In [1]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import warnings
warnings.filterwarnings('ignore')

%matplotlib inline
```

In [2]:

```
training_data = pd.read_csv("../Data/aps_failure_training_set.csv", na_values="na")
training_data.head()
```

Out[2]:

	class	aa_000	ab_000	ac_000	ad_000	ae_000	af_000	ag_000	ag_001	ag_002	...
0	neg	76698	NaN	2.130706e+09	280.0	0.0	0.0	0.0	0.0	0.0	...
1	neg	33058	NaN	0.000000e+00	NaN	0.0	0.0	0.0	0.0	0.0	...
2	neg	41040	NaN	2.280000e+02	100.0	0.0	0.0	0.0	0.0	0.0	...
3	neg	12	0.0	7.000000e+01	66.0	0.0	10.0	0.0	0.0	0.0	...
4	neg	60874	NaN	1.368000e+03	458.0	0.0	0.0	0.0	0.0	0.0	...

5 rows × 171 columns

In [3]:

```
sns.set_style('whitegrid')
```

Preprocessing and data cleaning

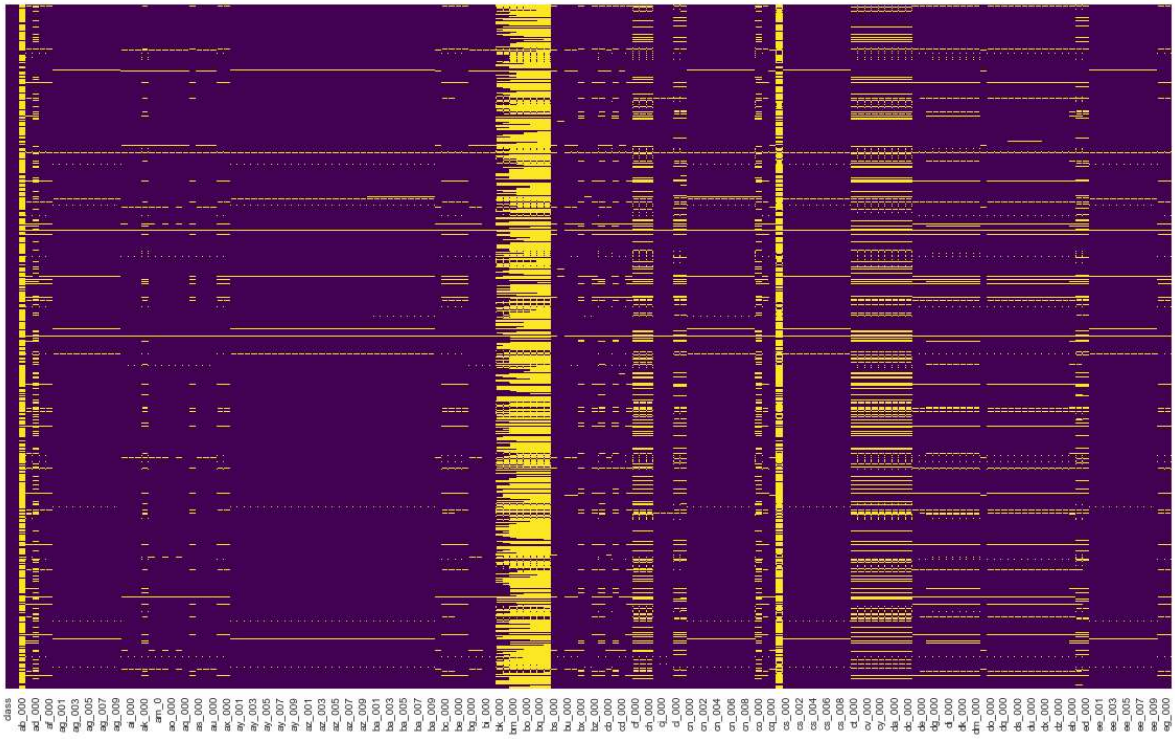
In [4]:

```
#for fininding missing values
#this headmap is illustrating the missing values with yellow bars

plt.figure(figsize=(20,12))
sns.heatmap(training_data.isnull(),yticklabels=False,cbar=False,cmap='viridis')
```

Out[4]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x19068d7fd68>
```



Missing value handling

We are going to use different approches with missing values:

1. Removing the column having more than 10,000 missing values (**Ref)
2. Removing the column having 80% missing values (**Self intuition)
3. Keeping all the features
4. Later, we will try to implement some feature engineering

