In [2]:

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



#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 sklearn.model_selection import StratifiedKFold,train_test_split,RandomizedSear
from sklearn.metrics import matthews_corrcoef,make_scorer
from catboost import CatBoostClassifier
from lightgbm import LGBMClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.ensemble import RandomForestClassifier

from tqdm import tqdm
import ast
import pickle

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

In [14]:

```
def compute_spectra(signals, *, m = 1000):
    '''
    computes the mean and percentiles by combining different phases belonging to th
    '''
    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 [15]:

```
#https://www.kaggle.com/junkoda/handmade-features
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 [16]:

```
def compute_spectra_features(spectra,peaks):
    '''
    extracts the features from peaks found in percentiles and means
    '''

    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 percentile 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 [17]:

```
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
```

In [18]:

```
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 [19]:

```
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 [82]:

```
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(8)
    #mean
    filtered_features[0] = np.mean(signal_features,axis=0)
    #standard deviation
    filtered_features[1] = np.std(signal_features,axis=0)
    #maximum
    filtered_features[2] = np.max(signal_features,axis=0)
    #minimum
    filtered_features[3] = np.min(signal_features,axis=0)
    #mean
    filtered_features[4] = np.mean(signal_features,axis=0)
    #percentiles
    filtered_features[4:] = np.percentile(signal_features,percentiles,axis=0).T

    return filtered_features
```

In [114]:

```
def final_fun_1(X):
    '''
    returns the prediction for raw input
    '''

    spectra = compute_spectra(X)

    peaks = max_windowed(spectra['percentile'])

    spectra_features = compute_spectra_features(spectra, peaks)

    signal_features = []
    for i in tqdm(range(0,len(X),3)):

        std = 0
        maxs = 0
        bw_low = 0

        #arrays to store features
        fourier_features = np.empty((3,8))
        psd_features = np.empty((3,129))
        features = []

        for j in range(3):

            signal = raw_signal_data[i+j]
            #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)))


        signal_features.append(features)
```

```
    total_features = np.concatenate((spectra_features,signal_features),axis=1)

    #loading the saved models
    models = []
    for i in range(20):
        filename = 'finalmodels/model'+str(i)+'.sav'
        model = pickle.load(open(filename,'rb'))
        models.append(model)

    #prediction
    y_test_probas = np.empty((total_features.shape[0], 20))

    for i, model in enumerate(models):
        y_test_probas[:, i] = model.predict_proba(total_features)[:, 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 phas
    y_pred = np.repeat(y_test_proba > 0.25, 3)

    return y_pred
```

In [111]:

```
def final_fun_2(X,Y):
    '''
    returns the matthews correlation score for raw input
    '''

    spectra = compute_spectra(X)

    peaks = max_windowed(spectra['percentile'])

    spectra_features = compute_spectra_features(spectra, peaks)

    signal_features = []
    for i in tqdm(range(0,len(X),3)):

        std = 0
        maxs = 0
        bw_low = 0

        #arrays to store features
        fourier_features = np.empty((3,8))
        psd_features = np.empty((3,129))
        features = []

        for j in range(3):

            signal = raw_signal_data[i+j]
            #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)))


        signal_features.append(features)
```

```
    total_features = np.concatenate((spectra_features,signal_features),axis=1)

    #loading the saved models
    models = []
    for i in range(20):
        filename = 'finalmodels/model'+str(i)+'.sav'
        model = pickle.load(open(filename,'rb'))
        models.append(model)

    #prediction
    y_test_probas = np.empty((total_features.shape[0], 20))

    for i, model in enumerate(models):
        y_test_probas[:, i] = model.predict_proba(total_features)[:, 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 phas
    y_pred = np.repeat(y_test_proba > 0.25, 3)

    return matthews_corrcoef(y_pred,Y)
```

In [ ]: