APS Failure Prediction

Required installtions

In [1]:
```
import warnings
warnings.filterwarnings('ignore')
```

In [ ]:
```
!pip3 install --upgrade xgboost
```

```
Requirement already up-to-date: xgboost in c:\users\hp\appdata\local\programs\python\python36\lib\site-packages (1.4.1)
Requirement already satisfied, skipping upgrade: scipy in c:\users\hp\appdata\local\programs\python\python36\lib\site-packages (from xgboost) (1.5.4)
Requirement already satisfied, skipping upgrade: numpy in c:\users\hp\appdata\local\programs\python\python36\lib\site-packages (from xgboost) (1.19.5)
WARNING: You are using pip version 20.2.1; however, version 21.1.1 is available.
You should consider upgrading via the 'c:\users\hp\appdata\local\programs\python\python36\python.exe -m pip install --upgrade pip' command.
```

In [ ]:
```
!pip3 install phik
```

```
Requirement already satisfied: phik in c:\users\hp\appdata\local\programs\python\python36\lib\site-packages (0.11.2)
Requirement already satisfied: pandas>=0.25.1 in c:\users\hp\appdata\local\programs\python\python36\lib\site-packages (from phik) (1.1.0)
Requirement already satisfied: numpy>=1.18.0 in c:\users\hp\appdata\local\programs\python\python36\lib\site-packages (from phik) (1.19.5)
Requirement already satisfied: matplotlib>=2.2.3 in c:\users\hp\appdata\local\programs\python\python36\lib\site-packages (from phik) (2.2.3)
Requirement already satisfied: scipy>=1.5.2 in c:\users\hp\appdata\local\programs\python\python36\lib\site-packages (from phik) (1.5.4)
Requirement already satisfied: joblib>=0.14.1 in c:\users\hp\appdata\local\programs\python\python36\lib\site-packages (from phik) (0.16.0)
Requirement already satisfied: python-dateutil>=2.7.3 in c:\users\hp\appdata\local\programs\python\python36\lib\site-packages (from pandas>=0.25.1->phik) (2.8.1)
Requirement already satisfied: pytz>=2017.2 in c:\users\hp\appdata\local\programs\python\python36\lib\site-packages (from pandas>=0.25.1->phik) (2020.1)
Requirement already satisfied: kiwisolver>=1.0.1 in c:\users\hp\appdata\local\programs\python\python36\lib\site-packages (from matplotlib>=2.2.3->phik) (1.2.0)
Requirement already satisfied: cycler>=0.10 in c:\users\hp\appdata\local\programs\python\python36\lib\site-packages (from matplotlib>=2.2.3->phik) (0.10.0)
```

```
Requirement already satisfied: six>=1.10 in c:\users\hp\appdata\local\programs\python\python36\lib\site-packages (from matplotlib>=2.2.3->phik) (1.15.0)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=2.0.1 in c:\users\hp\appdata\local\programs\python\python36\lib\site-packages (from matplotlib>=2.2.3->phik) (2.4.7)
WARNING: You are using pip version 20.2.1; however, version 21.1.1 is available.
You should consider upgrading via the 'c:\users\hp\appdata\local\programs\python\python36\python.exe -m pip install --upgrade pip' command.
```

Imports

In [2]:

```
import pandas as pd
import numpy as np
import csv
import matplotlib.pyplot as plt
import seaborn as sns
import random
from sklearn.impute import SimpleImputer
from sklearn.preprocessing import StandardScaler
from sklearn.manifold import TSNE
import itertools
import pickle
from sklearn.model_selection import train_test_split
from sklearn.model_selection import GridSearchCV
from sklearn.model_selection import RandomizedSearchCV
from sklearn.calibration import CalibratedClassifierCV
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.svm import SVC
from sklearn.ensemble import RandomForestClassifier
from xgboost import XGBClassifier
from sklearn.metrics import log_loss
from sklearn.metrics import confusion_matrix
from sklearn.metrics import f1_score
import os.path
from sklearn.utils import class_weight
```

Extracting data from csv file

As the csv file contains some informations about the data before the actual dataset, we need to find the row number and starting from that row, we will extract data.

In [ ]:

```python
def find_index(csv_file, input_text):
    """This function returns the row number in a csv file as the next row index of the input text
    """
    o = open(csv_file, 'r')
    myData = csv.reader(o)
    index = 1
    for row in myData:
        if len(row) > 0 and input_text in str(row[0]):
            return index+1
        else : index+=1
start_row = find_index('aps_failure_training_set.csv', '----')
```

Create dataframe from train and test csv files

In [ ]:

```python
train_data = pd.read_csv('aps_failure_training_set.csv', skiprows = start_row)
train_data.head()
```

Out[ ]:

	class	aa_000	ab_000	ac_000	ad_000	ae_000	af_000	ag_000	ag_001	ag_002	...	ee_002	ee_003	ee_004	ee_005	ee_006	ee_007	e
0	neg	76698	na	2130706438	280	0	0	0	0	0	...	1240520	493384	721044	469792	339156	157956	
1	neg	33058	na	0	na	0	0	0	0	0	...	421400	178064	293306	245416	133654	81140	
2	neg	41040	na	228	100	0	0	0	0	0	...	277378	159812	423992	409564	320746	158022	
3	neg	12	0	70	66	0	10	0	0	0	...	240	46	58	44	10	0	
4	neg	60874	na	1368	458	0	0	0	0	0	...	622012	229790	405298	347188	286954	311560	4

5 rows × 171 columns

In [ ]:

```python
print(train_data.shape)
```

```
(60000, 171)
```

In [ ]:

```python
test_data = pd.read_csv('aps_failure_test_set.csv', skiprows = start_row)
test_data.head()
```

Out[ ]:

	class	aa_000	ab_000	ac_000	ad_000	ae_000	af_000	ag_000	ag_001	ag_002	...	ee_002	ee_003	ee_004	ee_005	ee_006	ee_007	ee_00
0	neg	60	0	20	12	0	0	0	0	0	...	1098	138	412	654	78	88	

	class	aa_000	ab_000	ac_000	ad_000	ae_000	af_000	ag_000	ag_001	ag_002	...	ee_002	ee_003	ee_004	ee_005	ee_006	ee_007	ee_00
1	neg	82	0	68	40	0	0	0	0	0	...	1068	276	1620	116	86	462	
2	neg	66002	2	212	112	0	0	0	0	0	...	495076	380368	440134	269556	1315022	153680	51
3	neg	59816	na	1010	936	0	0	0	0	0	...	540820	243270	483302	485332	431376	210074	28166
4	neg	1814	na	156	140	0	0	0	0	0	...	7646	4144	18466	49782	3176	482	7

5 rows × 171 columns

In [ ]:
```
print(test_data.shape)
```

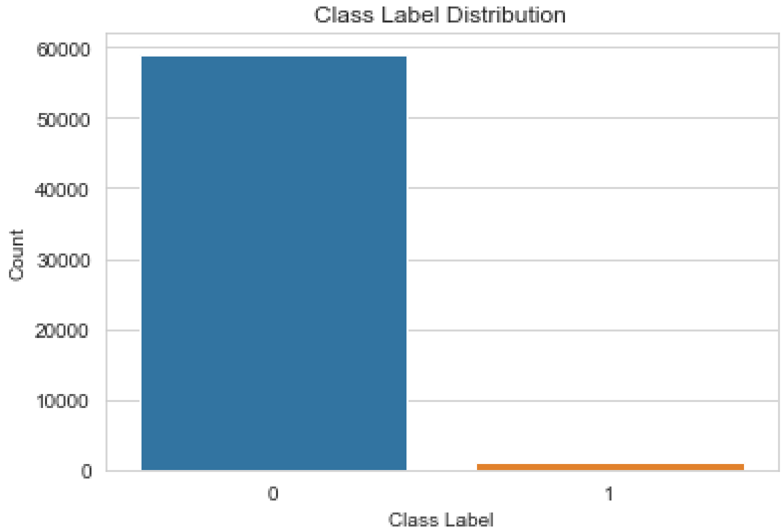
```
(16000, 171)
```

Convert string categorical data to numeric

In [ ]:
```
train_data.loc[train_data['class'] == 'neg', 'class'] = 0
train_data.loc[train_data['class'] == 'pos', 'class'] = 1

train_data['class'].value_counts()
```

Out[ ]:
```
0    59000
1     1000
Name: class, dtype: int64
```

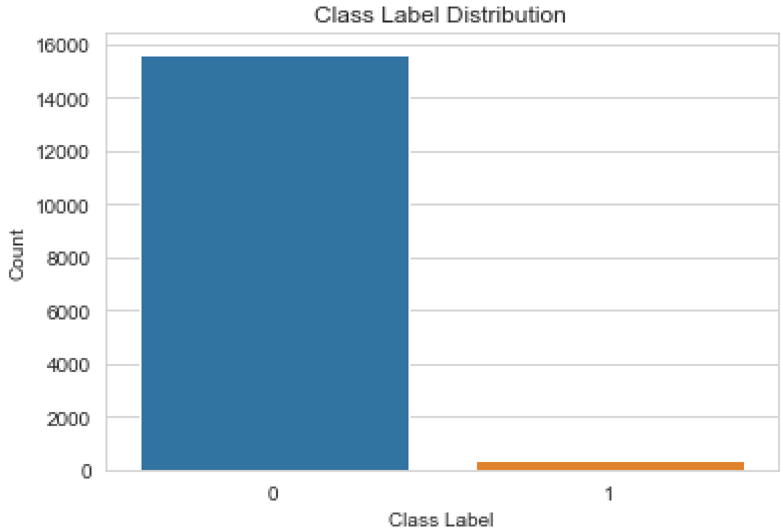
In [ ]:
```
# Class label distribution in train set
sns.barplot(train_data['class'].unique(),train_data['class'].value_counts())
plt.title('Class Label Distribution')
plt.xlabel('Class Label')
plt.ylabel('Count')
plt.show()
```



In [ ]:
```
test_data.loc[test_data['class'] == 'neg', 'class'] = 0
test_data.loc[test_data['class'] == 'pos', 'class'] = 1

test_data['class'].value_counts()
```

Out[ ]:
```
0    15625
1      375
Name: class, dtype: int64
```

In [ ]:
```
# Class label distribution in train set
sns.barplot(test_data['class'].unique(),test_data['class'].value_counts())
plt.title('Class Label Distribution')
plt.xlabel('Class Label')
plt.ylabel('Count')
plt.show()
```



From the above counts of class labels, we can see the datasets are an imbalanced datasets.

Handling NaN values

Converting 'na' to NaN

As the NaN are represented as strings ('na'), we need to convert 'na' to NaN, so that we can do preprocessing on NaN elements.

In [ ]:

```
train_data.replace(to_replace = 'na',
                  value = np.NaN, inplace = True)
test_data.replace(to_replace = 'na',
                  value = np.NaN, inplace = True)
```

In [ ]:

```
train_data.head()
```

Out[ ]:

	class	aa_000	ab_000	ac_000	ad_000	ae_000	af_000	ag_000	ag_001	ag_002	...	ee_002	ee_003	ee_004	ee_005	ee_006	ee_007	e
0	0	76698	NaN	2130706438	280	0	0	0	0	0	...	1240520	493384	721044	469792	339156	157956	
1	0	33058	NaN	0	NaN	0	0	0	0	0	...	421400	178064	293306	245416	133654	81140	
2	0	41040	NaN	228	100	0	0	0	0	0	...	277378	159812	423992	409564	320746	158022	

	class	aa_000	ab_000	ac_000	ad_000	ae_000	af_000	ag_000	ag_001	ag_002	...	ee_002	ee_003	ee_004	ee_005	ee_006	ee_007	e
3	0	12	0	70	66	0	10	0	0	0	...	240	46	58	44	10	0	
4	0	60874	NaN	1368	458	0	0	0	0	0	...	622012	229790	405298	347188	286954	311560	4

5 rows × 171 columns

In [ ]:
```
test_data.head()
```

Out[ ]:

	class	aa_000	ab_000	ac_000	ad_000	ae_000	af_000	ag_000	ag_001	ag_002	...	ee_002	ee_003	ee_004	ee_005	ee_006	ee_007	ee_00
0	0	60	0	20	12	0	0	0	0	0	...	1098	138	412	654	78	88	
1	0	82	0	68	40	0	0	0	0	0	...	1068	276	1620	116	86	462	
2	0	66002	2	212	112	0	0	0	0	0	...	495076	380368	440134	269556	1315022	153680	51
3	0	59816	NaN	1010	936	0	0	0	0	0	...	540820	243270	483302	485332	431376	210074	28166
4	0	1814	NaN	156	140	0	0	0	0	0	...	7646	4144	18466	49782	3176	482	7

5 rows × 171 columns

In [ ]:
```
test_data
```

Out[ ]:

	class	aa_000	ab_000	ac_000	ad_000	ae_000	af_000	ag_000	ag_001	ag_002	...	ee_002	ee_003	ee_004	ee_005	ee_006	ee_00
0	0	60	0	20	12	0	0	0	0	0	...	1098	138	412	654	78	8
1	0	82	0	68	40	0	0	0	0	0	...	1068	276	1620	116	86	46
2	0	66002	2	212	112	0	0	0	0	0	...	495076	380368	440134	269556	1315022	15368
3	0	59816	NaN	1010	936	0	0	0	0	0	...	540820	243270	483302	485332	431376	21007
4	0	1814	NaN	156	140	0	0	0	0	0	...	7646	4144	18466	49782	3176	48
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	.
15995	0	81852	NaN	2130706432	892	0	0	0	0	0	...	632658	273242	510354	373918	349840	31784

	class	aa_000	ab_000	ac_000	ad_000	ae_000	af_000	ag_000	ag_001	ag_002	...	ee_002	ee_003	ee_004	ee_005	ee_006	ee_00
15996	0	18	0	52	46	8	26	0	0	0	...	266	44	46	14	2	
15997	0	79636	NaN	1670	1518	0	0	0	0	0	...	806832	449962	778826	581558	375498	22286
15998	0	110	NaN	36	32	0	0	0	0	0	...	588	210	180	544	1004	133
15999	0	8	0	6	4	2	2	0	0	0	...	46	10	48	14	42	4

16000 rows × 171 columns

Checking NaN values in columns

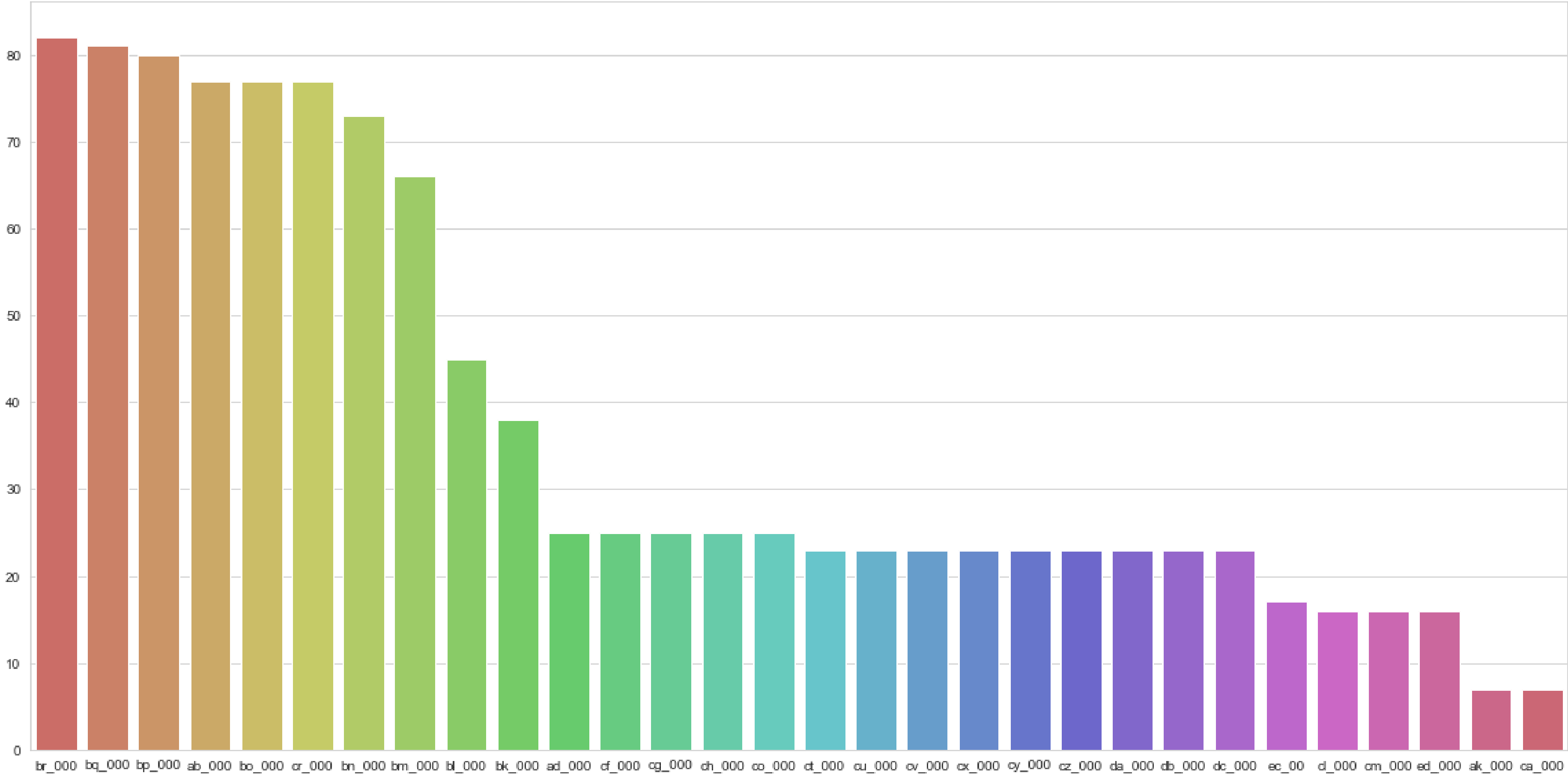
Checking NaN values for columns in train set

In [ ]:
```
nan_percentage_train_col = train_data.isna().sum().values / train_data.shape[0]
nan_percentage_train_col = np.round(nan_percentage_train_col,2) * 100
nan_percentage_train_col = nan_percentage_train_col.astype('int64')
nan_percentage_train_col.shape
```

Out[ ]:
```
(171,)
```

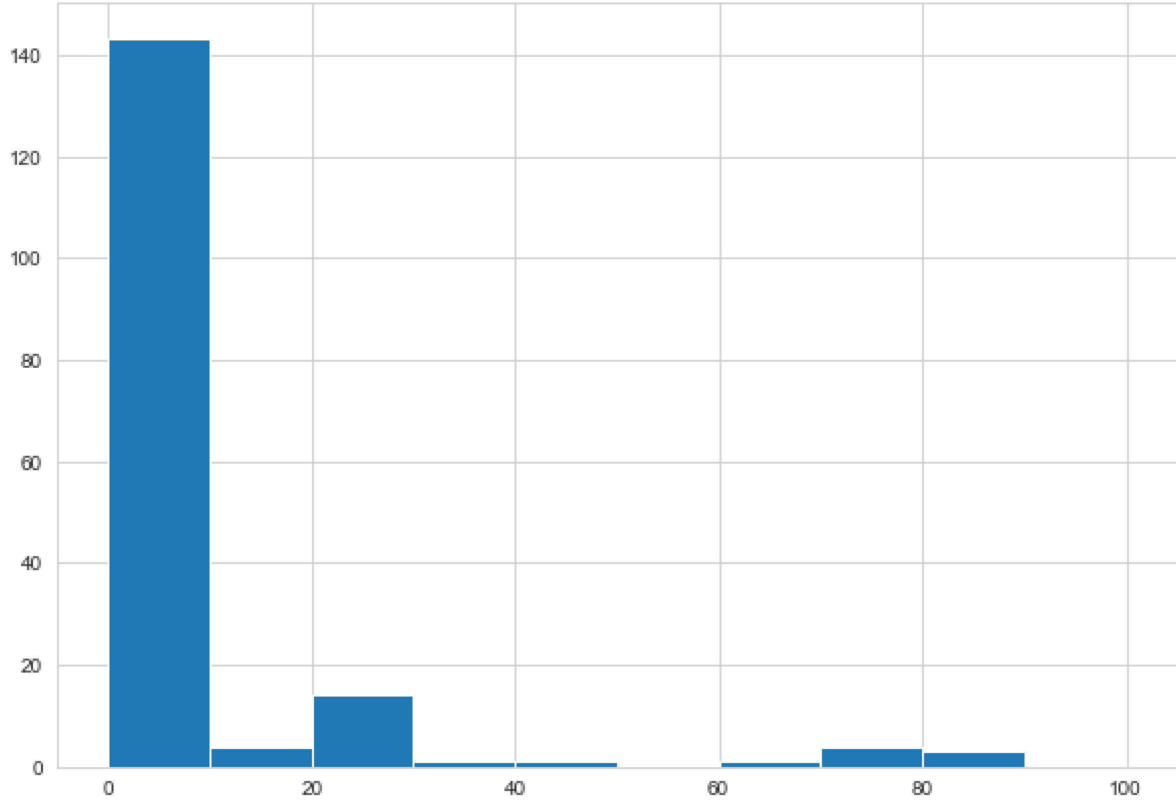
In [ ]:
```
# Create dict using column names and NaN percentage values and sort it
nan_percentage_train_col_dict = dict(zip(train_data.columns, nan_percentage_train_col))
nan_percentage_train_col_dict = {k: v for k, v in sorted(nan_percentage_train_col_dict.items(), key=lambda item: iter
```

In [ ]:
```
# Filtering out columns with less nan values for future use
sns.set_style(style="whitegrid")
plt.figure(figsize=(20,10))
plot = sns.barplot(x= list(nan_percentage_train_col_dict.keys())[:30],y = list(nan_percentage_train_col_dict.values()
plt.show()
```



Except 1st 10 columns as shown in above plot, rest all the columns has NaN values less than 30%. The columns with less NaN values can be more useful than others.

