Data Set Information:

---

1. Title: APS Failure at Scania Trucks
2. Source Information

-- Creator: Scania CV AB VagnmakarvÃ¤gen 1 151 32 SÃ¶dertÃ¤lje StockholIm Sweden -- Donor: Tony Lindgren (tony '@' dsv.su.se) and Jonas Biteus (jonas.biteus '@' scania.com) -- Date: September, 2016

3. Past Usage: Industrial Challenge 2016 at The 15th International Symposium on Intelligent Data Analysis (IDA)

-- Results: The top three contestants | Score | Number of Type 1 faults | Number of Type 2 faults

---

Camila F. Costa and Mario A. Nascimento | 9920 | 542 | 9 Christopher Gondek, Daniel Hafner and Oliver R. Sampson | 10900 | 490 | 12 Sumeet Garnaik, Sushovan Das, Rama Syamala Sreepada and Bidyut Kr. Patra | 11480 | 398 | 15

4. Relevant Information:

-- Introduction The dataset consists of data collected from heavy Scania trucks in everyday usage. The system in focus is the Air Pressure system (APS) which generates pressurised air that are utilized in various functions in a truck, such as braking and gear changes. The datasets' positive class consists of component failures for a specific component of the APS system. The negative class consists of trucks with failures for components not related to the APS. The data consists of a subset of all available data, selected by experts.

-- Challenge metric

Cost-metric of miss-classification:

Predicted class | True class | | pos | neg |

---

pos | - | Cost_1 |

neg | Cost_2 | - |

Cost_1 = 10 and cost_2 = 500

The total cost of a prediction model the sum of 'Cost_1' multiplied by the number of Instances with type 1 failure and 'Cost_2' with the number of instances with type 2 failure, resulting in a 'Total_cost'.

In this case Cost_1 refers to the cost that an unnessecary check needs to be done by an mechanic at an workshop, while Cost_2 refer to the cost of missing a faulty truck, which may cause a breakdown.

Total_cost = Cost_1*No_Instances* + *Cost_2*No_Instances.

5. Number of Instances: The training set contains 60000 examples in total in which 59000 belong to the negative class and 1000 positive class. The test set contains 16000 examples.
6. Number of Attributes: 171
7. Attribute Information: The attribute names of the data have been anonymized for proprietary reasons. It consists of both single numerical counters and histograms consisting of bins with different conditions. Typically the histograms have open-ended conditions at each end. For example if we measuring the ambient temperature 'T' then the histogram could be defined with 4 bins where:

bin 1 collect values for temperature T < -20 bin 2 collect values for temperature T >= -20 and T < 0 bin 3 collect values for temperature T >= 0 and T < 20 bin 4 collect values for temperature T > 20

| b1 | b2 | b3 | b4 |

-20 0 20

The attributes are as follows: class, then anonymized operational data. The operational data have an identifier and a bin id, like 'Identifier_Bin'. In total there are 171 attributes, of which 7 are histogram variabels. Missing values are denoted by 'na'.

In [1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import warnings
import seaborn as sns
warnings.filterwarnings('ignore')
%matplotlib inline
```

In [2]:
```
df1=pd.read_csv(r"C:\Users\Mukul\Downloads\aps_failure_training_set.csv")
```

In [3]:
```
df2=pd.read_csv(r"C:\Users\Mukul\Downloads\aps_failure_test_set.csv")
```

In [4]:
```
df=pd.concat([df1,df2])
```

In [8]:
```
### joining both dataset and resetting index
df=pd.concat([df1, df2])

df.reset_index(inplace=True)
## dropping index feature as it is not required
df.drop('index',axis=1,inplace=True)


# replacing na values with NaN Values
df=df.replace('na' , np.NaN)


df.head()
```

Out[8]:

	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_008	ee_00
0	neg	76698	NaN	2130706438	280	0	0	0	0	0	...	1240520	493384	721044	469792	339156	157956	73224	
1	neg	33058	NaN	0	NaN	0	0	0	0	0	...	421400	178064	293306	245416	133654	81140	97576	150
2	neg	41040	NaN	228	100	0	0	0	0	0	...	277378	159812	423992	409564	320746	158022	95128	51
3	neg	12	0	70	66	0	10	0	0	0	...	240	46	58	44	10	0	0	
4	neg	60874	NaN	1368	458	0	0	0	0	0	...	622012	229790	405298	347188	286954	311560	433954	121

5 rows × 171 columns

In [11]:
```
df.tail()
```

Out[11]:

	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_008	ee
75995	neg	81852	NaN	2130706432	892	0	0	0	0	0	...	632658	273242	510354	373918	349840	317840	960024	2
75996	neg	18	0	52	46	8	26	0	0	0	...	266	44	46	14	2	0	0	
75997	neg	79636	NaN	1670	1518	0	0	0	0	0	...	806832	449962	778826	581558	375498	222866	358934	1
75998	neg	110	NaN	36	32	0	0	0	0	0	...	588	210	180	544	1004	1338	74	
75999	neg	8	0	6	4	2	2	0	0	0	...	46	10	48	14	42	46	0	

5 rows × 171 columns

In [12]:
```
df.shape
```

Out[12]: (76000, 171)

In [13]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 76000 entries, 0 to 75999
Columns: 171 entries, class to eg_000
dtypes: int64(1), object(170)
memory usage: 99.2+ MB
```

In [14]:
```
# check  the null values
df.isnull().sum()
```

Out[14]:
```
class         0
aa_000        0
ab_000    58692
ac_000     4261
ad_000    18842
          ...
ee_007      863
ee_008      863
ee_009      863
ef_000     3486
eg_000     3485
Length: 171, dtype: int64
```

In [15]:
```
# checking the percentage of missing Values
df.isnull().mean()*100
```

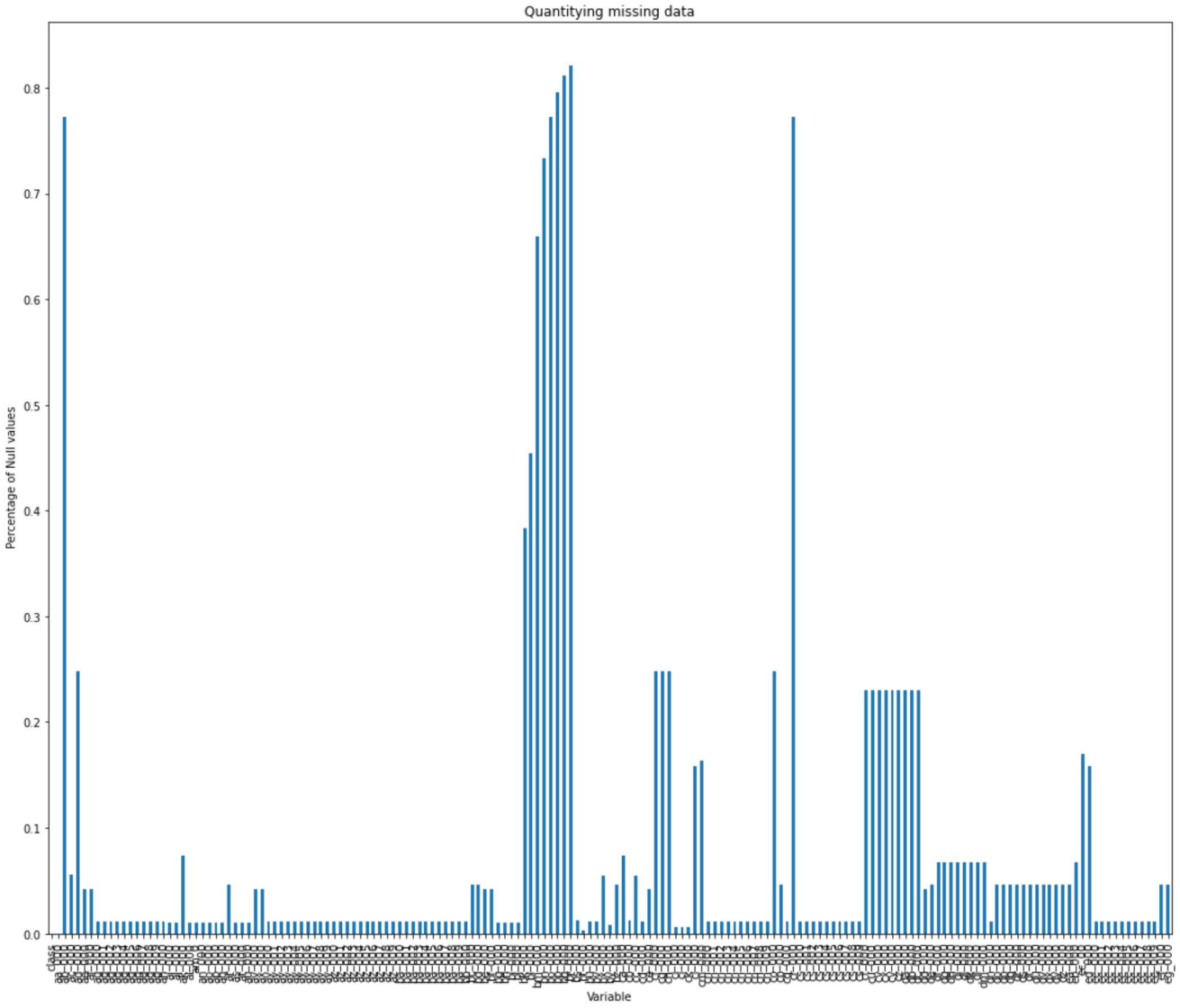
Out[15]:
```
class      0.000000
aa_000     0.000000
ab_000    77.226316
ac_000     5.606579
ad_000    24.792105
             ...
ee_007     1.135526
ee_008     1.135526
ee_009     1.135526
ef_000     4.586842
eg_000     4.585526
Length: 171, dtype: float64
```

In [16]:
```
df.isnull().sum().sum()
```

Out[16]:
```
1078695
```

In [17]:
```
df.isnull().mean().plot.bar(figsize=(18,15))
plt.ylabel(' Percentage of Null values')
plt.xlabel("Variable")
plt.title('Quantitying missing data')
```

Out[17]: Text(0.5, 1.0, 'Quantitying missing data')



In [19]:
```
# first 20 features with highest null values
pd.DataFrame(df.isnull().sum().sort_values(ascending=False)[:10]).rename(columns={0:'Null Value Count'})
```

Out[19]:

	Null Value Count
br_000	62393
bq_000	61703
bp_000	60461
bo_000	58709
cr_000	58692
ab_000	58692
bn_000	55722
bm_000	50095
bl_000	34503
bk_000	29128

In [21]:
```
# filling null values in feature with median of that feature
for feature in [feature for feature in df.columns if feature not in ['class']]:
    df[feature]=df[feature].fillna(df[feature].median())
```

In [23]:
```
## Checking Null values from top
pd.DataFrame(df.isnull().sum().sort_values(ascending=False)[:5]).rename(columns={0:'Null Value Count'})
```

Out[23]:

	Null Value Count
class	0
cs_003	0
cn_009	0
co_000	0
cp_000	0

In [ ]:

In [ ]:

In [24]:
```
# check unique values
df.nunique()
```

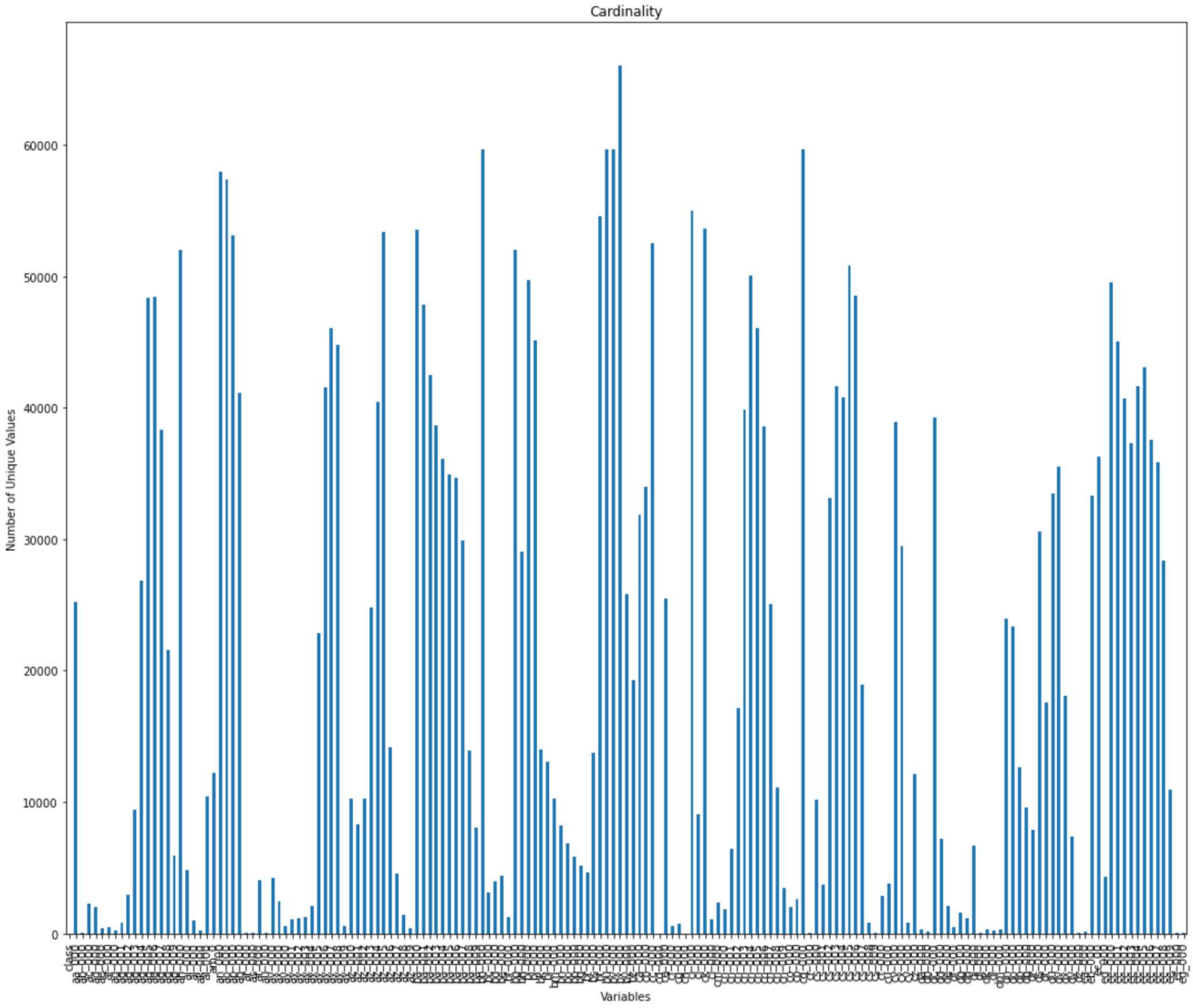
Out[24]:
```
class         2
aa_000    25211
ab_000       31
ac_000     2230
ad_000     2028
          ...
ee_007    35860
ee_008    28413
ee_009    10948
ef_000       31
eg_000       56
Length: 171, dtype: int64
```

In [25]:
```
df.nunique().sum()
```

Out[25]:
```
3450493
```

In [26]:
```
df.nunique().plot.bar(figsize=(18,15))
plt.ylabel('Number of Unique Values')
plt.xlabel('Variables')
plt.title("Cardinality")
```

Out[26]: Text(0.5, 1.0, 'Cardinality')



In [27]:
```
# check the duplicated values
df.duplicated().sum()
```

Out[27]: 0

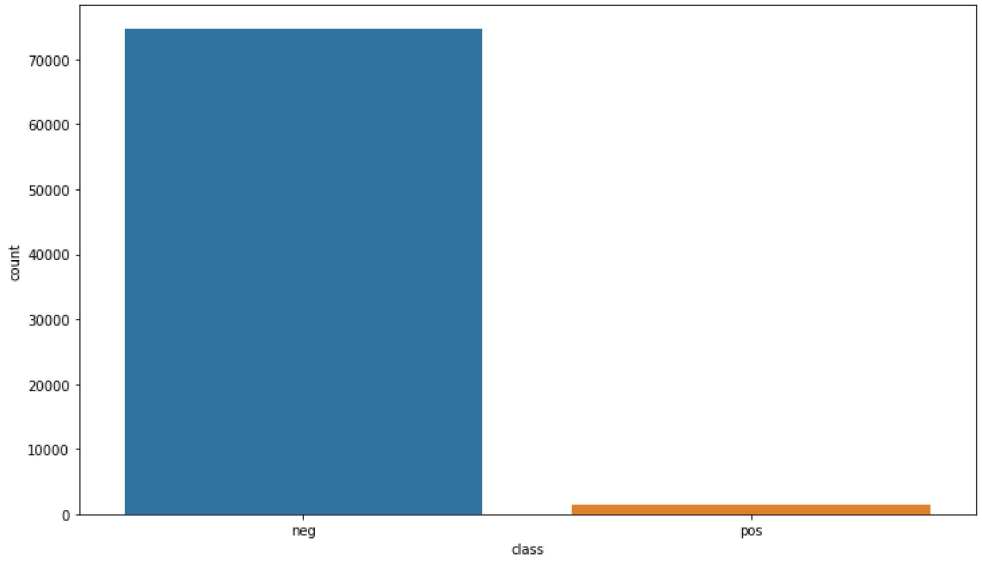
In [28]:
```
# check the unique values
df['class'].unique()
```

Out[28]: array(['neg', 'pos'], dtype=object)

In [29]:
```
df['class'].value_counts()
```

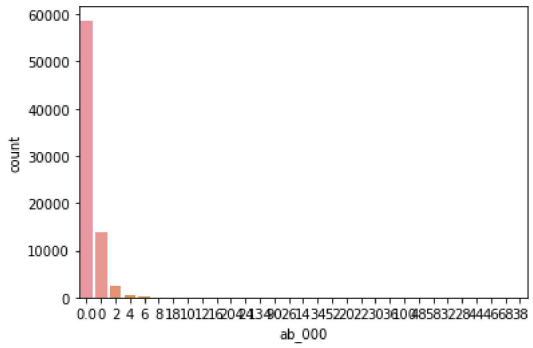
Out[29]:
```
neg    74625
pos     1375
Name: class, dtype: int64
```

In [30]:
```
plt.figure(figsize=(12,7))
sns.countplot(df['class'])
plt.show()
```



In [31]:
```
sns.countplot(df['ab_000'])
```

Out[31]:
```
<AxesSubplot:xlabel='ab_000', ylabel='count'>
```

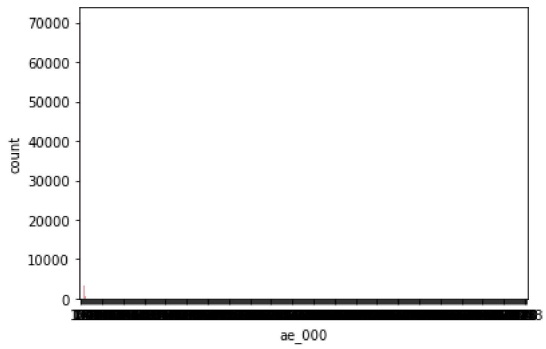


In [32]:
```
df['ae_000'].unique()
```

Out[32]:
```
array(['0', '16', '104', 0.0, '2', '222', '4', '290', '1286', '170', '6',
       '806', '1464', '274', '18', '8', '342', '64', '34', '192', '144',
       '176', '1512', '120', '58', '114', '460', '12', '112', '1314',
       '20', '550', '36', '1424', '180', '324', '204', '254', '106',
       '154', '92', '256', '202', '352', '26', '124', '2240', '466',
       '778', '28', '100', '14', '276', '42', '82', '22', '384', '282',
       '386', '98', '164', '108', '86', '38', '266', '1056', '134', '10',
       '200', '150', '326', '350', '398', '424', '32', '128', '752',
       '314', '732', '2312', '400', '158', '668', '30', '212', '224',
       '950', '4726', '60', '226', '388', '640', '258', '62', '206', '44',
       '414', '458', '78', '492', '308', '52', '50', '162', '24', '662',
       '536', '456', '130', '140', '184', '368', '146', '118', '138',
       '872', '530', '1118', '356', '218', '126', '102', '1130', '12048',
       '1508', '396', '374', '658', '816', '556', '216', '66', '760',
       '136', '220', '280', '46', '462', '788', '360', '416', '348', '74',
       '1124', '88', '242', '580', '84', '378', '94', '406', '442',
       '2750', '4698', '40', '748', '292', '952', '284', '270', '1346',
       '506', '422', '322', '642', '48', '714', '318', '54', '68', '156',
       '70', '3702', '1654', '454', '1452', '76', '380', '72', '294',
       '382', '214', '328', '2276', '248', '680', '182', '160', '938',
       '390', '56', '814', '624', '1200', '1466', '148', '236', '744',
       '172', '1274', '836', '1582', '804', '2672', '470', '654', '794',
       '1372', '664', '484', '1468', '168', '110', '1912', '116', '524',
       '234', '1270', '310', '2468', '612', '228', '486', '246', '2600',
       '142', '1710', '478', '570', '96', '746', '90', '480', '3478',
       '538', '1292', '122', '344', '2572', '600', '468', '962', '540',
       '436', '826', '920', '302', '618', '614', '666', '2322', '2662',
       '152', '1420', '296', '366', '188', '488', '894', '1714', '572',
       '522', '702', '320', '208', '238', '500', '762', '542', '336',
       '446', '21050', '404', '698', '2702', '2306', '548', '1614', '864',
       '300', '438', '490', '482', '1520', '186', '304', '940', '132',
       '878', '1012', '376', '2218', '80', '8332', '260', '194', '566',
       '1852', '4520', '1026', '716', '340', '1860', '166', '1176', '784',
       '1116', '504', '174', '264', '430', '604', '1398', '2110', '288',
       '764', '412', '286', '602', '5386', '1412', '2048', '936', '730',
       '2370', '832', '250', '272', '464', '946', '626', '606', '362',
       '700', '1700', '402', '782', '526', '516', '1172', '734', '630',
       '11044', '210', '692', '1502', '262', '244', '972', '230', '2120',
       '4716', '514', '428', '178', '2088'], dtype=object)
```

In [33]:
```
sns.countplot(df['ae_000'])
```

Out[33]:
```
<AxesSubplot:xlabel='ae_000', ylabel='count'>
```



In [34]:
```
for feature in df.columns:
  print(df[feature].value_counts())
```

```
neg    74625
pos     1375
Name: class, dtype: int64
8         1293
10        1201
12        1158
14        1135
6          745
          ...
60302        1
119474       1
3088         1
75834        1
81852        1
Name: aa_000, Length: 25211, dtype: int64
0.0    58692
0      13842
2       2484
4        537
```

Converting Datatype

In [37]:
```
# Converting datatypes of numerical feature to int or float from object
for feature in [feature for feature in df.columns if feature not in ['class' , 'aa_000']]:
    try:
        df[feature]=df[feature].astype('int64')
    except:
        df[feature]=df[feature].astype('float64')
```

In [40]:
```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 76000 entries, 0 to 75999
Columns: 171 entries, class to eg_000
dtypes: float64(6), int64(164), object(1)
memory usage: 99.2+ MB
```

In [41]:
```
df.dtypes
```

Out[41]:
```
class     object
aa_000     int64
ab_000     int64
ac_000     int64
ad_000     int64
           ...
ee_007     int64
ee_008     int64
ee_009     int64
ef_000     int64
eg_000     int64
Length: 171, dtype: object
```

Segregate Numerical and Categorical Features

In [42]:
```
# Numerical columns
num_columns=[feature for feature in df.columns if df[feature].dtypes!='object']
num_columns
```

Out[42]:
```
['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',
```

In [43]:
```
len(num_columns)
```

Out[43]:
```
170
```

In [44]:
```
# check the Categorical columns
cat_columns=[feature for feature in df.columns if df[feature].dtypes=='object']
cat_columns
```

Out[44]:
```
['class']
```

In [45]:
```
len(cat_columns)
```

Out[45]:
```
1
```

In [46]:
```
# Getting count of each Numerical values from dataframe
for feature in num_columns:
  print(df[feature].value_counts())
```

```
...
284          1
22990        1
9324         1
8670         1
13320        1
Name: ag_002, Length: 2952, dtype: int64
0          60196
30            28
22            28
2             27
16            26
...
1326224       1
9369826       1
5399416       1
432658        1
2388752       1
Name: ag_003, Length: 9383, dtype: int64
```

In [47]:
```
# check the numerical columns unique valuse
for feature in num_columns:
  print(df[feature].unique())
```

```
[ 2801180  3477820  1040120 ... 39261520   298950 10074490]
[2445.8  2211.76 1018.64 ... 1714.88 4206.98 1546.74]
[ 2712  2334  1020 ... 20496  5066  5120]
[ 965866  664504  262032 ... 1093142 2830166  883900]
[1706908  824154  453378 ...   12290 1370722  983308]
[1240520  421400  277378 ...  484302    3716  632658]
[493384 178064 159812 ... 129744  51562 201836]
[721044 293306 423992 ... 305924  11802 510354]
[469792 245416 409564 ... 282268  38696 373918]
[339156 133654 320746 ... 345866 111826 349840]
[157956  81140 158022 ... 227924  92102 317840]
[ 73224  97576  95128 ... 177996 298810 960024]
[    0  1500   514 ... 49346 24458 25566]
[  0   4   2   8   6  84  22 134  26  10 350 320  40  30  20 482 166  12
 276 252  18  82  14 236 144  86 362  74 340  28]
[   0   32  164   54    4    2   10    6   14   12    8  144   68   86
   26   20   28   16   34   94  222   36  892   48   72  152   78  910
   70  172   46   80  182   24   56  102  430   42   18  162   40  200
 1146   44  108   92   22  606  416   88  124 1720   62   66  168]
```

Statical Based Analysis

In [48]:
```
df.describe()
```

Out[48]:

	aa_000	ab_000	ac_000	ad_000	ae_000	af_000	ag_000	ag_001	ag_002	ag_003	...	
count	7.600000e+04	76000.000000	7.600000e+04	7.600000e+04	76000.000000	76000.000000	7.600000e+04	7.600000e+04	7.600000e+04	7.600000e+04	...	7.600
mean	6.115976e+04	0.165237	3.364557e+08	1.133174e+05	6.454184	10.381474	1.981626e+02	1.191128e+03	9.587595e+03	9.258930e+04	...	4.471
std	2.647366e+05	1.609493	7.769769e+08	3.113852e+07	150.197624	197.330569	1.833153e+04	5.070174e+04	1.709273e+05	8.197977e+05	...	1.170
min	0.000000e+00	0.000000	0.000000e+00	0.000000e+00	0.000000	0.000000	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	...	0.000
25%	8.600000e+02	0.000000	2.000000e+01	4.200000e+01	0.000000	0.000000	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	...	3.082
50%	3.081300e+04	0.000000	1.540000e+02	1.280000e+02	0.000000	0.000000	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	...	2.354
75%	4.884000e+04	0.000000	8.480000e+02	2.920000e+02	0.000000	0.000000	0.000000e+00	0.000000e+00	0.000000e+00	0.000000e+00	...	4.350
max	4.294967e+07	204.000000	2.130707e+09	8.584298e+09	21050.000000	20070.000000	3.376892e+06	1.047252e+07	1.914916e+07	7.305747e+07	...	7.793

8 rows × 170 columns

In [49]:
```
df.describe().T
```

Out[49]:

	count	mean	std	min	25%	50%	75%	max
aa_000	76000.0	6.115976e+04	2.647366e+05	0.0	860.0	30813.0	48840.0	4.294967e+07
ab_000	76000.0	1.652368e-01	1.609493e+00	0.0	0.0	0.0	0.0	2.040000e+02
ac_000	76000.0	3.364557e+08	7.769769e+08	0.0	20.0	154.0	848.0	2.130707e+09
ad_000	76000.0	1.133174e+05	3.113852e+07	0.0	42.0	128.0	292.0	8.584298e+09
ae_000	76000.0	6.454184e+00	1.501976e+02	0.0	0.0	0.0	0.0	2.105000e+04
...	...	...	...	...	...	...	...	...
ee_007	76000.0	3.440830e+05	1.698180e+06	0.0	122.0	41260.0	166632.0	1.195801e+08
ee_008	76000.0	1.383521e+05	4.667835e+05	0.0	0.0	3862.0	136421.5	1.926740e+07
ee_009	76000.0	8.329316e+03	4.848501e+04	0.0	0.0	0.0	1908.0	4.570398e+06
ef_000	76000.0	8.128947e-02	4.082657e+00	0.0	0.0	0.0	0.0	4.820000e+02
eg_000	76000.0	2.090000e-01	9.999687e+00	0.0	0.0	0.0	0.0	1.720000e+03

170 rows × 8 columns

In [50]:
```
# check covariance
df.cov()
```

Out[50]:

	aa_000	ab_000	ac_000	ad_000	ae_000	af_000	ag_000	ag_001	ag_002	ag_003
aa_000	7.008547e+10	4.240848e+03	-7.785737e+12	-6.601167e+09	9.454919e+05	1.751320e+06	5.671167e+07	1.025314e+09	8.832006e+09	6.672947e+10
ab_000	4.240848e+03	2.590468e+00	-3.112164e+06	-1.866857e+04	8.250324e+00	1.688245e+01	-6.083843e+00	5.582198e+02	6.605238e+03	4.966469e+04
ac_000	-7.785737e+12	-3.112164e+06	6.036931e+17	-3.794526e+13	-4.673741e+08	-1.498406e+09	2.147474e+11	-2.811720e+11	-2.998519e+12	-2.991981e+13
ad_000	-6.601167e+09	-1.866857e+04	-3.794526e+13	9.696074e+14	-7.272759e+05	-1.169290e+06	-2.158851e+07	-1.334675e+08	-1.073488e+09	-1.039052e+10
ae_000	9.454919e+05	8.250324e+00	-4.673741e+08	-7.272759e+05	2.255933e+04	2.460005e+04	-1.278992e+03	-7.260846e+03	2.507139e+04	2.324903e+06
...	...	...	...	...	...	...	...	...	...	...
ee_007	1.604761e+11	2.937441e+04	-6.450009e+13	-3.819348e+10	8.714962e+06	1.456917e+07	2.382054e+08	5.854768e+09	5.281512e+10	6.621189e+11
ee_008	3.208323e+10	-1.977318e+04	-1.743145e+12	-1.540448e+10	-6.672840e+05	-1.126033e+06	8.762079e+05	1.269533e+08	7.294246e+08	8.006854e+09
ee_009	1.717292e+09	-1.331063e+03	4.527769e+11	-9.391006e+08	-5.338144e+04	-8.615025e+04	-9.614317e+05	-8.678888e+06	-7.512454e+07	-7.158349e+08
ef_000	6.104712e+03	3.399409e-01	-1.966868e+07	-9.147151e+03	3.901344e+01	7.534014e+01	-6.472093e+00	1.986413e+02	3.572187e+03	3.378395e+04
eg_000	2.445737e+04	1.243482e+00	1.264366e+08	-2.354522e+04	4.639863e+01	8.764537e+01	2.648826e+01	1.756286e+03	2.201690e+04	7.879793e+04

170 rows × 170 columns

In [51]:
```
# check quantile
df.quantile()
```

Out[51]:
```
aa_000    30813.0
ab_000        0.0
ac_000      154.0
ad_000      128.0
ae_000        0.0
           ...
ee_007    41260.0
ee_008     3862.0
ee_009        0.0
ef_000        0.0
eg_000        0.0
Name: 0.5, Length: 170, dtype: float64
```

In [52]:
```
# check count values
df.count()
```

Out[52]:
```
class     76000
aa_000    76000
ab_000    76000
ac_000    76000
ad_000    76000
          ...
ee_007    76000
ee_008    76000
ee_009    76000
ef_000    76000
eg_000    76000
Length: 171, dtype: int64
```

In [53]:
```
# check minmum values
df.min()
```

Out[53]:
```
class     neg
aa_000      0
ab_000      0
ac_000      0
ad_000      0
          ...
ee_007      0
ee_008      0
ee_009      0
ef_000      0
eg_000      0
Length: 171, dtype: object
```

In [54]:
```
# check maxmimum values
df.max()
```

Out[54]:
```
class            pos
aa_000      42949672
ab_000           204
ac_000    2130706796
ad_000    8584297742
                 ...
ee_007     119580108
ee_008      19267396
ee_009       4570398
ef_000           482
eg_000          1720
Length: 171, dtype: object
```

In [55]:
```
# check standard deviation
df.std()
```

Out[55]:
```
aa_000    2.647366e+05
ab_000    1.609493e+00
ac_000    7.769769e+08
ad_000    3.113852e+07
ae_000    1.501976e+02
                 ...
ee_007    1.698180e+06
ee_008    4.667835e+05
ee_009    4.848501e+04
ef_000    4.082657e+00
eg_000    9.999687e+00
Length: 170, dtype: float64
```

In [56]:
```
# check skewness
df.skew()
```

Out[56]:
```
aa_000    112.926238
ab_000     53.888614
ac_000      1.876290
ad_000    275.680972
ae_000     91.301964
                 ...
ee_007     15.078929
ee_008     14.354372
ee_009     33.936322
ef_000     77.188722
eg_000    113.494491
Length: 170, dtype: float64
```

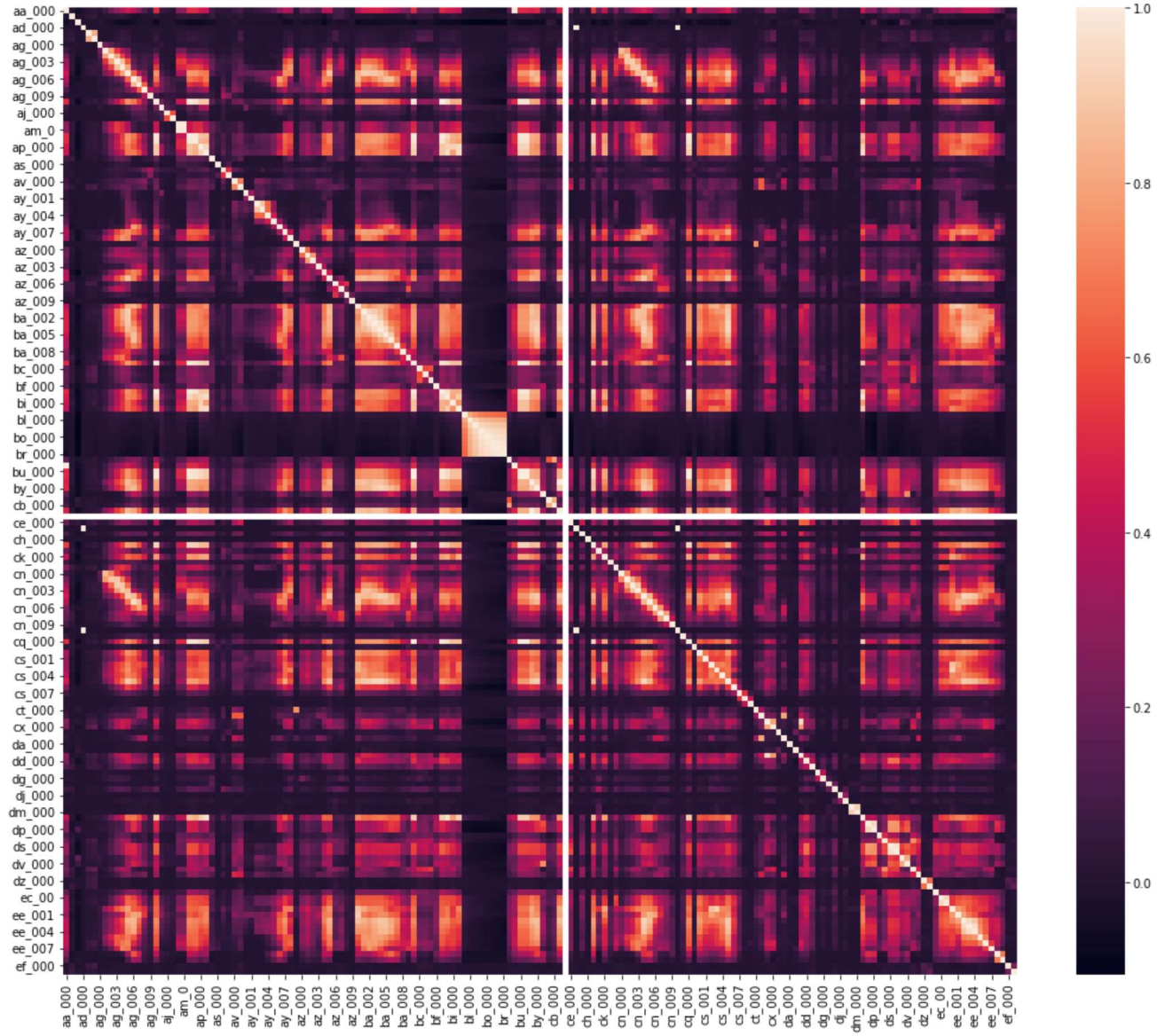
In [57]:
```
# check the correlation
df.corr()
```

Out[57]:

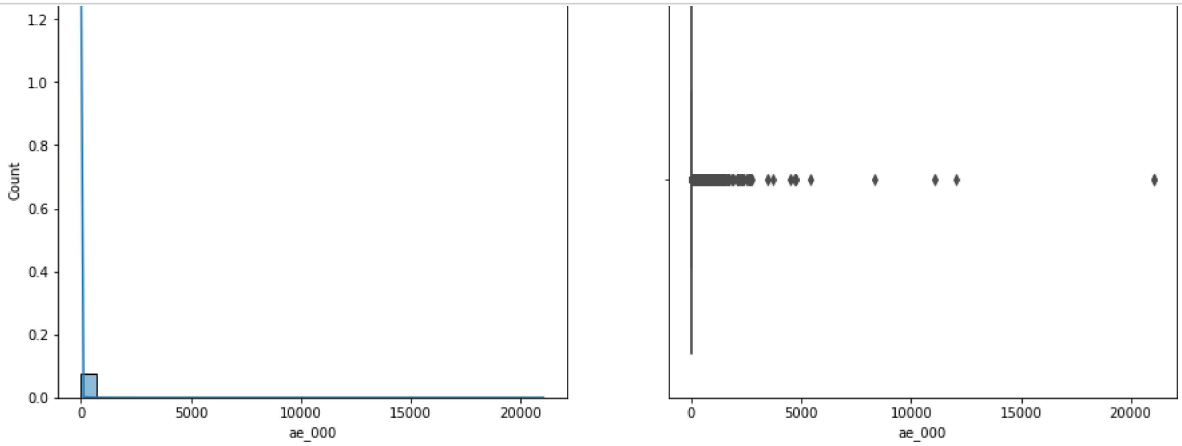
	aa_000	ab_000	ac_000	ad_000	ae_000	af_000	ag_000	ag_001	ag_002	ag_003	...	ee_002	ee_003	ee_004	ee_00
aa_000	1.000000	0.009953	-0.037851	-0.000801	0.023778	0.033524	0.011686	0.076387	0.195179	0.307466	...	0.414098	0.419884	0.409628	0.41975
ab_000	0.009953	1.000000	-0.002489	-0.000372	0.034129	0.053156	-0.000206	0.006841	0.024010	0.037640	...	-0.000332	0.010885	0.001855	0.01207
ac_000	-0.037851	-0.002489	1.000000	-0.001568	-0.004005	-0.009773	0.015077	-0.007137	-0.022578	-0.046973	...	-0.047540	-0.041136	-0.040610	-0.04143
ad_000	-0.000801	-0.000372	-0.001568	1.000000	-0.000156	-0.000190	-0.000038	-0.000085	-0.000202	-0.000407	...	-0.001343	-0.001350	-0.001318	-0.00109
ae_000	0.023778	0.034129	-0.004005	-0.000156	1.000000	0.830001	-0.000465	-0.000953	0.000977	0.018881	...	0.009860	0.016805	0.010357	0.02387
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
ee_007	0.356954	0.010747	-0.048884	-0.000722	0.034168	0.043477	0.007652	0.067999	0.181955	0.475604	...	0.442405	0.433122	0.419859	0.38036
ee_008	0.259626	-0.026319	-0.004806	-0.001060	-0.009518	-0.012225	0.000102	0.005364	0.009142	0.020924	...	0.442837	0.470860	0.448713	0.39435
ee_009	0.133790	-0.017057	0.012019	-0.000622	-0.007330	-0.009004	-0.001082	-0.003530	-0.009065	-0.018009	...	0.224864	0.236117	0.221077	0.18413
ef_000	0.005648	0.051733	-0.006200	-0.000072	0.063622	0.093517	-0.000086	0.000960	0.005119	0.010094	...	0.002271	0.008862	0.001752	0.02564
eg_000	0.009239	0.077262	0.016273	-0.000076	0.030893	0.044417	0.000145	0.003464	0.012881	0.009612	...	0.002475	0.005299	0.005932	0.00715

170 rows × 170 columns

In [58]:
```
plt.figure(figsize=(18,15))
sns.heatmap(df.corr())
plt.show()
```

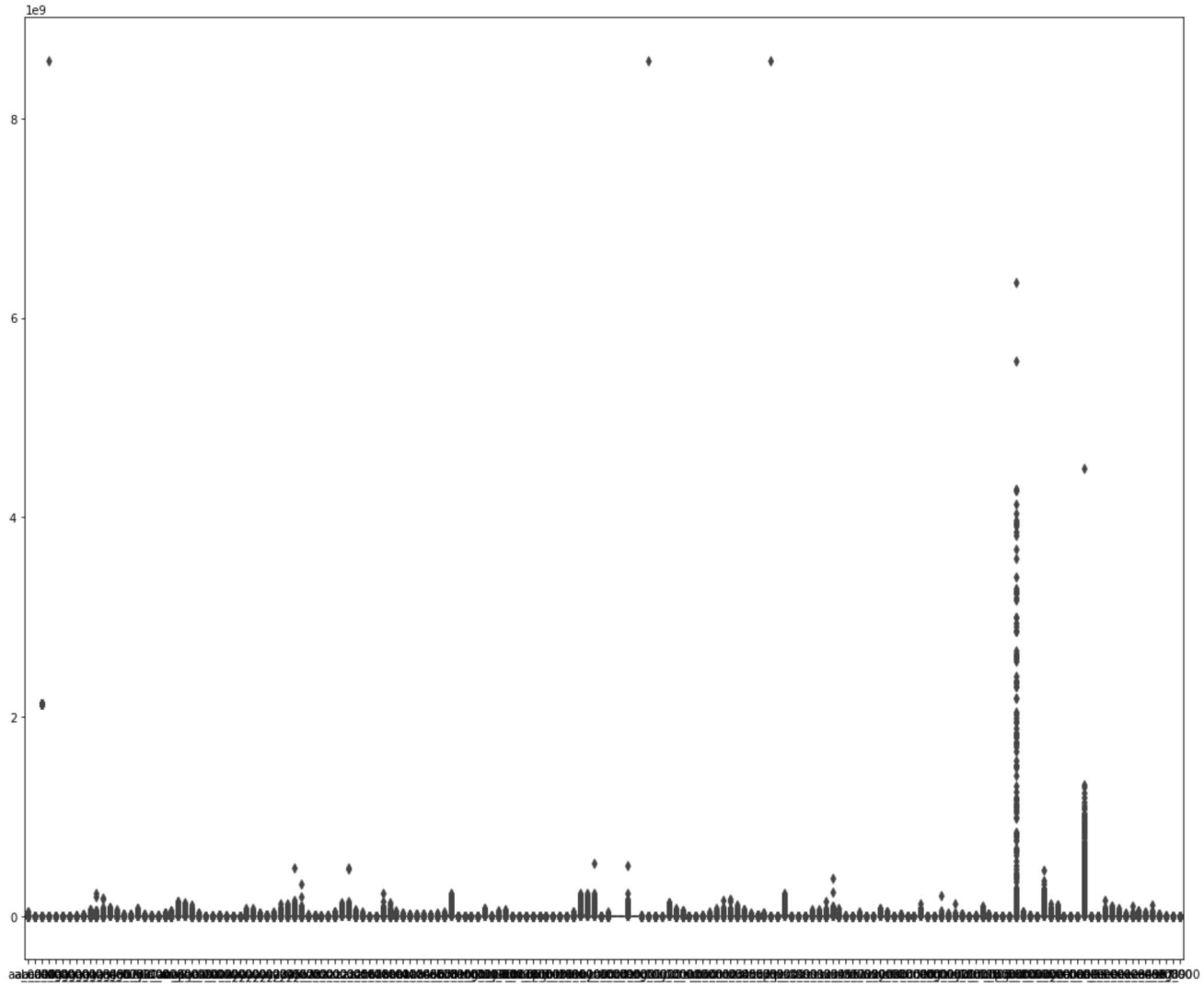


In [59]:
```
for feature in num_columns:
    plt.figure(figsize=(15,6))
    plt.subplot(121)
    sns.histplot(data=df, x=feature, kde=True, bins=30)
    plt.title("{}'s distribution".format(feature),fontweight="bold", fontsize=15)

    plt.subplot(122)
    sns.boxplot(data=df, x=feature, color='blue')
    plt.title("{}'s BoxPlot".format(feature),fontweight="bold", fontsize=15)
    plt.show()
```

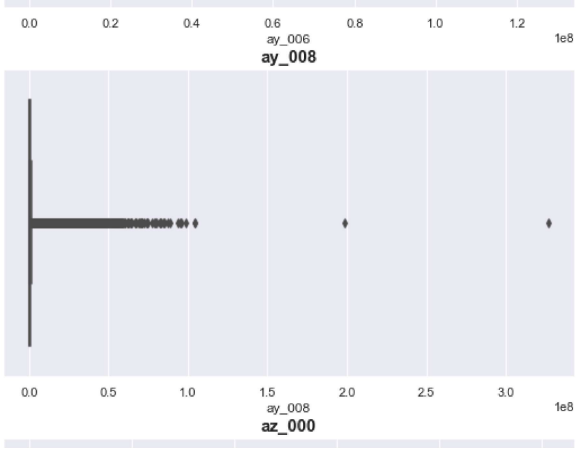


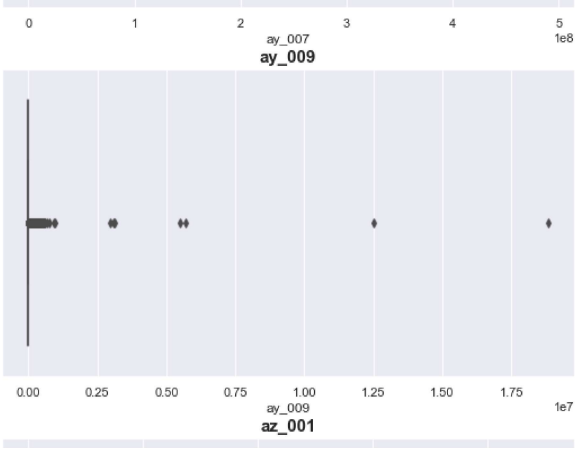
Boxplot

In [60]:
```
plt.figure(figsize=(18,15))
sns.boxplot(data=df , orient='v')
plt.show()
```



In [61]:
```
### Checking outliers in numerical features

plt.figure(figsize=(20,510))
for feature in enumerate(num_columns):
    plt.subplot(85, 2, feature[0]+1)
    sns.set(rc={'figure.figsize':(10,6)})
    sns.boxplot(data=df, x=feature[1], color='red')
    plt.title("{}".format(feature[1]), fontweight="bold", fontsize=15)
```



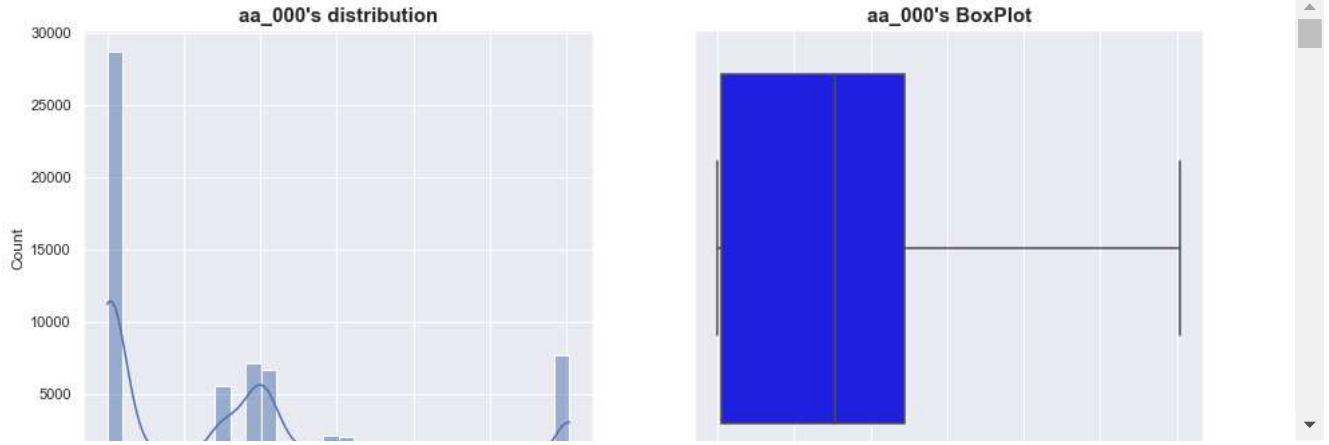


Handle Outliers

In [62]:
```
### Function to Cap Outlier
def remove_outliers(in_data, in_col):
    # Finding the IQR
    first_quartile = in_data[in_col].quantile(0.25)
    third_quartile = in_data[in_col].quantile(0.75)
    iqr = third_quartile - first_quartile
    upper_limit = third_quartile + 1.5 * iqr
    lower_limit = first_quartile - 1.5 * iqr
    in_data.loc[(in_data[in_col]>upper_limit), in_col]= upper_limit
    in_data.loc[(in_data[in_col]<lower_limit), in_col]= lower_limit
    return in_data
```

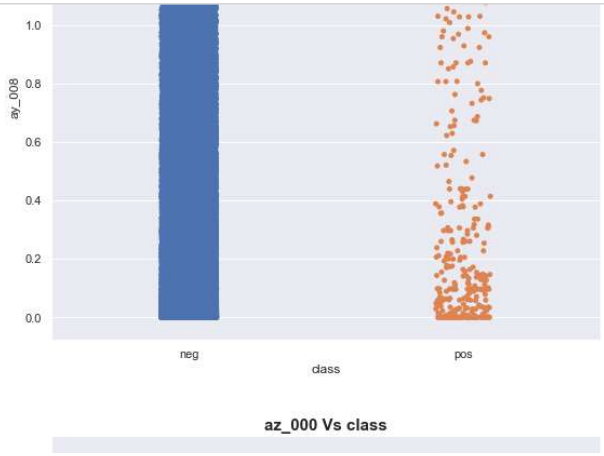
In [63]:
```
### Capping outliers from continuous features
for feature in num_columns:
    dataset=remove_outliers(df, feature)
```

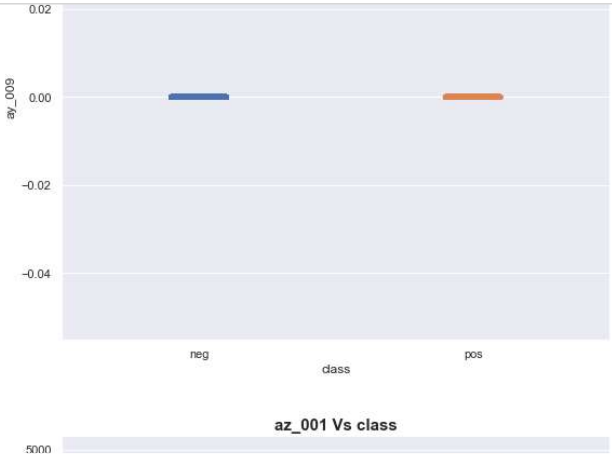
In [64]:
```
for feature in num_columns:
    plt.figure(figsize=(15,6))
    plt.subplot(121)
    sns.histplot(data=df, x=feature, kde=True, bins=30)
    plt.title("{}'s distribution".format(feature),fontweight="bold", fontsize=15)

    plt.subplot(122)
    sns.boxplot(data=df, x=feature, color='blue')
    plt.title("{}'s BoxPlot".format(feature),fontweight="bold", fontsize=15)
    plt.show()
```



Numerical Feature vS Target Feature

In [184]:
```
### Checking distribution of Numerical features with respect to target feature class
plt.figure(figsize=(20,840))
for feature in enumerate(num_columns):
    plt.subplot(85, 2, feature[0]+1)
    sns.set(rc={'figure.figsize':(8,10)})
    sns.stripplot(data=df, y=feature[1], x='class')
    plt.title("{} Vs class".format(feature[1]), fontsize=15, fontweight='bold')
```





Dependent and Independent Feature

In [67]:
```
# Dependent Feature
X=df.drop('class',axis=1)
X
```

Out[67]:

	aa_000	ab_000	ac_000	ad_000	ae_000	af_000	ag_000	ag_001	ag_002	ag_003	...	ee_002	ee_003	ee_004	ee_005	ee_006	ee_007	ee_008
0	76698	0	2090	280	0	0	0	0	0	0	...	1083002	493384	721044.0	469792	339156.0	157956	73224.00
1	33058	0	0	128	0	0	0	0	0	0	...	421400	178064	293306.0	245416	133654.0	81140	97576.00
2	41040	0	228	100	0	0	0	0	0	0	...	277378	159812	423992.0	409564	320746.0	158022	95128.00
3	12	0	70	66	0	0	0	0	0	0	...	240	46	58.0	44	10.0	0	0.00
4	60874	0	1368	458	0	0	0	0	0	0	...	622012	229790	405298.0	347188	286954.0	311560	341053.75
...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...	...
75995	81852	0	2090	667	0	0	0	0	0	0	...	632658	273242	510354.0	373918	349840.0	317840	341053.75
75996	18	0	52	46	0	0	0	0	0	0	...	266	44	46.0	14	2.0	0	0.00
75997	79636	0	1670	667	0	0	0	0	0	0	...	806832	449962	778826.0	581558	375498.0	222866	341053.75
75998	110	0	36	32	0	0	0	0	0	0	...	588	210	180.0	544	1004.0	1338	74.00
75999	8	0	6	4	0	0	0	0	0	0	...	46	10	48.0	14	42.0	46	0.00

76000 rows × 170 columns

In [ ]:
```
```

In [68]:
```
# Independent Feature
y=df['class']
y
```

Out[68]:
```
0        neg
1        neg
2        neg
3        neg
4        neg
        ...
75995    neg
75996    neg
75997    neg
75998    neg
75999    neg
Name: class, Length: 76000, dtype: object
```

In [69]:
```
y.head()
```

Out[69]:
```
0    neg
1    neg
2    neg
3    neg
4    neg
Name: class, dtype: object
```

In [70]:
```
y.unique()
```

Out[70]:
```
array(['neg', 'pos'], dtype=object)
```

In [71]:
```
### encoding target feature
y=y.replace({'neg':0, 'pos':1})
```

In [72]:
```
y.unique()
```

Out[72]:
```
array([0, 1], dtype=int64)
```

Split_Train_Test

In [76]:
```
from sklearn.model_selection import train_test_split
```

In [78]:
```
X_train,X_test,y_train,y_test=train_test_split(X,y,test_size=0.35 , random_state=10)
```

In [79]:
```
X_train.shape , y_train.shape
```

Out[79]:
```
((49400, 170), (49400,))
```

In [81]:
```
X_test.shape , y_test.shape
```

Out[81]:
```
((26600, 170), (26600,))
```

Model Traning

In [91]:
```
from sklearn.preprocessing import StandardScaler , MinMaxScaler, RobustScaler
from sklearn.linear_model import LogisticRegression
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier , GradientBoostingClassifier
from xgboost import XGBClassifier
import xgboost

#from sklearn.metrics import accuracy_score , classification_report , ConfusionMatrix
from sklearn.metrics import precision_score , recall_score , f1_score , roc_auc_score

from sklearn.model_selection import train_test_split , RepeatedStratifiedKFold

from sklearn.compose import ColumnTransformer
```

In [92]:
```
# Logistic Regression
from sklearn.linear_model import LogisticRegression
logistic=LogisticRegression()
logistic.fit(X_train , y_train)
y_pred=logistic.predict(X_test)

from sklearn.metrics import accuracy_score
accuracy_score_logistic=accuracy_score(y_test , y_pred)
print(f"Accuracy Score of Logistic Regression Model ={accuracy_score_logistic*100} %")
```

```
Accuracy Score of Logistic Regression Model =98.28571428571429 %
```

In [94]:
```
# confusion Matricx
from sklearn.metrics import confusion_matrix
conf_mat_log=confusion_matrix(y_test , y_pred)
print(conf_mat_log)
tp,tn,fp,fn=conf_mat_log.ravel()
```

```
[[26046    64]
 [  392    98]]
```

In [96]:
```
conf_mat_accuracy=(tp+tn)/(tp+fp+fn+tn)

Precision=tp/(tp+fp)
Recall = tp/(tp+fn)
f1_score = 2*(Precision * Recall)/(Precision + Recall)
Error_rate=(fp+fn) / (tp+tn+fp+fn)
print(f"Accuracy score confusion matricx ={round(conf_mat_accuracy * 100,2)} %")
print(f" f1 Score  = {f1_score}")
print(f"Precision ={Precision}")
print(f" Recall ={Recall}")
print(f"Misclassification Rate  = {Error_rate}")
```

```
Accuracy score confusion matricx =98.16 %
 f1 Score  = 0.9906812217108516
Precision =0.9851728572509267
 Recall =0.9962515299877601
Misclassification Rate  = 0.018421052631578946
```

2. Model Training by using Support Vector Classifier

In [99]:
```
from sklearn.svm import SVC
sv_class=SVC()
sv_class.fit(X_train, y_train)
y_pred_sv_class=sv_class.predict(X_test)

from sklearn.metrics import accuracy_score
accuracy_score_svc=accuracy_score(y_test , y_pred_sv_class)
print(f"Accuracy Sore of Support Vector Classification ={accuracy_score_svc}")
```

```
Accuracy Sore of Support Vector Classification =0.9816165413533835
```

In [101]:
```
# Confusion Matrics
from sklearn.metrics import confusion_matrix
conf_mat_svc=confusion_matrix(y_test,y_pred_sv_class)
print(conf_mat_svc)
tp,tn,fp,fn=conf_mat_svc.ravel()
```

```
[[26110     0]
 [  489     1]]
```

In [103]:
```
conf_mat_accuracy=(tp+tn)/(tp+tn+fp+fn)
Precision = tp/(tp+fp)
Recall = tp/(tp+fn)
f1_score=2*(Precision * Recall) / (Precision+Recall)
Error_rate= (fp+fn) / (tp+fp+tn+fn)
print(f"Accuracy using Confusion matrix ={round(conf_mat_accuracy*100,2)} %")
print(f"Recall Score = {f1_score}")
print(f"Precision = {Precision}")
print(f"F1 Score ={f1_score}")
print(f"Error Rate ={Error_rate}")
print(f"Misclassification Rate ={Error_rate}")
```

```
Accuracy using Confusion matrix =98.16 %
Recall Score = 0.9907038512616202
Precision = 0.981615850219933
F1 Score =0.9907038512616202
Error Rate =0.018421052631578946
Misclassification Rate =0.018421052631578946
```

3. MOdel Training by using Decision Tree Classifier

In [105]:
```
from sklearn.tree import DecisionTreeClassifier

model_tree=DecisionTreeClassifier()
model_tree.fit(X_train,y_train)
y_pred_tree=model_tree.predict(X_test)

accuracy_score_tree=accuracy_score(y_test,y_pred_tree)
print(f" Accuracy Score of Decision Tree Classifier ={accuracy_score_tree}")
```

```
 Accuracy Score of Decision Tree Classifier =0.9842857142857143
```

In [106]:
```
# Confusion Matrics

conf_mat_tree=confusion_matrix(y_test, y_pred_tree)
print(conf_mat_tree)
tp,fp,tn,fn = conf_mat_tree.ravel()
```

```
[[25891   219]
 [  199   291]]
```

In [108]:
```
conf_mat_accuracy=(tp+fp)/(tp+fp+tn+fp)

Precision = tp/(tp+fp)
Recall = tp/(tp+fn)
f1_score= 2*(Precision * Recall)/(Precision+ Recall)
Error_rate = (fp+fn)/(tp+fp+tn+fp)
print(f" Accuarcy using Confusion Matrix ={ round(conf_mat_accuracy * 100,2)}%")
print(f" Precision  ={Precision}")
print(f" Recall ={Recall}")
print(f"F1 Score ={f1_score}")
print(f"Error rate ={ Error_rate}")
print(f" Misclassification Rate ={Error_rate}")
```

```
 Accuarcy using Confusion Matrix =98.42%
 Precision  =0.9916124090386825
 Recall =0.9888854938507371
F1 Score =0.9902470741222367
Error rate =0.01922496984318456
 Misclassification Rate =0.01922496984318456
```

4. Model Training by using Random Forest Classifier

In [145]:
```
from sklearn.ensemble import RandomForestClassifier
random_class=RandomForestClassifier()

random_class.fit(X_train,y_train)
y_pred_random_class=random_class.predict(X_test)

from sklearn.metrics import accuracy_score
accuracy_score_random_class= accuracy_score(y_test,y_pred_random_class)
print(f" Accuracy score of Randon Forest Classifier ={accuracy_score_random_class}")
```

```
 Accuracy score of Randon Forest Classifier =0.991015037593985
```

In [120]:
```
# Confusion Matrics

conf_mat_random_fore= confusion_matrix(y_test, y_pred_random_class)
print(conf_mat_random_fore)
tp,fp,tn,fn = conf_mat_random_fore.ravel()
```

```
[[26084    26]
 [  216   274]]
```

In [150]:
```
conf_mat_accuracy= (tp+tn)/(tp+fp+tn+fn)

Precision=tp/(tp+fp)
Recall=tp/(tp+fn)
f1_score = 2*(Precision*Recall)/(Precision+Recall)
Error_rate= (fp+fn)/(tp+fp+tn+fn)
print(f"Accuracy using confusion matrix = {round(conf_mat_accuracy*100,2)}%")
print(f"Precision ={Precision}")
print(f"Recall = {Recall}")
print(f"F1 Score = {f1_score}")
print(f"Error Rate ={Error_rate}")
print(f" Misclassification Rate={Error_rate}")
```

```
Accuracy using confusion matrix = 98.8%
Precision =0.9984680199157411
Recall = 0.9894113628600706
F1 Score = 0.9939190605997065
Error Rate =0.011992481203007518
 Misclassification Rate=0.011992481203007518
```

5. Model Training by using Extra tree Classifier

In [148]:
```
from sklearn.ensemble import ExtraTreesClassifier
et_model=ExtraTreesClassifier()
et_model.fit(X_train,y_train)
y_pred_et_model=et_model.predict(X_test)

from sklearn.metrics import accuracy_score
accuracy_score_et_model=accuracy_score(y_test,y_pred_et_model)
print(f" Accuracy score of Extra Tree Classiier ={accuracy_score_et_model}")
```

```
 Accuracy score of Extra Tree Classiier =0.9903383458646616
```

In [126]:
```
# Confusion Matrics

conf_mat_et_model = confusion_matrix(y_test,y_pred_et_model)
print(conf_mat_et_model)
tp,fp,tn,fn=conf_mat_et_model.ravel()
```

```
[[26070    40]
 [  211   279]]
```

In [149]:
```
conf_mat_accuracy= (tp+tn)/(tp+fp+tn+fn)

Precision=tp/(tp+fp)
Recall=tp/(tp+fn)
f1_score = 2*(Precision*Recall)/(Precision+Recall)
Error_rate= (fp+fn)/(tp+fp+tn+fn)
print(f"Accuracy using Confusion Matrix = {round(conf_mat_accuracy*100,2)}")
print(f"Precision ={Precision}")
print(f"Recall = {Recall}")
print(f"F1 Score = {f1_score}")
print(f"Error Rate ={Error_rate}")
print(f" Misclassification Rate={Error_rate}")
```

```
Accuracy using Confusion Matrix = 98.8
Precision =0.9984680199157411
Recall = 0.9894113628600706
F1 Score = 0.9939190605997065
Error Rate =0.011992481203007518
 Misclassification Rate=0.011992481203007518
```

6. Model Training by Using Bagging Classifier

In [151]:
```
from sklearn.ensemble import BaggingClassifier

bagg_model=BaggingClassifier()
bagg_model.fit(X_train,y_train)
y_pred_bagg_model=bagg_model.predict(X_test)

accuracy_score_bagg_model=accuracy_score(y_test,y_pred_bagg_model)
print(accuracy_score_bagg_model)
print(f"Accuracy Score of Bagging  Claasifier ={accuracy_score_bagg_model}")
```

```
0.9892481203007519
Accuracy Score of Bagging  Claasifier =0.9892481203007519
```

In [166]:
```
# Confusion Model

conf_mat_bagg_model = confusion_matrix(y_test,y_pred_bagg_model)
print(conf_mat_bagg_model)
tp,fp,tn,fn=conf_mat_bagg_model.ravel()
```

```
[[26054    56]
 [  230   260]]
```

In [ ]:

In [167]:
```
conf_mat_bagg_model= (tp+tn)/(tp+fp+tn+fn)

Precision=tp/(tp+fp)
Recall=tp/(tp+fn)
f1_score = 2*(Precision*Recall)/(Precision+Recall)
Error_rate= (fp+fn)/(tp+fp+tn+fn)
print(f"Accuracy using Confusion Matrix = {round(conf_mat_bagg_model*100)}")
print(f"Precision ={Precision}")
print(f"Recall = {Recall}")
print(f"F1 Score = {f1_score}")
print(f"Error Rate ={Error_rate}")
print(f" Misclassification Rate={Error_rate}")
```

```
Accuracy using Confusion Matrix = 99
Precision =0.9978552278820375
Recall = 0.9901193281143118
F1 Score = 0.9939722264611628
Error Rate =0.0118796992481203
 Misclassification Rate=0.0118796992481203
```

7. Model Traning by using AdaBoost Classifier

In [158]:
```
from sklearn.ensemble  import AdaBoostClassifier
ada_model=AdaBoostClassifier()
ada_model.fit(X_train,y_train)
y_pred_ada=ada_model.predict(X_test)

from sklearn.metrics import accuracy_score
accuracy_score_ada=accuracy_score(y_test,y_pred_ada)
print(f"Accuracy score of Ada BoostClaasifier={accuracy_score_ada}")
```

```
Accuracy score of Ada BoostClaasifier=0.9865413533834586
```

In [159]:
```
# confusion matrix

conf_mat_ada=confusion_matrix(y_test,y_pred_ada)
print(conf_mat_ada)
tp,fp,tn,fn=conf_mat_ada.ravel()
```

```
[[26007   103]
 [  255   235]]
```

In [162]:
```
conf_mat_accuarcy=(tp+tn)/(tp+fp+tn+fn)

Precision = tp/(tp+fp)
Recall=tp/(tp+tn)
f1_score=2*(Precision*Recall)/(Precision+Recall)
Error_rate=(fp+fn)/(tp+fp+tn+fn)
print(f"Accuarcy using confusion matrix={round(conf_mat_accuracy*100)}")
print(f"Precision ={Precision}")
print(f"Recall={Recall}")
print(f"F1 Score={f1_score}")
print(f"Error_rate={Error_rate}")
```

```
Accuarcy using confusion matrix=99
Precision =0.9960551512830333
Recall=0.9902901530728809
F1 Score=0.9931642862598336
Error_rate=0.012706766917293232
```

8. Model Trainig using by Gradient Boosting Classifier

In [171]:
```
from sklearn.ensemble import GradientBoostingClassifier
boost_model=GradientBoostingClassifier()
boost_model.fit(X_train,y_train)
y_pred_boost_model=boost_model.predict(X_test)

accuracy_score_boost_model=accuracy_score(y_test,y_pred_boost_model)
print(f"Accuracy score  of GradientBoostClassifier={GradientBoostingClassifier}")
```

```
Accuracy score  of GradientBoostClassifier=<class 'sklearn.ensemble._gb.GradientBoostingClassifier'>
```

In [182]:
```
# confusion matrix

conf_mat_boost=confusion_matrix(y_test,y_pred_boost_model)
print(conf_mat_boost)
tp,fp,tn,fn=conf_mat_boost.ravel()
```

```
[[26047    63]
 [  228   262]]
```

In [183]:
```
conf_mat_accuracy=(tp+fp)/(tp+fp+tn+fn)

Precision=(tp)/(tp+fp)
Recall=(tp)/(tp+fn)
f1_score=2*(Precision*Recall)/(Precision+Recall)
Error_rate=(fp+fn)/(tp+fp+tn+fn)
print(f"Accuracy Score of confusion matrix={round(conf_mat_accuracy*100)}%")
print(f"Precision={Precision}")
print(f"Recall={Recall}")
print(f"F1 Score={f1_score}")
print(f"Error Rate ={Error_rate}")
print(f"MisClassification Rtae ={Error_rate}")
```

```
Accuracy Score of confusion matrix=98%
Precision=0.9975871313672923
Recall=0.9900414306891178
F1 Score=0.9937999580304852
Error Rate =0.012218045112781954
MisClassification Rtae =0.012218045112781954
```

9.Extreme GradientBoostingClassifier Model ---XGB

In [176]:
```
import xgboost
from xgboost import XGBClassifier

model_xgb=xgboost.XGBClassifier()
model_xgb.fit(X_train,y_train)
y_pred_xgb=model_xgb.predict(X_test)

accuracy_score_xgb=accuracy_score(y_test,y_pred_xgb)
print(f" Accuracy score of XGB model ={accuracy_score_xgb}")
```

```
 Accuracy score of XGB model =0.9921428571428571
```

In [180]:
```
# confusion matrix

conf_mat_xgb=confusion_matrix(y_test,y_pred_xgb)
print(conf_mat_xgb)
tp,fp,tn,fn=conf_mat_xgb.ravel()
```

```
[[26062    48]
 [  161   329]]
```

In [181]:
```
conf_mat_accuracy=(tp+fp)/(tp+fp+tn+fn)

Precision=(tp)/(tp+fp)
Recall=(tp)/(tp+fn)
f1_score=2*(Precision*Recall)/(Precision+Recall)
Error_rate=(fp+fn)/(tp+fp+tn+fn)
print(f"Accuracy Score of confusion matrix={round(conf_mat_accuracy*100)}%")
print(f"Precision={Precision}")
print(f"Recall={Recall}")
print(f"F1 Score={f1_score}")
print(f"Error Rate ={Error_rate}")
print(f"MisClassification Rtae ={Error_rate}")
```

```
Accuracy Score of confusion matrix=98%
Precision=0.9981616238988893
Recall=0.9875336288886363
F1 Score=0.9928191843964876
Error Rate =0.014172932330827068
MisClassification Rtae =0.014172932330827068
```

Final Result of All Classifier Model

In [190]:
```
print(f"1.Accuracy Score of Logistic Regression Model ={accuracy_score_logistic*100} %")
print(f"2.Accuracy Sore of Support Vector Classification ={accuracy_score_svc*100}%")
print(f"3. Accuracy Score of Decision Tree Classifier ={accuracy_score_tree*100}%")
print(f"4. Accuracy score of Randon Forest Classifier ={accuracy_score_random_class*100}%")
print(f"5. Accuracy score of Extra Tree Classiier ={accuracy_score_et_model*100}%")
print(f"6.Accuracy Score of Bagging  Claasifier ={accuracy_score_bagg_model*100} %")
print(f"7.Accuracy score of Ada BoostClaasifier={accuracy_score_ada*100} %")
print(f"8.Accuracy score  of GradientBoostClassifier={GradientBoostingClassifier} %")
print(f"9. Accuracy score of XGB model ={accuracy_score_xgb*100} %")
```

```
1.Accuracy Score of Logistic Regression Model =98.28571428571429 %
2.Accuracy Sore of Support Vector Classification =98.16165413533835%
3. Accuracy Score of Decision Tree Classifier =98.42857142857143%
4. Accuracy score of Randon Forest Classifier =99.1015037593985%
5. Accuracy score of Extra Tree Classiier =99.03383458646616%
6.Accuracy Score of Bagging  Claasifier =98.92481203007519 %
7.Accuracy score of Ada BoostClaasifier=98.65413533834587 %
8.Accuracy score  of GradientBoostClassifier=<class 'sklearn.ensemble._gb.GradientBoostingClassifier'> %
9. Accuracy score of XGB model =99.21428571428571 %
```

Final Report

Heighest Accuracy Score is XGB Classifier 99.2142%

Thank you.....

In [ ]: