Loading Data

```
#importing libraries

import matplotlib
import matplotlib.pyplot as plt
import random

import pandas as pd
import numpy as np

from tqdm import tqdm

from sklearn.decomposition import PCA
from sklearn.manifold import TSNE

from sklearn.model_selection import train_test_split
```

```
#!curl --header 'Host: doc-08-3c-docs.googleusercontent.com' --user-agent 'Mozilla/5.0 (X11; Linux x86_64; rv:81.0) Gecko/20100101 Firefox/81.0' --header 'Accept: text/html,appl
#!unzip /content/falls.zip
#!pip install seaborn --upgrade
import seaborn as sns

print(sns.__version__)
```

```
0.11.0
```

```
df = pd.read_csv("/content/falls/equip_failures_training_set.csv")

df.head()
```

	id	target	sensor1_measure	sensor2_measure	sensor3_measure	sensor4_measure	sensor5_measure	sensor6_measure	sensor7_histogram_bin0	sensor7_histogram_bin1	sensor7_h
0	1	0	76698	na	2130706438	280	0	0	0	0	
1	2	0	33058	na	0	na	0	0	0	0	
2	3	0	41040	na	228	100	0	0	0	0	
3	4	0	12	0	70	66	0	10	0	0	
4	5	0	60874	na	1368	458	0	0	0	0	

5 rows × 172 columns

Make dataset interpretable to machine

Replace na with np.nan

```
"""Instead of nan value we have na, so we will replace na with np.nan"""

df = df.replace('na', np.NaN)
df.head()
```

	id	target	sensor1_measure	sensor2_measure	sensor3_measure	sensor4_measure	sensor5_measure	sensor6_measure	sensor7_histogram_bin0	sensor7_histogram_bin1	sensor7_h
0	1	0	76698	NaN	2130706438	280	0	0	0	0	
1	2	0	33058	NaN	0	NaN	0	0	0	0	
2	3	0	41040	NaN	228	100	0	0	0	0	
3	4	0	12	0	70	66	0	10	0	0	
4	5	0	60874	NaN	1368	458	0	0	0	0	

5 rows × 172 columns

Change data-type of dataframe

```
df.dtypes
```

```
id                          int64
target                      int64
sensor1_measure             int64
sensor2_measure            object
sensor3_measure            object
                            ...
sensor105_histogram_bin7   object
sensor105_histogram_bin8   object
sensor105_histogram_bin9   object
sensor106_measure          object
sensor107_measure          object
Length: 172, dtype: object
```

```
"We could see that few coloumns are of int type, and other are of object type,So for using data we need to make them float data type"

df = df.astype("float32")
df.dtypes
```

```
id                         float32
target                     float32
```

Drop useless coloumn from feature

...

```
"""id coloumn is just index, we don't need it , so we will drop it"""
df = df.drop(["id"],axis=1)
df.head()
```

	target	sensor1_measure	sensor2_measure	sensor3_measure	sensor4_measure	sensor5_measure	sensor6_measure	sensor7_histogram_bin0	sensor7_histogram_bin1	sensor7_hist
0	0.0	76698.0	NaN	2.130706e+09	280.0	0.0	0.0	0.0	0.0	
1	0.0	33058.0	NaN	0.000000e+00	NaN	0.0	0.0	0.0	0.0	
2	0.0	41040.0	NaN	2.280000e+02	100.0	0.0	0.0	0.0	0.0	
3	0.0	12.0	0.0	7.000000e+01	66.0	0.0	10.0	0.0	0.0	
4	0.0	60874.0	NaN	1.368000e+03	458.0	0.0	0.0	0.0	0.0	

5 rows × 171 columns

Train test split

Here we are creating 4 dataset

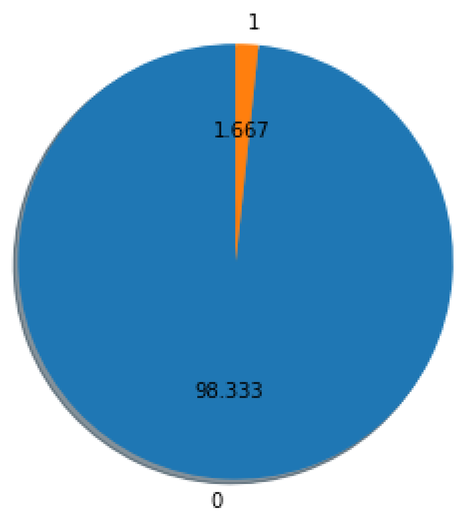
- df_train = it donot contain target
- df_test = it donot contain target
- df_train_t = it contain target for purpose EDA and for purpose feature selection based on collinearity with target
- df_test_t = it contain target for purpose EDA and for purpose feature selection based on collinearity with target

```
#We are not dropping target because, target will be used for EDA in next cells

y = df["target"].tolist()
df_ = df.drop(["target"],axis=1)

#For feature engineering
df_train , df_test , y_train , y_test = train_test_split( df_ , y , test_size=0.15, stratify = y ,  random_state=42)


#For EDA
df_train_t , df_test_t , y_train_t , y_test_t = train_test_split( df , y , test_size=0.15, stratify = y ,  random_state=42)


print("train = ",df_train.shape)
print("test = ",df_test.shape)
```

```
train =  (51000, 170)
test =  (9000, 170)
```

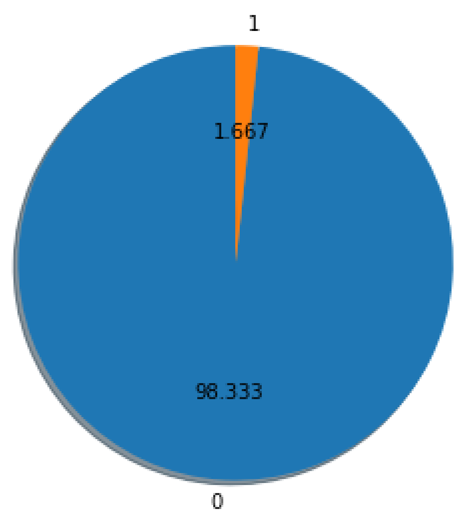
```
#train Percentage view of data distribution

plt.figure(figsize=(5,5))
plt.pie(df_train_t['target'].value_counts(),startangle=90,autopct="%.3f",labels=[0,1],shadow=True)
```

```
([<matplotlib.patches.Wedge at 0x7f5783ac0860>,
  <matplotlib.patches.Wedge at 0x7f5783ad4358>],
 [Text(-0.05756949701481714, -1.0984924911047236, '0'),
  Text(0.05756943916265415, 1.0984924941366225, '1')],
 [Text(-0.03140154382626389, -0.5991777224207583, '98.333'),
  Text(0.03140151227053862, 0.5991777240745213, '1.667')])
```




```
#test Percentage view of data distribution

plt.figure(figsize=(5,5))
plt.pie(df_test_t['target'].value_counts(),startangle=90,autopct="%.3f",labels=[0,1],shadow=True)
```

```
([<matplotlib.patches.Wedge at 0x7f5783ab03c8>,
  <matplotlib.patches.Wedge at 0x7f5783ab0e80>],
 [Text(-0.05756949701481714, -1.0984924911047236, '0'),
  Text(0.05756943916265415, 1.0984924941366225, '1')],
 [Text(-0.03140154382626389, -0.5991777224207583, '98.333'),
  Text(0.03140151227053862, 0.5991777240745213, '1.667')])
```



Feature Engineering

For each feature create new feature, that tells presence of nan, because nan values also contains some information

```
coloumns = df_train.columns

#Train
for coloumn in tqdm(coloumns):
    df_train[coloumn + "_nan"] = [1.0 if np.isnan(x) else 0.0 for x in df_train[coloumn]]
    df_train_t[coloumn + "_nan"] = [1.0 if np.isnan(x) else 0.0 for x in df_train_t[coloumn]]



#Test
for coloumn in tqdm(coloumns):
    df_test[coloumn + "_nan"] = [1.0 if np.isnan(x) else 0.0 for x in df_test[coloumn]]
    df_test_t[coloumn + "_nan"] = [1.0 if np.isnan(x) else 0.0 for x in df_test_t[coloumn]]
```

```
  0%|          | 0/170 [00:00<?, ?it/s]/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:5: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  """
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:6: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy

100%|██████████| 170/170 [00:27<00:00,  6.23it/s]
  0%|          | 0/170 [00:00<?, ?it/s]/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:12: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  if sys.path[0] == '':
/usr/local/lib/python3.6/dist-packages/ipykernel_launcher.py:13: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
  del sys.path[0]
100%|██████████| 170/170 [00:05<00:00, 32.54it/s]
```

```
df_train.head()
```

	sensor1_measure	sensor2_measure	sensor3_measure	sensor4_measure	sensor5_measure	sensor6_measure	sensor7_histogram_bin0	sensor7_histogram_bin1	sensor7_histogra
54001	36.0	0.0	1.000000e+01	10.0	0.0	0.0	0.0	0.0	
47399	41968.0	NaN	2.130706e+09	504.0	0.0	0.0	0.0	0.0	
49418	7230.0	NaN	2.130706e+09	86.0	0.0	0.0	0.0	0.0	
57927	236940.0	0.0	2.130706e+09	496.0	0.0	0.0	0.0	0.0	
29123	20.0	0.0	2.200000e+01	4.0	12.0	14.0	0.0	0.0	