In [ ]:
```
fig, ax = plt.subplots(figsize =(10, 7))
ax.hist(nan_percentage_train_col, bins = [0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100])
plt.show()
```



We can see, there are 7 columns each having more 70% NaN values. So we can remove the columns having 70% or more NaN values.

Checking NaN values in rows

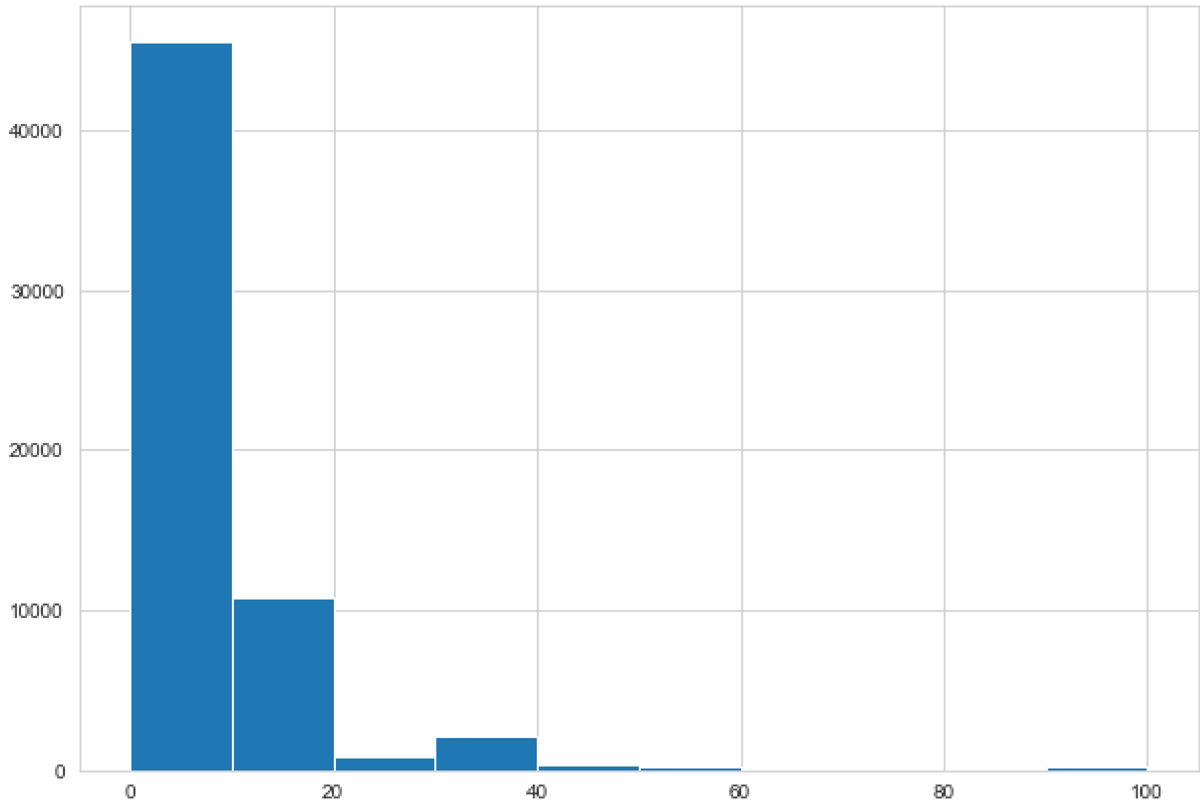
Checking NaN values for rows in train set

In [ ]:
```
nan_percentage_train_row = train_data.isna().sum(axis=1).values / train_data.shape[1]
nan_percentage_train_row = np.round(nan_percentage_train_row,2) * 100
nan_percentage_train_row = nan_percentage_train_row.astype('int64')
nan_percentage_train_row.shape
```

Out[ ]: (60000,)

In [ ]:
```
fig, ax = plt.subplots(figsize =(10, 7))
```

```
ax.hist(nan_percentage_train_row, bins = [0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100])
plt.show()
```



We can see there can be few data points having 90-100 percentage of NaN values in train set.

Data Preprocessing

In [ ]:
```
columns = train_data.columns[1:]
```

Replce NaN values (Imputation) and standardize data

In [ ]:
```
# For class 1 data point
imp = SimpleImputer(missing_values=np.nan, strategy='median')
```

```
X_train = imp.fit_transform(train_data.iloc[:, 1:])
X_test = imp.transform(test_data.iloc[:, 1:])
pickle.dump(imp, open('imputer.pkl', 'wb'))

print("Number of NaN after imputation", np.count_nonzero(np.isnan(X_train)))
```

```
Number of NaN after imputation 0
```

standardizing data for better EDA and modeling

In [ ]:
```
scaler = StandardScaler()
X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
pickle.dump(scaler, open('scaler.pkl', 'wb'))
```

In [ ]:
```
y_train = train_data.iloc[:, 0]
y_test = test_data.iloc[:, 0]
```

In [ ]:
```
train_data = pd.DataFrame(data = X_train, columns= columns)
test_data = pd.DataFrame(data = X_test, columns= columns)
train_data['class'] = y_train
test_data['class'] = y_test
```

Remove columns which have same values for all rows

In [ ]:
```
const_col = list(train_data.columns[train_data.nunique() <= 1])
print(const_col)
const_col = list(test_data.columns[test_data.nunique() <= 1])
print(const_col)
```

```
['cd_000']
['cd_000']
```

For both train and test data, 'cd_000' column has constant value. So we can drop that column.

In [ ]:
```
print(train_data.columns.get_loc("cd_000"))
```

```
89
```

In [ ]:
```
train_data.drop(const_col, axis=1, inplace = True)
test_data.drop(const_col, axis=1, inplace = True)
```

```
print("After dropping column in train set shape: ", train_data.shape)
print("After dropping column in test set shape: ", train_data.shape)
```

```
After dropping column in train set shape:  (60000, 170)
After dropping column in test set shape:  (60000, 170)
```

In [ ]:

```
train_data.head()
```

Out[ ]:

	aa_000	ab_000	ac_000	ad_000	ae_000	af_000	ag_000	ag_001	ag_002	ag_003	...	ee_003	ee_004	ee_005	
0	0.119381	-0.096307	2.310224	-0.004085	-0.041322	-0.051358	-0.010762	-0.02837	-0.056929	-0.115643	...	0.524393	0.239087	0.070072	0
1	-0.180697	-0.096307	-0.432859	-0.004089	-0.041322	-0.051358	-0.010762	-0.02837	-0.056929	-0.115643	...	-0.059135	-0.129021	-0.131171	-0
2	-0.125811	-0.096307	-0.432859	-0.004090	-0.041322	-0.051358	-0.010762	-0.02837	-0.056929	-0.115643	...	-0.092912	-0.016553	0.016053	-0
3	-0.407928	-0.096307	-0.432859	-0.004091	-0.041322	-0.002669	-0.010762	-0.02837	-0.056929	-0.115223	...	-0.388574	-0.381387	-0.351244	-0
4	0.010572	-0.096307	-0.432857	-0.004080	-0.041322	-0.051358	-0.010762	-0.02837	-0.056929	-0.115643	...	0.036588	-0.032641	-0.039892	-0

5 rows × 170 columns

Feature Selection

We need to use the features which are mainly responsible for predicting the target variable. (features with high correlation with class variable).

phi_k correlation can be used.

In [ ]:

```
import phik
```

In [ ]:

```
phi_k_corr = train_data.phik_matrix()
```

```
interval columns not set, guessing: ['aa_000', 'ab_000', 'ac_000', 'ad_000', 'ae_000', 'af_000', 'ag_000', 'ag_001',
'ag_002', 'ag_003', 'ag_004', 'ag_005', 'ag_006', 'ag_007', 'ag_008', 'ag_009', 'ah_000', 'ai_000', 'aj_000', 'ak_00
0', 'al_000', 'am_0', 'an_000', 'ao_000', 'ap_000', 'aq_000', 'ar_000', 'as_000', 'at_000', 'au_000', 'av_000', 'ax_0
00', 'ay_000', 'ay_001', 'ay_002', 'ay_003', 'ay_004', 'ay_005', 'ay_006', 'ay_007', 'ay_008', 'ay_009', 'az_000', 'a
z_001', 'az_002', 'az_003', 'az_004', 'az_005', 'az_006', 'az_007', 'az_008', 'az_009', 'ba_000', 'ba_001', 'ba_002',
'ba_003', 'ba_004', 'ba_005', 'ba_006', 'ba_007', 'ba_008', 'ba_009', 'bb_000', 'bc_000', 'bd_000', 'be_000', 'bf_00
0', 'bg_000', 'bh_000', 'bi_000', 'bj_000', 'bk_000', 'bl_000', 'bm_000', 'bn_000', 'bo_000', 'bp_000', 'bq_000', 'br
_000', 'bs_000', 'bt_000', 'bu_000', 'bv_000', 'bx_000', 'by_000', 'bz_000', 'ca_000', 'cb_000', 'cc_000', 'ce_000',
'cf_000', 'cg_000', 'ch_000', 'ci_000', 'cj_000', 'ck_000', 'cl_000', 'cm_000', 'cn_000', 'cn_001', 'cn_002', 'cn_00
3', 'cn_004', 'cn_005', 'cn_006', 'cn_007', 'cn_008', 'cn_009', 'co_000', 'cp_000', 'cq_000', 'cr_000', 'cs_000', 'cs
```

```
_001', 'cs_002', 'cs_003', 'cs_004', 'cs_005', 'cs_006', 'cs_007', 'cs_008', 'cs_009', 'ct_000', 'cu_000', 'cv_000',
'cx_000', 'cy_000', 'cz_000', 'da_000', 'db_000', 'dc_000', 'dd_000', 'de_000', 'df_000', 'dg_000', 'dh_000', 'di_00
0', 'dj_000', 'dk_000', 'dl_000', 'dm_000', 'dn_000', 'do_000', 'dp_000', 'dq_000', 'dr_000', 'ds_000', 'dt_000', 'du
_000', 'dv_000', 'dx_000', 'dy_000', 'dz_000', 'ea_000', 'eb_000', 'ec_00', 'ed_000', 'ee_000', 'ee_001', 'ee_002',
'ee_003', 'ee_004', 'ee_005', 'ee_006', 'ee_007', 'ee_008', 'ee_009', 'ef_000', 'eg_000', 'class']
```

In [ ]:

```
# Get correlation column-value pair for 'class'
corr_class = phi_k_corr.loc['class', :].to_dict()
corr_class.pop('class')
corr_class = {k: v for k, v in sorted(corr_class.items(), key=lambda item: item[1],reverse=True)[:30]}
```

In [ ]:

```
# Check whether these columns are correlated with each others or not
def filter_cols(corr_class):
    final_cols = set()
    corr_columns = list(corr_class.keys())
    for col in corr_columns:
        # get correlated columns for each column with values in decreasing order
        corr_col = phi_k_corr.loc[col, :].to_dict()
        corr_col.pop(col)
        corr_col_items = filter(lambda item: item[1] > 0.9, corr_col.items())
        corr_col = {k: v for k, v in sorted(corr_col_items, key=lambda item: item[1],reverse=True)}
        col_corr_columns = list(corr_col.keys())
        # get the common columns between the already existing columns and correlated columns
        common = list(set(corr_columns) & set(col_corr_columns))
        # check for each col
        if len(common) != 0:
            # We will add only one column chosen from the common_sorted list and the column itself
            #Selected column will have heighest correlation value with class label
            common.append(col)
            col_corr = {k:v for k,v in corr_class.items() if k in common}
            selected_col = max(col_corr, key= col_corr.get)
            final_cols.add(selected_col)
        else:
            # add column to final set, if it doesnt have any matched correlated column
            selected_col = col
            final_cols.add(selected_col)
    print('Final columns to proceed for EDA are: ', final_cols)
    print('Total final columns:', len(final_cols))
    return final_cols
```

In [ ]:

```
final_cols = filter_cols(corr_class)
```

```
Final columns to proceed for EDA are:  {'bh_000', 'cn_004', 'ee_000', 'bj_000', 'cc_000', 'ee_005', 'ay_008', 'aq_00
0', 'ba_005', 'cs_004', 'ci_000', 'ap_000', 'cn_001', 'ee_006', 'an_000', 'ao_000', 'bb_000', 'ck_000'}
Total final columns: 18
```

These are the most correlated features for the target class. We can use these 20 columns for EDA to get better output.

Get percentile values for each columns to get rid of outliers

In [ ]:
```
# 10,20...100 percentile values
def print_percentiles1(df, cols):
    for col in cols:
        for i in range(10,110,10):
            print('{}th Percentile value of column {} : {}'.format(i, col,np.percentile(df[col],i)))
        print('----------------Column {} END ---------------'.format(col))
```

In [ ]:
```
print_percentiles1(train_data, final_cols)
```

```
10th Percentile value of column bh_000 : -0.3790293441357247
20th Percentile value of column bh_000 : -0.3765462900319117
30th Percentile value of column bh_000 : -0.3724122691037549
40th Percentile value of column bh_000 : -0.3321022631205778
50th Percentile value of column bh_000 : -0.20639104524774682
60th Percentile value of column bh_000 : -0.1584337609342094
70th Percentile value of column bh_000 : -0.10146087698629573
80th Percentile value of column bh_000 : 0.003390044867587177
90th Percentile value of column bh_000 : 0.2973228951799113 4
100th Percentile value of column bh_000 : 20.755802646098974
----------------Column bh_000 END ---------------
10th Percentile value of column cn_004 : -0.3808408267119443
20th Percentile value of column cn_004 : -0.3788635531092587
30th Percentile value of column cn_004 : -0.36942079879778394
40th Percentile value of column cn_004 : -0.33731385279908177
50th Percentile value of column cn_004 : -0.22630560365708824
60th Percentile value of column cn_004 : -0.15051462796284432
70th Percentile value of column cn_004 : -0.07029146859828461
80th Percentile value of column cn_004 : 0.04082987029111493
90th Percentile value of column cn_004 : 0.3627731402543856
100th Percentile value of column cn_004 : 50.4936236914768
----------------Column cn_004 END ---------------
10th Percentile value of column ee_000 : -0.30084669036251915
20th Percentile value of column ee_000 : -0.2984681903503712
30th Percentile value of column ee_000 : -0.29203827564242
40th Percentile value of column ee_000 : -0.2575395806728418
50th Percentile value of column ee_000 : -0.19450362940789226
```

```
60th Percentile value of column ee_000 : -0.1559810673650316
70th Percentile value of column ee_000 : -0.10601126139583036
80th Percentile value of column ee_000 : -0.015873692773525128
90th Percentile value of column ee_000 : 0.2273512573997535
100th Percentile value of column ee_000 : 30.90008002550293
----------------Column ee_000 END ---------------
10th Percentile value of column bj_000 : -0.2781424131357906
20th Percentile value of column bj_000 : -0.2762908816858898
30th Percentile value of column bj_000 : -0.27231511260354463
40th Percentile value of column bj_000 : -0.2464550587978459
50th Percentile value of column bj_000 : -0.19442382324215385
60th Percentile value of column bj_000 : -0.16434002542050963
70th Percentile value of column bj_000 : -0.12522048914568223
80th Percentile value of column bj_000 : -0.05735683797187588
90th Percentile value of column bj_000 : 0.1538706615122914
100th Percentile value of column bj_000 : 24.968469323630597
----------------Column bj_000 END ---------------
10th Percentile value of column cc_000 : -0.3952838731902859
20th Percentile value of column cc_000 : -0.3922710599320991
30th Percentile value of column cc_000 : -0.38458230311475994
40th Percentile value of column cc_000 : -0.32365497500403706
50th Percentile value of column cc_000 : -0.17105895400046486
60th Percentile value of column cc_000 : -0.13146852432536543
70th Percentile value of column cc_000 : -0.08901742599596568
80th Percentile value of column cc_000 : 0.007682669948012241
90th Percentile value of column cc_000 : 0.2888263648429468
100th Percentile value of column cc_000 : 15.46675452214507
----------------Column cc_000 END ---------------
10th Percentile value of column ee_005 : -0.3512583001986739
20th Percentile value of column ee_005 : -0.3506110993018916
30th Percentile value of column ee_005 : -0.3408781521127268
40th Percentile value of column ee_005 : -0.29550825709597606
50th Percentile value of column ee_005 : -0.1808837403971903
60th Percentile value of column ee_005 : -0.1070249875018211
70th Percentile value of column ee_005 : -0.038104908411942
80th Percentile value of column ee_005 : 0.0721811379737687
90th Percentile value of column ee_005 : 0.3296634135964347
100th Percentile value of column ee_005 : 51.16220916719786
----------------Column ee_005 END ---------------
10th Percentile value of column ay_008 : -0.2619184944724901
20th Percentile value of column ay_008 : -0.26080199579166574
30th Percentile value of column ay_008 : -0.2586058758343291
40th Percentile value of column ay_008 : -0.253100260734825
50th Percentile value of column ay_008 : -0.23819584467457927
60th Percentile value of column ay_008 : -0.2100228176333392
```

```
70th Percentile value of column ay_008 : -0.15436584707023077
80th Percentile value of column ay_008 : -0.04736011374066457
90th Percentile value of column ay_008 : 0.21801381876412257
100th Percentile value of column ay_008 : 26.07780703251284
----------------Column ay_008 END ---------------
10th Percentile value of column aq_000 : -0.3493890519687957
20th Percentile value of column aq_000 : -0.3479393282078672
30th Percentile value of column aq_000 : -0.344924800506042
40th Percentile value of column aq_000 : -0.3143362339978703
50th Percentile value of column aq_000 : -0.20773696986507606
60th Percentile value of column aq_000 : -0.15667893416727308
70th Percentile value of column aq_000 : -0.09458625316313211
80th Percentile value of column aq_000 : 0.007186965248170011
90th Percentile value of column aq_000 : 0.2852205208812105
100th Percentile value of column aq_000 : 19.99405062770798
----------------Column aq_000 END ---------------
10th Percentile value of column ba_005 : -0.37058904077824417
20th Percentile value of column ba_005 : -0.3702889171549093
30th Percentile value of column ba_005 : -0.36939644427499246
40th Percentile value of column ba_005 : -0.3518684348732271
50th Percentile value of column ba_005 : -0.20484576979156213
60th Percentile value of column ba_005 : -0.14182612728330957
70th Percentile value of column ba_005 : -0.06547665200444075
80th Percentile value of column ba_005 : 0.07011090812866205
90th Percentile value of column ba_005 : 0.39179999339815974
100th Percentile value of column ba_005 : 37.556771702122504
----------------Column ba_005 END ---------------
10th Percentile value of column cs_004 : -0.21325430189601258
20th Percentile value of column cs_004 : -0.21268610297525706
30th Percentile value of column cs_004 : -0.21076140185290257
40th Percentile value of column cs_004 : -0.19678681119505798
50th Percentile value of column cs_004 : -0.16930130346965078
60th Percentile value of column cs_004 : -0.1523571462338098
70th Percentile value of column cs_004 : -0.12983814310226255
80th Percentile value of column cs_004 : -0.09053690973106214
90th Percentile value of column cs_004 : 0.03624892245368018
100th Percentile value of column cs_004 : 36.07982538507858
----------------Column cs_004 END ---------------
10th Percentile value of column ci_000 : -0.4162655593825422
20th Percentile value of column ci_000 : -0.41405923651479054
30th Percentile value of column ci_000 : -0.4070204054266747
40th Percentile value of column ci_000 : -0.3514599818384025
50th Percentile value of column ci_000 : -0.1936128039202838
60th Percentile value of column ci_000 : -0.14814753520932397
70th Percentile value of column ci_000 : -0.1101905657306674
```

```
80th Percentile value of column ci_000 : 0.00323935231073315
90th Percentile value of column ci_000 : 0.36684363959697874
100th Percentile value of column ci_000 : 16.50186736995178
----------------Column ci_000 END ---------------
10th Percentile value of column ap_000 : -0.3212142730856814
20th Percentile value of column ap_000 : -0.3185806594844594
30th Percentile value of column ap_000 : -0.3119733877143347
40th Percentile value of column ap_000 : -0.2694450844795145
50th Percentile value of column ap_000 : -0.2082797015041634
60th Percentile value of column ap_000 : -0.17161252291364476
70th Percentile value of column ap_000 : -0.12446673866934652
80th Percentile value of column ap_000 : -0.041506803670515874
90th Percentile value of column ap_000 : 0.2029813237169575
100th Percentile value of column ap_000 : 25.040062911918795
----------------Column ap_000 END ---------------
10th Percentile value of column cn_001 : -0.08879049327479388
20th Percentile value of column cn_001 : -0.08879049327479388
30th Percentile value of column cn_001 : -0.08879049327479388
40th Percentile value of column cn_001 : -0.08879049327479388
50th Percentile value of column cn_001 : -0.08879049327479388
60th Percentile value of column cn_001 : -0.08879049327479388
70th Percentile value of column cn_001 : -0.08879049327479388
80th Percentile value of column cn_001 : -0.08879049327479388
90th Percentile value of column cn_001 : -0.08205146395283505
100th Percentile value of column cn_001 : 59.294021962956734
----------------Column cn_001 END ---------------
10th Percentile value of column ee_006 : -0.31064288755452263
20th Percentile value of column ee_006 : -0.3105488545703816
30th Percentile value of column ee_006 : -0.30895593581903286
40th Percentile value of column ee_006 : -0.2992272832798038
50th Percentile value of column ee_006 : -0.2237376036113997
60th Percentile value of column ee_006 : -0.1477202108359962
70th Percentile value of column ee_006 : -0.09120939642274077
80th Percentile value of column ee_006 : -0.0009791061603916069
90th Percentile value of column ee_006 : 0.2630698388248393б
100th Percentile value of column ee_006 : 29.411116554427824
----------------Column ee_006 END ---------------
10th Percentile value of column an_000 : -0.4427608837961114
20th Percentile value of column an_000 : -0.438868117693785
30th Percentile value of column an_000 : -0.4303648158525458
40th Percentile value of column an_000 : -0.3587232307627656
50th Percentile value of column an_000 : -0.19688785479948112
60th Percentile value of column an_000 : -0.14725034513206148
70th Percentile value of column an_000 : -0.09721108678763131
80th Percentile value of column an_000 : 0.0279758706145501
```

```
90th Percentile value of column an_000 : 0.41735302124067397
100th Percentile value of column an_000 : 17.73099214949411
----------------Column an_000 END ---------------
10th Percentile value of column ao_000 : -0.4388363705636256
20th Percentile value of column ao_000 : -0.43494827406501074
30th Percentile value of column ao_000 : -0.4258514242216618
40th Percentile value of column ao_000 : -0.3539612869910947
50th Percentile value of column ao_000 : -0.19824653912091078
60th Percentile value of column ao_000 : -0.14812734955341772
70th Percentile value of column ao_000 : -0.09868985206228077
80th Percentile value of column ao_000 : 0.02625634352120773
90th Percentile value of column ao_000 : 0.4198883740073929
100th Percentile value of column ao_000 : 17.564270680432145
----------------Column ao_000 END ---------------
10th Percentile value of column bb_000 : -0.41341292050461587
20th Percentile value of column bb_000 : -0.4092000998512446
30th Percentile value of column bb_000 : -0.4020876848414522
40th Percentile value of column bb_000 : -0.34073124741740174
50th Percentile value of column bb_000 : -0.19779469128962315
60th Percentile value of column bb_000 : -0.15728690551027075
70th Percentile value of column bb_000 : -0.10751700832241343
80th Percentile value of column bb_000 : 0.0035435292440028653
90th Percentile value of column bb_000 : 0.3411511243412503
100th Percentile value of column bb_000 : 17.392786516951077
----------------Column bb_000 END ---------------
10th Percentile value of column ck_000 : -0.3252629737723142
20th Percentile value of column ck_000 : -0.32261456996246013
30th Percentile value of column ck_000 : -0.31679852546793436
40th Percentile value of column ck_000 : -0.2779428069633934
50th Percentile value of column ck_000 : -0.2121833967881883
60th Percentile value of column ck_000 : -0.17149567216143702
70th Percentile value of column ck_000 : -0.11748506119351207
80th Percentile value of column ck_000 : -0.02151915728316174
90th Percentile value of column ck_000 : 0.25075668868698614
100th Percentile value of column ck_000 : 25.159268197880493
----------------Column ck_000 END ---------------
```

In [ ]:

```
# 91,92...100 percentile values
def print_percentiles2(df, cols):
    for col in cols:
        for i in range(90,101):
            print('{}th Percentile value of column {} : {}'.format(i, col,np.percentile(df[col],i)))
        print('----------------Column {} END ---------------'.format(col))
```

In [ ]:
```
print_percentiles2(train_data, final_cols)
```

```
90th Percentile value of column bh_000 : 0.29732289517991134
91th Percentile value of column bh_000 : 0.36867609639989735
92th Percentile value of column bh_000 : 0.4536633778873521
93th Percentile value of column bh_000 : 0.5658037805414229
94th Percentile value of column bh_000 : 0.7252381750776817
95th Percentile value of column bh_000 : 0.9994460662368209
96th Percentile value of column bh_000 : 1.4040111279898457
97th Percentile value of column bh_000 : 2.114346604263077
98th Percentile value of column bh_000 : 3.296667344301698
99th Percentile value of column bh_000 : 5.07856072467774
100th Percentile value of column bh_000 : 20.755802646098974
----------------Column bh_000 END ---------------
90th Percentile value of column cn_004 : 0.3627731402543856
91th Percentile value of column cn_004 : 0.4386926537076137
92th Percentile value of column cn_004 : 0.5330662216256509
93th Percentile value of column cn_004 : 0.6477886543403917
94th Percentile value of column cn_004 : 0.7892789271125847
95th Percentile value of column cn_004 : 0.99702291226148
96th Percentile value of column cn_004 : 1.312727857053626
97th Percentile value of column cn_004 : 1.73754476713479
98th Percentile value of column cn_004 : 2.691038564085921
99th Percentile value of column cn_004 : 4.825945917127057
100th Percentile value of column cn_004 : 50.4936236914768
----------------Column cn_004 END ---------------
90th Percentile value of column ee_000 : 0.2273512573997535
91th Percentile value of column ee_000 : 0.2807429691161639
92th Percentile value of column ee_000 : 0.34760545160013967
93th Percentile value of column ee_000 : 0.43985788699030476
94th Percentile value of column ee_000 : 0.5594366817313456
95th Percentile value of column ee_000 : 0.7334010831062217
96th Percentile value of column ee_000 : 1.004006772136586
97th Percentile value of column ee_000 : 1.4949557402061802
98th Percentile value of column ee_000 : 2.300458659184261
99th Percentile value of column ee_000 : 4.0720164105273495
100th Percentile value of column ee_000 : 30.90008002550293
----------------Column ee_000 END ---------------
90th Percentile value of column bj_000 : 0.1538706615122914
91th Percentile value of column bj_000 : 0.20255403673112987
92th Percentile value of column bj_000 : 0.26639645307118515
93th Percentile value of column bj_000 : 0.350283855820331
94th Percentile value of column bj_000 : 0.47100659902673075
95th Percentile value of column bj_000 : 0.6548538899096662
96th Percentile value of column bj_000 : 0.9624758263630135
```

```
97th Percentile value of column bj_000 : 1.561678161068446
98th Percentile value of column bj_000 : 2.5692455722639123
99th Percentile value of column bj_000 : 4.417380847566572
100th Percentile value of column bj_000 : 24.968469323630597
----------------Column bj_000 END ---------------
90th Percentile value of column cc_000 : 0.2888263648429468
91th Percentile value of column cc_000 : 0.34965122856897296
92th Percentile value of column cc_000 : 0.41291386000473473
93th Percentile value of column cc_000 : 0.5038356695507629
94th Percentile value of column cc_000 : 0.6599663190298674
95th Percentile value of column cc_000 : 0.8642459025341733
96th Percentile value of column cc_000 : 1.2071634276314303
97th Percentile value of column cc_000 : 1.9056514972434937
98th Percentile value of column cc_000 : 3.4198442091628523
99th Percentile value of column cc_000 : 5.408548056321577
100th Percentile value of column cc_000 : 15.46675452214507
----------------Column cc_000 END ---------------
90th Percentile value of column ee_005 : 0.3296634135964347
91th Percentile value of column ee_005 : 0.381448310805787
92th Percentile value of column ee_005 : 0.45222654528635
93th Percentile value of column ee_005 : 0.5314148755887245
94th Percentile value of column ee_005 : 0.6482864063544815
95th Percentile value of column ee_005 : 0.7894709500552451
96th Percentile value of column ee_005 : 0.9972628341645652
97th Percentile value of column ee_005 : 1.3626663436500621
98th Percentile value of column ee_005 : 1.939115141531166
99th Percentile value of column ee_005 : 3.7348287762481456
100th Percentile value of column ee_005 : 51.16220916719786
----------------Column ee_005 END ---------------
90th Percentile value of column ay_008 : 0.21801381876412257
91th Percentile value of column ay_008 : 0.2730100137497781
92th Percentile value of column ay_008 : 0.3406081749892388
93th Percentile value of column ay_008 : 0.43233391201691057
94th Percentile value of column ay_008 : 0.5611872957646807
95th Percentile value of column ay_008 : 0.7414226772838609
96th Percentile value of column ay_008 : 1.0423098927181238
97th Percentile value of column ay_008 : 1.5302918228968168
98th Percentile value of column ay_008 : 2.3804883538513173
99th Percentile value of column ay_008 : 4.540186594714076
100th Percentile value of column ay_008 : 26.07780703251284
----------------Column ay_008 END ---------------
90th Percentile value of column aq_000 : 0.2852205208812105
91th Percentile value of column aq_000 : 0.34385517777212415
92th Percentile value of column aq_000 : 0.4145348298315653
93th Percentile value of column aq_000 : 0.5167733669124651
```

```
94th Percentile value of column aq_000 : 0.6559125944972407
95th Percentile value of column aq_000 : 0.8493946567756809
96th Percentile value of column aq_000 : 1.194253697129617
97th Percentile value of column aq_000 : 1.796407475821092
98th Percentile value of column aq_000 : 2.7501642799193227
99th Percentile value of column aq_000 : 4.869935019124949
100th Percentile value of column aq_000 : 19.99405062770798
----------------Column aq_000 END ---------------
90th Percentile value of column ba_005 : 0.39179999339815974
91th Percentile value of column ba_005 : 0.45319241300863156
92th Percentile value of column ba_005 : 0.53183135765414
93th Percentile value of column ba_005 : 0.6286483557755871
94th Percentile value of column ba_005 : 0.7556801417164705
95th Percentile value of column ba_005 : 0.9358793014105291
96th Percentile value of column ba_005 : 1.1713763176839433
97th Percentile value of column ba_005 : 1.5521996292380196
98th Percentile value of column ba_005 : 2.2960016915016874
99th Percentile value of column ba_005 : 4.41292477441283
100th Percentile value of column ba_005 : 37.556771702122504
----------------Column ba_005 END ---------------
90th Percentile value of column cs_004 : 0.036248922453680l8
91th Percentile value of column cs_004 : 0.0696972124335054
92th Percentile value of column cs_004 : 0.11974951599490466
93th Percentile value of column cs_004 : 0.18574736234080136
94th Percentile value of column cs_004 : 0.2832809469169975
95th Percentile value of column cs_004 : 0.4394575179250489
96th Percentile value of column cs_004 : 0.6800494245593779
97th Percentile value of column cs_004 : 1.1144646489134646
98th Percentile value of column cs_004 : 2.056183558213869
99th Percentile value of column cs_004 : 4.039976020551333
100th Percentile value of column cs_004 : 36.07982538507858
----------------Column cs_004 END ---------------
90th Percentile value of column ci_000 : 0.36684363959697874
91th Percentile value of column ci_000 : 0.43212900731752985
92th Percentile value of column ci_000 : 0.5508253091445097
93th Percentile value of column ci_000 : 0.694725539139558
94th Percentile value of column ci_000 : 0.869850948815383
95th Percentile value of column ci_000 : 1.1247279748848074
96th Percentile value of column ci_000 : 1.5086597593372248
97th Percentile value of column ci_000 : 2.2886574393034502
98th Percentile value of column ci_000 : 3.557604908098832
99th Percentile value of column ci_000 : 5.355472938344671
100th Percentile value of column ci_000 : 16.50186736995178
----------------Column ci_000 END ---------------
90th Percentile value of column ap_000 : 0.20298132371695757
```

```
91th Percentile value of column ap_000 : 0.26807577047964526
92th Percentile value of column ap_000 : 0.34426041228499815
93th Percentile value of column ap_000 : 0.4418010241789896
94th Percentile value of column ap_000 : 0.5831663203064328
95th Percentile value of column ap_000 : 0.7948435732939998
96th Percentile value of column ap_000 : 1.1844480345656223
97th Percentile value of column ap_000 : 1.8735861677490577
98th Percentile value of column ap_000 : 2.993428476516791
99th Percentile value of column ap_000 : 4.88962472357661
100th Percentile value of column ap_000 : 25.040062911918795
----------------Column ap_000 END ---------------
90th Percentile value of column cn_001 : -0.08205146395283505
91th Percentile value of column cn_001 : -0.07828153556420102
92th Percentile value of column cn_001 : -0.0722921056179452
93th Percentile value of column cn_001 : -0.06265271591676513
94th Percentile value of column cn_001 : -0.0450827939558278
95th Percentile value of column cn_001 : -0.01611699746837359
96th Percentile value of column cn_001 : 0.05616434984368616
97th Percentile value of column cn_001 : 0.20576138988197937
98th Percentile value of column cn_001 : 0.5984029340531003
99th Percentile value of column cn_001 : 1.8861336458578524
100th Percentile value of column cn_001 : 59.294021962956734
----------------Column cn_001 END ---------------
90th Percentile value of column ee_006 : 0.26306983882483936
91th Percentile value of column ee_006 : 0.326210542817941
92th Percentile value of column ee_006 : 0.3966400443173621
93th Percentile value of column ee_006 : 0.5010700486422798
94th Percentile value of column ee_006 : 0.6355891785909624
95th Percentile value of column ee_006 : 0.8120712611698477
96th Percentile value of column ee_006 : 1.0900061697627637
97th Percentile value of column ee_006 : 1.6077162547817647
98th Percentile value of column ee_006 : 2.5142038696852893
99th Percentile value of column ee_006 : 4.3017011633448865
100th Percentile value of column ee_006 : 29.411116554427824
----------------Column ee_006 END ---------------
90th Percentile value of column an_000 : 0.41735302124067397
91th Percentile value of column an_000 : 0.4936921342835373
92th Percentile value of column an_000 : 0.605704919547059
93th Percentile value of column an_000 : 0.7608407260049038
94th Percentile value of column an_000 : 0.940271106948819
95th Percentile value of column an_000 : 1.1777588297138437
96th Percentile value of column an_000 : 1.5760810457283743
97th Percentile value of column an_000 : 2.3475420265816545
98th Percentile value of column an_000 : 3.6154040650012793
99th Percentile value of column an_000 : 5.36448147131257
```

```
100th Percentile value of column an_000 : 17.73099214949411
----------------Column an_000 END ---------------
90th Percentile value of column ao_000 : 0.4198883740073929
91th Percentile value of column ao_000 : 0.4989449354760282
92th Percentile value of column ao_000 : 0.6175493438386889
93th Percentile value of column ao_000 : 0.7601607642072491
94th Percentile value of column ao_000 : 0.9343046421701668
95th Percentile value of column ao_000 : 1.1750173897604619
96th Percentile value of column ao_000 : 1.5769466123402045
97th Percentile value of column ao_000 : 2.3351593108508624
98th Percentile value of column ao_000 : 3.5540185022420516
99th Percentile value of column ao_000 : 5.3092538938095215
100th Percentile value of column ao_000 : 17.564270680432145
----------------Column ao_000 END ---------------
90th Percentile value of column bb_000 : 0.3411511243412503
91th Percentile value of column bb_000 : 0.4246408420912744
92th Percentile value of column bb_000 : 0.5432983179795196
93th Percentile value of column bb_000 : 0.6721900284616054
94th Percentile value of column bb_000 : 0.8468090604858626
95th Percentile value of column bb_000 : 1.0724729030198283
96th Percentile value of column bb_000 : 1.4980876770257665
97th Percentile value of column bb_000 : 2.3183253078773416
98th Percentile value of column bb_000 : 3.5690597334764114
99th Percentile value of column bb_000 : 5.397285370528981
100th Percentile value of column bb_000 : 17.392786516951077
----------------Column bb_000 END ---------------
90th Percentile value of column ck_000 : 0.25075668868698614
91th Percentile value of column ck_000 : 0.3113560814449297
92th Percentile value of column ck_000 : 0.3918339170007585
93th Percentile value of column ck_000 : 0.49552866377913896
94th Percentile value of column ck_000 : 0.6439504475865652
95th Percentile value of column ck_000 : 0.8745845467648632
96th Percentile value of column ck_000 : 1.2392453411129765
97th Percentile value of column ck_000 : 1.82381571326412
98th Percentile value of column ck_000 : 2.716146834548527
99th Percentile value of column ck_000 : 4.509354578108089
100th Percentile value of column ck_000 : 25.159268197880493
----------------Column ck_000 END ---------------
```

We need to remove the outliers which are impacting EDA plot results. As we can see from the above numbers, the difference between 99th and 100th percentile values are more as compared to others. So we can keep the values till 99th percentile values and discard others. But the dataset contains multiple columns, we cant simply discard rows based percentile value of a single column. So we can replace the values which are more than 99th percentile with the 99th percentile value.

```
In [ ]: def manage_outliers(df):
            for col in df.columns:
                if col != 'class':
                    if (((df[col].dtype)=='float64') | ((df[col].dtype)=='int64')):
                        # Replces values from 99-100 percentile values with 99th percentile as 100th percentile values are ou
                        cut_off = np.percentile(df[col], 99)
                        df[col][df[col] >= cut_off] = cut_off
                else:
                    df[col]=df[col]
            return df
```

```
In [ ]: train_data = manage_outliers(train_data.copy())
```

```
In [ ]: print_percentiles1(train_data, final_cols)
```

```
10th Percentile value of column bh_000 : -0.3790293441357247
20th Percentile value of column bh_000 : -0.3765462900319117
30th Percentile value of column bh_000 : -0.3724122691037549
40th Percentile value of column bh_000 : -0.3321022631205778
50th Percentile value of column bh_000 : -0.20639104524774682
60th Percentile value of column bh_000 : -0.1584337609342094
70th Percentile value of column bh_000 : -0.10146087698629573
80th Percentile value of column bh_000 : 0.003390044867587177
90th Percentile value of column bh_000 : 0.29732289517991134
100th Percentile value of column bh_000 : 5.07856072467774
----------------Column bh_000 END ---------------
10th Percentile value of column cn_004 : -0.3808408267119443
20th Percentile value of column cn_004 : -0.3788635531092587
30th Percentile value of column cn_004 : -0.36942079879778394
40th Percentile value of column cn_004 : -0.33731385279908177
50th Percentile value of column cn_004 : -0.22630560365708824
60th Percentile value of column cn_004 : -0.15051462796284432
70th Percentile value of column cn_004 : -0.07029146859828461
80th Percentile value of column cn_004 : 0.04082987029111493
90th Percentile value of column cn_004 : 0.3627731402543856
100th Percentile value of column cn_004 : 4.825945917127057
----------------Column cn_004 END ---------------
10th Percentile value of column ee_000 : -0.30084669036251915
20th Percentile value of column ee_000 : -0.2984681903503712
30th Percentile value of column ee_000 : -0.29203827564242
40th Percentile value of column ee_000 : -0.2575395806728418
50th Percentile value of column ee_000 : -0.19450362940789226
60th Percentile value of column ee_000 : -0.1559810673650316
```

```
70th Percentile value of column ee_000 : -0.10601126139583036
80th Percentile value of column ee_000 : -0.015873692773525128
90th Percentile value of column ee_000 : 0.2273512573997535
100th Percentile value of column ee_000 : 4.0720164105273495
----------------Column ee_000 END ---------------
10th Percentile value of column bj_000 : -0.2781424131357906
20th Percentile value of column bj_000 : -0.2762908816858898
30th Percentile value of column bj_000 : -0.27231511260354463
40th Percentile value of column bj_000 : -0.2464550587978459
50th Percentile value of column bj_000 : -0.19442382324215385
60th Percentile value of column bj_000 : -0.16434002542050963
70th Percentile value of column bj_000 : -0.12522048914568223
80th Percentile value of column bj_000 : -0.05735683797187588
90th Percentile value of column bj_000 : 0.1538706615122914
100th Percentile value of column bj_000 : 4.417380847566572
----------------Column bj_000 END ---------------
10th Percentile value of column cc_000 : -0.3952838731902859
20th Percentile value of column cc_000 : -0.3922710599320991
30th Percentile value of column cc_000 : -0.38458230311475994
40th Percentile value of column cc_000 : -0.32365497500403706
50th Percentile value of column cc_000 : -0.17105895400046486
60th Percentile value of column cc_000 : -0.13146852432536543
70th Percentile value of column cc_000 : -0.08901742599596568
80th Percentile value of column cc_000 : 0.007682669948012241
90th Percentile value of column cc_000 : 0.2888263648429468
100th Percentile value of column cc_000 : 5.408548056321577
----------------Column cc_000 END ---------------
10th Percentile value of column ee_005 : -0.3512583001986739
20th Percentile value of column ee_005 : -0.3506110993018916
30th Percentile value of column ee_005 : -0.3408781521127268
40th Percentile value of column ee_005 : -0.29550825709597606
50th Percentile value of column ee_005 : -0.1808837403971903
60th Percentile value of column ee_005 : -0.1070249875018211
70th Percentile value of column ee_005 : -0.038104908411942
80th Percentile value of column ee_005 : 0.07218113797376878
90th Percentile value of column ee_005 : 0.3296634135964347
100th Percentile value of column ee_005 : 3.7348287762481456
----------------Column ee_005 END ---------------
10th Percentile value of column ay_008 : -0.2619184944724901
20th Percentile value of column ay_008 : -0.26080199579166574
30th Percentile value of column ay_008 : -0.2586058758343291
40th Percentile value of column ay_008 : -0.253100260734825
50th Percentile value of column ay_008 : -0.23819584467457927
60th Percentile value of column ay_008 : -0.2100228176333392
70th Percentile value of column ay_008 : -0.15436584707023077
```

```
80th Percentile value of column ay_008 : -0.04736011374066457
90th Percentile value of column ay_008 : 0.21801381876412257
100th Percentile value of column ay_008 : 4.540186594714076
----------------Column ay_008 END ---------------
10th Percentile value of column aq_000 : -0.3493890519687957
20th Percentile value of column aq_000 : -0.3479393282078672
30th Percentile value of column aq_000 : -0.344924800506042
40th Percentile value of column aq_000 : -0.3143362339978703
50th Percentile value of column aq_000 : -0.20773696986507606
60th Percentile value of column aq_000 : -0.15667893416727308
70th Percentile value of column aq_000 : -0.09458625316313211
80th Percentile value of column aq_000 : 0.007186965248170011
90th Percentile value of column aq_000 : 0.2852205208812105
100th Percentile value of column aq_000 : 4.869935019124949
----------------Column aq_000 END ---------------
10th Percentile value of column ba_005 : -0.37058904077824417
20th Percentile value of column ba_005 : -0.3702889171549093
30th Percentile value of column ba_005 : -0.36939644427499246
40th Percentile value of column ba_005 : -0.3518684348732271
50th Percentile value of column ba_005 : -0.20484576979156213
60th Percentile value of column ba_005 : -0.14182612728330957
70th Percentile value of column ba_005 : -0.06547665200444075
80th Percentile value of column ba_005 : 0.07011090812866205
90th Percentile value of column ba_005 : 0.39179999339815974
100th Percentile value of column ba_005 : 4.41292477441283
----------------Column ba_005 END ---------------
10th Percentile value of column cs_004 : -0.21325430189601258
20th Percentile value of column cs_004 : -0.21268610297525706
30th Percentile value of column cs_004 : -0.21076140185290257
40th Percentile value of column cs_004 : -0.19678681119505798
50th Percentile value of column cs_004 : -0.16930130346965078
60th Percentile value of column cs_004 : -0.1523571462338098
70th Percentile value of column cs_004 : -0.12983814310226255
80th Percentile value of column cs_004 : -0.09053690973106214
90th Percentile value of column cs_004 : 0.03624892245368018
100th Percentile value of column cs_004 : 4.039976020551333
----------------Column cs_004 END ---------------
10th Percentile value of column ci_000 : -0.4162655593825422
20th Percentile value of column ci_000 : -0.41405923651479054
30th Percentile value of column ci_000 : -0.4070204054266747
40th Percentile value of column ci_000 : -0.3514599818384025
50th Percentile value of column ci_000 : -0.1936128039202838
60th Percentile value of column ci_000 : -0.14814753520932397
70th Percentile value of column ci_000 : -0.1101905657306674
80th Percentile value of column ci_000 : 0.00323935231073315
```

```
90th Percentile value of column ci_000 : 0.36684363959697874
100th Percentile value of column ci_000 : 5.355472938344671
----------------Column ci_000 END ---------------
10th Percentile value of column ap_000 : -0.3212142730856814
20th Percentile value of column ap_000 : -0.3185806594844594
30th Percentile value of column ap_000 : -0.3119733877143347
40th Percentile value of column ap_000 : -0.2694450844795145
50th Percentile value of column ap_000 : -0.2082797015041634
60th Percentile value of column ap_000 : -0.17161252291364476
70th Percentile value of column ap_000 : -0.12446673866934652
80th Percentile value of column ap_000 : -0.041506803670515874
90th Percentile value of column ap_000 : 0.20298132371695757
100th Percentile value of column ap_000 : 4.88962472357661
----------------Column ap_000 END ---------------
10th Percentile value of column cn_001 : -0.08879049327479388
20th Percentile value of column cn_001 : -0.08879049327479388
30th Percentile value of column cn_001 : -0.08879049327479388
40th Percentile value of column cn_001 : -0.08879049327479388
50th Percentile value of column cn_001 : -0.08879049327479388
60th Percentile value of column cn_001 : -0.08879049327479388
70th Percentile value of column cn_001 : -0.08879049327479388
80th Percentile value of column cn_001 : -0.08879049327479388
90th Percentile value of column cn_001 : -0.08205146395283505
100th Percentile value of column cn_001 : 1.8861336458578524
----------------Column cn_001 END ---------------
10th Percentile value of column ee_006 : -0.31064288755452263
20th Percentile value of column ee_006 : -0.3105488545703816
30th Percentile value of column ee_006 : -0.30895593581903286
40th Percentile value of column ee_006 : -0.2992272832798038
50th Percentile value of column ee_006 : -0.2237376036113997
60th Percentile value of column ee_006 : -0.1477202108359962
70th Percentile value of column ee_006 : -0.09120939642274077
80th Percentile value of column ee_006 : -0.0009791061603916069
90th Percentile value of column ee_006 : 0.26306983882483936
100th Percentile value of column ee_006 : 4.3017011633448865
----------------Column ee_006 END ---------------
10th Percentile value of column an_000 : -0.4427608837961114
20th Percentile value of column an_000 : -0.438868117693785
30th Percentile value of column an_000 : -0.4303648158525458
40th Percentile value of column an_000 : -0.3587232307627656
50th Percentile value of column an_000 : -0.19688785479948112
60th Percentile value of column an_000 : -0.14725034513206148
70th Percentile value of column an_000 : -0.09721108678763131
80th Percentile value of column an_000 : 0.0279758706145501
90th Percentile value of column an_000 : 0.41735302124067397
```

```
100th Percentile value of column an_000 : 5.36448147131257
----------------Column an_000 END ---------------
10th Percentile value of column ao_000 : -0.4388363705636256
20th Percentile value of column ao_000 : -0.43494827406501074
30th Percentile value of column ao_000 : -0.4258514242216618
40th Percentile value of column ao_000 : -0.3539612869910947
50th Percentile value of column ao_000 : -0.19824653912091078
60th Percentile value of column ao_000 : -0.14812734955341772
70th Percentile value of column ao_000 : -0.09868985206228077
80th Percentile value of column ao_000 : 0.02625634352120773
90th Percentile value of column ao_000 : 0.4198883740073929
100th Percentile value of column ao_000 : 5.3092538938095215
----------------Column ao_000 END ---------------
10th Percentile value of column bb_000 : -0.41341292050461587
20th Percentile value of column bb_000 : -0.4092000998512446
30th Percentile value of column bb_000 : -0.4020876848414522
40th Percentile value of column bb_000 : -0.34073124741740174
50th Percentile value of column bb_000 : -0.19779469128962315
60th Percentile value of column bb_000 : -0.15728690551027075
70th Percentile value of column bb_000 : -0.10751700832241343
80th Percentile value of column bb_000 : 0.0035435292440028653
90th Percentile value of column bb_000 : 0.3411511243412503
100th Percentile value of column bb_000 : 5.397285370528981
----------------Column bb_000 END ---------------
10th Percentile value of column ck_000 : -0.3252629737723142
20th Percentile value of column ck_000 : -0.32261456996246013
30th Percentile value of column ck_000 : -0.31679852546793436
40th Percentile value of column ck_000 : -0.2779428069633934
50th Percentile value of column ck_000 : -0.2121833967881883
60th Percentile value of column ck_000 : -0.17149567216143702
70th Percentile value of column ck_000 : -0.11748506119351207
80th Percentile value of column ck_000 : -0.021519157283161 74
90th Percentile value of column ck_000 : 0.25075668868698614
100th Percentile value of column ck_000 : 4.509354578108089
----------------Column ck_000 END ---------------
```

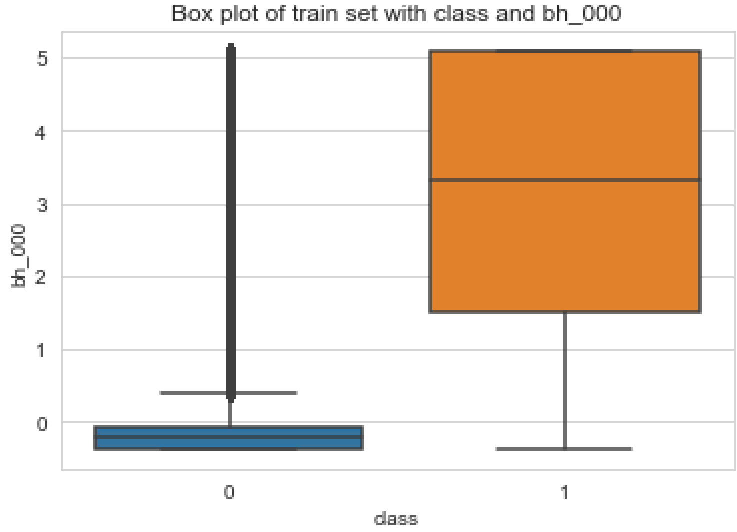
Now we dont have outliers in our dataset.

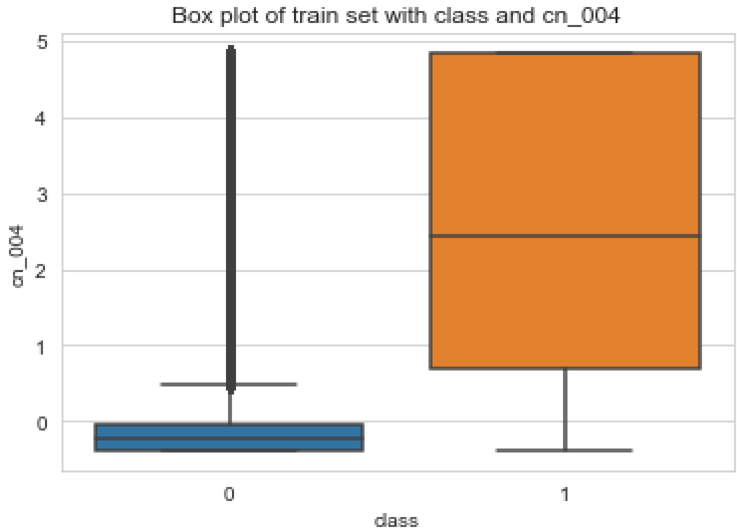
EDA

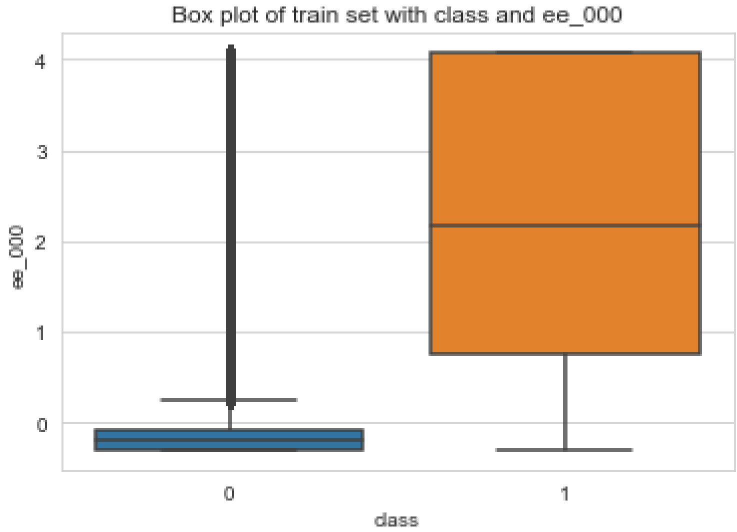
Now we have 18 columns which got determined by phi_k correlation. We can do univariate analysis with these columns.

Box-plot

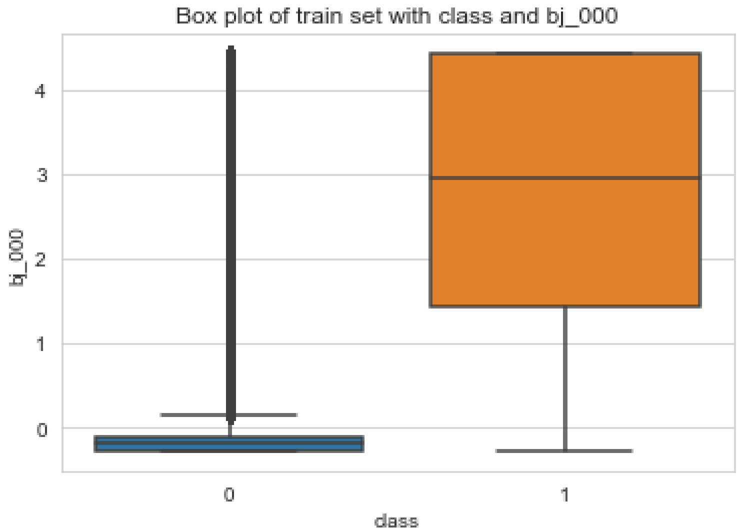
```
In [ ]: for col in final_cols:
            sns.boxplot(data = train_data, x= "class", y= col)
            plt.title("Box plot of train set with class and {}".format(col))
            plt.show()
```



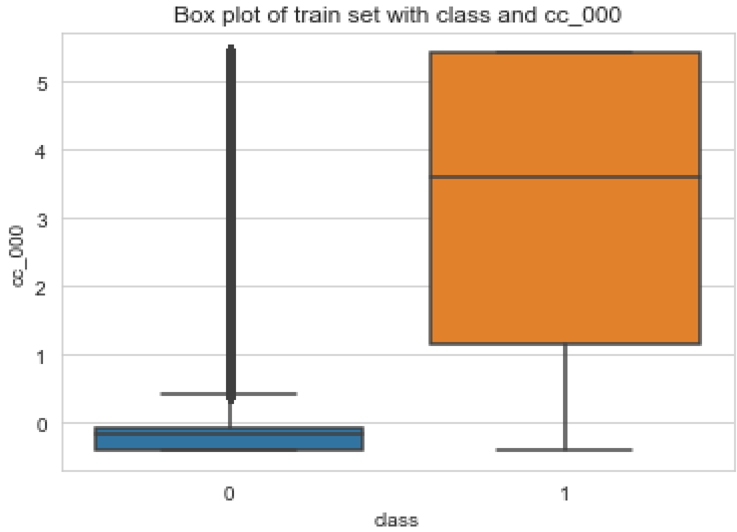


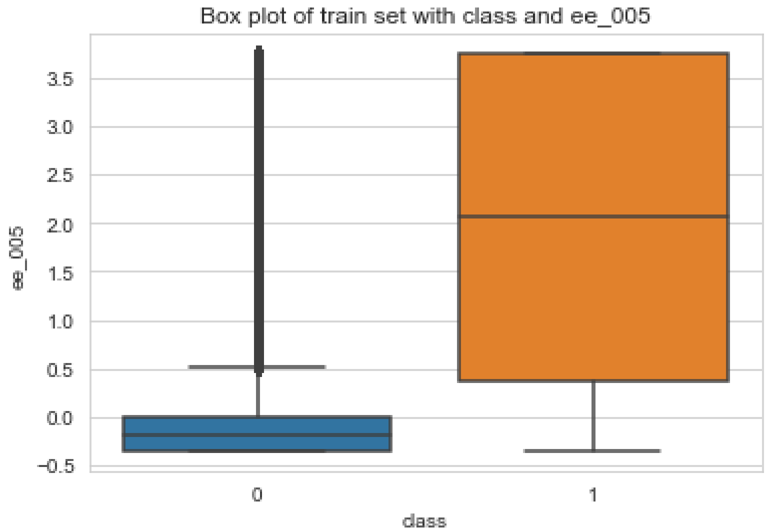


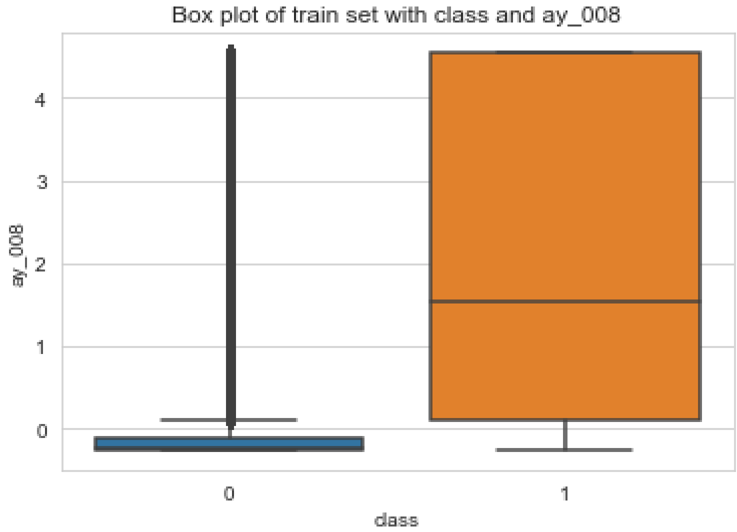
Box plot of train set with class and ee_000
ee_000
4
3
2
1
0
0
1
class

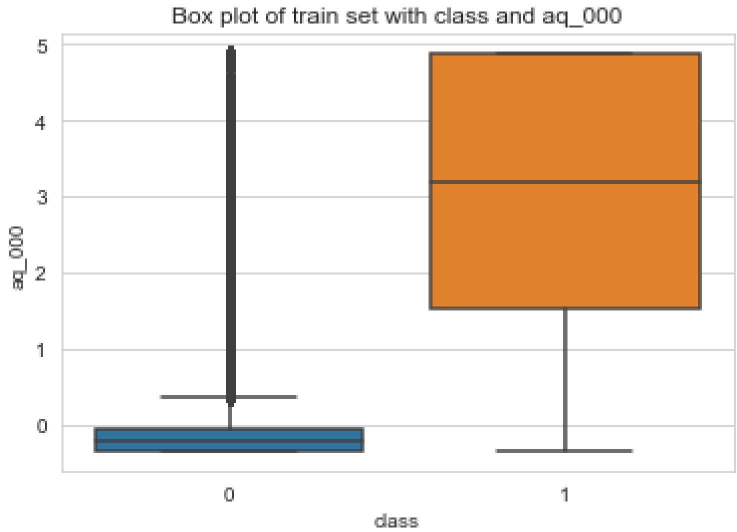


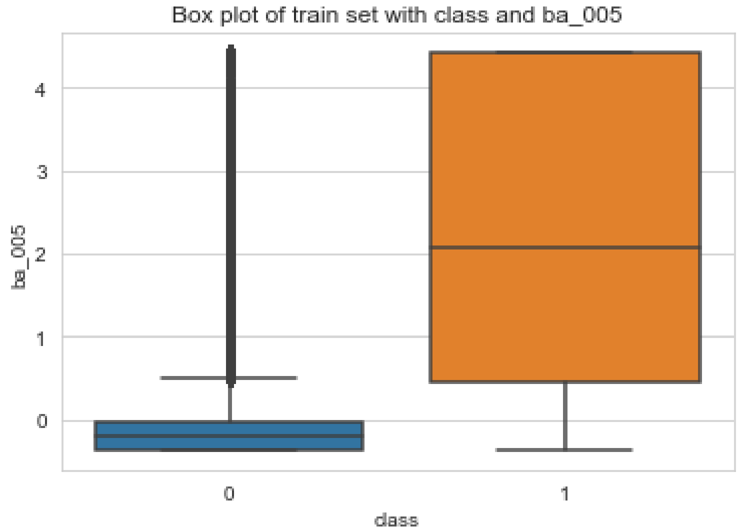
Box plot of train set with class and bj_000
bj_000
4
3
2
1
0
0
1
class

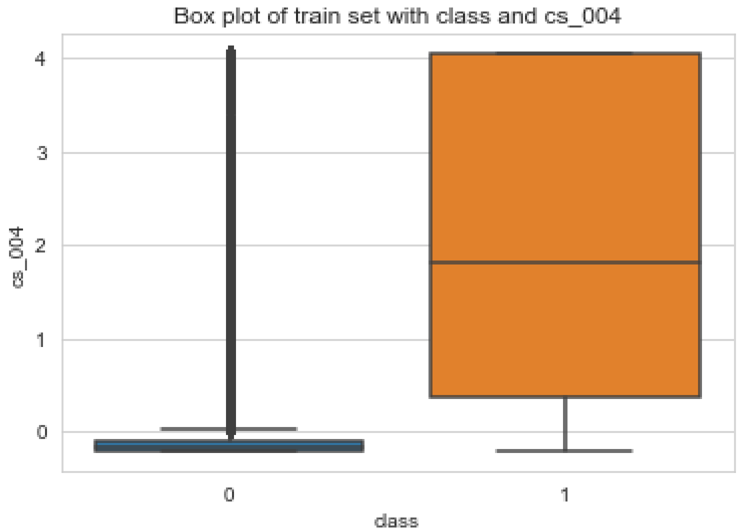

Box plot of train set with class and cc_000
cc_000
0
1
2
3
4
5
0
1
class

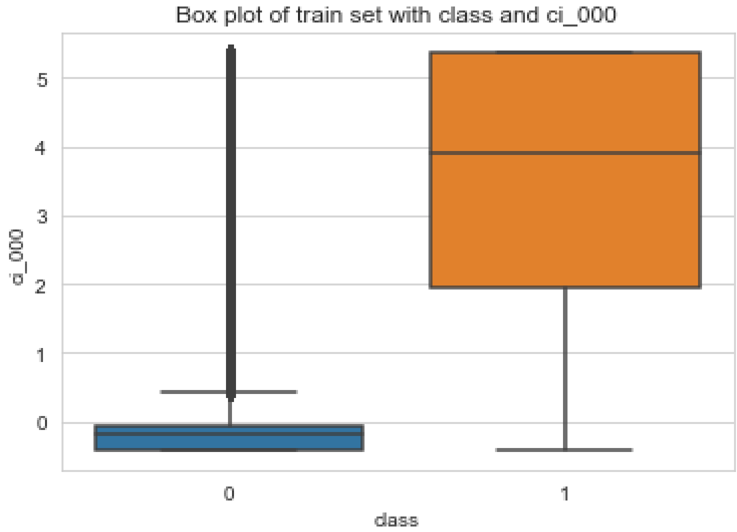

Box plot of train set with class and ee_005
ee_005
−0.5
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
0
1
class

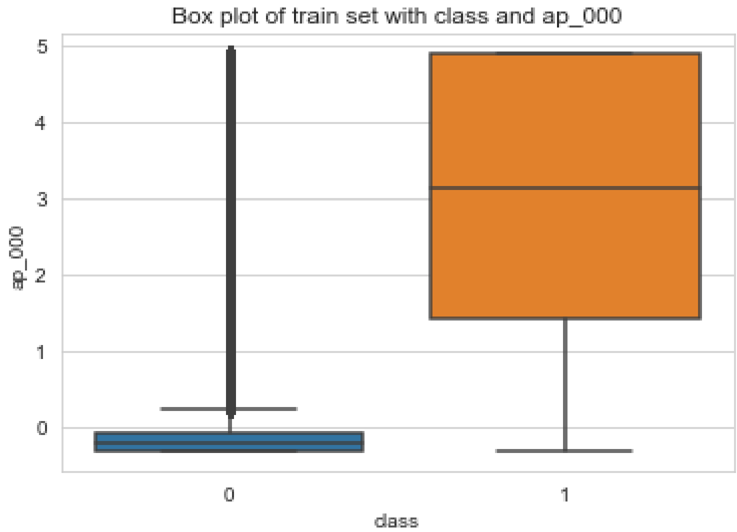

Box plot of train set with class and ay_008
ay_008
4
3
2
1
0
0
1
class

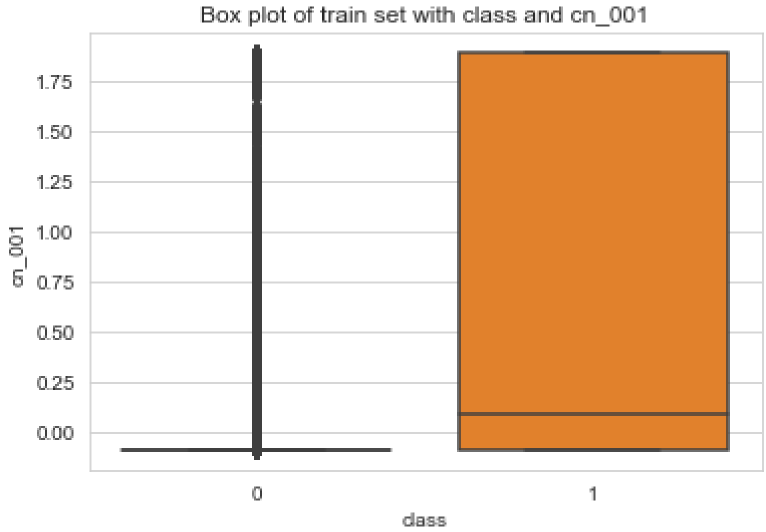

Box plot of train set with class and aq_000
aq_000
5
4
3
2
1
0
0
1
class

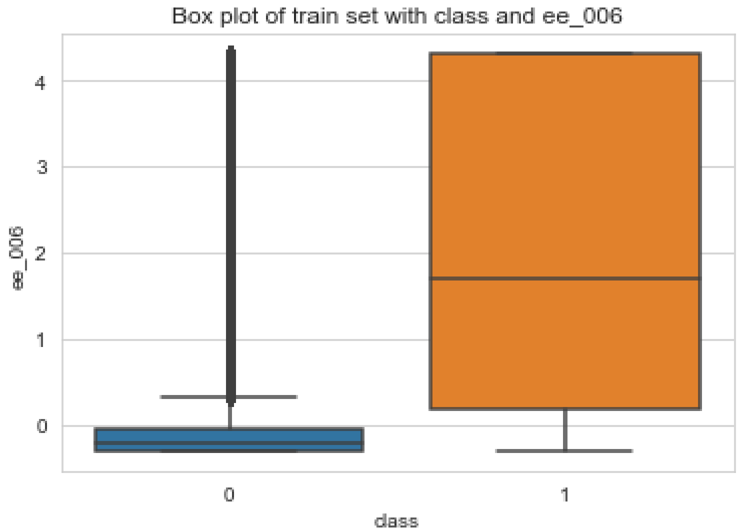

Box plot of train set with class and ba_005
ba_005
4
3
2
1
0
0
1
class

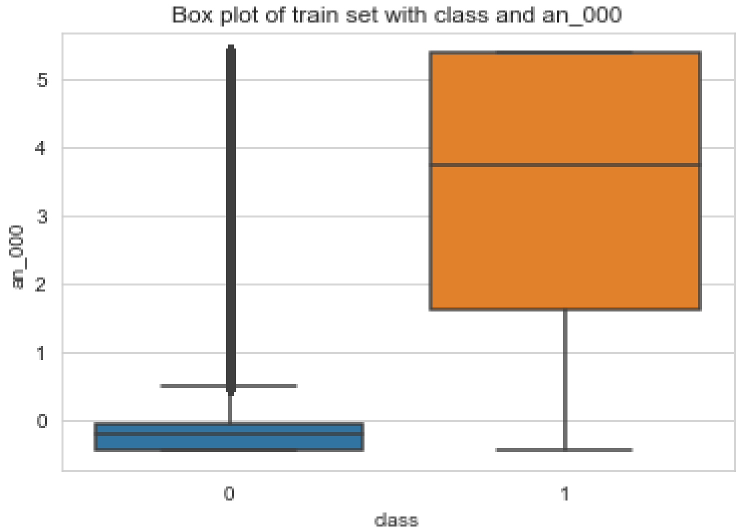

Box plot of train set with class and cs_004
cs_004
4
3
2
1
0
0
1
class

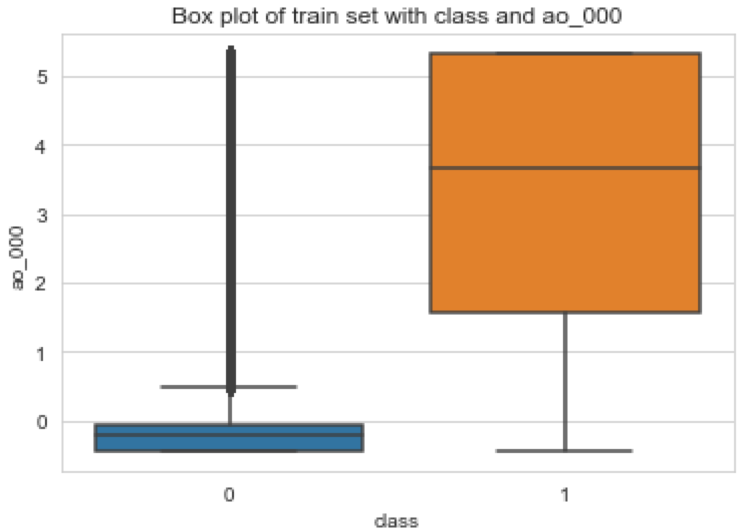

Box plot of train set with class and ci_000
ci_000
5
4
3
2
1
0
0
1
class

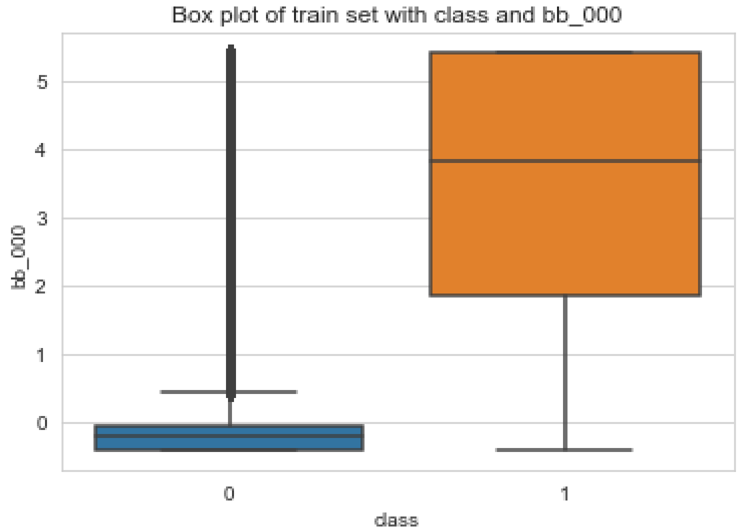

Box plot of train set with class and ap_000
ap_000
5
4
3
2
1
0
0
1
class

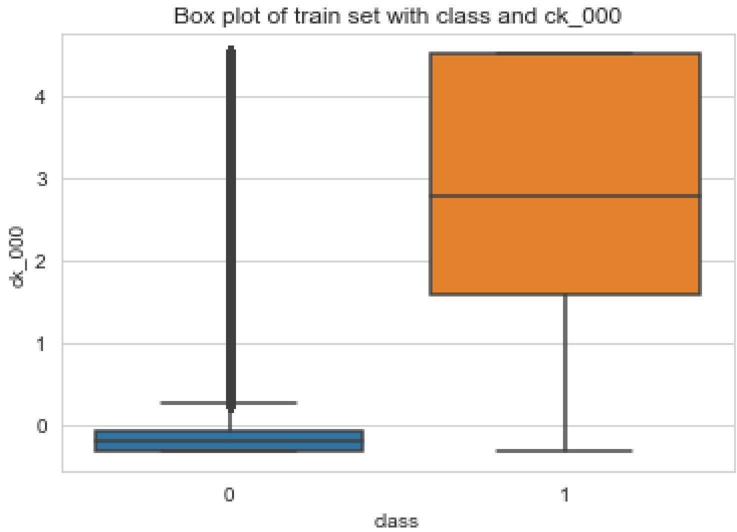

Box plot of train set with class and cn_001
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
cn_001
0
1
class


Box plot of train set with class and ee_006
4
3
2
1
0
ee_006
0
1
class


Box plot of train set with class and an_000
an_000
class
0
1
2
3
4
5


Box plot of train set with class and ao_000
ao_000
class
0
1
2
3
4
5





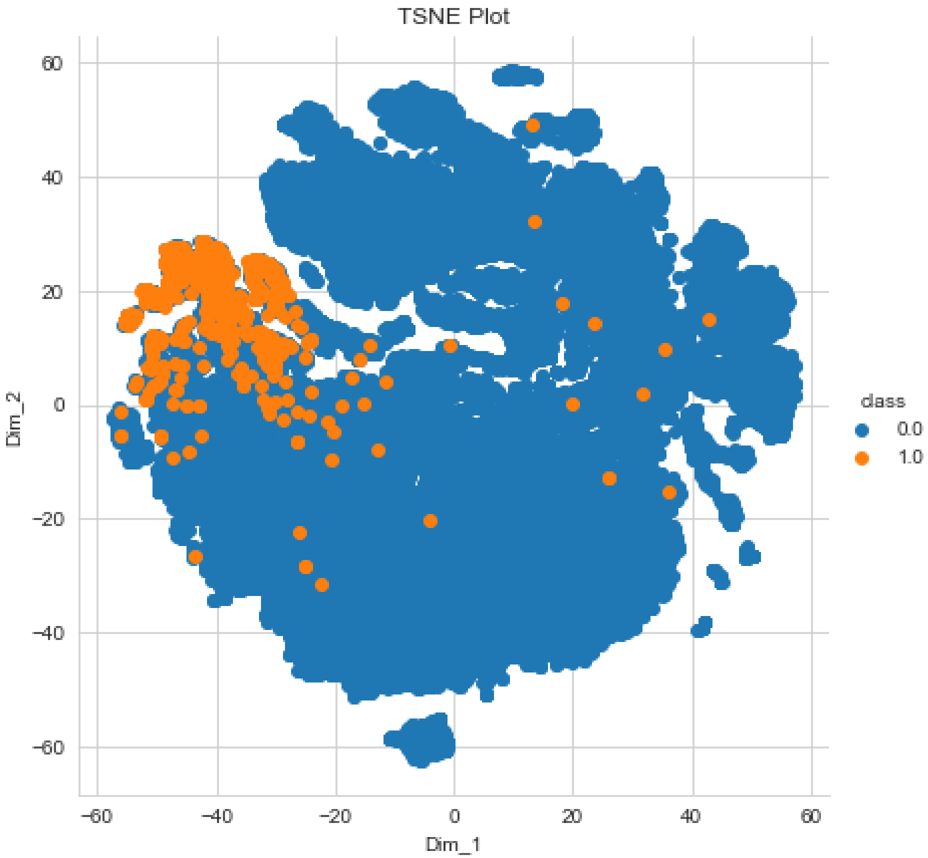
We can see from the above box-plots, each of these important features (high correlation with class label) can easily separate both classes.

t-SNE

EDA with tSNE to check whether the data points are separable in space or not.

In [ ]:
```
tsne = TSNE(n_components=2, random_state=0, perplexity=50)
```

```
tsne_data = tsne.fit_transform(train_data.loc[:, final_cols])

# creating a new data fram which help us in ploting the result data
tsne_data = np.vstack((tsne_data.T, train_data['class'].values)).T
tsne_df = pd.DataFrame(data=tsne_data, columns=("Dim_1", "Dim_2", "class"))

# Ploting the result of tsne
sns.FacetGrid(tsne_df, hue="class", size=6).map(plt.scatter, 'Dim_1', 'Dim_2').add_legend()
plt.title('TSNE Plot')
plt.show()
```



With the help of tSNE, we have ploted the data points in 2D to check whether the features useful in separating the classes or not. As we can see, due to class imbalance there are very few points for class-1 (orange colors) and all the class-0 points (blue points) are surrounding class-1 and even they all are overlapped. So with tSNE, we can conclude that by combining the existing features its difficult to separate the classes.

But after dealing with imbalance of dataset, it would be easy to separate as in box-plot, we have already seen that classes are getting separated for each feature values.

Models

We have numerical 169 dimensional data and its a classification problem. We can try out logistic regression, SVM, kernel SVM,random forest and XGBoost classifier with hyper-parameter tunning using GridSearchCV.

Metrics:

Here we can use recall to get best hyper-parameters as we are concerned about TPR, FPR values. In addition to confusion, precision and recall matrix, we can use F1-score and log-loss for model evaluation

In [ ]:

```
# Save dataframe to pickle
train_data.to_pickle("./train_data.pkl")
test_data.to_pickle("./test_data.pkl")
```

In [3]:

```
# Load dataframe from pickle
train_data = pd.read_pickle("./train_data.pkl")
test_data = pd.read_pickle("./test_data.pkl")

print(train_data.shape)
print(test_data.shape)
```

```
(60000, 170)
(16000, 170)
```

In [4]:

```
# Separate X and y values from train and test data
X_train = train_data.iloc[:, :-1].astype('float64')
X_test = test_data.iloc[:, :-1].astype('float64')
y_train = train_data.iloc[:, -1].astype('int64')
y_test = test_data.iloc[:, -1].astype('int64')

print(X_train.shape)
print(X_test.shape)
print(y_train.shape)
print(y_test.shape)
```

```
(60000, 169)
(16000, 169)
(60000,)
(16000,)
```

Confusion Matrix

In [5]:
```python
def plot_confusion_matrix(test_y, predict_y):
    C = confusion_matrix(test_y, predict_y)
    print("Number of misclassified points {}%".format(round(((len(test_y)-np.trace(C))/len(test_y)*100),2)))
    fp = (int)(C[0][1])
    fn = (int)(C[1][0])
    print('Number of False Positives: ', fp)
    print('Number of False Negatives: ', fn)
    cost = (10 * fp) + (500 * fn)
    print('Total Cost (cost1+cost2): ', cost)

    A =(((C.T)/(C.sum(axis=1))).T)

    B =(C/C.sum(axis=0))
    plt.figure(figsize=(20,4))

    labels = [0,1]
    # representing A in heatmap format
    cmap=sns.light_palette("blue")
    plt.subplot(1, 3, 1)
    sns.heatmap(C, annot=True, cmap=cmap, fmt=".3f", xticklabels=labels, yticklabels=labels)
    plt.xlabel('Predicted Class')
    plt.ylabel('Original Class')
    plt.title("Confusion matrix")

    plt.subplot(1, 3, 2)
    sns.heatmap(B, annot=True, cmap=cmap, fmt=".3f", xticklabels=labels, yticklabels=labels)
    plt.xlabel('Predicted Class')
    plt.ylabel('Original Class')
    plt.title("Precision matrix")

    plt.subplot(1, 3, 3)
    # representing B in heatmap format
    sns.heatmap(A, annot=True, cmap=cmap, fmt=".3f", xticklabels=labels, yticklabels=labels)
    plt.xlabel('Predicted Class')
    plt.ylabel('Original Class')
```

```
    plt.title("Recall matrix")

    plt.show()
```

Logistic Regression

In [6]:
```
clf = LogisticRegression(penalty='l2',class_weight='balanced', solver='liblinear', max_iter= 1000)
values = [10 ** x for x in range(-5, 4)]
parameters = {'C' : values}
grid_clf = GridSearchCV(clf, parameters, cv=3, scoring='f1_micro')
grid_clf.fit(X_train, y_train)

print(grid_clf.best_params_)
```

```
{'C': 10}
```

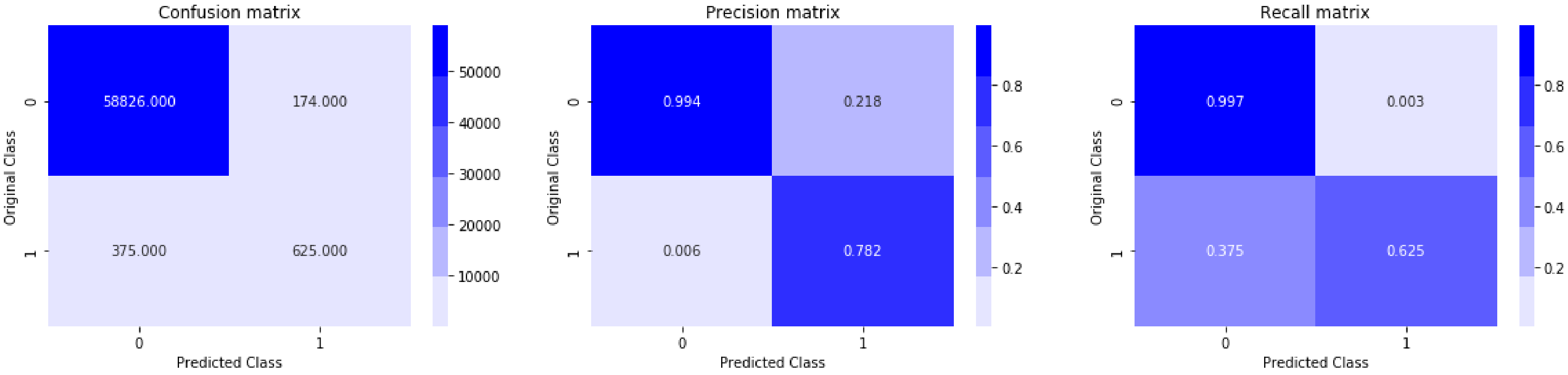
Best value for hyper-parameter 'C' is 10 for logistic regression.

In [7]:
```
c= grid_clf.best_params_.get('C')
```

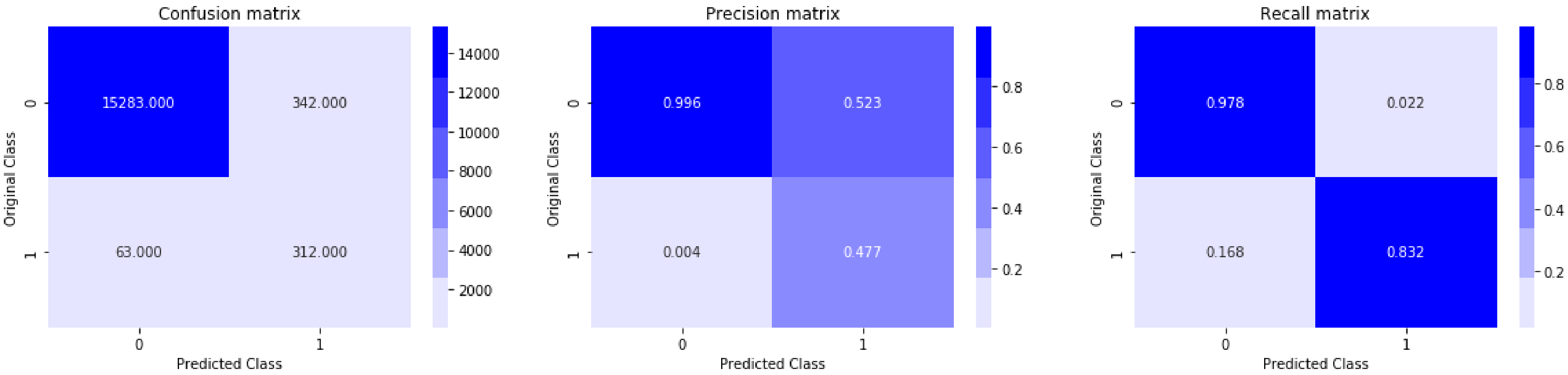
In [8]:
```
log_clf = LogisticRegression(penalty='l2',class_weight='balanced', C = c, solver='liblinear', max_iter= 1000)
log_clf.fit(X_train, y_train)
log_clf_cal = CalibratedClassifierCV(log_clf, method="sigmoid")
log_clf_cal.fit(X_train, y_train)
pred_y_train = log_clf_cal.predict(X_train)
pred_y_test = log_clf_cal.predict(X_test)
pickle.dump(log_clf_cal, open("lr.sav", 'wb'))

print("-"*50, "Train set", "-"*50)
print('Log-loss: ', log_loss(y_train, pred_y_train, labels=log_clf.classes_, eps=1e-15))
print('Micro F1-score: ', f1_score(y_train, pred_y_train, average='micro'))
plot_confusion_matrix(y_train, pred_y_train)
print("-"*50, "Test set", "-"*50)
print('Log-loss: ', log_loss(y_test, pred_y_test, labels=log_clf.classes_, eps=1e-15))
print('Micro F1-score: ', f1_score(y_test, pred_y_test, average='micro'))
plot_confusion_matrix(y_test, pred_y_test)
```

```
-------------------------------------------------- Train set --------------------------------------------------
Log-loss:  0.31603212284598126
Micro F1-score:  0.99085
Number of misclassified points 0.92%
Number of False Positives:  174
```

```
Number of False Negatives:  375
Total Cost (cost1+cost2):  189240
```




```
-------------------------------------------------- Test set --------------------------------------------------
Log-loss:  0.87427986888912479
Micro F1-score:  0.9746875
Number of misclassified points 2.53%
Number of False Positives:  342
Number of False Negatives:  63
Total Cost (cost1+cost2):  34920
```



SVC

In [ ]:
```python
clf = SVC(class_weight='balanced')
values = [10 ** x for x in range(-5, 4)]
```

```
parameters = {'C' : values}
grid_clf = GridSearchCV(clf, parameters, cv=3, scoring='f1_micro')
grid_clf.fit(X_train, y_train)

print(grid_clf.best_params_)
```

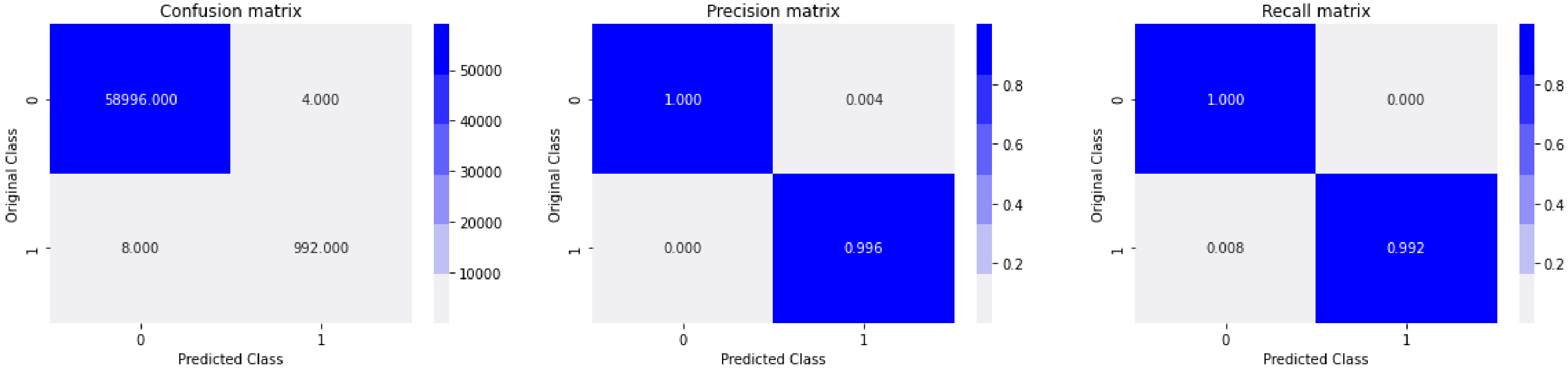
```
{'C': 10}
```

Best value for hyper-parameter 'C' is 10 for svc.

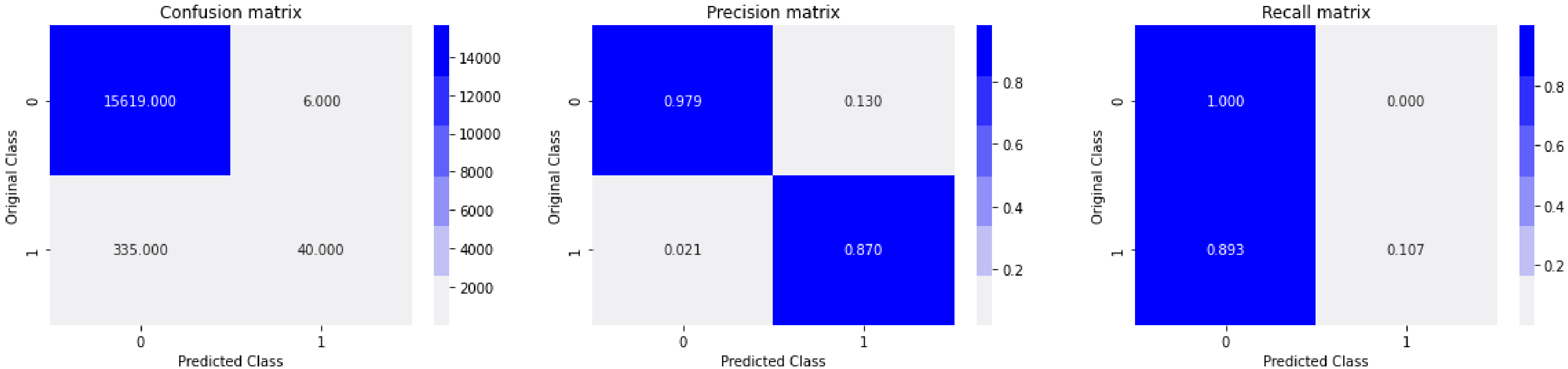
In [ ]:
```
c= grid_clf.best_params_.get('C')
```

In [ ]:
```
svc_clf = SVC(class_weight='balanced', C = c)
svc_clf.fit(X_train, y_train)
svc_clf_cal = CalibratedClassifierCV(svc_clf, method="sigmoid")
svc_clf_cal.fit(X_train, y_train)
pred_y_train = svc_clf_cal.predict(X_train)
pred_y_test = svc_clf_cal.predict(X_test)
pickle.dump(svc_clf_cal, open("svc.sav", 'wb'))

print("-"*50, "Train set", "-"*50)
print('Log-loss: ', log_loss(y_train, pred_y_train, labels=svc_clf.classes_, eps=1e-15))
print('Micro F1-score: ', f1_score(y_train, pred_y_train, average='micro'))
plot_confusion_matrix(y_train, pred_y_train)
print("-"*50, "Test set", "-"*50)
print('Log-loss: ', log_loss(y_test, pred_y_test, labels=svc_clf.classes_, eps=1e-15))
print('Micro F1-score: ', f1_score(y_test, pred_y_test, average='micro'))
plot_confusion_matrix(y_test, pred_y_test)
```

```
-------------------------------------------------- Train set --------------------------------------------------
Log-loss:  0.006907808585478481
Micro F1-score:  0.9998
Number of misclassified points 0.02%
Number of False Positives:  4
Number of False Negatives:  8
Total Cost (cost1+cost2):  4040
```




```
-------------------------------------------------- Test set --------------------------------------------------
Log-loss:  0.7361079717655712
Micro F1-score:  0.9786875
Number of misclassified points 2.13%
Number of False Positives:  6
Number of False Negatives:  335
Total Cost (cost1+cost2):  167560
```



Kernel SVC

In [ ]:
```python
clf = SVC(class_weight='balanced', kernel = 'rbf')
c_values = [10 ** x for x in range(-5, 4)]
parameters = {'C' : c_values}
```

```
grid_clf = GridSearchCV(clf, parameters, cv=3, scoring='f1_micro')
grid_clf.fit(X_train, y_train)

print(grid_clf.best_params_)
```

```
{'C': 10}
```

In [ ]:
```
c= grid_clf.best_params_.get('C')
```

In [6]:
```
clf = SVC(class_weight='balanced', C = c, kernel = 'rbf')
gamma_values = [10 ** x for x in range(-3, 3)]
parameters = {'gamma' : gamma_values}
grid_clf = GridSearchCV(clf, parameters, cv=3, scoring='f1_micro')
grid_clf.fit(X_train, y_train)

print(grid_clf.best_params_)
```

```
{'gamma': 0.01}
```

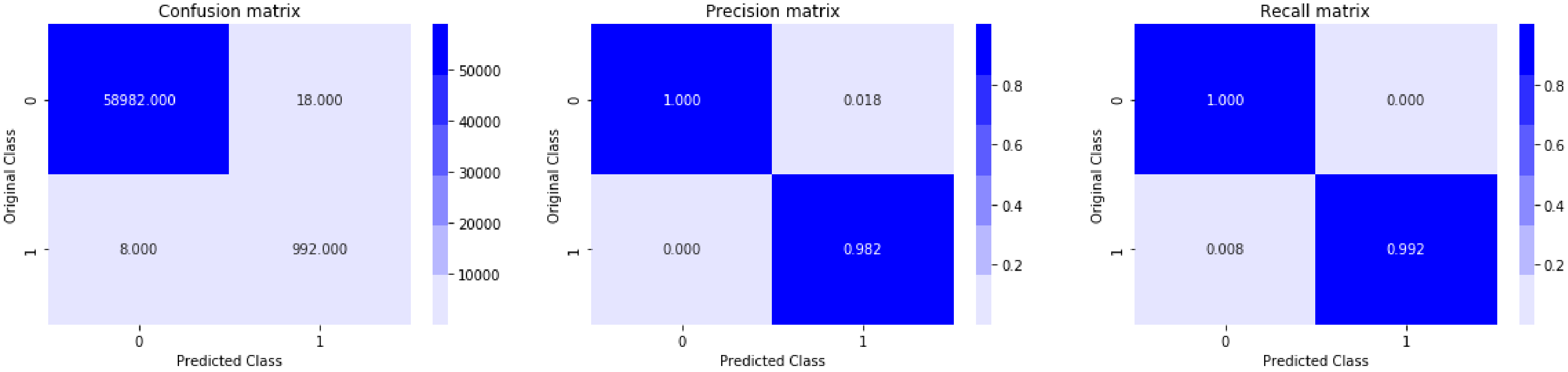
In [7]:
```
gamma= grid_clf.best_params_.get('gamma')
```

Best value for hyper-parameter 'C' and 'gamma' are 10 and 0.01 for kernel svc.

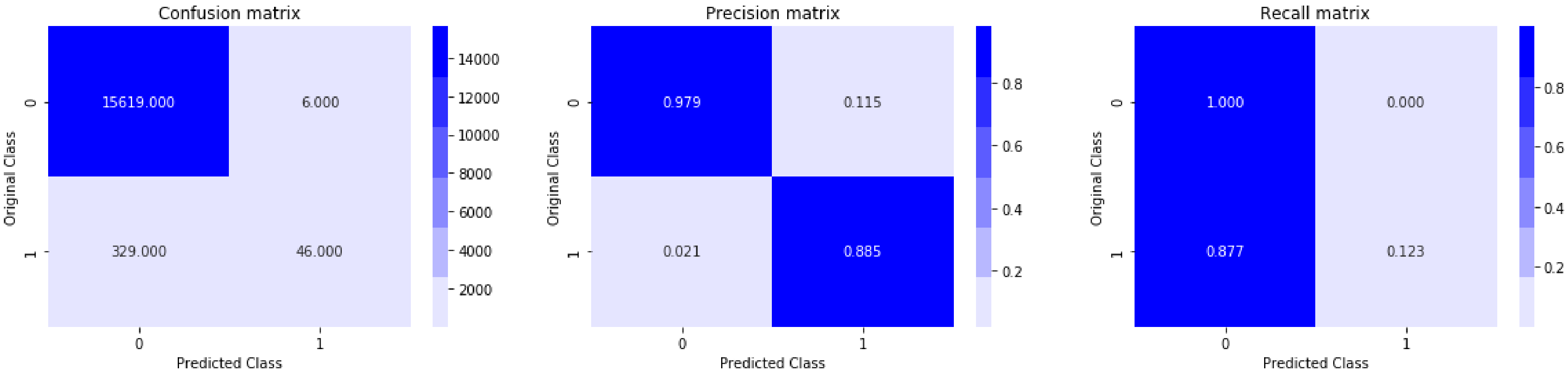
In [8]:
```
svc_clf = SVC(class_weight='balanced', C = 10, gamma = gamma, kernel = 'rbf')
svc_clf.fit(X_train, y_train)
svc_clf_cal = CalibratedClassifierCV(svc_clf, method="sigmoid")
svc_clf_cal.fit(X_train, y_train)
pred_y_train = svc_clf_cal.predict(X_train)
pred_y_test = svc_clf_cal.predict(X_test)
pickle.dump(svc_clf_cal, open("ksvc.sav", 'wb'))

print("-"*50, "Train set", "-"*50)
print('Log-loss: ', log_loss(y_train, pred_y_train, labels=svc_clf.classes_, eps=1e-15))
print('Micro F1-score: ', f1_score(y_train, pred_y_train, average='micro'))
plot_confusion_matrix(y_train, pred_y_train)
print("-"*50, "Test set", "-"*50)
print('Log-loss: ', log_loss(y_test, pred_y_test, labels=svc_clf.classes_, eps=1e-15))
print('Micro F1-score: ', f1_score(y_test, pred_y_test, average='micro'))
plot_confusion_matrix(y_test, pred_y_test)
```

```
-------------------------------------------------- Train set --------------------------------------------------
Log-loss:  0.014967042983691135
Micro F1-score:  0.9995666666666667
```

```
Number of misclassified points 0.04%
Number of False Positives:  18
Number of False Negatives:  8
Total Cost (cost1+cost2):  4180
```




```
-------------------------------------------------- Test set --------------------------------------------------
Log-loss:  0.7231559306174796
Micro F1-score:  0.9790625
Number of misclassified points 2.09%
Number of False Positives:  6
Number of False Negatives:  329
Total Cost (cost1+cost2):  164560
```



Random Forest Classifier

In [9]:
```
clf = RandomForestClassifier(class_weight='balanced')
parameters = {'n_estimators' : [10,50,100,500,1000,2000,3000]}
grid_clf = GridSearchCV(clf, parameters, cv=3, scoring='f1_micro')
grid_clf.fit(X_train, y_train)

print(grid_clf.best_params_)
```

```
{'n_estimators': 1000}
```

In [10]:
```
n_estimators = grid_clf.best_params_.get('n_estimators')
```

In [11]:
```
clf = RandomForestClassifier(class_weight='balanced', n_estimators=1000)
parameters = {'max_depth': [1, 5, 10, 50]}
grid_clf = GridSearchCV(clf, parameters, cv=3, scoring='f1_micro')
grid_clf.fit(X_train, y_train)

print(grid_clf.best_params_)
```

```
{'max_depth': 50}
```

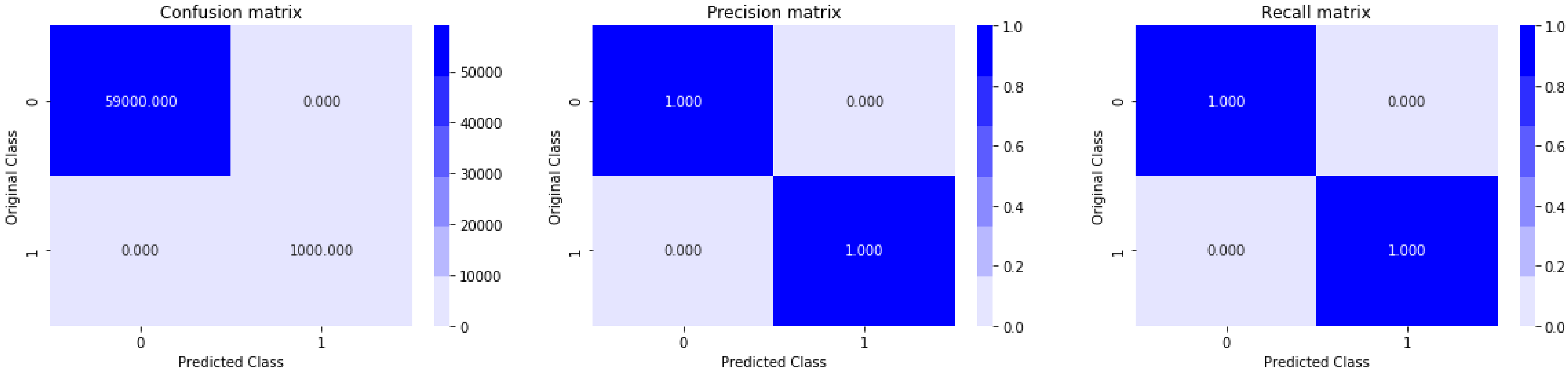
In [12]:
```
max_depth = grid_clf.best_params_.get('max_depth')
```

Best value for hyper-parameter 'max_depth' and 'n_estimators' are 50 and 1000 for random forest.

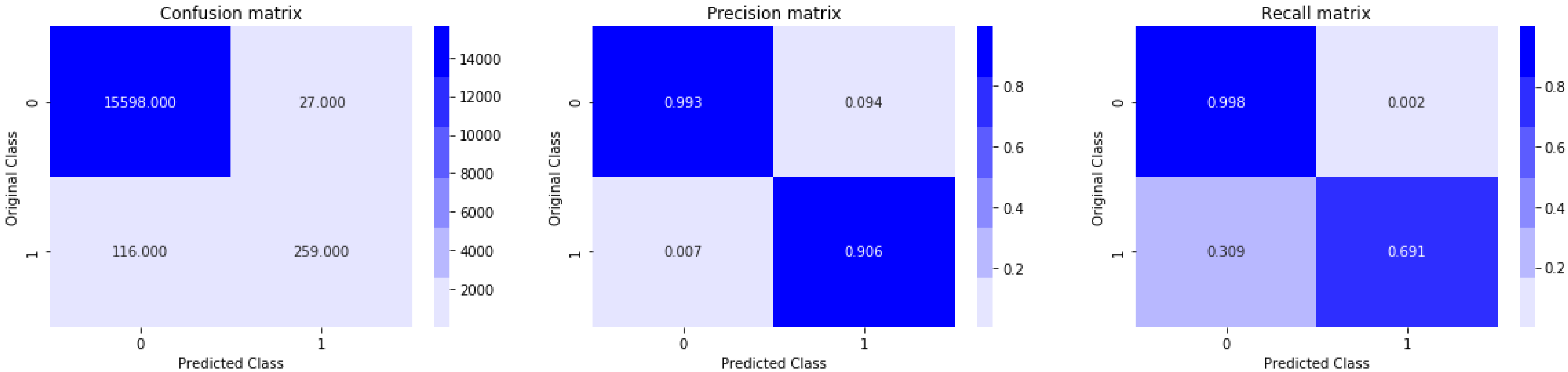
In [13]:
```
rfc_clf = RandomForestClassifier(class_weight='balanced', max_depth = max_depth, n_estimators = n_estimators)
rfc_clf.fit(X_train, y_train)
rfc_clf_cal = CalibratedClassifierCV(rfc_clf, method="sigmoid")
rfc_clf_cal.fit(X_train, y_train)
pred_y_train = rfc_clf_cal.predict(X_train)
pred_y_test = rfc_clf_cal.predict(X_test)
pickle.dump(rfc_clf_cal, open("rfc.sav", 'wb'))

print("-"*50, "Train set", "-"*50)
print('Log-loss: ', log_loss(y_train, pred_y_train, labels=svc_clf.classes_, eps=1e-15))
print('Micro F1-score: ', f1_score(y_train, pred_y_train, average='micro'))
plot_confusion_matrix(y_train, pred_y_train)
print("-"*50, "Test set", "-"*50)
print('Log-loss: ', log_loss(y_test, pred_y_test, labels=svc_clf.classes_, eps=1e-15))
print('Micro F1-score: ', f1_score(y_test, pred_y_test, average='micro'))
plot_confusion_matrix(y_test, pred_y_test)
```

```
-------------------------------------------------- Train set --------------------------------------------------
```

```
Log-loss:  9.99200722162641e-16
Micro F1-score:  1.0
Number of misclassified points 0.0%
Number of False Positives:  0
Number of False Negatives:  0
Total Cost (cost1+cost2):  0
```




```
---------------------------------------------------- Test set ----------------------------------------------------
Log-loss:  0.30869166335017786
Micro F1-score:  0.9910625
Number of misclassified points 0.89%
Number of False Positives:  27
Number of False Negatives:  116
Total Cost (cost1+cost2):  58270
```



XGBClassifier

In [14]:
```
clf = XGBClassifier(class_weight='balanced', nthread=-1, verbosity=0)
parameters={
     'n_estimators':[100,200,500,1000,2000],
     'max_depth':[3,5,10],
}
grid_clf = RandomizedSearchCV(clf, parameters, cv=3, scoring='f1_micro', n_jobs=-1)
grid_clf.fit(X_train, y_train)

print(grid_clf.best_params_)
```

```
{'n_estimators': 2000, 'max_depth': 5}
```

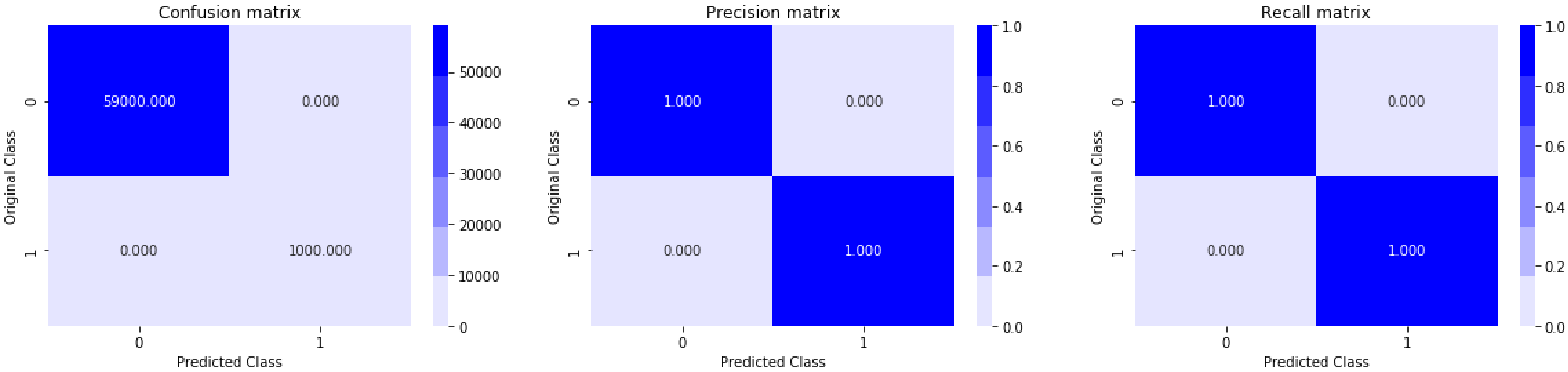
In [15]:
```
max_depth = grid_clf.best_params_.get('max_depth')
n_estimators = grid_clf.best_params_.get('n_estimators')
```

Best value for hyper-parameter 'max_depth' and 'n_estimators' are 5 and 2000 for xgbclassifier.

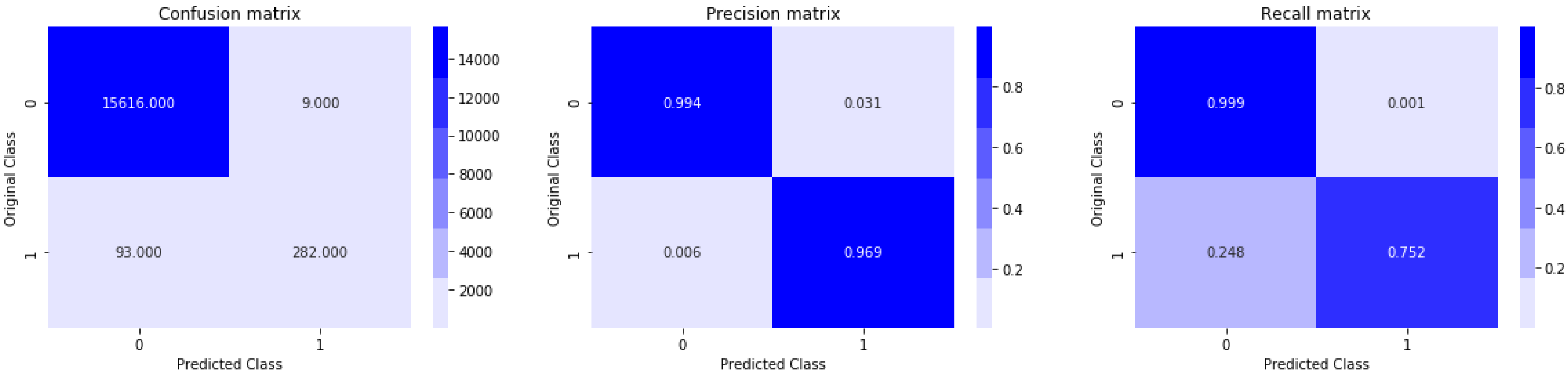
In [18]:
```
xgbc_clf = XGBClassifier(class_weight='balanced', max_depth = max_depth, n_estimators = n_estimators, nthread=-1, ve
xgbc_clf.fit(X_train, y_train)
xgbc_clf_cal = CalibratedClassifierCV(xgbc_clf, method="sigmoid")
xgbc_clf_cal.fit(X_train, y_train)
pred_y_train = xgbc_clf_cal.predict(X_train)
pred_y_test = xgbc_clf_cal.predict(X_test)
pickle.dump(xgbc_clf_cal, open("xgbc.sav", 'wb'))

print("-"*50, "Train set", "-"*50)
print('Log-loss: ', log_loss(y_train, pred_y_train, labels=svc_clf.classes_, eps=1e-15))
print('Micro F1-score: ', f1_score(y_train, pred_y_train, average='micro'))
plot_confusion_matrix(y_train, pred_y_train)
print("-"*50, "Test set", "-"*50)
print('Log-loss: ', log_loss(y_test, pred_y_test, labels=svc_clf.classes_, eps=1e-15))
print('Micro F1-score: ', f1_score(y_test, pred_y_test, average='micro'))
plot_confusion_matrix(y_test, pred_y_test)
```

```
-------------------------------------------------- Train set --------------------------------------------------
Log-loss:  9.99200722162641e-16
Micro F1-score:  1.0
Number of misclassified points 0.0%
Number of False Positives:  0
```

```
Number of False Negatives:  0
Total Cost (cost1+cost2):  0
```




```
----------------------------------------------- Test set -----------------------------------------------
Log-loss:  0.22018514929111102
Micro F1-score:  0.993625
Number of misclassified points 0.64%
Number of False Positives:  9
Number of False Negatives:  93
Total Cost (cost1+cost2):  46590
```



For F1-score and log-loss, RandomForest and XGBoost performer better than others.

But if we consider cost (primary objective is to minimize the cost), SVC didnt perform well. XGBoost and RandomForest performed better than SVC. But logistic regression performed better then others giving lowest cost.

Custom stacking model

In [10]:

```
# Load dataframe from pickle
train_data = pd.read_pickle("./train_data.pkl")
test_data = pd.read_pickle("./test_data.pkl")

print(train_data.shape)
print(train_data.shape)
```

```
(60000, 170)
(60000, 170)
```

In [11]:

```
# Spliting data to train base models and meta model of stacking model
D1, D2 = train_test_split(train_data, test_size=0.5, stratify = train_data['class'])

print(D1.shape)
print(D2.shape)
```

```
(30000, 170)
(30000, 170)
```

We can choose 3 base models as logistic regression, random forest classifier, XGBClassifier as these classifers gave less cost. We can use Decision Tree as a meta classifier.

Train base models

In [12]:

```
def train_model(clf, filename, df, params= dict()):
    """This function takes model, X_train and y_train as input.
    It fits the model with hyper-tunning and calibration and save it in pickle
    """
    X_train = df.iloc[:, :-1]
    y_train = df.iloc[:,-1].astype('int64')
    modelName = filename.split(".")[0]

    if (len(params.keys())) > 1:
        print("Training model " + modelName + " with Randomized search.")
        cv_clf = RandomizedSearchCV(clf, params, cv=3, scoring='f1_micro', n_jobs=-1)
    else:
        print("Training model " + modelName + " with Grid search.")
        cv_clf = GridSearchCV(clf, params, cv=3, scoring='f1_micro', n_jobs=-1)
```

```
    cv_clf.fit(X_train, y_train)

    best_params = cv_clf.best_params_
    print('Best params for model: ', best_params)
    clf.set_params(**cv_clf.best_params_)
    clf.fit(X_train, y_train)
    clf_cal = CalibratedClassifierCV(clf, method="sigmoid")
    clf_cal.fit(X_train, y_train)
    pickle.dump(clf_cal, open(filename, 'wb'))
    print("Saved model: " + filename)
```

In [13]:

```
def predict_base_models(filenames, X_input):
    y_preds = dict()
    for filename in filenames:
        modelName = filename.split(".")[0]
        clf = pickle.load(open(filename, 'rb'))
        y_pred = clf.predict_proba(X_input)
        y_preds[modelName] = y_pred[:, 1]

    x_df = pd.DataFrame.from_dict(y_preds)
    x_df.reset_index(inplace= True)
    x_df.drop('index', axis=1, inplace=True)
    print("Y_preds df shape: ", x_df.shape)
    return x_df
```

In [14]:

```
def trainStackingModel(classifiers, k, D1, D2, test_data):
    """This function takes models with parameters and dataframes to train and test  as input.
    It trains all base models and using them, it trains the meta model and finally use them to predict the testset da
    """
    # get base and meta model input
    base_models = classifiers[:-1]
    meta_model = classifiers[-1]
    y_preds = list()
    base_filenames = list()
    base_dir = 'K-' + str(k)
    if not os.path.exists(base_dir):
        os.makedirs(base_dir)

    # Train base models
    for i, base_clf_inp in enumerate(base_models):
        # For each model, get the actual model and its parameters with which it will be hyper-tunned
```

```
        base_clf = base_clf_inp[0]
        base_params = base_clf_inp[1]
        # Get 50% sampled data from D1 to train the base model
        sample_D1 = D1.sample(frac= 0.5, replace= True)
        base_filename = base_dir + '/M' + str(i+1) + '_' + type(base_clf).__name__ + '.sav'
        train_model(base_clf, base_filename, sample_D1, base_params)
        base_filenames.append(base_filename)

    # Do base model predictions
    X_train_D2 = D2.iloc[:, :-1]
    y_train_D2 = D2.iloc[:,-1].astype('int64')
    meta_df = predict_base_models(base_filenames, X_train_D2)
    # train meta-model
    meta_df['Y'] = y_train_D2.values
    meta_clf = meta_model[0]
    meta_params = meta_model[1]
    meta_filename = base_dir + '/Meta' + type(meta_clf).__name__ + '.sav'
    train_model(meta_clf, meta_filename, meta_df, meta_params)

    # Test stacking model after training K models
    print("Started prediction for X_test")
    X_test = test_data.iloc[:, :-1]
    y_test = test_data.iloc[:, -1]
    base_y_preds = predict_base_models(base_filenames, X_test)
    saved_meta_clf = pickle.load(open(meta_filename, 'rb'))
    y_pred_test = saved_meta_clf.predict(base_y_preds)
    print('Log-loss: ', log_loss(y_test, y_pred_test, eps=1e-15))
    print('Micro F1-score: ', f1_score(y_test, y_pred_test, average='micro'))
    plot_confusion_matrix(y_test, y_pred_test)
```

We can try different combination of K models and check their performance. Models to be used are logistic regression, svc, kernel svc, decision tree, random forest and xgbClassifier. We can try different k values. (k-1 Base models and one meta-model). Models can be repeated as we have only 6 models.

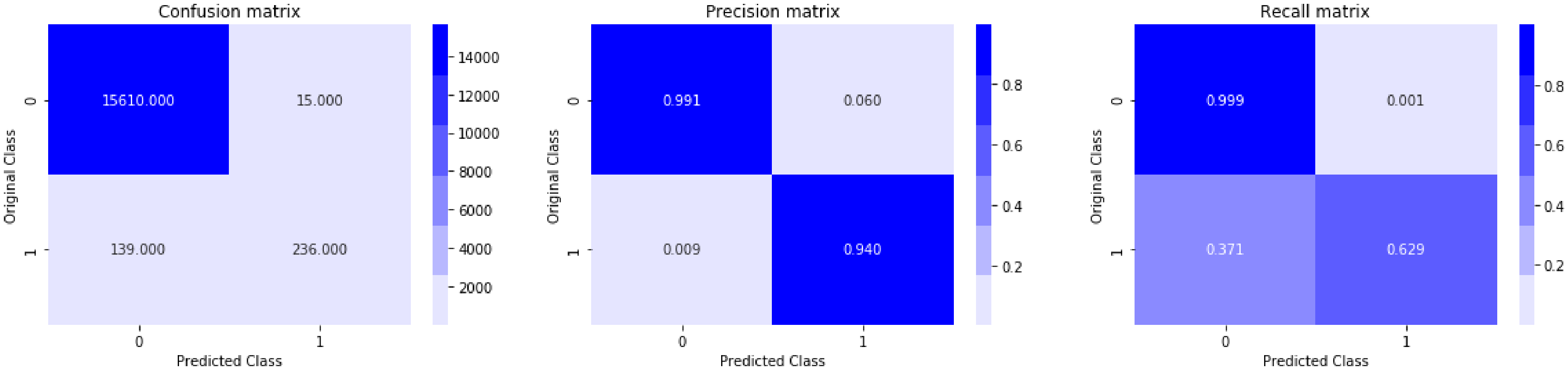
In [15]:
```
def getAllModels():
  """This function returns all the models and their parameter values"""
  def getLogisticRrgression():
    # Model 1: Logistic Regression
    lr_model = LogisticRegression(penalty='l2',class_weight='balanced', solver='liblinear', max_iter= 1000)
    lr_params = {'C' : [10 ** x for x in range(-5, 4)]}
    return (lr_model, lr_params)
```

```
    def getSVC():
        # Model 2: SVC
        svc_model = SVC(class_weight= 'balanced')
        svc_params = {'C' : [10 ** x for x in range(-5, 4)]}
        return (svc_model, svc_params)

    def getKSVC():
        # model 3: Kernel SVC
        k_svc_model = SVC(class_weight='balanced', kernel = 'rbf')
        k_svc_params = {'C' : [10 ** x for x in range(-5, 4)], 'gamma' : [10 ** x for x in range(-3, 3)]}
        return (k_svc_model, k_svc_params)

    def getDecisionTree():
        # Model 4: Decision Tree
        dt_model = DecisionTreeClassifier()
        dt_params = {'min_samples_split': [10, 100, 200, 500], 'max_depth': [5, 10, 50, 75]}
        return (dt_model, dt_params)

    def getRandomForest():
        # Model 5: Random Forest
        rf_model = RandomForestClassifier(class_weight='balanced')
        rf_params = {'n_estimators' : [750,1000, 1200, 1500], 'max_depth': [5, 10, 50, 75]}
        return (rf_model, rf_params)

    def getXGBoost():
        # model 6: XGBoost
        xgbc_model = XGBClassifier(class_weight='balanced', nthread=-1, verbosity=0)
        xgbc_params = parameters={'n_estimators':[750,1000, 1200, 1500],'max_depth':[3, 5, 7, 10]}
        return (xgbc_model, xgbc_params)

    return [getLogisticRrgression, getSVC, getKSVC, getDecisionTree, getRandomForest, getXGBoost]
```

For k=5, 10, 15, 20

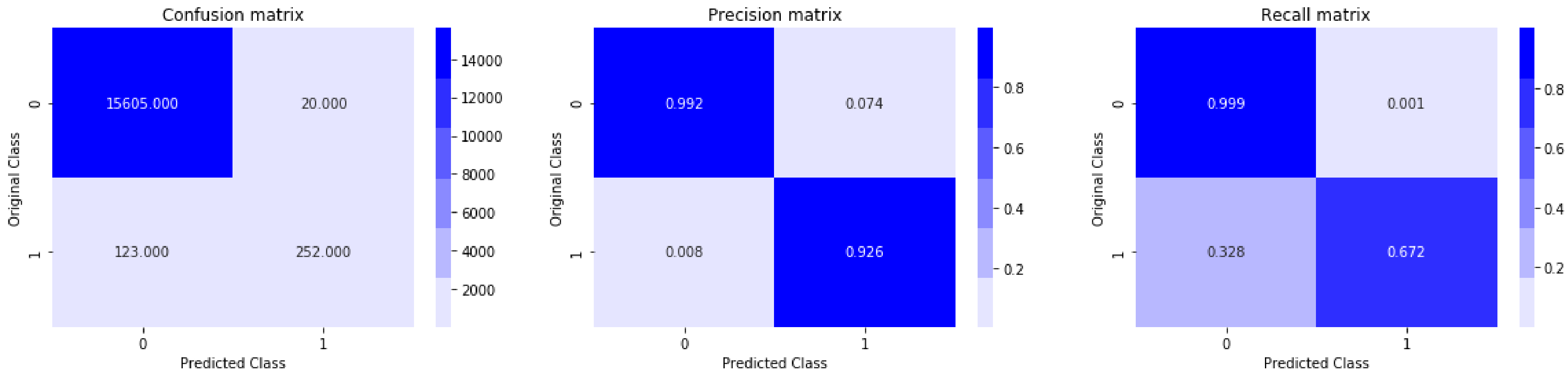
In [16]:
```
for k in range(5,21,5):
    print("Started training {} models".format(k))
    classifiers = getAllModels()
    no_clfs = len(classifiers)
    k_models = [classifiers[i]() for i in np.random.randint(low = 0, high = no_clfs, size = k)]
    trainStackingModel(k_models, k, D1, D2, test_data)
```

```
Started training 5 models
Training model K-5/M1_DecisionTreeClassifier with Randomized search.
Best params for model:  {'min_samples_split': 10, 'max_depth': 10}
Saved model: K-5/M1_DecisionTreeClassifier.sav
Training model K-5/M2_XGBClassifier with Randomized search.
Best params for model:  {'n_estimators': 750, 'max_depth': 7}
Saved model: K-5/M2_XGBClassifier.sav
Training model K-5/M3_DecisionTreeClassifier with Randomized search.
Best params for model:  {'min_samples_split': 10, 'max_depth': 5}
Saved model: K-5/M3_DecisionTreeClassifier.sav
Training model K-5/M4_DecisionTreeClassifier with Randomized search.
Best params for model:  {'min_samples_split': 10, 'max_depth': 5}
Saved model: K-5/M4_DecisionTreeClassifier.sav
Y_preds df shape:  (30000, 4)
Training model K-5/MetaRandomForestClassifier with Randomized search.
Best params for model:  {'n_estimators': 1000, 'max_depth': 50}
Saved model: K-5/MetaRandomForestClassifier.sav
Started prediction for X_test
Y_preds df shape:  (16000, 4)
Log-loss:  0.3324364724236071
Micro F1-score:  0.990375
Number of misclassified points 0.96%
Number of False Positives:  15
Number of False Negatives:  139
Total Cost (cost1+cost2):  69650
```



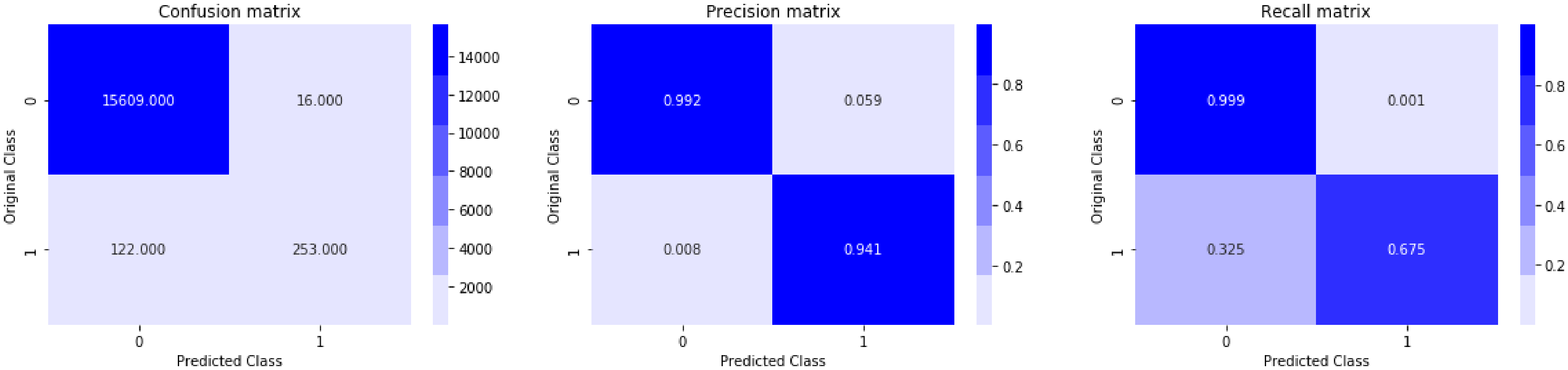

```
Started training 10 models
Training model K-10/M1_LogisticRegression with Grid search.
Best params for model:  {'C': 1000}
Saved model: K-10/M1_LogisticRegression.sav
Training model K-10/M2_DecisionTreeClassifier with Randomized search.
```

```
Best params for model:  {'min_samples_split': 10, 'max_depth': 50}
Saved model: K-10/M2_DecisionTreeClassifier.sav
Training model K-10/M3_SVC with Grid search.
Best params for model:  {'C': 1}
Saved model: K-10/M3_SVC.sav
Training model K-10/M4_RandomForestClassifier with Randomized search.
Best params for model:  {'n_estimators': 1500, 'max_depth': 50}
Saved model: K-10/M4_RandomForestClassifier.sav
Training model K-10/M5_RandomForestClassifier with Randomized search.
Best params for model:  {'n_estimators': 1200, 'max_depth': 50}
Saved model: K-10/M5_RandomForestClassifier.sav
Training model K-10/M6_SVC with Grid search.
Best params for model:  {'C': 1}
Saved model: K-10/M6_SVC.sav
Training model K-10/M7_XGBClassifier with Randomized search.
Best params for model:  {'n_estimators': 1000, 'max_depth': 5}
Saved model: K-10/M7_XGBClassifier.sav
Training model K-10/M8_XGBClassifier with Randomized search.
Best params for model:  {'n_estimators': 750, 'max_depth': 5}
Saved model: K-10/M8_XGBClassifier.sav
Training model K-10/M9_SVC with Randomized search.
Best params for model:  {'gamma': 0.01, 'C': 10}
Saved model: K-10/M9_SVC.sav
Y_preds df shape:  (30000, 9)
Training model K-10/MetaLogisticRegression with Grid search.
Best params for model:  {'C': 0.001}
Saved model: K-10/MetaLogisticRegression.sav
Started prediction for X_test
Y_preds df shape:  (16000, 9)
Log-loss:  0.3086913135263029
Micro F1-score:  0.9910625
Number of misclassified points 0.89%
Number of False Positives:  20
Number of False Negatives:  123
Total Cost (cost1+cost2):  61700
```




```
Started training 15 models
Training model K-15/M1_XGBClassifier with Randomized search.
Best params for model:  {'n_estimators': 750, 'max_depth': 5}
Saved model: K-15/M1_XGBClassifier.sav
Training model K-15/M2_SVC with Grid search.
Best params for model:  {'C': 100}
Saved model: K-15/M2_SVC.sav
Training model K-15/M3_SVC with Grid search.
Best params for model:  {'C': 10}
Saved model: K-15/M3_SVC.sav
Training model K-15/M4_XGBClassifier with Randomized search.
Best params for model:  {'n_estimators': 1000, 'max_depth': 3}
Saved model: K-15/M4_XGBClassifier.sav
Training model K-15/M5_RandomForestClassifier with Randomized search.
Best params for model:  {'n_estimators': 1000, 'max_depth': 75}
Saved model: K-15/M5_RandomForestClassifier.sav
Training model K-15/M6_RandomForestClassifier with Randomized search.
Best params for model:  {'n_estimators': 750, 'max_depth': 75}
Saved model: K-15/M6_RandomForestClassifier.sav
Training model K-15/M7_XGBClassifier with Randomized search.
Best params for model:  {'n_estimators': 750, 'max_depth': 7}
Saved model: K-15/M7_XGBClassifier.sav
Training model K-15/M8_XGBClassifier with Randomized search.
Best params for model:  {'n_estimators': 750, 'max_depth': 5}
Saved model: K-15/M8_XGBClassifier.sav
Training model K-15/M9_SVC with Grid search.
Best params for model:  {'C': 10}
Saved model: K-15/M9_SVC.sav
Training model K-15/M10_LogisticRegression with Grid search.
Best params for model:  {'C': 1000}
```

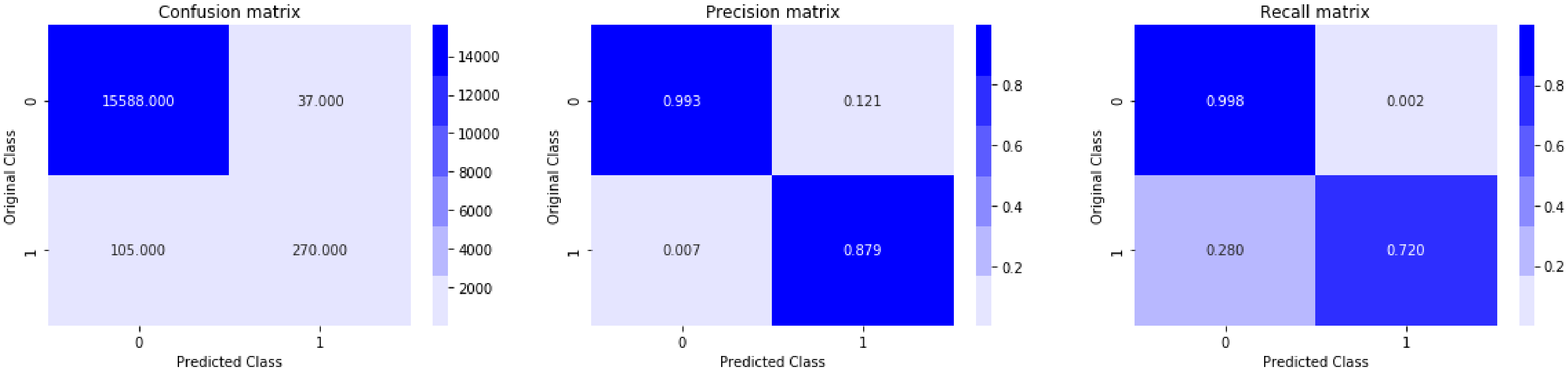
```
Saved model: K-15/M10_LogisticRegression.sav
Training model K-15/M11_XGBClassifier with Randomized search.
Best params for model:  {'n_estimators': 1500, 'max_depth': 3}
Saved model: K-15/M11_XGBClassifier.sav
Training model K-15/M12_SVC with Grid search.
Best params for model:  {'C': 10}
Saved model: K-15/M12_SVC.sav
Training model K-15/M13_SVC with Grid search.
Best params for model:  {'C': 1}
Saved model: K-15/M13_SVC.sav
Training model K-15/M14_DecisionTreeClassifier with Randomized search.
Best params for model:  {'min_samples_split': 10, 'max_depth': 5}
Saved model: K-15/M14_DecisionTreeClassifier.sav
Y_preds df shape:  (30000, 14)
Training model K-15/MetaLogisticRegression with Grid search.
Best params for model:  {'C': 0.001}
Saved model: K-15/MetaLogisticRegression.sav
Started prediction for X_test
Y_preds df shape:  (16000, 14)
Log-loss:  0.29789774600353575
Micro F1-score:  0.991375
Number of misclassified points 0.86%
Number of False Positives:  16
Number of False Negatives:  122
Total Cost (cost1+cost2):  61160
```




```
Started training 20 models
Training model K-20/M1_LogisticRegression with Grid search.
Best params for model:  {'C': 100}
Saved model: K-20/M1_LogisticRegression.sav
Training model K-20/M2_LogisticRegression with Grid search.
```

```
Best params for model:  {'C': 1000}
Saved model: K-20/M2_LogisticRegression.sav
Training model K-20/M3_DecisionTreeClassifier with Randomized search.
Best params for model:  {'min_samples_split': 10, 'max_depth': 10}
Saved model: K-20/M3_DecisionTreeClassifier.sav
Training model K-20/M4_XGBClassifier with Randomized search.
Best params for model:  {'n_estimators': 1000, 'max_depth': 7}
Saved model: K-20/M4_XGBClassifier.sav
Training model K-20/M5_DecisionTreeClassifier with Randomized search.
Best params for model:  {'min_samples_split': 10, 'max_depth': 50}
Saved model: K-20/M5_DecisionTreeClassifier.sav
Training model K-20/M6_DecisionTreeClassifier with Randomized search.
Best params for model:  {'min_samples_split': 100, 'max_depth': 75}
Saved model: K-20/M6_DecisionTreeClassifier.sav
Training model K-20/M7_SVC with Randomized search.
Best params for model:  {'gamma': 0.001, 'C': 100}
Saved model: K-20/M7_SVC.sav
Training model K-20/M8_RandomForestClassifier with Randomized search.
Best params for model:  {'n_estimators': 750, 'max_depth': 75}
Saved model: K-20/M8_RandomForestClassifier.sav
Training model K-20/M9_RandomForestClassifier with Randomized search.
Best params for model:  {'n_estimators': 1200, 'max_depth': 50}
Saved model: K-20/M9_RandomForestClassifier.sav
Training model K-20/M10_RandomForestClassifier with Randomized search.
Best params for model:  {'n_estimators': 1000, 'max_depth': 75}
Saved model: K-20/M10_RandomForestClassifier.sav
Training model K-20/M11_RandomForestClassifier with Randomized search.
Best params for model:  {'n_estimators': 1500, 'max_depth': 75}
Saved model: K-20/M11_RandomForestClassifier.sav
Training model K-20/M12_SVC with Grid search.
Best params for model:  {'C': 1000}
Saved model: K-20/M12_SVC.sav
Training model K-20/M13_XGBClassifier with Randomized search.
Best params for model:  {'n_estimators': 1000, 'max_depth': 10}
Saved model: K-20/M13_XGBClassifier.sav
Training model K-20/M14_SVC with Randomized search.
Best params for model:  {'gamma': 0.01, 'C': 1}
Saved model: K-20/M14_SVC.sav
Training model K-20/M15_LogisticRegression with Grid search.
Best params for model:  {'C': 1000}
Saved model: K-20/M15_LogisticRegression.sav
Training model K-20/M16_XGBClassifier with Randomized search.
Best params for model:  {'n_estimators': 1000, 'max_depth': 5}
Saved model: K-20/M16_XGBClassifier.sav
Training model K-20/M17_SVC with Grid search.
```

```
Best params for model:  {'C': 10}
Saved model: K-20/M17_SVC.sav
Training model K-20/M18_LogisticRegression with Grid search.
Best params for model:  {'C': 100}
Saved model: K-20/M18_LogisticRegression.sav
Training model K-20/M19_DecisionTreeClassifier with Randomized search.
Best params for model:  {'min_samples_split': 10, 'max_depth': 10}
Saved model: K-20/M19_DecisionTreeClassifier.sav
Y_preds df shape:  (30000, 19)
Training model K-20/MetaLogisticRegression with Grid search.
Best params for model:  {'C': 0.001}
Saved model: K-20/MetaLogisticRegression.sav
Started prediction for X_test
Y_preds df shape:  (16000, 19)
Log-loss:  0.30653348957389054
Micro F1-score:  0.991125
Number of misclassified points 0.89%
Number of False Positives:  37
Number of False Negatives:  105
Total Cost (cost1+cost2):  52870
```



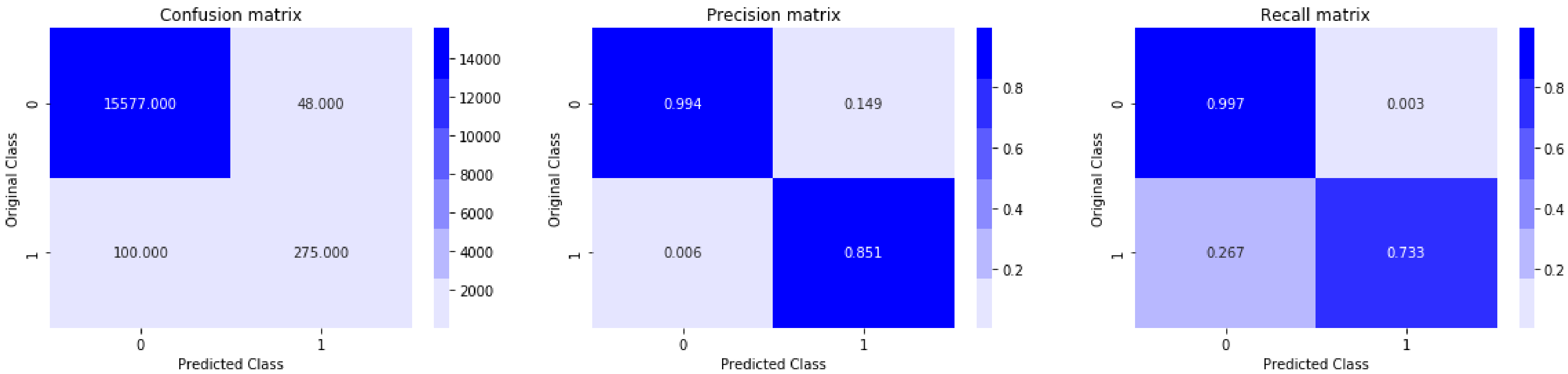
Stacking classifier with 20 base models (adding meta model also), gave the best result as compared to the same with less number of base models.

But with increase in k, the cost doesnt get reduced. It simply based on the type of base models used mostly. Now after stacking also, the logistic regression gave the best result alone if compared to other models and stacking models.

If we can use feasible number of base models which gave better results (logistic regression, random forest and xgbclassifier) while used individually may improve the performance of stacking model.

In [17]:
```
classifiers = getAllModels()
k_models = list()
# use logistic regression, random forest and xgboot as base models and decision tree as meta model
k_models.append(classifiers[0]())
k_models.append(classifiers[-2]())
k_models.append(classifiers[-1]())
k_models.append(classifiers[-3]())

trainStackingModel(k_models, 'top_base', D1, D2, test_data)
```

```
Training model K-top_base/M1_LogisticRegression with Grid search.
Best params for model:  {'C': 1000}
Saved model: K-top_base/M1_LogisticRegression.sav
Training model K-top_base/M2_RandomForestClassifier with Randomized search.
Best params for model:  {'n_estimators': 1500, 'max_depth': 50}
Saved model: K-top_base/M2_RandomForestClassifier.sav
Training model K-top_base/M3_XGBClassifier with Randomized search.
Best params for model:  {'n_estimators': 750, 'max_depth': 7}
Saved model: K-top_base/M3_XGBClassifier.sav
Y_preds df shape:  (30000, 3)
Training model K-top_base/MetaDecisionTreeClassifier with Randomized search.
Best params for model:  {'min_samples_split': 500, 'max_depth': 5}
Saved model: K-top_base/MetaDecisionTreeClassifier.sav
Started prediction for X_test
Y_preds df shape:  (16000, 3)
Log-loss:  0.31948608044521537
Micro F1-score:  0.99075
Number of misclassified points 0.92%
Number of False Positives:  48
Number of False Negatives:  100
Total Cost (cost1+cost2):  50480
```



So we can consider Logistic regression with whole training dataset with proper hyper-parameter tunning (C: 10) as our best model.