Problem statement

- For the given different type of result dataset we need to apply our custom build performance metrics and write down our observations.

PERFORMANCE METRICS

In [75]:

```
import numpy as np
import pandas as pd
# other than these two packages we did not import any other packages
```

1)Confusion matrix

In [76]:

```
def confusion_matrix(df):
    TP=0
    TN=0
    FP=0
    FN=0
    for index,row in df.iterrows():                          #iterate over each row of dataframe
        if row["y"]==row["proba"] and row["y"]==1:           #if predicted and actual values are equal to 1, add one to TRUE POSITIV
            TP+=1
        elif row["y"]==row["proba"] and row["y"]==0:
            TN+=1                                            # #if predicted and actual values are equal to 0, add one to TRUE NEGAT
        elif row["y"]!=row["proba"] and row["y"]==0:
            FP+=1                                            #if predicted and actual values are NOT equal and actual value is zero,
        else:
            FN+=1                                            #if predicted and actual values are NOT equal and actual value is one,
    C=[TP,FP,FN,TN]
    return np.reshape(C,(2,2))
```

2)f1 score

In [77]:

```
def f1_score(df):
    TP=0
    TN=0
    FP=0
    FN=0
    pr=0
    re=0
    for index,row in df.iterrows():                        #loop to calulate TRUE POSITIVE,TRUE NEGATIVE,FALSE POSITIVE,FALSE NEGATI
        if row["y"]==row["proba"] and row["y"]==1:
            TP+=1
        elif row["y"]==row["proba"] and row["y"]==0:
            TN+=1
        elif row["y"]!=row["proba"] and row["y"]==0:
            FP+=1
        else:
            FN+=1
    C=[TP,FP,FN,TN]

    pr=TP/(TP+FP)                #precision formula
    re=TP/(TP+FN)                #recall formula

    return (2*pr*re)/(pr+re)
```

3)auc

In [78]:

```
from tqdm import tqdm
def auc(df):
    df_sorted = df.sort_values('proba', ascending=False)      #sorting df based on probability score
    df_sorted.reset_index(drop=True,inplace=True)
    TP=0
    TN=0
    FP=0
    FN=0
    TPR=[]
    FPR=[]
    actual = df_sorted['y'].tolist()
    proba = [0]*len(actual)                                 #initialise all values as zero so that we can change every value based

    for i in tqdm(range(0,len(actual))):
        proba[i]=1                                          #at each iteration change proba value from zero to one at each element
        for j in range(0,len(actual)):
            if actual[j]==proba[j] and actual[j]==1:
                TP+=1
            elif actual[j]==proba[j] and actual[j]==0:
                TN+=1
            elif actual[j]!=proba[j] and actual[j]==0:
                FP+=1
            else:
                FN+=1

        tpr=(TP/(TP+FN))
        fpr=(FP/(FP+TN))
        TPR.append(tpr)
        FPR.append(fpr)
        TP=0
        TN=0
        FP=0
        FN=0

    return np.trapz(TPR, FPR)                               #use trapizoidal rule to calculate area under the ROC curve drawn by TP
                                                            #https://stackoverflow.com/questions/39537443/how-to-calculate-a-partia
```

4)Accuracy

In [79]:

```
def accuracy(df):
    TP=0
    TN=0
    FP=0
    FN=0
    pr=0
    re=0
    for index,row in df.iterrows():
        if row["y"]==row["proba"] and row["y"]==1:
            TP+=1
        elif row["y"]==row["proba"] and row["y"]==0:
            TN+=1
        elif row["y"]!=row["proba"] and row["y"]==0:
            FP+=1
        else:
            FN+=1
    C=[TP,FP,FN,TN]

    return (TP+TN)/(TP+FP+FN+TN)                        #formula for accuracy
```

5)Custom metric

In [80]:

```
from tqdm import tqdm
def Custom_metric(df_sorted):
    TP=0
    TN=0
    FP=0
    FN=0
    TPR=[]
    FPR=[]
    actual = df_sorted['y'].tolist()
    original_proba=df_sorted["proba"].tolist()              #observed value
    new_proba = [0]*len(actual)                             #initialise all values as zero so that we can change every value ba
    best_threshold=0
    A=0

    for i in tqdm(range(0,len(actual)-1)):

        new_proba[i]=1                                      #at each iteration change proba value from zero to one at each ele

        for j in range(0,len(actual)-1):
            if actual[j]==new_proba[j] and actual[j]==1:
                TP+=1
            elif actual[j]==new_proba[j] and actual[j]==0:
                TN+=1
            elif actual[j]!=new_proba[j] and actual[j]==0:
                FP+=1
            else:
                FN+=1

        A=(500*FN)+(100*FP)                 #custom metric which penalises the FALSE NEGATIVE more compared to FALSE POSITIVE

        if i==0:
            A_best=A

        if A_best>A:
            A_best=A
            best_threshold=original_proba[i]            #best_threshold variable stores,for which probability threshold value re
            best_index=i

        TP=0
        TN=0
        FP=0
        FN=0
    return best_threshold
```

6)MSE

In [81]:

```
def MSE(df_d):

    y = df_d['y'].tolist()          #converting df columns to list
    pred = df_d['pred'].tolist()

    mse=0
    sq_residual=0

    for i in tqdm(range(0,len(y))):
        sq_residual+=(y[i]-pred[i])**2          #sum of squares of residual
    mse=sq_residual/(len(y))
    return mse
```

7) modificed MAPE

In [82]:

```
def MAPE(df_d):

    y = df_d['y'].tolist()
    pred = df_d['pred'].tolist()

    mape=0
    residual=0
    sum_of_actual=0

    for i in tqdm(range(0,len(y))):
        residual+=abs(y[i]-pred[i])            #absolute value of residual
        sum_of_actual+=y[i]                    #to avoid "divide by zero error", we use sum of actual on the denominator
    mape=(residual/sum_of_actual)*100
    return mape
```

8) r squared error

In [83]:

```
def R_sq(df_d):

    y = df_d['y'].tolist()
    pred = df_d['pred'].tolist()

    r_sq=0
    sum_sq_residual=0
    total_sum_of_squares=0
    e=0

    for i in range(0,len(y)):
        e+=y[i]
    y_bar=e/len(y)                  #mean of observed data

    for i in tqdm(range(0,len(y))):
        sum_sq_residual+=(y[i]-pred[i])**2      #sum of squares of residual
        total_sum_of_squares+=(y[i]-y_bar)**2   #total sum of squares

    R_sq=1-(sum_sq_residual/total_sum_of_squares)
    return R_sq
```

5_a1 - Confusion matrix

This data has number of positive points >> number of negatives points

In [84]:

```
import numpy as np
import pandas as pd
# other than these two you should not import any other packages# 5_A
```

In [85]:

```
df_a=pd.read_csv('5_a.csv')
df_a.head(5)
```

Out[85]:

	y	proba
0	1.0	0.637387
1	1.0	0.635165
2	1.0	0.766586
3	1.0	0.724564
4	1.0	0.889199

In [86]:

```
df_a.loc[(df_a.proba >= 0.5 ), 'proba'] = 1    #having threshold as 0.5 and what ever probability value greater than or equal to
df_a.loc[(df_a.proba < 0.5 ), 'proba'] = 0
```

In [87]:

```
confusion_matrix(df_a)
```

Out[87]:

```
array([[10000,   100],
       [    0,     0]])
```

observation

- this data we can see number of positive points >> number of negatives points...
- we can see that for the given 5_a.csv data ,all the predicted value are 1 when the threshold is 0.5

5_a2- f1 score

In [88]:

```
f1_score(df_a)
```

Out[88]:

```
0.9950248756218906
```

observation

- we have FN as 0 so the recall is 100%...so even though we have FP as 100 (not a 100% precision) we get good f1 score
- for the highly imbalanced dataset and we are having only 100 FP, we are getting higher f1 score...so for this dataset we can use f1 score if we care more about recall than precision..

5_a3 - Auc

In [89]:

```
df_a=pd.read_csv('5_a.csv')     #reread the data since previouly we have modified our proba column values based on threshold

auc(df_a)
```

```
100%|██████████████████████████████████████████████████████████████████████████| 10100/10100 [00:20<00:00, 497.21i
t/s]
```

Out[89]:

```
0.48829900000000004
```

observation

- AUC tells how much the model is capable of distinguishing between classes
- we can see that for the given prediction of all value as 1 ,we are getting lower auc(which is close to random model where auc=0.5)...when AUC is 0.5, it means the model has no class separation capacity whatsoever.so our model(auc=0.488) all so does not have much class separation capacity
- for the highly imbalanced dataset and we are having 100 FP, we are getting lower auc...so we can say that for the highly imbalance data we can use auc metric. since auc even peanalise for the FP=100 when there is FN=0 and TN=0...

5_a4 -- accuracy

In [90]:

```
df_a.loc[(df_a.proba >= 0.5 ), 'proba'] = 1    #having threshold as 0.5 and what ever probability value greater than or equal to
df_a.loc[(df_a.proba < 0.5 ), 'proba'] = 0

accuracy(df_a)
```

Out[90]:

```
0.9900990099009901
```

observation

- though for the highly imbalanced dataset and we are having 100 FP, we are getting higher accuracy...since accuracy not giving much weightage to class with very few datapoints....so we can say that for the highly imbalance data, we can avoid using accuracy metric.

5_b1---Confusion matrix

This data has number of positive points << number of negatives points

In [91]:

```
df_b=pd.read_csv('5_b.csv')
df_b.head(5)
```

Out[91]:

	y	proba
0	0.0	0.281035
1	0.0	0.465152
2	0.0	0.352793
3	0.0	0.157818
4	0.0	0.276648

In [92]:

```
df_b.loc[(df_b.proba >= 0.5 ), 'proba'] = 1         #replacing every probability score to 1 where predicted probality score is mo
df_b.loc[(df_b.proba < 0.5 ), 'proba'] = 0
```

In [93]:

```
confusion_matrix(df_b)
```

Out[93]:

```
array([[  55,  239],
       [  45, 9761]])
```

observation

- this data we can see number of positive points << number of negatives points...

5_b2 ---f1 score

In [94]:

```
f1_score(df_b)
```

Out[94]:

```
0.2791878172588833
```

observation

- here FN and FP are high...so the recall and precision are low...so fl score is low
- even for the highly imbalanced dataset and since recall and precision are low, we are getting low f1 score...so fl score is prefered for imbalance dataset

5_b3---auc

In [95]:

```
df_b=pd.read_csv('5_b.csv')

auc(df_b)
```

```
100%|██████████████████████████████████████████████████████████████████████████| 10100/10100 [00:25<00:00, 398.28i
t/s]
```

Out[95]:

```
0.9377570000000001
```

observation

- AUC tells how much the model is capable of distinguishing between classes
- here our model has 93.77% chance of predicting correcly
- for the highly imbalanced dataset , we are getting good auc value since our model classifies positive point TP=55 even though positive class has lesser number of datapoint...so auc is prefered for imbalance dataset

5_b4--accuracy

In [66]:

```
df_b.loc[(df_b.proba >= 0.5 ), 'proba'] = 1         #replacing every probability score to 1 where predicted probality score is mo
df_b.loc[(df_b.proba < 0.5 ), 'proba'] = 0
accuracy(df_b)
```

Out[66]:

```
0.9718811881188119
```

observation

- though for the highly imbalanced dataset and we are having less FP=239 and FN=45 compared to TP+TN of 10000 datapoints , we are getting higher accuracy...since accuracy not giving much weightage to class with very few datapoints....so we can say that for the highly imbalance data, we can avoid using accuracy metric.

5_c--custom metric

In [67]:

```
df_c=pd.read_csv('5_c.csv')
df_c.head(5)
```

Out[67]:

	y	proba
0	0	0.458521
1	0	0.505037
2	0	0.418652
3	0	0.412057
4	0	0.375579

In [68]:

```
df_c_sorted = df_c.sort_values('proba', ascending=False)
df_c_sorted.reset_index(drop=True,inplace=True)
```

In [69]:

```
Custom_metric(df_c_sorted)
```

```
100%|██████████████████████████████████████████████████████████████████████████| 2851/2851 [00:02<00:00, 1048.10i
t/s]
```

Out[69]:

```
0.230039028
```

observation

- for the probability value of 0.230039028 we are getting small value for our custom metric for the given data...

In [70]:

```
#drawing confusion matric with the optimal threshold

df_c_sorted.loc[(df_c_sorted.proba >= 0.230039028 ), 'proba'] = 1
df_c_sorted.loc[(df_c_sorted.proba <0.230039028), 'proba'] = 0
confusion_matrix(df_c_sorted)
```

Out[70]:

```
array([[ 969, 1020],
       [  78,  785]])
```

5_d reading data from csv

In [71]:

```
df_d=pd.read_csv('5_d.csv')
df_d.head()
```

Out[71]:

	y	pred
0	101.0	100.0
1	120.0	100.0
2	131.0	113.0
3	164.0	125.0
4	154.0	152.0

5_d1 Mean Square Error

In [72]:

```
MSE(df_d)
```

```
100%|██████████████████████████████████████████████████████████████████████| 157200/157200 [00:00<00:00, 1260752.55i
t/s]
```

Out[72]:

```
177.16569974554707
```

Observation

- we can not interprete since the upper limit is infinity..
- since the errors are squared before taking average ,it peanalises more if the error is larger compared to other errors(ie,outlier)

5_d2 MAPE

In [73]:

```
MAPE(df_d)
```

```
100%|██████████████████████████████████████████████████████████████████████| 157200/157200 [00:00<00:00, 1437652.01i
t/s]
```

Out[73]:

```
12.91202994009687
```

Observation

- the average deviation between the forecasted value and actual values was 12.9%

5_d3 R^2 error

In [74]:

```
R_sq(df_d)
```

```
100%|██████████████████████████████████████████████████████████████████████| 157200/157200 [00:00<00:00, 933976.70i
t/s]
```

Out[74]:

```
0.9563582786990964
```

Observation

- the linear regression model fits the data well and explains 95.6% variance of dependent variable....