5 rows × 340 columns

Drop features with more than 50% nan values

```
"""These dictionary will contain number of null values for each coloumns"""

null = dict(df_train.isnull().sum())


def features_with_high_nan(dataframe = df,percentage=50):

    """This function will give list of coloumns that have nan values above the percentage"""

    coloumns = df.columns
    high_nan_features = []

    total_rows = len(df)
    null = dict(df.isnull().sum())

    restricted_nan = int((percentage/100)*total_rows)

    for coloumn in coloumns:
        if null[coloumn] >= restricted_nan:
            high_nan_features.append(coloumn)

    return(high_nan_features)


high_nan_features = features_with_high_nan(dataframe = df_train,percentage=50)
print(high_nan_features)
```

```
['sensor2_measure', 'sensor38_measure', 'sensor39_measure', 'sensor40_measure', 'sensor41_measure', 'sensor42_measure', 'sensor43_measure', 'sensor68_measure']
```

```
#Train
df_train = df_train.drop(high_nan_features,axis=1)
df_train_t = df_train_t.drop(high_nan_features,axis=1)


#Test
df_test = df_test.drop(high_nan_features,axis=1)
df_test_t = df_test_t.drop(high_nan_features,axis=1)

df_train.head()
```

	sensor1_measure	sensor3_measure	sensor4_measure	sensor5_measure	sensor6_measure	sensor7_histogram_bin0	sensor7_histogram_bin1	sensor7_histogram_bin2	sensor7_h
54001	36.0	1.000000e+01	10.0	0.0	0.0	0.0	0.0	0.0	

47399	41968.0	2.130706e+09	504.0	0.0	0.0	0.0	0.0	0.0
49418	7230.0	2.130706e+09	86.0	0.0	0.0	0.0	0.0	0.0
57927	236940.0	2.130706e+09	496.0	0.0	0.0	0.0	0.0	0.0
29123	20.0	2.200000e+01	4.0	12.0	14.0	0.0	0.0	0.0

Replace nan with median of that coloumn , because values of each feature is either very low or very high, replacing nan with mean is not sensible at all

```
#Train
df_train = df_train.fillna(df_train.median())
df_train_t = df_train_t.fillna(df_train_t.median())


#Test
df_test = df_test.fillna(df_train.median())                                     #Here we are filling test nan values with, train median
df_test_t = df_test_t.fillna(df_train_t.median())

df_test.head()
```

	sensor1_measure	sensor3_measure	sensor4_measure	sensor5_measure	sensor6_measure	sensor7_histogram_bin0	sensor7_histogram_bin1	sensor7_histogram_bin2	sensor7_hi
19886	346.0	36.0	28.0	0.0	0.0	0.0	0.0	0.0	
14400	39642.0	352.0	288.0	0.0	0.0	0.0	0.0	0.0	
13932	1142.0	84.0	72.0	0.0	0.0	0.0	0.0	0.0	
49050	78342.0	1132.0	970.0	0.0	0.0	0.0	0.0	0.0	
52585	2172.0	38.0	38.0	0.0	0.0	0.0	0.0	0.0	

5 rows × 332 columns

We have 100 simple sensor, and 7 time based sensor. Here we will extract min, max and mean from those time based sensors

```
time_based_sensor = []
sensor_name = []

for sensor in df_train.columns:
    if (("histogram" in sensor) and ("nan" not in sensor)):
        sensor_name.append(sensor)

        if len(sensor_name) == 10:
            time_based_sensor.append(sensor_name)
            sensor_name = []


print(time_based_sensor)
```

```
[['sensor7_histogram_bin0', 'sensor7_histogram_bin1', 'sensor7_histogram_bin2', 'sensor7_histogram_bin3', 'sensor7_histogram_bin4', 'sensor7_histogram_bin5', 'sensor7_histo
```

```
def mean(a,b,c,d,e,f,g,h,i,j):
    list_ = [a,b,c,d,e,f,g,h,i,j]
    return np.mean(list_)


def min_(a,b,c,d,e,f,g,h,i,j):
    list_ = [a,b,c,d,e,f,g,h,i,j]
    return min(list_)


def max_(a,b,c,d,e,f,g,h,i,j):
    list_ = [a,b,c,d,e,f,g,h,i,j]
    return max(list_)



#Train
for i in tqdm(range(0,len(time_based_sensor))):
    df_train[time_based_sensor[i][0].split("_")[0] + "_mean"] = df_train.apply(lambda row : mean(row[time_based_sensor[i][0]] , row[time_based_sensor[i][1]] , row[time_based_ser
                                                                                        row[time_based_sensor[i][3]] , row[time_based_sensor[i][4]] , row[time_based_sensor[i][5]] ,
                                                                                        row[time_based_sensor[i][6]] , row[time_based_sensor[i][7]] , row[time_based_sensor[i][8]] ,
                                                                                        row[time_based_sensor[i][9]] ) , axis = 1)


    df_train[time_based_sensor[i][0].split("_")[0] + "_min"] = df_train.apply(lambda row : min_(row[time_based_sensor[i][0]] , row[time_based_sensor[i][1]] , row[time_based_sens
                                                                                        row[time_based_sensor[i][3]] , row[time_based_sensor[i][4]] , row[time_based_sensor[i][5]] ,
                                                                                        row[time_based_sensor[i][6]] , row[time_based_sensor[i][7]] , row[time_based_sensor[i][8]] ,
                                                                                        row[time_based_sensor[i][9]] ) , axis = 1)

    df_train[time_based_sensor[i][0].split("_")[0] + "_max"] = df_train.apply(lambda row : max_(row[time_based_sensor[i][0]] , row[time_based_sensor[i][1]] , row[time_based_sens
                                                                                        row[time_based_sensor[i][3]] , row[time_based_sensor[i][4]] , row[time_based_sensor[i][5]] ,
                                                                                        row[time_based_sensor[i][6]] , row[time_based_sensor[i][7]] , row[time_based_sensor[i][8]] ,
                                                                                        row[time_based_sensor[i][9]] ) , axis = 1)



    df_train_t[time_based_sensor[i][0].split("_")[0] + "_mean"] = df_train_t.apply(lambda row : mean(row[time_based_sensor[i][0]] , row[time_based_sensor[i][1]] , row[time_basec
                                                                                        row[time_based_sensor[i][3]] , row[time_based_sensor[i][4]] , row[time_based_sensor[i][5]] ,
                                                                                        row[time_based_sensor[i][6]] , row[time_based_sensor[i][7]] , row[time_based_sensor[i][8]] ,
                                                                                        row[time_based_sensor[i][9]] ) , axis = 1)


    df_train_t[time_based_sensor[i][0].split("_")[0] + "_min"] = df_train_t.apply(lambda row : min_(row[time_based_sensor[i][0]] , row[time_based_sensor[i][1]] , row[time_based_
                                                                                        row[time_based_sensor[i][3]] , row[time_based_sensor[i][4]] , row[time_based_sensor[i][5]] ,
                                                                                        row[time_based_sensor[i][6]] , row[time_based_sensor[i][7]] , row[time_based_sensor[i][8]] ,
                                                                                        row[time_based_sensor[i][9]] ) , axis = 1)

    df_train_t[time_based_sensor[i][0].split("_")[0] + "_max"] = df_train_t.apply(lambda row : max_(row[time_based_sensor[i][0]] , row[time_based_sensor[i][1]] , row[time_based_
                                                                                        row[time_based_sensor[i][3]] , row[time_based_sensor[i][4]] , row[time_based_sensor[i][5]] ,
                                                                                        row[time_based_sensor[i][6]] , row[time_based_sensor[i][7]] , row[time_based_sensor[i][8]] ,
                                                                                        row[time_based_sensor[i][9]] ) , axis = 1)
```

```
#Test
for i in tqdm(range(0,len(time_based_sensor))):
    df_test[time_based_sensor[i][0].split("_")[0] + "_mean"] = df_test.apply(lambda row : mean(row[time_based_sensor[i][0]] , row[time_based_sensor[i][1]] , row[time_based_senso
                                                                                    row[time_based_sensor[i][3]] , row[time_based_sensor[i][4]] , row[time_based_sensor[i][5]] ,
                                                                                    row[time_based_sensor[i][6]] , row[time_based_sensor[i][7]] , row[time_based_sensor[i][8]] ,
                                                                                    row[time_based_sensor[i][9]] ) , axis = 1)


    df_test[time_based_sensor[i][0].split("_")[0] + "_min"] = df_test.apply(lambda row : min_(row[time_based_sensor[i][0]] , row[time_based_sensor[i][1]] , row[time_based_sensor
                                                                                    row[time_based_sensor[i][3]] , row[time_based_sensor[i][4]] , row[time_based_sensor[i][5]] ,
                                                                                    row[time_based_sensor[i][6]] , row[time_based_sensor[i][7]] , row[time_based_sensor[i][8]] ,
                                                                                    row[time_based_sensor[i][9]] ) , axis = 1)

    df_test[time_based_sensor[i][0].split("_")[0] + "_max"] = df_test.apply(lambda row : max_(row[time_based_sensor[i][0]] , row[time_based_sensor[i][1]] , row[time_based_sensor
                                                                                    row[time_based_sensor[i][3]] , row[time_based_sensor[i][4]] , row[time_based_sensor[i][5]] ,
                                                                                    row[time_based_sensor[i][6]] , row[time_based_sensor[i][7]] , row[time_based_sensor[i][8]] ,
                                                                                    row[time_based_sensor[i][9]] ) , axis = 1)


    df_test_t[time_based_sensor[i][0].split("_")[0] + "_mean"] = df_test_t.apply(lambda row : mean(row[time_based_sensor[i][0]] , row[time_based_sensor[i][1]] , row[time_based_s
                                                                                    row[time_based_sensor[i][3]] , row[time_based_sensor[i][4]] , row[time_based_sensor[i][5]] ,
                                                                                    row[time_based_sensor[i][6]] , row[time_based_sensor[i][7]] , row[time_based_sensor[i][8]] ,
                                                                                    row[time_based_sensor[i][9]] ) , axis = 1)


    df_test_t[time_based_sensor[i][0].split("_")[0] + "_min"] = df_test_t.apply(lambda row : min_(row[time_based_sensor[i][0]] , row[time_based_sensor[i][1]] , row[time_based_se
                                                                                    row[time_based_sensor[i][3]] , row[time_based_sensor[i][4]] , row[time_based_sensor[i][5]] ,
                                                                                    row[time_based_sensor[i][6]] , row[time_based_sensor[i][7]] , row[time_based_sensor[i][8]] ,
                                                                                    row[time_based_sensor[i][9]] ) , axis = 1)

    df_test_t[time_based_sensor[i][0].split("_")[0] + "_max"] = df_test_t.apply(lambda row : max_(row[time_based_sensor[i][0]] , row[time_based_sensor[i][1]] , row[time_based_se
                                                                                    row[time_based_sensor[i][3]] , row[time_based_sensor[i][4]] , row[time_based_sensor[i][5]] ,
                                                                                    row[time_based_sensor[i][6]] , row[time_based_sensor[i][7]] , row[time_based_sensor[i][8]] ,
                                                                                    row[time_based_sensor[i][9]] ) , axis = 1)



df_train.head()
```

```
100%|██████████| 7/7 [04:06<00:00, 35.18s/it]
100%|██████████| 7/7 [00:43<00:00,  6.22s/it]
```

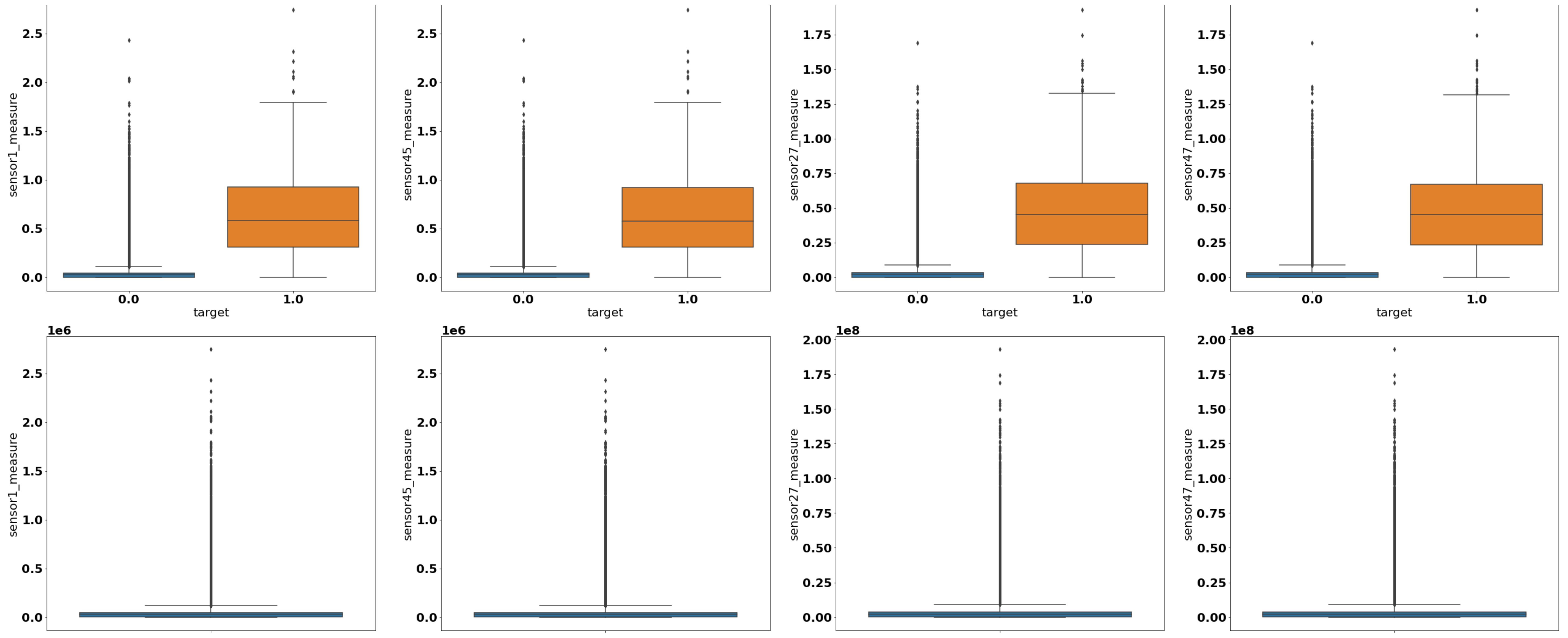
	sensor1_measure	sensor3_measure	sensor4_measure	sensor5_measure	sensor6_measure	sensor7_histogram_bin0	sensor7_histogram_bin1	sensor7_histogram_bin2	sensor7_h
54001	36.0	1.000000e+01	10.0	0.0	0.0	0.0	0.0	0.0	
47399	41968.0	2.130706e+09	504.0	0.0	0.0	0.0	0.0	0.0	
49418	7230.0	2.130706e+09	86.0	0.0	0.0	0.0	0.0	0.0	
57927	236940.0	2.130706e+09	496.0	0.0	0.0	0.0	0.0	0.0	
29123	20.0	2.200000e+01	4.0	12.0	14.0	0.0	0.0	0.0	

5 rows × 353 columns

We are not removing outliers, because most of the values of class 1 lies in outlies only

```
def get_highly_correlated_feature(dataframe=df,correlation="pearson",top_features=10,with_="target"):
    """
    correlation can be {'pearson', 'kendall', 'spearman'}
    top feature: it will give you top n correlated feature
    with: pass the coloumn name, with whome you want correlation
    datagrame: pass pandas dataframe
    """

    pearson_corr_dict = dataframe.corr(method=correlation)[with_].to_dict()

    #sorted_dict = dict(sorted(pearson_corr_dict.items(), key=lambda x: abs(x[1]) , reverse=True))
    #print(sorted_dict)

    top_n_features = dict(sorted(pearson_corr_dict.items(), key=lambda x: abs(x[1]) , reverse=True)[:top_features])

    return top_n_features

top_n_features = get_highly_correlated_feature(dataframe=df_train_t,correlation="pearson",top_features=5,with_="target")


font = {'family' : 'DejaVu Sans',
        'weight' : 'bold',
        'size'   : 22}

matplotlib.rc('font', **font)

figure, axis = plt.subplots(1 , 4, figsize=(50, 10), squeeze=False)

top_coloumns = list(top_n_features.keys())[1:]



for i in tqdm(range(0,4)):

    sns.boxplot( x= "target",    y=top_coloumns[i]       , data=df_train_t   , orient='v'       , ax = axis[0,i])


figure, axis = plt.subplots(1 , 4, figsize=(50, 10), squeeze=False)

top_coloumns = list(top_n_features.keys())[1:]


for i in tqdm(range(0,4)):

    sns.boxplot( y=top_coloumns[i]       , data=df_train_t   , orient='v'       , ax = axis[0,i])
```

```
100%|██████████| 4/4 [00:00<00:00, 30.28it/s]
100%|██████████| 4/4 [00:00<00:00, 84.83it/s]
```



Feature Selection

Removing all the features which are least correlated to our target

```
def get_least_correlated_feature(dataframe=df,correlation="pearson",bottom_features=10,with_="target"):
    """
    correlation can be {‘pearson’, ‘kendall’, ‘spearman’}
    top feature: it will give you top n correlated feature
    with: pass the coloumn name, with whome you want correlation
    datagrame: pass pandas dataframe
    """

    pearson_corr_dict = dataframe.corr(method=correlation)[with_].to_dict()

    #sorted_dict = dict(sorted(pearson_corr_dict.items(), key=lambda x: abs(x[1]) , reverse=True))
    #print(sorted_dict)

    bottom_n_features = dict(sorted(pearson_corr_dict.items(), key=lambda x: abs(x[1]) , reverse=False)[:bottom_features])

    return bottom_n_features

bottom_n_features = get_least_correlated_feature(dataframe=df_train_t,correlation="pearson",bottom_features=50,with_="target")

print(bottom_n_features)
```

```
{'sensor25_histogram_bin9': 0.000404517327265069, 'sensor4_measure': -0.0005747918610703464, 'sensor56_measure': -0.0005748712717671971, 'sensor5_measure': 0.0049155354265
```

```
#train
df_train = df_train.drop(bottom_n_features.keys(),axis=1)
df_train_t = df_train_t.drop(bottom_n_features.keys(),axis=1)

#test
df_test = df_test.drop(bottom_n_features.keys(),axis=1)
df_test_t = df_test_t.drop(bottom_n_features.keys(),axis=1)
```

```
df_train.head()
```

	sensor1_measure	sensor7_histogram_bin2	sensor7_histogram_bin3	sensor7_histogram_bin4	sensor7_histogram_bin5	sensor7_histogram_bin6	sensor8_measure	sensor12_mea
54001	36.0	0.0	3554.0	5894.0	3500.0	1250.0	3498.0	45
47399	41968.0	0.0	0.0	42052.0	1319986.0	1088958.0	1113500.0	
49418	7230.0	0.0	5158.0	104828.0	266524.0	90672.0	193304.0	94
57927	236940.0	0.0	102444.0	1775500.0	6072816.0	1820694.0	6238706.0	80(
29123	20.0	0.0	0.0	4.0	2668.0	2238.0	2000.0	

5 rows × 303 columns

Removing all the intercorrelated features

```
def get_threshold_highly_correlated_feature(dataframe=df,correlation="pearson",threshold=0.9,with_="sensor" , target = "target" , verbose=0):
    """
    correlation can be {‘pearson’, ‘kendall’, ‘spearman’}
    top threshold : it will give you top correlated feature whose correlation is more than threshold
    with: pass the coloumn name, with whome you want correlation
    datagrame: pass pandas dataframe
    """

    pearson_corr_dict = dataframe.corr(method=correlation)[with_].to_dict()

    #sorted_dict = dict(sorted(pearson_corr_dict.items(), key=lambda x: abs(x[1]) , reverse=True))
    #print(sorted_dict)

    top_features = dict(sorted(pearson_corr_dict.items(), key=lambda x: x[1] , reverse=True))

    if verbose == 1:
        print(top_features)

    del top_features[target]
    del top_features[with_]
```

```
    keys = list(top_features.keys())

    top_threshold = []

    i = 0

    while(top_features[keys[i]] > threshold):
        top_threshold.append(keys[i])
        i = i + 1

    #print(top_features)

    return top_threshold

print(get_threshold_highly_correlated_feature(dataframe=df_train_t,correlation="pearson",threshold=0.9,with_="sensor1_measure" , target = "target" , verbose=1))
```

```
{'sensor1_measure': 1.0, 'sensor45_measure': 0.9986261506454477, 'sensor15_measure': 0.9090568739804951, 'sensor27_measure': 0.9048353181987419, 'sensor14_measure': 0.9044(
['sensor45_measure', 'sensor15_measure', 'sensor27_measure', 'sensor14_measure', 'sensor46_measure', 'sensor47_measure']
```

```
def remove_intercorrelated_features(dataframe = df , correlation="pearson" , threshold = 0.9 , target = "target", verbose=0):
    """
    we do not want to remove features that are highly correlated to dataframe,
    but we want to remove highly intercollinearity.

    algorithm:- first select highly collinear feature to target, then remove all the highly intercollinear feature to that target.
    this keep on doing until reach the last feature
    """

    i = 2

    while(i < len(dataframe.columns)):

        top_feature = get_highly_correlated_feature(dataframe=dataframe, correlation=correlation , top_features=i, with_= target)

        top_coloumns = list(top_feature.keys())[1:]

        if verbose == 1:
            print("top_coloumns = ",top_coloumns)

        key = top_coloumns[-1]

        if verbose == 1:
            print("key = ",key)

        top_threshold = get_threshold_highly_correlated_feature(dataframe=dataframe,correlation=correlation , threshold=threshold ,with_=key , target = target , verbose=verbose)

        for thres in top_threshold:
            if thres in top_coloumns:
                top_threshold.remove(thres)

        if verbose == 1:
            print("top_threshold = ",top_threshold)

        dataframe = dataframe.drop(top_threshold , axis=1)

        i = i + 1

        print( str(dataframe.shape[1] - i ) + " itteration remaining" )

    return dataframe
```

```
#This cell will take, too much time to execute

df_filtered = remove_intercorrelated_features(dataframe = df_train_t , correlation="pearson" , threshold = 0.9 , target = "target" , verbose=0)
```

```
df_filtered.head()
```

	target	sensor1_measure	sensor7_histogram_bin2	sensor7_histogram_bin3	sensor7_histogram_bin4	sensor13_measure	sensor17_measure	sensor24_histogram_bin7	sensor24_
54001	0.0	36.0	0.0	3554.0	5894.0	6846.0	446.0	0.0	
47399	0.0	41968.0	0.0	0.0	42052.0	0.0	251130.0	642424.0	
49418	0.0	7230.0	0.0	5158.0	104828.0	14362.0	18516.0	146588.0	
57927	0.0	236940.0	0.0	102444.0	1775500.0	108746.0	1051812.0	2430398.0	
29123	0.0	20.0	0.0	0.0	4.0	0.0	276.0	0.0	

5 rows × 233 columns

These filtered coloumns are least intercorrelated and are most correlated to target

```
usefull_features = list(df_filtered.columns)[1:]
print(usefull_features)
print(len(usefull_features))
```

```
['sensor1_measure', 'sensor7_histogram_bin2', 'sensor7_histogram_bin3', 'sensor7_histogram_bin4', 'sensor13_measure', 'sensor17_measure', 'sensor24_histogram_bin7', 'sensor
232
```

```
total_features = list(df_train.columns)
print(total_features)
print(len(total_features))
```

```
['sensor1_measure', 'sensor7_histogram_bin2', 'sensor7_histogram_bin3', 'sensor7_histogram_bin4', 'sensor7_histogram_bin5', 'sensor7_histogram_bin6', 'sensor8_measure', 'se
303
```

```
useless_features = [x for x in total_features if x not in usefull_features]
print(useless_features)
print(len(useless_features))
```

```
['sensor7_histogram_bin5', 'sensor7_histogram_bin6', 'sensor8_measure', 'sensor12_measure', 'sensor14_measure', 'sensor15_measure', 'sensor16_measure', 'sensor26_histogram_
71
```

```
#Train
df_train = df_train.drop(useless_features , axis=1)
df_train_t = df_train_t.drop(useless_features , axis=1)

#Test
df_test = df_test.drop(useless_features , axis=1)
df_test_t = df_test_t.drop(useless_features , axis=1)


print("train_size = ",df_train.shape)
print("test_size = ",df_test.shape)
```

```
train_size =  (51000, 232)
test_size =  (9000, 232)
```

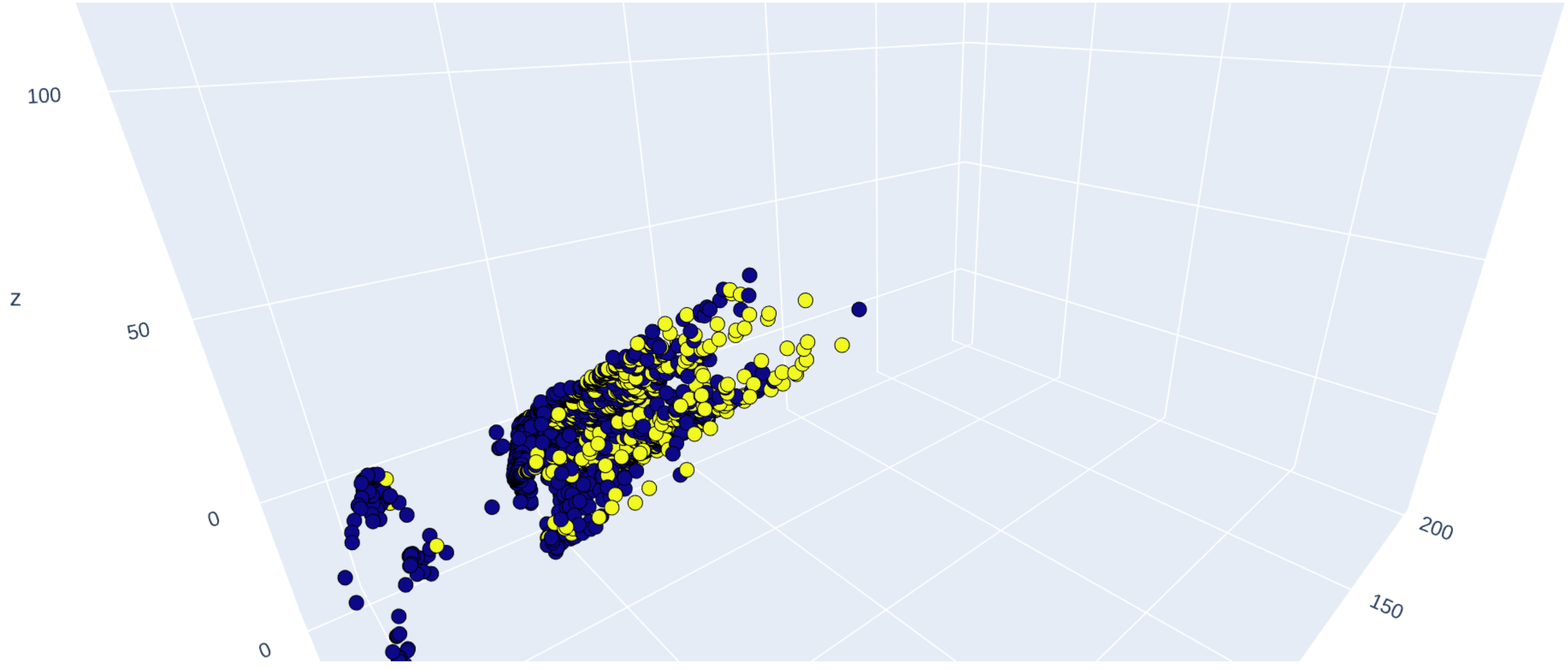
Feature Scaling

```
from sklearn.preprocessing import StandardScaler

scaler = StandardScaler()

scaler.fit(df_train)

#Train
df_train = scaler.transform(df_train)

#Test
df_test = scaler.transform(df_test)
```

PCA 3d plot

```
pca = PCA(n_components=3)
pca_result = pca.fit_transform(df_train)

x = pca_result[:,0]
y = pca_result[:,1]
z = pca_result[:,2]
```

```
import plotly.express as px

pca_df = pd.DataFrame(list(zip(x, y, z, y_train)), columns =['x', 'y', 'z', 'target'])

fig = px.scatter_3d(pca_df, x='x', y='y', z='z',
              color='target')

fig.update_traces(marker=dict(size=5,
                              line=dict(width=0.5,
                                        color='DarkSlateGrey')),
                  selector=dict(mode='markers'))
fig.show()
```



Observation

- Still data is not completely seperable, but better than before

Conclusion

- Data may be more seperable in higher dimentions