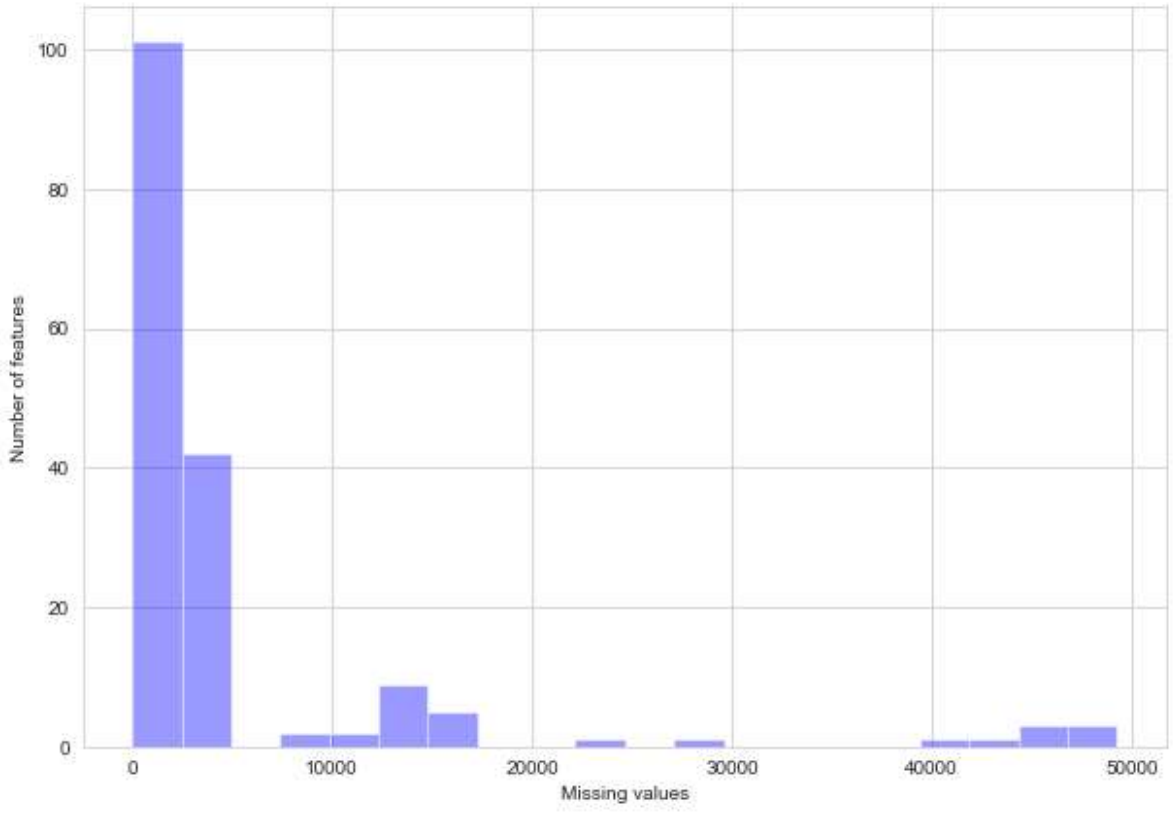
For the rest of the missing values, we are replacing them with their mean() for now (Ref)

First Approach

In [5]:

```
missing_data = training_data.isna().sum().to_frame()

plt.figure(figsize=(10,7))
ax= sns.distplot(missing_data[0],kde = False,bins = 20,color='b')
ax.set(xlabel='Missing values', ylabel='Number of features')
plt.show()
```



In [6]:

```
missing_data[missing_data[0]>10000]
```

Out[6]:

	0
ab_000	46329
ad_000	14861
bk_000	23034
bl_000	27277
bm_000	39549
bn_000	44009
bo_000	46333
bp_000	47740
bq_000	48722
br_000	49264
cf_000	14861
cg_000	14861
ch_000	14861
co_000	14861
cr_000	46329
ct_000	13808
cu_000	13808
cv_000	13808
cx_000	13808
cy_000	13808
cz_000	13808
da_000	13808
db_000	13808
dc_000	13808
ec_00	10239

we are dropping the columns that have more than 10,000 missing values

In [7]:

```
#sample_data varible holds the data for approach one
temp = "ab_000 ad_000 bk_000 bl_000 bm_000 bn_000 bo_000 bp_000 bq_000 br_000 cf_000 cg_000
temp = temp.split()

sample_data = training_data.drop(temp,axis=1)
sample_data.head()
```

Out[7]:

	class	aa_000	ac_000	ae_000	af_000	ag_000	ag_001	ag_002	ag_003	ag_004	...
0	neg	76698	2.130706e+09	0.0	0.0	0.0	0.0	0.0	0.0	37250.0	...
1	neg	33058	0.000000e+00	0.0	0.0	0.0	0.0	0.0	0.0	18254.0	...
2	neg	41040	2.280000e+02	0.0	0.0	0.0	0.0	0.0	0.0	1648.0	...
3	neg	12	7.000000e+01	0.0	10.0	0.0	0.0	0.0	318.0	2212.0	...
4	neg	60874	1.368000e+03	0.0	0.0	0.0	0.0	0.0	0.0	43752.0	...

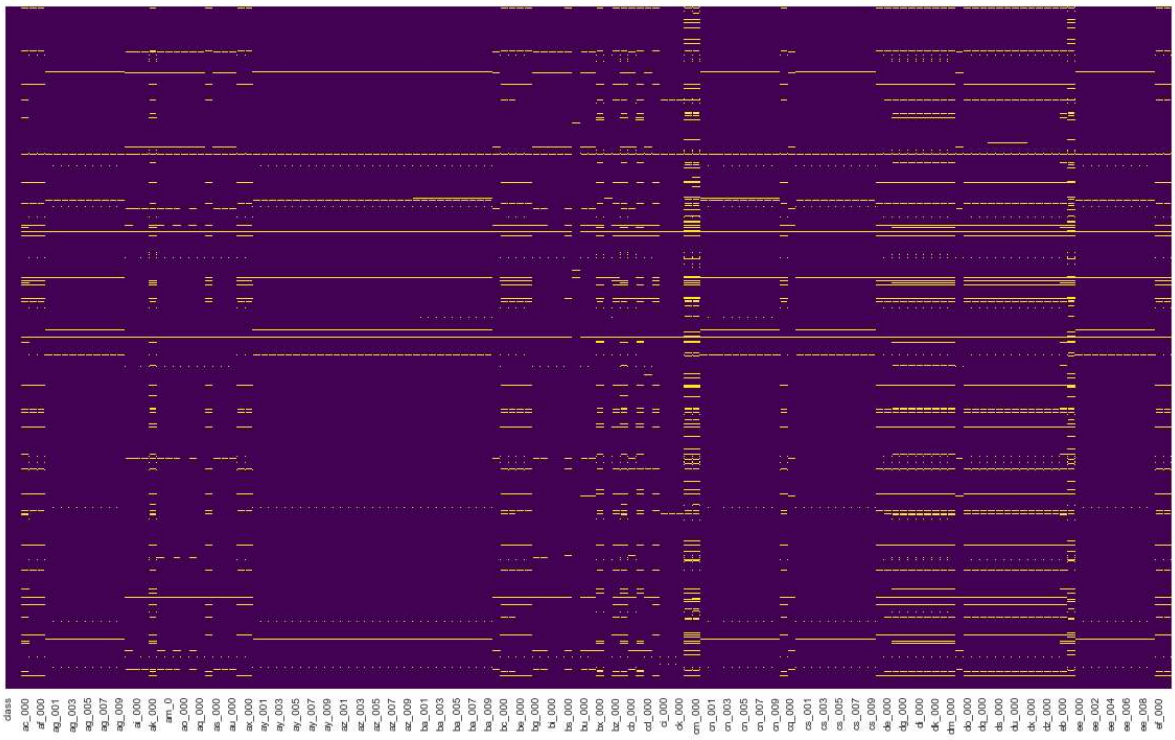
5 rows × 146 columns

In [8]:

```
#after dropping columns having more than 10,000 missing values
plt.figure(figsize=(20,12))
sns.heatmap(sample_data.isnull(),yticklabels=False,cbar=False,cmap='viridis')
```

Out[8]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x190005c4278>
```



Here we can see much improvement in the heatmap

For the rest of the values we are going to replace them with their mean() -- (**Ref)

In [9]:

```
sample_data.fillna(sample_data.mean(),inplace=True)
```

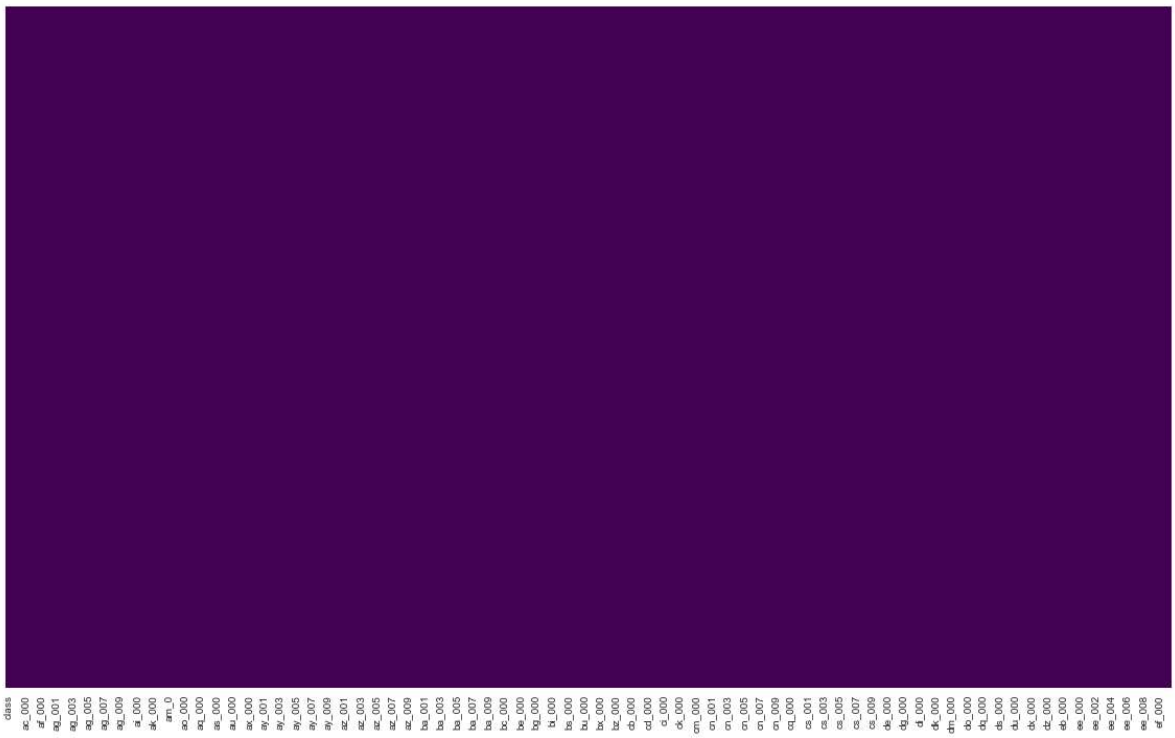
After mean(), the heatmap for missing values will look like

In [10]:

```
plt.figure(figsize=(20,12))
sns.heatmap(sample_data.isnull(),yticklabels=False,cbar=False,cmap='viridis')
```

Out[10]:

```
<matplotlib.axes._subplots.AxesSubplot at 0x190001adbe0>
```



So we have handled the missing value problem in this approach

Now We need to handle the classification variables(class) by taking a dummy value instead of a string

In [11]:

```
#as all the other values are numerical except Class column so we can replace them with 1 an

sample_data = sample_data.replace('neg',0)
sample_data = sample_data.replace('pos',1)

sample_data.head(15)
```

Out[11]:

	class	aa_000	ac_000	ae_000	af_000	ag_000	ag_001	ag_002	ag_003	ag_004
0	0	76698	2.130706e+09	0.0	0.0	0.0	0.0	0.0	0.0	37250.0
1	0	33058	0.000000e+00	0.0	0.0	0.0	0.0	0.0	0.0	18254.0
2	0	41040	2.280000e+02	0.0	0.0	0.0	0.0	0.0	0.0	1648.0
3	0	12	7.000000e+01	0.0	10.0	0.0	0.0	0.0	318.0	2212.0
4	0	60874	1.368000e+03	0.0	0.0	0.0	0.0	0.0	0.0	43752.0
5	0	38312	2.130706e+09	0.0	0.0	0.0	0.0	0.0	0.0	9128.0
6	0	14	6.000000e+00	0.0	0.0	0.0	0.0	0.0	0.0	1202.0
7	0	102960	2.130706e+09	0.0	0.0	0.0	0.0	0.0	0.0	2130.0
8	0	78696	0.000000e+00	0.0	0.0	0.0	0.0	0.0	0.0	458.0
9	1	153204	1.820000e+02	0.0	0.0	0.0	0.0	0.0	11804.0	684444.0
10	0	39196	2.040000e+02	0.0	0.0	0.0	0.0	0.0	0.0	4352.0
11	0	45912	0.000000e+00	0.0	0.0	0.0	0.0	0.0	0.0	2106.0
12	0	2104	3.600000e+01	0.0	0.0	0.0	0.0	9744.0	13148.0	98310.0
13	0	118950	1.390000e+03	0.0	0.0	0.0	0.0	0.0	0.0	40932.0
14	0	24416	0.000000e+00	0.0	0.0	0.0	0.0	0.0	0.0	556.0

15 rows × 146 columns

Model implementation

First Approach

As our data is ready, so here we are going to try to model and fit our dataset with logistic regression model

In [12]:

```
#here the predictors (X) and responce (y) is separated from the sample_data for this model

X = sample_data.drop('class',axis=1)
y = sample_data['class']
```

In [13]:

```
X.head()
```

Out[13]:

	aa_000	ac_000	ae_000	af_000	ag_000	ag_001	ag_002	ag_003	ag_004	ag_005
0	76698	2.130706e+09	0.0	0.0	0.0	0.0	0.0	0.0	37250.0	1432864.0
1	33058	0.000000e+00	0.0	0.0	0.0	0.0	0.0	0.0	18254.0	653294.0
2	41040	2.280000e+02	0.0	0.0	0.0	0.0	0.0	0.0	1648.0	370592.0
3	12	7.000000e+01	0.0	10.0	0.0	0.0	0.0	318.0	2212.0	3232.0
4	60874	1.368000e+03	0.0	0.0	0.0	0.0	0.0	0.0	43752.0	1966618.0

5 rows × 145 columns

In [14]:

```
y.head()
```

Out[14]:

```
0    0
1    0
2    0
3    0
4    0
Name: class, dtype: int64
```

Although we have our separate test data, but for chekcking this time we are going to split out training data into (train,test) where test data will have 30% portion of the entire training data

In [15]:

```
from sklearn.model_selection import train_test_split
```

In [16]:

```
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.3, random_state=101)
```

In [17]:

```
from sklearn.linear_model import LogisticRegression
```

In [18]:

```
logmodel = LogisticRegression()
```

In [19]:

```
#fitting the data
logmodel.fit(X_train,y_train)
```

Out[19]:

```
LogisticRegression(C=1.0, class_weight=None, dual=False, fit_intercept=True,
          intercept_scaling=1, max_iter=100, multi_class='warn',
          n_jobs=None, penalty='l2', random_state=None, solver='warn',
          tol=0.0001, verbose=0, warm_start=False)
```

In [20]:

```
prediction = logmodel.predict(X_test)
```

In [21]:

```
from sklearn.metrics import classification_report
```

In [22]:

```
print(classification_report(y_test,prediction))
```

```
              precision    recall  f1-score   support

           0       0.99      1.00      1.00     17691
           1       0.78      0.61      0.68       309

   micro avg       0.99      0.99      0.99     18000
   macro avg       0.89      0.80      0.84     18000
weighted avg       0.99      0.99      0.99     18000
```

Here we can see, Negative class prediction is about 99% correct , where Positive class prediction guarantees upto 78%

Now, for the cost

In [23]:

```
from sklearn.metrics import confusion_matrix
```

In [52]:

```
tn, fp, fn, tp = confusion_matrix(y_test,prediction).ravel()
```

In [70]:

```
cost = 10*fp+500*fn
```

In [72]:

```
values = {'Score':[cost],'Number of Type 1 faults':[fp],'Number of Type 2 faults':[fn]}
pd.DataFrame(values)
```

Out[72]:

	Score	Number of Type 1 faults	Number of Type 2 faults
0	61530	53	122

By observing this , we can see that estimation for "Number of Type 1 faults" is quite good , on the other hand estimation for "Number of Type 2 faults " is not that good

Visualization

Classification Report

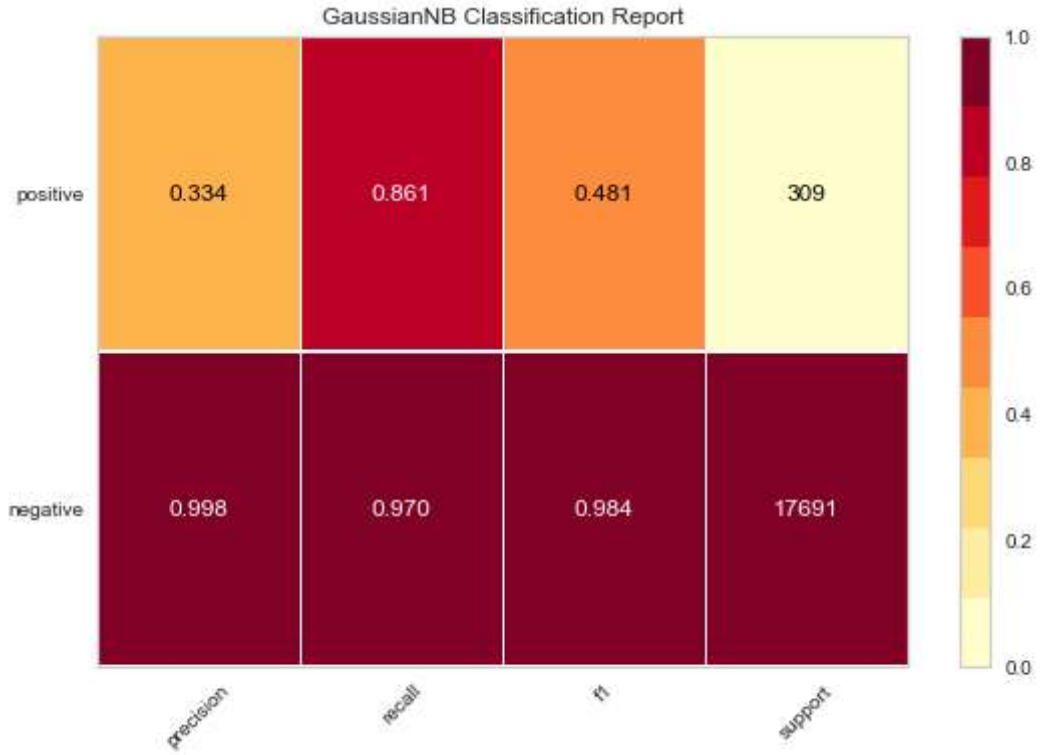
In [102]:

```
from sklearn.naive_bayes import GaussianNB
from yellowbrick.classifier import ClassificationReport
```

In [178]:

```
# Instantiate the classification model and visualizer
bayes = GaussianNB()
visualizer = ClassificationReport(bayes, classes=["negative", "positive"], support=True)

visualizer.fit(X_train, y_train)  # Fit the visualizer and the model
visualizer.score(X_test, y_test)  # Evaluate the model on the test data
g = visualizer.poof()             # Draw/show/poof the data
```



In [165]:

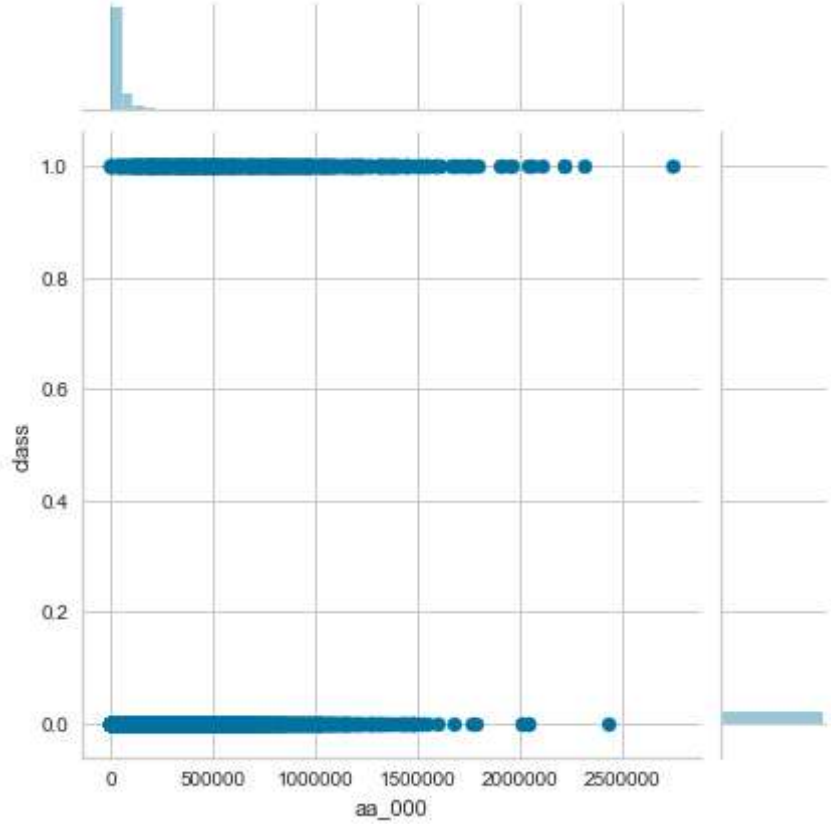
```
temp =sample_data.columns.tolist()
```

```
aa_000
```

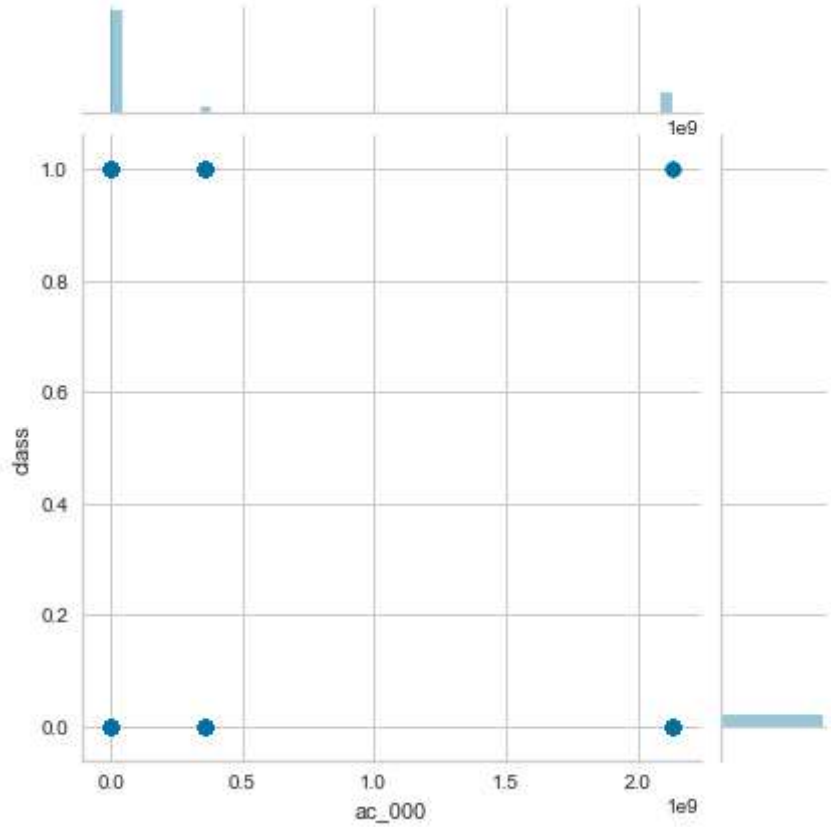
In [182]:

```
for i in range(10):
    plt.figure(figsize=(7,5))
    sns.jointplot(temp[i+1],'class',data = sample_data)
    print()
```

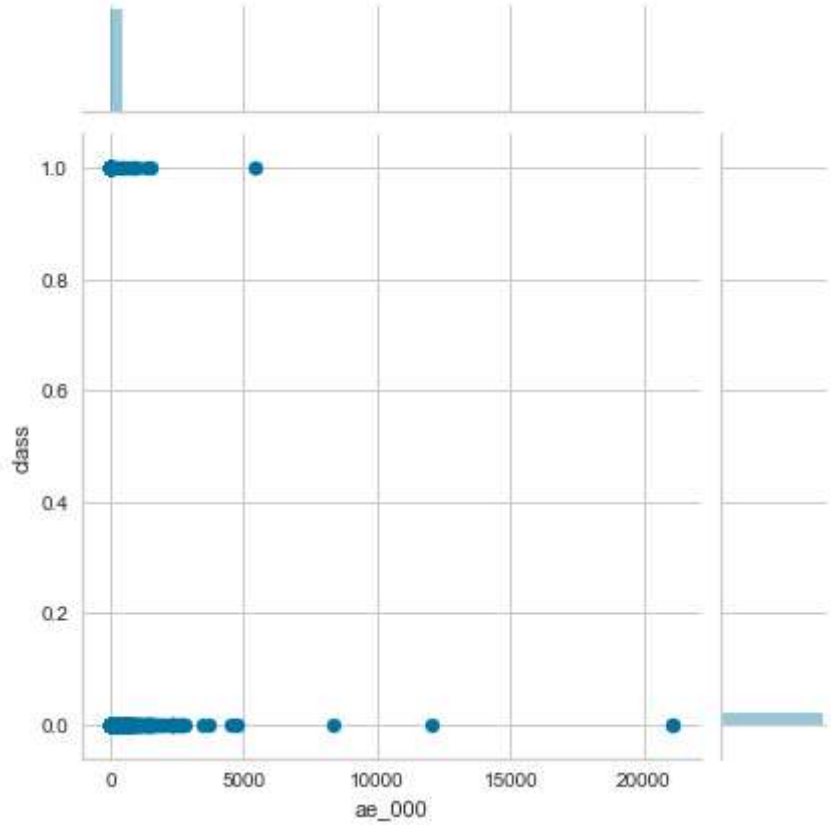
<Figure size 504x360 with 0 Axes>



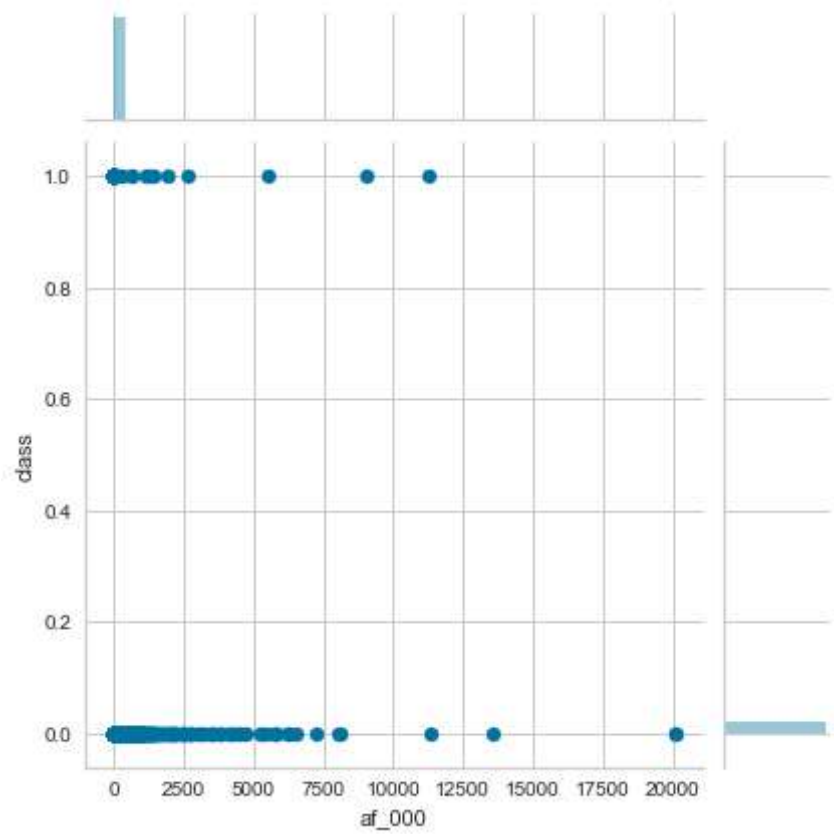
<Figure size 504x360 with 0 Axes>



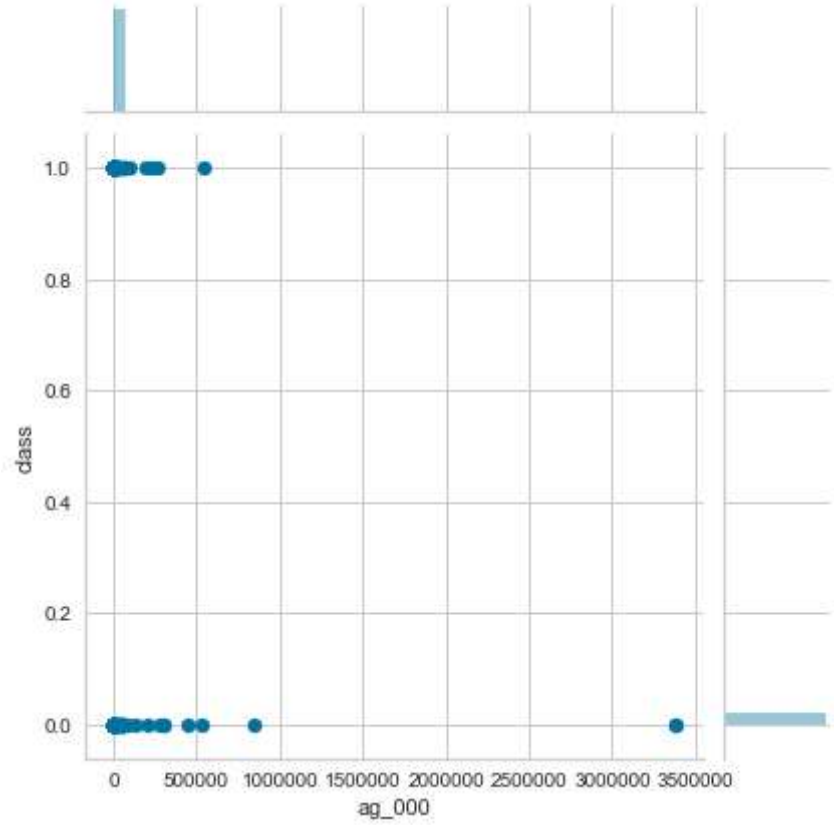
<Figure size 504x360 with 0 Axes>



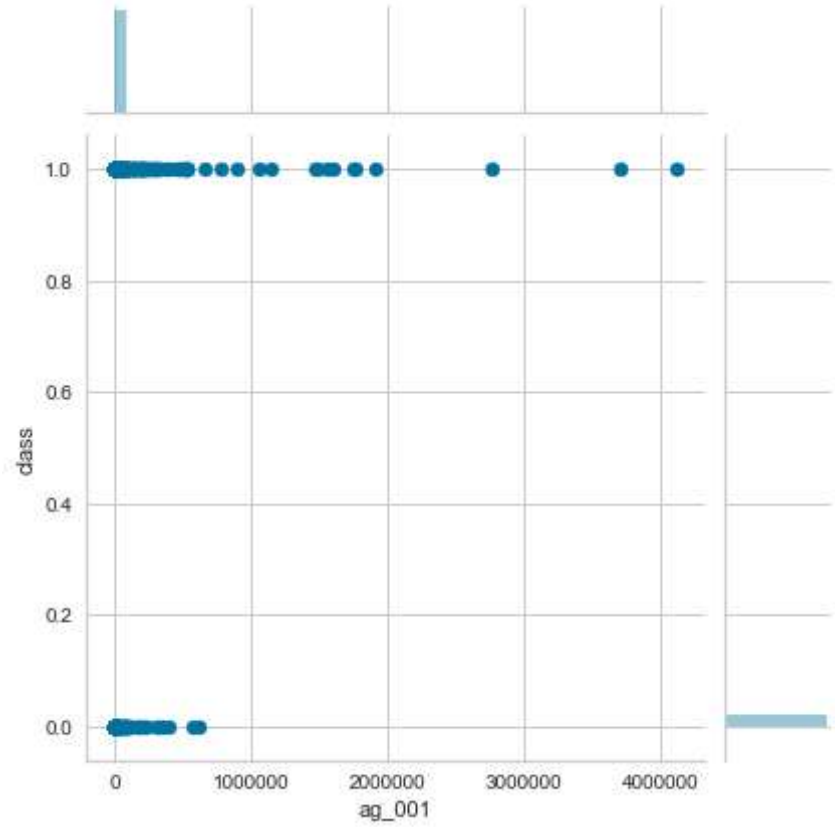
<Figure size 504x360 with 0 Axes>



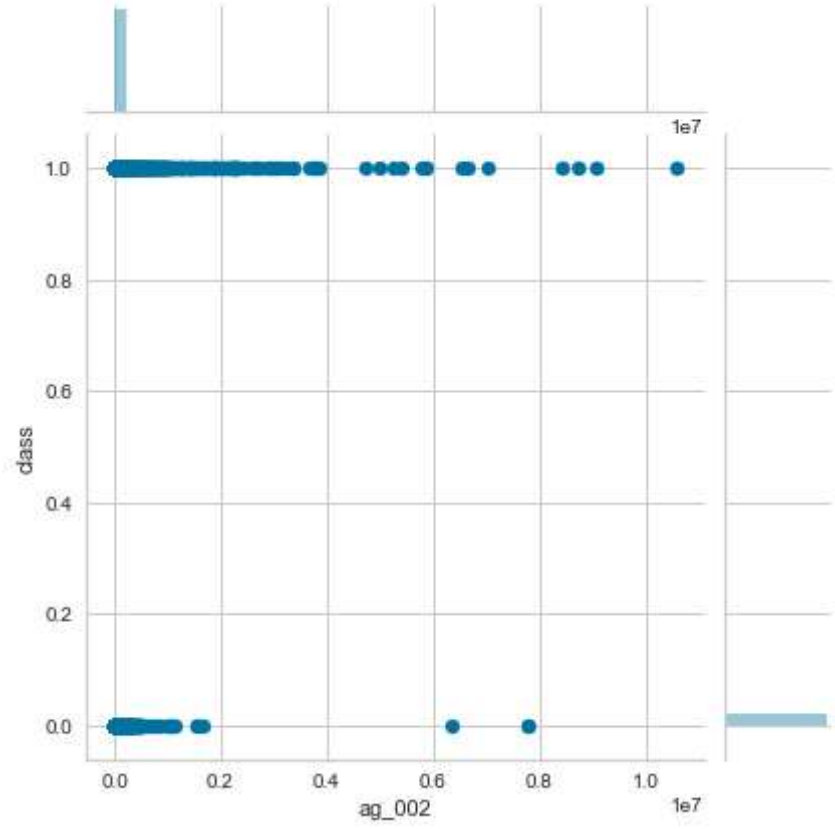
<Figure size 504x360 with 0 Axes>



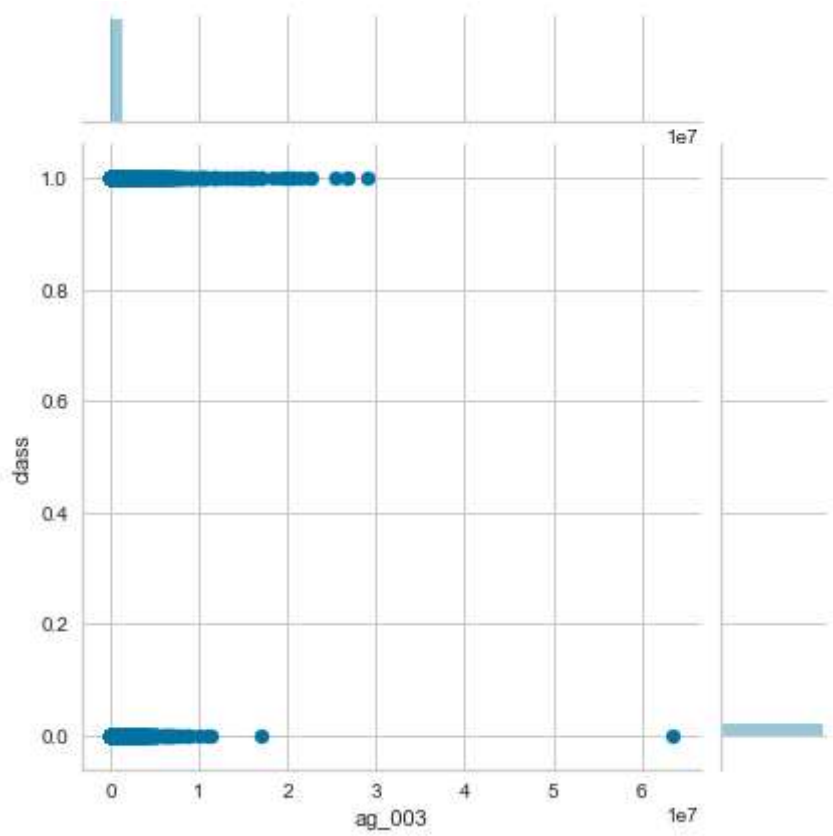
<Figure size 504x360 with 0 Axes>



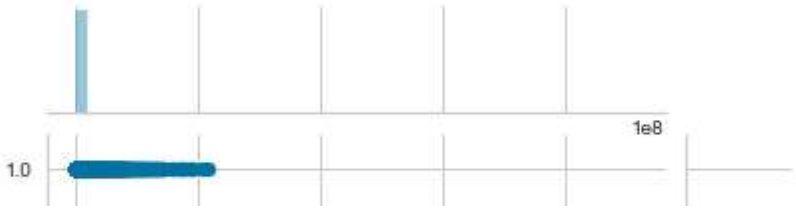
<Figure size 504x360 with 0 Axes>



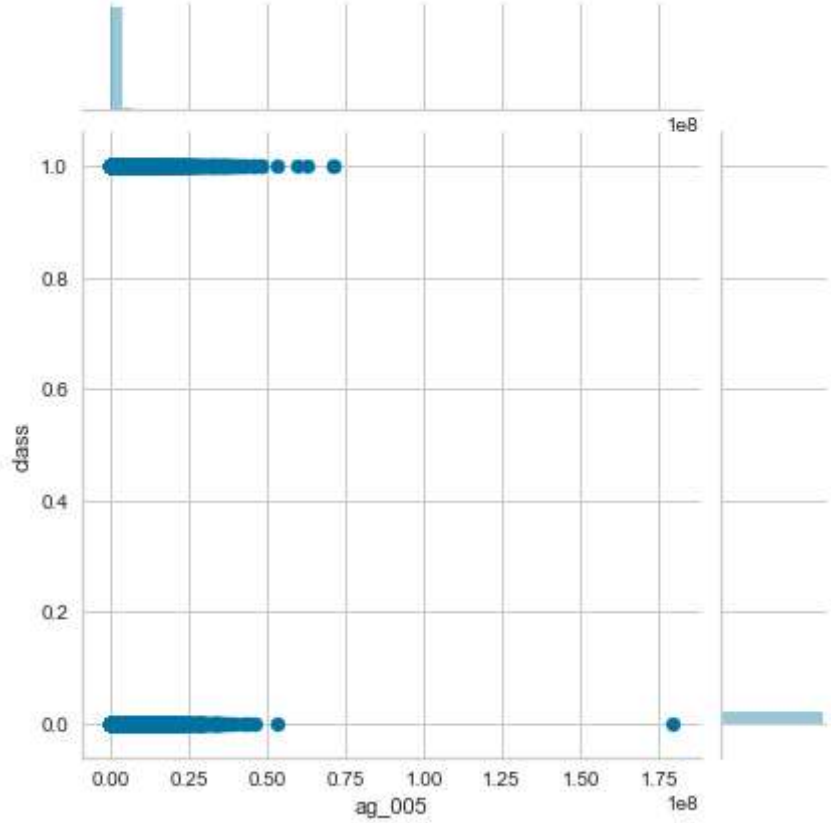
<Figure size 504x360 with 0 Axes>




```
<Figure size 504x360 with 0 Axes>
```




```
<Figure size 504x360 with 0 Axes>
```



In [ ]: