + abs(range_needed[0])) + range_needed[0]
    else:
        return signal_std * (range_needed[1] - range_needed[0]) + range_needed[0]
```

In [ ]:

```
# This is one of the most important peace of code of this Kernel
# Any power line contain 3 phases of 800000 measurements, or 2.4 millions data
# It would be praticaly impossible to build a NN with an input of that size
# The ideia here is to reduce it each phase to a matrix of <n_dim> bins by n features
# Each bean is a set of 5000 measurements (800000 / 160), so the features are extracted fro
def transform_signal(signal, n_dim=160, min_max=(-1,1)):
    # convert data into -1 to 1
    signal_std = standardize_data(signal, min_data=min_num, max_data=max_num)
    # bucket or chunk size, 5000 in this case (800000 / 160)
    bucket_size = int(800000  / n_dim)
    # new_ts will be the container of the new data
    new_signal = []
    # this for iteract any chunk/bucket until reach the whole sample_size (800000)
    for i in range(0, 800000 , bucket_size):
        # cut each bucket to ts_range
        signal_range = signal_std[i:i + bucket_size]
        # calculate each feature
        mean = signal_range.mean()
        std = signal_range.std() # standard deviation
        std_top = mean + std # I have to test it more, but is is like a band
        std_bot = mean - std
        # I think that the percentiles are very important, it is like a distribuiton analys
        percentil_calc = np.percentile(signal_range, [0, 1, 25, 50, 75, 99, 100])
        max_range = percentil_calc[-1] - percentil_calc[0] # this is the amplitude of the c
        relative_percentile = percentil_calc - mean # maybe it could heap to understand the
        # now, we just add all the features to new_ts and convert it to np.array
        new_signal.append(np.concatenate([np.asarray([mean, std, std_top, std_bot, max_rang
    return np.asarray(new_signal)
```

In [ ]:

```
# this function take a piece of data and convert using transform_signal(), but it does to e
# if we would try to do in one time, could exceed the RAM Memmory
def data_preparation(start, end,praq_train):
    # load a piece of data from file
    praq_train = pq.read_pandas('/content/train.parquet', columns=[str(i) for i in range(st
    X = []
    y = []
    # using tdqm to evaluate processing time
    # takes each index from df_train and iteract it from start to end
    # it is divided by 3 because for each id_measurement there are 3 id_signal, and the sta
    for id_measurement in tqdm(df_metadata_train.index.levels[0].unique()[int(start/3):int(
        X_signal = []
        # for each phase of the signal
        for phase in [0,1,2]:
            # extract from df_train both signal_id and target to compose the new data sets
            signal_id, target = df_metadata_train.loc[id_measurement].loc[phase]
            # but just append the target one time, to not triplicate it
            if phase == 0:
                y.append(target)
            # extract and transform data into sets of features
            X_signal.append(transform_signal(praq_train[str(signal_id)]))
        # concatenate all the 3 phases in one matrix
        X_signal = np.concatenate(X_signal, axis=1)
        # add the data to X
        X.append(X_signal)
    X = np.asarray(X)
    y = np.asarray(y)
    return X, y
```

In [ ]:

```
# this code is very simple, divide the total size of the df_train into 8 sets and process i
X = []
y= []
for i in range(8):

    begin = 3*int(8712//3 *(i/8))

    end = 3*int(8712//3 *((i+1)/8))

    print('{}-{}'.format(begin,end))

    raw_signal_data_train= pq.read_pandas('/content/train.parquet', columns=[str(i) for i i

    X_temp, y_temp = data_preparation(begin, end,raw_signal_data_train)
    X.append(X_temp)
    y.append(y_temp)

    del raw_signal_data_train
```

```
0-1089
100%|██████████| 363/363 [01:30<00:00,  3.99it/s]
1089-2178
100%|██████████| 363/363 [01:33<00:00,  3.89it/s]
2178-3267
100%|██████████| 363/363 [01:52<00:00,  3.22it/s]
3267-4356
100%|██████████| 363/363 [01:16<00:00,  4.72it/s]
4356-5445
100%|██████████| 363/363 [01:16<00:00,  4.74it/s]
5445-6534
100%|██████████| 363/363 [01:15<00:00,  4.80it/s]
6534-7623
100%|██████████| 363/363 [01:15<00:00,  4.81it/s]
7623-8712
100%|██████████| 363/363 [01:15<00:00,  4.79it/s]
```

In [ ]:

```
X = np.concatenate(X)
y = np.concatenate(y)
```

In [ ]:

```
X=np.load('X.npy')
y=np.load('y.npy')
```

In [ ]:

```
# The X shape here is very important. It is also important undertand a little how a LSTM wo
# X.shape[0] is the number of id_measuremts contained in train data
# X.shape[1] is the number of chunks resultant of the transformation, each of this date ent
# This way the LSTM can understand the position of a data relative with other and activate
# a serie of inputs in a specifc order.
# X.shape[3] is the number of features multiplied by the number of phases (3)
print(X.shape, y.shape)
# save data into file, a numpy specific format
np.save("X.npy",X)
np.save("y.npy",y)
```

```
(2904, 160, 57) (2904,)
```

LSTM MODEL

In [ ]:

```
# https://www.kaggle.com/suicaokhoailang/lstm-attention-baseline-0-652-lb

class Attention(Layer):
    def __init__(self, step_dim,
                 W_regularizer=None, b_regularizer=None,
                 W_constraint=None, b_constraint=None,
                 bias=True, **kwargs):
        self.supports_masking = True
        self.init = initializers.get('glorot_uniform')

        self.W_regularizer = regularizers.get(W_regularizer)
        self.b_regularizer = regularizers.get(b_regularizer)

        self.W_constraint = constraints.get(W_constraint)
        self.b_constraint = constraints.get(b_constraint)

        self.bias = bias
        self.step_dim = step_dim
        self.features_dim = 0
        super(Attention, self).__init__(**kwargs)

    def build(self, input_shape):
        assert len(input_shape) == 3

        self.W = self.add_weight(shape=(input_shape[-1],),
                                 initializer=self.init,
                                 name='{}_W'.format(self.name),
                                 regularizer=self.W_regularizer,
                                 constraint=self.W_constraint)
        self.features_dim = input_shape[-1]

        if self.bias:
            self.b = self.add_weight(shape=(input_shape[1],),
                                     initializer='zero',
                                     name='{}_b'.format(self.name),
                                     regularizer=self.b_regularizer,
                                     constraint=self.b_constraint)
        else:
            self.b = None

        self.built = True

    def compute_mask(self, input, input_mask=None):
        return None

    def call(self, x, mask=None):
        features_dim = self.features_dim
        step_dim = self.step_dim

        eij = K.reshape(K.dot(K.reshape(x, (-1, features_dim)),
                        K.reshape(self.W, (features_dim, 1))), (-1, step_dim))

        if self.bias:
            eij += self.b

        eij = K.tanh(eij)

        a = K.exp(eij)
```

```
        if mask is not None:
            a *= K.cast(mask, K.floatx())

        a /= K.cast(K.sum(a, axis=1, keepdims=True) + K.epsilon(), K.floatx())

        a = K.expand_dims(a)
        weighted_input = x * a
        return K.sum(weighted_input, axis=1)

    def compute_output_shape(self, input_shape):
        return input_shape[0],  self.features_dim
```

In [ ]:

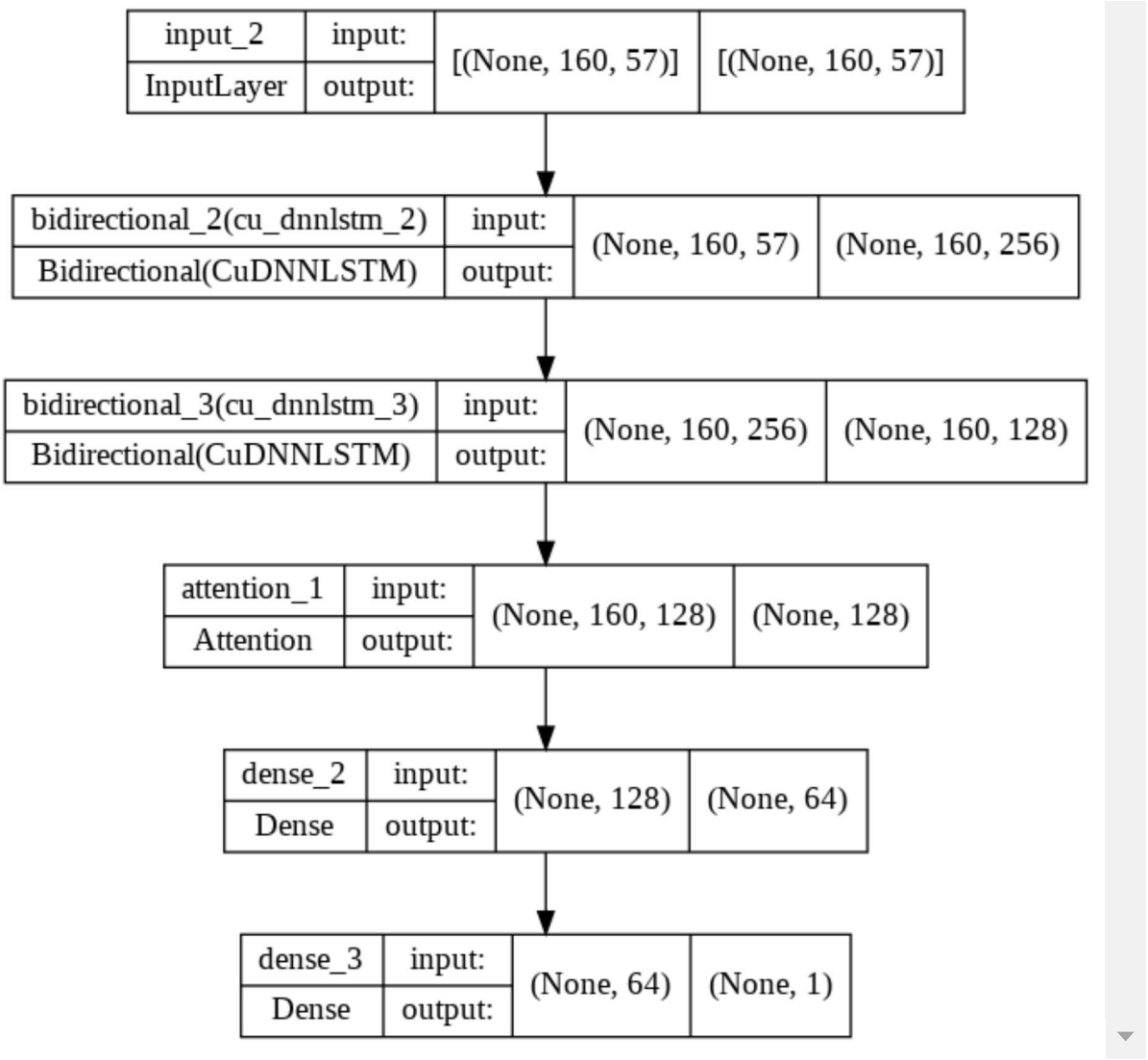
```
# This is NN LSTM Model creation
def model_lstm(input_shape):
    # The shape was explained above, must have this order
    inp = Input(shape=(input_shape[1], input_shape[2],))
    # This is the LSTM layer
    # Bidirecional implies that the 160 chunks are calculated in both ways, 0 to 159 and 15
    # although it appear that just 0 to 159 way matter, I have tested with and without, ana
    # 128 and 64 are the number of cells used, too many can overfit and too few can underfi
    x = Bidirectional(CuDNNLSTM(128, return_sequences=True))(inp)
#     x = Activation('relu')(x)
#     x = Dropout(0.25)(x)
#     x = BatchNormalization()(x)
    # The second LSTM can give more fire power to the model, but can overfit it too
    x = Bidirectional(CuDNNLSTM(64, return_sequences=True))(x)
#     x = Activation('relu')(x)
#     x = Dropout(0.25)(x)
#     x = BatchNormalization()(x)
    # Attention is a new tecnology that can be applyed to a Recurrent NN to give more meani
    # of the data, it helps more in longs chains of data. A normal RNN give all the respons
    # to the last cell. Google RNN Attention for more information :)
    #x = Dropout(0.2)(x)
    #x = BatchNormalization()(x)
    x = Attention(input_shape[1])(x)

    # A intermediate full connected (Dense) can help to deal with nonlinears outputs
    x = Dense(64, activation="relu")(x)
    # A binnary classification as this must finish with shape (1,)
    x = Dense(1, activation="sigmoid")(x)
    model = Model(inputs=inp, outputs=x)
    # Pay attention in the addition of matthews_correlation metric in the compilation, it i
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=[matthews_correlati

    return model
```

In [ ]:

```
X = X
y = y
# Number of folds for cross-validation
N_SPLITS = 5
# Perform stratified split of the training data.
# The indices of the data obtained after the split is returned.
splits = list(StratifiedKFold(n_splits=N_SPLITS, shuffle=True, random_state=2019).split(X,
input_shape = X[splits[0][0]].shape
model = model_lstm(input_shape)

print(model.summary())
plot_model(model, to_file='model.png', show_shapes=True, show_layer_names=True,
          expand_nested=True)
```

```
Model: "model_1"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 input_2 (InputLayer)        [(None, 160, 57)]         0

 bidirectional_2 (Bidirectio  (None, 160, 256)         191488
 nal)

 bidirectional_3 (Bidirectio  (None, 160, 128)         164864
 nal)

 attention_1 (Attention)     (None, 128)               288

 dense_2 (Dense)             (None, 64)                8256

 dense_3 (Dense)             (None, 1)                 65

=================================================================
Total params: 364,961
Trainable params: 364,961
Non-trainable params: 0
_________________________________________________________________
None
```

Out[36]:


input_2
InputLayer
input:
output:
[(None, 160, 57)]
[(None, 160, 57)]
bidirectional_2(cu_dnnlstm_2)
Bidirectional(CuDNNLSTM)
input:
output:
(None, 160, 57)
(None, 160, 256)
bidirectional_3(cu_dnnlstm_3)
Bidirectional(CuDNNLSTM)
input:
output:
(None, 160, 256)
(None, 160, 128)
attention_1
Attention
input:
output:
(None, 160, 128)
(None, 128)
dense_2
Dense
input:
output:
(None, 128)
(None, 64)
dense_3
Dense
input:
output:
(None, 64)
(None, 1)

In [ ]:

```
# First, create a set of indexes of the 5 folds
splits = list(StratifiedKFold(n_splits=N_SPLITS, shuffle=True, random_state=2019).split(X,
preds_val = []
y_val = []
# Then, iteract with each fold
# If you dont know, enumerate(['a', 'b', 'c']) returns [(0, 'a'), (1, 'b'), (2, 'c')]
for idx, (train_idx, val_idx) in enumerate(splits):
    K.clear_session() # I dont know what it do, but I imagine that it "clear session" :)
    print("Beginning fold {}".format(idx+1))
    # use the indexes to extract the folds in the train and validation data
    train_X, train_y, val_X, val_y = X[train_idx], y[train_idx], X[val_idx], y[val_idx]
    # instantiate the model for this fold
    model = model_lstm(train_X.shape)
    # This checkpoint helps to avoid overfitting. It just save the weights of the model if
    # validation matthews_correlation greater than the last one.
    ckpt = ModelCheckpoint('weights_{}.h5'.format(idx), save_best_only=True, save_weights_c
    # Train, train, train
    history=model.fit(train_X, train_y, batch_size=128, epochs=50, validation_data=[val_X,
    # loads the best weights saved by the checkpoint
    model.load_weights('weights_{}.h5'.format(idx))
    # Add the predictions of the validation to the list preds_val
    preds_val.append(model.predict(val_X, batch_size=512))
    # and the val true y
    y_val.append(val_y)

# concatenates all and prints the shape
preds_val = np.concatenate(preds_val)[...,0]
y_val = np.concatenate(y_val)
preds_val.shape, y_val.shape
```

```
Beginning fold 1
Train on 2323 samples, validate on 581 samples
Epoch 1/50
2323/2323 [==============================] - ETA: 0s - loss: 0.3408 - matt
hews_correlation: 0.0054
Epoch 1: val_matthews_correlation improved from -inf to 0.00000, saving mo
del to weights_0.h5
2323/2323 [==============================] - 2s 685us/sample - loss: 0.340
8 - matthews_correlation: 0.0054 - val_loss: 0.2394 - val_matthews_correla
tion: 0.0000e+00
Epoch 2/50
2323/2323 [==============================] - ETA: 0s - loss: 0.2354 - matt
hews_correlation: 0.0000e+00
Epoch 2: val_matthews_correlation did not improve from 0.00000
2323/2323 [==============================] - 1s 375us/sample - loss: 0.235
4 - matthews_correlation: 0.0000e+00 - val_loss: 0.2205 - val_matthews_cor
relation: 0.0000e+00
Epoch 3/50
2304/2323 [============================>.] - ETA: 0s - loss: 0.2242 - matt
```

In [ ]:

```
# The output of this kernel must be binary (0 or 1), but the output of the NN Model is floa
# So, find the best threshold to convert float to binary is crucial to the result
# this piece of code is a function that evaluates all the possible thresholds from 0 to 1 b
def threshold_search(y_true, y_proba):
    best_threshold = 0
    best_score = 0
    for threshold in tqdm([i * 0.01 for i in range(100)]):
        score = K.eval(matthews_correlation(y_true.astype(np.float64), (y_proba > threshold
        if score > best_score:
            best_threshold = threshold
            best_score = score
    search_result = {'threshold': best_threshold, 'matthews_correlation': best_score}
    return search_result
```

In [ ]:

```
best_threshold = threshold_search(y_val, preds_val)['threshold']
```

```
100%|██████████| 100/100 [00:12<00:00,  8.07it/s]
```

In [ ]:

```
%%time
# First we daclarete a series of parameters to initiate the loading of the main data
# it is too large, it is impossible to load in one time, so we are doing it in dividing in
first_sig = df_metadata_test.index[0]
n_parts = 30
max_line = len(df_metadata_test)
part_size = int(max_line / n_parts)
last_part = max_line % n_parts
print(first_sig, n_parts, max_line, part_size, last_part, n_parts * part_size + last_part)
# Here we create a list of lists with start index and end index for each of the 10 parts an
start_end = [[x, x+part_size] for x in range(first_sig, max_line + first_sig, part_size)]
start_end = start_end[:-1] + [[start_end[-1][0], start_end[-1][0] + last_part]]
print(start_end)
X_test = []
# now, very like we did above with the train data, we convert the test data part by part
# transforming the 3 phases 800000 measurement in matrix (160,57)
for start, end in start_end:
    signal_data = pq.read_pandas('/content/test.parquet', columns=[str(i) for i in range(st
    for i in tqdm(signal_data.columns):
        id_measurement, phase = df_metadata_test.loc[int(i)]
        subset_test_col = signal_data[i]
        subset_trans = transform_signal(subset_test_col)
        X_test.append([i, id_measurement, phase, subset_trans])
```

```
8712 30 20337 677 27 20337
[[8712, 9389], [9389, 10066], [10066, 10743], [10743, 11420], [11420, 1209
7], [12097, 12774], [12774, 13451], [13451, 14128], [14128, 14805], [14805,
15482], [15482, 16159], [16159, 16836], [16836, 17513], [17513, 18190], [181
90, 18867], [18867, 19544], [19544, 20221], [20221, 20898], [20898, 21575],
[21575, 22252], [22252, 22929], [22929, 23606], [23606, 24283], [24283, 2496
0], [24960, 25637], [25637, 26314], [26314, 26991], [26991, 27668], [27668,
28345], [28345, 29022], [29022, 29049]]
```

```
100%|██████████| 677/677 [00:46<00:00, 14.47it/s]
100%|██████████| 677/677 [00:46<00:00, 14.54it/s]
100%|██████████| 677/677 [00:46<00:00, 14.48it/s]
100%|██████████| 677/677 [00:47<00:00, 14.34it/s]
100%|██████████| 677/677 [00:46<00:00, 14.44it/s]
100%|██████████| 677/677 [00:47<00:00, 14.37it/s]
100%|██████████| 677/677 [00:47<00:00, 14.37it/s]
100%|██████████| 677/677 [00:46<00:00, 14.49it/s]
100%|██████████| 677/677 [00:46<00:00, 14.48it/s]
100%|██████████| 677/677 [00:46<00:00, 14.41it/s]
100%|██████████| 677/677 [00:47<00:00, 14.38it/s]
100%|██████████| 677/677 [00:49<00:00, 13.73it/s]
100%|██████████| 677/677 [00:47<00:00, 14.29it/s]
100%|██████████| 677/677 [00:48<00:00, 13.90it/s]
100%|██████████| 677/677 [00:47<00:00, 14.36it/s]
100%|██████████| 677/677 [00:47<00:00, 14.31it/s]
100%|██████████| 677/677 [00:47<00:00, 14.31it/s]
100%|██████████| 677/677 [00:47<00:00, 14.23it/s]
100%|██████████| 677/677 [00:47<00:00, 14.35it/s]
100%|██████████| 677/677 [00:47<00:00, 14.32it/s]
100%|██████████| 677/677 [00:47<00:00, 14.33it/s]
100%|██████████| 677/677 [00:47<00:00, 14.36it/s]
100%|██████████| 677/677 [00:47<00:00, 14.34it/s]
100%|██████████| 677/677 [00:46<00:00, 14.52it/s]
100%|██████████| 677/677 [00:47<00:00, 14.29it/s]
100%|██████████| 677/677 [00:47<00:00, 14.26it/s]
```

```
100%|██████████| 677/677 [00:47<00:00, 14.31it/s]
100%|██████████| 677/677 [00:47<00:00, 14.26it/s]
100%|██████████| 677/677 [00:47<00:00, 14.31it/s]
100%|██████████| 677/677 [00:47<00:00, 14.31it/s]
100%|██████████| 27/27 [00:01<00:00, 14.35it/s]

CPU times: user 25min 29s, sys: 1min 47s, total: 27min 16s
Wall time: 25min 55s
```

In [ ]:

```
X_test_input = np.asarray([np.concatenate([X_test[i][3],X_test[i+1][3], X_test[i+2][3]], ax
np.save("X_test.npy",X_test_input)
X_test_input.shape
```

Out[29]:

```
(6779, 160, 57)
```

In [ ]:

```
X_test=np.load("X_test.npy")
```

In [ ]:

```
preds_test = []
for i in range(N_SPLITS):
    model.load_weights('weights_{}.h5'.format(i))
    pred = model.predict(X_test_input, batch_size=300, verbose=1)
    pred_3 = []
    for pred_scalar in pred:
        for i in range(3):
            pred_3.append(pred_scalar)
    preds_test.append(pred_3)
```

In [ ]:

```
preds_test = (np.squeeze(np.mean(preds_test, axis=0)) > best_threshold).astype(np.int)
preds_test.shape
```

Out[42]:

```
(20337,)
```

In [ ]:

```
submission = pd.read_csv('sample_submission.csv')
print(len(submission))
submission.head()
```

```
20337
```

Out[43]:

	signal_id	target
0	8712	0
1	8713	0
2	8714	0
3	8715	0
4	8716	0

In [ ]:

```
submission['target'] = preds_test
submission.to_csv('submission.csv', index=False)
submission.head()
```

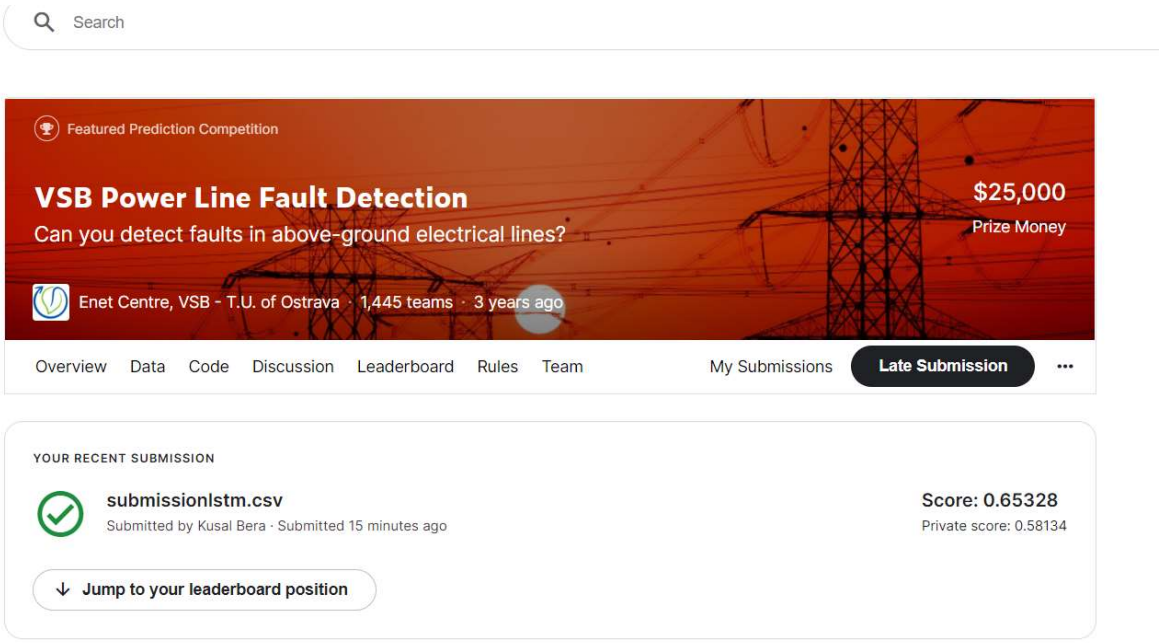
Out[44]:

	signal_id	target
0	8712	0
1	8713	0
2	8714	0
3	8715	0
4	8716	0

In [2]:

```
from IPython.display import Image
Image('/content/kaggle score1.png')
```

Out[2]:



In [ ]: