VSB Power Line Fault Detection

1. Business/Real-world Problem

1.1. What is Partial Discharge?

Here we deal with medium voltage overhead powerlines which are spread over hundreds of miles making manual fault detection almost impossible

These lines on some ocassions get damaged by either a tree branch or due to a flaw in the insulator.These damages lead to a power outage gradually over passage of time.This phenomenon is partial discharge.

It's textbook definition is, an electrical discharge that does not bridge the electrodes between an insulation system completely

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

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

1.2. Problem Statement

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.

1.3. Real-world/Business objectives and constraints.

1.Minimize binary-class error
2.probability estimates
3.There's no time limitation as partial discharge faults do damage overtime and not immediately so limit can be in hours
4.Detecting the partial discharge early can be useful financially

2. Machine Learning Problem

2.1. Data Overview

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

In total 4 files are given in which 2 correspond to train data and the rest correspond to test data
1.File containing signal data
2.File Containing meta data

Each signal contains 800,000 points and in total data of 8712 signals were given for training and 20337 signals were given for testing in the form of parquet data

Meta data consists of the phase of the signal and the target label
0-if partial discharge is not there
1-if partial discharge is present

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

2.2.1. Type of Machine Learning Problem

There are 2 different classes of malware that we need to classify a given a data point => Binary class classification problem

2.2.2. Performance Metric

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

Metric(s):
*Matthews correlation coefficient(MCC)
*Confusion matrix

2.2.3. Machine Learing 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 probabilites => Metric is Matthews correlation coefficient.
* Some Latency constraints.

3. Exploratory Data Analysis

Importing required libraries

In [1]:

```
#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')
```

Examining Metadata

Importing meta_data

In [4]:

```
meta_data = pd.read_csv('metadata_train.csv')
```

In [5]:

```
meta_data
```

Out[5]:

	signal_id	id_measurement	phase	target
0	0	0	0	0
1	1	0	1	0
2	2	0	2	0
3	3	1	0	1
4	4	1	1	1
...	...	...	...	...
8707	8707	2902	1	0
8708	8708	2902	2	0
8709	8709	2903	0	0
8710	8710	2903	1	0
8711	8711	2903	2	0

8712 rows × 4 columns

8712 rows
Consists of 4 columns:
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

Checking for null values

In [7]:

```
meta_data.isnull().values.any()
```

Out[7]:

```
False
```

no null values found

Examinig signal_id and id_measurement

In [8]:

```
list(range(0,len(meta_data))) == list(meta_data['signal_id'])
```

Out[8]:

```
True
```

So signal_id is basically a unique id for each signal or row

In [9]:

```
#checking if each id measurement has 3 phases

#iterating over the data frame with a step of 3
for i in range(0,len(meta_data),3):
    temp = []
    #cheking if the 3 values are 0,1,2
    for i in range(3):
        temp.append(meta_data.loc[i]['phase'])
    #if error break loop
    if(temp!=[0,1,2]):
        print('error')
        break
print('success')
```

```
success
```

As no error is printed,for each id_measurement there are 3 phases 0,1,2

Examining phase and target

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

In [10]:

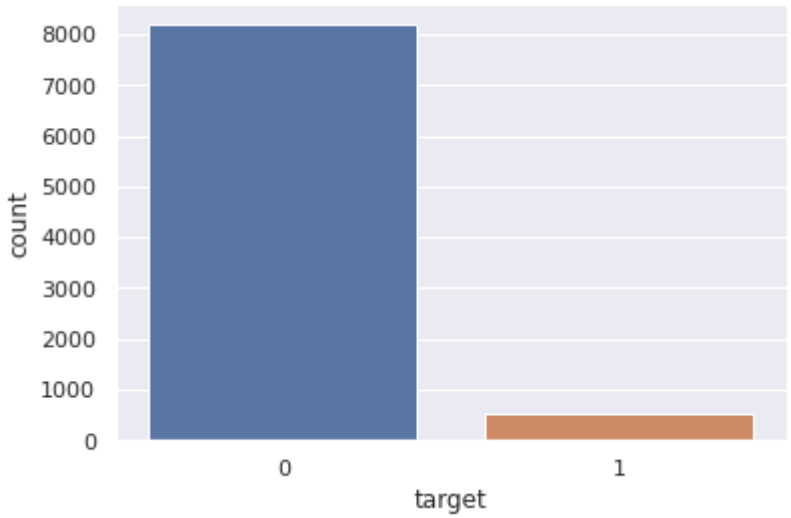
```
sns.set(style="darkgrid")
```

In [11]:

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

Out[11]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f2df8968190>
```



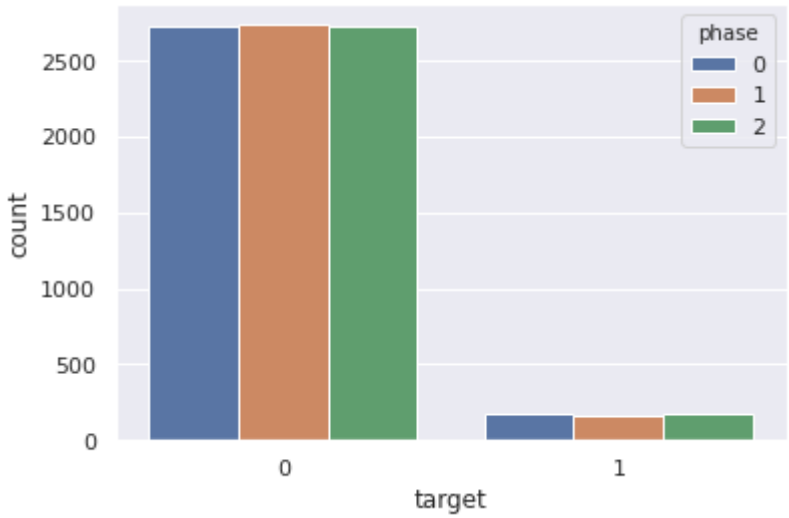
It can be observed that the data is highly imbalanced,let's try looking at each phase

In [12]:

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

Out[12]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f2de6a5ed10>
```



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

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

In [13]:

```
#iterating through each signal(including 3 phases)
for i in range(0,len(meta_data),3):
    temp1 = meta_data.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):
        temp2 = meta_data.loc[i+j]['target']
        #if different target value
        if temp1!=temp2:
            print(meta_data.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 meta_data

As the data is large first lets look at the first 9 signals

In [2]:

```
signal_data =pq.read_pandas('train.parquet',columns=[str(i) for i in range(9)]).to_
```

Analysis on signal data

In [17]:

```
signal_data
```

Out[17]:

	0	1	2	3	4	5	6	7	8
0	18	1	-19	-16	-5	19	-15	15	-1
1	18	0	-19	-17	-6	19	-17	16	0
2	17	-1	-20	-17	-6	19	-17	15	-3
3	18	1	-19	-16	-5	20	-16	16	0
4	18	0	-19	-16	-5	20	-17	16	-2
...	...	...	...	...	...	...	...	...	...
799995	19	2	-18	-15	-4	21	-16	16	-1
799996	19	1	-19	-15	-4	20	-17	15	-3
799997	17	0	-19	-15	-4	21	-16	14	-2
799998	19	1	-18	-14	-3	22	-16	17	-1
799999	17	0	-19	-14	-4	21	-17	14	-4

800000 rows × 9 columns

It can be observed that each signal consists of 800,000 points and column number here corresponds to signal id in meta_data

let's observe one phase of a three phase signal

In [18]:

```
signal_data['0'].describe()
```

Out[18]:

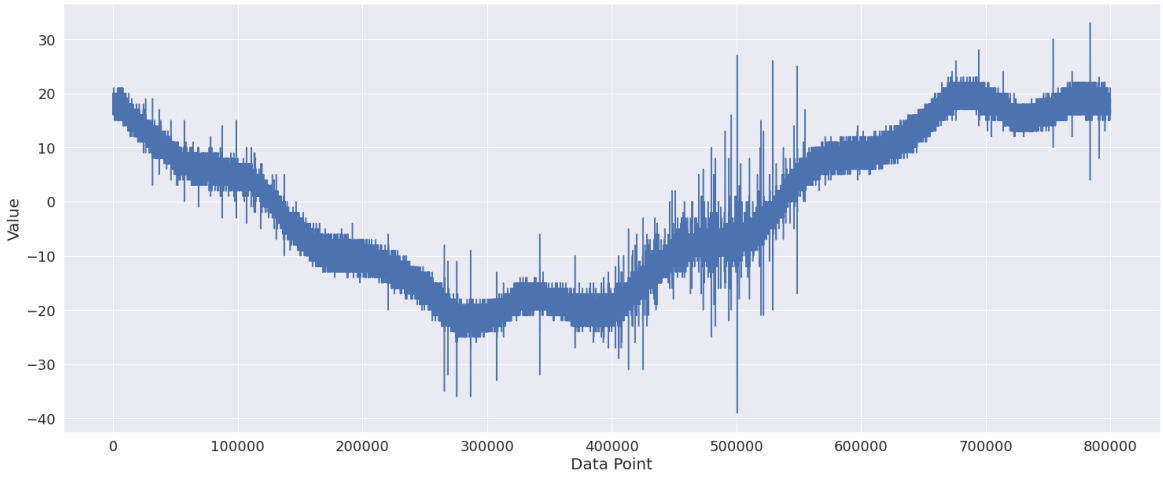
```
count    800000.000000
mean         -0.960271
std          13.870733
min         -39.000000
25%         -13.000000
50%          -1.000000
75%          11.000000
max          33.000000
Name: 0, dtype: float64
```

In [19]:

```
signal_data['0'].plot(figsize=(25,10))
plt.xlabel('Data Point',fontsize=20)
plt.ylabel('Value',fontsize=20)
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)
```

Out[19]:

```
(array([-50., -40., -30., -20., -10.,   0.,  10.,  20.,  30.,  40.]),
 <a list of 10 Text major ticklabel objects>)
```



values are between -39 and +33.
50% of the points are less than -1 and the mean is -0.9.
Some values are abnormally high and some values are abnormally low

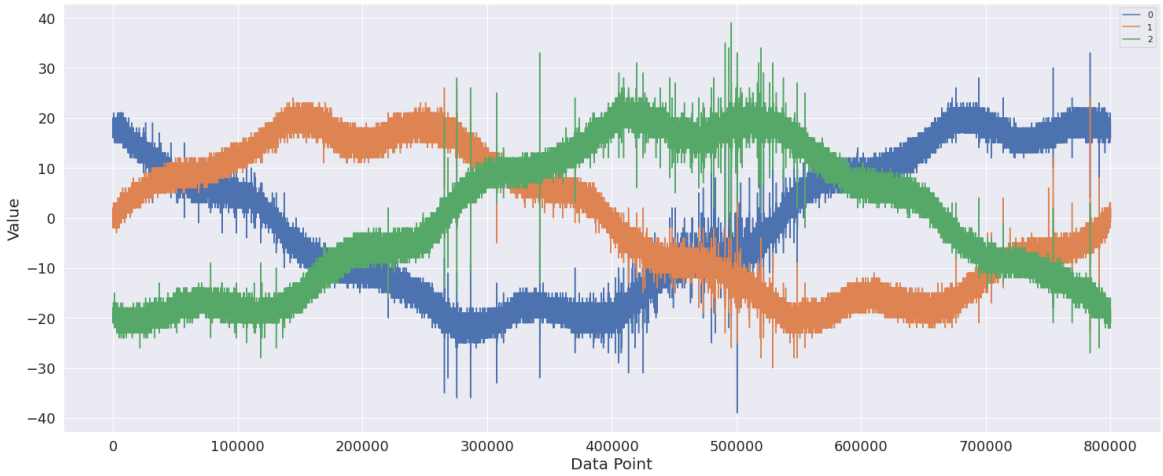
let's look at all 3 phases combined

In [20]:

```
signal_data.loc[:,'0':'2'].plot(figsize=(25,10))
plt.xlabel('Data Point',fontsize=20)
plt.ylabel('Value',fontsize=20)
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)
```

Out[20]:

```
(array([-50., -40., -30., -20., -10.,   0.,  10.,  20.,  30.,  40.,
50.]),
 <a list of 11 Text major ticklabel objects>)
```



let's check a signal which has partial discharge

In [21]:

```
meta_data.loc[meta_data['target']==1]
```

Out[21]:

	signal_id	id_measurement	phase	target
3	3	1	0	1
4	4	1	1	1
5	5	1	2	1
201	201	67	0	1
202	202	67	1	1
...	...	...	...	...
8483	8483	2827	2	1
8568	8568	2856	0	1
8569	8569	2856	1	1
8570	8570	2856	2	1
8630	8630	2876	2	1

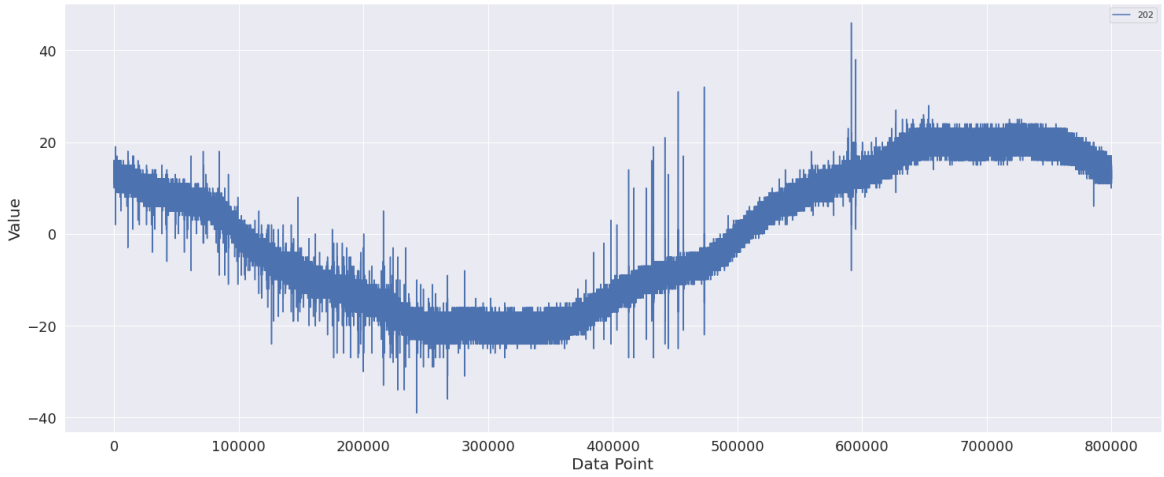
525 rows × 4 columns

In [23]:

```
pq.read_pandas('../input/vsb-power-line-fault-detection/train.parquet',columns=[str
plt.xlabel('Data Point',fontsize=20)
plt.ylabel('Value',fontsize=20)
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)
```

Out[23]:

```
(array([-60., -40., -20.,   0.,  20.,  40.,  60.]),
 <a list of 7 Text major ticklabel objects>)
```



visually there's not much difference between a signal which has partial disharge and which doesn't

magnifying the graph and looking at 150 points

In [ ]:

```
signal_data['0'][200:350].plot(figsize=(25,10))
plt.xlabel('Data Point',fontsize=20)
plt.ylabel('Value',fontsize=20)
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)
```

Taking the mean of signals corresponding to 1 and 0 target values both cobined and for each phase

In [ ]:

```
phase_sums_0 = [[0]*len(signal_data['0'])]*3
phase_sums_1 = [[0]*len(signal_data['0'])]*3
```

In [ ]:

```
#iterating through each signal
for i in tqdm(range(len(meta_data))):
    signal = pq.read_pandas('vsb-power-line-fault-detection/train.parquet',columns=

    #phase of the current signal
    phase = meta_data.loc[meta_data['signal_id'] == i]['phase'][i]

    #target value of current signal
    target = meta_data.loc[meta_data['signal_id'] == i]['target'][i]

    #storing the min,max of current signal and finding
    #the sums of all corresponding values for calculating mean
    if target == 0:
        phase_sums_0[phase] += signal
    elif target == 1:
        phase_sums_1[phase] += signal
```

Plotting the means of each phases

In [ ]:

```
mean_df = pd.DataFrame(columns=['ps0_0','ps0_1','ps0_2','ps1_0','ps1_1','ps1_2'])
```

In [ ]:

```
mean_df['ps0_0'] = phase_sums_0[0]/len(meta_data[(meta_data['phase']==0 )& (meta_da
mean_df['ps0_1'] = phase_sums_0[1]/len(meta_data[(meta_data['phase']==1 )& (meta_da
mean_df['ps0_2'] = phase_sums_0[2]/len(meta_data[(meta_data['phase']==2 )& (meta_da

mean_df['ps1_0'] = phase_sums_1[0]/len(meta_data[(meta_data['phase']==0 )& (meta_da
mean_df['ps1_1'] = phase_sums_1[1]/len(meta_data[(meta_data['phase']==1 )& (meta_da
mean_df['ps1_2'] = phase_sums_1[2]/len(meta_data[(meta_data['phase']==2 )& (meta_da
```

In [ ]:

```
#mean_df.to_csv('signal_means.csv',index=False)
```

In [14]:

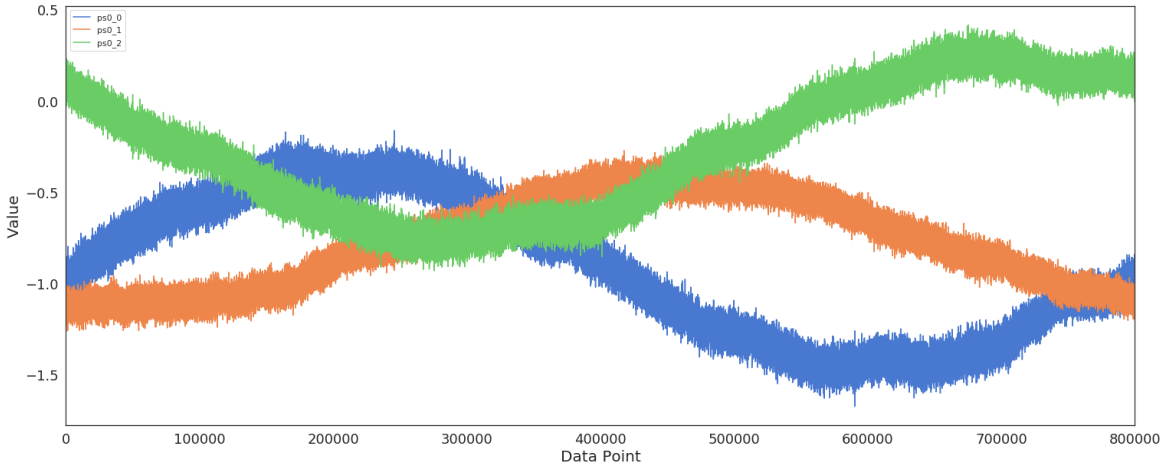
```
mean_df = pd.read_csv('signal_means.csv',index_col=False)
```

In [15]:

```
mean_df.loc[:,'ps0_0':'ps0_2'].plot(figsize=(25,10))
plt.xlabel('Data Point',fontsize=20)
plt.ylabel('Value',fontsize=20)
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)
```

Out[15]:

```
(array([-2. , -1.5, -1. , -0.5,  0. ,  0.5,  1. ]),
 <a list of 7 Text yticklabel objects>)
```

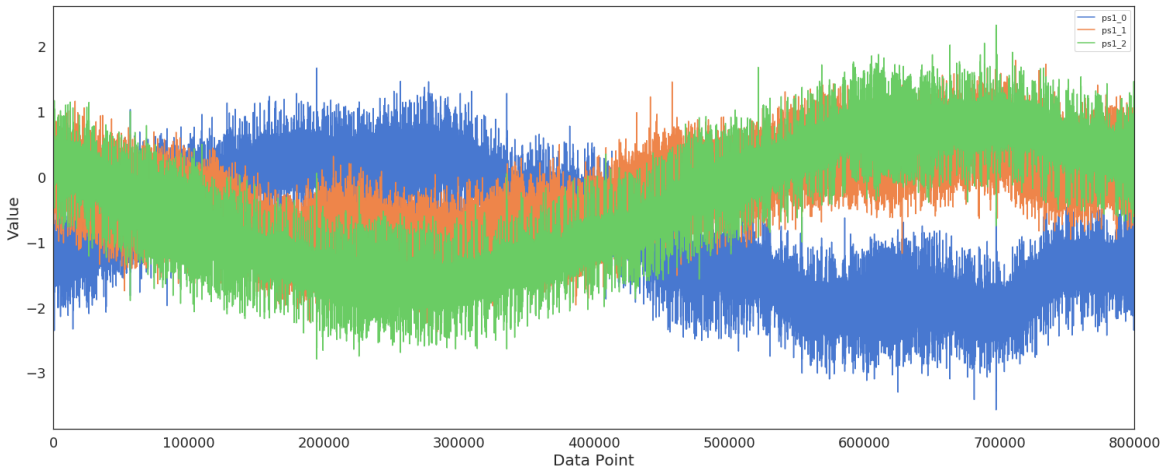


In [16]:

```
mean_df.loc[:,'ps1_0':'ps1_2'].plot(figsize=(25,10))
plt.xlabel('Data Point',fontsize=20)
plt.ylabel('Value',fontsize=20)
plt.xticks(fontsize=18)
plt.yticks(fontsize=18)
```

Out[16]:

```
(array([-4., -3., -2., -1.,  0.,  1.,  2.,  3.]),
 <a list of 8 Text yticklabel objects>)
```



A lot of difference can be found now between the signals with partial discharge(target=1) and signals without(target=0).The following can be observed:
1.In case of target value 0(no partial discharge) the values lie within an interval of 0.2 or 0.3 but in case of target value 1 the values lie in an interval of approximately 1.5
2.In the first plot the signals with different phases are seperated with a certain distance but in the second plot the signals with different phases are almost overlapping.
The above 2 differences can be added as features for our model

Extracting statistical features from our dataset

In [ ]:

```
stats_df = pd.DataFrame(columns=['signal_id','mean','std','min','max','band_width',
```

In [ ]:

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

    stats = []
    signal =  pq.read_pandas('vsb-power-line-fault-detection/train.parquet',columns

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

    #mean
    stats.append(signal.mean())

    #std
    stats.append(signal.std())

    #min
    stats.append(signal.min())

    #max
    stats.append(signal.max())

    #bandwidth
    band_width = [signal.mean()+signal.std(),signal.mean()-signal.std()]
    stats.append(band_width)

    #percentiles of signal
    stats.append(np.percentile(signal, [0, 1, 25, 50, 75, 99, 100]))

    #target
    stats.append(row['target'])



    stats_df.loc[len(stats_df)] = stats
```

In [ ]:

```
#stats_df.to_csv('stats_data.csv',index=False)
```

In [2]:

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

In [3]:

```
stats_df.head(2)
```

Out[3]:

	signal_id	mean	std	min	max	band_width	percentiles	target
0	0	-0.960271	13.870724	-39	33	[12.910453059725986, -14.830995559725986]	[-39. -22. -13. -1. 11. 20. 33.]	0
1	1	-0.194125	13.037134	-30	26	[12.843009385453541, -13.23125938545354]	[-30. -21. -12. 0. 12. 20. 26.]	0

Mean

In [4]:

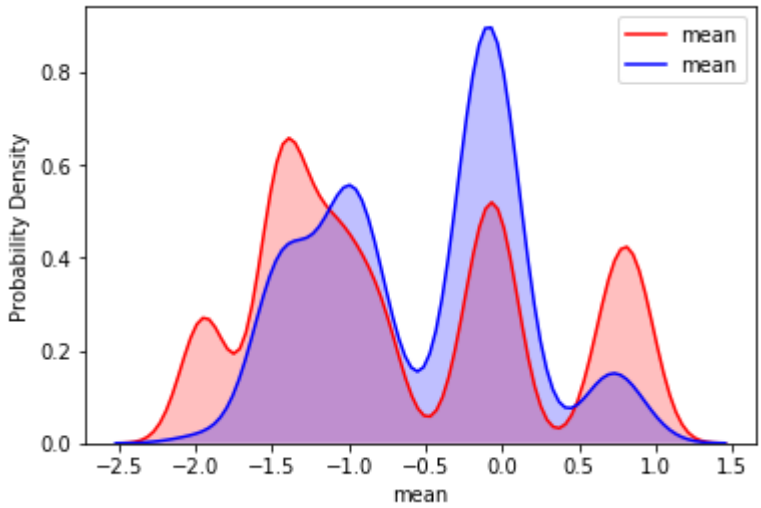
```
sns.kdeplot(stats_df.loc[(stats_df['target']==0),
            'mean'], color='r', shade=True, Label='target 0')

sns.kdeplot(stats_df.loc[(stats_df['target']==1),
            'mean'], color='b', shade=True, Label='target 1')

plt.xlabel('mean')
plt.ylabel('Probability Density')
```

Out[4]:

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



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 [5]:

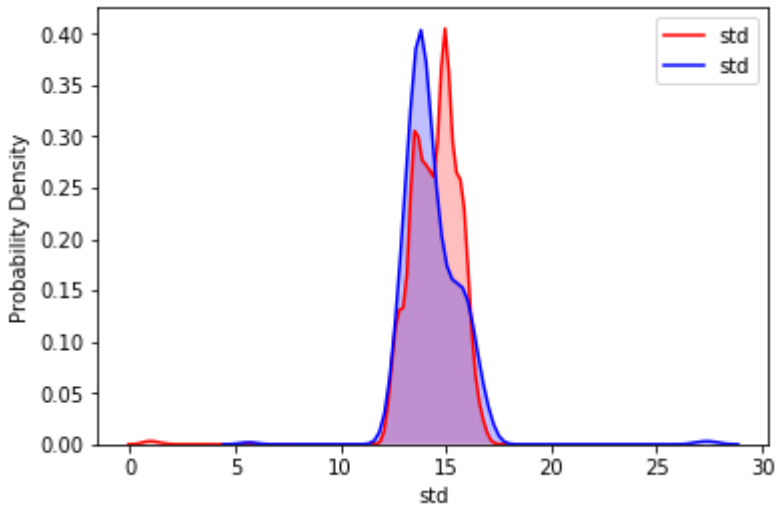
```
sns.kdeplot(stats_df.loc[(stats_df['target']==0),
            'std'], color='r', shade=True, Label='target 0')

sns.kdeplot(stats_df.loc[(stats_df['target']==1),
            'std'], color='b', shade=True, Label='target 1')

plt.xlabel('std')
plt.ylabel('Probability Density')
```

Out[5]:

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



It is more probable that std is greater than 15 if there is no partial discharge.From both mean and std it can be said that the signal is spread more if there is no partial discharge

Minimum of signal

In [6]:

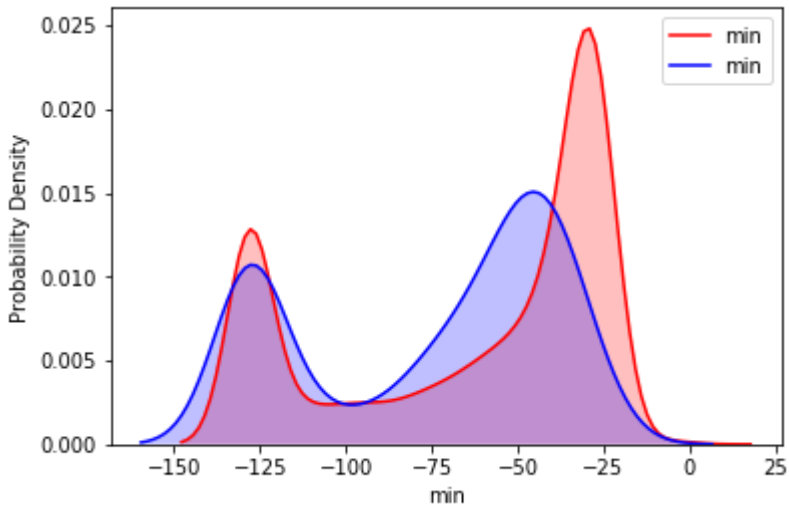
```
sns.kdeplot(stats_df.loc[(stats_df['target']==0),
            'min'], color='r', shade=True, Label='target 0')

sns.kdeplot(stats_df.loc[(stats_df['target']==1),
            'min'], color='b', shade=True, Label='target 1')

plt.xlabel('min')
plt.ylabel('Probability Density')
```

Out[6]:

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



if the minimum of signal is greater than -50 there's more probability of no partial discharge

Maximum of signal

In [7]:

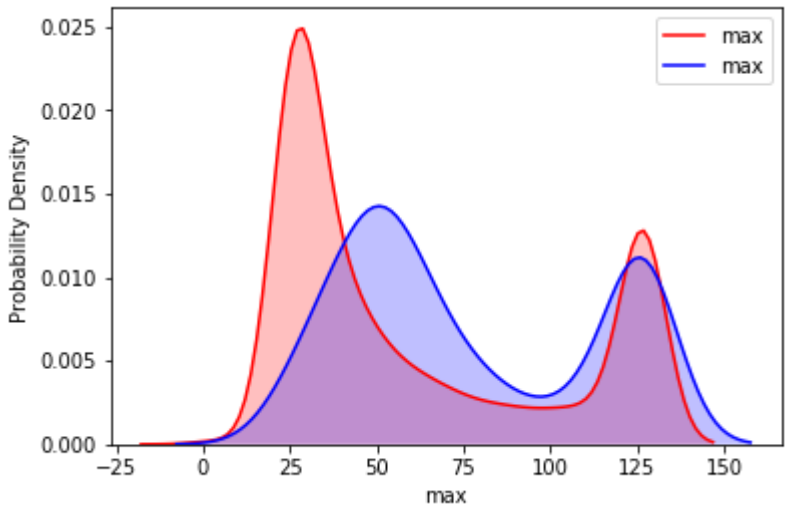
```
sns.kdeplot(stats_df.loc[(stats_df['target']==0),
            'max'], color='r', shade=True, Label='target 0')

sns.kdeplot(stats_df.loc[(stats_df['target']==1),
            'max'], color='b', shade=True, Label='target 1')

plt.xlabel('max')
plt.ylabel('Probability Density')
```

Out[7]:

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



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

Bandwidth of signal

In [8]:

```
bw1 = []
bw2 = []
#extracting bandwiths from stats_df
for index,row in stats_df.iterrows():
    bw = ast.literal_eval(row['band_width'])
    bw1.append(bw[0])
    bw2.append(bw[1])
stats_df['band_width_1'] = bw1
stats_df['band_width_2'] = bw2
```

In [9]:

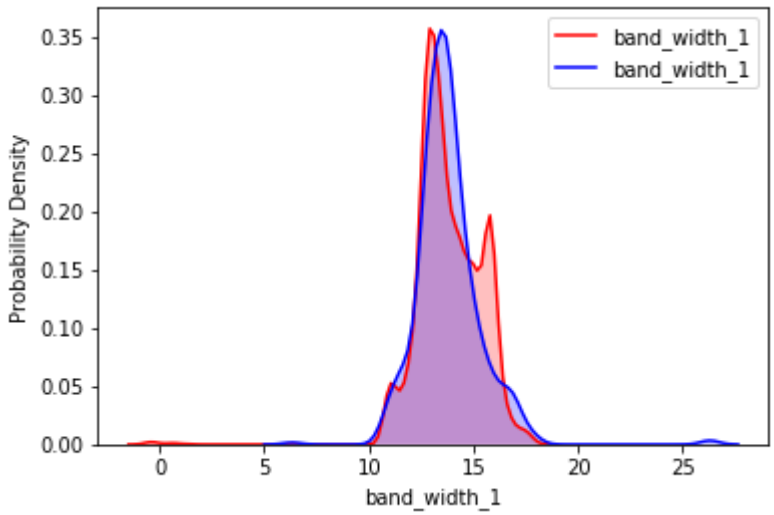
```
sns.kdeplot(stats_df.loc[(stats_df['target']==0),
            'band_width_1'], color='r', shade=True, Label='target 0')

sns.kdeplot(stats_df.loc[(stats_df['target']==1),
            'band_width_1'], color='b', shade=True, Label='target 1')

plt.xlabel('band_width_1')
plt.ylabel('Probability Density')
```

Out[9]:

```
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 [10]:

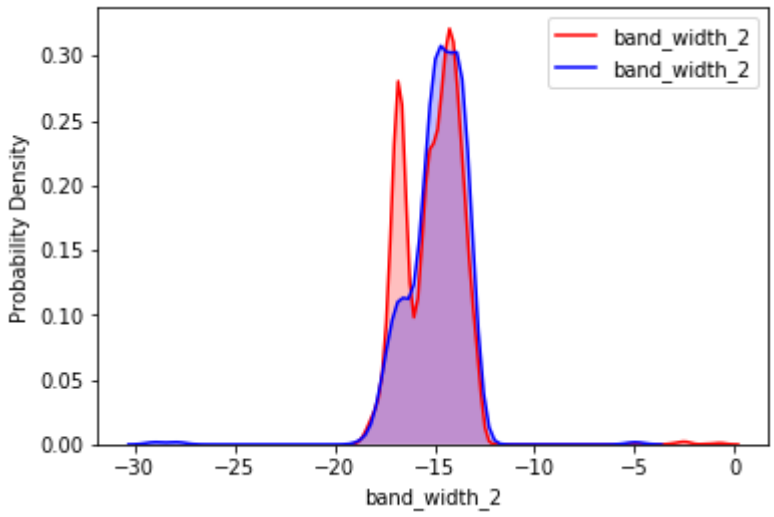
```
sns.kdeplot(stats_df.loc[(stats_df['target']==0),
            'band_width_2'], color='r', shade=True, Label='target 0')

sns.kdeplot(stats_df.loc[(stats_df['target']==1),
            'band_width_2'], color='b', shade=True, Label='target 1')

plt.xlabel('band_width_2')
plt.ylabel('Probability Density')
```

Out[10]:

```
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

Percentiles

In [11]:

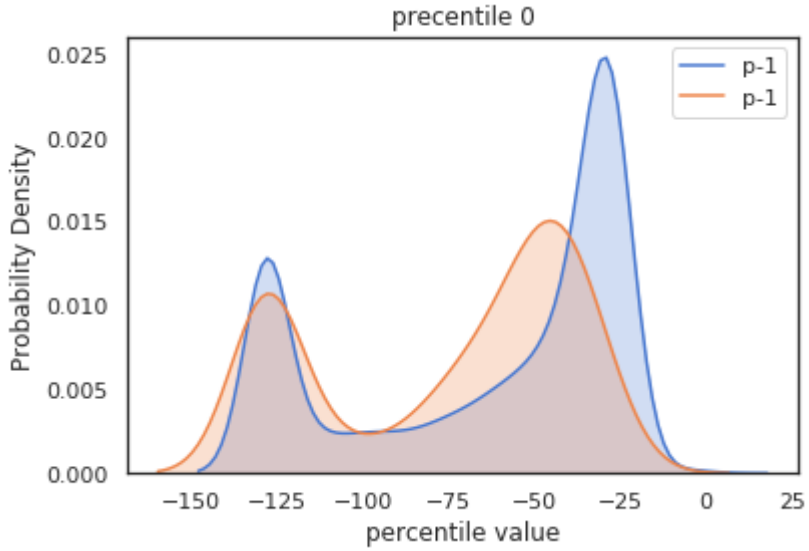
```
percentiles = pd.DataFrame(columns=['p-1','p-2','p-3','p-4','p-5','p-6','p-7','targ
```

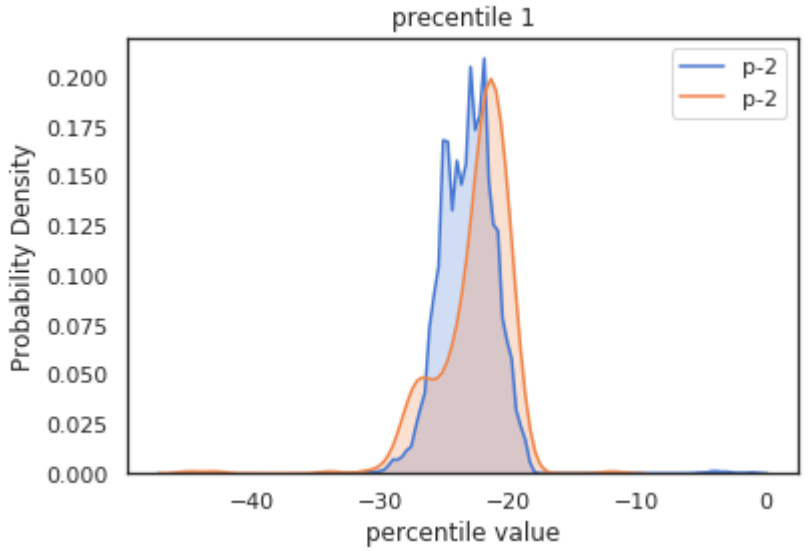
In [12]:

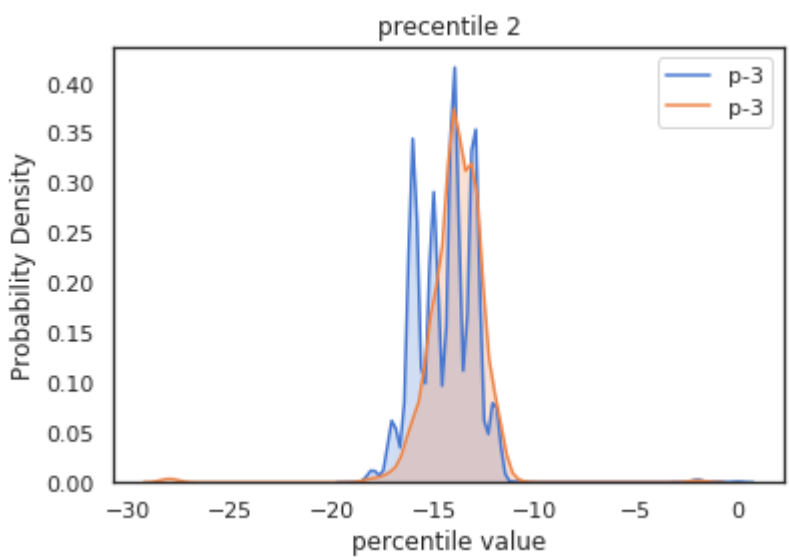
```
#extracting percentiles from stats _df
for index,row in stats_df.iterrows():
    pers = row['percentiles'][1:-1].split()
    pers = [float(i) for i in pers]
    pers.append(row['target'])
    percentiles.loc[len(percentiles)] = pers
```

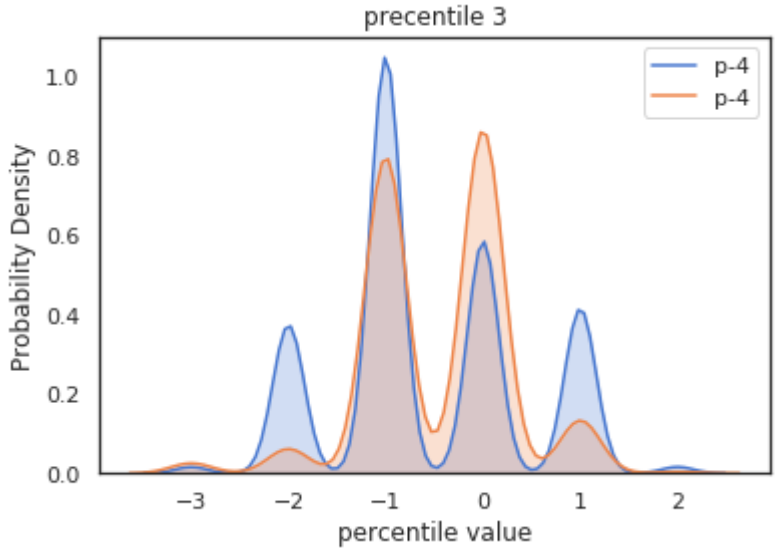
In [13]:

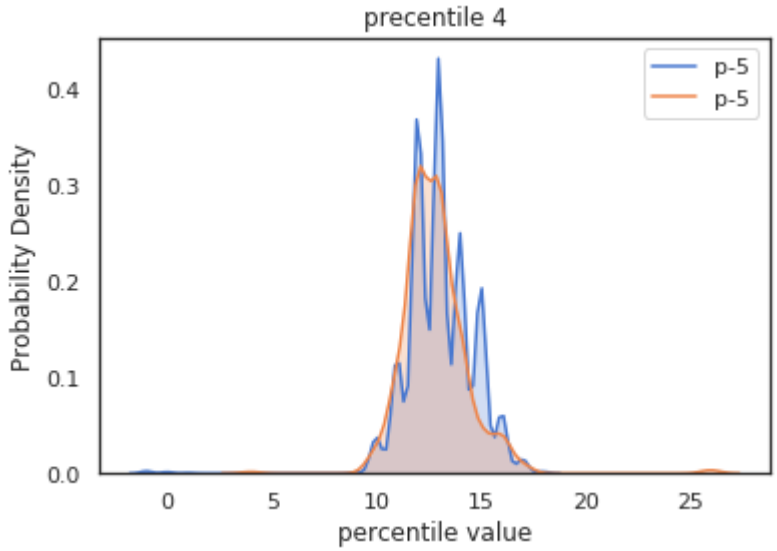
```
#plotting pdf of each percentile
for i in range(7):
    sns.set(style="white", palette="muted", color_codes=True)
    sns.kdeplot(percentiles.loc[(stats_df['target']==0),
            'p-'+str(i+1)], shade=True, Label='target 0')
    sns.kdeplot(percentiles.loc[(stats_df['target']==1),
            'p-'+str(i+1)], shade=True, Label='target 1')
    plt.title('precentile '+str(i))
    plt.xlabel('percentile value')
    plt.ylabel('Probability Density')
    plt.show()
```

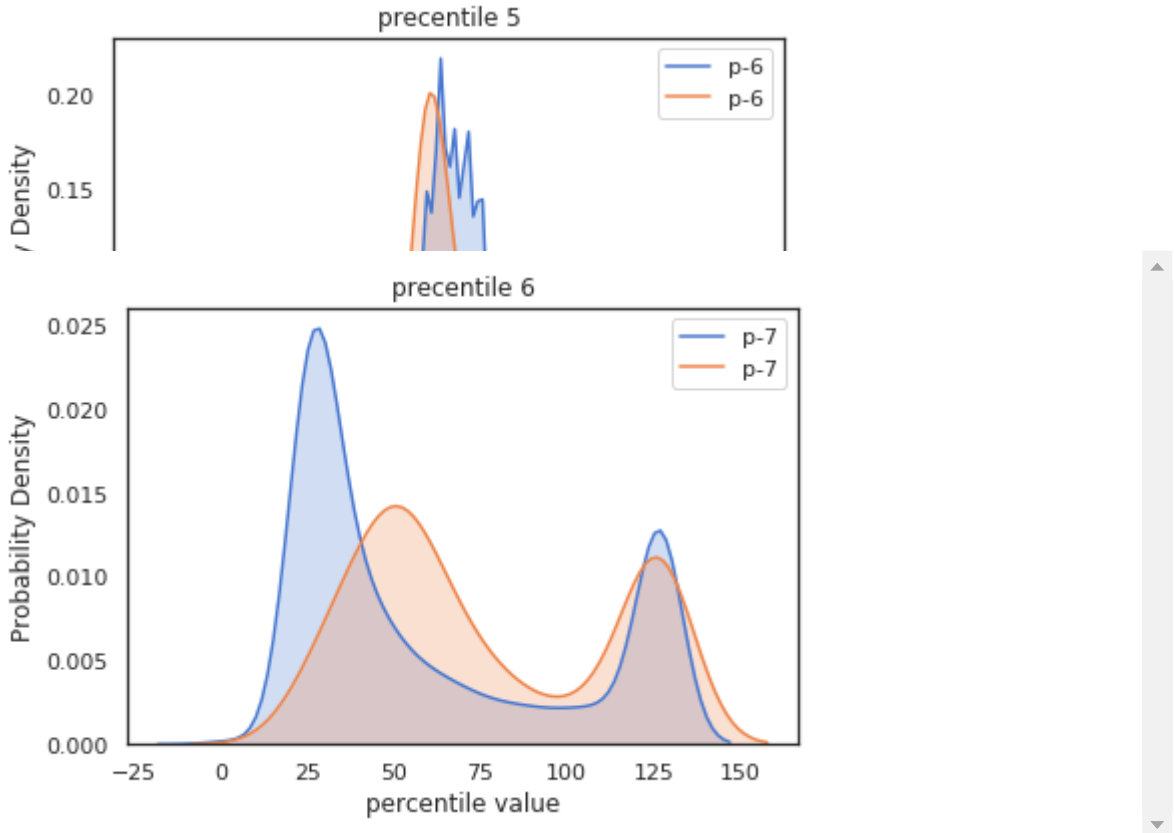





precentile 2
p-3
p-3
Probability Density
0.40
0.35
0.30
0.25
0.20
0.15
0.10
0.05
0.00
−30
−25
−20
−15
−10
−5
0
percentile value


precentile 3
p-4
p-4
Probability Density
1.0
0.8
0.6
0.4
0.2
0.0
−3
−2
−1
0
1
2
percentile value


precentile 4
p-5
p-5
Probability Density
0.4
0.3
0.2
0.1
0.0
0
5
10
15
20
25
percentile value



The difference between the curves can be observed clearly in the graphs of percentile 0 and percentile 99.In the other graphs the difference between the pdfs is not that clear but there is difference between the probability densities at certain mean ranges

Feature Engineering

- Refernce: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 [4]:

```
#reading the signal data
raw_signal_data = pq.read_pandas('../input/vsb-power-line-fault-detection/train.par

meta_train = np.array(meta_data)

print(raw_signal_data.shape)
```

```
(8712, 800000)
```

In [2]:

```
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)

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

    return res
```

In [ ]:

```
spectra = compute_spectra(raw_signal_data)
```

In [8]:

```
print(spectra['mean'].shape)
print(spectra['percentile'].shape)
```

```
(8712, 800)
(8712, 800, 6)
```

In [9]:

```
del raw_signal_data
```

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.

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

In [3]:

```
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_spec3 = np.empty((n_triplet, length, 3))
    for i_triplet in range(n_triplet):
        max_spec3[i_triplet, :, 0] = spec[3*i_triplet, :, 0] # phase 0
        max_spec3[i_triplet, :, 1] = spec[3*i_triplet + 1, :, 0] # phase 1
        max_spec3[i_triplet, :, 2] = spec[3*i_triplet + 2, :, 0] # phase 2

    x = tf.compat.v1.placeholder(tf.float32, [None, length, 3]) # input spectra bef
    # 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_spec3})

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

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

    return d
```

In [14]:

```
peaks = max_windowed(spectra['percentile'])
```

In [15]:

```
peaks['ibegin'].shape
```

Out[15]:

```
(2904,)
```

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 [4]:

```
def compute_spectra_features(spectra,peaks):


    percentiles = spectra['percentile']

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

    n_signals = n//3

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

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


    width = 150

    n_perc_features = 3

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

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


    for i_signal in range(n_signals):

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

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

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

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

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

    return spectra_features
```

In [ ]:

```
spectra_features = compute_spectra_features(spectra, peaks)
```

In [72]:

```
spectra_features.shape
```

Out[72]:

```
(2904, 57)
```

In [ ]:

```
#np.save('trainfeatures/train_spectra_features',spectra_features)
```

Fast Fourier Transform

signal processing reference:http://ataspinar.com/2018/04/04/machine-learning-with-signal-processing-techniques/ (http://ataspinar.com/2018/04/04/machine-learning-with-signal-processing-techniques/)

used to convert a signal from time domain to frequency domain

In [3]:

```
fourier_values = fft(signal_data['0'].values)
fourier_values.shape
```

Out[3]:

```
(800000,)
```

The above vector contains complex valued frequencies.We only need the real part from that.The useful values are present only the first half of the vector

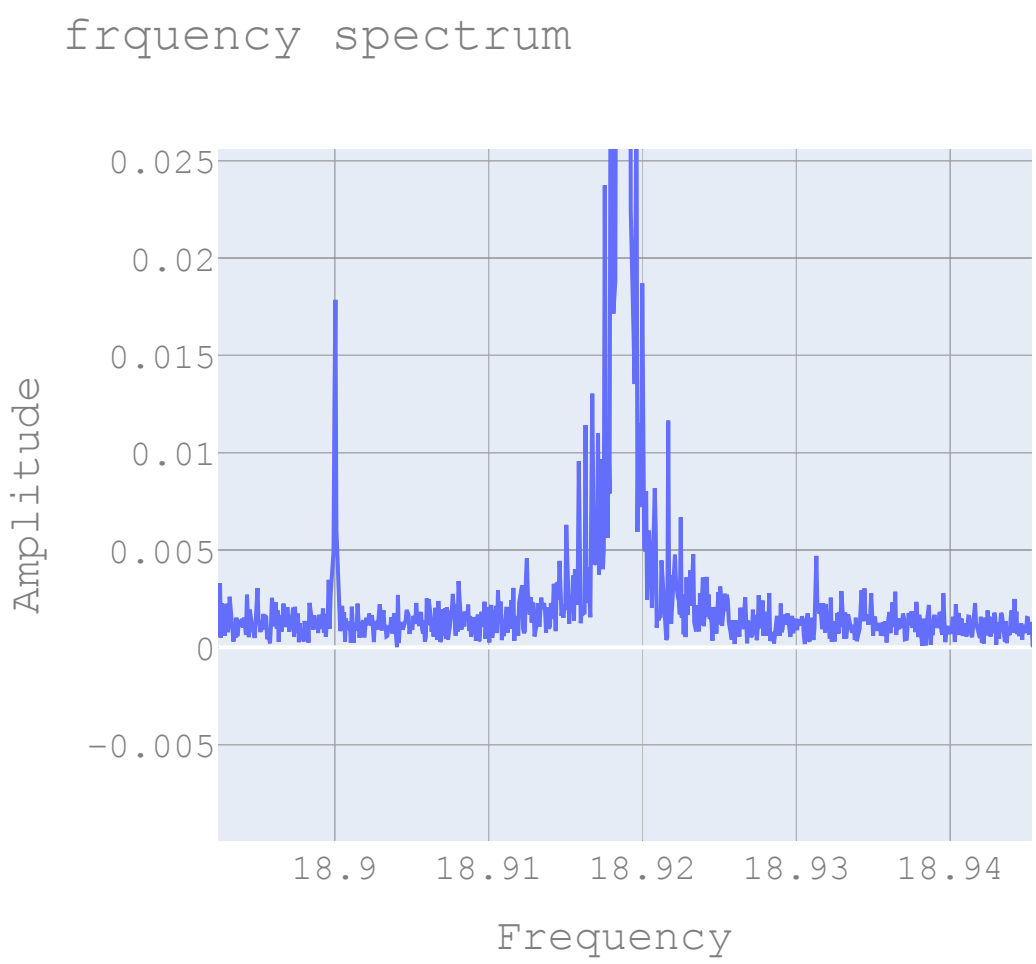
In [4]:

```
#N is number of points in our input signal
N=800000
#T is the time period which is inverseof frequency which is 50Hz
T = 1/50
#fft_values contains the filtered useful values from fft_values_ vector
fourier_values_filtered = 2.0/N * np.abs(fourier_values[0:N//2])
#f_values contains the frquency values
f_values = np.linspace(0.0, 1.0/(2.0*T), N//2)
```

In [5]:

```
= go.Figure(data=go.Scatter(x = f_values, y = fourier_values_filtered,mode = 'lines'
update_layout(
title="frquency spectrum",
xaxis_title="Frequency",
yaxis_title="Amplitude",
font=dict(
    family="Courier New, monospace",
    size=18,
    color="#7f7f7f"
)

show()
```



The graph has to be zoomed in as there are some amplitudes which are very large compared to others and there are 400,000 frequencies
In the above graph we can observe the amplitude values for different frequency values

In [6]:

```
def extract_fourier_features(signal,N=800000,T=1/50):
    '''
    converts a signal from time spectrum to frequency spectrum
    and returns only the features required as mentioned above
    '''
    fourier_values = fft(signal)
    fourier_values_filtered = 2.0/N * np.abs(fourier_values[0:N//2])
    return fourier_values_filtered
```

Power Spectral density

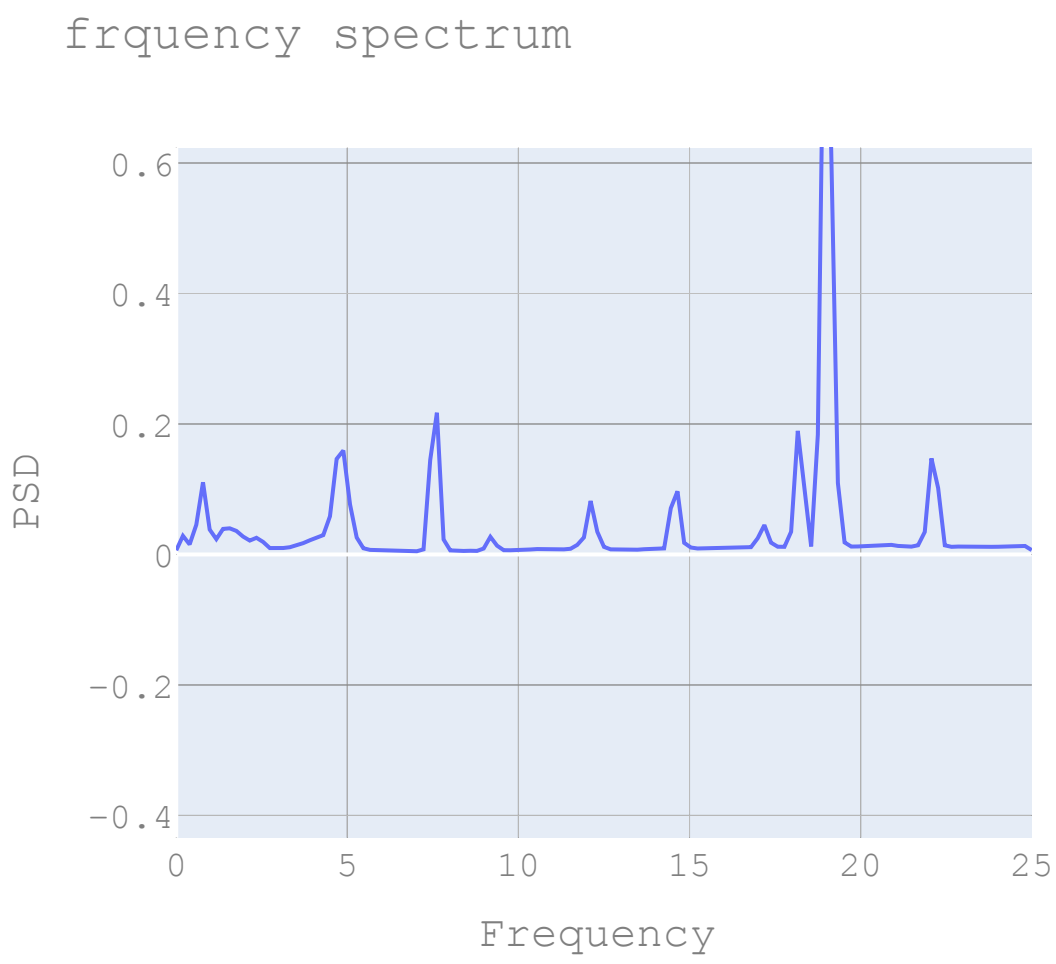
similiar to fft but this also considers the power distribution at each frequency

In [11]:

```
f_values, psd_values = welch(signal_data['0'].values, fs=50)
```

In [12]:

```
fig = go.Figure(data=go.Scatter(x = f_values, y = psd_values,mode = 'lines'))
fig.update_layout(
    title="frquency spectrum",
    xaxis_title="Frequency",
    yaxis_title="PSD",
    font=dict(
        family="Courier New, monospace",
        size=18,
        color="#7f7f7f"
    )
)
fig.show()
```



In the above graph we can observe the psd value values for different frequency values
There are less peaks in the above graph compared to fft but the peaks in the above graph can also be used as features for our model.

In [3]:

```
def extract_psd_features(signal,fs=50):
    '''
    returns the features from a time spectrum signal(similiar to fft
    but also considers power spectral density)
    '''
    f_values, psd_values = welch(signal, fs=50)
    return psd_values
```

In [4]:

```
def filter_features_fourier(features,mph,no_features=8):
    '''
    returns fourier transformed features by extracting peaks and
    considering only required number of peaks.
    mph-detect peaks that are greater than minimum peak height
    '''
    indices_peaks = detect_peaks(features,mph = mph)
    #print(indices_peaks)
    values = features[indices_peaks]
    if len(values)< no_features:
        return np.append(values , [0]*(no_features-len(values)))
    else:
        return values[:no_features]
```

In [65]:

```
def extract_stat_features(signal_features):
    '''
    used to extract statistical features from power spectral density
    '''
    percentiles = (5,25,50,75)
    #array to store
    filtered_features = np.zeros((len(signal_features),8))
    #mean
    filtered_features[:,0] = np.mean(signal_features,axis=1)
    #standard deviation
    filtered_features[:,1] = np.std(signal_features,axis=1)
    #maximum
    filtered_features[:,2] = np.max(signal_features,axis=1)
    #minimum
    filtered_features[:,3] = np.min(signal_features,axis=1)
    #mean
    filtered_features[:,4] = np.mean(signal_features,axis=1)
    #percentiles
    filtered_features[:,4:] = np.percentile(signal_features,percentiles,axis=1).T

    return filtered_features
```

Extracting statistical features and signal features from our signal dataset

we combine the phases by taking the average of the features that we calculate over each phase

In [70]:

```
total_features = []
for i in tqdm(range(0,8712,3)):

    std = 0
    maxs = 0
    bw_low = 0

    #arrays to store features
    fourier_features = np.empty((3,8))
    psd_features = np.empty((3,8))
    features = []

    for j in range(3):

        signal =  pq.read_pandas('../input/vsb-power-line-fault-detection/train.par
        #standard deviation
        std += signal.std()
        #max of signal
        maxs += signal.max()
        #lower bandwidth
        bw_low += signal.mean()-signal.std()

        #minimum peak height which can be used to filter fourier features
        mph = signal.min() + (signal.max() - np.abs(signal.min()))/10

        #fourier features
        fourier_features_ = extract_fourier_features(signal)

        fourier_features[j,:] = filter_features_fourier(fourier_features_,mph)

        #power spectral density features
        psd_features[j,:] = extract_psd_features(signal)

    #calculating the average of above features

    features.append(std/3)

    features.append(maxs/3)

    features.append(bw_low/3)

    features.extend(np.mean(fourier_features,axis=0))

    features.extend(extract_stat_features(np.mean(psd_features,axis=0)))

    total_features.append(features)
```

100%|██████████| 2904/2904 [57:35<00:00, 1.19s/it]

In [17]:

```
sig_features = np.array(total_features)
sig_features.shape
```

Out[17]:

```
(2904, 76)
```

In [76]:

```
#np.save('trainfeatures/train_signal_features',sig_features)
```

Labels

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

In [ ]:

```
y_train = meta_train[:, 3].astype(bool)
labels = np.sum(y_train.reshape(-1, 3), axis=1) >= 2
```

In [ ]:

```
#np.save('traindata/train_labels',labels)
```

These are our final features which can be used to train our model

Reading the test data

In [5]:

```
meta_data_test = pd.read_csv('metadata_test.csv')
```

In [6]:

```
meta_data_test
```

Out[6]:

	signal_id	id_measurement	phase
0	8712	2904	0
1	8713	2904	1
2	8714	2904	2
3	8715	2905	0
4	8716	2905	1
...	...	...	...
20332	29044	9681	1
20333	29045	9681	2
20334	29046	9682	0
20335	29047	9682	1
20336	29048	9682	2

20337 rows × 3 columns

There are 20337 test data points after we combine all the phases of each signal we would be left with 6779 data points

calculating spectra features of test dataset

In [24]:

```
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(4):

    begin = 8712 + 3*int(length//3 *(i/4))

    end = 8712 + 3*int(length//3 *((i+1)/4))

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

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

    #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
```

```
8712-13794
100%|██████████| 5082/5082 [01:40<00:00, 50.52it/s]
13794-18879
100%|██████████| 5085/5085 [01:42<00:00, 49.73it/s]
18879-23964
100%|██████████| 5085/5085 [01:41<00:00, 50.31it/s]
23964-29049
100%|██████████| 5085/5085 [01:41<00:00, 49.89it/s]
```

In [25]:

```
#concatenate all 4 parts
X_test_spectra = np.concatenate(X_tests, axis=0)
```

In [28]:

```
#np.save('testdata/test_spectra_features',X_test_spectra)
```

In [26]:

```
X_test_spectra.shape
```

Out[26]:

```
(6779, 57)
```

Applying feature engineering on test dataset

In [9]:

```
total_features = []
for i in tqdm(range(0,8712,3)):

    std = 0
    maxs = 0
    bw_low = 0

    #arrays to store features
    fourier_features = np.empty((3,8))
    psd_features = np.empty((3,8))
    features = []

    for j in range(3):

        signal =  pq.read_pandas('test.parquet',columns=[str(i+j)]).to_pandas()[str
        #standard deviation
        std += signal.std()
        #max of signal
        maxs += signal.max()
        #lower bandwidth
        bw_low += signal.mean()-signal.std()

        #minimum peak height which can be used to filter fourier features
        mph = signal.min() + (signal.max() - np.abs(signal.min()))/10

        #fourier features
        fourier_features_ = extract_fourier_features(signal)

        fourier_features[j,:] = filter_features_fourier(fourier_features_,mph)

        #power spectral density features
        psd_features[j,:] = extract_psd_features(signal)

    #calculating the average of above features

    features.append(std/3)

    features.append(maxs/3)

    features.append(bw_low/3)

    features.extend(np.mean(fourier_features,axis=0))

    features.extend(extract_stat_features(np.mean(psd_features,axis=0)))


    total_features.append(features)
```

. . .

In [10]:

```
test_signal_features = np.array(total_features)
```

In [5]:

```
test_signal_features.shape
```

Out[5]:

```
(6779, 76)
```

In [83]:

```
#np.save('testdata/test_signal_features',test_signal_features)
```