

# CHAPTER 1: DEFINITION OF THE PROBLEM STATEMENT AND EXPLORATORY DATA ANALYSIS

## INTRODUCTION TO OLA CABS

Ola Cabs is an Indian multinational ridesharing company, headquartered in Bangalore. ANI Technologies is the holding company of Ola Cabs. It also operates Ola Fleet, Ola Financial Services, Ola Foods and Ola Electric. Ola Cabs is India's largest mobility platform and one of the world's largest ride hailing companies, serving 250+ cities across India, Australia, New Zealand and the UK.

Ola app offers mobility solutions by connecting customers to drivers and a wide range of vehicles across bikes, auto-rickshaws, metered taxis and cabs, enabling convenience and transparency for hundreds of millions of customers and over 1.5 million driver partners

## DEFINITION OF THE PROBLEM

Recruiting and retaining the drivers is one of tough task for Ola Industry managers. Churn among drivers is high and it's very easy for drivers to stop working for the service on the fly or jump to its competitor 'Uber' depending on the rates.

As the companies get bigger, the high churn could become a bigger problem. To find new drivers, Ola is casting a wide net, including people who don't have cars for jobs also. But this acquisition is really costly. Losing drivers frequently impacts the morale of the organisation and acquiring new drivers is more expensive than retaining existing ones.

Assume You are working as a data scientist with the analytics department of Ola, focused on driver team attrition. You are provided with the monthly information for a segment of drivers for 2019 and 2020.

Task is `to predict whether a driver will be leaving the company or not, based on their attributes like

1. Demographics (city, age, gender etc.,)
2. Tenure information (joining date, last date)
3. Historical data regarding the performance of the driver (Quarterly rating, Monthly business acquired, grade, income)

## IMPORTING THE LIBRARIES AND DATASET

Importing all the required libraries

```
import numpy as np  
import pandas as pd
```

```

import matplotlib.pyplot as plt
import seaborn as sns
from itertools import combinations,permutations
from pandas.api.types import is_datetime64_any_dtype
import math, warnings,datetime,regex,mlflow,mlflow.sklearn,time,os
from scipy import stats
from sklearn.preprocessing import
    MinMaxScaler,StandardScaler,PowerTransformer
from sklearn.model_selection import
    train_test_split,GridSearchCV,RandomizedSearchCV,KFold,learning_curve,
    validation_curve,cross_val_score,cross_validate,cross_val_predict
from sklearn.decomposition import PCA
from imblearn.over_sampling import SMOTE
from sklearn.linear_model import LogisticRegression
from sklearn.tree import
    DecisionTreeClassifier,DecisionTreeRegressor,plot_tree,export_graphviz
from six import StringIO
from sklearn.ensemble import
    RandomForestClassifier,RandomForestRegressor,BaggingClassifier,StackingClassifier,
    VotingClassifier,AdaBoostClassifier,GradientBoostingClassifier
from xgboost import XGBClassifier
from lightgbm import LGBMClassifier
from sklearn.impute import KNNImputer
from sklearn.metrics import
    accuracy_score,precision_score,recall_score,precision_recall_fscore_support,
    precision_recall_curve,ConfusionMatrixDisplay,confusion_matrix,
    roc_auc_score,roc_curve,mean_absolute_error,mean_squared_error,
    classification_report,f1_score,fbeta_score,auc
from sklearn.neural_network import MLPClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.svm import SVC
from sklearn.tree import DecisionTreeClassifier,DecisionTreeRegressor
from sklearn.ensemble import
    RandomForestClassifier,BaggingClassifier,VotingClassifier,AdaBoostClassifier,
    StackingClassifier,GradientBoostingClassifier
from xgboost import XGBClassifier
from lightgbm import LGBMClassifier
from sklearn.base import BaseEstimator, ClassifierMixin, clone
pd.set_option('display.max_columns', None)

```

## Importing the Dataset

```

ola = pd.read_csv('ola_driver_scaler.csv')

# top 10 rows of the dataset
ola.head(10)

    Unnamed: 0      MMM-YY  Driver_ID   Age  Gender City Education_Level
\
```

0	0	01/01/19	1	28.0	0.0	C23	2
1	1	02/01/19	1	28.0	0.0	C23	2
2	2	03/01/19	1	28.0	0.0	C23	2
3	3	11/01/20	2	31.0	0.0	C7	2
4	4	12/01/20	2	31.0	0.0	C7	2
5	5	12/01/19	4	43.0	0.0	C13	2
6	6	01/01/20	4	43.0	0.0	C13	2
7	7	02/01/20	4	43.0	0.0	C13	2
8	8	03/01/20	4	43.0	0.0	C13	2
9	9	04/01/20	4	43.0	0.0	C13	2

Income	Dateofjoining	LastWorkingDate	Joining	Designation	Grade	\
57387	24/12/18	NaN		1	1	
57387	24/12/18	NaN		1	1	
57387	24/12/18	03/11/19		1	1	
67016	11/06/20	NaN		2	2	
67016	11/06/20	NaN		2	2	
65603	12/07/19	NaN		2	2	
65603	12/07/19	NaN		2	2	
65603	12/07/19	NaN		2	2	
65603	12/07/19	NaN		2	2	
65603	12/07/19	27/04/20		2	2	

Total	Business Value	Quarterly	Rating
0	2381060		2
1	-665480		2
2	0		2
3	0		1
4	0		1
5	0		1
6	0		1
7	0		1
8	350000		1
9	0		1

```
ola.drop(columns=["Unnamed: 0"], inplace= True)
```

Description regarding each column of the dataset

Column Name	Description
MMM-YY	Reporting Date(Monthly)

Column Name	Description
<b>Driver_ID</b>	Unique Id for drivers
<b>Age</b>	Age of the driver
<b>Gender</b>	Gender of the driver - Male: 0, Female: 1
<b>City</b>	City Code of the driver
<b>Education_Level</b>	Education level - 0 for 10+, 1 for 12+, 2 for graduate
<b>Income</b>	Monthly average Income of the driver
<b>Dateofjoining</b>	Joining date for the driver
<b>LastWorkingDate</b>	Last date of working for the driver
<b>Joining Designation</b>	Designation of the driver at the time of joining
<b>Grade</b>	Grade of the driver at the time of reporting
<b>Total Business Value</b>	the total business value acquired by the driver in a month (negative business indicates cancellation/refund or car EMI adjustments)
<b>Quarterly Rating</b>	Quarterly rating of the driver: 1,2,3,4,5(higher is better)

## ANALYSING BASIC METRICS OF DATASET

### Shape of the data

```
print(f"Number of rows in the dataset = {ola.shape[0]}")
print(f"Number of columns in the dataset = {ola.shape[1]}")
```

Number of rows in the dataset = 19104  
Number of columns in the dataset = 13

### Datatypes of all the attributes

```
ola.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 19104 entries, 0 to 19103
Data columns (total 13 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   MMM-YY          19104 non-null   object 
 1   Driver_ID       19104 non-null   int64  
 2   Age             19043 non-null   float64
 3   Gender          19052 non-null   float64
 4   City            19104 non-null   object 
 5   Education_Level 19104 non-null   int64  
 6   Income          19104 non-null   int64  
 7   Dateofjoining  19104 non-null   object 
```

```

8  LastWorkingDate      1616 non-null   object
9  Joining Designation  19104 non-null  int64
10 Grade                19104 non-null  int64
11 Total Business Value 19104 non-null  int64
12 Quarterly Rating     19104 non-null  int64
dtypes: float64(2), int64(7), object(4)
memory usage: 1.9+ MB

```

## Observation

Should convert MMM-YY, Dateofjoining and LastWorkingDate from object dtype to datetime64 dtype

## Missing value or Null Value Detection

```

# Null value count
ola.isnull().sum()

MMM-YY                  0
Driver_ID                0
Age                     61
Gender                   52
City                      0
Education_Level           0
Income                     0
Dateofjoining             0
LastWorkingDate          17488
Joining Designation       0
Grade                     0
Total Business Value      0
Quarterly Rating          0
dtype: int64

```

```

# Percentage of Null Values
(ola.isna().sum() / ola.shape[0]) * 100

MMM-YY                  0.000000
Driver_ID                0.000000
Age                     0.319305
Gender                   0.272194
City                      0.000000
Education_Level           0.000000
Income                     0.000000
Dateofjoining             0.000000
LastWorkingDate          91.541039
Joining Designation       0.000000
Grade                     0.000000
Total Business Value      0.000000
Quarterly Rating          0.000000
dtype: float64

```

```

# Filtering the columns which have zero null values and sorting in
descending order
def missing_ola(ola):
    total_missing_ola = ola.isna().sum().sort_values(ascending =
False)
    percentage_missing_ola =
((ola.isna().sum()/len(ola)*100)).sort_values(ascending = False)
    missingola = pd.concat([total_missing_ola,
percentage_missing_ola],axis = 1, keys=['Total', 'Percent'])
    return missingola

missing_ola = missing_ola(ola)
missing_ola[missing_ola["Total"]>0]

      Total    Percent
LastWorkingDate  17488  91.541039
Age              61     0.319305
Gender           52     0.272194

```

### Observation

91.54 % Null values are present in LastWorkingDate. This implies that 91.54% drivers has not churned off from OLA. Remaining 8.46% drivers has last working date, so they have churned off from OLA.

Age and gender may be related to other features like education, Income etc., So KNN Null Value Imputation is suitable

### Descriptive Statistics regarding each column of dataset

```
ola.describe()
```

	Driver_ID	Age	Gender	Education_Level	\
count	19104.000000	19043.000000	19052.000000	19104.000000	
mean	1415.591133	34.668435	0.418749	1.021671	
std	810.705321	6.257912	0.493367	0.800167	
min	1.000000	21.000000	0.000000	0.000000	
25%	710.000000	30.000000	0.000000	0.000000	
50%	1417.000000	34.000000	0.000000	1.000000	
75%	2137.000000	39.000000	1.000000	2.000000	
max	2788.000000	58.000000	1.000000	2.000000	
	Income	Joining	Designation	Grade	Total
Business_Value \					
count	19104.000000		19104.000000	19104.000000	
1.910400e+04					
mean	65652.025126		1.690536	2.252670	
5.716621e+05					
std	30914.515344		0.836984	1.026512	
1.128312e+06					

min	10747.000000	1.000000	1.000000	-
	6.000000e+06			
25%	42383.000000	1.000000	1.000000	
	0.000000e+00			
50%	60087.000000	1.000000	2.000000	
	2.500000e+05			
75%	83969.000000	2.000000	3.000000	
	6.997000e+05			
max	188418.000000	5.000000	5.000000	
	3.374772e+07			
Quarterly Rating				
count	19104.000000			
mean	2.008899			
std	1.009832			
min	1.000000			
25%	1.000000			
50%	2.000000			
75%	3.000000			
max	4.000000			

## Observation

mean of Age --> 34.66, median of Age --> 34 (may be less number of outliers as there is small difference between mean and median)

mean of Income --> 65652.025, median of Income --> 60087 (may be more number of outliers as there is large difference between mean and median)

mean of Total Business value --> 571662.1, median of Total Business value --> 250000 (may be more number of outliers as there is large difference between mean and median)

Driver\_ID has min --> 1, max --> 2788, count --> 19104 (These metrics explain that given dataset has duplicate driver\_ids because each driver\_id will have multiple logs into the dataset with monthly frequency)

ola.describe(include = "object")	MMM-YY	City	Dateofjoining	LastWorkingDate
count	19104	19104	19104	1616
unique	24	29	869	493
top	01/01/19	C20	23/07/15	29/07/20
freq	1022	1008	192	70

Number of unique values in each column of given dataset

```
for i in ola.columns:
    print(i,":",ola[i].nunique())
```

```
MMM-YY : 24
Driver_ID : 2381
```

```
Age : 36
Gender : 2
City : 29
Education_Level : 3
Income : 2383
Dateofjoining : 869
LastWorkingDate : 493
Joining_Designation : 5
Grade : 5
Total_Business_Value : 10181
Quarterly_Rating : 4
```

## Unique values of columns whose nunique < 50

```
for i in ola.columns:
    if ola[i].nunique() < 50:
        print(i,sorted(ola[i].unique()), "", sep = "\n")

MMM-YY
['01/01/19', '01/01/20', '02/01/19', '02/01/20', '03/01/19',
 '03/01/20', '04/01/19', '04/01/20', '05/01/19', '05/01/20',
 '06/01/19', '06/01/20', '07/01/19', '07/01/20', '08/01/19',
 '08/01/20', '09/01/19', '09/01/20', '10/01/19', '10/01/20',
 '11/01/19', '11/01/20', '12/01/19', '12/01/20']

Age
[21.0, 22.0, 23.0, 24.0, 25.0, 26.0, 27.0, 28.0, 29.0, 30.0, 31.0,
 32.0, 33.0, 34.0, 35.0, 36.0, 37.0, 38.0, 39.0, 40.0, 41.0, 42.0,
 43.0, nan, 44.0, 45.0, 46.0, 47.0, 48.0, 49.0, 50.0, 51.0, 52.0, 53.0,
 54.0, 55.0, 58.0]

Gender
[0.0, 1.0, nan]

City
['C1', 'C10', 'C11', 'C12', 'C13', 'C14', 'C15', 'C16', 'C17', 'C18',
 'C19', 'C2', 'C20', 'C21', 'C22', 'C23', 'C24', 'C25', 'C26', 'C27',
 'C28', 'C29', 'C3', 'C4', 'C5', 'C6', 'C7', 'C8', 'C9']

Education_Level
[0, 1, 2]

Joining_Designation
[1, 2, 3, 4, 5]

Grade
[1, 2, 3, 4, 5]

Quarterly_Rating
```

```
[1, 2, 3, 4]
```

### Observation

Joining Designation, Grade and Quarterly Rating having Encoding denoted with numericals from 1 to 5. Where 1 being least, 5 being highest

Interestingly Quarterly rating 5 is missing in data.

Education level 10+ is indicated with 0, 12+ is indicated with 1 and graduate with 2

Conversion of MMM-YY, Dateofjoining and LastWorkingDate from object dtype to datetime64 dtype

```
ola["MMM-YY"] = pd.to_datetime(ola["MMM-YY"], format="%m/%d/%y")
ola["Dateofjoining"] = pd.to_datetime(ola["Dateofjoining"], format="%d/%m/%y")
ola["LastWorkingDate"] = pd.to_datetime(ola["LastWorkingDate"], format="%d/%m/%y")
```

### Observation

Check the format of the date properly.

For MMM-YY --> format --> MM/DD/YY so %m/%d/%y (Use %y for YY, Use %Y for YYYY)

For Dateofjoining and LastWorkingDate --> format --> DD/MM/YY so %d/%m/%y

```
ola.dtypes
```

MMM-YY	datetime64[ns]
Driver_ID	int64
Age	float64
Gender	float64
City	object
Education_Level	int64
Income	int64
Dateofjoining	datetime64[ns]
LastWorkingDate	datetime64[ns]
Joining Designation	int64
Grade	int64
Total Business Value	int64
Quarterly Rating	int64
dtype: object	

Range of values of all numerical and date columns

```
for i in ola.columns:
    if i != "City":
        print(f"Maximum of {i}", ola[i].max())
```

```

print(f"Minimum of {i}",ola[i].min())
print()

Maximum of MMM-YY 2020-12-01 00:00:00
Minimum of MMM-YY 2019-01-01 00:00:00

Maximum of Driver_ID 2788
Minimum of Driver_ID 1

Maximum of Age 58.0
Minimum of Age 21.0

Maximum of Gender 1.0
Minimum of Gender 0.0

Maximum of Education_Level 2
Minimum of Education_Level 0

Maximum of Income 188418
Minimum of Income 10747

Maximum of Dateofjoining 2020-12-28 00:00:00
Minimum of Dateofjoining 2013-01-04 00:00:00

Maximum of LastWorkingDate 2020-12-28 00:00:00
Minimum of LastWorkingDate 2018-12-31 00:00:00

Maximum of Joining Designation 5
Minimum of Joining Designation 1

Maximum of Grade 5
Minimum of Grade 1

Maximum of Total Business Value 33747720
Minimum of Total Business Value -60000000

Maximum of Quarterly Rating 4
Minimum of Quarterly Rating 1

ola["MMM-YY"].max() - ola["MMM-YY"].min()

Timedelta('700 days 00:00:00')

```

### **Observation**

MMM-YY shows that data has about 24 months readings from Jan 2019 to Dec 2020.  
(700 days)

Range of driver id is from 1 to 2788. But unique Driver Id's are 2381. So for some reason, some driver ids are missing in data. May be it is because of Driver Churn in the middle

Age eligibility for driving is in range of 21 to 58

Income has large difference between Max and min

Drivers joining date starts from 2013 April. May be drivers who joined early may earn more total business value

Last working date starts from 2018 December, Because we monthly reading starting from 2019 Jan onwards. Last date of Last working date is December 2020

Total business value is negative and positive. Negative indicates loss value, cancellation or car EMI adjustments

```
for i in ola.columns:  
    print("Value Counts of {}".format(i),end="\n\n")  
    print(ola[i].value_counts(dropna= False),end="\n\n")
```

Value Counts of MMM-YY

MMM-YY

2019-01-01	1022
2019-02-01	944
2019-03-01	870
2020-12-01	819
2020-10-01	818
2020-08-01	812
2020-09-01	809
2020-07-01	806
2020-11-01	805
2019-12-01	795
2019-04-01	794
2020-01-01	782
2019-11-01	781
2020-06-01	770
2020-05-01	766
2019-05-01	764
2019-09-01	762
2020-02-01	761
2019-07-01	757
2019-08-01	754
2019-10-01	739
2020-04-01	729
2019-06-01	726
2020-03-01	719

Name: count, dtype: int64

Value Counts of Driver\_ID

Driver_ID	
2110	24
2617	24
1623	24
1642	24

```
1644    24
       ..
1614     1
445      1
2397     1
1619     1
469      1
Name: count, Length: 2381, dtype: int64
```

#### Value Counts of Age

Age	count
36.0	1283
33.0	1250
34.0	1234
30.0	1146
32.0	1143
35.0	1138
31.0	1076
29.0	1013
37.0	862
38.0	854
39.0	788
28.0	772
27.0	744
40.0	701
41.0	661
26.0	566
42.0	478
25.0	449
44.0	407
43.0	399
45.0	371
46.0	350
24.0	274
47.0	224
23.0	193
48.0	144
49.0	99
22.0	92
52.0	78
51.0	72
50.0	69
NaN	61
21.0	35
53.0	26
54.0	24
55.0	21
58.0	7

```
Name: count, dtype: int64
```

```
Value Counts of Gender
```

```
Gender
```

```
0.0    11074  
1.0    7978  
NaN      52
```

```
Name: count, dtype: int64
```

```
Value Counts of City
```

```
City
```

```
C20     1008  
C29      900  
C26     869  
C22     809  
C27     786  
C15     761  
C10     744  
C12     727  
C8      712  
C16     709  
C28     683  
C1      677  
C6      660  
C5      656  
C14     648  
C3      637  
C24     614  
C7      609  
C21     603  
C25     584  
C19     579  
C4      578  
C13     569  
C18     544  
C23     538  
C9      520  
C2      472  
C11     468  
C17     440
```

```
Name: count, dtype: int64
```

```
Value Counts of Education_Level
```

```
Education_Level
```

```
1      6864  
2      6327  
0      5913
```

```
Name: count, dtype: int64
```

```
Value Counts of Income
```

```
Income
```

```
48747      57
109652     32
68356      30
42260      28
67490      28
...
44706      1
72186      1
67162      1
22132      1
35091      1
```

```
Name: count, Length: 2383, dtype: int64
```

```
Value Counts of Dateofjoining
```

```
Dateofjoining
```

```
2015-07-23    192
2020-07-31    150
2019-04-07    146
2016-04-25    134
2015-07-30    118
...
2018-03-16    1
2018-09-26    1
2020-12-27    1
2018-12-29    1
2018-12-16    1
```

```
Name: count, Length: 869, dtype: int64
```

```
Value Counts of LastWorkingDate
```

```
LastWorkingDate
```

```
NAT          17488
2020-07-29    70
2019-09-22    26
2019-03-17    14
2020-11-28    13
...
2019-03-09    1
2020-11-17    1
2020-06-02    1
2020-05-12    1
2020-10-28    1
```

```
Name: count, Length: 494, dtype: int64
```

```
Value Counts of Joining Designation
```

```
Joining Designation
1    9831
2    5955
3    2847
4    341
5    130
Name: count, dtype: int64
```

```
Value Counts of Grade
```

```
Grade
2    6627
1    5202
3    4826
4    2144
5    305
Name: count, dtype: int64
```

```
Value Counts of Total Business Value
```

```
Total Business Value
0        6499
200000    288
250000    148
500000    131
300000    107
...
130520      1
275330      1
820160      1
203040      1
448370      1
Name: count, Length: 10181, dtype: int64
```

```
Value Counts of Quarterly Rating
```

```
Quarterly Rating
1    7679
2    5553
3    3895
4    1977
Name: count, dtype: int64
```

## Observation

Value counts of Driver Id is having max 24. Due to data of 24 months readings of each driver.

Age from 30 to 36 having more number of drivers. Middle aged persons should be targeted.

Gender ratio is  $(11074/7978 = ) 1.388$  male for 1 female driver.

Education levels almost have same count

2015, July 23 has highest peak in Date of joining (Should check monthly peaks)

2020, July 29 has highest driver count leaving the ola (Should check monthly graph)

Joining designation has exponential decay from 1 to 5

Grade 2 drivers are more than other grade

Total Business value for so many rows (6499) is equal to zero. It is not a good sign.

That means most of the drivers unable provide the business consistently.

Quarterly rating also in decreasing manner from 1 to 4

## CHAPTER 2: PRELIMINARY DATA PREPROCESSING STEPS (NULL VALUE IMPUTATION, FEATURE ENGINEERING & AGGREGATION)

```
# Creating a duplicate dataset for Data Preprocessing
ola_1 = ola.copy(deep = True)

ola_1.isna().sum()
```

MMM-YY	0
Driver_ID	0
Age	61
Gender	52
City	0
Education_Level	0
Income	0
Dateofjoining	0
LastWorkingDate	17488
Joining Designation	0
Grade	0
Total Business Value	0
Quarterly Rating	0

dtype: int64

### Observation

As Driver Id is repeating in the dataset, we can create a dictionary named as 'id\_gender\_dict'. In that dictionary, keys are Driver\_id and values are Most Frequent Value of Gender of that Driver\_ID.

Gender null values can be imputed by using that id\_gender\_dict if that dictionary do not have any null values. Null values will be created if Gender feature is not logged at any reporting date for a driver\_id. Then We have to use KNN imputation for Gender by using Hamming Distance as Distance metric as Gender is categorical Feature.

Depending on Reporting date, Age Feature will change even for a driver. Either we can use KNN Imputer or ffill. As it is temporal data, ffill is better in this case.

## IMPUTATION OF NULL VALUES

### Gender Feature Null Value Imputation using id\_gender\_dict

Creating id\_gender\_dict Dictionary

```
# Define a function to get the most frequent non-null gender
def get_most_frequent_gender(group):
    return group.dropna().mode().iloc[0] if not group.dropna().empty
else None

# Group by Driver_id and apply the function
grouped = ola_1.groupby('Driver_ID')
['Gender'].apply(get_most_frequent_gender).reset_index()

# Create the dictionary
id_gender_dict = pd.Series(grouped.Gender.values,
index=grouped.Driver_ID).to_dict()

print(id_gender_dict)

{1: 0.0, 2: 0.0, 4: 0.0, 5: 0.0, 6: 1.0, 8: 0.0, 11: 1.0, 12: 0.0, 13:
0.0, 14: 1.0, 16: 1.0, 17: 0.0, 18: 1.0, 20: 1.0, 21: 1.0, 22: 0.0,
24: 0.0, 25: 0.0, 26: 0.0, 29: 0.0, 30: 0.0, 31: 1.0, 34: 1.0, 35:
0.0, 36: 1.0, 37: 1.0, 38: 0.0, 39: 0.0, 40: 0.0, 41: 0.0, 42: 1.0,
43: 1.0, 44: 0.0, 45: 0.0, 46: 1.0, 47: 0.0, 49: 0.0, 50: 1.0, 51:
0.0, 52: 0.0, 54: 0.0, 55: 1.0, 56: 1.0, 57: 1.0, 58: 0.0, 59: 1.0,
60: 1.0, 61: 0.0, 62: 0.0, 63: 0.0, 64: 0.0, 65: 1.0, 66: 1.0, 67:
0.0, 68: 0.0, 69: 1.0, 70: 0.0, 71: 0.0, 72: 0.0, 74: 1.0, 75: 1.0,
77: 0.0, 78: 0.0, 79: 0.0, 80: 1.0, 81: 0.0, 82: 1.0, 83: 0.0, 84:
0.0, 85: 0.0, 86: 0.0, 87: 0.0, 88: 1.0, 89: 0.0, 90: 0.0, 91: 0.0,
93: 0.0, 95: 1.0, 96: 0.0, 97: 1.0, 98: 0.0, 99: 0.0, 101: 0.0, 102:
1.0, 103: 1.0, 104: 1.0, 105: 0.0, 106: 1.0, 107: 0.0, 108: 0.0, 109:
1.0, 110: 1.0, 111: 0.0, 112: 0.0, 113: 1.0, 114: 1.0, 115: 0.0, 116:
0.0, 117: 1.0, 118: 1.0, 119: 1.0, 120: 0.0, 121: 1.0, 122: 0.0, 123:
0.0, 125: 1.0, 127: 0.0, 129: 1.0, 130: 0.0, 131: 1.0, 132: 0.0, 133:
1.0, 134: 0.0, 135: 0.0, 136: 1.0, 137: 0.0, 138: 0.0, 139: 0.0, 140:
0.0, 141: 0.0, 142: 0.0, 143: 0.0, 144: 0.0, 145: 0.0, 147: 0.0, 148:
0.0, 149: 1.0, 150: 0.0, 151: 0.0, 152: 0.0, 153: 1.0, 154: 0.0, 155:
0.0, 156: 1.0, 158: 0.0, 159: 1.0, 161: 0.0, 162: 0.0, 163: 0.0, 164:
1.0, 165: 1.0, 167: 0.0, 168: 0.0, 169: 0.0, 170: 0.0, 171: 1.0, 172:
1.0, 173: 0.0, 174: 1.0, 175: 0.0, 176: 0.0, 177: 0.0, 178: 1.0, 179:
1.0, 180: 0.0, 181: 1.0, 182: 0.0, 183: 1.0, 184: 1.0, 185: 0.0, 187:
```

0.0,	188:	0.0,	190:	1.0,	191:	1.0,	192:	1.0,	193:	0.0,	194:	0.0,	195:
0.0,	196:	0.0,	197:	0.0,	199:	0.0,	200:	0.0,	201:	1.0,	202:	0.0,	203:
1.0,	204:	1.0,	205:	0.0,	206:	1.0,	207:	0.0,	208:	0.0,	210:	0.0,	211:
1.0,	213:	0.0,	215:	1.0,	216:	0.0,	217:	0.0,	218:	0.0,	219:	0.0,	220:
1.0,	221:	0.0,	222:	0.0,	223:	0.0,	225:	1.0,	226:	1.0,	227:	1.0,	228:
0.0,	229:	0.0,	230:	0.0,	231:	0.0,	232:	0.0,	233:	1.0,	234:	1.0,	235:
1.0,	237:	0.0,	238:	0.0,	241:	0.0,	242:	0.0,	243:	0.0,	244:	0.0,	245:
0.0,	246:	0.0,	247:	1.0,	252:	0.0,	256:	0.0,	257:	0.0,	258:	0.0,	260:
0.0,	261:	1.0,	262:	1.0,	263:	0.0,	264:	0.0,	265:	0.0,	266:	0.0,	267:
0.0,	268:	1.0,	270:	0.0,	271:	0.0,	272:	0.0,	273:	0.0,	274:	0.0,	275:
0.0,	276:	1.0,	278:	0.0,	279:	1.0,	280:	1.0,	281:	1.0,	282:	1.0,	283:
1.0,	284:	0.0,	286:	1.0,	287:	1.0,	288:	1.0,	289:	0.0,	290:	1.0,	291:
1.0,	292:	1.0,	293:	1.0,	294:	0.0,	295:	1.0,	296:	1.0,	297:	0.0,	298:
0.0,	299:	1.0,	303:	0.0,	305:	0.0,	306:	1.0,	307:	0.0,	308:	0.0,	309:
1.0,	310:	1.0,	312:	0.0,	313:	1.0,	315:	1.0,	316:	0.0,	317:	0.0,	318:
1.0,	319:	0.0,	320:	1.0,	321:	1.0,	322:	1.0,	323:	1.0,	324:	0.0,	325:
0.0,	326:	0.0,	327:	0.0,	328:	1.0,	329:	1.0,	330:	1.0,	331:	0.0,	332:
0.0,	333:	1.0,	334:	1.0,	335:	1.0,	336:	0.0,	337:	1.0,	339:	0.0,	340:
0.0,	341:	0.0,	342:	0.0,	343:	0.0,	344:	0.0,	345:	1.0,	346:	1.0,	347:
1.0,	348:	1.0,	349:	0.0,	350:	0.0,	351:	0.0,	352:	0.0,	353:	1.0,	354:
1.0,	355:	0.0,	357:	0.0,	358:	1.0,	359:	0.0,	360:	1.0,	361:	0.0,	362:
1.0,	363:	0.0,	364:	0.0,	365:	1.0,	366:	0.0,	368:	0.0,	369:	0.0,	371:
1.0,	372:	1.0,	373:	1.0,	375:	0.0,	376:	0.0,	377:	0.0,	378:	0.0,	379:
0.0,	382:	0.0,	383:	1.0,	384:	1.0,	385:	1.0,	386:	1.0,	387:	1.0,	388:
0.0,	389:	1.0,	390:	1.0,	391:	0.0,	392:	1.0,	394:	1.0,	397:	0.0,	398:
0.0,	399:	0.0,	400:	0.0,	401:	1.0,	402:	0.0,	403:	0.0,	404:	0.0,	405:
0.0,	406:	0.0,	407:	1.0,	409:	0.0,	410:	0.0,	411:	1.0,	412:	0.0,	414:
1.0,	415:	1.0,	416:	1.0,	417:	1.0,	418:	0.0,	419:	0.0,	420:	0.0,	421:
0.0,	422:	0.0,	423:	0.0,	424:	0.0,	425:	0.0,	426:	0.0,	427:	0.0,	429:
0.0,	430:	0.0,	431:	1.0,	432:	0.0,	433:	0.0,	434:	1.0,	435:	0.0,	436:
0.0,	438:	1.0,	439:	0.0,	440:	0.0,	441:	0.0,	442:	0.0,	443:	1.0,	444:
0.0,	445:	0.0,	446:	1.0,	448:	0.0,	449:	0.0,	450:	0.0,	451:	1.0,	452:
0.0,	453:	0.0,	454:	1.0,	456:	1.0,	457:	1.0,	458:	1.0,	459:	0.0,	460:
1.0,	461:	1.0,	462:	0.0,	463:	0.0,	464:	0.0,	465:	1.0,	466:	1.0,	467:
0.0,	469:	0.0,	470:	0.0,	471:	0.0,	472:	1.0,	473:	1.0,	474:	0.0,	475:
0.0,	476:	1.0,	477:	0.0,	478:	1.0,	479:	1.0,	481:	0.0,	482:	0.0,	483:
1.0,	484:	1.0,	486:	1.0,	487:	0.0,	488:	0.0,	489:	1.0,	491:	0.0,	492:
1.0,	493:	0.0,	494:	1.0,	496:	0.0,	497:	1.0,	498:	0.0,	499:	0.0,	500:
0.0,	501:	1.0,	502:	0.0,	503:	1.0,	504:	1.0,	505:	0.0,	507:	0.0,	508:
0.0,	509:	0.0,	510:	1.0,	511:	0.0,	512:	1.0,	513:	1.0,	515:	1.0,	516:
1.0,	517:	1.0,	518:	1.0,	519:	0.0,	520:	1.0,	521:	0.0,	522:	0.0,	523:
0.0,	524:	0.0,	525:	0.0,	527:	0.0,	528:	0.0,	529:	1.0,	530:	0.0,	531:
0.0,	532:	0.0,	533:	0.0,	534:	0.0,	535:	0.0,	536:	0.0,	537:	1.0,	538:
1.0,	539:	0.0,	540:	0.0,	541:	0.0,	542:	0.0,	543:	0.0,	544:	0.0,	545:
1.0,	546:	1.0,	547:	0.0,	548:	1.0,	549:	1.0,	550:	0.0,	551:	0.0,	552:
0.0,	553:	0.0,	554:	0.0,	556:	1.0,	558:	1.0,	559:	0.0,	560:	0.0,	561:
0.0,	562:	1.0,	563:	1.0,	564:	0.0,	565:	0.0,	566:	0.0,	568:	0.0,	569:
1.0,	570:	1.0,	571:	1.0,	572:	1.0,	573:	0.0,	574:	0.0,	575:	0.0,	576:
1.0,	577:	1.0,	578:	0.0,	579:	1.0,	580:	1.0,	581:	0.0,	582:	1.0,	583:

0.0,	584:	1.0,	585:	1.0,	587:	1.0,	588:	1.0,	591:	0.0,	592:	1.0,	593:
0.0,	594:	0.0,	595:	0.0,	596:	0.0,	597:	1.0,	598:	1.0,	599:	1.0,	600:
0.0,	601:	1.0,	602:	0.0,	603:	0.0,	604:	1.0,	605:	0.0,	606:	1.0,	607:
1.0,	608:	0.0,	609:	1.0,	610:	1.0,	611:	1.0,	612:	0.0,	613:	0.0,	614:
1.0,	615:	0.0,	617:	1.0,	618:	0.0,	619:	1.0,	620:	1.0,	621:	1.0,	622:
0.0,	623:	1.0,	624:	0.0,	625:	0.0,	626:	0.0,	627:	0.0,	629:	1.0,	630:
1.0,	633:	1.0,	634:	0.0,	635:	0.0,	636:	0.0,	638:	0.0,	639:	0.0,	640:
1.0,	642:	0.0,	643:	0.0,	644:	0.0,	645:	1.0,	646:	0.0,	647:	1.0,	648:
1.0,	649:	0.0,	650:	1.0,	651:	1.0,	652:	0.0,	653:	1.0,	654:	0.0,	655:
0.0,	658:	1.0,	659:	1.0,	660:	1.0,	662:	0.0,	663:	0.0,	664:	0.0,	667:
0.0,	668:	1.0,	671:	0.0,	672:	0.0,	673:	1.0,	674:	1.0,	677:	1.0,	678:
0.0,	680:	1.0,	681:	0.0,	682:	0.0,	683:	1.0,	684:	1.0,	686:	1.0,	687:
0.0,	688:	1.0,	689:	1.0,	690:	1.0,	691:	1.0,	692:	0.0,	693:	0.0,	694:
1.0,	695:	0.0,	696:	1.0,	697:	0.0,	698:	0.0,	700:	1.0,	701:	1.0,	702:
1.0,	703:	0.0,	704:	0.0,	705:	0.0,	706:	0.0,	707:	1.0,	708:	1.0,	709:
0.0,	710:	0.0,	711:	0.0,	712:	0.0,	713:	0.0,	714:	0.0,	715:	0.0,	716:
0.0,	718:	0.0,	721:	0.0,	722:	0.0,	723:	0.0,	724:	1.0,	725:	0.0,	726:
1.0,	727:	0.0,	729:	0.0,	731:	1.0,	733:	1.0,	734:	1.0,	735:	1.0,	736:
1.0,	737:	0.0,	738:	0.0,	739:	1.0,	740:	0.0,	742:	0.0,	743:	0.0,	744:
0.0,	745:	0.0,	747:	1.0,	748:	1.0,	751:	0.0,	752:	0.0,	753:	0.0,	754:
0.0,	755:	0.0,	756:	1.0,	757:	0.0,	758:	0.0,	759:	0.0,	761:	0.0,	762:
0.0,	763:	0.0,	764:	1.0,	765:	1.0,	766:	1.0,	767:	1.0,	768:	1.0,	769:
0.0,	770:	0.0,	771:	0.0,	772:	0.0,	773:	0.0,	776:	1.0,	777:	1.0,	778:
0.0,	779:	1.0,	781:	1.0,	782:	1.0,	783:	0.0,	784:	0.0,	785:	0.0,	787:
0.0,	789:	1.0,	790:	0.0,	791:	1.0,	792:	1.0,	793:	1.0,	794:	0.0,	796:
0.0,	797:	0.0,	798:	0.0,	799:	1.0,	800:	0.0,	801:	0.0,	802:	0.0,	803:
0.0,	804:	0.0,	805:	0.0,	806:	1.0,	807:	1.0,	808:	0.0,	810:	0.0,	811:
0.0,	812:	0.0,	813:	1.0,	814:	0.0,	816:	0.0,	818:	1.0,	819:	0.0,	820:
0.0,	821:	1.0,	822:	1.0,	824:	1.0,	825:	1.0,	826:	0.0,	827:	0.0,	828:
0.0,	829:	1.0,	830:	0.0,	831:	0.0,	832:	1.0,	834:	0.0,	835:	0.0,	836:
0.0,	838:	1.0,	839:	0.0,	840:	1.0,	841:	1.0,	842:	0.0,	843:	1.0,	844:
0.0,	845:	0.0,	847:	0.0,	848:	1.0,	849:	1.0,	850:	0.0,	851:	0.0,	852:
0.0,	853:	1.0,	855:	0.0,	856:	1.0,	858:	0.0,	859:	0.0,	860:	0.0,	863:
1.0,	864:	1.0,	865:	1.0,	866:	0.0,	867:	1.0,	868:	1.0,	869:	0.0,	870:
1.0,	871:	1.0,	872:	0.0,	873:	1.0,	874:	0.0,	875:	1.0,	876:	1.0,	877:
0.0,	879:	0.0,	881:	0.0,	882:	0.0,	883:	0.0,	885:	1.0,	886:	1.0,	887:
0.0,	888:	0.0,	889:	0.0,	890:	1.0,	892:	1.0,	893:	1.0,	894:	0.0,	895:
0.0,	896:	1.0,	897:	0.0,	899:	0.0,	901:	1.0,	902:	0.0,	904:	1.0,	905:
0.0,	906:	1.0,	907:	1.0,	908:	0.0,	909:	0.0,	910:	0.0,	912:	0.0,	913:
0.0,	914:	1.0,	915:	0.0,	916:	0.0,	917:	0.0,	918:	1.0,	919:	1.0,	920:
0.0,	921:	1.0,	922:	1.0,	923:	1.0,	924:	0.0,	926:	0.0,	927:	0.0,	928:
1.0,	929:	0.0,	930:	1.0,	931:	1.0,	933:	1.0,	935:	1.0,	936:	0.0,	937:
0.0,	938:	1.0,	939:	1.0,	940:	1.0,	941:	1.0,	942:	0.0,	943:	0.0,	944:
1.0,	945:	1.0,	946:	0.0,	947:	1.0,	948:	1.0,	949:	1.0,	951:	0.0,	952:
0.0,	953:	0.0,	954:	0.0,	955:	0.0,	956:	0.0,	958:	0.0,	959:	1.0,	960:
1.0,	961:	1.0,	962:	0.0,	963:	0.0,	964:	0.0,	965:	0.0,	967:	0.0,	968:
0.0,	970:	0.0,	971:	0.0,	972:	0.0,	973:	0.0,	974:	1.0,	975:	1.0,	977:
1.0,	978:	0.0,	979:	1.0,	980:	0.0,	981:	1.0,	982:	0.0,	984:	0.0,	985:
0.0,	988:	0.0,	990:	1.0,	991:	0.0,	992:	0.0,	994:	0.0,	995:	0.0,	996:

0.0, 997: 0.0, 998: 0.0, 999: 1.0, 1000: 1.0, 1001: 1.0, 1002: 0.0,  
1003: 0.0, 1004: 0.0, 1005: 1.0, 1007: 0.0, 1008: 0.0, 1009: 1.0,  
1010: 1.0, 1011: 0.0, 1012: 0.0, 1013: 1.0, 1014: 0.0, 1015: 0.0,  
1016: 1.0, 1017: 1.0, 1019: 1.0, 1020: 0.0, 1021: 0.0, 1023: 0.0,  
1024: 1.0, 1026: 0.0, 1027: 1.0, 1028: 0.0, 1029: 0.0, 1030: 0.0,  
1031: 0.0, 1032: 0.0, 1033: 0.0, 1035: 1.0, 1036: 1.0, 1037: 1.0,  
1038: 0.0, 1039: 0.0, 1040: 0.0, 1041: 1.0, 1043: 0.0, 1046: 0.0,  
1048: 1.0, 1049: 0.0, 1050: 1.0, 1051: 1.0, 1054: 1.0, 1055: 0.0,  
1056: 0.0, 1057: 0.0, 1058: 1.0, 1059: 0.0, 1060: 0.0, 1061: 0.0,  
1062: 1.0, 1063: 0.0, 1064: 0.0, 1065: 0.0, 1066: 0.0, 1067: 0.0,  
1069: 1.0, 1070: 0.0, 1072: 0.0, 1073: 0.0, 1075: 0.0, 1076: 0.0,  
1077: 1.0, 1078: 0.0, 1079: 0.0, 1080: 0.0, 1081: 1.0, 1082: 1.0,  
1083: 1.0, 1084: 0.0, 1085: 1.0, 1086: 0.0, 1087: 0.0, 1089: 0.0,  
1090: 0.0, 1091: 1.0, 1092: 1.0, 1093: 0.0, 1094: 1.0, 1095: 1.0,  
1096: 0.0, 1097: 0.0, 1098: 0.0, 1099: 1.0, 1101: 0.0, 1102: 0.0,  
1103: 0.0, 1104: 1.0, 1105: 0.0, 1106: 0.0, 1107: 0.0, 1108: 0.0,  
1109: 1.0, 1110: 0.0, 1111: 0.0, 1112: 1.0, 1113: 0.0, 1114: 1.0,  
1115: 0.0, 1116: 0.0, 1117: 0.0, 1118: 0.0, 1119: 1.0, 1122: 0.0,  
1123: 1.0, 1125: 1.0, 1126: 0.0, 1129: 0.0, 1130: 0.0, 1131: 1.0,  
1133: 1.0, 1134: 0.0, 1136: 1.0, 1137: 0.0, 1138: 0.0, 1139: 1.0,  
1140: 0.0, 1141: 1.0, 1143: 0.0, 1144: 0.0, 1145: 0.0, 1146: 0.0,  
1147: 1.0, 1149: 1.0, 1150: 1.0, 1151: 0.0, 1152: 1.0, 1153: 1.0,  
1154: 0.0, 1155: 1.0, 1156: 0.0, 1157: 0.0, 1158: 1.0, 1159: 0.0,  
1160: 0.0, 1161: 0.0, 1162: 0.0, 1163: 1.0, 1164: 1.0, 1165: 1.0,  
1166: 0.0, 1167: 0.0, 1169: 0.0, 1172: 0.0, 1173: 0.0, 1175: 1.0,  
1177: 0.0, 1178: 0.0, 1179: 1.0, 1180: 1.0, 1181: 1.0, 1182: 0.0,  
1183: 0.0, 1185: 0.0, 1186: 1.0, 1187: 1.0, 1188: 1.0, 1189: 0.0,  
1190: 0.0, 1191: 0.0, 1192: 1.0, 1193: 0.0, 1194: 1.0, 1195: 1.0,  
1196: 0.0, 1197: 1.0, 1198: 0.0, 1199: 0.0, 1200: 0.0, 1201: 0.0,  
1202: 1.0, 1203: 0.0, 1204: 0.0, 1205: 0.0, 1206: 1.0, 1207: 0.0,  
1208: 0.0, 1209: 0.0, 1210: 0.0, 1211: 1.0, 1212: 0.0, 1213: 0.0,  
1214: 0.0, 1215: 0.0, 1216: 0.0, 1217: 0.0, 1218: 0.0, 1219: 1.0,  
1220: 1.0, 1222: 1.0, 1223: 0.0, 1224: 0.0, 1225: 1.0, 1226: 0.0,  
1227: 1.0, 1228: 0.0, 1229: 0.0, 1230: 1.0, 1231: 1.0, 1232: 0.0,  
1233: 1.0, 1234: 0.0, 1235: 0.0, 1236: 0.0, 1238: 0.0, 1239: 1.0,  
1242: 1.0, 1243: 1.0, 1244: 1.0, 1245: 0.0, 1246: 1.0, 1247: 0.0,  
1248: 1.0, 1249: 0.0, 1250: 1.0, 1251: 1.0, 1252: 1.0, 1255: 0.0,  
1258: 1.0, 1259: 0.0, 1260: 1.0, 1261: 0.0, 1262: 0.0, 1263: 0.0,  
1264: 1.0, 1265: 0.0, 1267: 0.0, 1268: 0.0, 1269: 1.0, 1271: 1.0,  
1272: 1.0, 1273: 1.0, 1274: 0.0, 1275: 1.0, 1276: 0.0, 1277: 0.0,  
1278: 0.0, 1279: 1.0, 1280: 1.0, 1281: 0.0, 1283: 1.0, 1284: 1.0,  
1285: 0.0, 1286: 0.0, 1287: 1.0, 1288: 0.0, 1289: 0.0, 1290: 1.0,  
1294: 0.0, 1295: 0.0, 1296: 0.0, 1297: 1.0, 1298: 1.0, 1300: 0.0,  
1301: 0.0, 1303: 1.0, 1304: 1.0, 1305: 1.0, 1306: 1.0, 1307: 0.0,  
1308: 0.0, 1309: 0.0, 1312: 1.0, 1314: 0.0, 1315: 0.0, 1316: 1.0,  
1318: 0.0, 1319: 1.0, 1320: 0.0, 1321: 0.0, 1322: 0.0, 1323: 1.0,  
1324: 0.0, 1325: 0.0, 1326: 1.0, 1327: 1.0, 1328: 1.0, 1329: 0.0,  
1330: 1.0, 1331: 1.0, 1333: 1.0, 1334: 0.0, 1335: 0.0, 1336: 0.0,  
1337: 1.0, 1338: 0.0, 1339: 0.0, 1340: 0.0, 1341: 0.0, 1342: 1.0,

1343:	1.0,	1345:	0.0,	1347:	0.0,	1348:	0.0,	1349:	0.0,	1350:	0.0,
1351:	0.0,	1352:	0.0,	1353:	0.0,	1354:	1.0,	1356:	1.0,	1357:	0.0,
1358:	0.0,	1359:	0.0,	1360:	0.0,	1361:	0.0,	1362:	0.0,	1363:	1.0,
1364:	1.0,	1365:	0.0,	1366:	0.0,	1369:	1.0,	1370:	1.0,	1371:	0.0,
1372:	1.0,	1373:	0.0,	1374:	0.0,	1375:	1.0,	1376:	0.0,	1378:	1.0,
1379:	0.0,	1380:	1.0,	1381:	1.0,	1382:	1.0,	1383:	0.0,	1384:	0.0,
1385:	1.0,	1386:	1.0,	1387:	0.0,	1388:	0.0,	1390:	0.0,	1392:	1.0,
1393:	0.0,	1394:	0.0,	1395:	1.0,	1396:	0.0,	1397:	1.0,	1398:	1.0,
1399:	1.0,	1400:	0.0,	1401:	1.0,	1402:	1.0,	1403:	0.0,	1405:	0.0,
1406:	0.0,	1407:	0.0,	1409:	1.0,	1410:	0.0,	1411:	0.0,	1412:	0.0,
1413:	0.0,	1414:	1.0,	1415:	0.0,	1416:	1.0,	1417:	0.0,	1418:	0.0,
1419:	0.0,	1421:	0.0,	1422:	1.0,	1424:	0.0,	1425:	0.0,	1426:	0.0,
1428:	0.0,	1429:	1.0,	1430:	1.0,	1431:	1.0,	1432:	0.0,	1433:	0.0,
1434:	0.0,	1435:	1.0,	1436:	0.0,	1437:	1.0,	1438:	0.0,	1439:	1.0,
1441:	0.0,	1442:	0.0,	1443:	1.0,	1444:	1.0,	1446:	1.0,	1447:	0.0,
1448:	1.0,	1449:	1.0,	1450:	1.0,	1451:	1.0,	1452:	0.0,	1453:	0.0,
1454:	0.0,	1455:	0.0,	1456:	0.0,	1457:	0.0,	1458:	1.0,	1459:	0.0,
1461:	0.0,	1462:	1.0,	1463:	0.0,	1464:	1.0,	1465:	0.0,	1466:	0.0,
1468:	0.0,	1469:	0.0,	1470:	0.0,	1471:	0.0,	1472:	1.0,	1473:	0.0,
1474:	0.0,	1475:	0.0,	1476:	0.0,	1477:	0.0,	1479:	0.0,	1480:	1.0,
1481:	1.0,	1482:	1.0,	1483:	1.0,	1484:	0.0,	1485:	0.0,	1486:	1.0,
1487:	0.0,	1489:	0.0,	1490:	1.0,	1491:	0.0,	1492:	0.0,	1493:	0.0,
1494:	1.0,	1495:	0.0,	1498:	1.0,	1500:	0.0,	1501:	0.0,	1502:	1.0,
1504:	0.0,	1505:	1.0,	1506:	1.0,	1507:	1.0,	1508:	0.0,	1509:	0.0,
1510:	1.0,	1512:	0.0,	1513:	1.0,	1514:	0.0,	1515:	0.0,	1516:	1.0,
1517:	1.0,	1518:	1.0,	1520:	0.0,	1521:	0.0,	1522:	1.0,	1523:	1.0,
1524:	1.0,	1525:	1.0,	1526:	0.0,	1527:	0.0,	1528:	0.0,	1529:	1.0,
1530:	0.0,	1531:	0.0,	1532:	0.0,	1533:	1.0,	1534:	1.0,	1536:	0.0,
1540:	0.0,	1541:	0.0,	1542:	1.0,	1543:	0.0,	1544:	0.0,	1547:	1.0,
1548:	0.0,	1549:	1.0,	1550:	0.0,	1553:	1.0,	1555:	0.0,	1556:	1.0,
1557:	0.0,	1558:	0.0,	1559:	1.0,	1560:	0.0,	1561:	0.0,	1562:	1.0,
1563:	1.0,	1565:	1.0,	1566:	1.0,	1567:	0.0,	1568:	0.0,	1569:	0.0,
1570:	1.0,	1571:	0.0,	1572:	0.0,	1573:	1.0,	1574:	0.0,	1575:	0.0,
1576:	0.0,	1577:	1.0,	1578:	0.0,	1579:	0.0,	1580:	0.0,	1581:	0.0,
1582:	0.0,	1583:	1.0,	1585:	0.0,	1586:	1.0,	1587:	1.0,	1588:	0.0,
1589:	0.0,	1590:	1.0,	1591:	0.0,	1592:	1.0,	1593:	0.0,	1594:	0.0,
1595:	0.0,	1597:	1.0,	1598:	0.0,	1599:	1.0,	1600:	0.0,	1601:	0.0,
1602:	0.0,	1603:	1.0,	1605:	0.0,	1606:	1.0,	1607:	0.0,	1609:	1.0,
1610:	0.0,	1611:	0.0,	1612:	0.0,	1613:	0.0,	1614:	0.0,	1616:	0.0,
1617:	0.0,	1618:	0.0,	1619:	1.0,	1621:	0.0,	1622:	0.0,	1623:	0.0,
1624:	1.0,	1625:	0.0,	1626:	0.0,	1627:	0.0,	1628:	0.0,	1629:	0.0,
1630:	1.0,	1631:	1.0,	1632:	1.0,	1633:	1.0,	1634:	1.0,	1635:	0.0,
1636:	0.0,	1637:	1.0,	1638:	1.0,	1639:	0.0,	1640:	0.0,	1642:	0.0,
1643:	0.0,	1644:	1.0,	1645:	0.0,	1646:	1.0,	1647:	0.0,	1650:	1.0,
1651:	0.0,	1652:	0.0,	1653:	1.0,	1655:	1.0,	1657:	0.0,	1658:	1.0,
1659:	1.0,	1661:	0.0,	1662:	1.0,	1663:	0.0,	1664:	1.0,	1665:	0.0,
1666:	0.0,	1667:	0.0,	1668:	1.0,	1669:	0.0,	1670:	1.0,	1671:	1.0,
1672:	1.0,	1673:	1.0,	1674:	1.0,	1675:	1.0,	1676:	1.0,	1677:	1.0,
1678:	0.0,	1679:	0.0,	1680:	1.0,	1681:	0.0,	1682:	0.0,	1683:	1.0,

1685: 1.0,	1686: 0.0,	1687: 0.0,	1688: 0.0,	1689: 1.0,	1690: 0.0,
1692: 0.0,	1693: 1.0,	1694: 0.0,	1695: 0.0,	1696: 1.0,	1697: 1.0,
1699: 1.0,	1701: 1.0,	1702: 0.0,	1703: 1.0,	1704: 1.0,	1706: 1.0,
1707: 1.0,	1708: 0.0,	1709: 0.0,	1710: 1.0,	1712: 0.0,	1714: 0.0,
1715: 0.0,	1716: 1.0,	1717: 0.0,	1718: 1.0,	1719: 1.0,	1720: 0.0,
1721: 1.0,	1722: 0.0,	1723: 0.0,	1724: 0.0,	1725: 0.0,	1726: 0.0,
1727: 0.0,	1728: 0.0,	1729: 1.0,	1730: 0.0,	1731: 1.0,	1732: 1.0,
1733: 0.0,	1734: 0.0,	1735: 0.0,	1736: 0.0,	1737: 0.0,	1738: 0.0,
1739: 0.0,	1740: 1.0,	1741: 0.0,	1742: 1.0,	1743: 0.0,	1744: 0.0,
1745: 0.0,	1746: 1.0,	1749: 1.0,	1751: 0.0,	1752: 1.0,	1753: 0.0,
1754: 1.0,	1755: 0.0,	1756: 1.0,	1757: 1.0,	1758: 0.0,	1759: 0.0,
1760: 0.0,	1761: 0.0,	1763: 1.0,	1764: 0.0,	1766: 0.0,	1769: 0.0,
1770: 1.0,	1771: 1.0,	1773: 1.0,	1774: 0.0,	1775: 1.0,	1776: 0.0,
1778: 0.0,	1779: 0.0,	1781: 0.0,	1782: 1.0,	1783: 0.0,	1784: 1.0,
1785: 0.0,	1786: 0.0,	1788: 1.0,	1789: 0.0,	1791: 0.0,	1792: 0.0,
1793: 1.0,	1795: 0.0,	1796: 0.0,	1797: 0.0,	1800: 0.0,	1801: 1.0,
1802: 0.0,	1803: 0.0,	1804: 0.0,	1805: 0.0,	1807: 0.0,	1808: 0.0,
1809: 0.0,	1810: 0.0,	1811: 0.0,	1812: 1.0,	1813: 0.0,	1814: 1.0,
1815: 0.0,	1816: 1.0,	1817: 1.0,	1818: 0.0,	1819: 0.0,	1820: 1.0,
1823: 0.0,	1824: 0.0,	1825: 0.0,	1826: 0.0,	1828: 0.0,	1829: 1.0,
1830: 0.0,	1831: 0.0,	1832: 0.0,	1834: 0.0,	1835: 0.0,	1836: 1.0,
1837: 0.0,	1838: 1.0,	1839: 0.0,	1840: 1.0,	1841: 0.0,	1842: 1.0,
1843: 0.0,	1844: 1.0,	1845: 0.0,	1846: 1.0,	1847: 1.0,	1848: 0.0,
1849: 0.0,	1850: 1.0,	1851: 1.0,	1852: 0.0,	1853: 0.0,	1854: 0.0,
1855: 0.0,	1856: 1.0,	1858: 1.0,	1859: 0.0,	1860: 0.0,	1861: 0.0,
1862: 1.0,	1863: 0.0,	1864: 1.0,	1865: 0.0,	1866: 0.0,	1867: 1.0,
1868: 1.0,	1869: 1.0,	1871: 0.0,	1872: 1.0,	1873: 1.0,	1874: 0.0,
1875: 1.0,	1876: 0.0,	1877: 1.0,	1878: 1.0,	1879: 0.0,	1880: 1.0,
1882: 0.0,	1883: 1.0,	1884: 0.0,	1886: 0.0,	1887: 0.0,	1888: 1.0,
1889: 1.0,	1890: 1.0,	1891: 1.0,	1892: 0.0,	1893: 0.0,	1894: 1.0,
1895: 0.0,	1896: 1.0,	1897: 0.0,	1898: 0.0,	1899: 0.0,	1900: 0.0,
1901: 0.0,	1902: 1.0,	1904: 0.0,	1905: 0.0,	1906: 0.0,	1908: 1.0,
1909: 1.0,	1911: 1.0,	1912: 1.0,	1913: 0.0,	1914: 0.0,	1916: 1.0,
1917: 0.0,	1918: 1.0,	1919: 0.0,	1922: 0.0,	1924: 1.0,	1925: 0.0,
1926: 0.0,	1927: 0.0,	1928: 0.0,	1929: 0.0,	1930: 0.0,	1932: 1.0,
1933: 1.0,	1934: 0.0,	1935: 0.0,	1936: 0.0,	1937: 0.0,	1938: 0.0,
1939: 1.0,	1941: 1.0,	1942: 1.0,	1944: 1.0,	1947: 1.0,	1948: 0.0,
1949: 1.0,	1950: 1.0,	1951: 0.0,	1952: 0.0,	1954: 0.0,	1955: 0.0,
1956: 0.0,	1957: 1.0,	1958: 1.0,	1959: 0.0,	1960: 1.0,	1962: 0.0,
1963: 0.0,	1965: 0.0,	1967: 0.0,	1968: 0.0,	1970: 0.0,	1971: 0.0,
1972: 1.0,	1973: 0.0,	1975: 1.0,	1976: 1.0,	1977: 1.0,	1978: 0.0,
1979: 0.0,	1980: 0.0,	1981: 1.0,	1982: 1.0,	1983: 1.0,	1984: 0.0,
1987: 0.0,	1988: 0.0,	1989: 0.0,	1990: 0.0,	1991: 0.0,	1992: 1.0,
1993: 0.0,	1994: 0.0,	1995: 0.0,	1996: 0.0,	1997: 0.0,	1998: 0.0,
1999: 1.0,	2000: 0.0,	2001: 1.0,	2002: 0.0,	2003: 1.0,	2005: 0.0,
2007: 1.0,	2008: 1.0,	2009: 1.0,	2010: 0.0,	2012: 0.0,	2013: 1.0,
2014: 1.0,	2015: 1.0,	2016: 1.0,	2017: 0.0,	2018: 0.0,	2019: 1.0,
2020: 0.0,	2021: 0.0,	2022: 0.0,	2023: 1.0,	2024: 0.0,	2025: 0.0,
2026: 1.0,	2027: 1.0,	2028: 1.0,	2030: 1.0,	2031: 0.0,	2032: 0.0,

2033: 0.0,	2034: 1.0,	2035: 0.0,	2036: 1.0,	2037: 0.0,	2038: 1.0,
2039: 1.0,	2040: 0.0,	2041: 0.0,	2043: 0.0,	2044: 0.0,	2045: 0.0,
2047: 0.0,	2049: 1.0,	2050: 0.0,	2052: 0.0,	2053: 1.0,	2054: 0.0,
2055: 0.0,	2056: 0.0,	2057: 1.0,	2058: 1.0,	2059: 1.0,	2060: 0.0,
2061: 1.0,	2062: 0.0,	2063: 0.0,	2064: 0.0,	2065: 1.0,	2066: 1.0,
2067: 1.0,	2068: 0.0,	2069: 1.0,	2070: 0.0,	2071: 1.0,	2073: 0.0,
2074: 0.0,	2076: 1.0,	2078: 1.0,	2082: 0.0,	2083: 1.0,	2084: 1.0,
2085: 1.0,	2086: 0.0,	2087: 0.0,	2088: 1.0,	2089: 0.0,	2090: 0.0,
2091: 0.0,	2092: 0.0,	2093: 0.0,	2094: 0.0,	2095: 0.0,	2096: 0.0,
2097: 0.0,	2098: 0.0,	2100: 0.0,	2101: 0.0,	2102: 0.0,	2103: 0.0,
2104: 1.0,	2105: 0.0,	2106: 1.0,	2108: 1.0,	2109: 0.0,	2110: 0.0,
2113: 1.0,	2114: 1.0,	2115: 0.0,	2116: 1.0,	2117: 0.0,	2118: 0.0,
2120: 1.0,	2121: 1.0,	2122: 0.0,	2123: 1.0,	2124: 0.0,	2125: 1.0,
2126: 1.0,	2127: 1.0,	2128: 0.0,	2129: 1.0,	2130: 0.0,	2131: 0.0,
2132: 1.0,	2133: 0.0,	2134: 0.0,	2135: 1.0,	2137: 1.0,	2138: 0.0,
2139: 0.0,	2140: 0.0,	2141: 0.0,	2142: 1.0,	2143: 0.0,	2144: 1.0,
2145: 0.0,	2146: 0.0,	2148: 1.0,	2151: 0.0,	2152: 1.0,	2153: 1.0,
2154: 0.0,	2155: 0.0,	2156: 1.0,	2157: 1.0,	2159: 1.0,	2160: 1.0,
2161: 0.0,	2162: 0.0,	2163: 0.0,	2164: 0.0,	2165: 1.0,	2166: 0.0,
2167: 0.0,	2168: 0.0,	2169: 1.0,	2170: 0.0,	2171: 0.0,	2172: 0.0,
2173: 0.0,	2174: 0.0,	2175: 0.0,	2176: 0.0,	2178: 1.0,	2179: 1.0,
2180: 0.0,	2181: 1.0,	2182: 1.0,	2183: 1.0,	2186: 1.0,	2187: 0.0,
2188: 1.0,	2189: 1.0,	2190: 0.0,	2191: 0.0,	2192: 0.0,	2193: 0.0,
2194: 1.0,	2195: 0.0,	2196: 0.0,	2197: 1.0,	2198: 1.0,	2199: 0.0,
2200: 0.0,	2201: 1.0,	2202: 0.0,	2203: 0.0,	2204: 1.0,	2205: 1.0,
2206: 0.0,	2207: 0.0,	2208: 0.0,	2209: 1.0,	2210: 0.0,	2211: 0.0,
2213: 1.0,	2214: 0.0,	2215: 0.0,	2216: 0.0,	2217: 1.0,	2218: 1.0,
2219: 0.0,	2220: 0.0,	2221: 0.0,	2223: 0.0,	2224: 1.0,	2225: 1.0,
2226: 0.0,	2227: 1.0,	2228: 1.0,	2229: 0.0,	2230: 0.0,	2231: 1.0,
2232: 1.0,	2233: 0.0,	2235: 1.0,	2236: 1.0,	2238: 0.0,	2239: 1.0,
2240: 0.0,	2241: 0.0,	2242: 0.0,	2245: 1.0,	2246: 1.0,	2247: 1.0,
2248: 1.0,	2249: 1.0,	2250: 0.0,	2251: 0.0,	2252: 1.0,	2253: 0.0,
2254: 1.0,	2255: 0.0,	2256: 0.0,	2257: 1.0,	2258: 1.0,	2259: 0.0,
2260: 0.0,	2261: 1.0,	2263: 0.0,	2264: 1.0,	2265: 1.0,	2267: 1.0,
2268: 0.0,	2269: 0.0,	2270: 1.0,	2271: 0.0,	2272: 1.0,	2273: 0.0,
2274: 1.0,	2276: 0.0,	2277: 1.0,	2278: 1.0,	2279: 1.0,	2281: 0.0,
2282: 1.0,	2283: 0.0,	2284: 0.0,	2285: 1.0,	2286: 1.0,	2287: 1.0,
2288: 0.0,	2289: 1.0,	2290: 1.0,	2291: 1.0,	2292: 0.0,	2294: 1.0,
2295: 0.0,	2296: 0.0,	2298: 0.0,	2299: 1.0,	2300: 1.0,	2302: 0.0,
2303: 0.0,	2304: 0.0,	2306: 1.0,	2307: 1.0,	2308: 0.0,	2309: 1.0,
2310: 0.0,	2311: 0.0,	2313: 1.0,	2314: 0.0,	2315: 1.0,	2318: 1.0,
2319: 0.0,	2320: 0.0,	2321: 0.0,	2322: 1.0,	2323: 1.0,	2324: 0.0,
2326: 1.0,	2327: 1.0,	2329: 1.0,	2330: 1.0,	2331: 1.0,	2332: 1.0,
2333: 0.0,	2334: 1.0,	2335: 0.0,	2336: 0.0,	2338: 0.0,	2339: 0.0,
2340: 0.0,	2342: 1.0,	2343: 1.0,	2344: 0.0,	2345: 0.0,	2346: 0.0,
2347: 0.0,	2348: 1.0,	2350: 0.0,	2351: 0.0,	2354: 0.0,	2355: 1.0,
2356: 0.0,	2357: 1.0,	2358: 0.0,	2360: 1.0,	2361: 1.0,	2362: 1.0,
2363: 1.0,	2364: 0.0,	2365: 1.0,	2366: 1.0,	2367: 1.0,	2368: 0.0,
2369: 0.0,	2370: 0.0,	2371: 1.0,	2372: 0.0,	2373: 1.0,	2375: 0.0,

2377:	1.0,	2378:	1.0,	2379:	0.0,	2380:	1.0,	2381:	0.0,	2382:	1.0,
2384:	1.0,	2385:	1.0,	2386:	0.0,	2387:	0.0,	2388:	1.0,	2390:	0.0,
2391:	0.0,	2392:	1.0,	2393:	1.0,	2394:	0.0,	2395:	0.0,	2396:	0.0,
2397:	1.0,	2398:	0.0,	2399:	1.0,	2400:	0.0,	2402:	0.0,	2403:	0.0,
2404:	0.0,	2405:	0.0,	2406:	1.0,	2407:	1.0,	2408:	0.0,	2409:	0.0,
2410:	0.0,	2411:	1.0,	2412:	1.0,	2413:	1.0,	2414:	0.0,	2415:	0.0,
2416:	0.0,	2417:	0.0,	2418:	1.0,	2419:	1.0,	2420:	0.0,	2421:	0.0,
2422:	0.0,	2423:	1.0,	2424:	0.0,	2425:	0.0,	2426:	1.0,	2427:	1.0,
2428:	0.0,	2429:	0.0,	2430:	0.0,	2431:	1.0,	2432:	1.0,	2433:	0.0,
2435:	1.0,	2436:	0.0,	2438:	1.0,	2439:	0.0,	2440:	0.0,	2441:	1.0,
2442:	0.0,	2443:	0.0,	2445:	0.0,	2446:	1.0,	2447:	0.0,	2448:	0.0,
2449:	1.0,	2450:	1.0,	2451:	0.0,	2453:	1.0,	2454:	0.0,	2455:	0.0,
2457:	0.0,	2458:	0.0,	2459:	1.0,	2460:	1.0,	2461:	0.0,	2464:	0.0,
2465:	0.0,	2467:	1.0,	2468:	0.0,	2469:	1.0,	2470:	0.0,	2471:	0.0,
2472:	0.0,	2473:	1.0,	2474:	0.0,	2475:	0.0,	2476:	0.0,	2478:	1.0,
2480:	1.0,	2481:	1.0,	2482:	0.0,	2483:	0.0,	2484:	0.0,	2485:	0.0,
2486:	1.0,	2487:	1.0,	2488:	0.0,	2489:	1.0,	2490:	1.0,	2491:	0.0,
2492:	1.0,	2493:	1.0,	2494:	0.0,	2495:	1.0,	2496:	0.0,	2497:	1.0,
2498:	0.0,	2499:	0.0,	2500:	0.0,	2501:	1.0,	2502:	1.0,	2503:	1.0,
2504:	1.0,	2505:	0.0,	2506:	0.0,	2507:	0.0,	2508:	0.0,	2509:	1.0,
2510:	0.0,	2511:	1.0,	2513:	0.0,	2514:	0.0,	2515:	0.0,	2516:	0.0,
2517:	1.0,	2518:	1.0,	2520:	0.0,	2521:	0.0,	2522:	1.0,	2525:	1.0,
2527:	0.0,	2528:	1.0,	2529:	0.0,	2530:	1.0,	2532:	0.0,	2533:	0.0,
2535:	0.0,	2536:	1.0,	2537:	1.0,	2538:	0.0,	2539:	1.0,	2540:	0.0,
2541:	1.0,	2543:	0.0,	2544:	0.0,	2545:	0.0,	2546:	0.0,	2547:	0.0,
2549:	1.0,	2550:	0.0,	2551:	0.0,	2552:	1.0,	2553:	1.0,	2554:	0.0,
2555:	1.0,	2556:	0.0,	2558:	0.0,	2559:	0.0,	2560:	0.0,	2561:	1.0,
2562:	0.0,	2563:	0.0,	2565:	0.0,	2566:	0.0,	2567:	1.0,	2568:	0.0,
2569:	1.0,	2570:	0.0,	2571:	0.0,	2572:	1.0,	2573:	1.0,	2574:	0.0,
2575:	0.0,	2576:	0.0,	2577:	0.0,	2578:	1.0,	2579:	0.0,	2581:	1.0,
2582:	0.0,	2585:	0.0,	2586:	1.0,	2587:	0.0,	2588:	1.0,	2589:	1.0,
2590:	0.0,	2592:	0.0,	2593:	0.0,	2594:	0.0,	2595:	1.0,	2596:	0.0,
2597:	1.0,	2598:	1.0,	2599:	0.0,	2600:	1.0,	2601:	0.0,	2602:	0.0,
2603:	1.0,	2604:	1.0,	2605:	0.0,	2606:	0.0,	2607:	0.0,	2608:	0.0,
2609:	1.0,	2610:	1.0,	2611:	0.0,	2612:	0.0,	2613:	1.0,	2614:	0.0,
2615:	0.0,	2616:	0.0,	2617:	1.0,	2618:	1.0,	2620:	1.0,	2623:	1.0,
2624:	1.0,	2625:	0.0,	2626:	0.0,	2627:	1.0,	2628:	0.0,	2629:	1.0,
2630:	0.0,	2631:	0.0,	2632:	0.0,	2633:	1.0,	2634:	0.0,	2635:	0.0,
2637:	0.0,	2639:	1.0,	2640:	1.0,	2641:	1.0,	2642:	0.0,	2643:	0.0,
2644:	0.0,	2645:	1.0,	2646:	0.0,	2647:	0.0,	2648:	0.0,	2649:	0.0,
2650:	0.0,	2651:	0.0,	2653:	1.0,	2654:	0.0,	2655:	0.0,	2656:	1.0,
2659:	1.0,	2660:	0.0,	2661:	1.0,	2663:	0.0,	2664:	1.0,	2665:	1.0,
2666:	0.0,	2667:	0.0,	2668:	0.0,	2670:	1.0,	2671:	1.0,	2672:	1.0,
2674:	1.0,	2675:	0.0,	2676:	0.0,	2677:	0.0,	2678:	1.0,	2680:	0.0,
2682:	0.0,	2683:	1.0,	2684:	0.0,	2685:	1.0,	2686:	0.0,	2687:	0.0,
2690:	0.0,	2691:	1.0,	2692:	0.0,	2693:	0.0,	2694:	1.0,	2695:	0.0,
2696:	1.0,	2697:	0.0,	2698:	0.0,	2699:	0.0,	2700:	0.0,	2701:	0.0,
2702:	0.0,	2703:	0.0,	2704:	0.0,	2705:	0.0,	2706:	0.0,	2707:	0.0,
2708:	1.0,	2709:	1.0,	2710:	0.0,	2711:	1.0,	2712:	1.0,	2713:	1.0,

```

2714: 0.0, 2715: 1.0, 2716: 1.0, 2718: 1.0, 2719: 0.0, 2720: 1.0,
2721: 0.0, 2722: 0.0, 2723: 1.0, 2724: 1.0, 2725: 0.0, 2726: 0.0,
2727: 1.0, 2728: 1.0, 2729: 0.0, 2730: 1.0, 2731: 1.0, 2732: 1.0,
2733: 0.0, 2734: 1.0, 2735: 0.0, 2736: 1.0, 2737: 1.0, 2738: 0.0,
2739: 0.0, 2740: 0.0, 2742: 0.0, 2744: 1.0, 2745: 0.0, 2746: 1.0,
2747: 1.0, 2749: 0.0, 2751: 0.0, 2752: 0.0, 2753: 1.0, 2754: 0.0,
2755: 0.0, 2757: 0.0, 2758: 0.0, 2759: 0.0, 2760: 1.0, 2761: 0.0,
2762: 0.0, 2763: 1.0, 2764: 1.0, 2765: 0.0, 2766: 1.0, 2768: 1.0,
2770: 0.0, 2771: 0.0, 2772: 1.0, 2773: 0.0, 2774: 0.0, 2775: 0.0,
2776: 1.0, 2778: 0.0, 2779: 0.0, 2781: 0.0, 2782: 0.0, 2784: 0.0,
2785: 1.0, 2786: 0.0, 2787: 1.0, 2788: 0.0}

```

Check whether any driver has null value in the `id_gender_dict` dictionary

```

# Check if any values in the dictionary are None
has_none_values = any(value is None for value in
id_gender_dict.values())

print(f"Does id_gender_dict have None values? {has_none_values}")

Does id_gender_dict have None values? False

```

### Observation

As there are no null values in the dictionary, we can impute all the gender null values without any problem.

```

# Function to impute Gender based on Driver_id using id_gender_dict
def impute_gender(row):
    if pd.isnull(row['Gender']):
        return id_gender_dict.get(row['Driver_ID'])
    return row['Gender']

# Apply the function to each row
ola_1['Gender'] = ola_1.apply(impute_gender, axis=1)

ola_1.isna().sum()

      MMM-YY          0
      Driver_ID        0
      Age            61
      Gender          0
      City            0
      Education_Level     0
      Income           0
      Dateofjoining       0
      LastWorkingDate      17488
      Joining Designation     0
      Grade            0
      Total Business Value     0

```

```
Quarterly Rating      0
dtype: int64
```

## Age Feature Null Value Imputation using ffill() and bfill()

```
# Sort by 'Driver_ID' and 'Reporting Date' to ensure the most recent
# date comes first
ola_1 = ola_1.sort_values(by=['Driver_ID', 'MMM-YY'],
                           ascending=[True, False])

# Forward fill the 'Age' values within each 'Driver_ID' group
ola_1['Age'] = ola_1.groupby('Driver_ID')['Age'].ffill()

ola_1.isna().sum()

MMM-YY                  0
Driver_ID                0
Age                      7
Gender                    0
City                      0
Education_Level           0
Income                     0
Dateofjoining             0
LastWorkingDate          17488
Joining_Designation       0
Grade                      0
Total_Business_Value      0
Quarterly_Rating           0
dtype: int64
```

### Observation

As 7 null values are remained after ffill method, Use bfill method to impute those nan values.

ffill or bfill is better because we are imputing the null values to nearest possible age based on reporting date "MMM-YY" for respective driver.

```
# Backward fill the 'Age' values within each 'Driver_ID' group for
# remaining 7 null values
ola_1['Age'] = ola_1.groupby('Driver_ID')['Age'].bfill()

ola_1.isna().sum()

MMM-YY                  0
Driver_ID                0
Age                      0
Gender                    0
City                      0
Education_Level           0
Income                     0
```

```

Dateofjoining          0
LastWorkingDate        17488
Joining Designation    0
Grade                  0
Total Business Value   0
Quarterly Rating       0
dtype: int64

```

## FEATURE ENGINEERING AND AGGREGATION BY DRIVER\_ID

```
ola_1.head()
```

	MMM-YY	Driver_ID	Age	Gender	City	Education_Level	Income	\
2	2019-03-01	1	28.0	0.0	C23	2	57387	
1	2019-02-01	1	28.0	0.0	C23	2	57387	
0	2019-01-01	1	28.0	0.0	C23	2	57387	
4	2020-12-01	2	31.0	0.0	C7	2	67016	
3	2020-11-01	2	31.0	0.0	C7	2	67016	

	Dateofjoining	LastWorkingDate	Joining Designation	Grade	\
2	2018-12-24	2019-11-03	1	1	
1	2018-12-24	NaT	1	1	
0	2018-12-24	NaT	1	1	
4	2020-06-11	NaT	2	2	
3	2020-06-11	NaT	2	2	

	Total Business Value	Quarterly Rating
2	0	2
1	-665480	2
0	2381060	2
4	0	1
3	0	1

Convert Age and City to integers

```

ola_1['Age'] = ola_1['Age'].astype('int64')
ola_1['City'] = ola_1['City'].astype('str').str.extractall('(\d+)')
.unstack().fillna('').sum(axis=1).astype(int)

<>:2: SyntaxWarning: invalid escape sequence '\d'
<>:2: SyntaxWarning: invalid escape sequence '\d'
C:\Users\saina\AppData\Local\Temp\ipykernel_11784\4116700500.py:2:
SyntaxWarning: invalid escape sequence '\d'
    ola_1['City'] = ola_1['City'].astype('str').str.extractall('(\d+)')
.unstack().fillna('').sum(axis=1).astype(int)

ola_1.head()

```

	MMM-YY	Driver_ID	Age	Gender	City	Education_Level	Income	\
2	2019-03-01	1	28	0.0	23	2	57387	

1	2019-02-01	1	28	0.0	23	2	57387
0	2019-01-01	1	28	0.0	23	2	57387
4	2020-12-01	2	31	0.0	7	2	67016
3	2020-11-01	2	31	0.0	7	2	67016
2	Dateofjoining	LastWorkingDate	Joining	Designation	Grade	\	
2	2018-12-24	2019-11-03			1	1	
1	2018-12-24		NaT		1	1	
0	2018-12-24		NaT		1	1	
4	2020-06-11		NaT		2	2	
3	2020-06-11		NaT		2	2	
2	Total	Business	Value	Quarterly	Rating		
2			0		2		
1			-665480		2		
0			2381060		2		
4			0		1		
3			0		1		

Sort the rows properly for Feature Engineering

```
ola_1 = ola_1.sort_values(by=["Driver_ID", "MMM-YY"], ascending=[True, True])
```

```
ola_1.head()
```

0	2019-01-01	1	28	0.0	23	2	57387
1	2019-02-01	1	28	0.0	23	2	57387
2	2019-03-01	1	28	0.0	23	2	57387
3	2020-11-01	2	31	0.0	7	2	67016
4	2020-12-01	2	31	0.0	7	2	67016
0	Dateofjoining	LastWorkingDate	Joining	Designation	Grade	\	
0	2018-12-24		NaT		1	1	
1	2018-12-24		NaT		1	1	
2	2018-12-24	2019-11-03			1	1	
3	2020-06-11		NaT		2	2	
4	2020-06-11		NaT		2	2	
0	Total	Business	Value	Quarterly	Rating		
0			2381060		2		
1			-665480		2		
2			0		2		
3			0		1		
4			0		1		

## Creating Target Feature from LastWorkingDate Feature

```
ola_2 = ola_1.groupby("Driver_ID").agg({"LastWorkingDate":"last"})
["LastWorkingDate"].isna().reset_index()

ola_2.head()

   Driver_ID  LastWorkingDate
0          1            False
1          2             True
2          4            False
3          5            False
4          6             True

ola_2["LastWorkingDate"].value_counts()

LastWorkingDate
False    1616
True     765
Name: count, dtype: int64

ola_2["LastWorkingDate"].replace({True : 0,False: 1},inplace = True)

C:\Users\saina\AppData\Local\Temp\ipykernel_11784\3565191444.py:1:
FutureWarning: A value is trying to be set on a copy of a DataFrame or
Series through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never
work because the intermediate object on which we are setting values
always behaves as a copy.

For example, when doing 'df[col].method(value, inplace=True)', try
using 'df.method({col: value}, inplace=True)' or df[col] =
df[col].method(value) instead, to perform the operation inplace on the
original object.
```

```
ola_2["LastWorkingDate"].replace({True : 0,False: 1},inplace = True)
C:\Users\saina\AppData\Local\Temp\ipykernel_11784\3565191444.py:1:
FutureWarning: Downcasting behavior in `replace` is deprecated and
will be removed in a future version. To retain the old behavior,
explicitly call `result.infer_objects(copy=False)`. To opt-in to the
future behavior, set `pd.set_option('future.no_silent_downcasting',
True)`
ola_2["LastWorkingDate"].replace({True : 0,False: 1},inplace = True)
```

### Observation

0 means "Driver has not churned"

1 means "Driver has churned"

```
ola_2.head()
```

```

Driver_ID  LastWorkingDate
0           1                  1
1           2                  0
2           4                  1
3           5                  1
4           6                  0

ola_2.rename(columns={"LastWorkingDate": "Target"}, inplace= True)
ola_2.head()

Driver_ID  Target
0           1      1
1           2      0
2           4      1
3           5      1
4           6      0

```

Line Plots to check whether the Features are monotonically increasing or decreasing or constant or fluctuating with respect to "MMM-YY"

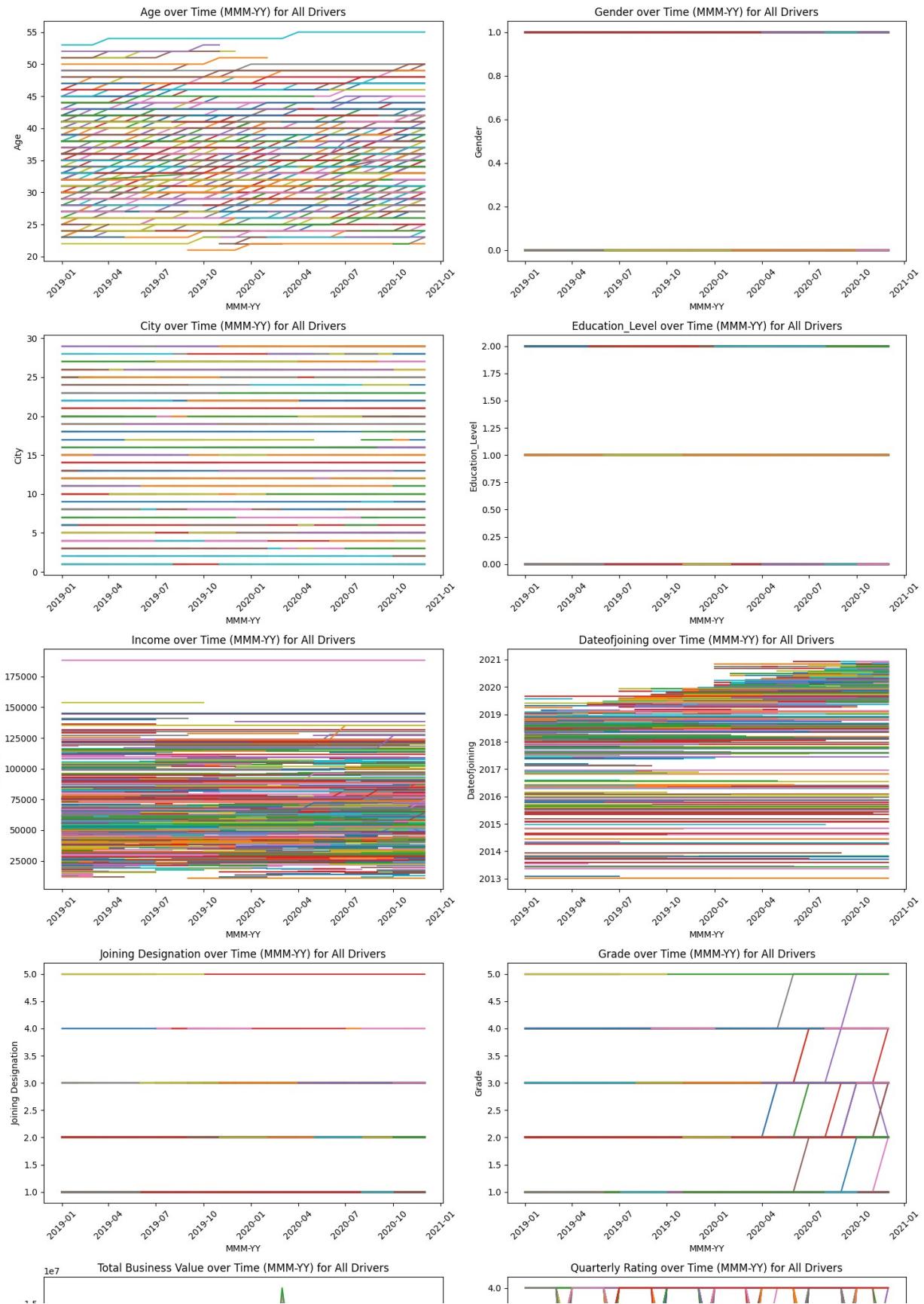
```

features =
["Age", "Gender", "City", "Education_Level", "Income", "Dateofjoining",
 "Joining Designation", "Grade", "Total Business Value",
 "Quarterly Rating"] # Selecting features that can be feature
engineered
plt.figure(figsize=(15,25))
k = 1
# Create line plots for each feature
sampled_driver_ids = np.random.choice(ola_1['Driver_ID'].unique(),
size=1000, replace=False)
for feature in features:
    plt.subplot(5,2,k)
    k += 1

    # Loop through each driver and plot their data
    for driver_id in sampled_driver_ids:
        driver_data = ola_1[ola_1['Driver_ID'] == driver_id]
        plt.plot(driver_data['MMM-YY'], driver_data[feature],
label=f'Driver {driver_id}')

    # Set plot title and labels
    plt.title(f'{feature} over Time (MMM-YY) for All Drivers')
    plt.xlabel('MMM-YY')
    plt.ylabel(feature)
    plt.xticks(rotation=45)
plt.tight_layout()
plt.show()

```



## Observation

Constant Features are (Gender, City, Education\_Level, Joining Designation, Dateofjoining)

monotonically increasing Feature are (Age, Income, Grade)

Fluctuating Features are (Total Business Value, Quarterly Rating)

For Constant features, No need to do any feature engineering, Just take first or last value while aggregating

Among monotonically increasing features, Age should be aggregated with last value or max value only. And can create features like Income\_raise and Grade\_raise (0 indicates constant, 1 indicates raise)

For Fluctuating Features, We need to capture mean, sum, std, range, increase/Decrease

For Income Feature also, We can calculate range. (No need to calculate mean, std because it is mostly constant, only increasing at one or two points)

Create `income_raise` Feature using increase/decrease/equal in Income

```
incm1 = (ola_1.groupby('Driver_ID').agg({'Income':'first'})  
['Income']).reset_index()  
incm2 = (ola_1.groupby('Driver_ID').agg({'Income':'last'})  
['Income']).reset_index()  
  
ola_2.head()
```

	Driver_ID	Target
0	1	1
1	2	0
2	4	1
3	5	1
4	6	0

```
ola_2 = ola_2.merge(incm1,on='Driver_ID')  
ola_2 = ola_2.merge(incm2,on='Driver_ID')
```

```
ola_2.head()
```

	Driver_ID	Target	Income_x	Income_y
0	1	1	57387	57387
1	2	0	67016	67016
2	4	1	65603	65603
3	5	1	46368	46368
4	6	0	78728	78728

```
def income_raise(row):  
    if row["Income_x"] == row["Income_y"]:  
        return 0  
    elif row["Income_x"] < row["Income_y"]:  
        return 1
```

```

    else:
        return -1

ola_2['Income_raise'] = ola_2.apply(income_raise, axis = 1)
ola_2.head()

   Driver_ID  Target  Income_x  Income_y  Income_raise
0            1       1      57387      57387          0
1            2       0      67016      67016          0
2            4       1      65603      65603          0
3            5       1      46368      46368          0
4            6       0      78728      78728          0

ola_2["Income_raise"].unique()
array([0, 1], dtype=int64)

```

### Observation

0 indicates income has not increased or not decreased

-1 indicates income has decreased. But No one has decrease in income

1 was not present --> indicates that for any driver, income was never raised in these 24 month period

### Create Grade\_raise Feature using increase/decrease/equal in Grade

```

GD1 = (ola_1.groupby('Driver_ID').agg({'Grade':'first'})
['Grade']).reset_index()
GD2 = (ola_1.groupby('Driver_ID').agg({'Grade':'last'})
['Grade']).reset_index()

ola_2 = ola_2.merge(GD1, on='Driver_ID')
ola_2 = ola_2.merge(GD2, on='Driver_ID')

def Grade_raise(row):
    if row["Grade_x"] == row["Grade_y"]:
        return 0
    elif row["Grade_x"] < row["Grade_y"]:
        return 1
    else:
        return -1

ola_2['Grade_raise'] = ola_2.apply(Grade_raise, axis = 1)
ola_2["Grade_raise"].unique()
array([0, 1], dtype=int64)

```

### Observation

- 0 indicates grade has not increased or not decreased
- 1 indicates grade has decreased, No one has Decrease in Grade.
- 1 was not present --> indicates that for any driver, grade was never raised in these 24 month period

Create `TBV_raise` Feature using increase/decrease/equal in Total Business Value

```

TBV1 = (ola_1.groupby('Driver_ID').agg({'Total Business Value':'first'})['Total Business Value']).reset_index()
TBV2 = (ola_1.groupby('Driver_ID').agg({'Total Business Value':'last'})['Total Business Value']).reset_index()

ola_2 = ola_2.merge(TBV1,on='Driver_ID')
ola_2 = ola_2.merge(TBV2,on='Driver_ID')

def TBV_raise(row):
    if row["Total Business Value_x"] == row["Total Business Value_y"]:
        return 0
    elif row["Total Business Value_x"] < row["Total Business Value_y"]:
        return 1
    else:
        return -1

ola_2['TBV_raise'] = ola_2.apply(TBV_raise, axis = 1)
ola_2["TBV_raise"].unique()
array([-1,  0,  1], dtype=int64)

```

### Observation

- 0 indicates that Total Business Value was not increase or not decreased
- 1 indicates Increased TBV
- 1 indicates Decreased TBV

Create `Promotion` Feature using increase/decrease/equal in Quarterly Rating

```

QR1 = (ola_1.groupby('Driver_ID').agg({'Quarterly Rating':'first'})['Quarterly Rating']).reset_index()
QR2 = (ola_1.groupby('Driver_ID').agg({'Quarterly Rating':'last'})['Quarterly Rating']).reset_index()

ola_2 = ola_2.merge(QR1,on='Driver_ID')
ola_2 = ola_2.merge(QR2,on='Driver_ID')

def promotion_demotion(row):

```

```

if row["Quarterly Rating_x"] == row["Quarterly Rating_y"]:
    return 0
elif row["Quarterly Rating_x"] < row["Quarterly Rating_y"]:
    return 1
else:
    return -1

ola_2['Rating_Promotion'] = ola_2.apply(promotion_demotion, axis = 1)

ola_2["Rating_Promotion"].unique()

array([ 0,  1, -1], dtype=int64)

```

### Observation

0 indicates No Promotion or Demotion in Quarterly Rating

1 indicates Promotion

-1 indicates Demotion

### Grouping all the remaining column using aggregate method

```

functions = {
    'MMM-YY': 'count',
    'Driver_ID': 'first',
    'Age': 'max',
    'Gender': 'last',
    'City': 'last',
    'Education_Level': 'last',
    'Income': ['sum', lambda x: x.max() - x.min()], # Sum and Range
    (max - min)
    'Dateofjoining': 'first',
    'LastWorkingDate': 'last',
    'Joining Designation': 'last',
    'Grade': 'last',
    'Total Business Value': ['sum', 'mean', 'std', lambda x: x.max() -
    x.min()], # Sum, Mean, Std, Range
    'Quarterly Rating': ['last', 'mean', 'sum', 'std', lambda x:
    x.max() - x.min()] # last, Mean, Sum, Std, Range
}

# Applying the aggregation functions
ola_1_aggregated = ola_1.groupby('Driver_ID').aggregate(functions)

# Renaming the columns for better readability
ola_1_aggregated.columns = [
    'Reportings', 'Driver_ID', 'Age', 'Gender', 'City',
    'Education_Level',
    'Income_sum', 'Income_range', 'Dateofjoining', 'LastWorkingDate',
    'Joining Designation', 'Grade', 'TBV_sum', 'TBV_mean', 'TBV_std',
    'TBV_range', 'QuarterlyRating_last',
]

```

```

    'QuarterlyRating_mean', 'QuarterlyRating_sum',
'QuarterlyRating_std', 'QuarterlyRating_range'
]

# Adding 'join_month' and 'join_year' based on 'Dateofjoining'
ola_1_aggregated['Join_month'] =
pd.to_datetime(ola_1_aggregated['Dateofjoining']).dt.month
ola_1_aggregated['Join_year'] =
pd.DatetimeIndex(ola_1_aggregated['Dateofjoining']).year

```

## Observation

Reportings is obtained by counting the Number of reporting dates in these two years  
join\_month and join\_year was extracted from DateofJoining. So that we can delete the DateofJoining feature.

Can calculate Tenure by subtracting Dateofjoining from LastWorkingDate. But as LastWorkingDate is Target feature, It is not good to have Tenure feature

Merging ola\_1\_aggregated and ola\_2

```

ola_1_aggregated.reset_index(drop=True, inplace=True)

ola_1 = ola_1_aggregated.merge(ola_2, on='Driver_ID')
ola_1.head()

   Reportings  Driver_ID  Age  Gender  City  Education_Level
Income_sum \
0            3         1   28      0.0    23                  2
172161
1            2         2   31      0.0     7                  2
134032
2            5         4   43      0.0    13                  2
328015
3            3         5   29      0.0     9                  0
139104
4            5         6   31      1.0    11                  1
393640

   Income_range  Dateofjoining  LastWorkingDate  Joining  Designation
Grade \
0            0    2018-12-24    2019-11-03        1
1
1            0    2020-06-11          NaT        2
2
2            0    2019-07-12    2020-04-27        2
2
3            0    2019-09-01    2019-07-03        1
1
4            0    2020-07-31          NaT        3
3

```

0	TBV_sum	TBV_mean	TBV_std	TBV_range	QuarterlyRating_last	\				
1	1715580	571860.0	1.601755e+06	3046540	2					
2	0	0.0	0.000000e+00	0	1					
3	350000	70000.0	1.565248e+05	350000	1					
4	120360	40120.0	6.948988e+04	120360	1					
5	1265000	253000.0	5.657252e+05	1265000	2					
0	QuarterlyRating_mean	QuarterlyRating_sum	QuarterlyRating_std	QuarterlyRating_range	Join_month	Join_year	Target	Income_x	Income_y	\
1	2.0	6	0.000000	0.000000	0	2018	1	57387	57387	
2	1.0	2	0.000000	0.000000	6	2020	0	67016	67016	
3	1.0	5	0.000000	0.000000	7	2019	1	65603	65603	
4	1.6	8	0.547723	0.547723	9	2019	1	46368	46368	
5	QuarterlyRating_range	Join_month	Join_year	Target	Income_x	Income_y	\	78728	78728	
0	0	12	2018	1	57387	57387				
1	0	6	2020	0	67016	67016				
2	0	7	2019	1	65603	65603				
3	0	9	2019	1	46368	46368				
4	1	7	2020	0	78728	78728				
0	Income_raise	Grade_x	Grade_y	Grade_raise	Total Business	Value_x	\			
1	0	1	1	0	2381060	2381060				
2	0	2	2	0	0	0				
3	0	2	2	0	0	0				
4	0	1	1	0	0	0				
5	0	3	3	0	0	0				
0	Total Business	Value_y	TBV_raise	Quarterly Rating_x	Quarterly Rating_y	\				
1	0	-1	2	2	2	2				
2	0	0	1	1	1	1				
3	0	0	1	1	1	1				
4	0	0	1	1	1	1				

1		0	0	1
4				
2				
	Rating_Promotion			
0		0		
1		0		
2		0		
3		0		
4		1		

## Droping the unwanted features

```
ola_1.isna().sum()
```

Reportings	0
Driver_ID	0
Age	0
Gender	0
City	0
Education_Level	0
Income_sum	0
Income_range	0
Dateofjoining	0
LastWorkingDate	765
Joining_Designation	0
Grade	0
TBV_sum	0
TBV_mean	0
TBV_std	181
TBV_range	0
QuarterlyRating_last	0
QuarterlyRating_mean	0
QuarterlyRating_sum	0
QuarterlyRating_std	181
QuarterlyRating_range	0
Join_month	0
Join_year	0
Target	0
Income_x	0
Income_y	0
Income_raise	0
Grade_x	0
Grade_y	0
Grade_raise	0
Total_Business_Value_x	0
Total_Business_Value_y	0
TBV_raise	0
Quarterly_Rating_x	0
Quarterly_Rating_y	0

```
Rating_Promotion          0
dtype: int64

np.sum(ola_1[(ola_1["QuarterlyRating_std"].isna() |
(ola_1["TBV_std"].isna()))]["Reportings"] != 1)
```

## Observation

So if the number of reportings are equal to 1, Std is showing as null values, So Those std can make as zeros

Out of 2381 Drivers, 765 Drivers are having null Last working date, means they are still not churned off.

```
ola_1['TBV_std'].fillna(0, inplace=True)
ola_1['QuarterlyRating_std'].fillna(0, inplace=True)

C:\Users\saina\AppData\Local\Temp\ipykernel_11784\3652393424.py:1:
FutureWarning: A value is trying to be set on a copy of a DataFrame or
Series through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never
work because the intermediate object on which we are setting values
always behaves as a copy.
```

For example, when doing `'df[col].method(value, inplace=True)'`, try using `'df.method({col: value}, inplace=True)'` or `df[col] = df[col].method(value)` instead, to perform the operation inplace on the original object.

```
ola_1['TBV_std'].fillna(0, inplace=True)
C:\Users\saina\AppData\Local\Temp\ipykernel_11784\3652393424.py:2:
FutureWarning: A value is trying to be set on a copy of a DataFrame or
Series through chained assignment using an inplace method.
The behavior will change in pandas 3.0. This inplace method will never
work because the intermediate object on which we are setting values
always behaves as a copy.
```

For example, when doing `'df[col].method(value, inplace=True)'`, try using `'df.method({col: value}, inplace=True)'` or `df[col] = df[col].method(value)` instead, to perform the operation inplace on the original object.

```

Value_x', 'Total Business Value_y',
        'Quarterly Rating_x', 'Quarterly
Rating_y'], axis=1, inplace=True)
ola_1['Gender'] = ola_1['Gender'].astype('int64')

```

## Observation

Driver\_ID is primary key. Not advisable to use in model building

Date of joining is separated as join\_month and join\_year. So we can delete this date feature

LastWorkingDate has many null values. It was converted as Target. So We should delete this feature. It is not advisable to create Tenure feature also.

Remaining features in above columns list are unwanted features which are created during feature engineering.

	Reportings	Age	Gender	City	Education_Level	Income_sum
Income_range \						
0	3	28	0	23	2	172161
0	2	31	0	7	2	134032
0	5	43	0	13	2	328015
0	3	29	0	9	0	139104
0	5	31	1	11	1	393640
TBV_range \	Joining	Designation	Grade	TBV_sum	TBV_mean	TBV_std
0			1	1715580	571860.0	1.601755e+06
3046540			2	0	0.0	0.000000e+00
0			2	350000	70000.0	1.565248e+05
350000			1	120360	40120.0	6.948988e+04
120360			3	1265000	253000.0	5.657252e+05
1265000						
QuarterlyRating_last \	QuarterlyRating_mean	QuarterlyRating_sum				
0	2	2.0				6
1	1	1.0				2
2	1	1.0				5
3	1	1.0				3
4	2	1.6				8

	QuarterlyRating_std	QuarterlyRating_range	Join_month	Join_year
Target \	0.000000	0	12	2018
0	0.000000	0	6	2020
1	0.000000	0	7	2019
2	0.000000	0	9	2019
3	0.000000	0	7	2020
4	0.547723	1	7	2020
0				
	Income_raise	Grade_raise	TBV_raise	Rating_Promotion
0	0	0	-1	0
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	1

Changing the order of columns, Taking Target Feature to End, DriverID to Start

```
ola = ola_1[['Age', 'City', 'Education_Level', 'Gender', 'Grade',
'Grade_raise', 'Income_raise', 'Income_range',
'Income_sum', 'Join_month', 'Join_year', 'Joining
Designation', 'QuarterlyRating_last', 'QuarterlyRating_mean',
'QuarterlyRating_range',
'QuarterlyRating_std', 'QuarterlyRating_sum', 'Rating_Promotion',
'Reportings',
'TBV_mean', 'TBV_raise', 'TBV_range',
'TBV_std', 'TBV_sum', 'Target']]
```

## STATISTICAL SUMMARY OF MODIFIED DATASET

```
sum(ola.isna().sum())
0
ola.describe().T
```

	count	mean	std
min \			
Age	2381.0	3.366317e+01	5.983375e+00
2.100000e+01			
City	2381.0	1.533557e+01	8.371843e+00
1.000000e+00			
Education_Level	2381.0	1.007560e+00	8.162900e-01
0.000000e+00			
Gender	2381.0	4.103318e-01	4.919972e-01

		25%	50%	75%
max				
Age	29.0	33.000000	3.700000e+01	
5.800000e+01				
City	8.0	15.000000	2.200000e+01	
0.000000e+00				
Grade	2381.0	2.096598e+00	9.415218e-01	
1.000000e+00				
Grade_raise	2381.0	1.805964e-02	1.331951e-01	
0.000000e+00				
Income_raise	2381.0	1.805964e-02	1.331951e-01	
0.000000e+00				
Income_range	2381.0	1.270987e+02	9.731710e+02	
0.000000e+00				
Income_sum	2381.0	5.267603e+05	6.231633e+05	
1.088300e+04				
Join_month	2381.0	6.958001e+00	3.221762e+00	
1.000000e+00				
Join_year	2381.0	2.018536e+03	1.609597e+00	
2.013000e+03				
Joining_Designation	2381.0	1.820244e+00	8.414334e-01	
1.000000e+00				
QuarterlyRating_last	2381.0	1.427971e+00	8.098389e-01	
1.000000e+00				
QuarterlyRating_mean	2381.0	1.566304e+00	7.196520e-01	
1.000000e+00				
QuarterlyRating_range	2381.0	7.866443e-01	9.851023e-01	
0.000000e+00				
QuarterlyRating_std	2381.0	3.320004e-01	4.021221e-01	
0.000000e+00				
QuarterlyRating_sum	2381.0	1.611844e+01	2.000545e+01	
1.000000e+00				
Rating_Promotion	2381.0	-4.199916e-02	5.840312e-01	-
1.000000e+00				
Reportings	2381.0	8.023520e+00	6.783590e+00	
1.000000e+00				
TBV_mean	2381.0	3.120854e+05	4.495705e+05	-
1.979329e+05				
TBV_raise	2381.0	-6.677866e-02	7.095928e-01	-
1.000000e+00				
TBV_range	2381.0	1.270168e+06	2.322940e+06	
0.000000e+00				
TBV_std	2381.0	3.800797e+05	5.893408e+05	
0.000000e+00				
TBV_sum	2381.0	4.586742e+06	9.127115e+06	-
1.385530e+06				
Target	2381.0	6.787064e-01	4.670713e-01	
0.000000e+00				

2.900000e+01			
Education_Level	0.0	1.000000	2.000000e+00
2.000000e+00			
Gender	0.0	0.000000	1.000000e+00
1.000000e+00			
Grade	1.0	2.000000	3.000000e+00
5.000000e+00			
Grade_raise	0.0	0.000000	0.000000e+00
1.000000e+00			
Income_raise	0.0	0.000000	0.000000e+00
1.000000e+00			
Income_range	0.0	0.000000	0.000000e+00
1.215500e+04			
Income_sum	139895.0	292980.000000	6.514560e+05
4.522032e+06			
Join_month	5.0	7.000000	1.000000e+01
1.200000e+01			
Join_year	2018.0	2019.000000	2.020000e+03
2.020000e+03			
Joining_Designation	1.0	2.000000	2.000000e+00
5.000000e+00			
QuarterlyRating_last	1.0	1.000000	2.000000e+00
4.000000e+00			
QuarterlyRating_mean	1.0	1.000000	2.000000e+00
4.000000e+00			
QuarterlyRating_range	0.0	0.000000	1.000000e+00
3.000000e+00			
QuarterlyRating_std	0.0	0.000000	5.477226e-01
1.643168e+00			
QuarterlyRating_sum	3.0	7.000000	1.900000e+01
9.600000e+01			
Rating_Promotion	0.0	0.000000	0.000000e+00
1.000000e+00			
Reportings	3.0	5.000000	1.000000e+01
2.400000e+01			
TBV_mean	0.0	150624.444444	4.294988e+05
3.972128e+06			
TBV_raise	-1.0	0.000000	0.000000e+00
1.000000e+00			
TBV_range	0.0	491420.000000	1.446060e+06
3.505256e+07			
TBV_std	0.0	200529.702804	4.826555e+05
8.074168e+06			
TBV_sum	0.0	817680.000000	4.173650e+06
9.533106e+07			
Target	0.0	1.000000	1.000000e+00
1.000000e+00			

## Observation

The dataset is reduced or aggregate to Driver level and there are no null values in the dataset

By observing min and max values of numerical columns - Income and Business Value will have significant outliers.

Reportings are count of occurrences. So it can be treated as numerical column.

## CHAPTER 3: DATA VISUALIZATION

### UNIVARIATE ANALYSIS

```
cont_cols = ['Age', 'Income_range', 'Income_sum',
'QuarterlyRating_mean', 'QuarterlyRating_range',
'QuarterlyRating_std',
        'QuarterlyRating_sum', 'Reportings',
'TBV_mean', 'TBV_range', 'TBV_std', 'TBV_sum']

cat_cols = ['City', 'Education_Level', 'Gender', 'Grade',
'Grade_raise', 'Income_raise', 'Join_month', 'Join_year',
        'Joining
Designation', 'QuarterlyRating_last', 'Rating_Promotion', 'TBV_raise', 'Ta
rget']
```

#### Observation

Driver\_ID is primary Key so not included in Categorical Cols

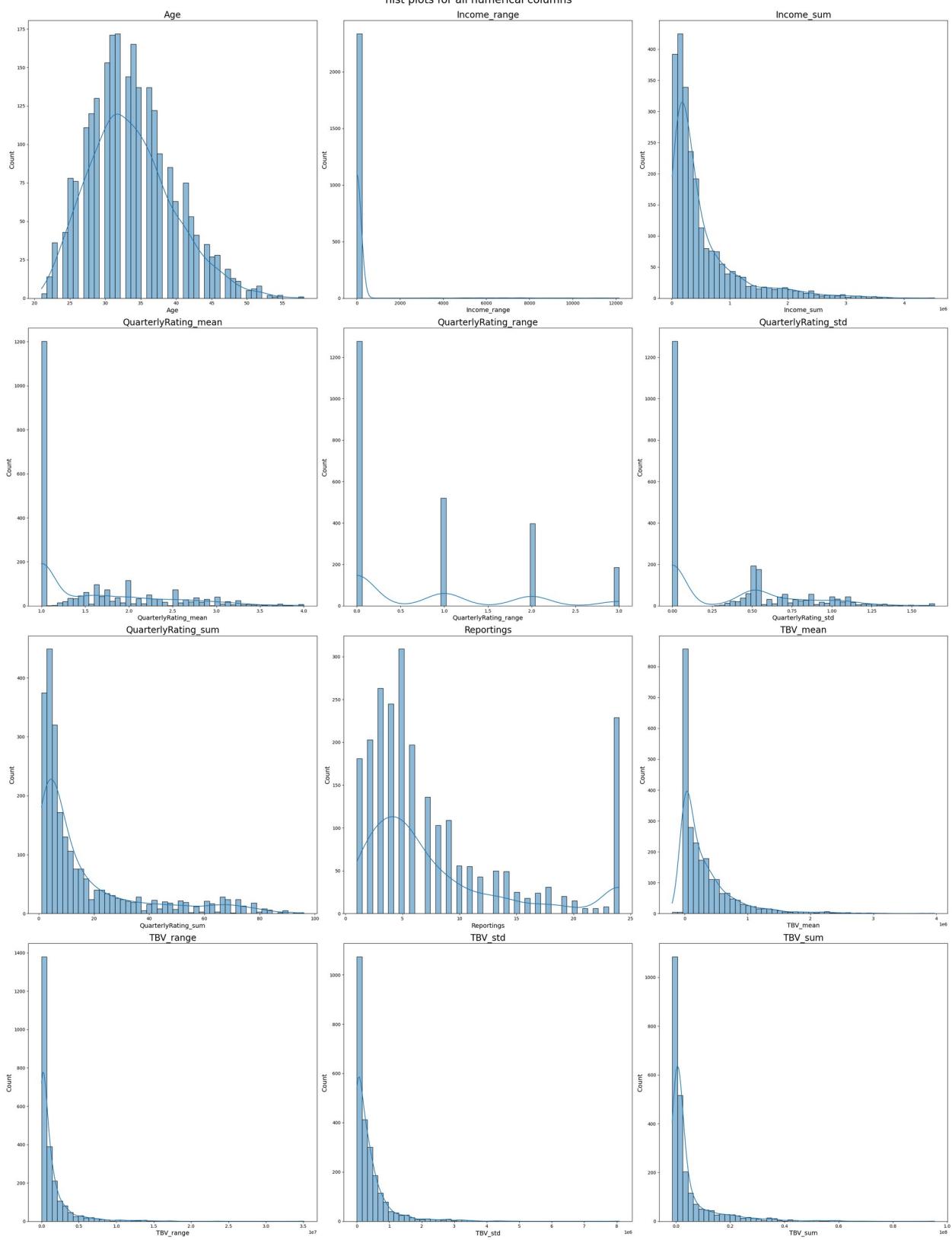
#### Distribution plots of all Numerical Columns

```
fig = plt.figure(figsize = (3*10,len(cont_cols)*10/3))
plt.suptitle("hist plots for all numerical columns\n", fontsize=24)
k = 1
for i in cont_cols:
    plt.subplot(math.ceil(len(cont_cols)/3), 3, k)
    plt.title("{}".format(i), fontsize = 20)
    k += 1
    plot = sns.histplot(data=ola,x = i,kde = True,bins = 50)

    # Increase label and legend font sizes
    plot.set_xlabel(plot.get_xlabel(), fontsize=14)
    plot.set_ylabel(plot.get_ylabel(), fontsize=14)

    warnings.filterwarnings('ignore')
plt.tight_layout()
plt.subplots_adjust(top=0.96)
plt.show()
warnings.filterwarnings('ignore')
```

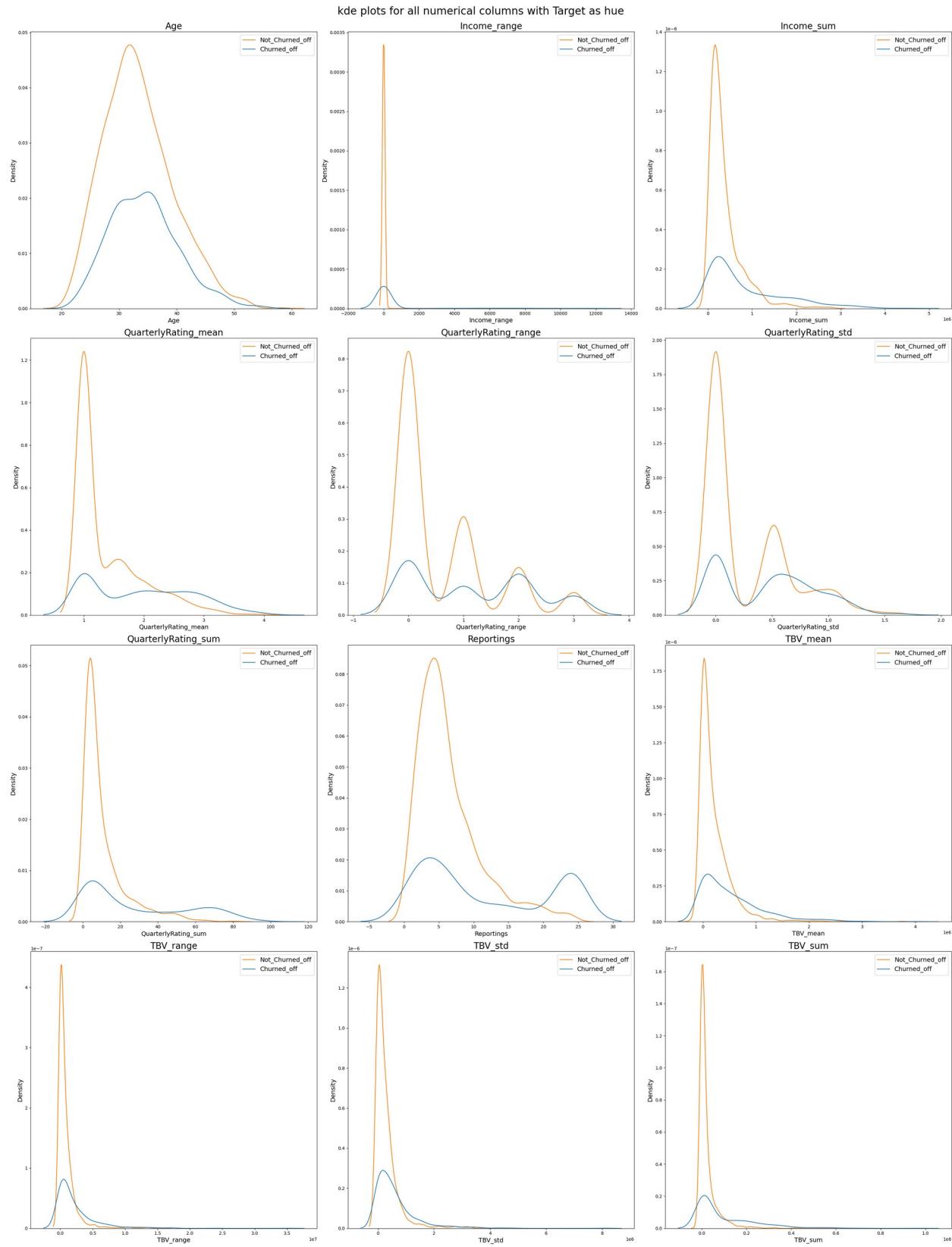
hist plots for all numerical columns



```
fig = plt.figure(figsize = (3*10,len(cont_cols)*10/3))
plt.suptitle("kde plots for all numerical columns with Target as hue\\n", fontsize=24)
k = 1
for i in cont_cols:
    plt.subplot(math.ceil(len(cont_cols)/3),3,k)
    plt.title("{}".format(i), fontsize = 20)
    k += 1
    plot = sns.kdeplot(data=ola,x = i,hue = "Target")

    # Increase label and legend font sizes
    plot.set_xlabel(plot.get_xlabel(), fontsize=14)
    plot.set_ylabel(plot.get_ylabel(), fontsize=14)
    plt.legend(plot.get_legend_handles_labels,labels =
    ["Not_Churned_off","Churned_off"],fontsize = 14, title_fontsize = 16)

warnings.filterwarnings('ignore')
plt.tight_layout()
plt.subplots_adjust(top=0.96)
plt.show()
warnings.filterwarnings('ignore')
```



## Observation

Income and TBV was left Skewed, These features may have outliers.

Age in range of 25 to 40 are significant

Reportings range from 1 to 24. Indicating 24 months data. Distribution is concentrated in the range of 1-9. Indicating many drivers are leaving in first 9 months itself.

On observing kde plot shows that there is imbalanced distribution between Churn and not churn drivers.

kde plot of reportings, We can observe that there is unusual spike at 24 because of not churned drivers only.

kde plot of not churned drivers clearly dominates the churned off drivers.

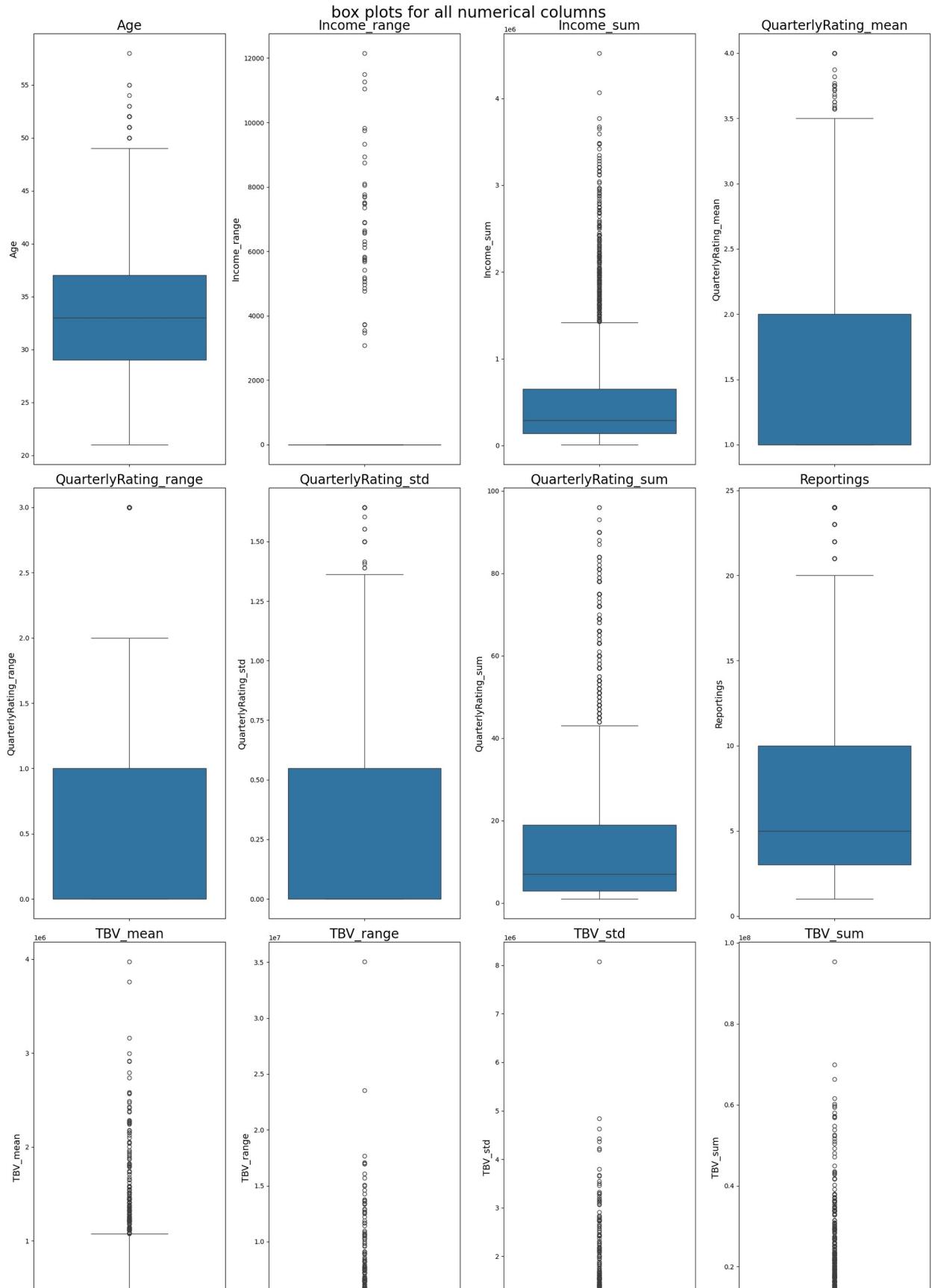
## Box plots of all Numerical columns

```
fig = plt.figure(figsize = (4*5, len(cont_cols)*5/(4/2)))
plt.suptitle("box plots for all numerical columns\n", fontsize=24)
k = 1
for i in cont_cols:
    plt.subplot(math.ceil(len(cont_cols)/4), 4, k)
    plt.title("{}".format(i), fontsize = 20)
    k += 1
    plot = sns.boxplot(data=ola,y = i)

    # Increase label and legend font sizes
    plot.set_xlabel(plot.get_xlabel(), fontsize=14)
    plot.set_ylabel(plot.get_ylabel(), fontsize=14)

warnings.filterwarnings('ignore')
plt.tight_layout()
plt.subplots_adjust(top=0.96)
plt.show()
warnings.filterwarnings('ignore')
```

box plots for all numerical columns



## Observation

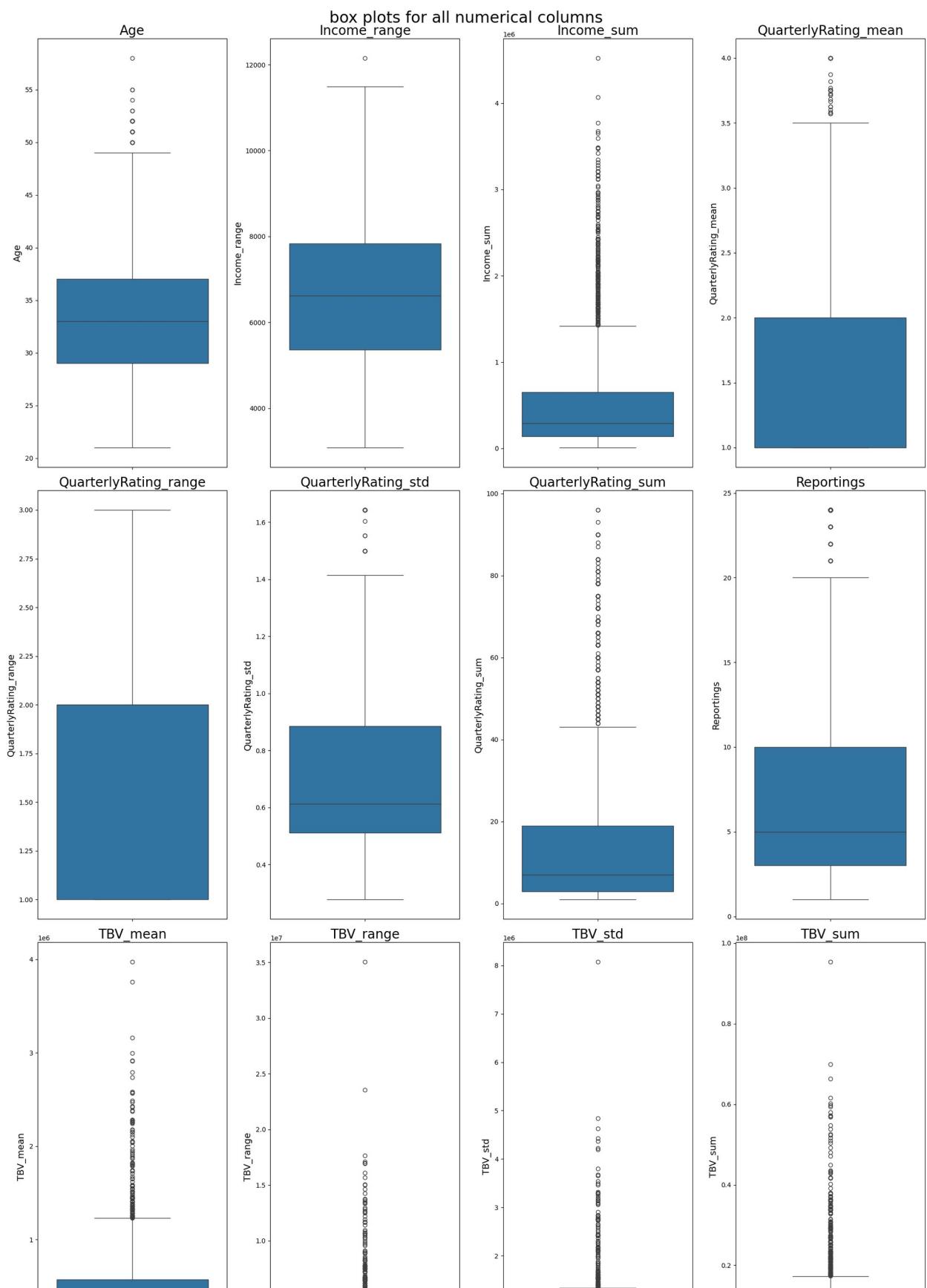
May be because of lot of zeros in the distribution, Box plots are showing lot of outliers

Lets remove the zeros and plot them again

```
fig = plt.figure(figsize = (4*5,len(cont_cols)*5/(4/2)))
plt.suptitle("box plots for all numerical columns\n",fontsize=24)
k = 1
for i in cont_cols:
    plt.subplot(math.ceil(len(cont_cols)/4),4,k)
    plt.title("{}".format(i),fontsize = 20)
    k += 1
    plot = sns.boxplot(data=ola[ola[i] != 0],y = i)

    # Increase label and legend font sizes
    plot.set_xlabel(plot.get_xlabel(), fontsize=14)
    plot.set_ylabel(plot.get_ylabel(), fontsize=14)

    warnings.filterwarnings('ignore')
plt.tight_layout()
plt.subplots_adjust(top=0.96)
plt.show()
warnings.filterwarnings('ignore')
```

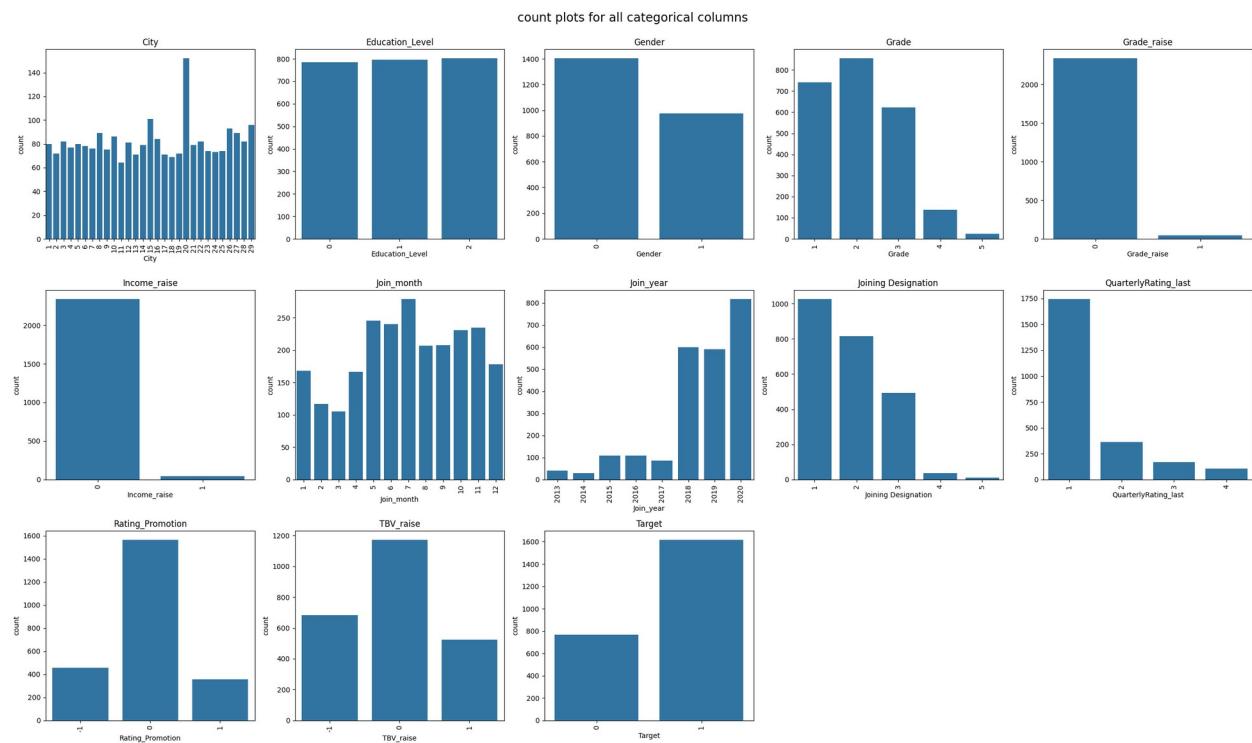


## Observation

Clearly we can identify There are many outliers with large range in TBV and Income. Those should be handled by transformations because we have very small dataset.

## Count plots of all categorical columns

```
fig = plt.figure(figsize=(25,15))
fig.suptitle("count plots for all categorical columns\n", fontsize =
"xx-large" )
k = 1
for i in cat_cols:
    plt.subplot(3,5,k)
    plt.title("{}\n".format(i))
    sns.countplot(data=ola,x = i)
    plt.xticks(rotation = 90)
    k = k+1
plt.tight_layout()
plt.show()
```



## Observation

Male --> 0, Female --> 1, Male Drivers > Female Drivers

Education level is uniformly distributed among 10+, 12+, Graduate

Joining Designation is lower for most of the drivers. JD count is decreasing with increase in JD value

Grade 2 > Grade 1 Drivers. Grade 4 and Grade 5 are very less.

Quarterly rating is 1 for most of the drivers

In july month, most of the drivers has joined in ola

In data, Joining year ranges from 2013 to 2020. Most of the members are new Drivers. This indicates, Old Drivers are churning mostly.

Rating promotion was rarely increased or decreased

Income is constant for most of the drivers. Income was never increased in these 24 months. But decreased for few.

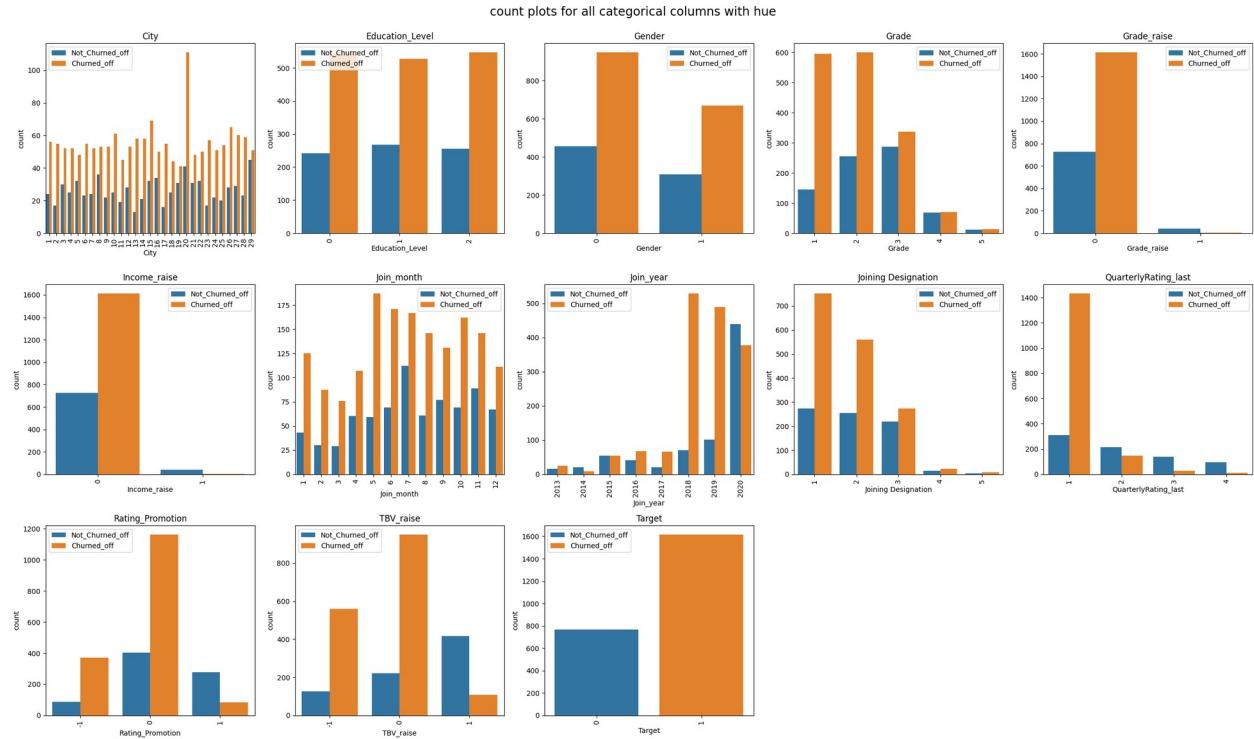
Grade is constant for most of the drivers. Grade was never increased in these 24 months. But decreased for few.

Total Business Value is constant for approx 50% drivers, 25% decrease, 25% increase

City 20 has more number of drivers

Target feature is imbalanced. 1 indicates churn, 0 indicate no churn. 1 dominates 0

```
fig = plt.figure(figsize=(25,15))
fig.suptitle("count plots for all categorical columns with hue\\n", fontsize = "xx-large" )
k = 1
for i in cat_cols:
    plt.subplot(3,5,k)
    plt.title("{}".format(i))
    sns.countplot(data=ola,x = i, hue = "Target")
    plt.legend(plot.get_legend_handles_labels,labels =
    ["Not_Churned_off","Churned_off"])
    plt.xticks(rotation = 90)
    k = k+1
plt.tight_layout()
plt.show()
```



## Observation

Positive Rating Promotion and TBV Raise is significantly low for Churned off Drivers.

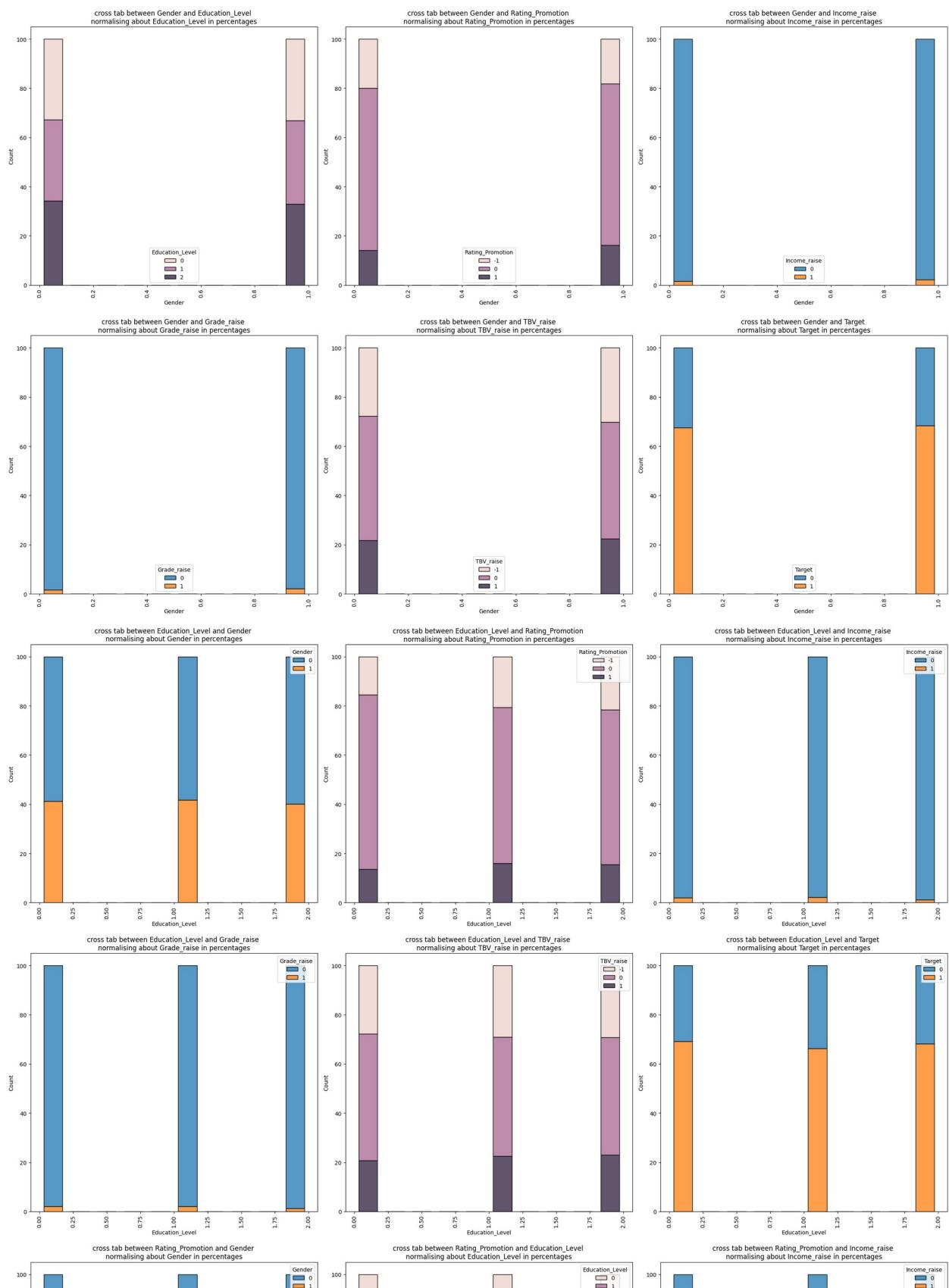
## BIVARIATE ANALYSIS OF IMPORTANT FEATURES

### Categorical Vs Categorical

```
imp_cat_features =
['Gender', 'Education_Level', 'Rating_Promotion', 'Income_raise',
'Grade_raise', 'TBV_raise', "Target"]
cat_perm = list(permutations(imp_cat_features, 2))
fig = plt.figure(figsize = (3*8, len(cat_perm)*8/3))
plt.suptitle("Stacked hist plots of imp_categorical_features permutation\n", fontsize="xx-large")
k = 1
for p,q in cat_perm:
    if (ola[p].nunique()<50) and (ola[q].nunique()<50):
        plt.subplot(math.ceil(len(cat_perm)/3), 3, k)
        plt.title(f"cross tab between {p} and {q} \nnormalising about {q} in percentages")
        k += 1
        plot = ola.groupby([p])
[q].value_counts(normalize=True).mul(100).reset_index(name='percentage')
        sns.histplot(x = p , hue = q, weights= 'percentage', multiple
= 'stack', data=plot, shrink = 0.7)
        plt.xticks(rotation = 90)
```

```
    warnings.filterwarnings('ignore')
plt.tight_layout()
plt.subplots_adjust(top=0.97)
plt.show()
warnings.filterwarnings('ignore')
```

Stacked hist plots of imp\_categorical\_features permutation



## Observation

Gender and Education Level has no effect on raises

if Rating is promoted , There is high chance he/she may raise TBV. Viceversa is also True

if income decreases, Rating may decrease or increase. TBV may Decrease.

High correlation between Grade raise and income raise. If income decreases, Grade definitely decreases. and vice versa.

When TBV raises or rating promotion increases, Churn off also raises

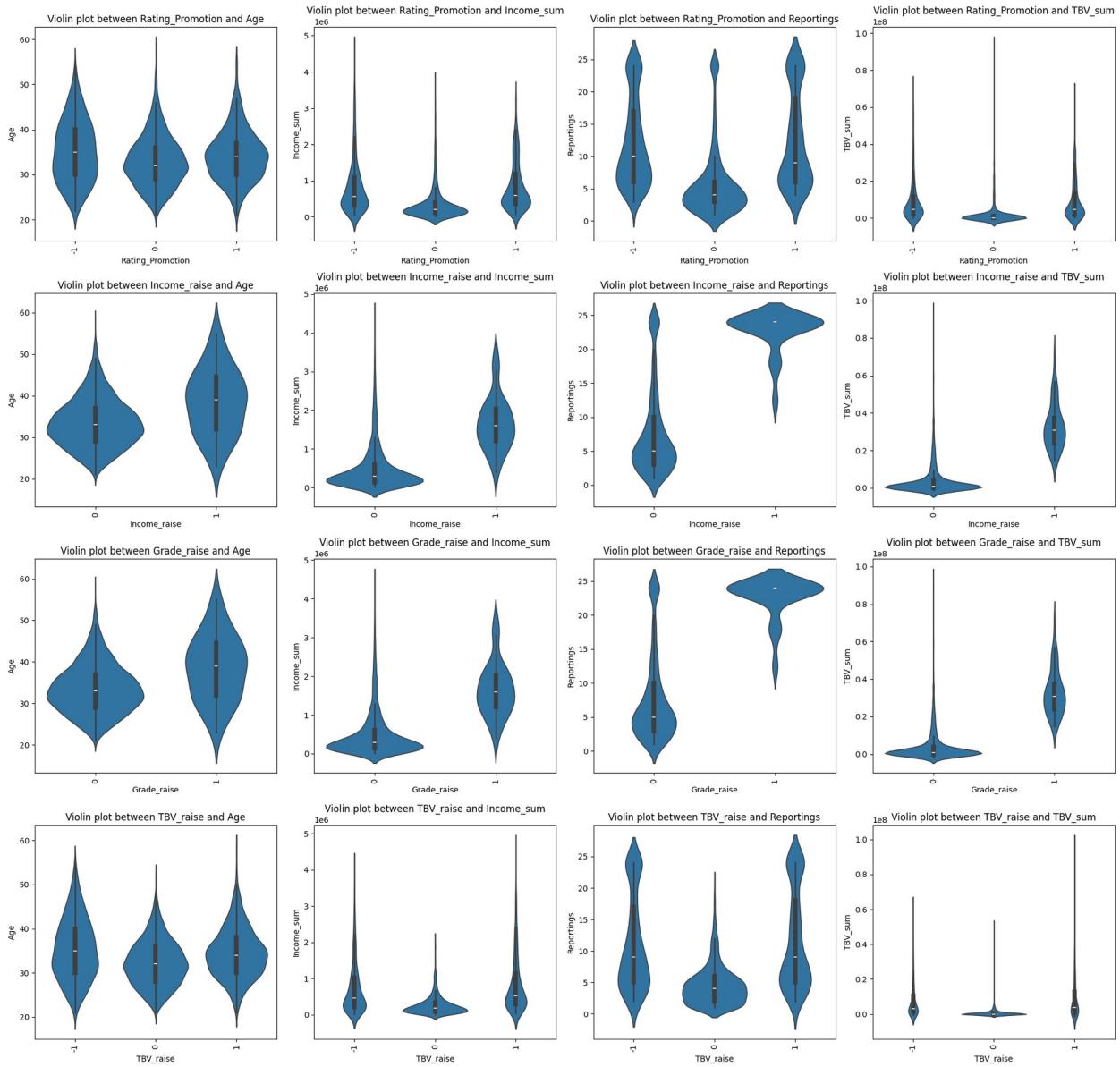
## Numerical Vs Categorical

```
imp_cat_features = ['Rating_Promotion','Income_raise', 'Grade_raise',
'TBV_raise'] # Gender and Education Level was removed because of no
effect
imp_cont_features = ['Age', 'Income_sum','Reportings', 'TBV_sum']
Cat_Vs_cont = []
for i in range(len(imp_cat_features)):
    for j in range(len(imp_cont_features)):
        if (ola[imp_cat_features[i]].nunique()<50):
            Cat_Vs_cont.append((imp_cat_features[i],
imp_cont_features[j]))
print(Cat_Vs_cont)
print(len(Cat_Vs_cont))

[('Rating_Promotion', 'Age'), ('Rating_Promotion', 'Income_sum'),
('Rating_Promotion', 'Reportings'), ('Rating_Promotion', 'TBV_sum'),
('Income_raise', 'Age'), ('Income_raise', 'Income_sum'),
('Income_raise', 'Reportings'), ('Income_raise', 'TBV_sum'),
('Grade_raise', 'Age'), ('Grade_raise', 'Income_sum'), ('Grade_raise',
'Reportings'), ('Grade_raise', 'TBV_sum'), ('TBV_raise', 'Age'),
('TBV_raise', 'Income_sum'), ('TBV_raise', 'Reportings'),
('TBV_raise', 'TBV_sum')]
16

fig = plt.figure(figsize = (4*5,len(Cat_Vs_cont)*5/4))
plt.suptitle("violin plot of Numerical column with respect to
Categorical Columns",fontsize = "xx-large")
k = 1
for p,q in Cat_Vs_cont:
    plt.subplot(math.ceil(len(Cat_Vs_cont)/5),4,k)
    plt.title(f"Violin plot between {p} and {q}")
    k +=1
    sns.violinplot(data = ola,x = p,y= q)
    plt.xticks(rotation = 90)
plt.tight_layout()
plt.subplots_adjust(top=0.93)
plt.show()
warnings.filterwarnings('ignore')
```

Violin plot of Numerical column with respect to Categorical Columns



## Observation

Age was well distributed in all plots

We already know that income raise, TBV Raise and Grade raise are correlated.

Number of reportings are significantly higher if income was decreased or Grade was decreased

```
imp_cat_features = ["Gender", "Education_Level", "Joining
Designation", "Grade", "QuarterlyRating_last", "Join_month", "Join_year"]
imp_cont_features = ['Age', 'Income_sum', 'Reportings', 'TBV_sum']
Cat_Vs_cont = []
for i in range(len(imp_cat_features)):
```

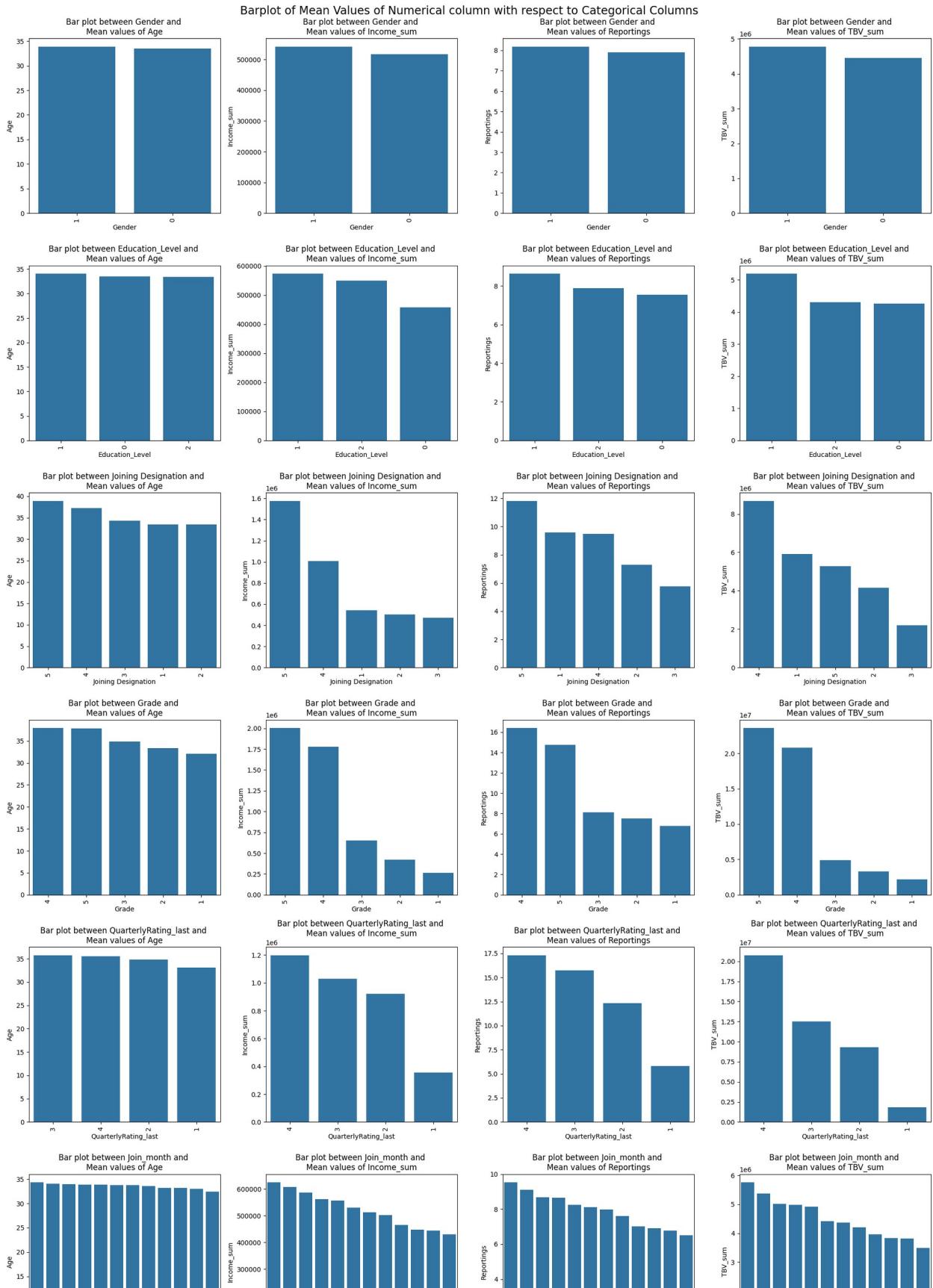
```

for j in range(len(imp_cont_features)):
    if (ola[imp_cat_features[i]].nunique()<50):
        Cat_Vs_cont.append((imp_cat_features[i],
imp_cont_features[j]))
print(Cat_Vs_cont)
print(len(Cat_Vs_cont))

[('Gender', 'Age'), ('Gender', 'Income_sum'), ('Gender',
'Reportings'), ('Gender', 'TBV_sum'), ('Education_Level', 'Age'),
('Education_Level', 'Income_sum'), ('Education_Level', 'Reportings'),
('Education_Level', 'TBV_sum'), ('Joining_Designation', 'Age'),
('Joining_Designation', 'Income_sum'), ('Joining_Designation',
'Reportings'), ('Joining_Designation', 'TBV_sum'), ('Grade', 'Age'),
('Grade', 'Income_sum'), ('Grade', 'Reportings'), ('Grade',
'TBV_sum'), ('QuarterlyRating_last', 'Age'), ('QuarterlyRating_last',
'Income_sum'), ('QuarterlyRating_last', 'Reportings'),
('QuarterlyRating_last', 'TBV_sum'), ('Join_month', 'Age'),
('Join_month', 'Income_sum'), ('Join_month', 'Reportings'),
('Join_month', 'TBV_sum'), ('Join_year', 'Age'), ('Join_year',
'Income_sum'), ('Join_year', 'Reportings'), ('Join_year', 'TBV_sum')]
28

fig = plt.figure(figsize = (4*5,len(Cat_Vs_cont)*5/4))
plt.suptitle("Barplot of Mean Values of Numerical column with respect
to Categorical Columns\n",fontsize = "xx-large")
k = 1
for p,q in Cat_Vs_cont:
    plt.subplot(math.ceil(len(Cat_Vs_cont)/4),4,k)
    plt.title(f"Bar plot between {p} and \nMean values of {q}")
    k += 1
    df = pd.DataFrame(ola.groupby([p])[q].mean().reset_index())
    sns.barplot(data = df,x = p,y= q,order =
df.sort_values(q,ascending = False)[p])
    plt.xticks(rotation = 90)
plt.tight_layout()
plt.subplots_adjust(top=0.96)
plt.show()
warnings.filterwarnings('ignore')

```



## Observation

Income for Joining Designation 5 and 4 significantly higher than remaining Joining Designations

Income and Quarterly Rating directly correlated

Income is higher for old employees (2013 to 2015)

TBV is directly related to join year, quarterly rating,

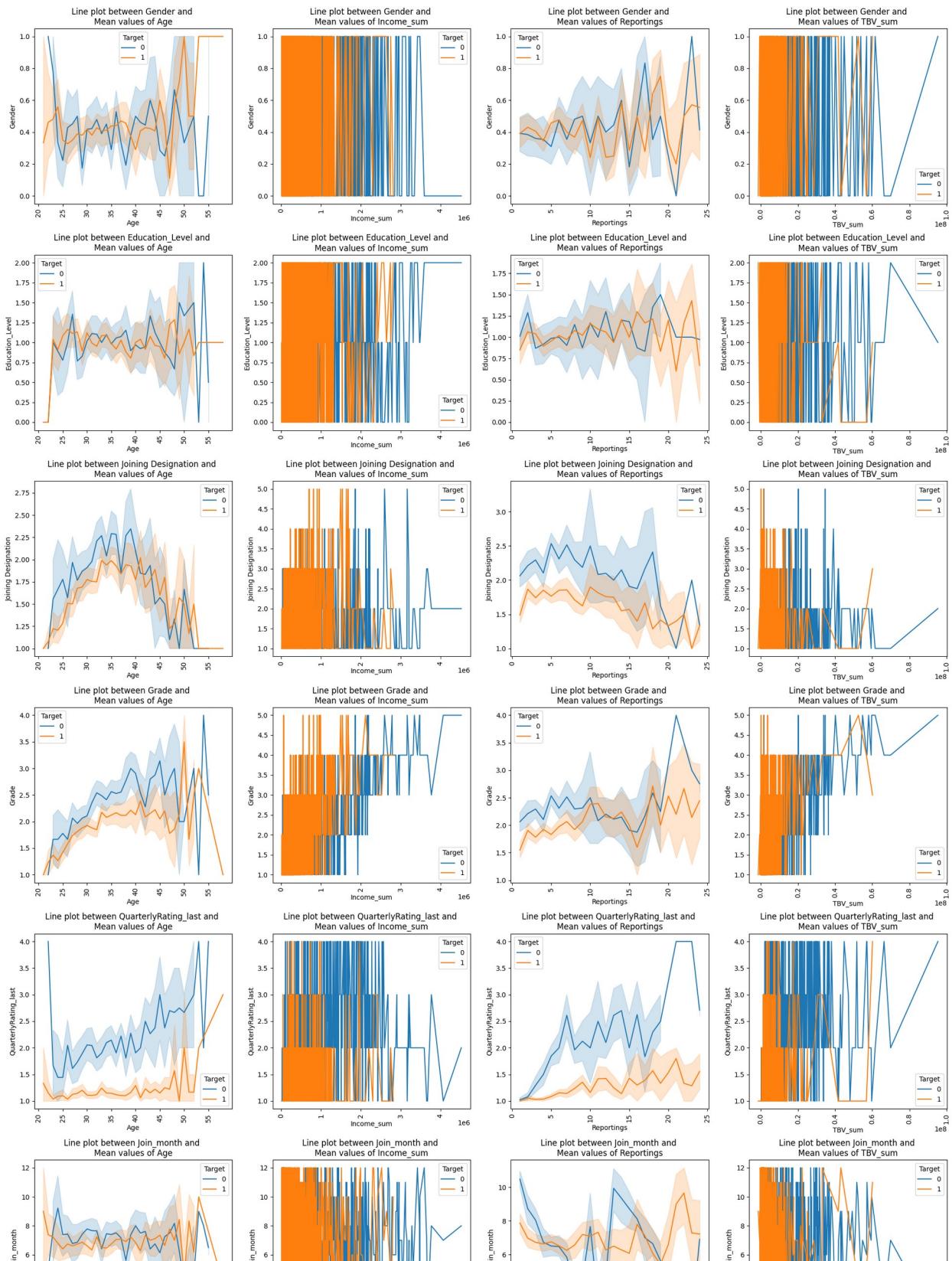
TBV of jD = 4 is higher

Reportings are higher for older employees

## MULTI-VARIATE ANALYSIS OF IMPORTANT FEATURES

```
fig = plt.figure(figsize = (4*5,len(Cat_Vs_cont)*5/4))
plt.suptitle("Lineplot of Mean Values of Numerical column with respect
to Categorical Columns\n",fontsize = "xx-large")
k = 1
for p,q in Cat_Vs_cont:
    plt.subplot(math.ceil(len(Cat_Vs_cont)/4),4,k)
    plt.title(f"Line plot between {p} and \nMean values of {q}")
    k += 1
    sns.lineplot(data = ola,x=q,y=p,hue = "Target")
    plt.xticks(rotation = 90)
plt.tight_layout()
plt.subplots_adjust(top=0.93)
plt.show()
warnings.filterwarnings('ignore')
```

Lineplot of Mean Values of Numerical column with respect to Categorical Columns



## Observation

Images are showing Aggregated Grade feature with 95 % confidence region.

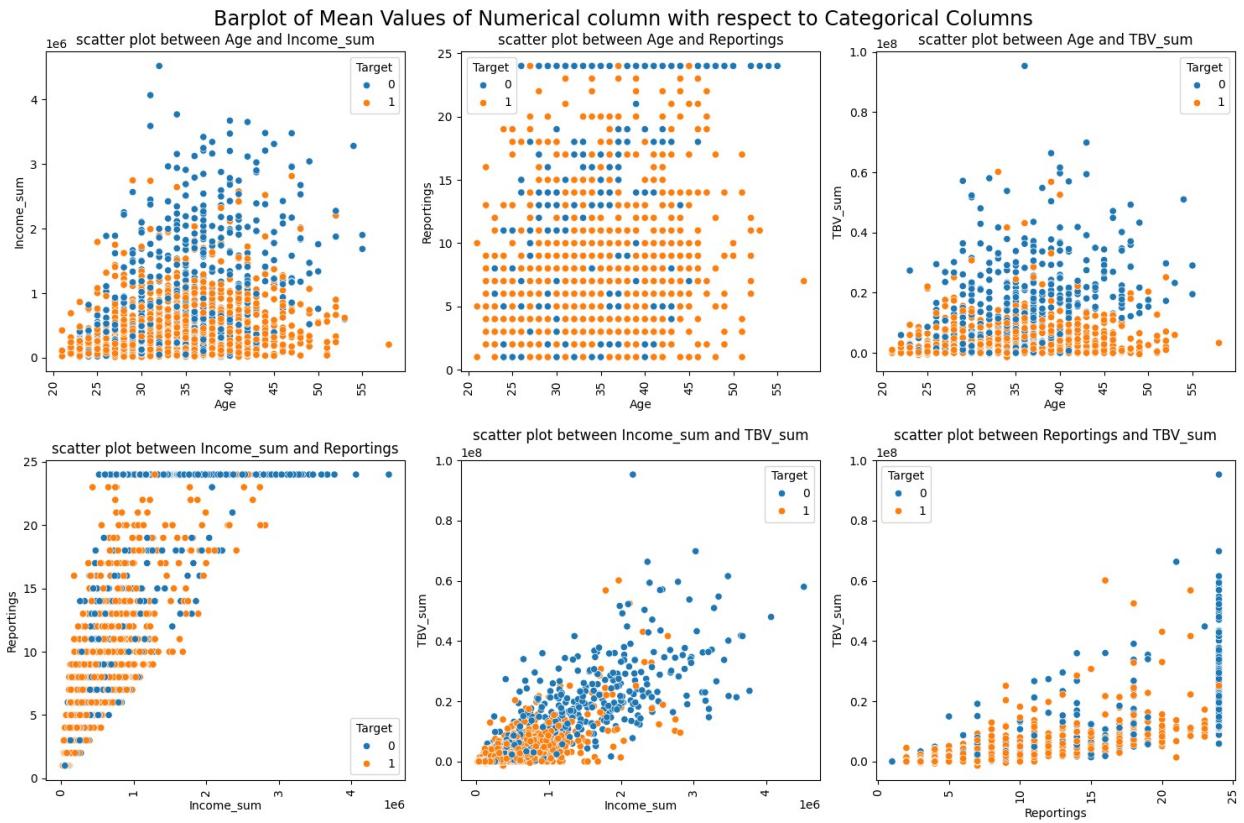
Joining Designation, Grade, Quarterly Rating of Churned off(1) Drivers is Lesser compared to Joining Designation, Grade, Quarterly Rating of not churned off Drivers(0) with respect to their similar age.

## Numerical vs Numerical vs Target

```
cont_cols = ['Age', 'Income_sum', 'Reportings', 'TBV_sum']

cont_comb = list(combinations(cont_cols,2))

fig = plt.figure(figsize = (3*5,len(cont_comb)*5/3))
plt.suptitle("Barplot of Mean Values of Numerical column with respect to Categorical Columns\n", fontsize = "xx-large")
k = 1
for p,q in cont_comb:
    plt.subplot(math.ceil(len(cont_comb)/3),3,k)
    plt.title(f"scatter plot between {p} and {q}")
    k += 1
    sns.scatterplot(data = ola,x=p,y=q,hue = "Target")
    plt.xticks(rotation = 90)
plt.tight_layout()
plt.subplots_adjust(top=0.93)
plt.show()
warnings.filterwarnings('ignore')
```

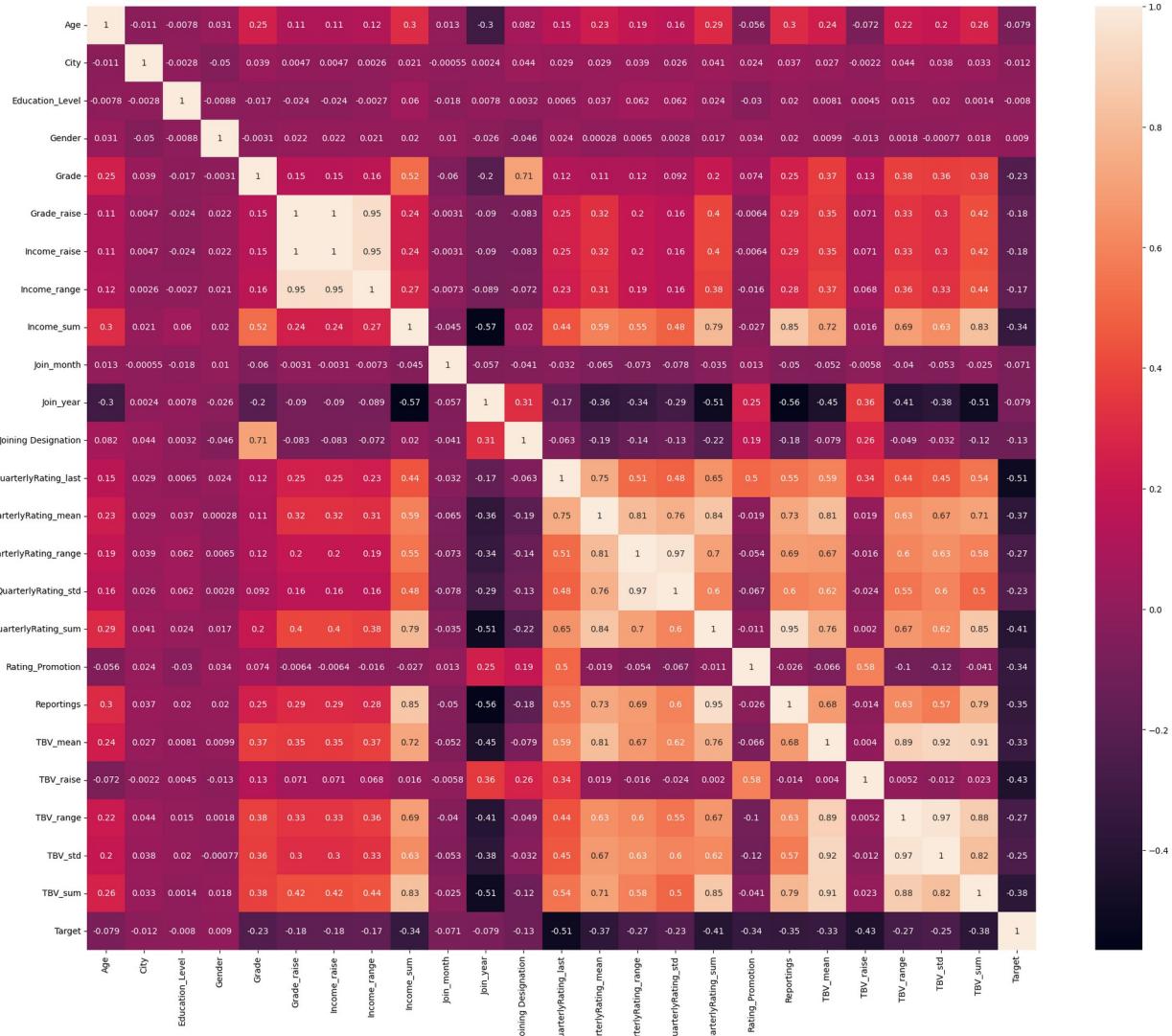


## Observation

Blue dots (not churned off Drivers ) are concentrated upper and right wards. This indicates Not churned off dominates the churned off drivers in TBV, Income, Reportings at all Ages.

## HEAT MAPS

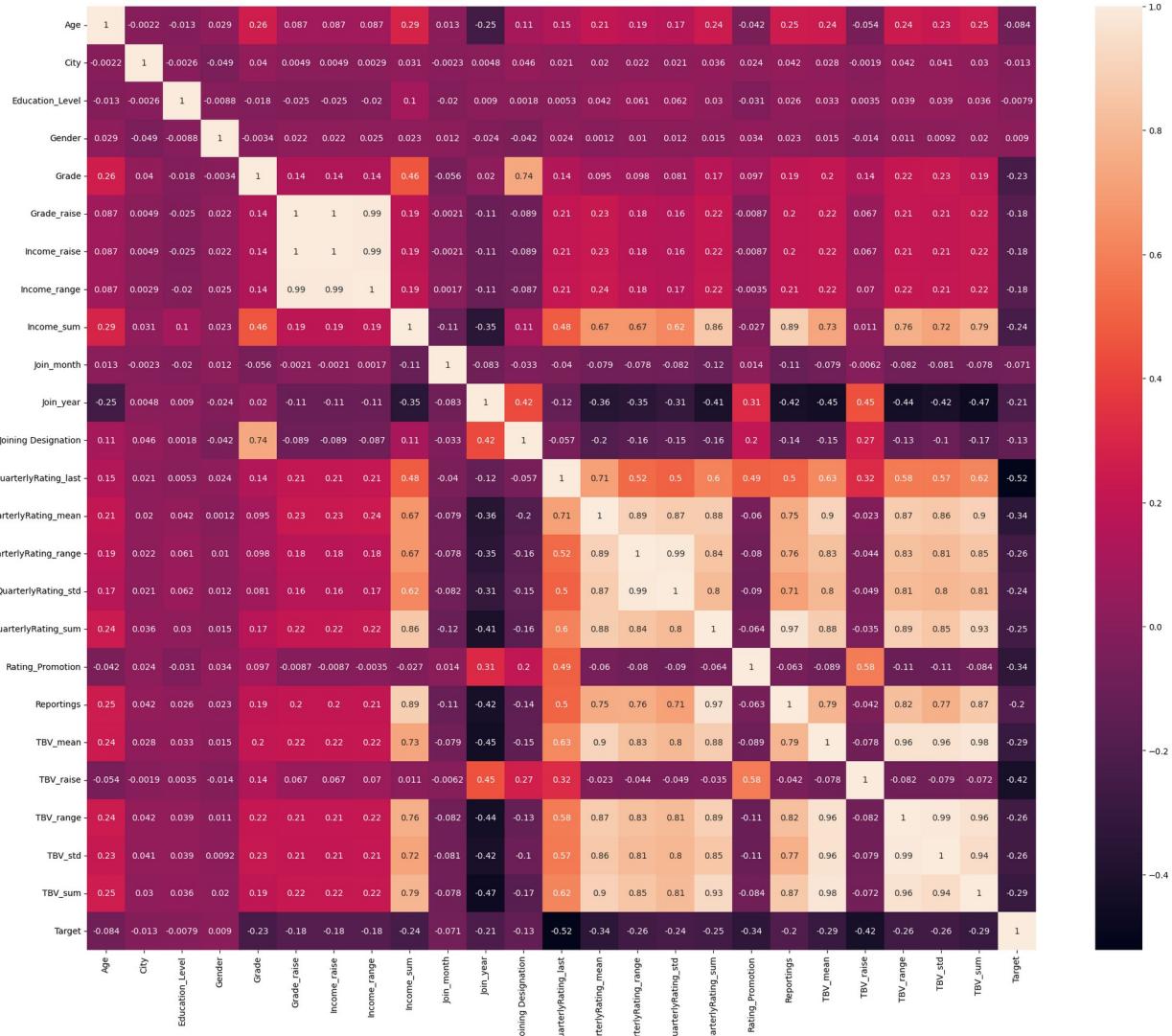
```
corr = ola.corr()
plt.figure(figsize=(25,20))
sns.heatmap(corr, annot=True)
plt.show()
```



```

corr = ola.corr(method= 'spearman')
plt.figure(figsize=(25,20))
sns.heatmap(corr,annot=True)
plt.show()

```



## Observation

High positive correlation pairs are observed between Income, Grade, Quarterly Rating and TBV related pairs

High negative correlation pairs are observed between Join month , join year with respect to other features

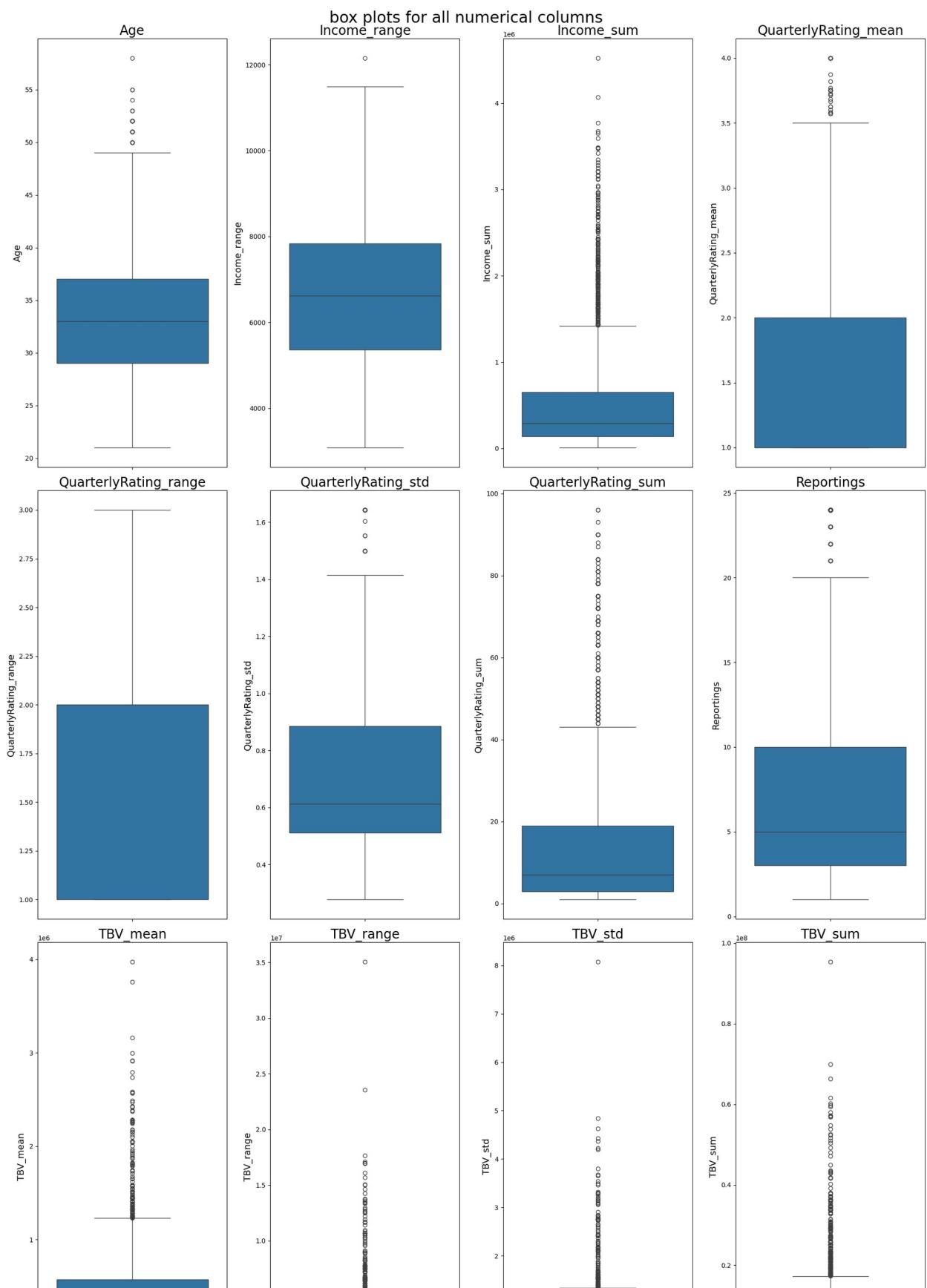
# CHAPTER 4: DATA PREPROCESSING AND BASIC MODEL BUILDING

## OUTLIER TREATMENT AND BOX PLOTS OF NUMERICAL COLUMNS

```
cont_cols = ['Age', 'Income_range', 'Income_sum',
'QuarterlyRating_mean', 'QuarterlyRating_range',
'QuarterlyRating_std',
    'QuarterlyRating_sum', 'Reportings',
'TBV_mean', 'TBV_range', 'TBV_std', 'TBV_sum']
fig = plt.figure(figsize = (4*5,len(cont_cols)*5/(4/2)))
plt.suptitle("box plots for all numerical columns\n", fontsize=24)
k = 1
for i in cont_cols:
    plt.subplot(math.ceil(len(cont_cols)/4),4,k)
    plt.title("{}".format(i), fontsize = 20)
    k += 1
    plot = sns.boxplot(data=ola[ola[i] != 0], y = i)

    # Increase label and legend font sizes
    plot.set_xlabel(plot.get_xlabel(), fontsize=14)
    plot.set_ylabel(plot.get_ylabel(), fontsize=14)

    warnings.filterwarnings('ignore')
plt.tight_layout()
plt.subplots_adjust(top=0.96)
plt.show()
warnings.filterwarnings('ignore')
```



## Observation

Clearly all the numerical columns have some outliers outside the upper whisker. We have small dataset. So we can't remove them

Age and Reportings are meaningfull outliers. No need to handle them

Income\_sum, QuarterlyRating\_sum, TBV\_mean, TBV\_range,TBV\_std,TBV\_sum has many outliers

For features with positive and Negative, It is better to use Yeo Johnson Transformation

If features has only Positive values, it is better to use simple log transformation.

## Transformation of Income and TBV to reduce the impact of Outliers

### Income log Transformation

```
outlier_treatment_features =
[ "Income_sum", "QuarterlyRating_sum", "TBV_mean", "TBV_range", "TBV_std", "TBV_sum"]

for i in outlier_treatment_features:
    print(i, ola[i].min())

Income_sum 10883
QuarterlyRating_sum 1
TBV_mean -197932.85714285713
TBV_range 0
TBV_std 0.0
TBV_sum -1385530
```

## Observation

TBV\_mean, TBV\_sum has positive and Negative Features

```
# creating two datasets, one with outliers, another without outliers
ola_no_out = ola.copy(deep=True)

# Initialize the PowerTransformer with method='yeo-johnson'
pt = PowerTransformer(method='yeo-johnson')

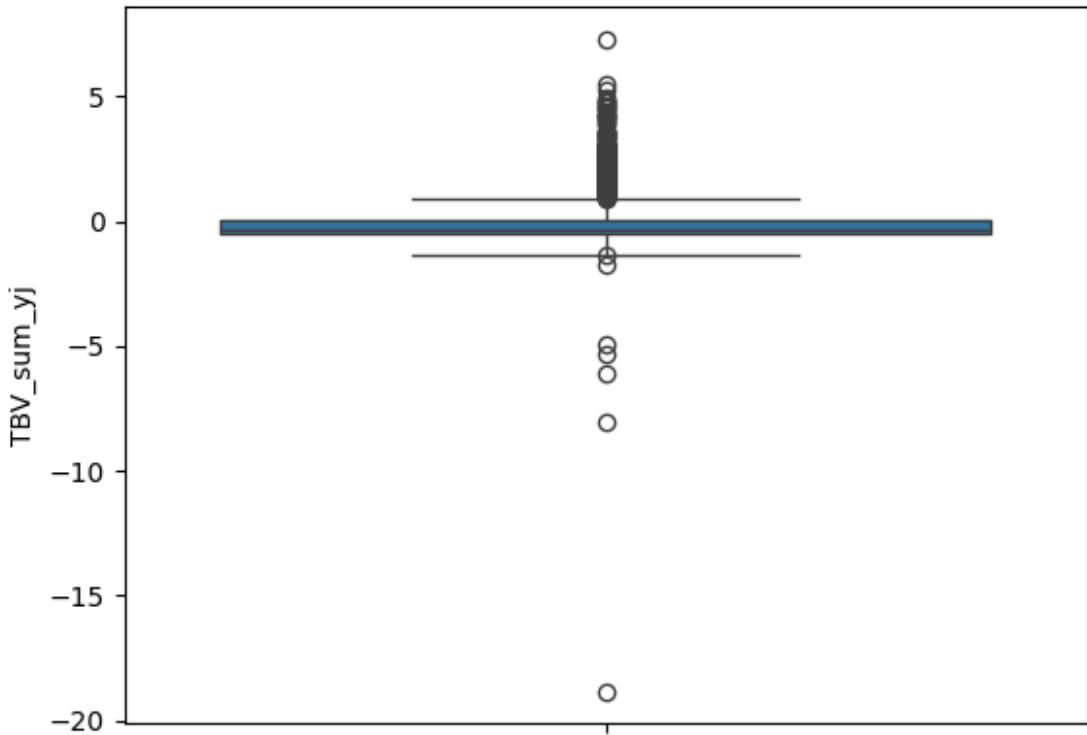
# Fit and transform the TBV column
ola_no_out['TBV_sum_yj'] = pt.fit_transform(ola_no_out[['TBV_sum']])
ola_no_out['TBV_mean_yj'] = pt.fit_transform(ola_no_out[['TBV_mean']])

# If you want to store the lambda value for TBV transformation
tbv_lambda = pt.lambdas_[0]

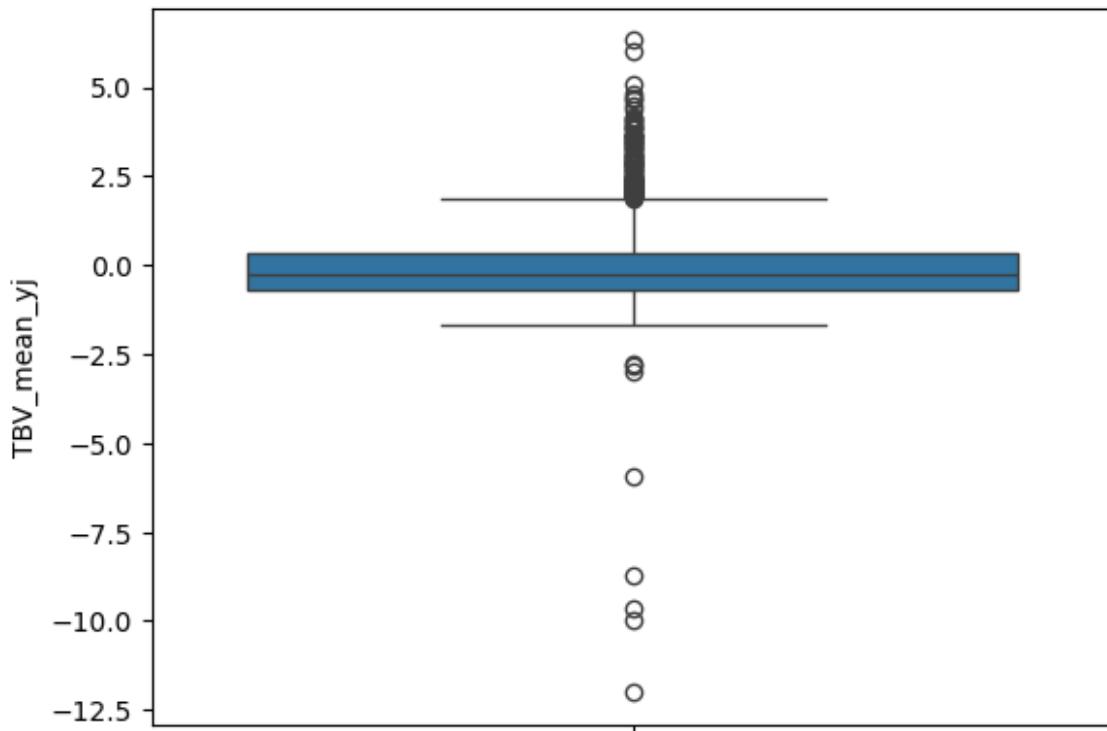
# To inverse transform later if needed
# ola_no_out['TBV_Original'] =
# pt.inverse_transform(ola_no_out[['TBV_sum_yj']])

sns.boxplot(data = ola_no_out,y = "TBV_sum_yj")
```

```
<Axes: ylabel='TBV_sum_yj'>
```



```
sns.boxplot(data = ola_no_out,y = "TBV_mean_yj")  
<Axes: ylabel='TBV_mean_yj'>
```



```
ola_no_out.drop(["TBV_sum_yj", "TBV_mean_yj"], axis = 1, inplace= True)
```

### Observation

Even after applying yeo\_johnson method, Outliers are huge.

So Lets try to seperate the TBV magnitude and sign as two seperate columns and apply log or box cox transformation on magnitude column

TBV negative indicates that Driver has Significant number of cancellation/refunds/EMI adjustments through out his tenure.

```
# Separate positive and negative values
ola_no_out['TBV_sum_positive'] = ola_no_out['TBV_sum'].apply(lambda x:
x if x > 0 else 0)
ola_no_out['TBV_sum_negative'] = ola_no_out['TBV_sum'].apply(lambda x:
-x if x < 0 else 0)

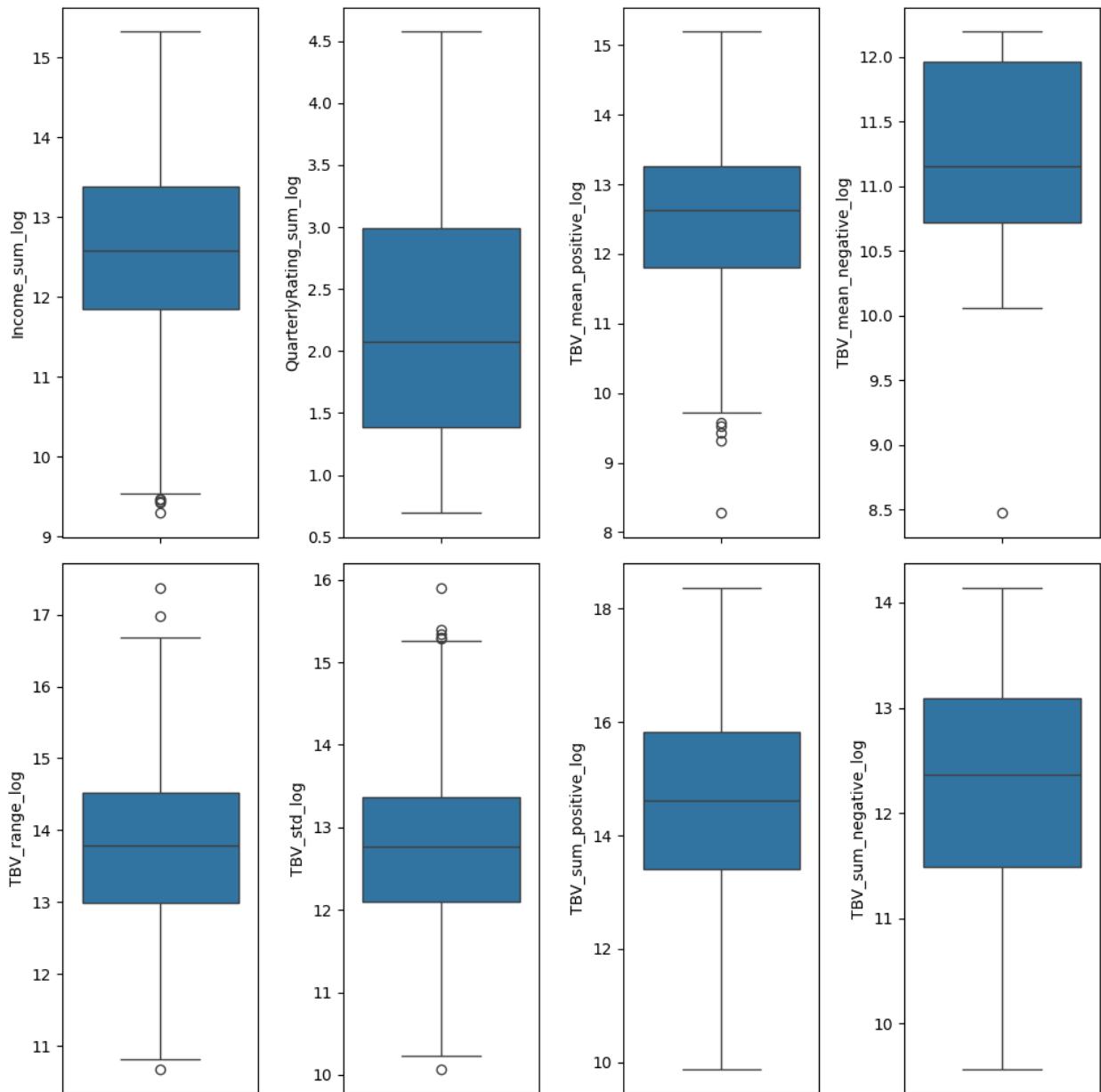
ola_no_out['TBV_mean_positive'] = ola_no_out['TBV_mean'].apply(lambda
x: x if x > 0 else 0)
ola_no_out['TBV_mean_negative'] = ola_no_out['TBV_mean'].apply(lambda
x: -x if x < 0 else 0)

log_features =
["Income_sum", "QuarterlyRating_sum", "TBV_mean_positive", "TBV_mean_nega
tive",
"TBV_range", "TBV_std", "TBV_sum_positive", "TBV_sum_negative"]
```

```

plt.figure(figsize=(10,10))
k = 1
for i in log_features:
    plt.subplot(2,4,k)
    k += 1
    str = "_" .join([i,"log"])
    ola_no_out[str] = np.log1p(ola_no_out[i])
    sns.boxplot(data = ola_no_out[ola_no_out[str] != 0], y = str)
plt.tight_layout()
plt.show()

```



## Observation

All the outliers are handled by using log transformation successfully

```
ola_no_out.drop(["Income_sum", "QuarterlyRating_sum", "TBV_mean", "TBV_range", "TBV_std", "TBV_sum"], axis = 1, inplace= True)
```

Checking which is better among ola or ola\_no\_out by using simple Logistic regression model

Dataset with outliers

```
# Separating Target feature from the dataset
y = ola["Target"]
X = ola.drop("Target",axis = 1)

# Train test split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# standard Scaling
st = StandardScaler()
X_train = st.fit_transform(X_train.values)
X_test = st.transform(X_test.values)

# Logistic Regression model
model = LogisticRegression(random_state=42)
model.fit(X_train, y_train)
y_pred_train = model.predict(X_train)
y_pred_test = model.predict(X_test)

# accuracy
print("train_accuracy",accuracy_score(y_train,y_pred_train))
print("test_accuracy",accuracy_score(y_test,y_pred_test))

train_accuracy 0.8592436974789915
test_accuracy 0.870020964360587
```

Dataset without outliers

```
# Separating Target feature from the dataset
y = ola_no_out["Target"]
X = ola_no_out.drop("Target",axis = 1)

# Train test split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# standard Scaling
st = StandardScaler()
X_train = st.fit_transform(X_train.values)
X_test = st.transform(X_test.values)
```

```
# Logistic Regression model
model = LogisticRegression(random_state=42)
model.fit(X_train, y_train)
y_pred_train = model.predict(X_train)
y_pred_test = model.predict(X_test)

# accuracy
print("train_accuracy",accuracy_score(y_train,y_pred_train))
print("test_accuracy",accuracy_score(y_test,y_pred_test))

train_accuracy 0.8865546218487395
test_accuracy 0.909853249475891
```

### Observation

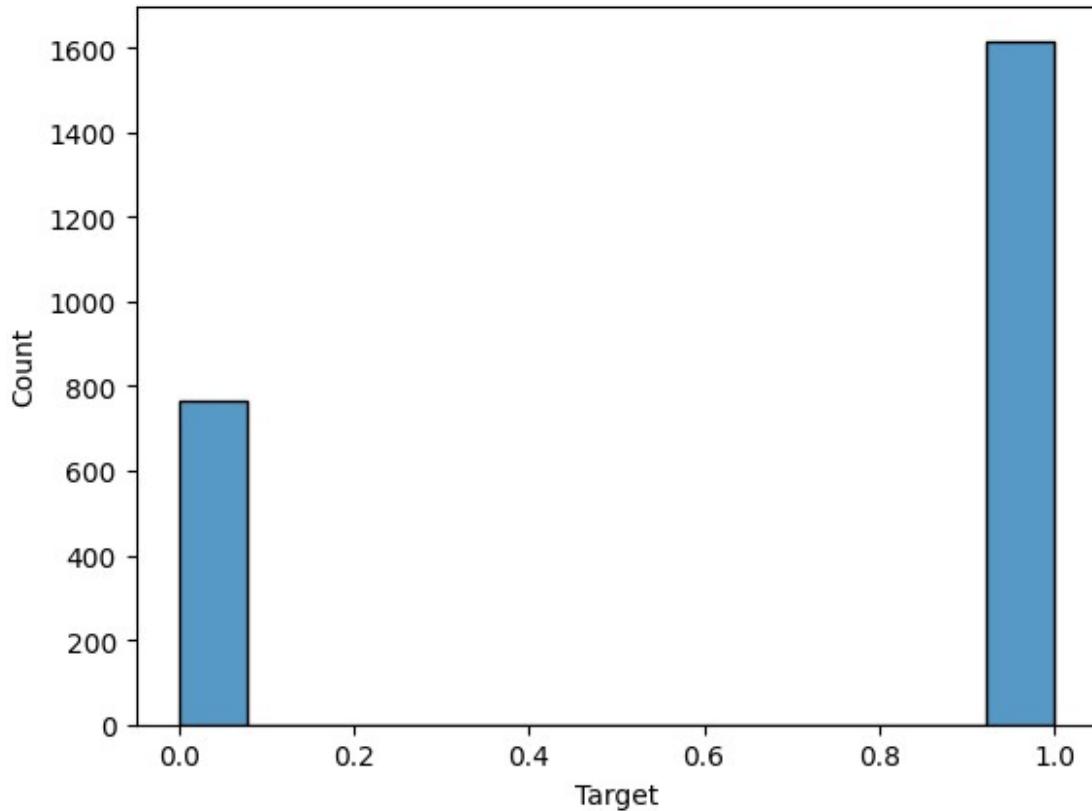
Clearly ola\_no\_out has Good Accuracy than the ola because of outlier treatment

## BALANCED OR IMBALANCED

```
ola = ola_no_out
ola.shape
(2381, 31)
ola[ "Target" ].value_counts()

Target
1    1616
0     765
Name: count, dtype: int64

sns.histplot(data = ola, x = "Target")
plt.show()
```



### Observation

ola preliminary pre processed data has 31 columns and 2381 rows. It is small dataset. So better to use Kfold validation instead of train\_val\_test split.

It is highly imbalanced Binary Class data. we can compare the performance of models before and after balancing the datasets

### Balancing using SMOTE

```
# Separating Target feature from the dataset
y = ola["Target"]
X = ola.drop("Target",axis = 1)

# Train test split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# SMOTE balancing
smt = SMOTE(random_state=42)
X_train , y_train = smt.fit_resample(X_train,y_train)

# standard Scaling
st = StandardScaler()
X_train = st.fit_transform(X_train.values)
X_test = st.transform(X_test.values)
```

```

# Logistic Regression model
model = LogisticRegression(random_state=42)
model.fit(X_train, y_train)
y_pred_train = model.predict(X_train)
y_pred_test = model.predict(X_test)

# accuracy
print("train_accuracy",accuracy_score(y_train,y_pred_train))
print("test_accuracy",accuracy_score(y_test,y_pred_test))

train_accuracy 0.8657874321179209
test_accuracy 0.8637316561844863

```

## Balancing by using class weight parameter

```

# Separating Target feature from the dataset
y = ola["Target"]
X = ola.drop("Target",axis = 1)

# Train test split
X_train, X_test, y_train, y_test = train_test_split(X, y,
test_size=0.2, random_state=42)

# standard Scaling
st = StandardScaler()
X_train = st.fit_transform(X_train.values)
X_test = st.transform(X_test.values)

# Logistic Regression model
model = LogisticRegression(random_state=42,class_weight="balanced")
model.fit(X_train, y_train)
y_pred_train = model.predict(X_train)
y_pred_test = model.predict(X_test)

# accuracy
print("train_accuracy",accuracy_score(y_train,y_pred_train))
print("test_accuracy",accuracy_score(y_test,y_pred_test))

train_accuracy 0.8797268907563025
test_accuracy 0.8867924528301887

```

## Balancing by using both SMOTE and class weight parameter

```

# Separating Target feature from the dataset
y = ola["Target"]
X = ola.drop("Target",axis = 1)

# Train test split
X_train, X_test, y_train, y_test = train_test_split(X, y,

```

```

test_size=0.2, random_state=42)

# SMOTE balancing
smt = SMOTE(random_state=42)
X_train , y_train = smt.fit_resample(X_train,y_train)

# standard Scaling
st = StandardScaler()
X_train = st.fit_transform(X_train.values)
X_test = st.transform(X_test.values)

# Logistic Regression model
model = LogisticRegression(random_state=42,class_weight="balanced")
model.fit(X_train, y_train)
y_pred_train = model.predict(X_train)
y_pred_test = model.predict(X_test)

# accuracy
print("train_accuracy",accuracy_score(y_train,y_pred_train))
print("test_accuracy",accuracy_score(y_test,y_pred_test))

train_accuracy 0.8657874321179209
test_accuracy 0.8637316561844863

```

### Observation

After observing above balancing methods and its accuracies, Clearly Dataset before balancing has better accuracy(90%) than the Dataset after balancing (88%). So We should apply next models for both before and after balancing Datasets for better comparison.

## ENCODING REQUIRED OR NOT

```

ola.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2381 entries, 0 to 2380
Data columns (total 31 columns):
 #   Column           Non-Null Count  Dtype  
--- 
 0   Age              2381 non-null    int64  
 1   City             2381 non-null    int32  
 2   Education_Level 2381 non-null    int64  
 3   Gender           2381 non-null    int64  
 4   Grade            2381 non-null    int64  
 5   Grade_raise     2381 non-null    int64  
 6   Income_raise    2381 non-null    int64  
 7   Income_range    2381 non-null    int64  
 8   Join_month      2381 non-null    int32  
 9   Join_year       2381 non-null    int32 

```

```

10 Joining_Designation      2381 non-null   int64
11 QuarterlyRating_last     2381 non-null   int64
12 QuarterlyRating_mean     2381 non-null   float64
13 QuarterlyRating_range    2381 non-null   int64
14 QuarterlyRating_std      2381 non-null   float64
15 Rating_Promotion        2381 non-null   int64
16 Reportings               2381 non-null   int64
17 TBV_raise                2381 non-null   int64
18 Target                    2381 non-null   int64
19 TBV_sum_positive          2381 non-null   int64
20 TBV_sum_negative          2381 non-null   int64
21 TBV_mean_positive         2381 non-null   float64
22 TBV_mean_negative         2381 non-null   float64
23 Income_sum_log            2381 non-null   float64
24 QuarterlyRating_sum_log   2381 non-null   float64
25 TBV_mean_positive_log    2381 non-null   float64
26 TBV_mean_negative_log    2381 non-null   float64
27 TBV_range_log             2381 non-null   float64
28 TBV_std_log               2381 non-null   float64
29 TBV_sum_positive_log     2381 non-null   float64
30 TBV_sum_negative_log     2381 non-null   float64
dtypes: float64(12), int32(3), int64(16)
memory usage: 548.9 KB

```

### Observation

Encoding is not required as all the features are integers. Can directly go for Scaling the data. Because here Categorical features like Education Level, Joining Designation, Grade, Quarterly Rating, join\_month, join\_year, Rating Promotion etc., are Ordinal Features. So Label Encoding is suitable. These are already in order in given dataset

Remember SMOTE should be applied before Scaling the data for creating meaningful intermediate points.

Apply the train test split, before doing any other pre processing steps like scaling, balancing, PCA

PCA gets effected by scale, so it is better to do PCA after Scaling

## TRAIN TEST SPLIT FOR MODELS

### Before Balancing Dataset

```

# Separating Target feature from the dataset
y = ola["Target"]
X = ola.drop("Target",axis = 1)
print("X shape",X.shape)
print("y shape",y.shape)

# Split the dataset into training and testing sets
X_train_imb, X_test_imb, y_train_imb, y_test_imb = train_test_split(X,

```

```

y, test_size=0.2, random_state=42)

print("X_train_imb shape", X_train_imb.shape)
print("X_test_imb shape", X_test_imb.shape)
print("y_train_imb shape", y_train_imb.shape)
print("y_test_imb shape", y_test_imb.shape)
print("y_train_imb value_counts", y_train_imb.value_counts())
print("y_test_imb value_counts", y_test_imb.value_counts())

X shape (2381, 30)
y shape (2381,)
X_train_imb shape (1904, 30)
X_test_imb shape (477, 30)
y_train_imb shape (1904,)
y_test_imb shape (477,)
y_train_imb value_counts Target
1    1289
0    615
Name: count, dtype: int64
y_test_imb value_counts Target
1    327
0    150
Name: count, dtype: int64

```

## After Balancing the Dataset

```

# Separating Target feature from the dataset
y = ola["Target"]
X = ola.drop("Target", axis = 1)
print("X shape", X.shape)
print("y shape", y.shape)

# Split the dataset into training and testing sets
X_train_bal, X_test_bal, y_train_bal, y_test_bal = train_test_split(X,
y, test_size=0.2, random_state=42)

# SMOTE balancing
smt = SMOTE(random_state=42)
X_train_bal, y_train_bal = smt.fit_resample(X_train_bal, y_train_bal)

print("X_train_bal shape", X_train_bal.shape)
print("X_test_bal shape", X_test_bal.shape)
print("y_train_bal shape", y_train_bal.shape)
print("y_test_bal shape", y_test_bal.shape)
print("y_train_bal value_counts", y_train_bal.value_counts())
print("y_test_bal value_counts", y_test_bal.value_counts())

X shape (2381, 30)
y shape (2381,)
X_train_bal shape (2578, 30)

```

```
X_test_bal shape (477, 30)
y_train_bal shape (2578,)
y_test_bal shape (477,)
y_train_bal value_counts Target
0    1289
1    1289
Name: count, dtype: int64
y_test_bal value_counts Target
1    327
0    150
Name: count, dtype: int64
```

### Observation

bal indicates train and test sets formed from imbalanced ola dataset

X\_test should not be balanced as it is test data

Balancing should be done before scaling for creating meaningful features

## STANDARD SCALING OF BALANCED AND IMBALANCED DATASETS

### imbalanced Dataset

```
st = StandardScaler()
X_train_imb= st.fit_transform(X_train_imb.values)
X_test_imb = st.transform(X_test_imb.values)
# No need to do scaling for target features
```

### Balanced Dataset

```
st = StandardScaler()
X_train_bal= st.fit_transform(X_train_bal.values)
X_test_bal = st.transform(X_test_bal.values)
```

## PCA VISUALISATION OF BALANCED AND IMBALANCED DATASETS

### Imbalanced Dataset

```
plt.figure(figsize=(12,5))
plt.subplot(1,2,1)

# Converting multidimensional dataset to 2D train dataset using PCA instance
pca = PCA(n_components=2,random_state=42)
visualize = pca.fit_transform(X_train_imb)
```

```

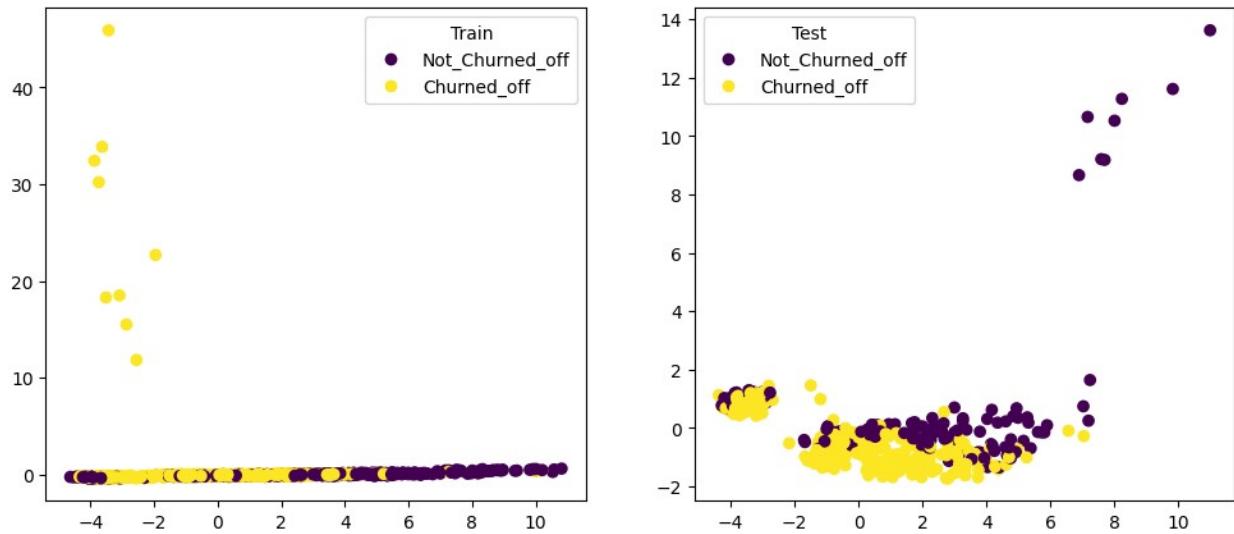
# 2D Scatter Plot of Train Dataset to visualize
scatter = plt.scatter(visualize[:,0],visualize[:,1],c =
y_train_imb.values)
plt.legend(handles = scatter.legend_elements()[0],labels =
["Not_Churned_off","Churned_off"], title = "Train")

plt.subplot(1,2,2)

# Converting multidimensional dataset to 2D test dataset using PCA
instance
pca = PCA(n_components=2,random_state=42)
visualize = pca.fit_transform(X_test_imb)

# 2D Scatter Plot of Test Dataset to visualize
scatter = plt.scatter(visualize[:,0],visualize[:,1],c =
y_test_imb.values)
plt.legend(handles = scatter.legend_elements()[0],labels =
["Not_Churned_off","Churned_off"],title = "Test")
plt.show()

```



```

plt.figure(figsize=(12,5))
plt.subplot(1,2,1)

# Converting multidimensional dataset to 2D train dataset using PCA
instance
pca = PCA(n_components=2,random_state=42)
visualize = pca.fit_transform(X_train_bal)

# 2D Scatter Plot of Train Dataset to visualize
scatter = plt.scatter(visualize[:,0],visualize[:,1],c =
y_train_bal.values)
plt.legend(handles = scatter.legend_elements()[0],labels =

```

```

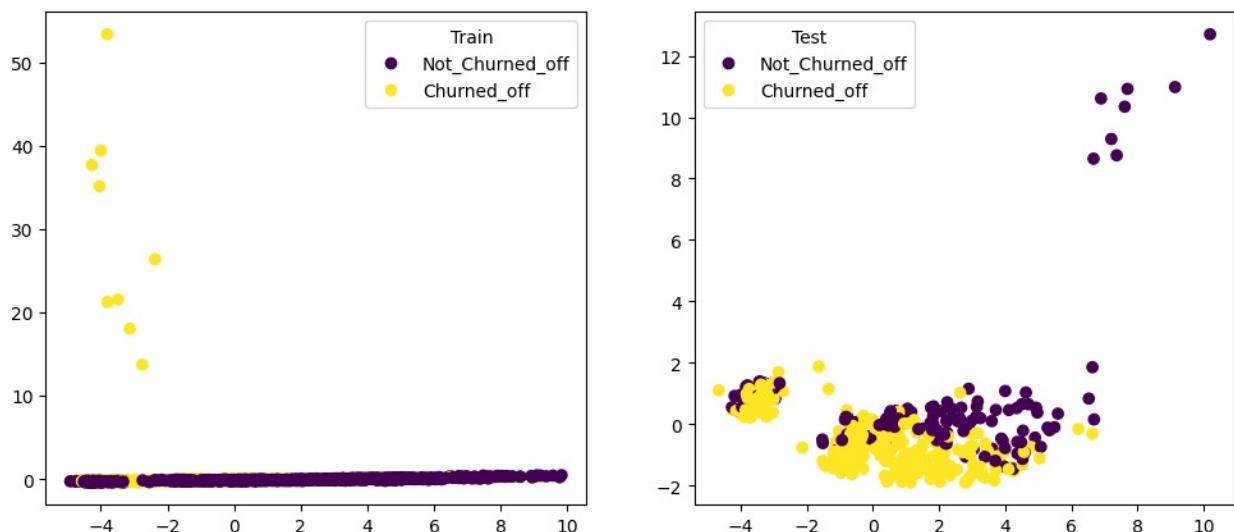
["Not_Churned_off","Churned_off"], title = "Train")

plt.subplot(1,2,2)

# Converting multidimensional dataset to 2D test dataset using PCA
instance
pca = PCA(n_components=2,random_state=42)
visualize = pca.fit_transform(X_test_bal)

# 2D Scatter Plot of Test Dataset to visualize
scatter = plt.scatter(visualize[:,0],visualize[:,1],c =
y_test_bal.values)
plt.legend(handles = scatter.legend_elements()[0],labels =
["Not_Churned_off","Churned_off"],title = "Test")
plt.show()

```



### Observation

Can clearly observe that more Dark blue points are created because of SMOTE which are Not\_churned\_off (minority Class)

**CREATING ALL THE REQUIRED FUNCTIONS TO LOG METRICS, PARAMS, ARTIFACTS, PLOTS INTO MLFLOW AND ALSO PRINT THEM**

auc\_plots function

```

def auc_plots(model, X, y):
    try:
        y_pred_prob = model.predict_proba(X)[:, 1] # Consider only
positive class

```

```

fpr, tpr, _ = roc_curve(y, y_pred_prob)
pr, re, _ = precision_recall_curve(y, y_pred_prob)
roc_auc = auc(fpr, tpr)
pr_auc = auc(re, pr)
return fpr, tpr, pr, re, roc_auc, pr_auc
except Exception as e:
    print(f"Error in auc_plots: {e}")
    return None, None, None, None, None, None

```

## plot\_learning\_curve function

```

def plot_learning_curve(model, X, y, run_name):
    try:
        train_sizes, train_scores, validation_scores =
learning_curve(model, X, y, cv=5, n_jobs=-1)
        train_mean = train_scores.mean(axis=1)
        train_std = train_scores.std(axis=1)
        validation_mean = validation_scores.mean(axis=1)
        validation_std = validation_scores.std(axis=1)

        plt.figure(figsize=(10, 6))
        plt.plot(train_sizes, train_mean, 'o-', color='blue',
label='Training score')
        plt.plot(train_sizes, validation_mean, 'o-', color='red',
label='Validation score')
        plt.fill_between(train_sizes, train_mean - train_std,
train_mean + train_std, alpha=0.1, color='blue')
        plt.fill_between(train_sizes, validation_mean -
validation_std, validation_mean + validation_std, alpha=0.1,
color='red')

        plt.xlabel('Training examples')
        plt.ylabel('Score')
        plt.title('Learning Curve')
        plt.legend(loc='best')
        plt.grid(True)

        # Save the learning curve plot
        plt.savefig(f"{run_name}_learning_curve.png")
        plt.show()
        plt.close()
        return f"{run_name}_learning_curve.png"

    except Exception as e:
        print(f"Error in plot_learning_curve: {e}")
        return None

```

## mlflow\_logging\_and\_metric\_printing function

```
def mlflow_logging_and_metric_printing(model, run_name, bal_type,
X_train, y_train, X_test, y_test, y_pred_train,
y_pred_test, hyper_tuning_score = 0, **params):
    mlflow.set_experiment("Ola_Ensemble_Learning")

    with mlflow.start_run(run_name=run_name):
        try:
            # Log parameters
            if params:
                mlflow.log_params(params)
            mlflow.log_param("bal_type", bal_type)

            # Calculate metrics
            train_metrics = {
                "Accuracy_train": accuracy_score(y_train,
y_pred_train),
                "Precision_train": precision_score(y_train,
y_pred_train),
                "Recall_train": recall_score(y_train, y_pred_train),
                "F1_score_train": f1_score(y_train, y_pred_train),
                "F2_score_train": fbeta_score(y_train, y_pred_train,
beta=2) # Emphasize recall
            }

            test_metrics = {
                "Accuracy_test": accuracy_score(y_test, y_pred_test),
                "Precision_test": precision_score(y_test,
y_pred_test),
                "Recall_test": recall_score(y_test, y_pred_test),
                "F1_score_test": f1_score(y_test, y_pred_test),
                "F2_score_test": fbeta_score(y_test, y_pred_test,
beta=2) # Emphasize recall
            }

            tuning_metrics =
{ "hyper_parameter_tuning_best_est_score":hyper_tuning_score}

            # Compute AUC metrics
            train_fpr, train_tpr, train_pr, train_re, train_roc_auc,
train_pr_auc = auc_plots(model, X_train, y_train)
            test_fpr, test_tpr, test_pr, test_re, test_roc_auc,
test_pr_auc = auc_plots(model, X_test, y_test)

            # Log AUC metrics
            if train_roc_auc is not None:
                train_metrics["Roc_auc_train"] = train_roc_auc
                mlflow.log_metric("Roc_auc_train", train_roc_auc)
```

```

if train_pr_auc is not None:
    train_metrics["Pr_auc_train"] = train_pr_auc
    mlflow.log_metric("Pr_auc_train", train_pr_auc)
if test_roc_auc is not None:
    test_metrics["Roc_auc_test"] = test_roc_auc
    mlflow.log_metric("Roc_auc_test", test_roc_auc)
if test_pr_auc is not None:
    test_metrics["Pr_auc_test"] = test_pr_auc
    mlflow.log_metric("Pr_auc_test", test_pr_auc)

# Print metrics
print("Train Metrics:")
for key, value in train_metrics.items():
    print(f"{key}: {value:.4f}")
print("\nTest Metrics:")
for key, value in test_metrics.items():
    print(f"{key}: {value:.4f}")
print("\nTuning Metrics:")
for key, value in tuning_metrics.items():
    print(f"{key}: {value:.4f}")

# Classification Reports
train_clf_report = classification_report(y_train,
y_pred_train)
test_clf_report = classification_report(y_test,
y_pred_test)

# Print classification reports
print("\nTrain Classification Report:")
print(train_clf_report)
print("\nTest Classification Report:")
print(test_clf_report)

# Log metrics
mlflow.log_metrics(train_metrics)
mlflow.log_metrics(test_metrics)
mlflow.log_metrics(tuning_metrics)

# Convert classification reports to DataFrames
train_clf_report_dict = classification_report(y_train,
y_pred_train, output_dict=True)
train_clf_report_df =
pd.DataFrame(train_clf_report_dict).transpose()
test_clf_report_dict = classification_report(y_test,
y_pred_test, output_dict=True)
test_clf_report_df =
pd.DataFrame(test_clf_report_dict).transpose()

# Save classification reports and log as artifacts

```

```

train_clf_report_df.to_csv(f"{run_name}_train_classification_report.csv")

mlflow.log_artifact(f"{run_name}_train_classification_report.csv")

test_clf_report_df.to_csv(f"{run_name}_test_classification_report.csv")
)

mlflow.log_artifact(f"{run_name}_test_classification_report.csv")

# Plot confusion matrices
fig, axes = plt.subplots(1, 2, figsize=(12, 6))

ConfusionMatrixDisplay(confusion_matrix=confusion_matrix(y_train,
y_pred_train), display_labels=["Not_churned_off",
"Churned_off"]).plot(ax=axes[0], cmap="Blues")
    axes[0].set_title('Train Confusion Matrix')

ConfusionMatrixDisplay(confusion_matrix=confusion_matrix(y_test,
y_pred_test), display_labels=["Not_churned_off",
"Churned_off"]).plot(ax=axes[1], cmap="Blues")
    axes[1].set_title('Test Confusion Matrix')

    plt.tight_layout()
    plt.savefig(f'{run_name}_confusion_matrix.png')
    mlflow.log_artifact(f'{run_name}_confusion_matrix.png')

# ROC and Precision-Recall curves
fig, axes = plt.subplots(2, 2, figsize=(12, 12))
datasets = [("Train", X_train, y_train), ("Test", X_test,
y_test)]

for i, (name, X, y) in enumerate(datasets):
    fpr, tpr, pr, re, roc_auc, pr_auc = auc_plots(model,
X, y)
        if fpr is None:
            return

        # ROC AUC Curve
        axes[i, 0].plot(fpr, tpr, color='blue', lw=2,
label=f'ROC curve (area = {roc_auc:.2f})')
        axes[i, 0].plot([0, 1], [0, 1], color='grey', lw=2,
linestyle='--')
        axes[i, 0].set_xlim([0.0, 1.0])
        axes[i, 0].set_ylim([0.0, 1.0])
        axes[i, 0].set_xlabel('False Positive Rate')
        axes[i, 0].set_ylabel('True Positive Rate')
        axes[i, 0].set_title(f'Receiver Operating

```

```

Characteristic ({name} data)')
    axes[i, 0].legend(loc='lower right')

        # Precision-Recall Curve
        axes[i, 1].plot(re, pr, color='green', lw=2,
label=f'Precision-Recall curve (area = {pr_auc:.2f})')
        axes[i, 1].set_xlabel('Recall')
        axes[i, 1].set_ylabel('Precision')
        axes[i, 1].set_title(f'Precision-Recall Curve ({name}
data)')
    axes[i, 1].legend(loc='lower left')

    plt.tight_layout()
    plt.savefig(f"{run_name}_auc_plots.png")
    mlflow.log_artifact(f"{run_name}_auc_plots.png")

    # Plot learning curves
    learning_curve_file = plot_learning_curve(model, X_train,
y_train, run_name)
    if learning_curve_file:
        mlflow.log_artifact(learning_curve_file)

    # Log the model
    mlflow.sklearn.log_model(model, f'{run_name}_model')
    print("MLFLOW Logging is completed")

except Exception as e:
    print(f"Error in mlflow_logging_and_metric_printing: {e}")

# To view the logged data, run the following command in the
terminal:
# mlflow ui

```

## Observation

`fbeta_score = fbeta_score(y_test_imb,y_pred_imb,beta=2)`

Recall is given priority by reducing False Negatives i.e, Most of the employees who are likely to churn are identified correctly

above functions are used to logged the metrics into mlflow and print the metrics in jupyter notebook.

For comparison sake, We will use this function for best estimators of Grid or Random search CV.

## Simple\_Logistic\_Regession\_on\_Imbalanced Dataset

```

# Initialize DataFrames
time_df = pd.DataFrame(columns=[ "Model", "bal_type", "Training_Time",
"Testing_Time","Tuning_Time"])

```

```

feature_importance_df = pd.DataFrame()
ola_features = ola.columns.drop("Target")

# Model details
name = "Simple_Logistic_Regression_on_Imbalanced_Dataset"
model = LogisticRegression(random_state=42, n_jobs = -1)

# Record training time
start_train_time = time.time()
model.fit(X_train_imb, y_train_imb)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_imb_train = model.predict(X_train_imb)
y_pred_imb_test = model.predict(X_test_imb)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")

Model: Simple_Logistic_Regression_on_Imbalanced_Dataset
Training Time: 2.7384 seconds
Testing Time: 0.0023 seconds

```

## log\_time\_and\_feature\_importance\_df function

```

def log_time_and_feature_importances_df(time_df,
                                         feature_importance_df, name, training_time, testing_time, model,
                                         ola_features, bal_type, tuning_time=0):
    # Store the times in the DataFrame using pd.concat
    time_df = pd.concat([time_df, pd.DataFrame({
        "Model": [name],
        "bal_type": [bal_type],
        "Training_Time": [training_time],
        "Testing_Time": [testing_time],
        "Tuning_Time": [tuning_time]
    })], ignore_index=True)

    # Feature Importances
    if hasattr(model, "coef_"):
        # For linear models like Logistic Regression or Linear SVC

```

```

feature_importances = model.coef_.flatten()

elif hasattr(model, "feature_importances_"):
    # For tree-based models like Decision Trees, RandomForest,
    GradientBoosting, etc.
    feature_importances = model.feature_importances_

else:
    # If the model does not have feature importances, we skip
    logging for this model
    feature_importances = None

if feature_importances is not None:
    # Create a DataFrame with feature importances
    importance_df = pd.DataFrame(feature_importances,
index=ola_features, columns=[name])
    feature_importance_df = pd.concat([feature_importance_df,
importance_df], axis=1)

    return time_df, feature_importance_df

def feature_importances_df_new(feature_importance_df, name, model,
ola_features, bal_type):
    # Feature Importances
    if hasattr(model, "coef_"):
        # For linear models like Logistic Regression or Linear SVC
        feature_importances = model.coef_.flatten()

    elif hasattr(model, "feature_importances_"):
        # For tree-based models like Decision Trees, RandomForest,
        GradientBoosting, etc.
        feature_importances = model.feature_importances_

    else:
        # If the model does not have feature importances, we skip
        logging for this model
        feature_importances = None

    if feature_importances is not None:
        # Create a DataFrame with feature importances
        importance_df = pd.DataFrame(feature_importances,
index=ola_features, columns=[name])
        feature_importance_df = pd.concat([feature_importance_df,
importance_df], axis=1)

    return feature_importance_df

time_df, feature_importance_df =
log_time_and_feature_importances_df(time_df, feature_importance_df, name
, training_time, testing_time, model, ola_features, "Imbalanced")

```

```

# if mlflow.active_run() is not None:
#     mlflow.end_run()

params = {"random_state":42,"n_jobs":-1}
print(name)
mlflow_logging_and_metric_printing(model,name,"Imbalanced",X_train_imb
,y_train_imb,X_test_imb,y_test_imb,y_pred_imb_train,y_pred_imb_test,**
params)

Simple_Logistic_Regression_on_Imbalanced_Dataset
Train Metrics:
Accuracy_train: 0.8866
Precision_train: 0.8942
Recall_train: 0.9441
F1_score_train: 0.9185
F2_score_train: 0.9337
Roc_auc_train: 0.9460
Pr_auc_train: 0.9723

Test Metrics:
Accuracy_test: 0.9099
Precision_test: 0.9104
Recall_test: 0.9633
F1_score_test: 0.9361
F2_score_test: 0.9522
Roc_auc_test: 0.9427
Pr_auc_test: 0.9598

Tuning Metrics:
hyper_parameter_tuning_best_est_score: 0.0000

Train Classification Report:
precision    recall   f1-score   support
      0       0.87      0.77      0.81      615
      1       0.89      0.94      0.92     1289
   accuracy          0.89          0.89          0.89      1904
   macro avg       0.88      0.85      0.87      1904
weighted avg       0.89      0.89      0.88      1904

Test Classification Report:
precision    recall   f1-score   support
      0       0.91      0.79      0.85      150
      1       0.91      0.96      0.94      327
   accuracy          0.91          0.88          0.91      477
   macro avg       0.91      0.88      0.89      477

```

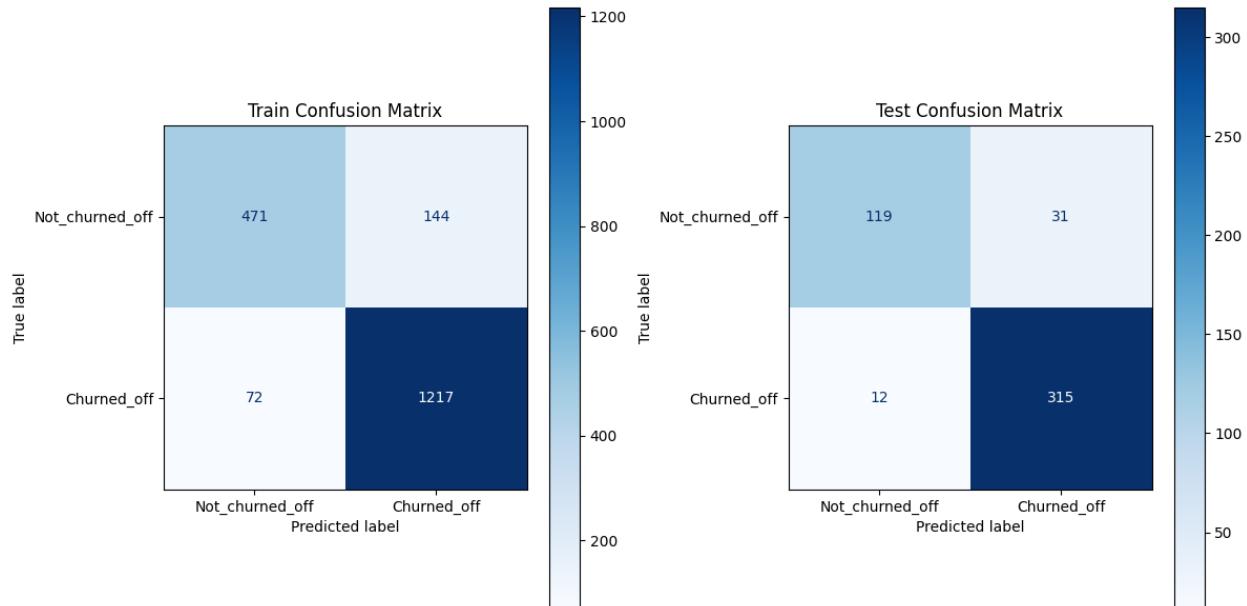
**weighted avg**

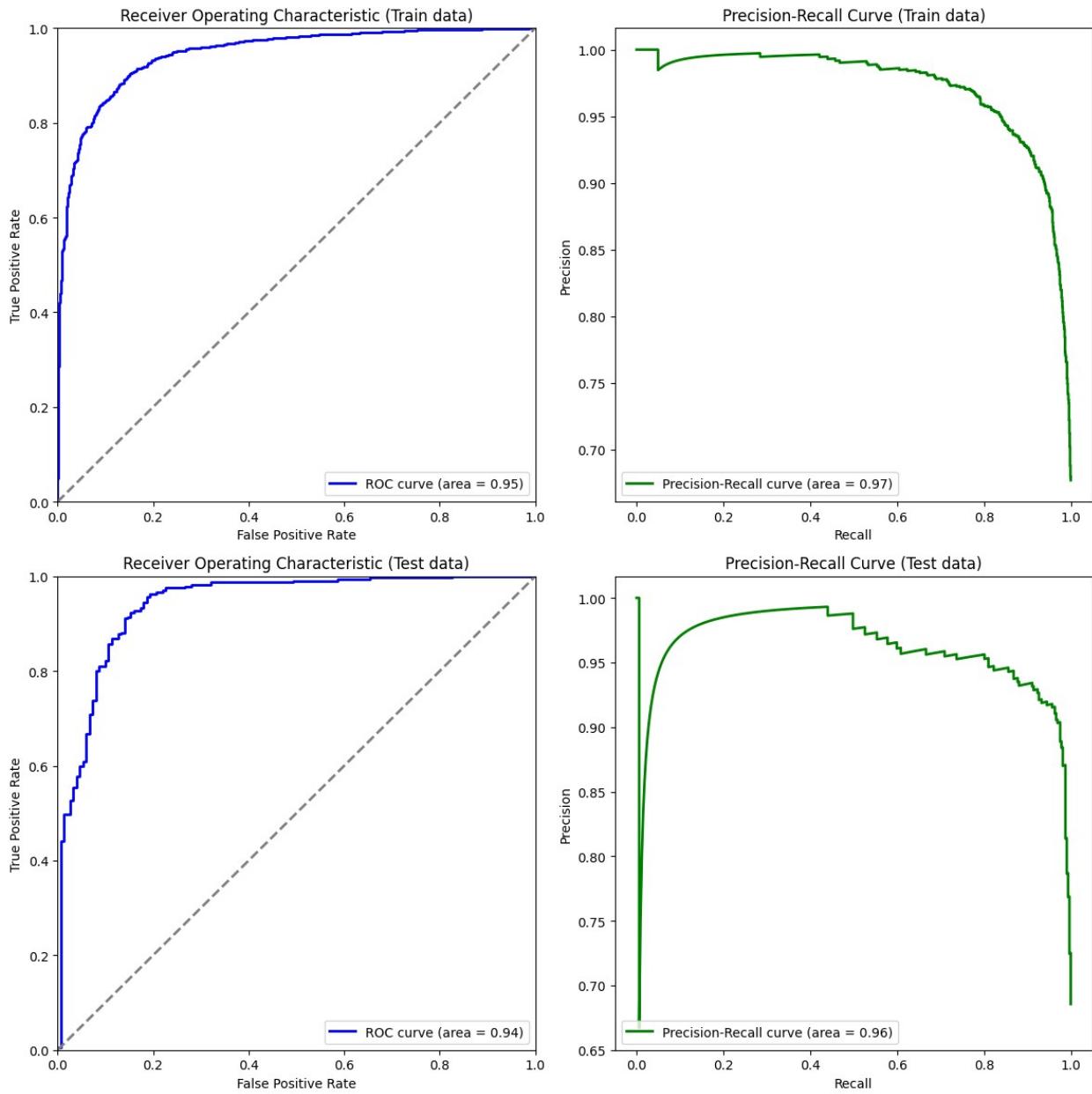
**0.91**

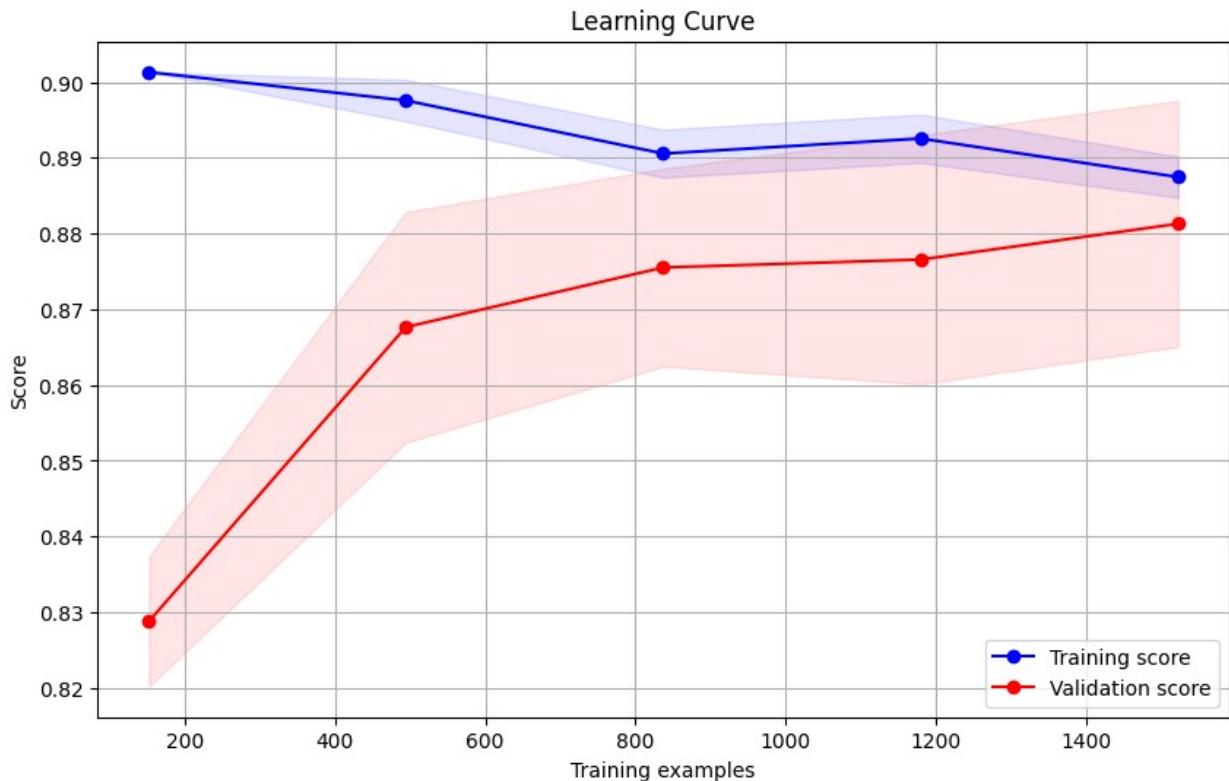
**0.91**

**0.91**

**477**







MLFL0W Logging is completed

### Simple\_Logistic\_Regresssion\_on\_Balanced Dataset

```
# Model details
name = "Simple_Logistic_Regresssion_on_balanced_Dataset"
model = LogisticRegression(random_state=42, n_jobs = -1)
bal_type = "Balanced"

# Record training time
start_train_time = time.time()
model.fit(X_train_bal, y_train_bal)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_bal_train = model.predict(X_train_bal)
y_pred_bal_test = model.predict(X_test_bal)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time
```

```

# Print model name and times
print(f"Model: {name}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")

# log time and feature importances into df
time_df,feature_importance_df =
log_time_and_feature_importances_df(time_df,feature_importance_df,name
,training_time,testing_time,model,ola_features,bal_type)

Model: Simple_Logistic_Regresssion_on_balanced_Dataset
Training Time: 0.0362 seconds
Testing Time: 0.0000 seconds

params = {"random_state":42,"n_jobs":-1}
print(name)
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_bal,y_t
rain_bal,X_test_bal,y_test_bal,y_pred_bal_train,y_pred_bal_test,**para
ms)

Simple_Logistic_Regresssion_on_balanced_Dataset
Train Metrics:
Accuracy_train: 0.8658
Precision_train: 0.8527
Recall_train: 0.8844
F1_score_train: 0.8682
F2_score_train: 0.8779
Roc_auc_train: 0.9424
Pr_auc_train: 0.9463

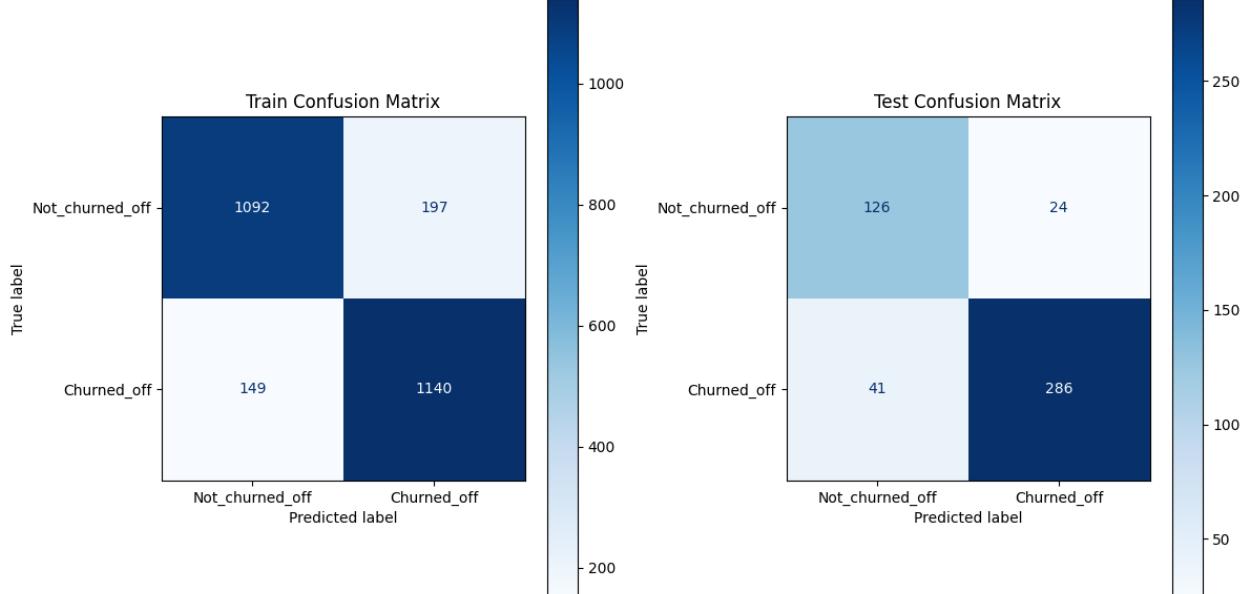
Test Metrics:
Accuracy_test: 0.8637
Precision_test: 0.9226
Recall_test: 0.8746
F1_score_test: 0.8980
F2_score_test: 0.8838
Roc_auc_test: 0.9292
Pr_auc_test: 0.9552

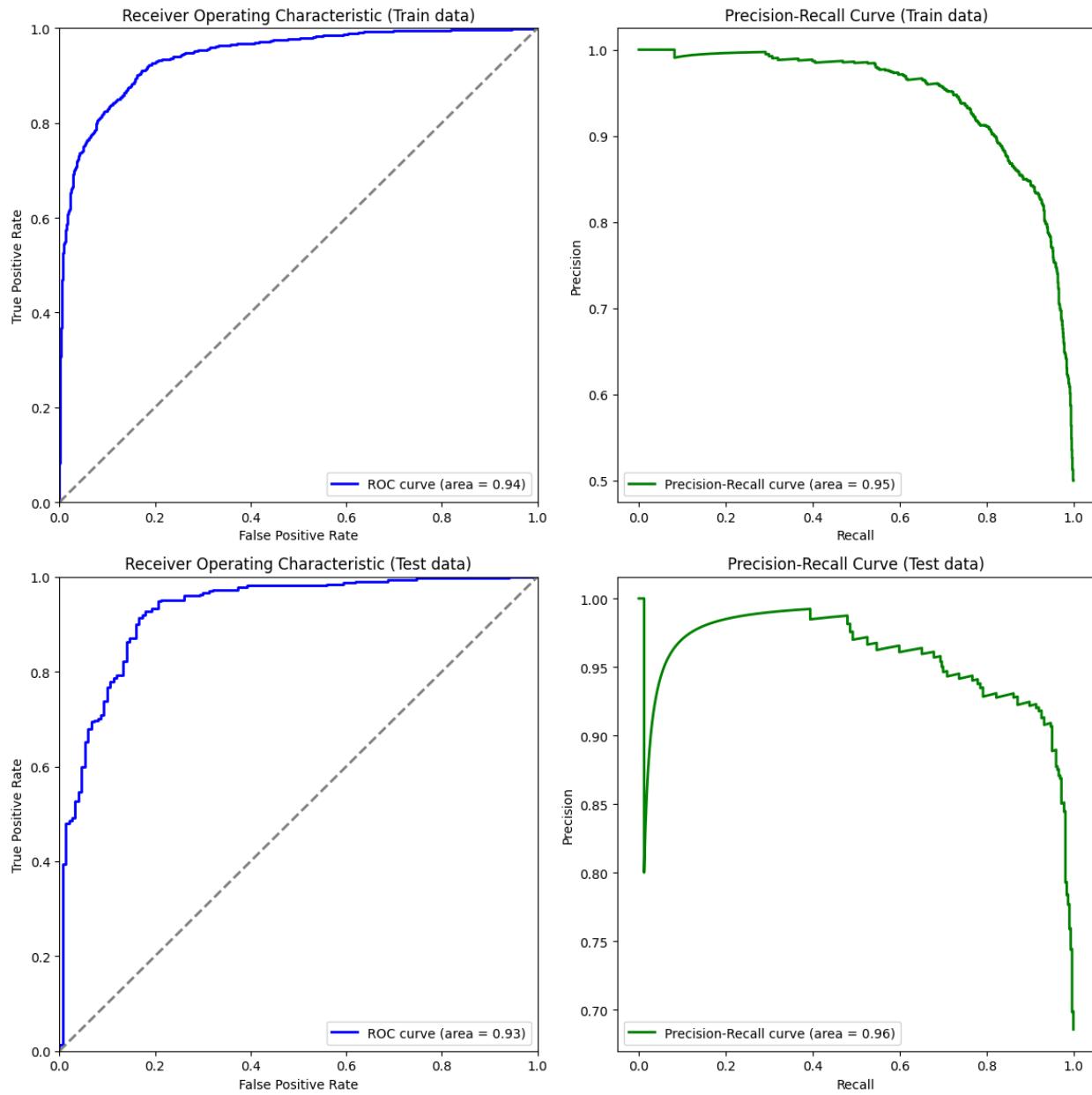
Tuning Metrics:
hyper_parameter_tuning_best_est_score: 0.0000

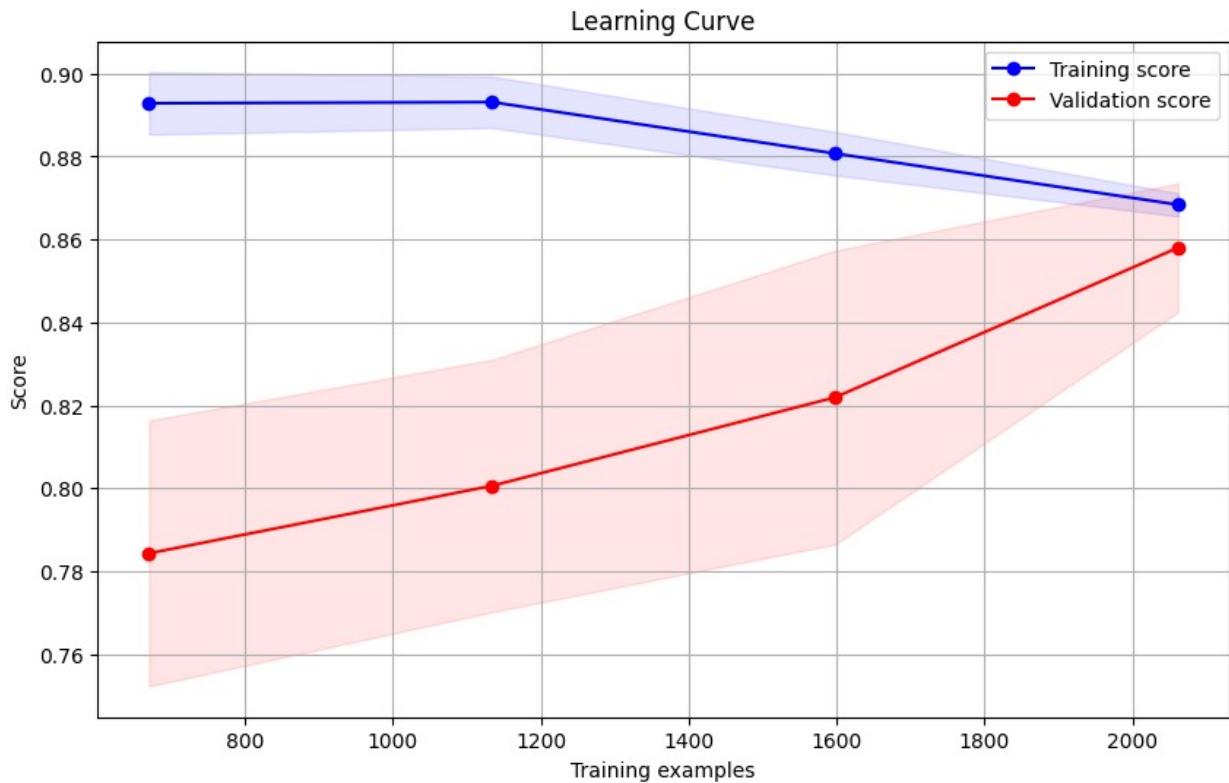
Train Classification Report:
precision    recall   f1-score   support
      0       0.88      0.85      0.86     1289
      1       0.85      0.88      0.87     1289
      accuracy           0.87      0.87      0.87     2578
      macro avg        0.87      0.87      0.87     2578
      weighted avg     0.87      0.87      0.87     2578

```

Test Classification Report:					
	precision	recall	f1-score	support	
0	0.75	0.84	0.79	150	
1	0.92	0.87	0.90	327	
accuracy			0.86	477	
macro avg	0.84	0.86	0.85	477	
weighted avg	0.87	0.86	0.87	477	







MLFL0W Logging is completed

### all\_logged\_metrics function

```

def all_logged_metrics():
    # Set the experiment name
    experiment_name = "Ola_Ensemble_Learning"

    # Get the experiment details
    experiment = mlflow.get_experiment_by_name(experiment_name)
    experiment_id = experiment.experiment_id

    # Retrieve all runs from the experiment
    runs_df = mlflow.search_runs(experiment_ids=[experiment_id])

    # Extract metrics columns
    metrics_columns = [col for col in runs_df.columns if
col.startswith("metrics.")]
    metrics_df = runs_df[metrics_columns]

    # Add run_name as a column
    metrics_df['run_name'] = runs_df['tags.mlflow.runName']
    metrics_df["bal_type"] = runs_df["params.bal_type"]

    # Combine all params into a dictionary
    params_columns = [col for col in runs_df.columns if

```

```

col.startswith("params."))

    metrics_df["params_dict"] = runs_df[params_columns].apply(lambda
row: row.dropna().to_dict(), axis=1)

    # Sort remaining columns alphabetically
    sorted_columns = sorted(metrics_columns)

    # Rearrange columns: first column is 'run_name', followed by
'bal_type', 'params_dict', and then sorted metric columns
    ordered_columns = ['run_name', "bal_type", "params_dict"] +
sorted_columns
    metrics_df = metrics_df[ordered_columns]

    # If you want to view it in a more readable format
    return metrics_df

```

Visualising all\_logged\_metrics, time\_df and feature\_imporatance df

All\_logged\_metrics\_plots

```

# Example usage
pd.set_option('display.max_colwidth', None)  # Show full content in
cells
all_logged_metrics_df = all_logged_metrics()
all_logged_metrics_df.head()

                    run_name      bal_type \
0  Simple_Logistic_Regresssion_on_balanced_Dataset  Balanced
1  Simple_Logistic_Regression_on_Imbalanced_Dataset  Imbalanced
2  Two_Level_Cascading_Classifier_on_Balanced_Dataset  Balanced
3  Two_Level_Cascading_Classifier_on_Imbalanced_Dataset  Imbalanced
4  Tuned_Stacking_Classifier_on_Balanced_Dataset  Balanced

params_dict \
0  {'params.bal_type': 'Balanced', 'params.random_state': '42', 'params.n_jobs': '-1'}
1  {'params.bal_type': 'Imbalanced', 'params.random_state': '42', 'params.n_jobs': '-1'}
2  {'params.bal_type': 'Balanced'}
3  {'params.bal_type': 'Imbalanced'}
4  {'params.bal_type': 'Balanced', 'params.passthrough': 'True', 'params.stack_method': 'predict'}

    metrics.Accuracy_test  metrics.Accuracy_train
metrics.F1_score_test \
0                  0.863732                0.865787
0.897959

```

1	0.909853	0.886555	
0.936107			
2	0.899371	0.999224	
0.926154			
3	0.911950	0.957458	
0.936747			
4	0.899371	0.999224	
0.926154			
	metrics.F1_score_train	metrics.F2_score_test	
	metrics.F2_score_train		
0	0.868241	0.883807	
0.877868			
1	0.918491	0.952237	
0.933712			
2	0.999224	0.922747	
0.999224			
3	0.968906	0.945289	
0.974969			
4	0.999224	0.922747	
0.999224			
	metrics.Pr_auc_test	metrics.Pr_auc_train	
	metrics.Precision_test		
0	0.955210	0.946333	
0.922581			
1	0.959764	0.972315	
0.910405			
2	NaN	NaN	
0.931889			
3	NaN	NaN	
0.922849			
4	0.963908	0.999996	
0.931889			
	metrics.Precision_train	metrics.Recall_test	metrics.Recall_train
\			
0	0.852655	0.874618	0.884407
0.884407			
1	0.894195	0.963303	0.944143
0.944143			
2	0.999224	0.920489	0.999224
0.999224			
3	0.958967	0.951070	0.979054
0.979054			
4	0.999224	0.920489	0.999224
0.999224			
	metrics.Roc_auc_test	metrics.Roc_auc_train	
0	0.929154	0.942393	
0.942393			
1	0.942650	0.945990	
0.945990			

```

2           NaN           NaN
3           NaN           NaN
4      0.938899      0.999996

    metrics.hyper_parameter_tuning_best_est_score
0                      0.00000
1                      0.00000
2                     0.92282
3                     0.92282
4                     0.92282

pd.reset_option('display.max_colwidth')

import matplotlib.pyplot as plt
import seaborn as sns

def all_logged_metrics_df_plots(df):
    # Melt the DataFrame to make it easier to plot
    metrics_columns = df.columns[3:] # Select only metric columns
    df_melted = df.melt(id_vars=['run_name', 'bal_type'],
                         value_vars=metrics_columns,
                         var_name='Metric',
                         value_name='Value')

    # Create subplots with 8 rows and 2 columns
    n_rows = 8
    n_cols = 2
    fig, axes = plt.subplots(n_rows, n_cols, figsize=(15, 30))
    axes = axes.flatten() # Flatten the axes array for easy iteration

    # Plot each metric in a separate subplot
    for i, metric in enumerate(metrics_columns):
        sns.barplot(x='bal_type', y='Value', hue='run_name',
                    data=df_melted[df_melted['Metric'] == metric], ax=axes[i])
        axes[i].set_title(metric)
        axes[i].set_xlabel('')
        axes[i].set_ylabel('Value')
        axes[i].legend_.remove() # Remove legend from individual plots

    # Remove any unused subplots
    for j in range(len(metrics_columns), len(axes)):
        fig.delaxes(axes[j])

    # Add a single legend for all subplots
    handles, labels = axes[0].get_legend_handles_labels()
    fig.legend(handles, labels, loc='upper center', ncol=3,
               bbox_to_anchor=(0.5, 1.03))

    # Adjust layout

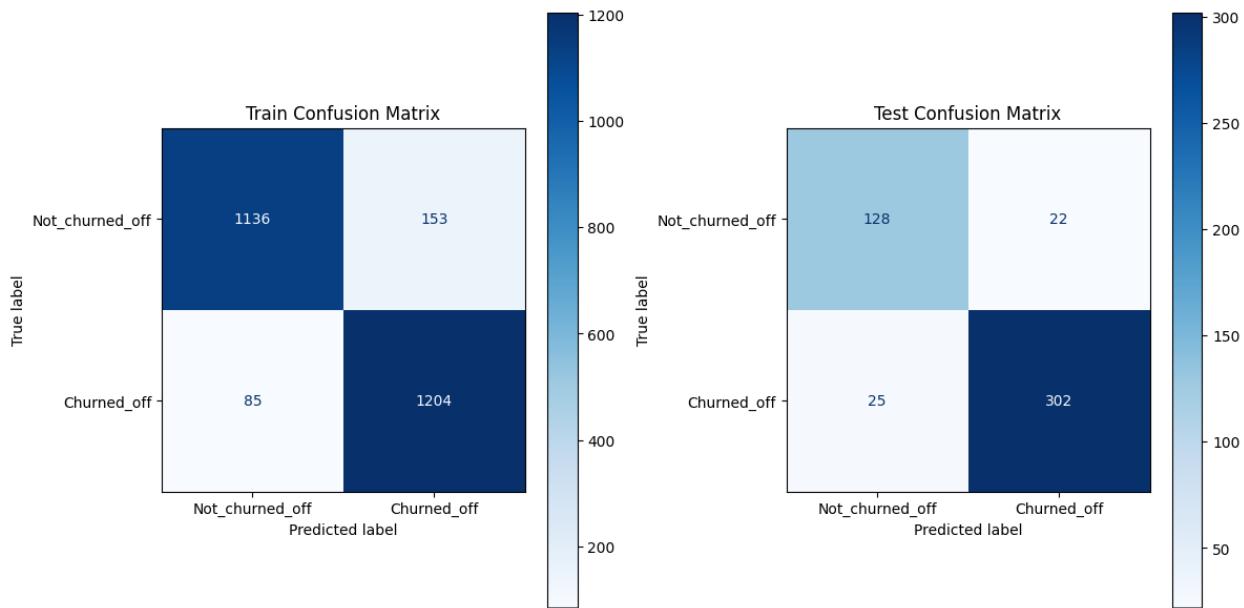
```

```

plt.tight_layout()
plt.subplots_adjust(top=0.95) # Adjust top to make space for the
global legend
plt.show()

# Example usage with your DataFrame
# all_logged_metrics_df_plots(all_logged_metrics_df)

```



time\_df\_plots

time\_df

```

Model      bal_type \
0 Simple_Logistic_Regression_on_Imbalanced_Dataset  Imbalanced
1 Simple_Logistic_Regresssion_on_balanced_Dataset   Balanced

Training_Time  Testing_Time Tuning_Time
0            2.738420     0.002282      0
1            0.036155     0.000000      0

def time_df_plots(time_df):
    # Create subplots for Training Time and Testing Time
    fig, axes = plt.subplots(1, 3, figsize=(25, 10))

    # Plot Training Time
    sns.barplot(x='Model', y='Training_Time', hue='bal_type',
    data=time_df, ax=axes[0])
    axes[0].set_title('Training Time by Model and Dataset Balance')
    axes[0].set_xticklabels(axes[0].get_xticklabels(), rotation=45,
    ha='right')
    axes[0].set_xlabel('Model')

```

```

axes[0].set_ylabel('Training Time (seconds)')

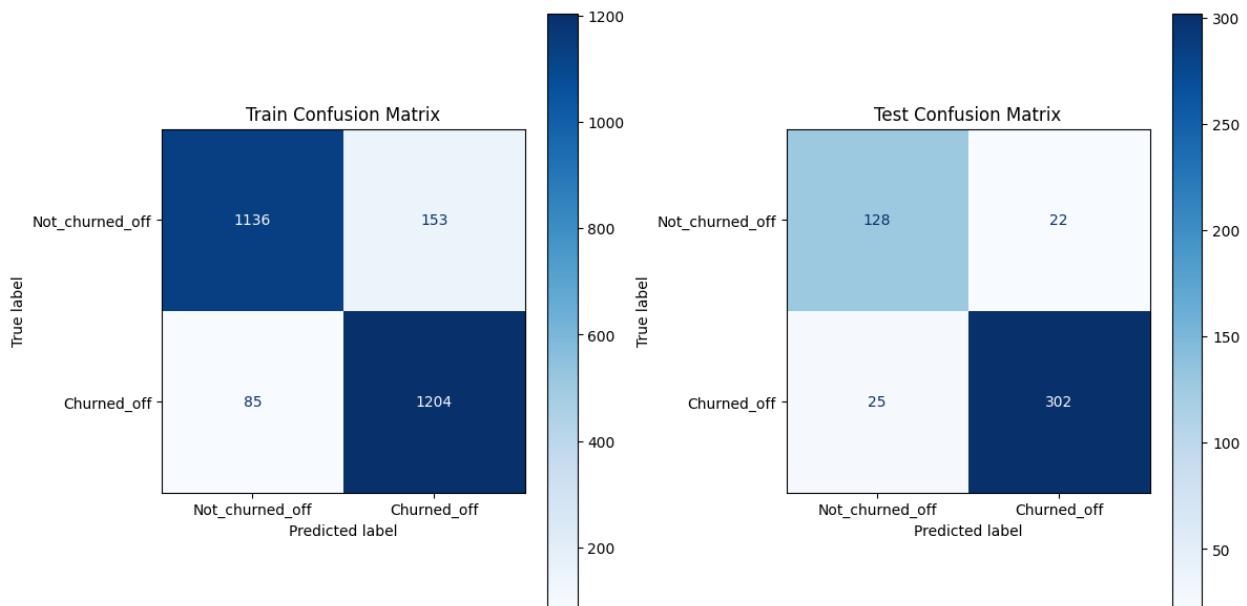
# Plot Testing Time
sns.barplot(x='Model', y='Testing_Time', hue='bal_type',
data=time_df, ax=axes[1])
axes[1].set_title('Testing Time by Model and Dataset Balance')
axes[1].set_xticklabels(axes[1].get_xticklabels(), rotation=45,
ha='right')
axes[1].set_xlabel('Model')
axes[1].set_ylabel('Testing Time (seconds)')

# Plot Tuning Time
sns.barplot(x='Model', y='Tuning_Time', hue='bal_type',
data=time_df, ax=axes[2])
axes[2].set_title('Testing Time by Model and Dataset Balance')
axes[2].set_xticklabels(axes[2].get_xticklabels(), rotation=45,
ha='right')
axes[2].set_xlabel('Model')
axes[2].set_ylabel('Tuning Time (seconds)')

# Adjust layout
plt.tight_layout()

# Show the plot
plt.show()
# time_df_plots(time_df)

```



feature\_importance\_df\_plots

feature\_importance\_df

```
Simple_Logistic_Regression_on_Imbalanced_Dataset \n
Age\n
0.109896\n
City\n
0.069281\n
Education_Level\n
0.037378\n
Gender\n
0.047752\n
Grade\n
0.778416\n
Grade_raise\n
0.200566\n
Income_raise\n
0.200566\n
Income_range\n
0.722095\n
Join_month\n
0.365371\n
Join_year\n
2.013868\n
Joining_Designation\n
0.130488\n
QuarterlyRating_last\n
0.720011\n
QuarterlyRating_mean\n
1.805258\n
QuarterlyRating_range\n
0.145085\n
QuarterlyRating_std\n
0.067082\n
Rating_Promotion\n
0.148453\n
Reportings\n
3.542816\n
TBV_raise\n
0.966004\n
TBV_sum_positive\n
2.495274\n
TBV_sum_negative\n
0.039156\n
TBV_mean_positive\n
1.608817\n
TBV_mean_negative\n
0.059553\n
Income_sum_log\n
0.326551\n
QuarterlyRating_sum_log\n
3.744988
```

```
TBV_mean_positive_log  
0.372347  
TBV_mean_negative_log  
0.145310  
TBV_range_log  
0.114218  
TBV_std_log  
0.306035  
TBV_sum_positive_log  
0.113332  
TBV_sum_negative_log  
0.143163
```

#### Simple\_Logistic\_Regesssion\_on\_balanced\_Dataset

```
Age  
0.095305  
City  
0.098565  
Education_Level  
0.256508  
Gender  
0.335353  
Grade  
0.293402  
Grade_raise  
0.095718  
Income_raise  
0.095718  
Income_range  
0.239048  
Join_month  
0.312485  
Join_year  
2.256950  
Joining_Designation  
0.143158  
QuarterlyRating_last  
0.178099  
QuarterlyRating_mean  
2.392407  
QuarterlyRating_range  
1.534083  
QuarterlyRating_std  
1.242552  
Rating_Promotion  
0.440989  
Reportings  
3.473494  
TBV_raise
```

```
1.014269
TBV_sum_positive
3.752669
TBV_sum_negative
0.052252
TBV_mean_positive
1.715850
TBV_mean_negative
0.065882
Income_sum_log
0.345360
QuarterlyRating_sum_log
4.377599
TBV_mean_positive_log
0.046581
TBV_mean_negative_log
0.128259
TBV_range_log
0.083093
TBV_std_log
0.225809
TBV_sum_positive_log
0.090800
TBV_sum_negative_log
0.127302

import matplotlib.pyplot as plt
import seaborn as sns

def feature_importance_plots(feature_importance_df):
    # Reset the index to get the features as a column
    feature_importance_df_reset = feature_importance_df.reset_index()

    # Melt the DataFrame to have a long-form DataFrame suitable for
    # seaborn
    feature_importance_melted =
    feature_importance_df_reset.melt(id_vars='index',
    var_name='Model',
    value_name='Importance')

    # Create a seaborn horizontal barplot
    plt.figure(figsize=(10, 30)) # Adjust the figure size to be
    # taller
    sns.barplot(x='Importance', y='index', hue='Model',
    data=feature_importance_melted, orient='h')

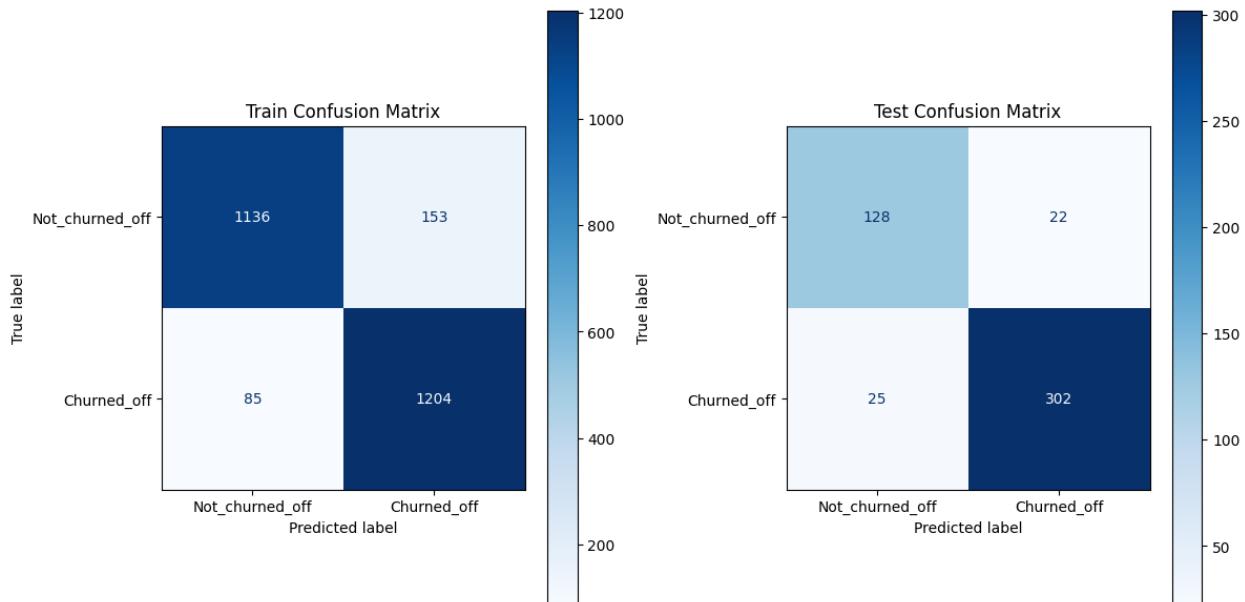
    # Set title and labels
    plt.title('Feature Importance Comparison')
```

```

plt.xlabel('Importance Value')
plt.ylabel('Features')
plt.legend(loc = "lower right")
# Display the plot
plt.tight_layout()
plt.show()

# Example usage with your feature_importance_df
# feature_importance_plots(feature_importance_df)

```



## Observation

We have created predefined functions for our easeness to log and print the metrics of the models

Basically we have to do Grid or Random search CV, Then find the best estimator. On that best estimator, we can apply `mlflow_logging_and_metric_printing` function, `log_time_and_feature_importances_df` function

if we want to visualize all metrics, time and feature importances for all the models, then use `all_logged_metrics` function, get the df from that function and use it in `All_logged_metrics_plots` function. We can use '`time_df_plots`' and '`feature_importance_df_plots`' functions also.

plots are modified at the end for better visualization

# CHAPTER 5: MODEL BUILDING AND ENSEMBLE LEARNING

## LOGISTIC REGRESSION MODEL

```
y_train_bal.value_counts()  
  
Target  
0    1289  
1    1289  
Name: count, dtype: int64
```

### Imbalanced Dataset

#### Hyper parameter Tuning

```
# Define the parameter grid for RandomizedSearchCV  
start_tune_time = time.time()  
param_dist = {  
    'penalty': ['l1', 'l2', 'elasticnet', 'none'],  
    'C': stats.uniform(loc=0.01, scale=5000-0.01), # Uniform  
    distribution from 0.01 to 5000  
    'solver': ['saga'], # saga solver supports all penalties  
    'class_weight': ['balanced']  
}  
  
# Initialize the Logistic Regression model  
logReg = LogisticRegression(max_iter=10000, random_state=42)  
  
# Setup RandomizedSearchCV  
random_search = RandomizedSearchCV(  
    estimator=logReg,  
    param_distributions=param_dist,  
    n_iter=200, # Number of parameter settings to try  
    cv=5, # Number of folds in cross-validation  
    verbose=1,  
    random_state=42,  
    n_jobs=-1  
)  
  
# Fit RandomizedSearchCV  
random_search.fit(X_train_imb, y_train_imb)  
  
# Best model and hyperparameters  
print("Best parameters found:", random_search.best_params_)  
print("Best score:", random_search.best_score_)  
tuning_score = random_search.best_score_  
end_tune_time = time.time()
```

```

tuning_time = end_tune_time - start_tune_time
print("Tuning_time", tuning_time)

Fitting 5 folds for each of 200 candidates, totalling 1000 fits
Best parameters found: {'C': 1872.706848835624, 'class_weight':
'balanced', 'penalty': 'l1', 'solver': 'saga'}
Best score: 0.87761983699406
Tuning_time 101.47512793540955

```

Logging Best Logistic Regression Model into MLFLOW

```

# Model details
name = "Tuned_Logistic_Regression_on_Imbalanced_Dataset"
bal_type = "Imbalanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time
start_train_time = time.time()
model.fit(X_train_imb, y_train_imb)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_imb_train = model.predict(X_train_imb)
y_pred_imb_test = model.predict(X_test_imb)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df,
mlflow_logging_and_metric_printing
time_df,feature_importance_df =
log_time_and_feature_importances_df(time_df,feature_importance_df,name
,training_time,testing_time,model,ola_features,bal_type,tuning_time)
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_imb,y_t
rain_imb,X_test_imb,y_test_imb,y_pred_imb_train,y_pred_imb_test,tuning
_score,**params)

```

```

Model: Tuned_Logistic_Regression_on_Imbalanced_Dataset
params: {'C': 1872.706848835624, 'class_weight': 'balanced',
'penalty': 'l1', 'solver': 'saga'}
Training Time: 1.0464 seconds
Testing Time: 0.0012 seconds
Tuning Time: 101.4751 seconds
Train Metrics:
Accuracy_train: 0.8797
Precision_train: 0.9337
Recall_train: 0.8852
F1_score_train: 0.9088
F2_score_train: 0.8945
Roc_auc_train: 0.9482
Pr_auc_train: 0.9738

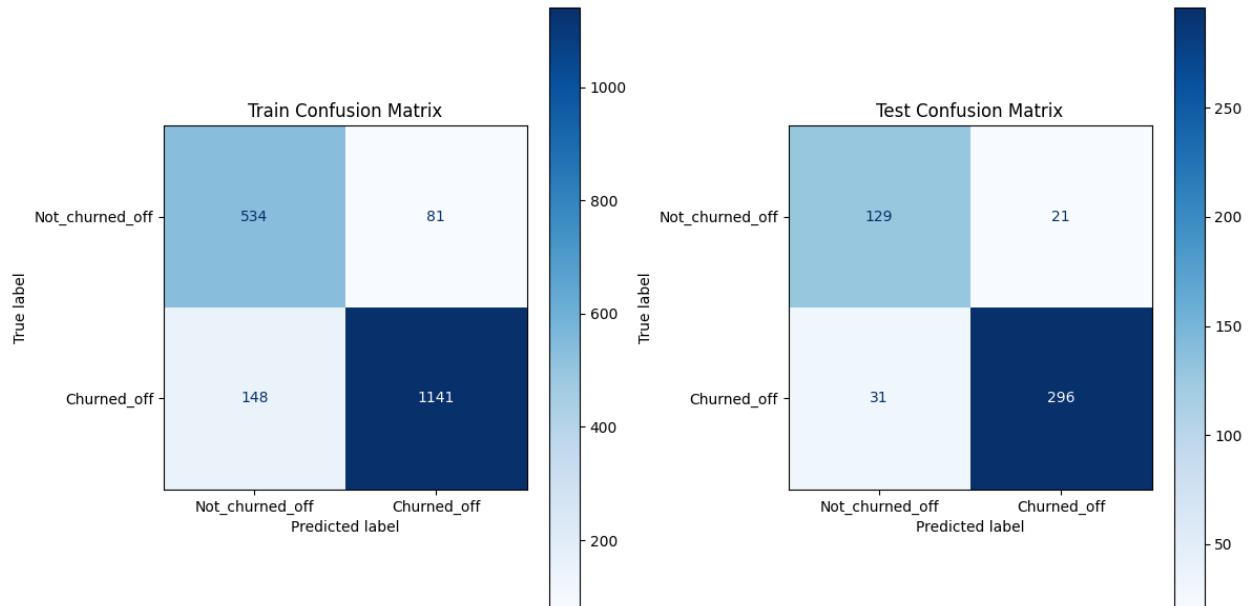
Test Metrics:
Accuracy_test: 0.8910
Precision_test: 0.9338
Recall_test: 0.9052
F1_score_test: 0.9193
F2_score_test: 0.9108
Roc_auc_test: 0.9421
Pr_auc_test: 0.9598

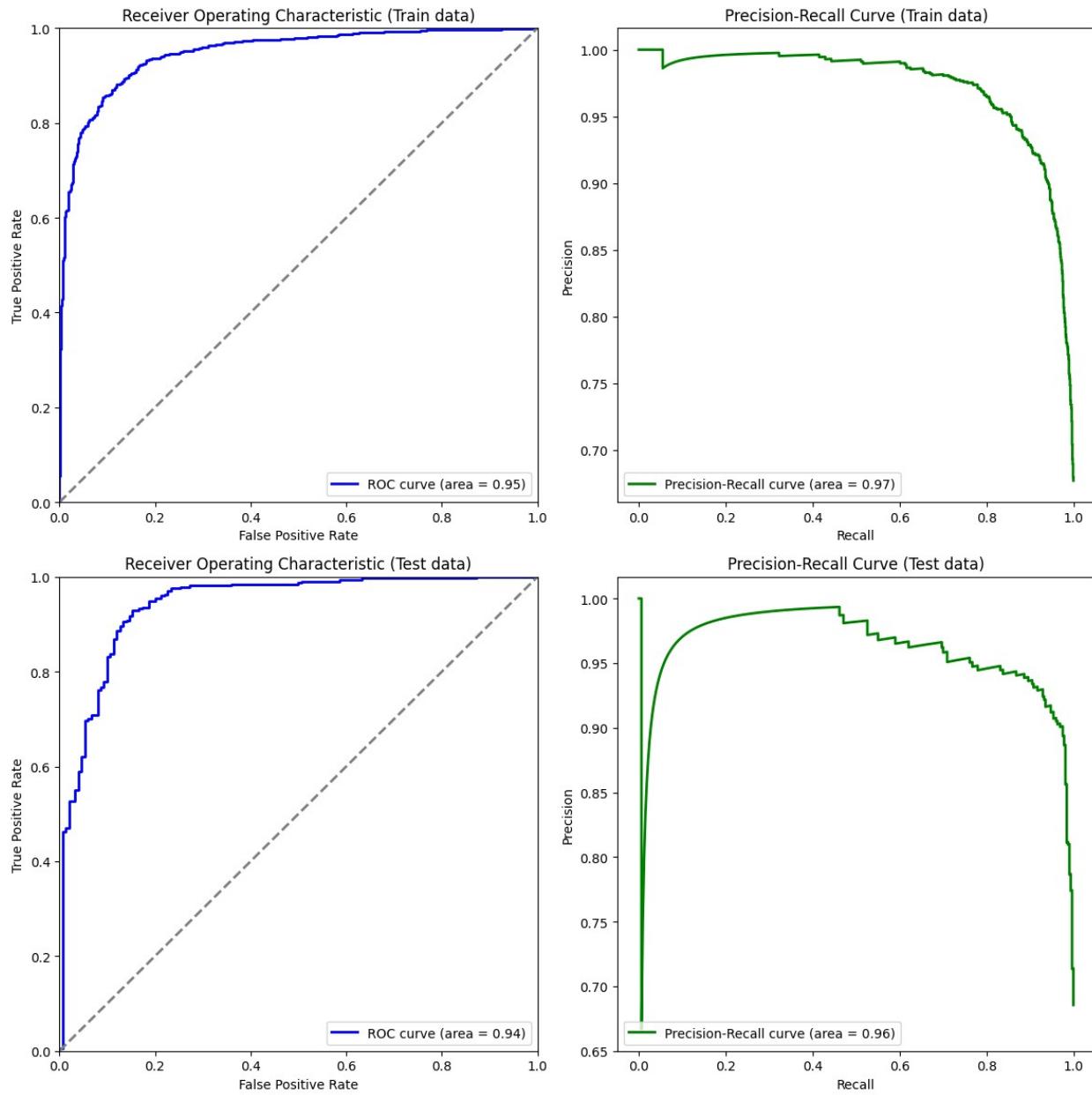
Tuning Metrics:
hyper_parameter_tuning_best_est_score: 0.8776

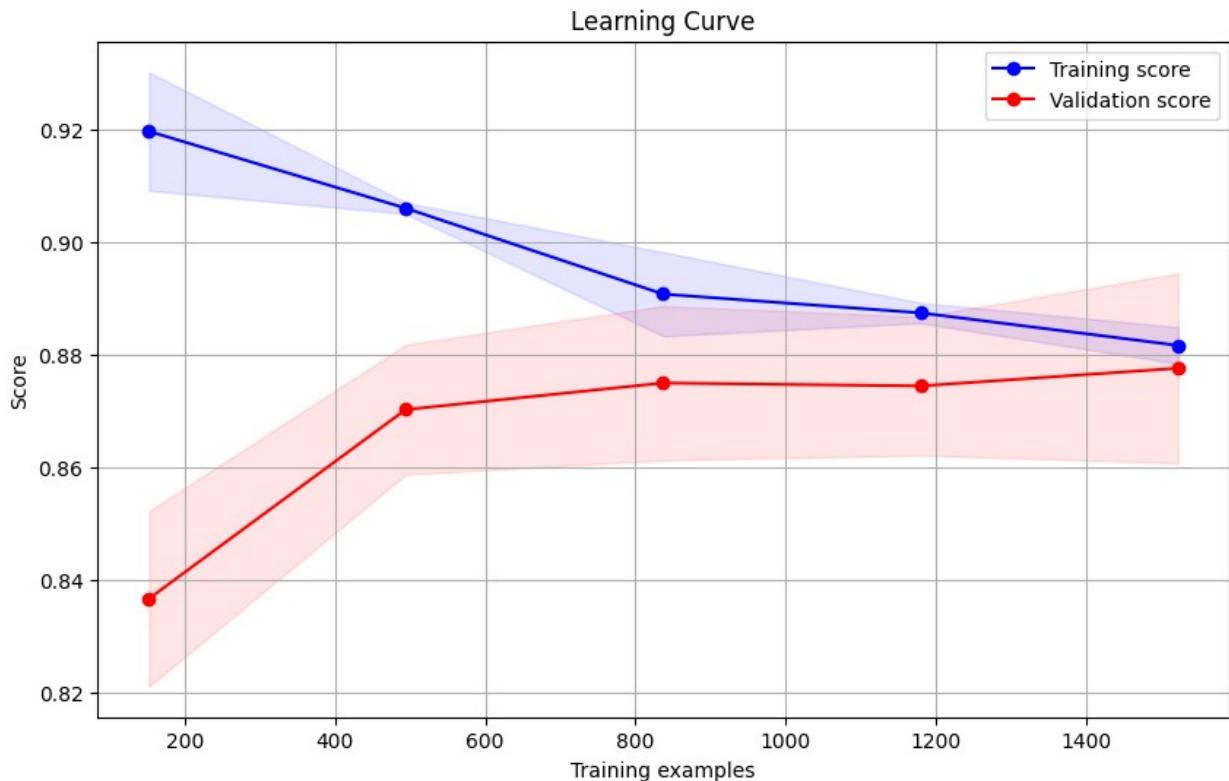
Train Classification Report:
      precision    recall   f1-score  support
          0       0.78      0.87      0.82      615
          1       0.93      0.89      0.91     1289
accuracy               0.88      1904
macro avg              0.86      0.88      0.87      1904
weighted avg            0.89      0.88      0.88      1904

Test Classification Report:
      precision    recall   f1-score  support
          0       0.81      0.86      0.83      150
          1       0.93      0.91      0.92      327
accuracy               0.89      477
macro avg              0.87      0.88      0.88      477
weighted avg            0.89      0.89      0.89      477

```







MLFL0W Logging is completed

## Balanced Dataset

### Hyper parameter Tuning

```
# Define the parameter grid for RandomizedSearchCV
start_tune_time = time.time()
param_dist = {
    'penalty': ['l1', 'l2', 'elasticnet', 'none'],
    'C': stats.uniform(loc=0.01, scale=5000-0.01), # Uniform distribution from 0.01 to 5000
    'solver': ['saga'], # saga solver supports all penalties
    'class_weight': ['balanced']
}

# Initialize the Logistic Regression model
logReg = LogisticRegression(max_iter=10000, random_state = 42)

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator= logReg,
    param_distributions=param_dist,
    n_iter=200, # Number of parameter settings to try
    cv=5, # Number of folds in cross-validation
```

```

    verbose=1,
    random_state=42,
    n_jobs=-1
)

# Fit RandomizedSearchCV
random_search.fit(X_train_bal, y_train_bal)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning_time", tuning_time)

Fitting 5 folds for each of 200 candidates, totalling 1000 fits
Best parameters found: {'C': 1872.706848835624, 'class_weight':
'balanced', 'penalty': 'l1', 'solver': 'saga'}
Best score: 0.8622969820124935
Tuning_time 98.46020817756653

```

## Logging Best Logistic Regression Model into MLFLOW

```

# Model details
name = "Tuned_Logistic_Regression_on_Balanced_Dataset"
bal_type = "Balanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time
start_train_time = time.time()
model.fit(X_train_bal, y_train_bal)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_bal_train = model.predict(X_train_bal)
y_pred_bal_test = model.predict(X_test_bal)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")

```

```

print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df,
mlflow_logging_and_metric_printing
time_df,feature_importance_df =
log_time_and_feature_importances_df(time_df,feature_importance_df,name
,training_time,testing_time,model,ola_features,bal_type,tuning_time)
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_bal,y_t
rain_bal,X_test_bal,y_test_bal,y_pred_bal_train,y_pred_bal_test,tuning
_score,**params)

Model: Tuned_Logistic_Regression_on_Balanced_Dataset
params: {'C': 1872.706848835624, 'class_weight': 'balanced',
'penalty': 'l1', 'solver': 'saga'}
Training Time: 1.5594 seconds
Testing Time: 0.0023 seconds
Tuning Time: 98.4602 seconds
Train Metrics:
Accuracy_train: 0.8693
Precision_train: 0.8574
Recall_train: 0.8860
F1_score_train: 0.8714
F2_score_train: 0.8801
Roc_auc_train: 0.9427
Pr_auc_train: 0.9460

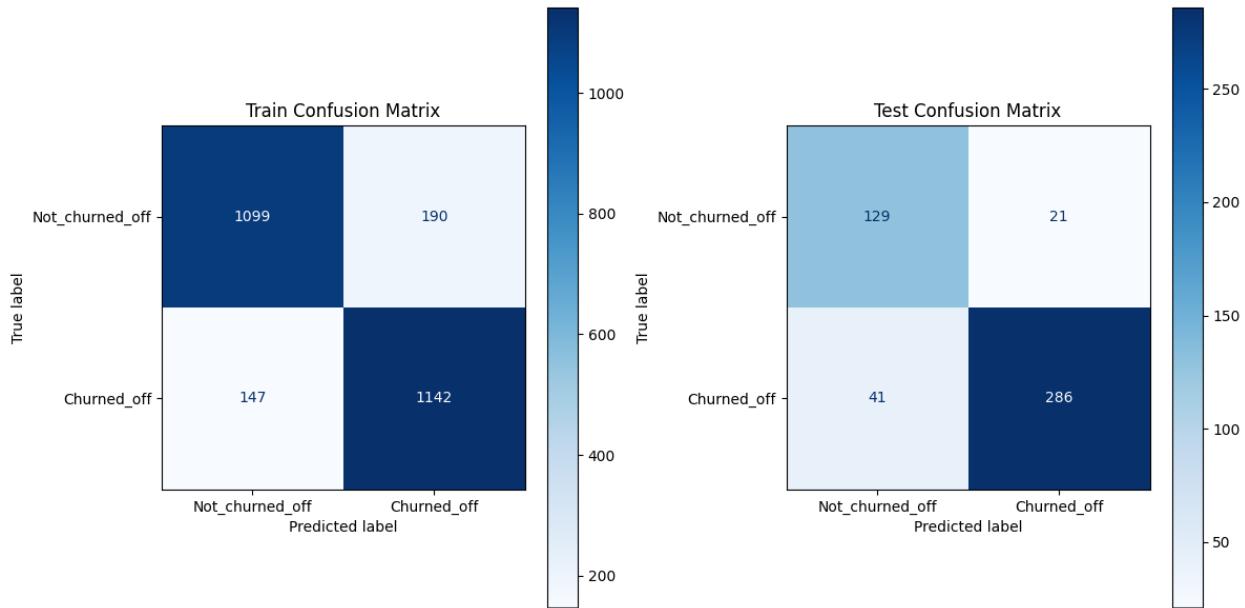
Test Metrics:
Accuracy_test: 0.8700
Precision_test: 0.9316
Recall_test: 0.8746
F1_score_test: 0.9022
F2_score_test: 0.8854
Roc_auc_test: 0.9298
Pr_auc_test: 0.9545

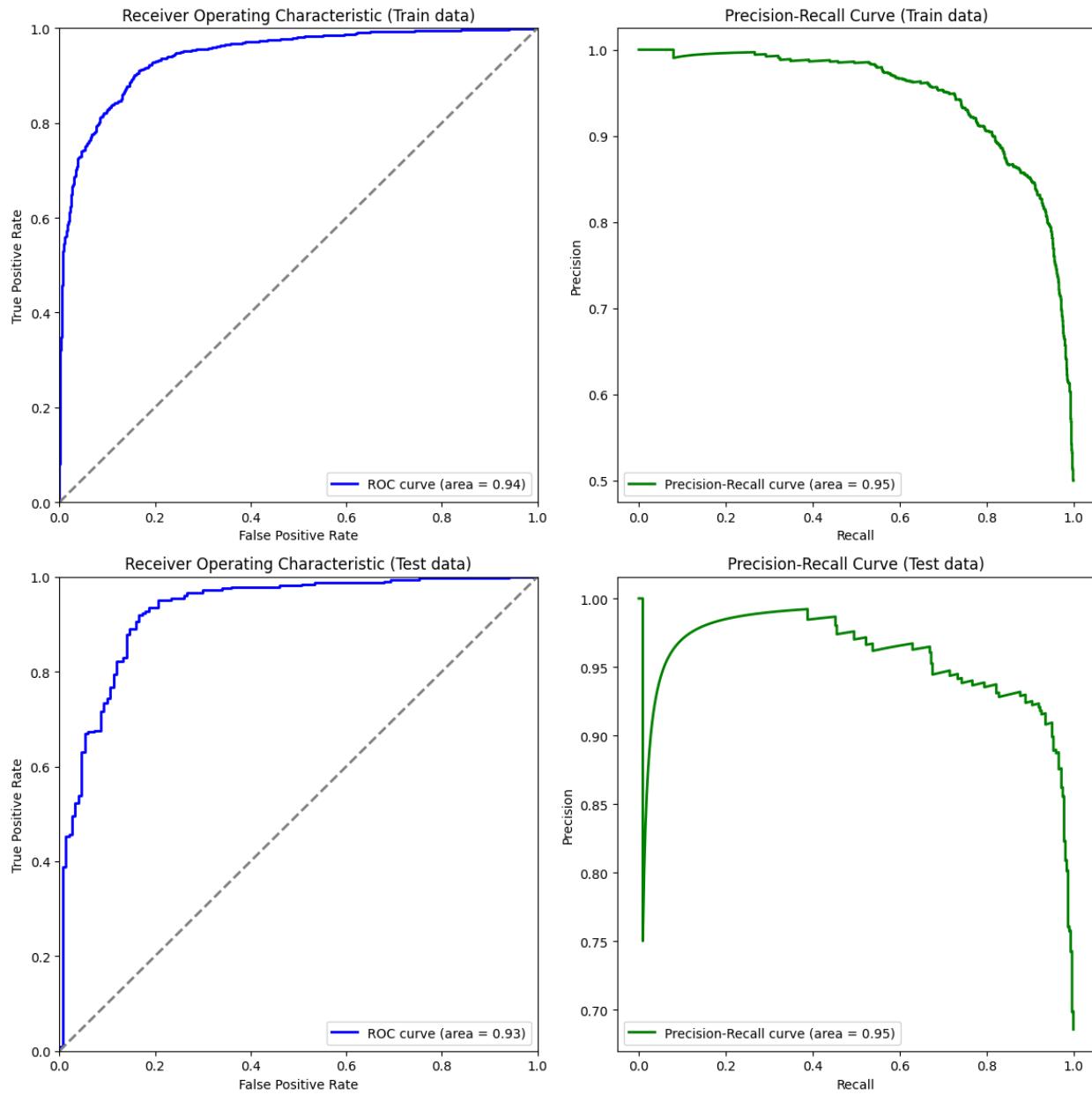
Tuning Metrics:
hyper_parameter_tuning_best_est_score: 0.8623

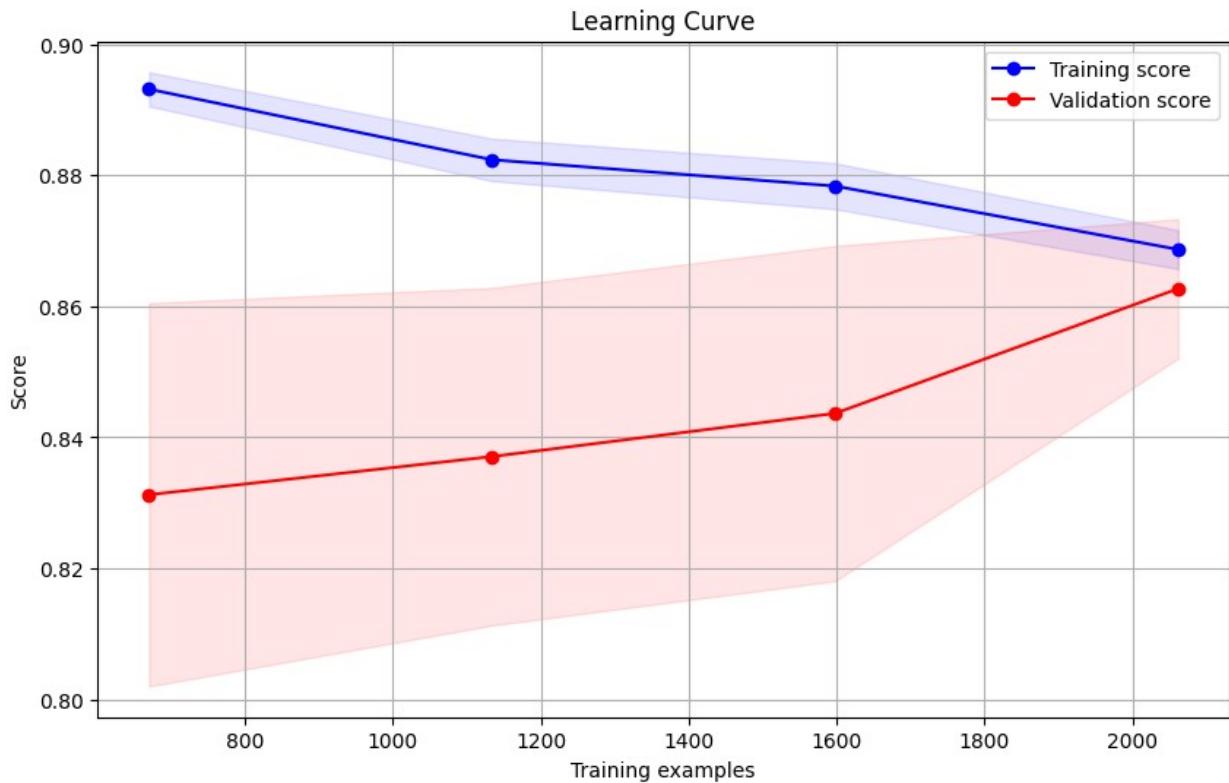
Train Classification Report:
      precision    recall   f1-score   support
          0       0.88      0.85      0.87     1289
          1       0.86      0.89      0.87     1289
          accuracy           0.87           0.87     2578
          macro avg       0.87      0.87      0.87     2578
          weighted avg     0.87      0.87      0.87     2578

```

Test Classification Report:					
	precision	recall	f1-score	support	
0	0.76	0.86	0.81	150	
1	0.93	0.87	0.90	327	
accuracy			0.87	477	
macro avg	0.85	0.87	0.85	477	
weighted avg	0.88	0.87	0.87	477	







MLFL0W Logging is completed

## MULTI\_LAYER\_PERCEPTRON NEURAL NETWORK MODEL

### Imbalanced Dataset

#### Hyper parameter Tuning

```
# Define the parameter grid
start_tune_time = time.time()
param_dist = {
    'hidden_layer_sizes': [(50,), (100,), (100, 50), (50, 50, 50)],
    'activation': ['identity', 'logistic', 'tanh', 'relu'],
    'solver': ['adam'], # 'adam' is suitable for all activation
functions
    'alpha': stats.uniform(0.0001, 5000), # Uniform distribution from
0.0001 to 5000
    'learning_rate': ['constant', 'invscaling', 'adaptive'],
    'learning_rate_init': stats.uniform(0.0001, 0.1), # Small
learning rates for better convergence
    'max_iter': stats.randint(200, 1000), # Increase max_iter for
better convergence
    'early_stopping': [True, False], # Use early stopping to prevent
overfitting
    'tol': [1e-4, 1e-5, 1e-6], # Lower tolerance for more precise
```

```

convergence
}

# Initialize the MLPClassifier
mlp = MLPClassifier(max_iter=10000, random_state=42)

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=mlp,
    param_distributions=param_dist,
    n_iter=200, # Number of parameter settings to try
    cv=5, # Number of folds in cross-validation
    verbose=1,
    random_state=42,
    n_jobs=-1 # Use all available cores
)

# Fit RandomizedSearchCV
random_search.fit(X_train_imb, y_train_imb)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning_time", tuning_time)

Fitting 5 folds for each of 200 candidates, totalling 1000 fits
Best parameters found: {'activation': 'logistic', 'alpha': 3.893929205071642, 'early_stopping': False, 'hidden_layer_sizes': (50,), 'learning_rate': 'constant', 'learning_rate_init': 0.030524224295953774, 'max_iter': 221, 'solver': 'adam', 'tol': 0.0001}
Best score: 0.8209020582953446
Tuning_time 34.14068078994751

```

## Observation

n\_iter increased to 200 in RandomizedSearchCV, Because 100 iter/candidate providing 0.67 score only.

## Logging Best MLPClassifier Model into MLFLOW

```

# Model details
name ="Tuned_MLPClassifier_on_Imbalanced_Dataset"
bal_type = "Imbalanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time

```

```

start_train_time = time.time()
model.fit(X_train_imb, y_train_imb)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_imb_train = model.predict(X_train_imb)
y_pred_imb_test = model.predict(X_test_imb)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df,
mlflow_logging_and_metric_printing
time_df,feature_importance_df =
log_time_and_feature_importances_df(time_df,feature_importance_df,name
,training_time,testing_time,model,ola_features,bal_type,tuning_time)
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_imb,y_t
rain_imb,X_test_imb,y_test_imb,y_pred_imb_train,y_pred_imb_test,tuning
_score,**params)

Model: Tuned_MLPClassifier_on_Imbalanced_Dataset
params: {'activation': 'logistic', 'alpha': 3.893929205071642,
'early_stopping': False, 'hidden_layer_sizes': (50,), 'learning_rate':
'constant', 'learning_rate_init': 0.030524224295953774, 'max_iter':
221, 'solver': 'adam', 'tol': 0.0001}
Training Time: 0.2157 seconds
Testing Time: 0.0073 seconds
Tuning Time: 34.1407 seconds
Train Metrics:
Accuracy_train: 0.8309
Precision_train: 0.8300
Recall_train: 0.9434
F1_score_train: 0.8831
F2_score_train: 0.9183
Roc_auc_train: 0.8991
Pr_auc_train: 0.9487

Test Metrics:

```

```
Accuracy_test: 0.8281
Precision_test: 0.8267
Recall_test: 0.9480
F1_score_test: 0.8832
F2_score_test: 0.9210
Roc_auc_test: 0.8858
Pr_auc_test: 0.9342
```

Tuning Metrics:

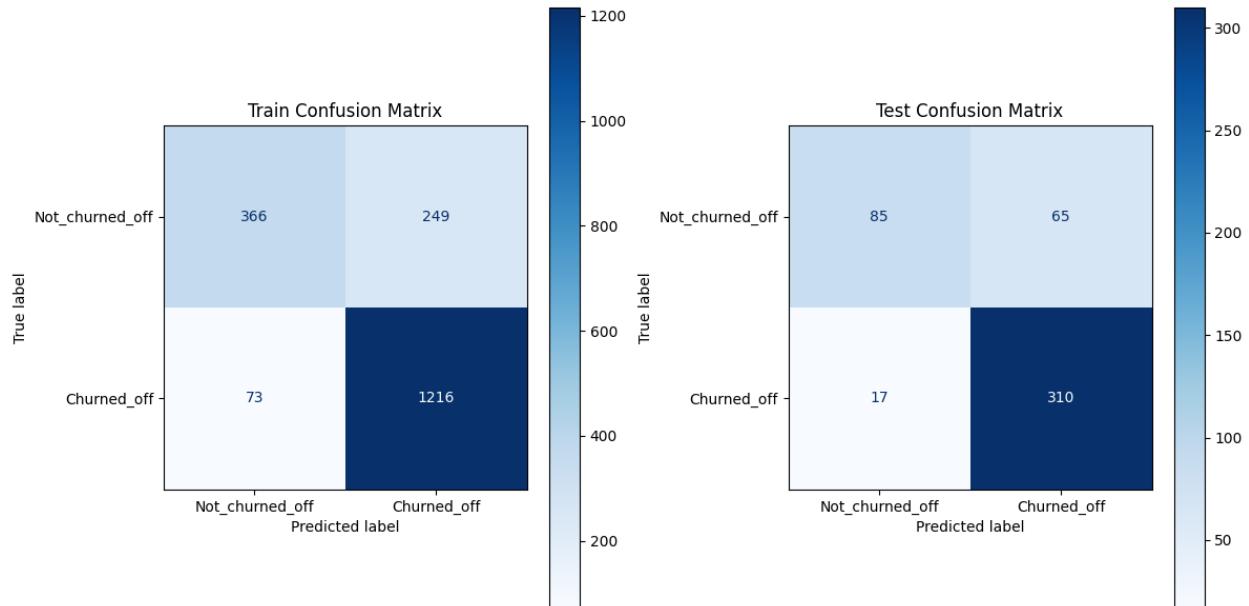
```
hyper_parameter_tuning_best_est_score: 0.8209
```

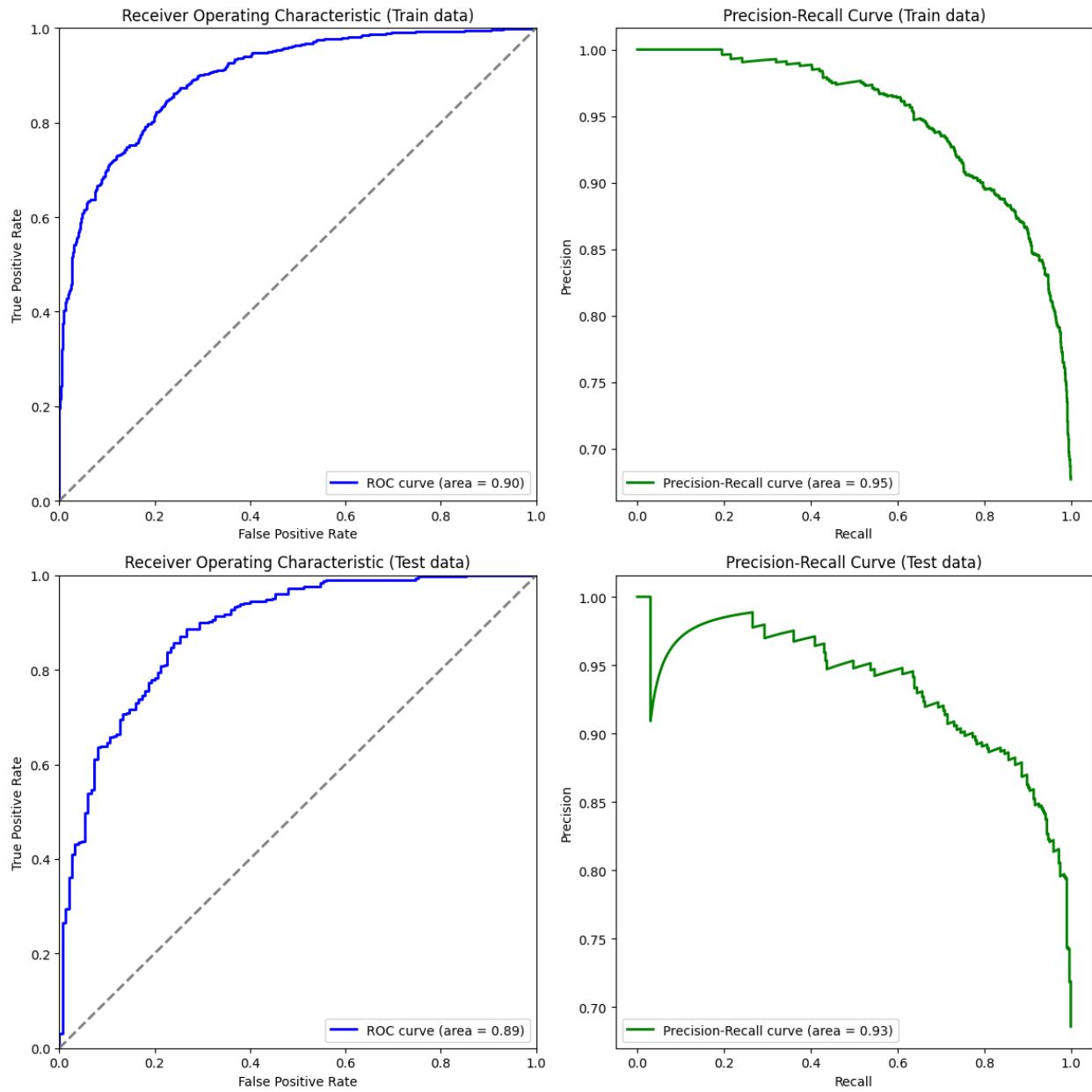
Train Classification Report:

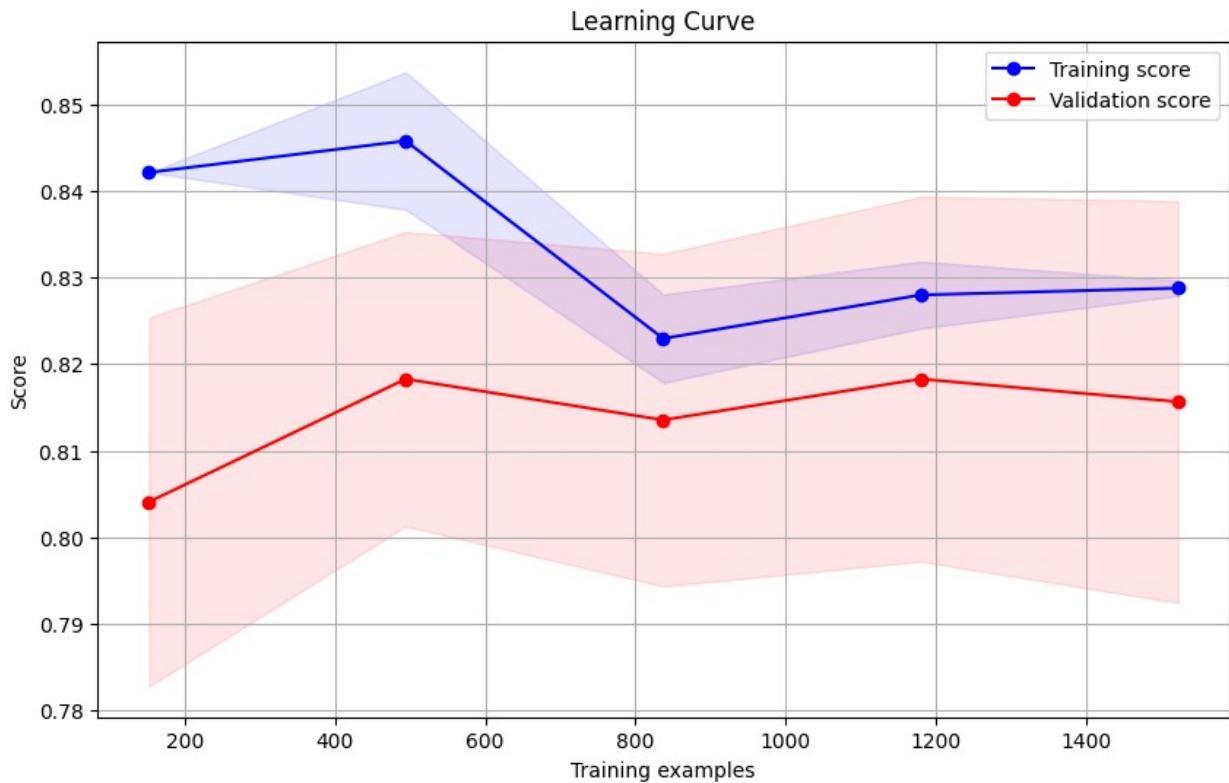
	precision	recall	f1-score	support
0	0.83	0.60	0.69	615
1	0.83	0.94	0.88	1289
accuracy			0.83	1904
macro avg	0.83	0.77	0.79	1904
weighted avg	0.83	0.83	0.82	1904

Test Classification Report:

	precision	recall	f1-score	support
0	0.83	0.57	0.67	150
1	0.83	0.95	0.88	327
accuracy			0.83	477
macro avg	0.83	0.76	0.78	477
weighted avg	0.83	0.83	0.82	477







MLFLOW Logging is completed

## Balanced Dataset

### Hyper parameter Tuning

```
# Define the parameter grid
start_tune_time = time.time()
param_dist = {
    'hidden_layer_sizes': [(50,), (100,), (100, 50), (50, 50, 50)],
    'activation': ['identity', 'logistic', 'tanh', 'relu'],
    'solver': ['adam'], # 'adam' is suitable for all activation
functions
    'alpha': stats.uniform(0.0001, 5000), # Uniform distribution from
0.0001 to 5000
    'learning_rate': ['constant', 'invscaling', 'adaptive'],
    'learning_rate_init': stats.uniform(0.0001, 0.1), # Small
learning rates for better convergence
    'max_iter': stats.randint(200, 1000), # Increase max_iter for
better convergence
    'early_stopping': [True, False], # Use early stopping to prevent
overfitting
    'tol': [1e-4, 1e-5, 1e-6], # Lower tolerance for more precise
convergence
}
```

```

# Initialize the MLPClassifier
mlp = MLPClassifier(max_iter=10000, random_state=42)

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=mlp,
    param_distributions=param_dist,
    n_iter=200, # Number of parameter settings to try
    cv=5, # Number of folds in cross-validation
    verbose=1,
    random_state=42,
    n_jobs=-1 # Use all available cores
)

# Fit RandomizedSearchCV
random_search.fit(X_train_bal, y_train_bal)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning_time", tuning_time)

Fitting 5 folds for each of 200 candidates, totalling 1000 fits
Best parameters found: {'activation': 'logistic', 'alpha': 3.893929205071642, 'early_stopping': False, 'hidden_layer_sizes': (50,), 'learning_rate': 'constant', 'learning_rate_init': 0.030524224295953774, 'max_iter': 221, 'solver': 'adam', 'tol': 0.0001}
Best score: 0.7878166629035901
Tuning_time 47.7030189037323

```

## Logging Best MLPClassifier Model into MLFLOW

```

# Model details
name ="Tuned_MLPClassifier_on_Balanced_Dataset"
bal_type = "Balanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time
start_train_time = time.time()
model.fit(X_train_bal, y_train_bal)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

```

```

# Record testing time
start_test_time = time.time()
y_pred_bal_train = model.predict(X_train_bal)
y_pred_bal_test = model.predict(X_test_bal)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df,
mlflow_logging_and_metric_printing
time_df,feature_importance_df =
log_time_and_feature_importances_df(time_df,feature_importance_df,name
,training_time,testing_time,model,ola_features,bal_type,tuning_time)
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_bal,y_t
rain_bal,X_test_bal,y_test_bal,y_pred_bal_train,y_pred_bal_test,tuning
_score,**params)

Model: Tuned_MLPClassifier_on_Balanced_Dataset
params: {'activation': 'logistic', 'alpha': 3.893929205071642,
'early_stopping': False, 'hidden_layer_sizes': (50,), 'learning_rate':
'constant', 'learning_rate_init': 0.030524224295953774, 'max_iter':
221, 'solver': 'adam', 'tol': 0.0001}
Training Time: 0.3507 seconds
Testing Time: 0.0092 seconds
Tuning Time: 47.7030 seconds
Train Metrics:
Accuracy_train: 0.7995
Precision_train: 0.7801
Recall_train: 0.8340
F1_score_train: 0.8061
F2_score_train: 0.8226
Roc_auc_train: 0.8945
Pr_auc_train: 0.9019

Test Metrics:
Accuracy_test: 0.8113
Precision_test: 0.8738
Recall_test: 0.8471
F1_score_test: 0.8602
F2_score_test: 0.8523
Roc_auc_test: 0.8727

```

Pr\_auc\_test: 0.9315

Tuning Metrics:

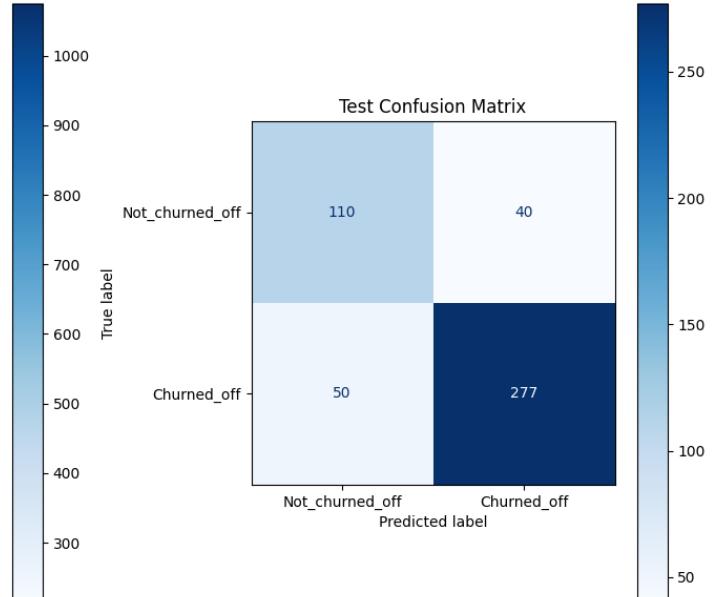
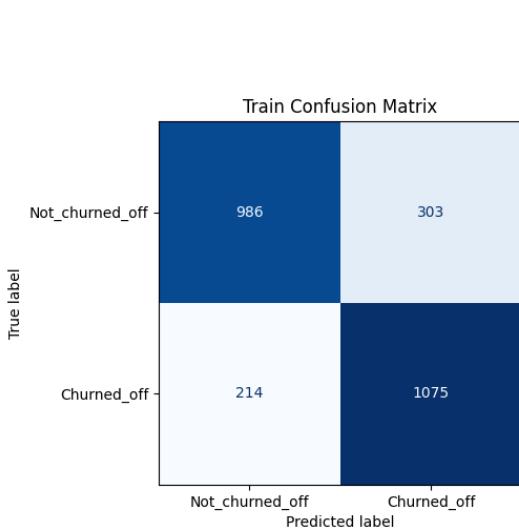
hyper\_parameter\_tuning\_best\_est\_score: 0.7878

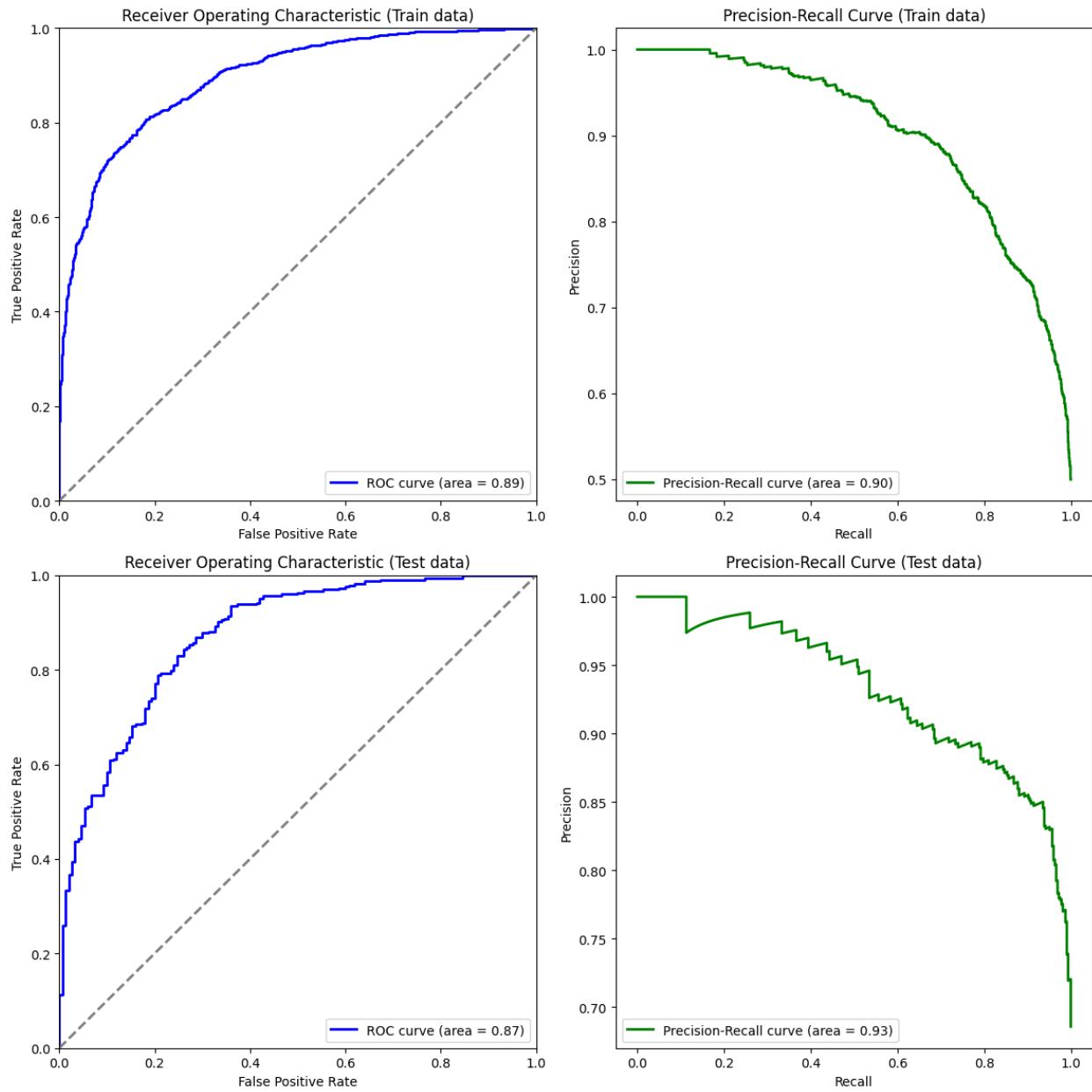
Train Classification Report:

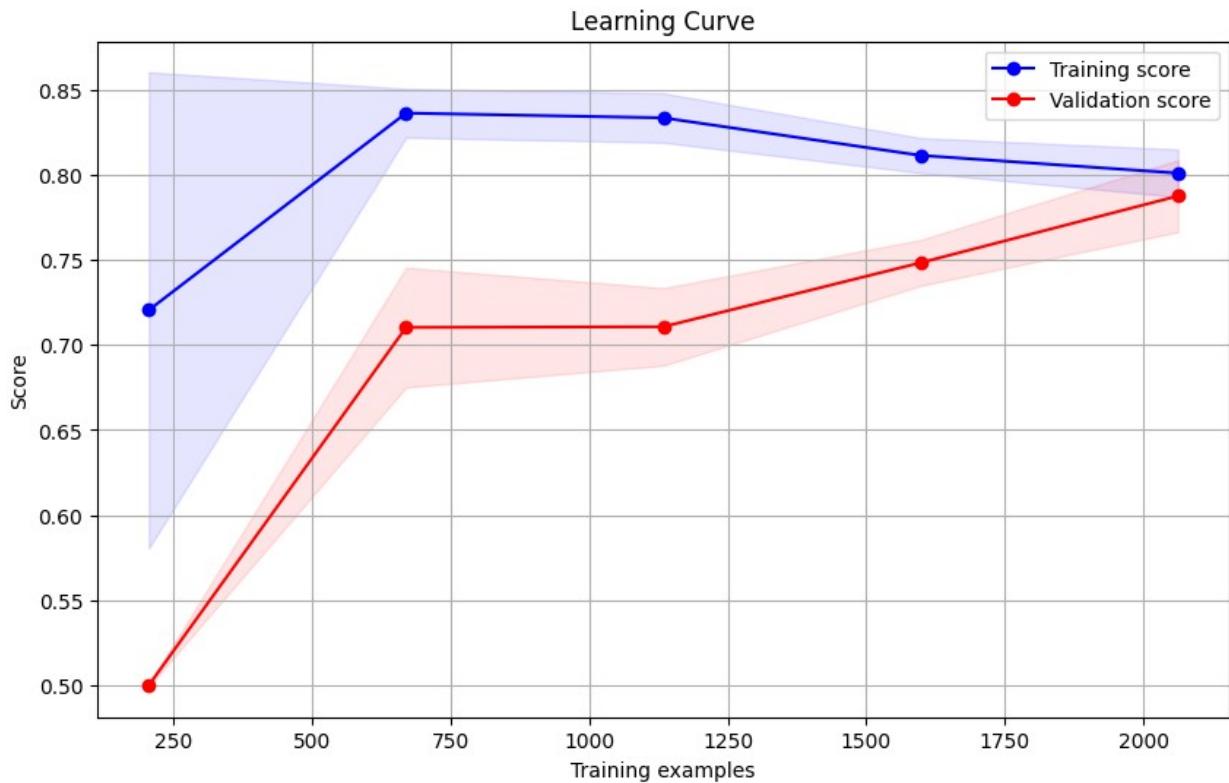
	precision	recall	f1-score	support
0	0.82	0.76	0.79	1289
1	0.78	0.83	0.81	1289
accuracy			0.80	2578
macro avg	0.80	0.80	0.80	2578
weighted avg	0.80	0.80	0.80	2578

Test Classification Report:

	precision	recall	f1-score	support
0	0.69	0.73	0.71	150
1	0.87	0.85	0.86	327
accuracy			0.81	477
macro avg	0.78	0.79	0.78	477
weighted avg	0.82	0.81	0.81	477







MLFL0W Logging is completed

## K NEAREST NEIGHBORS MODEL

### Imbalanced Dataset

#### Hyper parameter Tuning

```
# Define the parameter grid
start_tune_time = time.time()
param_dist = {
    'n_neighbors': stats.randint(1, 35),
    'weights': ['uniform', 'distance'],
    'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute'],
    'leaf_size': stats.randint(10, 50),
    'p': [1, 2, 3, 4], # Manhattan power
    'metric': ['minkowski', 'euclidean', 'manhattan', 'chebyshev']
}

# Initialize the KNN model
knn = KNeighborsClassifier()

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=knn,
```

```

param_distributions=param_dist,
n_iter=200, # Number of parameter settings to try
cv=5, # Number of folds in cross-validation
verbose=1,
random_state=42,
n_jobs=-1 # Use all available cores
)

# Fit RandomizedSearchCV
random_search.fit(X_train_imb, y_train_imb)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning_time", tuning_time)

Fitting 5 folds for each of 200 candidates, totalling 1000 fits
Best parameters found: {'algorithm': 'kd_tree', 'leaf_size': 28,
'metric': 'manhattan', 'n_neighbors': 11, 'p': 3, 'weights':
'distance'}
Best score: 0.8749896394529632
Tuning_time 9.158213376998901

```

### Logging Best K Nearest Neighbor Model into MLFLOW

```

# Model details
name = "Tuned_KNN_on_Imbalanced_Dataset"
bal_type = "Imbalanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time
start_train_time = time.time()
model.fit(X_train_imb, y_train_imb)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_imb_train = model.predict(X_train_imb)
y_pred_imb_test = model.predict(X_test_imb)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time

```

```

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df,
mlflow_logging_and_metric_printing
time_df, feature_importance_df =
log_time_and_feature_importances_df(time_df,feature_importance_df,name
,training_time,testing_time,model,ola_features,bal_type,tuning_time)
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_imb,y_t
rain_imb,X_test_imb,y_test_imb,y_pred_imb_train,y_pred_imb_test,tuning
_score,**params)

Model: Tuned_KNN_on_Imbalanced_Dataset
params: {'algorithm': 'kd_tree', 'leaf_size': 28, 'metric':
'manhattan', 'n_neighbors': 11, 'p': 3, 'weights': 'distance'}
Training Time: 0.0064 seconds
Testing Time: 0.1407 seconds
Tuning Time: 9.1582 seconds
Train Metrics:
Accuracy_train: 1.0000
Precision_train: 1.0000
Recall_train: 1.0000
F1_score_train: 1.0000
F2_score_train: 1.0000
Roc_auc_train: 1.0000
Pr_auc_train: 1.0000

Test Metrics:
Accuracy_test: 0.8700
Precision_test: 0.8775
Recall_test: 0.9419
F1_score_test: 0.9086
F2_score_test: 0.9283
Roc_auc_test: 0.9123
Pr_auc_test: 0.9522

Tuning Metrics:
hyper_parameter_tuning_best_est_score: 0.8750

Train Classification Report:
      precision    recall   f1-score   support
          0         1.00      1.00      1.00       615
          1         1.00      1.00      1.00      1289

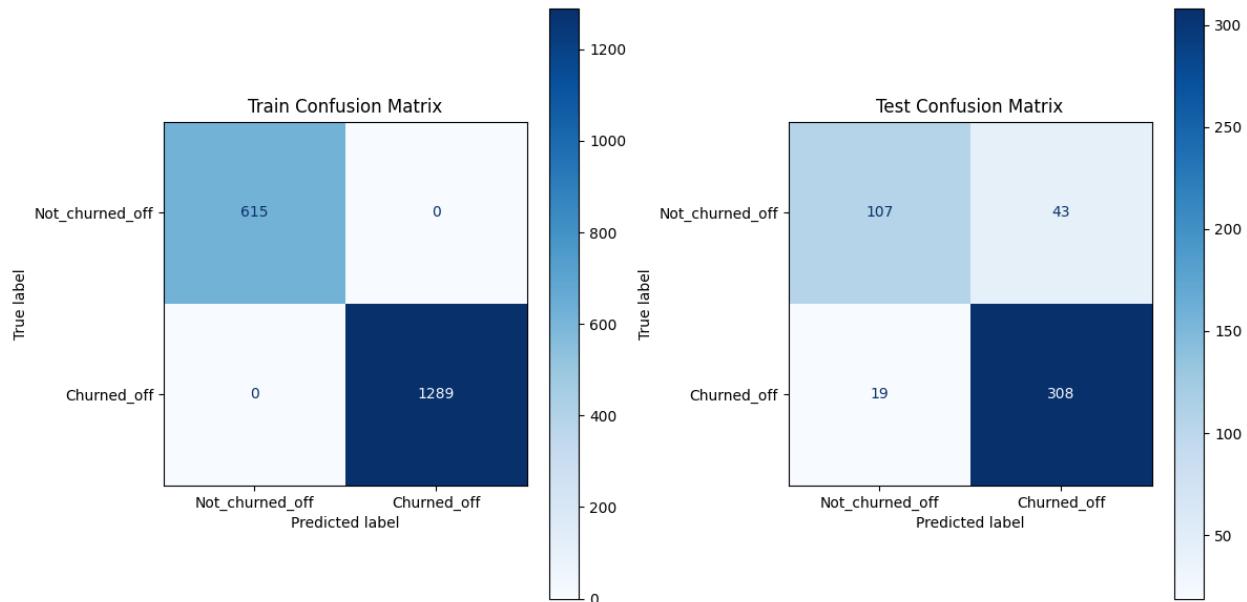
```

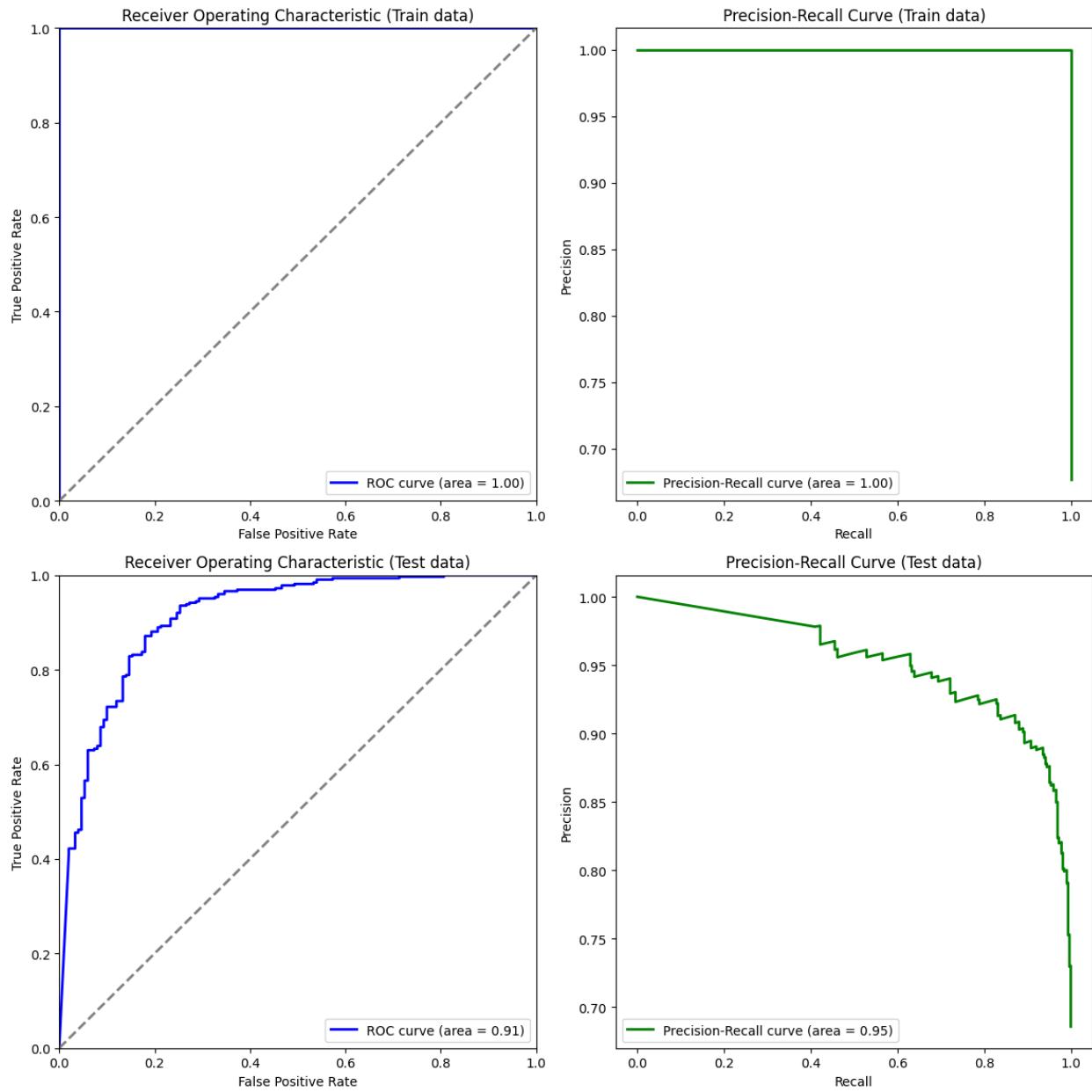
accuracy			1.00	1904
macro avg	1.00	1.00	1.00	1904
weighted avg	1.00	1.00	1.00	1904

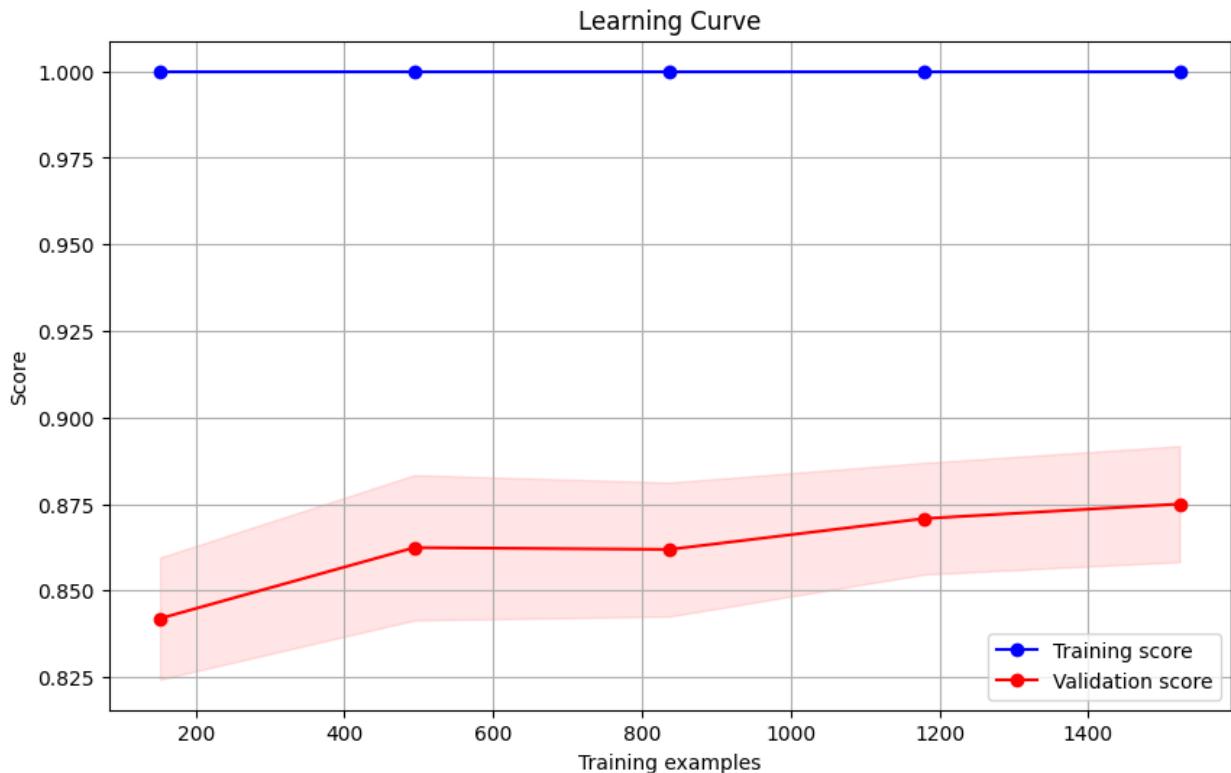
#### Test Classification Report:

	precision	recall	f1-score	support
0	0.85	0.71	0.78	150
1	0.88	0.94	0.91	327

accuracy		0.87	477
macro avg	0.86	0.83	0.84
weighted avg	0.87	0.87	0.87







MLFLOW Logging is completed

## Balanced Dataset

### Hyper parameter Tuning

```
# Define the parameter grid
start_tune_time = time.time()
param_dist = {
    'n_neighbors': stats.randint(1, 35),
    'weights': ['uniform', 'distance'],
    'algorithm': ['auto', 'ball_tree', 'kd_tree', 'brute'],
    'leaf_size': stats.randint(10, 50),
    'p': [1, 2, 3, 4], # Manhattan power
    'metric': ['minkowski', 'euclidean', 'manhattan', 'chebyshev']
}

# Initialize the KNN model
knn = KNeighborsClassifier()

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=knn,
    param_distributions=param_dist,
    n_iter=200, # Number of parameter settings to try
    cv=5, # Number of folds in cross-validation
```

```

        verbose=1,
        random_state=42,
        n_jobs=-1 # Use all available cores
    )

# Fit RandomizedSearchCV
random_search.fit(X_train_bal, y_train_bal)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning_time", tuning_time)

Fitting 5 folds for each of 200 candidates, totalling 1000 fits
Best parameters found: {'algorithm': 'ball_tree', 'leaf_size': 49,
'metric': 'minkowski', 'n_neighbors': 16, 'p': 1, 'weights':
'distance'}
Best score: 0.8758794310228041
Tuning_time 14.376519441604614

```

## Logging Best K Nearest Neighbor Model into MLFLOW

```

# Model details
name = "Tuned_KNN_on_Balanced_Dataset"
bal_type = "Balanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time
start_train_time = time.time()
model.fit(X_train_bal, y_train_bal)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_bal_train = model.predict(X_train_bal)
y_pred_bal_test = model.predict(X_test_bal)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")

```

```

print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df,
mlflow_logging_and_metric_printing
time_df,feature_importance_df =
log_time_and_feature_importances_df(time_df,feature_importance_df,name
,training_time,testing_time,model,ola_features,bal_type,tuning_time)
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_bal,y_t
rain_bal,X_test_bal,y_test_bal,y_pred_bal_train,y_pred_bal_test,tuning
_score,**params)

Model: Tuned_KNN_on_Balanced_Dataset
params: {'algorithm': 'ball_tree', 'leaf_size': 49, 'metric':
'minkowski', 'n_neighbors': 16, 'p': 1, 'weights': 'distance'}
Training Time: 0.0061 seconds
Testing Time: 0.1884 seconds
Tuning Time: 14.3765 seconds
Train Metrics:
Accuracy_train: 1.0000
Precision_train: 1.0000
Recall_train: 1.0000
F1_score_train: 1.0000
F2_score_train: 1.0000
Roc_auc_train: 1.0000
Pr_auc_train: 1.0000

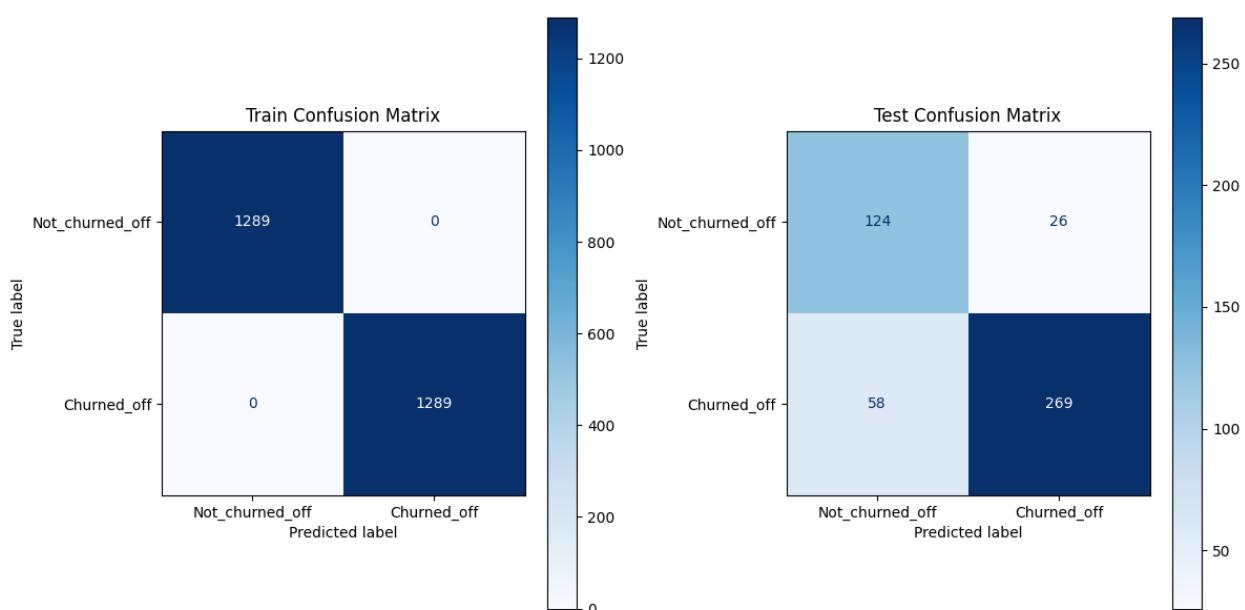
Test Metrics:
Accuracy_test: 0.8239
Precision_test: 0.9119
Recall_test: 0.8226
F1_score_test: 0.8650
F2_score_test: 0.8391
Roc_auc_test: 0.8987
Pr_auc_test: 0.9411

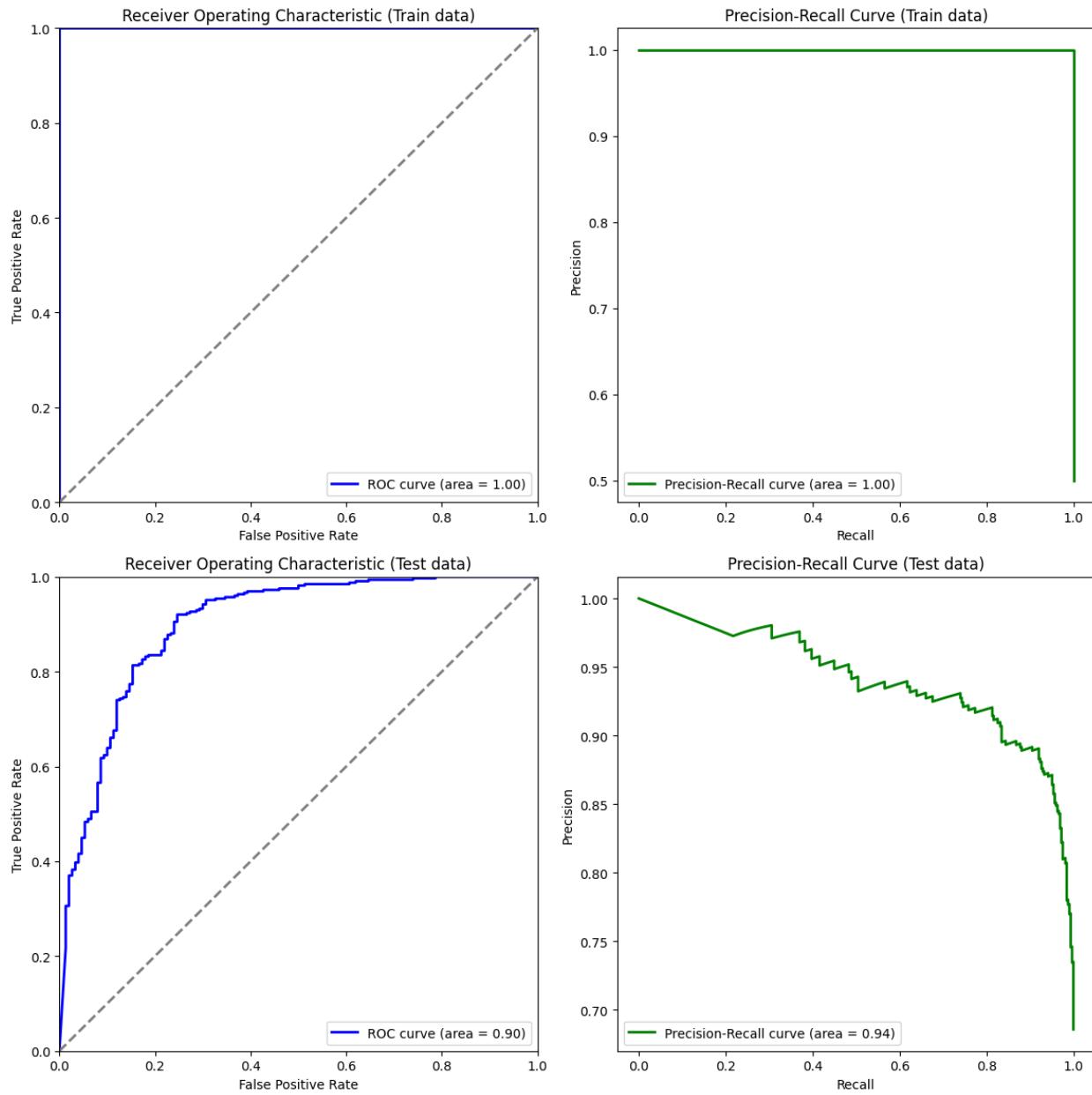
Tuning Metrics:
hyper_parameter_tuning_best_est_score: 0.8759

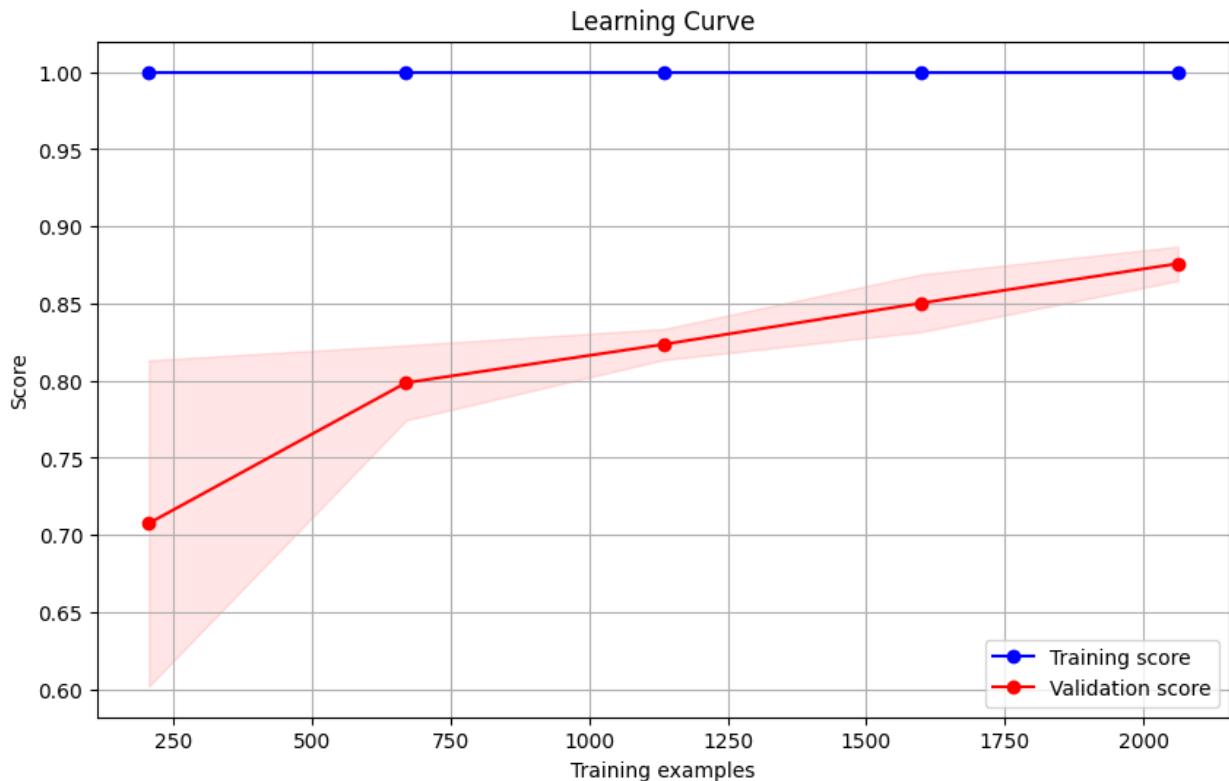
Train Classification Report:
      precision    recall   f1-score   support
          0         1.00     1.00     1.00     1289
          1         1.00     1.00     1.00     1289
          accuracy                           1.00     2578
          macro avg       1.00       1.00       1.00     2578
          weighted avg    1.00       1.00       1.00     2578

```

Test Classification Report:					
	precision	recall	f1-score	support	
0	0.68	0.83	0.75	150	
1	0.91	0.82	0.86	327	
accuracy			0.82	477	
macro avg	0.80	0.82	0.81	477	
weighted avg	0.84	0.82	0.83	477	







MLFL0W Logging is completed

## SUPPORT VECTOR MACHINE MODEL

### Imbalanced Dataset

#### Hyper parameter Tuning

```
# Define the parameter grid
start_tune_time = time.time()
param_dist = {
    'C': stats.expon(scale=100), # Exponential distribution for C
    'kernel': ['linear', 'poly', 'rbf', 'sigmoid'],
    'degree': stats.randint(2, 5), # Degrees for polynomial kernel
    'gamma': ['scale', 'auto'], # Options for gamma
    'shrinking': [True, False],
    'max_iter': stats.randint(100, 2000), # Random integers for max_iter
    'random_state': [42] # Fixed random state for reproducibility
}

# Initialize the SVC model
svc = SVC(class_weight='balanced', probability=True)

# Setup RandomizedSearchCV
```

```

random_search = RandomizedSearchCV(
    estimator=svc,
    param_distributions=param_dist,
    n_iter=200, # Number of parameter settings to try
    cv=5, # Number of folds in cross-validation
    verbose=1,
    random_state=42,
    n_jobs=-1 # Use all available cores
)

# Fit RandomizedSearchCV
random_search.fit(X_train_imb, y_train_imb)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning_time", tuning_time)

Fitting 5 folds for each of 200 candidates, totalling 1000 fits
Best parameters found: {'C': 6.436932213118794, 'degree': 4, 'gamma': 'auto', 'kernel': 'rbf', 'max_iter': 1130, 'random_state': 42, 'shrinking': True}
Best score: 0.8839218124050283
Tuning_time 28.28290867805481

```

## Logging Best Support Vector Machine Model into MLFLOW

```

# Model details
name = "Tuned_SVC_on_Imbalanced_Dataset"
bal_type = "Imbalanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time
start_train_time = time.time()
model.fit(X_train_imb, y_train_imb)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_imb_train = model.predict(X_train_imb)
y_pred_imb_test = model.predict(X_test_imb)
end_test_time = time.time()

```

```

# Calculate testing time
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df,
mlflow_logging_and_metric_printing
time_df,feature_importance_df =
log_time_and_feature_importances_df(time_df,feature_importance_df,name
,training_time,testing_time,model,ola_features,bal_type,tuning_time)
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_imb,y_t
rain_imb,X_test_imb,y_test_imb,y_pred_imb_train,y_pred_imb_test,tuning
_score,**params)

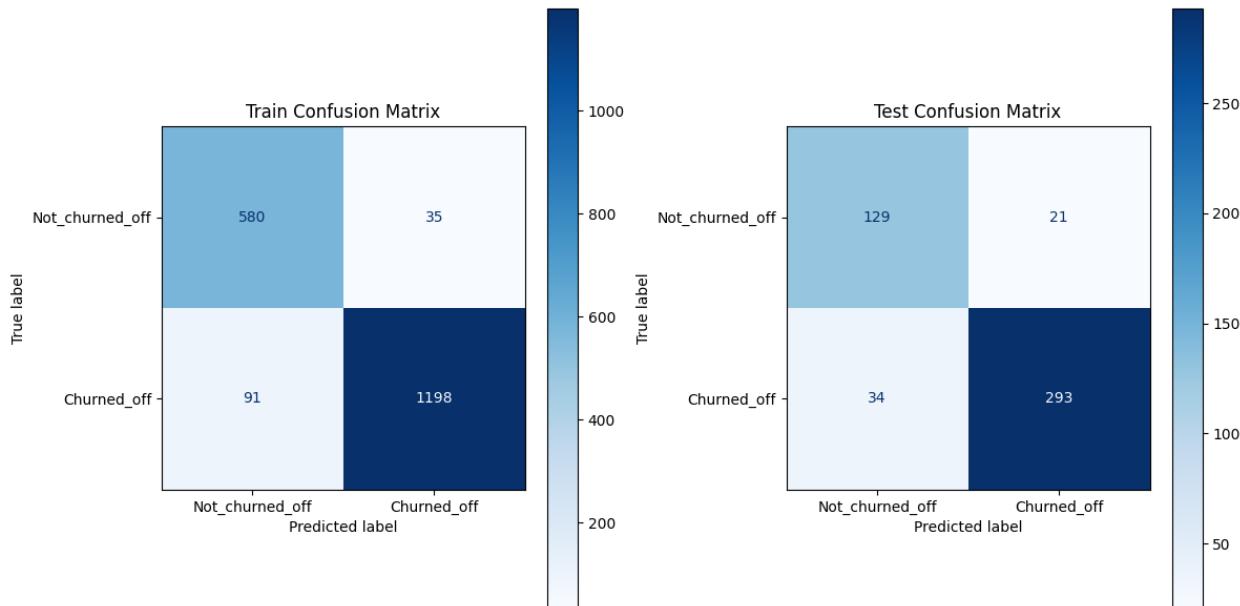
Model: Tuned_SVC_on_Imbalanced_Dataset
params: {'C': 6.436932213118794, 'degree': 4, 'gamma': 'auto',
'kernel': 'rbf', 'max_iter': 1130, 'random_state': 42, 'shrinking':
True}
Training Time: 0.3759 seconds
Testing Time: 0.1145 seconds
Tuning Time: 28.2829 seconds
Train Metrics:
Accuracy_train: 0.9338
Precision_train: 0.9716
Recall_train: 0.9294
F1_score_train: 0.9500
F2_score_train: 0.9375
Roc_auc_train: 0.9821
Pr_auc_train: 0.9915

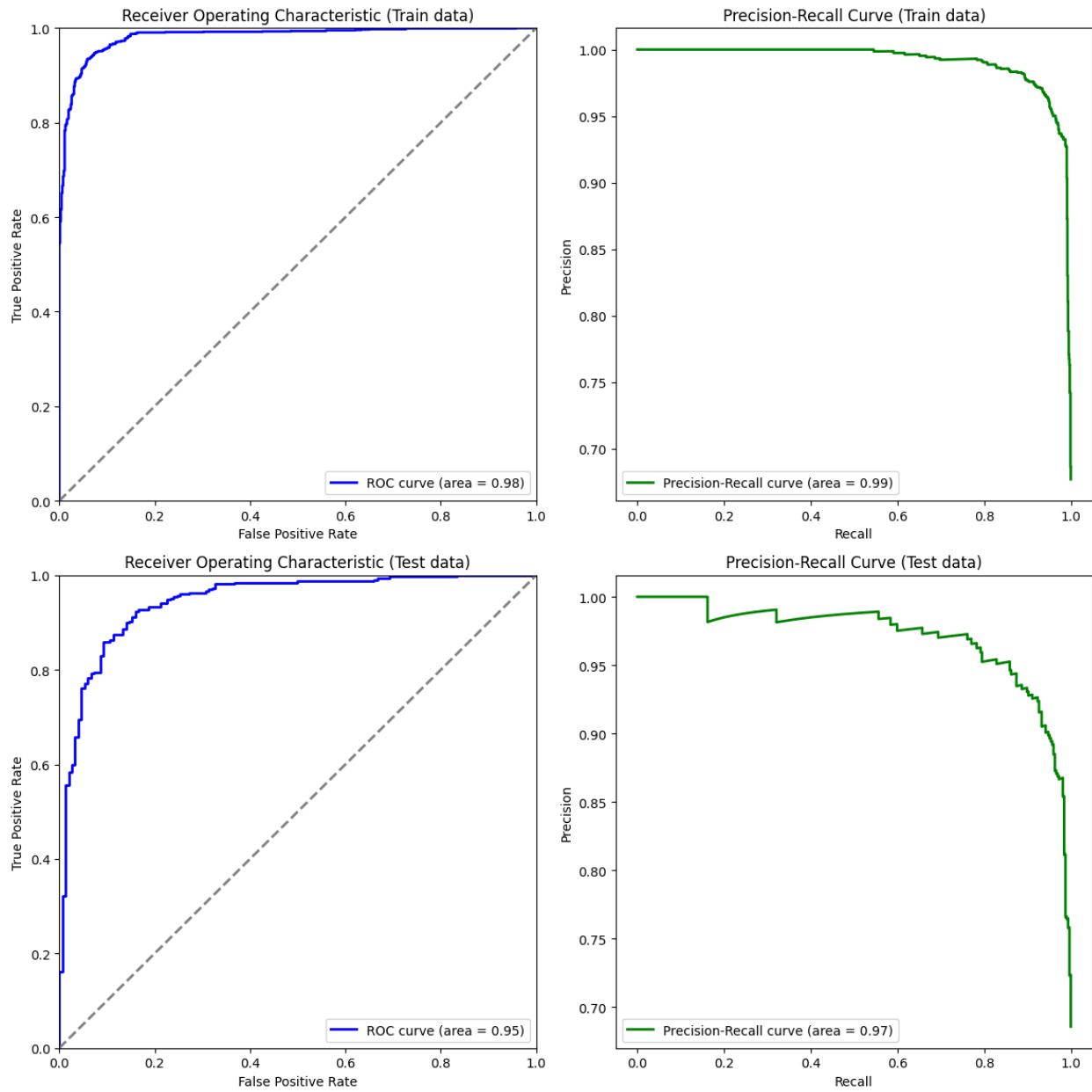
Test Metrics:
Accuracy_test: 0.8847
Precision_test: 0.9331
Recall_test: 0.8960
F1_score_test: 0.9142
F2_score_test: 0.9032
Roc_auc_test: 0.9455
Pr_auc_test: 0.9708

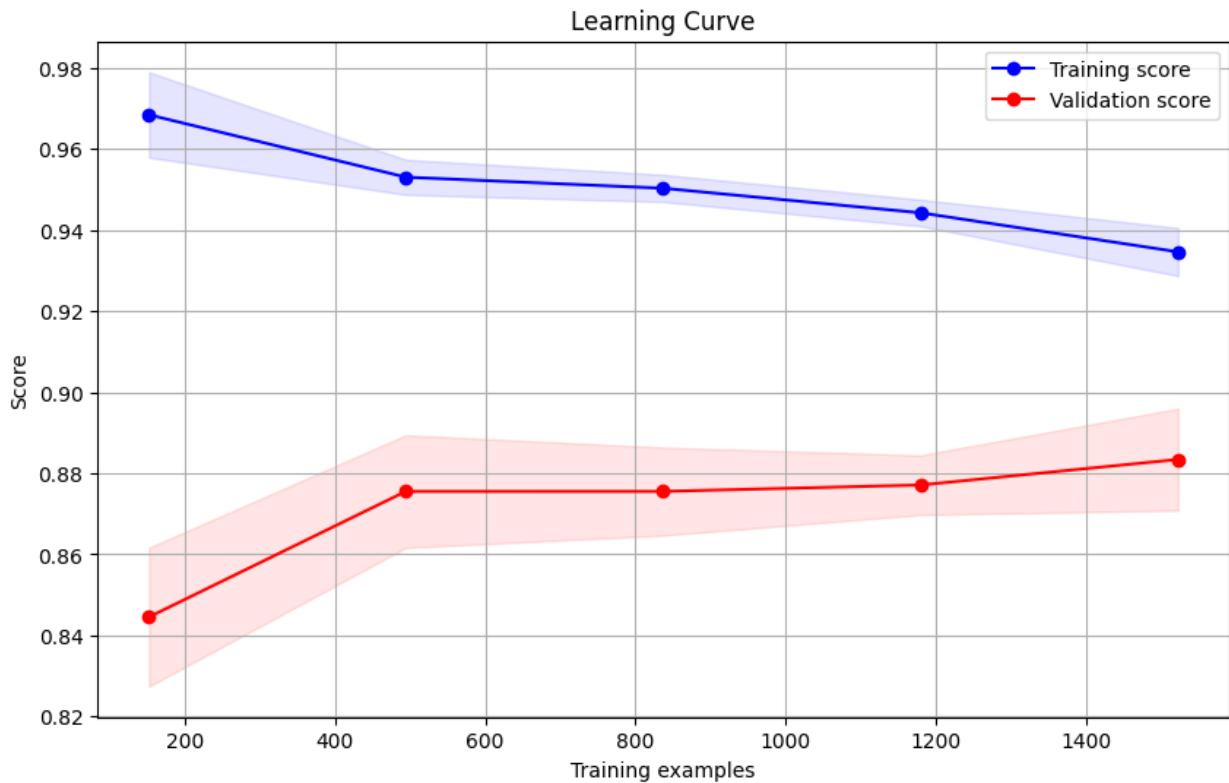
Tuning Metrics:
hyper_parameter_tuning_best_est_score: 0.8839

Train Classification Report:
    precision    recall   f1-score   support
```

	0	0.86	0.94	0.90	615
	1	0.97	0.93	0.95	1289
accuracy				0.93	1904
macro avg		0.92	0.94	0.93	1904
weighted avg		0.94	0.93	0.93	1904
<b>Test Classification Report:</b>					
		precision	recall	f1-score	support
	0	0.79	0.86	0.82	150
	1	0.93	0.90	0.91	327
accuracy				0.88	477
macro avg		0.86	0.88	0.87	477
weighted avg		0.89	0.88	0.89	477







MLFL0W Logging is completed

## Balanced Dataset

### Hyper parameter Tuning

```
# Define the parameter grid
start_tune_time = time.time()
param_dist = {
    'C': stats.expon(scale=100), # Exponential distribution for C
    'kernel': ['linear', 'poly', 'rbf', 'sigmoid'],
    'degree': stats.randint(2, 5), # Degrees for polynomial kernel
    'gamma': ['scale', 'auto'], # Options for gamma
    'shrinking': [True, False],
    'max_iter': stats.randint(100, 2000), # Random integers for
max_iter
    'random_state': [42] # Fixed random state for reproducibility
}

# Initialize the SVC model
svc = SVC(class_weight='balanced', probability=True)

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=svc,
```

```

param_distributions=param_dist,
n_iter=200, # Number of parameter settings to try
cv=5, # Number of folds in cross-validation
verbose=1,
random_state=42,
n_jobs=-1 # Use all available cores
)

# Fit RandomizedSearchCV
random_search.fit(X_train_bal, y_train_bal)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning_time", tuning_time)

Fitting 5 folds for each of 200 candidates, totalling 1000 fits
Best parameters found: {'C': 81.5862458136845, 'degree': 2, 'gamma': 'auto', 'kernel': 'rbf', 'max_iter': 1677, 'random_state': 42, 'shrinking': True}
Best score: 0.8976096936855573
Tuning_time 48.674527645111084

```

### Logging Best Support Vector Machine Model into MLFLOW

```

# Model details
name = "Tuned_SVC_on_Balanced_Dataset"
bal_type = "Balanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time
start_train_time = time.time()
model.fit(X_train_bal, y_train_bal)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_bal_train = model.predict(X_train_bal)
y_pred_bal_test = model.predict(X_test_bal)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time

```

```

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df,
mlflow_logging_and_metric_printing
time_df, feature_importance_df =
log_time_and_feature_importances_df(time_df, feature_importance_df, name
, training_time, testing_time, model, ola_features, bal_type, tuning_time)
mlflow_logging_and_metric_printing(model, name, bal_type, X_train_bal, y_train_bal, X_test_bal, y_test_bal, y_pred_bal_train, y_pred_bal_test, tuning_score, **params)

Model: Tuned_SVC_on_Balanced_Dataset
params: {'C': 81.5862458136845, 'degree': 2, 'gamma': 'auto',
'kernel': 'rbf', 'max_iter': 1677, 'random_state': 42, 'shrinking':
True}
Training Time: 0.6545 seconds
Testing Time: 0.1660 seconds
Tuning Time: 48.6745 seconds
Train Metrics:
Accuracy_train: 0.9728
Precision_train: 0.9765
Recall_train: 0.9690
F1_score_train: 0.9727
F2_score_train: 0.9705
Roc_auc_train: 0.9929
Pr_auc_train: 0.9941

Test Metrics:
Accuracy_test: 0.8365
Precision_test: 0.8854
Recall_test: 0.8746
F1_score_test: 0.8800
F2_score_test: 0.8768
Roc_auc_test: 0.9112
Pr_auc_test: 0.9591

Tuning Metrics:
hyper_parameter_tuning_best_est_score: 0.8976

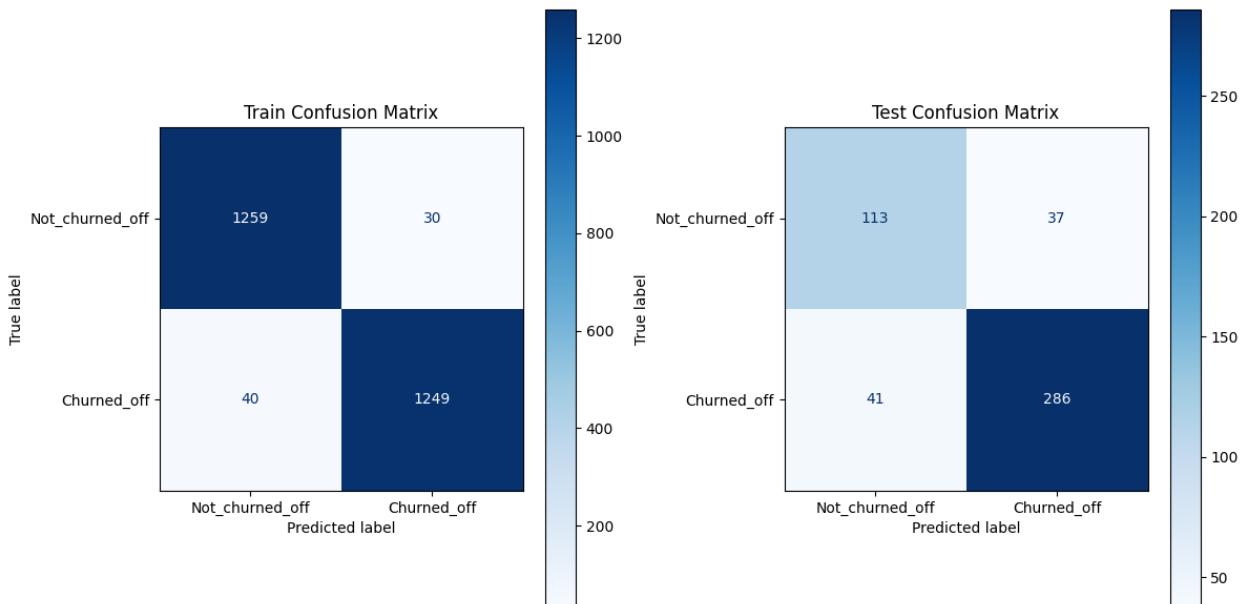
Train Classification Report:
precision    recall   f1-score   support
      0       0.97      0.98      0.97      1289
      1       0.98      0.97      0.97      1289

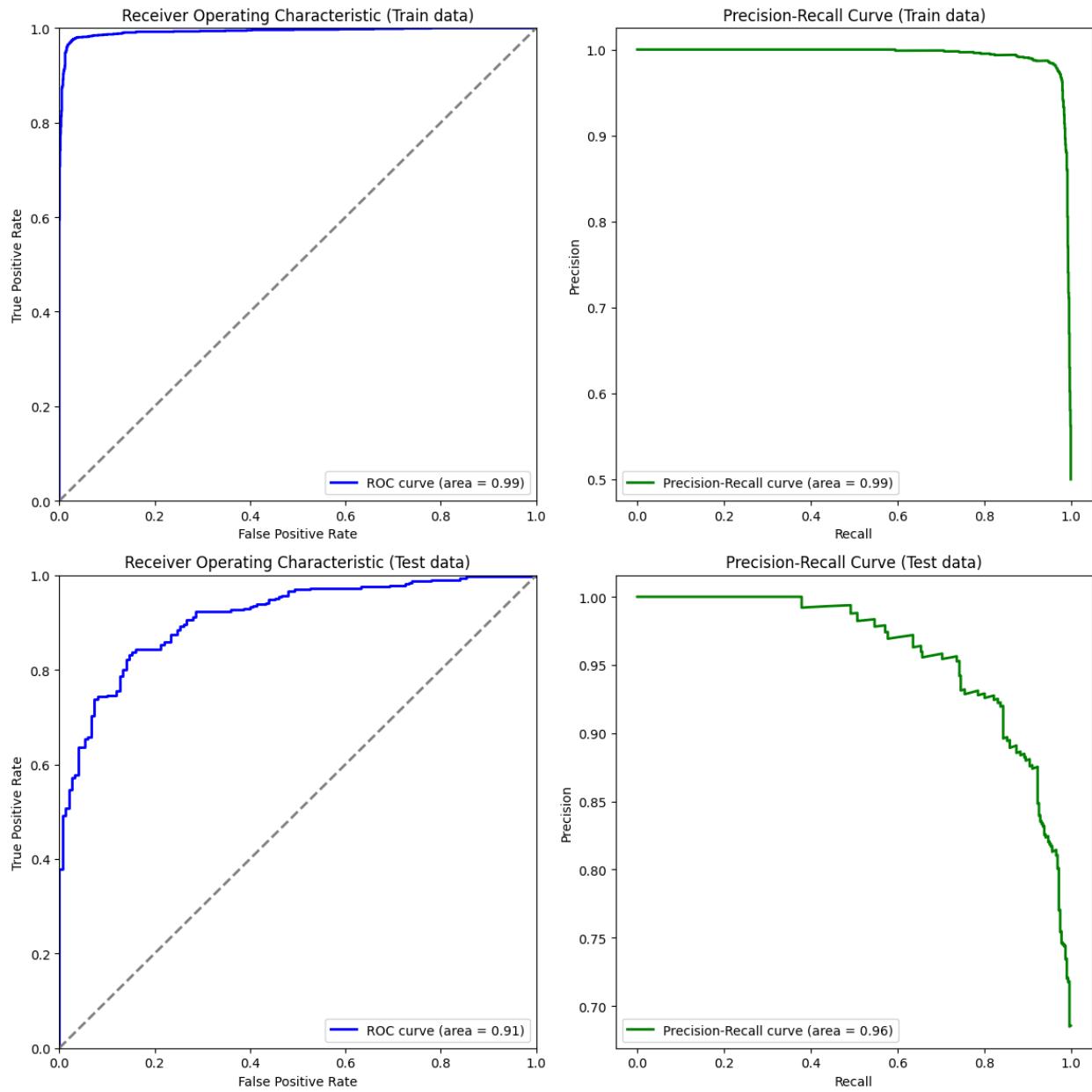
```

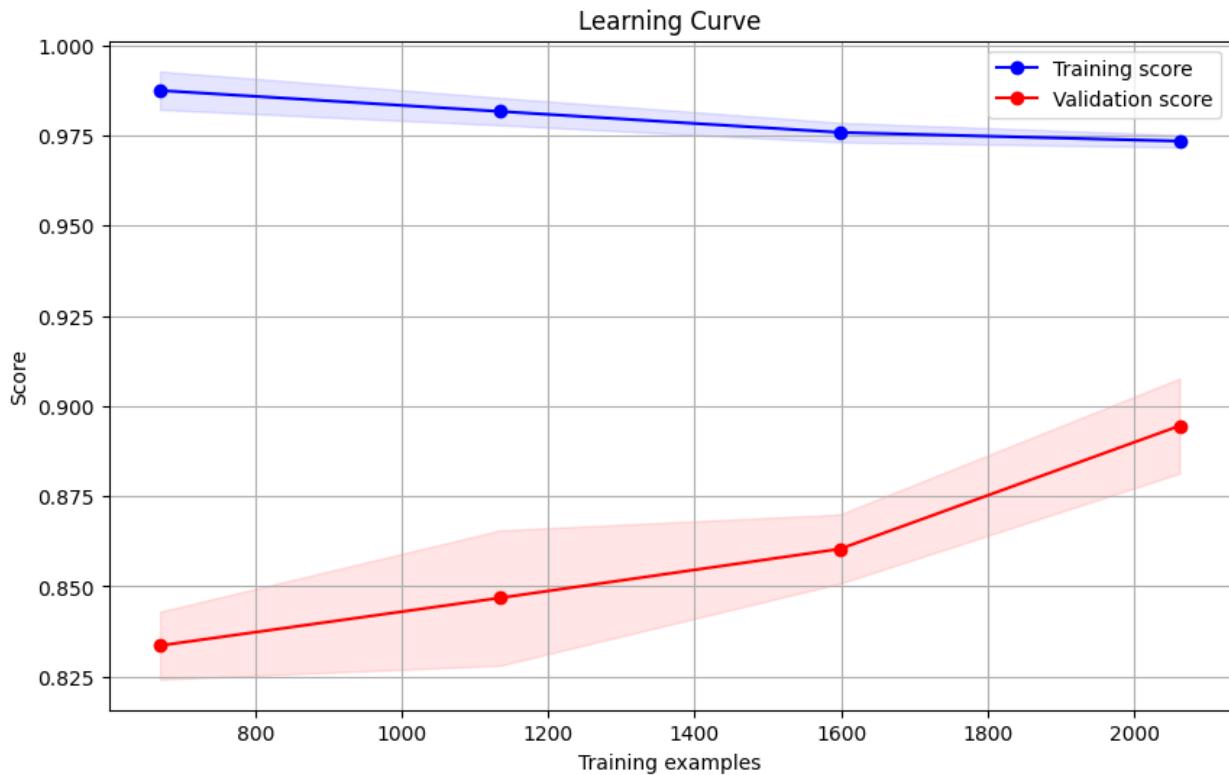
accuracy		0.97	0.97	2578
macro avg	0.97	0.97	0.97	2578
weighted avg	0.97	0.97	0.97	2578

#### Test Classification Report:

	precision	recall	f1-score	support
0	0.73	0.75	0.74	150
1	0.89	0.87	0.88	327
accuracy			0.84	477
macro avg	0.81	0.81	0.81	477
weighted avg	0.84	0.84	0.84	477







MLFLOW Logging is completed

## DECISION TREE MODEL

Imbalanced Dataset

Hyper parameter Tuning

```
# Define the parameter grid
start_tune_time = time.time()
param_dist = {
    'criterion': ['gini', 'entropy', 'log_loss'],
    'splitter': ['best', 'random'],
    'max_depth': stats.randint(2, 20), # Random integers for max_depth
    'min_samples_split': stats.randint(2, 20), # Random integers for min_samples_split
    'min_samples_leaf': stats.randint(1, 20), # Random integers for min_samples_leaf
    'min_weight_fraction_leaf': stats.uniform(0.0, 0.5), # Uniform distribution for min_weight_fraction_leaf
    'max_features': [None, 'auto', 'sqrt', 'log2'], # Options for max_features
    'max_leaf_nodes': stats.randint(2, 100), # Random integers for max_leaf_nodes
```

```

        'min_impurity_decrease': stats.uniform(0.0, 0.1), # Uniform
distribution for min_impurity_decrease
        'ccp_alpha': stats.uniform(0.0, 0.1), # Uniform distribution for
ccp_alpha
        'random_state': [42] # Fixed random state for reproducibility
    }

# Initialize the Decision Tree classifier
dtc = DecisionTreeClassifier()

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=dtc,
    param_distributions=param_dist,
    n_iter=200, # Number of parameter settings to try
    cv=5, # Number of folds in cross-validation
    verbose=1,
    random_state=42,
    n_jobs=-1 # Use all available cores
)

# Fit RandomizedSearchCV
random_search.fit(X_train_imb, y_train_imb)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning_time", tuning_time)

Fitting 5 folds for each of 200 candidates, totalling 1000 fits
Best parameters found: {'ccp_alpha': 0.01504168911035282, 'criterion':
'gini', 'max_depth': 18, 'max_features': None, 'max_leaf_nodes': 34,
'min_impurity_decrease': 0.046869315979497034, 'min_samples_leaf': 3,
'min_samples_split': 2, 'min_weight_fraction_leaf':
0.0068359824134986424, 'random_state': 42, 'splitter': 'best'}
Best score: 0.8340267992816687
Tuning_time 1.4809610843658447

```

Logging Best Decision Tree Model into MLFLOW

```

# Model details
name = "Tuned_Decision_Tree_on_Imbalanced_Dataset"
bal_type = "Imbalanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time

```

```

start_train_time = time.time()
model.fit(X_train_imb, y_train_imb)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_imb_train = model.predict(X_train_imb)
y_pred_imb_test = model.predict(X_test_imb)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df,
mlflow_logging_and_metric_printing
time_df,feature_importance_df =
log_time_and_feature_importances_df(time_df,feature_importance_df,name
,training_time,testing_time,model,ola_features,bal_type,tuning_time)
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_imb,y_t
rain_imb,X_test_imb,y_test_imb,y_pred_imb_train,y_pred_imb_test,tuning
_score,**params)

Model: Tuned_Decision_Tree_on_Imbalanced_Dataset
params: {'ccp_alpha': 0.01504168911035282, 'criterion': 'gini',
'max_depth': 18, 'max_features': None, 'max_leaf_nodes': 34,
'min_impurity_decrease': 0.046869315979497034, 'min_samples_leaf': 3,
'min_samples_split': 2, 'min_weight_fraction_leaf':
0.0068359824134986424, 'random_state': 42, 'splitter': 'best'}
Training Time: 0.0095 seconds
Testing Time: 0.0020 seconds
Tuning Time: 1.4810 seconds
Train Metrics:
Accuracy_train: 0.8451
Precision_train: 0.8596
Recall_train: 0.9216
F1_score_train: 0.8896
F2_score_train: 0.9085
Roc_auc_train: 0.8062
Pr_auc_train: 0.9177

```

```
Test Metrics:  
Accuracy_test: 0.8658  
Precision_test: 0.8663  
Recall_test: 0.9511  
F1_score_test: 0.9067  
F2_score_test: 0.9328  
Roc_auc_test: 0.8184  
Pr_auc_test: 0.9261
```

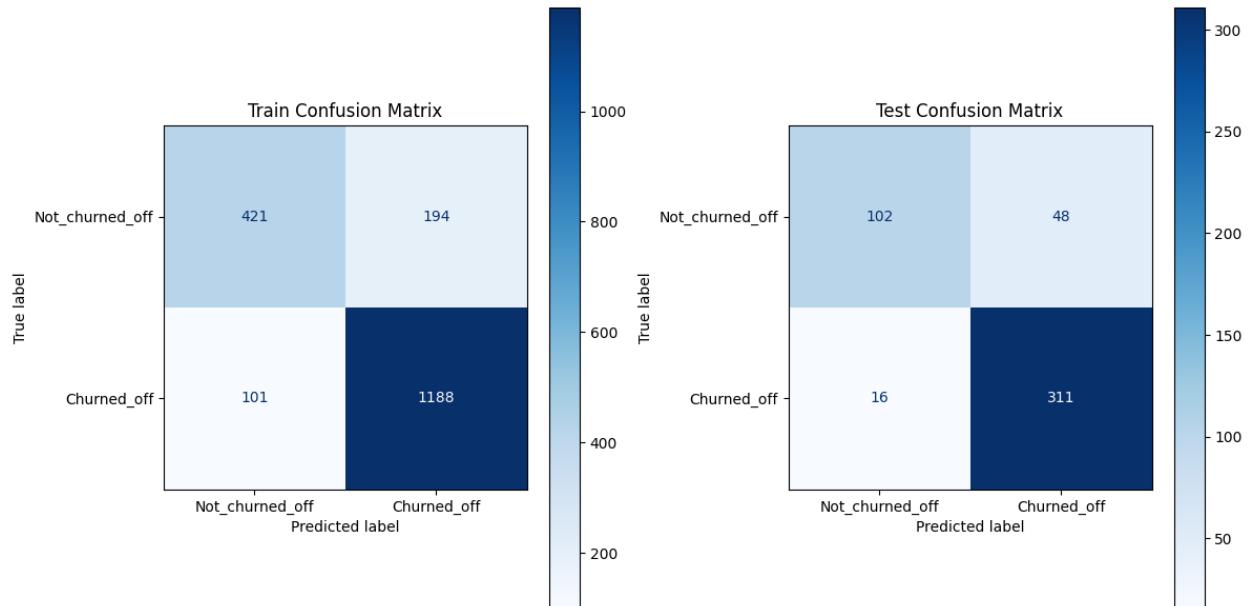
```
Tuning Metrics:  
hyper_parameter_tuning_best_est_score: 0.8340
```

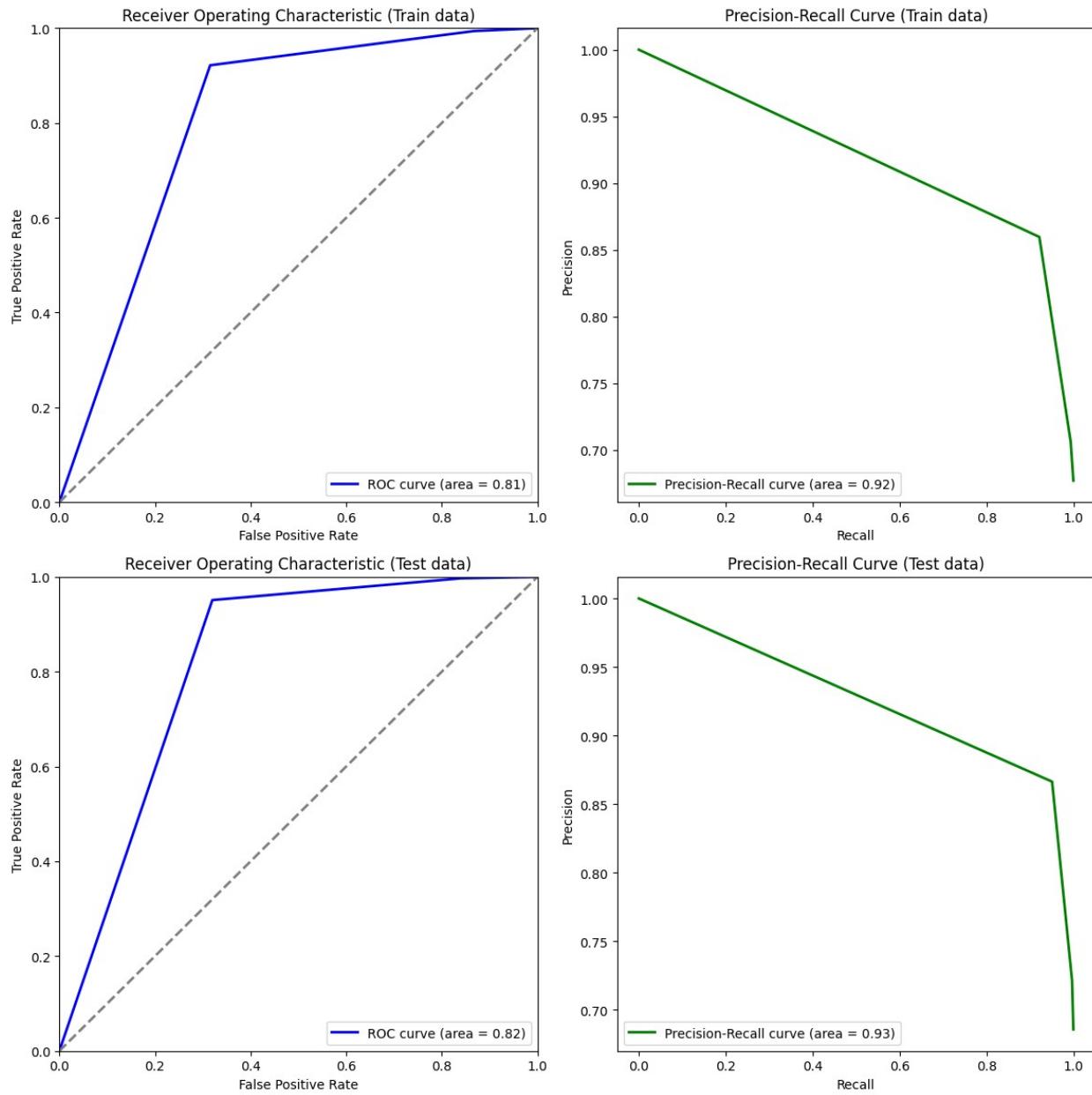
```
Train Classification Report:
```

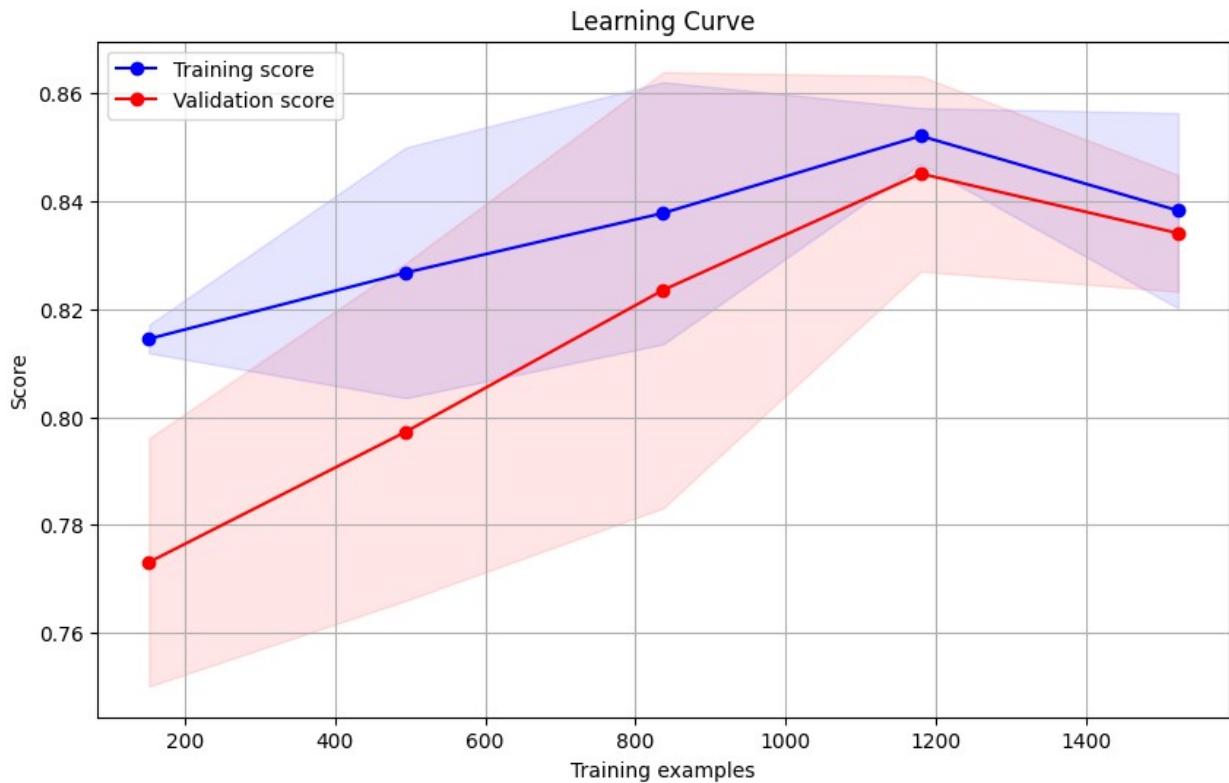
	precision	recall	f1-score	support
0	0.81	0.68	0.74	615
1	0.86	0.92	0.89	1289
accuracy			0.85	1904
macro avg	0.83	0.80	0.82	1904
weighted avg	0.84	0.85	0.84	1904

```
Test Classification Report:
```

	precision	recall	f1-score	support
0	0.86	0.68	0.76	150
1	0.87	0.95	0.91	327
accuracy			0.87	477
macro avg	0.87	0.82	0.83	477
weighted avg	0.87	0.87	0.86	477







MLFL0W Logging is completed

## Balanced Dataset

### Hyper parameter Tuning

```
# Define the parameter grid
start_tune_time = time.time()
param_dist = {
    'criterion': ['gini', 'entropy', 'log_loss'],
    'splitter': ['best', 'random'],
    'max_depth': stats.randint(2, 20), # Random integers for max_depth
    'min_samples_split': stats.randint(2, 20), # Random integers for min_samples_split
    'min_samples_leaf': stats.randint(1, 20), # Random integers for min_samples_leaf
    'min_weight_fraction_leaf': stats.uniform(0.0, 0.5), # Uniform distribution for min_weight_fraction_leaf
    'max_features': [None, 'auto', 'sqrt', 'log2'], # Options for max_features
    'max_leaf_nodes': stats.randint(2, 100), # Random integers for max_leaf_nodes
    'min_impurity_decrease': stats.uniform(0.0, 0.1), # Uniform distribution for min_impurity_decrease
```

```

    'ccp_alpha': stats.uniform(0.0, 0.1), # Uniform distribution for
ccp_alpha
    'random_state': [42] # Fixed random state for reproducibility
}

# Initialize the Decision Tree classifier
dtc = DecisionTreeClassifier()

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=dtc,
    param_distributions=param_dist,
    n_iter=200, # Number of parameter settings to try
    cv=5, # Number of folds in cross-validation
    verbose=1,
    random_state=42,
    n_jobs=-1 # Use all available cores
)

# Fit RandomizedSearchCV
random_search.fit(X_train_bal, y_train_bal)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning_time", tuning_time)

Fitting 5 folds for each of 200 candidates, totalling 1000 fits
Best parameters found: {'ccp_alpha': 0.011483682473920355,
'criterion': 'entropy', 'max_depth': 15, 'max_features': 'log2',
'max_leaf_nodes': 39, 'min_impurity_decrease': 0.05812382214226123,
'min_samples_leaf': 11, 'min_samples_split': 5,
'min_weight_fraction_leaf': 0.16032109607975714, 'random_state': 42,
'splitter': 'best'}
Best score: 0.8153653947467449
Tuning_time 1.478015422821045

```

## Logging Best Decision Tree Model into MLFLOW

```

# Model details
name = "Tuned_Decision_Tree_on_Balanced_Dataset"
bal_type = "Balanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time
start_train_time = time.time()

```

```

model.fit(X_train_bal, y_train_bal)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_bal_train = model.predict(X_train_bal)
y_pred_bal_test = model.predict(X_test_bal)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df,
mlflow_logging_and_metric_printing
time_df,feature_importance_df =
log_time_and_feature_importances_df(time_df,feature_importance_df,name
,training_time,testing_time,model,ola_features,bal_type,tuning_time)
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_bal,y_t
rain_bal,X_test_bal,y_test_bal,y_pred_bal_train,y_pred_bal_test,tuning
_score,**params)

Model: Tuned_Decision_Tree_on_Balanced_Dataset
params: {'ccp_alpha': 0.011483682473920355, 'criterion': 'entropy',
'max_depth': 15, 'max_features': 'log2', 'max_leaf_nodes': 39,
'min_impurity_decrease': 0.05812382214226123, 'min_samples_leaf': 11,
'min_samples_split': 5, 'min_weight_fraction_leaf':
0.16032109607975714, 'random_state': 42, 'splitter': 'best'}
Training Time: 0.0068 seconds
Testing Time: 0.0016 seconds
Tuning Time: 1.4780 seconds
Train Metrics:
Accuracy_train: 0.8169
Precision_train: 0.9397
Recall_train: 0.6773
F1_score_train: 0.7872
F2_score_train: 0.7173
Roc_auc_train: 0.8781
Pr_auc_train: 0.9027

Test Metrics:

```

```
Accuracy_test: 0.7505
Precision_test: 0.9561
Recall_test: 0.6667
F1_score_test: 0.7856
F2_score_test: 0.7096
Roc_auc_test: 0.8654
Pr_auc_test: 0.9402
```

Tuning Metrics:

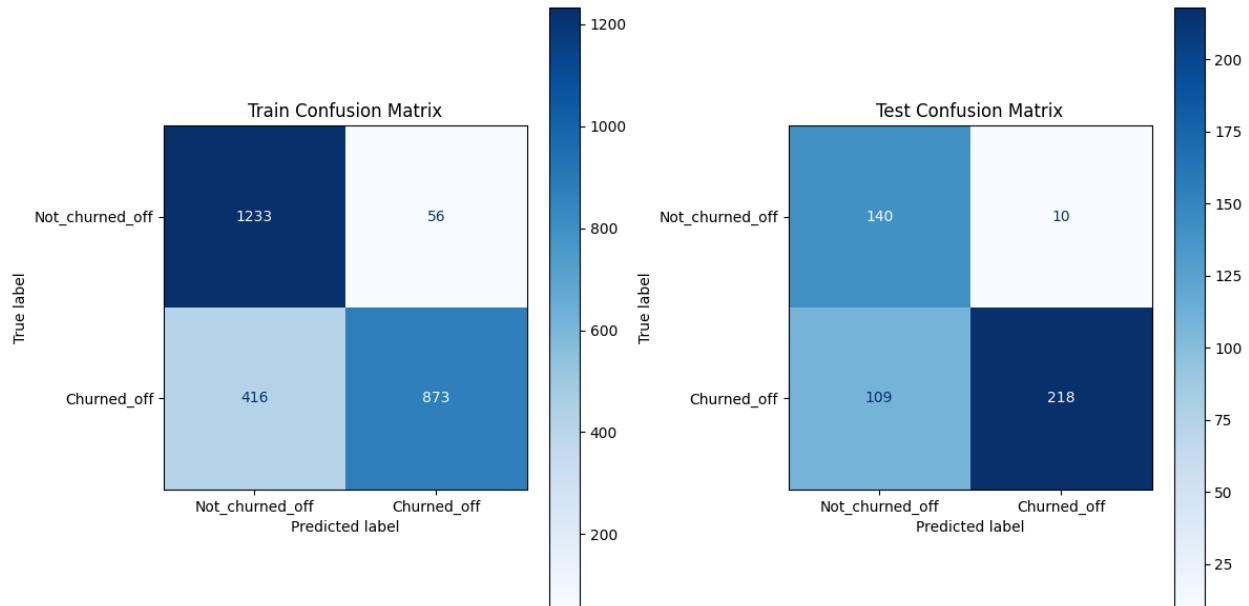
```
hyper_parameter_tuning_best_est_score: 0.8154
```

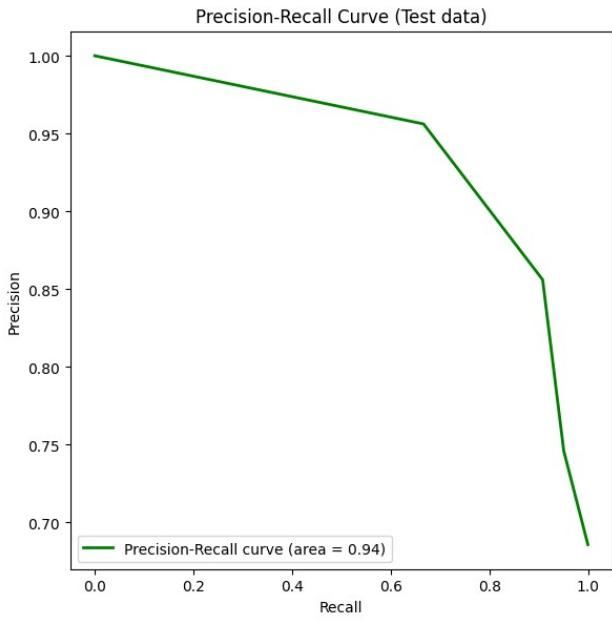
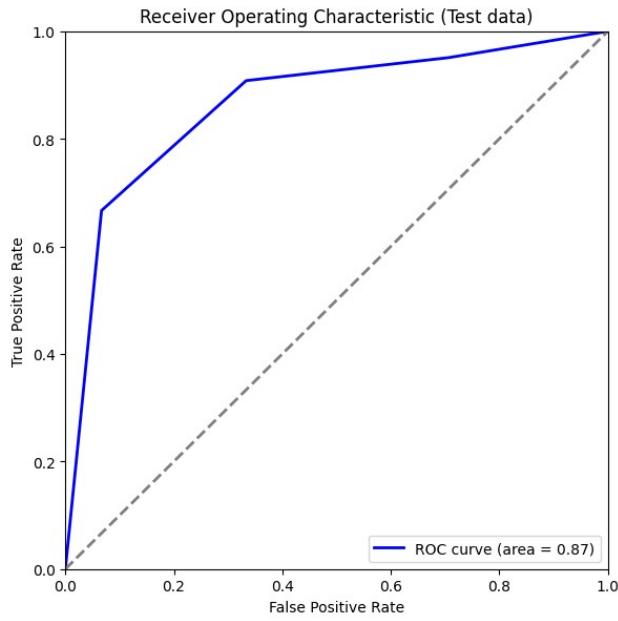
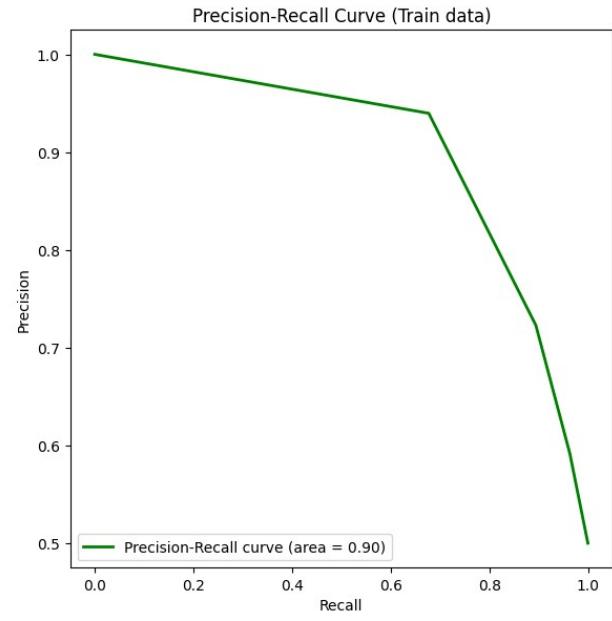
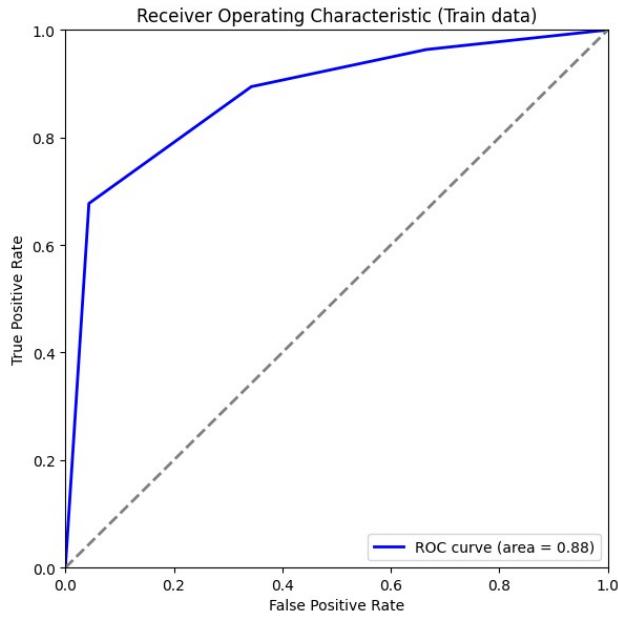
Train Classification Report:

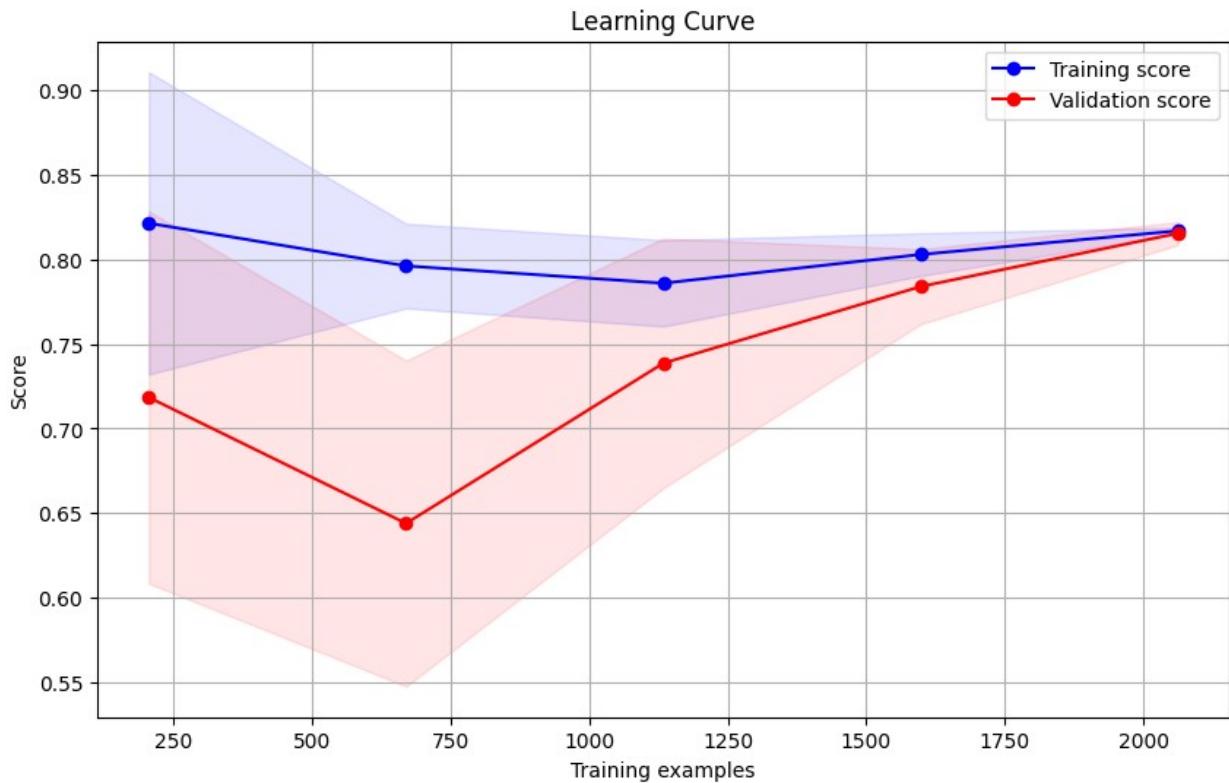
	precision	recall	f1-score	support
0	0.75	0.96	0.84	1289
1	0.94	0.68	0.79	1289
accuracy			0.82	2578
macro avg	0.84	0.82	0.81	2578
weighted avg	0.84	0.82	0.81	2578

Test Classification Report:

	precision	recall	f1-score	support
0	0.56	0.93	0.70	150
1	0.96	0.67	0.79	327
accuracy			0.75	477
macro avg	0.76	0.80	0.74	477
weighted avg	0.83	0.75	0.76	477







MLFL0W Logging is completed

## RANDOM FOREST MODEL

### Imbalanced Dataset

#### Hyper parameter Tuning

```
# Define the parameter grid
start_tune_time = time.time()
param_dist = {
    'n_estimators': stats.randint(50, 1000), # Random integers for n_estimators
    'criterion': ['gini', 'entropy', 'log_loss'],
    'max_depth': stats.randint(2, 20), # Random integers for max_depth
    'min_samples_split': stats.randint(2, 20), # Random integers for min_samples_split
    'min_samples_leaf': stats.randint(1, 20), # Random integers for min_samples_leaf
    'min_weight_fraction_leaf': stats.uniform(0.0, 0.5), # Uniform distribution for min_weight_fraction_leaf
    'max_features': ['auto', 'sqrt', 'log2', None], # Options for max_features
    'max_leaf_nodes': stats.randint(2, 100), # Random integers for max_leaf_nodes
}
```

```

max_leaf_nodes
    'min_impurity_decrease': stats.uniform(0.0, 0.1), # Uniform
distribution for min_impurity_decrease
    'bootstrap': [True, False], # Whether bootstrap samples are used
when building trees
    'oob_score': [True, False], # Whether to use out-of-bag samples
to estimate the generalization accuracy
    'ccp_alpha': stats.uniform(0.0, 0.1), # Uniform distribution for
ccp_alpha
    'random_state': [42] # Fixed random state for reproducibility
}

# Initialize the RandomForest classifier
rfc = RandomForestClassifier(class_weight="balanced")

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=rfc,
    param_distributions=param_dist,
    n_iter=200, # Number of parameter settings to try
    cv=5, # Number of folds in cross-validation
    verbose=1,
    random_state=42,
    n_jobs=-1 # Use all available cores
)

# Fit RandomizedSearchCV
random_search.fit(X_train_imb, y_train_imb)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning_time", tuning_time)

Fitting 5 folds for each of 200 candidates, totalling 1000 fits
Best parameters found: {'bootstrap': True, 'ccp_alpha':
0.028792961647572612, 'criterion': 'log_loss', 'max_depth': 13,
'max_features': 'sqrt', 'max_leaf_nodes': 14, 'min_impurity_decrease':
0.013911619414306187, 'min_samples_leaf': 18, 'min_samples_split': 2,
'min_weight_fraction_leaf': 0.009055091910420254, 'n_estimators': 877,
'oob_score': True, 'random_state': 42}
Best score: 0.8833954966155545
Tuning_time 191.6145143508911

rfc_imb = random_search.best_estimator_

```

## Logging Best Random Forest Model into MLFLOW

```
# Model details
name = "Tuned_Random_Forest_on_Imbalanced_Dataset"
bal_type = "Imbalanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time
start_train_time = time.time()
model.fit(X_train_imb, y_train_imb)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_imb_train = model.predict(X_train_imb)
y_pred_imb_test = model.predict(X_test_imb)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df,
mlflow_logging_and_metric_printing
time_df,feature_importance_df =
log_time_and_feature_importances_df(time_df,feature_importance_df,name
,training_time,testing_time,model,ola_features,bal_type,tuning_time)
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_imb,y_t
rain_imb,X_test_imb,y_test_imb,y_pred_imb_train,y_pred_imb_test,tuning
_score,**params)

Model: Tuned_Random_Forest_on_Imbalanced_Dataset
params: {'bootstrap': True, 'ccp_alpha': 0.028792961647572612,
'criterion': 'log_loss', 'max_depth': 13, 'max_features': 'sqrt',
'max_leaf_nodes': 14, 'min_impurity_decrease': 0.013911619414306187,
'min_samples_leaf': 18, 'min_samples_split': 2,
'min_weight_fraction_leaf': 0.009055091910420254, 'n_estimators': 877,
'oob_score': True, 'random_state': 42}
Training Time: 3.7044 seconds
Testing Time: 0.2037 seconds
Tuning Time: 191.6145 seconds
```

Train Metrics:  
Accuracy\_train: 0.8929  
Precision\_train: 0.9242  
Recall\_train: 0.9170  
F1\_score\_train: 0.9206  
F2\_score\_train: 0.9184  
Roc\_auc\_train: 0.9484  
Pr\_auc\_train: 0.9735

Test Metrics:  
Accuracy\_test: 0.8847  
Precision\_test: 0.9224  
Recall\_test: 0.9083  
F1\_score\_test: 0.9153  
F2\_score\_test: 0.9110  
Roc\_auc\_test: 0.9424  
Pr\_auc\_test: 0.9683

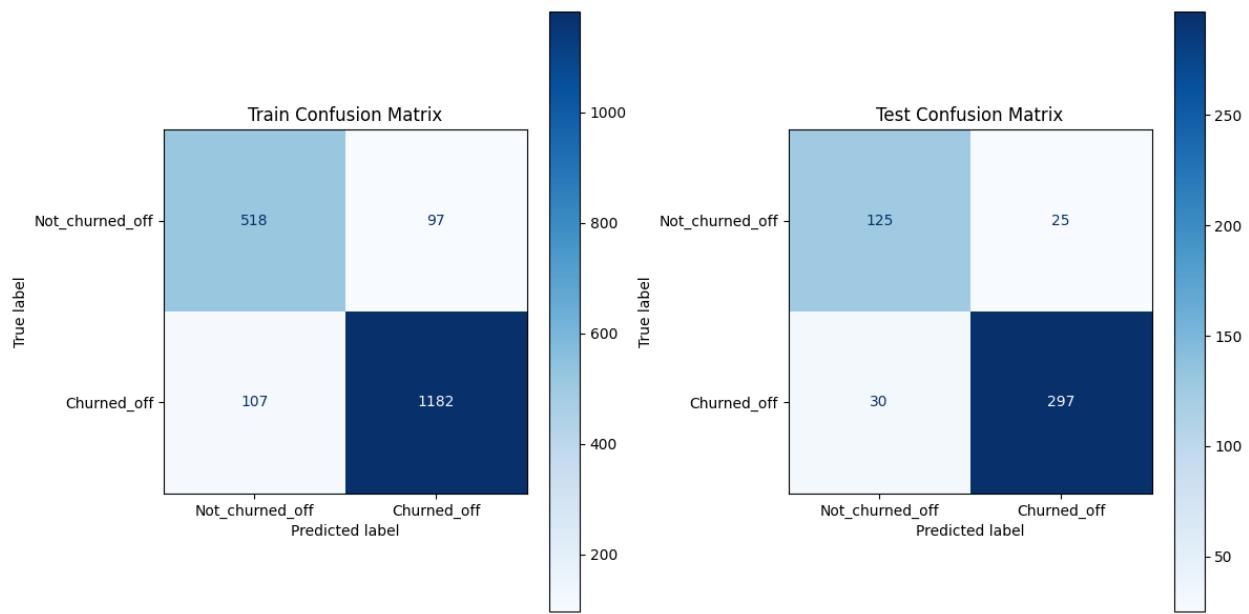
Tuning Metrics:  
hyper\_parameter\_tuning\_best\_est\_score: 0.8834

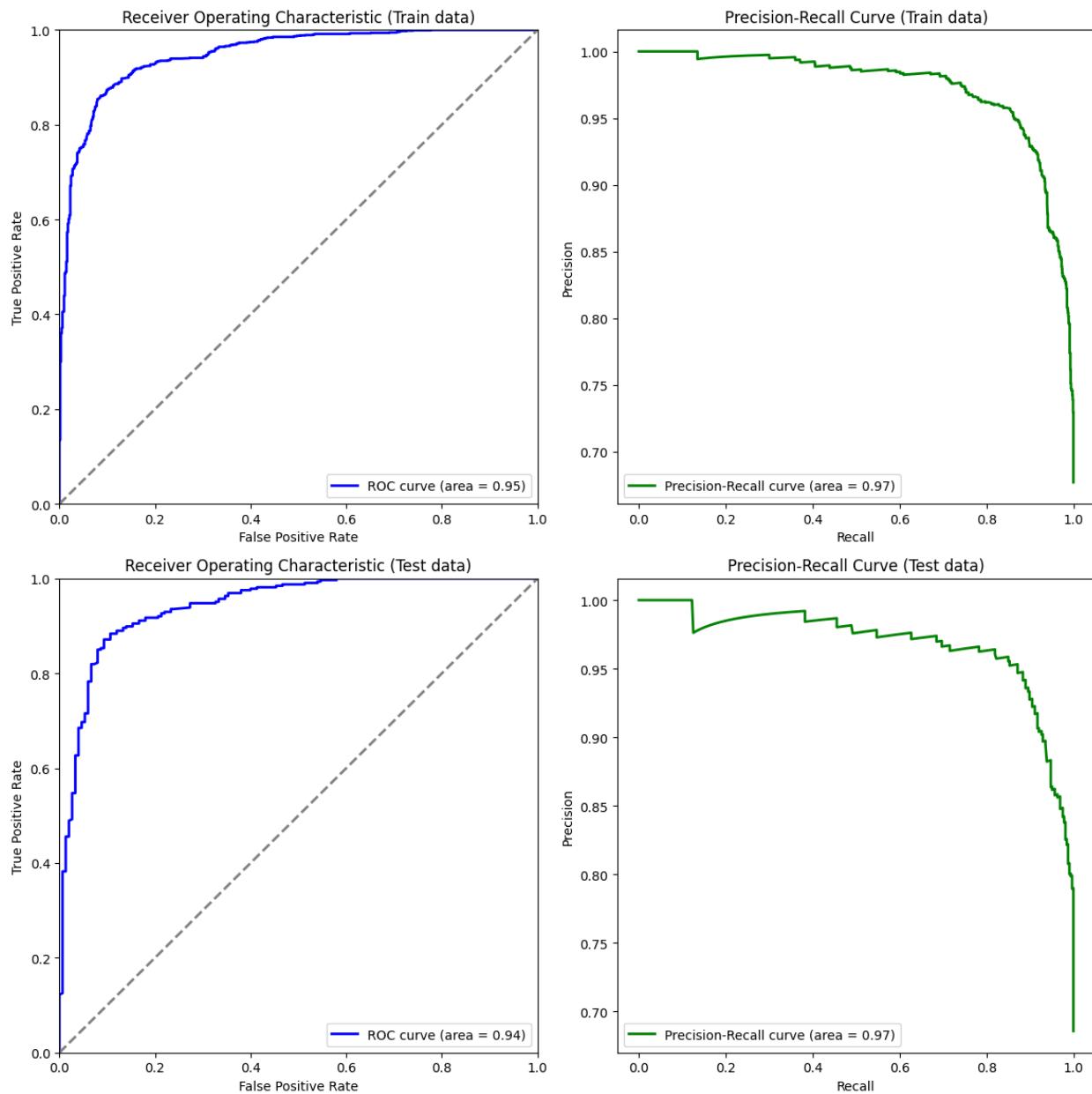
Train Classification Report:

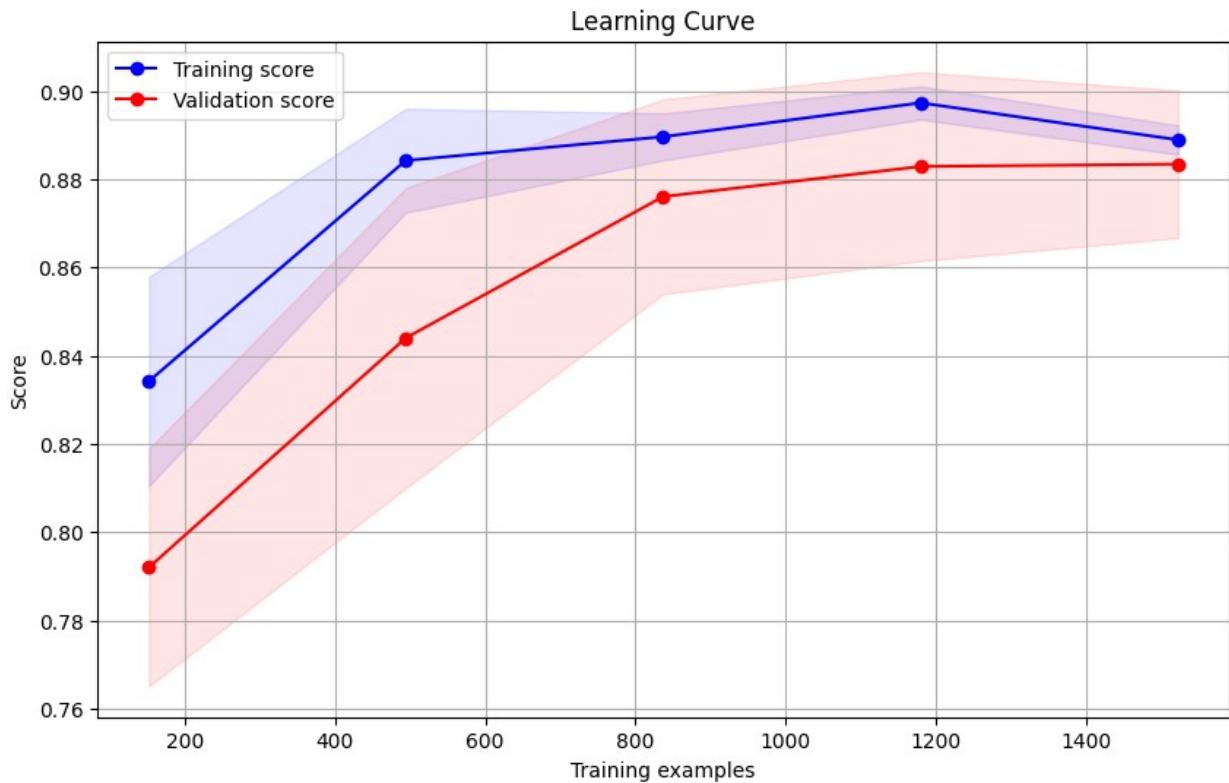
	precision	recall	f1-score	support
0	0.83	0.84	0.84	615
1	0.92	0.92	0.92	1289
accuracy			0.89	1904
macro avg	0.88	0.88	0.88	1904
weighted avg	0.89	0.89	0.89	1904

Test Classification Report:

	precision	recall	f1-score	support
0	0.81	0.83	0.82	150
1	0.92	0.91	0.92	327
accuracy			0.88	477
macro avg	0.86	0.87	0.87	477
weighted avg	0.89	0.88	0.89	477







MLFL0W Logging is completed

## Balanced Dataset

### Hyper parameter Tuning

```
# Define the parameter grid
start_tune_time = time.time()
param_dist = {
    'n_estimators': stats.randint(50, 1000), # Random integers for n_estimators
    'criterion': ['gini', 'entropy', 'log_loss'],
    'max_depth': stats.randint(2, 20), # Random integers for max_depth
    'min_samples_split': stats.randint(2, 20), # Random integers for min_samples_split
    'min_samples_leaf': stats.randint(1, 20), # Random integers for min_samples_leaf
    'min_weight_fraction_leaf': stats.uniform(0.0, 0.5), # Uniform distribution for min_weight_fraction_leaf
    'max_features': ['auto', 'sqrt', 'log2', None], # Options for max_features
    'max_leaf_nodes': stats.randint(2, 100), # Random integers for max_leaf_nodes
    'min_impurity_decrease': stats.uniform(0.0, 0.1), # Uniform
```

```

distribution for min_impurity_decrease
    'bootstrap': [True, False], # Whether bootstrap samples are used
when building trees
    'oob_score': [True, False], # Whether to use out-of-bag samples
to estimate the generalization accuracy
    'ccp_alpha': stats.uniform(0.0, 0.1), # Uniform distribution for
ccp_alpha
    'random_state': [42] # Fixed random state for reproducibility
}

# Initialize the RandomForest classifier
rfc = RandomForestClassifier(class_weight="balanced")

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=rfc,
    param_distributions=param_dist,
    n_iter=200, # Number of parameter settings to try
    cv=5, # Number of folds in cross-validation
    verbose=1,
    random_state=42,
    n_jobs=-1 # Use all available cores
)

# Fit RandomizedSearchCV
random_search.fit(X_train_bal, y_train_bal)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning_time", tuning_time)

Fitting 5 folds for each of 200 candidates, totalling 1000 fits
Best parameters found: {'bootstrap': True, 'ccp_alpha':
0.028792961647572612, 'criterion': 'log_loss', 'max_depth': 13,
'max_features': 'sqrt', 'max_leaf_nodes': 14, 'min_impurity_decrease':
0.013911619414306187, 'min_samples_leaf': 18, 'min_samples_split': 2,
'min_weight_fraction_leaf': 0.009055091910420254, 'n_estimators': 877,
'oob_score': True, 'random_state': 42}
Best score: 0.852979604124332
Tuning_time 209.51067566871643

rfc_bal = random_search.best_estimator_

```

## Logging Best Random Forest Model into MLFLOW

```
# Model details
name = "Tuned_Random_Forest_on_Balanced_Dataset"
bal_type = "Balanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time
start_train_time = time.time()
model.fit(X_train_bal, y_train_bal)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_bal_train = model.predict(X_train_bal)
y_pred_bal_test = model.predict(X_test_bal)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df,
mlflow_logging_and_metric_printing
time_df,feature_importance_df =
log_time_and_feature_importances_df(time_df,feature_importance_df,name
,training_time,testing_time,model,ola_features,bal_type,tuning_time)
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_bal,y_t
rain_bal,X_test_bal,y_test_bal,y_pred_bal_train,y_pred_bal_test,tuning
_score,**params)

Model: Tuned_Random_Forest_on_Balanced_Dataset
params: {'bootstrap': True, 'ccp_alpha': 0.028792961647572612,
'criterion': 'log_loss', 'max_depth': 13, 'max_features': 'sqrt',
'max_leaf_nodes': 14, 'min_impurity_decrease': 0.013911619414306187,
'min_samples_leaf': 18, 'min_samples_split': 2,
'min_weight_fraction_leaf': 0.009055091910420254, 'n_estimators': 877,
'oob_score': True, 'random_state': 42}
Training Time: 4.0574 seconds
Testing Time: 0.2175 seconds
Tuning Time: 209.5107 seconds
```

Train Metrics:  
Accuracy\_train: 0.8638  
Precision\_train: 0.8271  
Recall\_train: 0.9201  
F1\_score\_train: 0.8711  
F2\_score\_train: 0.8998  
Roc\_auc\_train: 0.9412  
Pr\_auc\_train: 0.9453

Test Metrics:  
Accuracy\_test: 0.8805  
Precision\_test: 0.9141  
Recall\_test: 0.9113  
F1\_score\_test: 0.9127  
F2\_score\_test: 0.9119  
Roc\_auc\_test: 0.9335  
Pr\_auc\_test: 0.9653

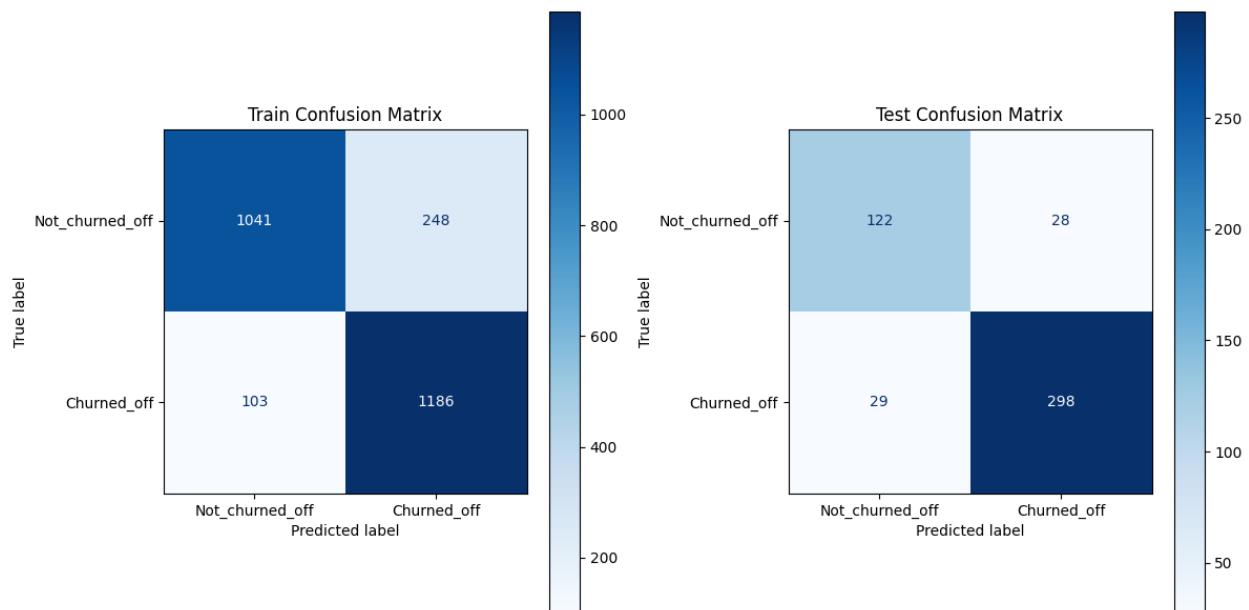
Tuning Metrics:  
hyper\_parameter\_tuning\_best\_est\_score: 0.8530

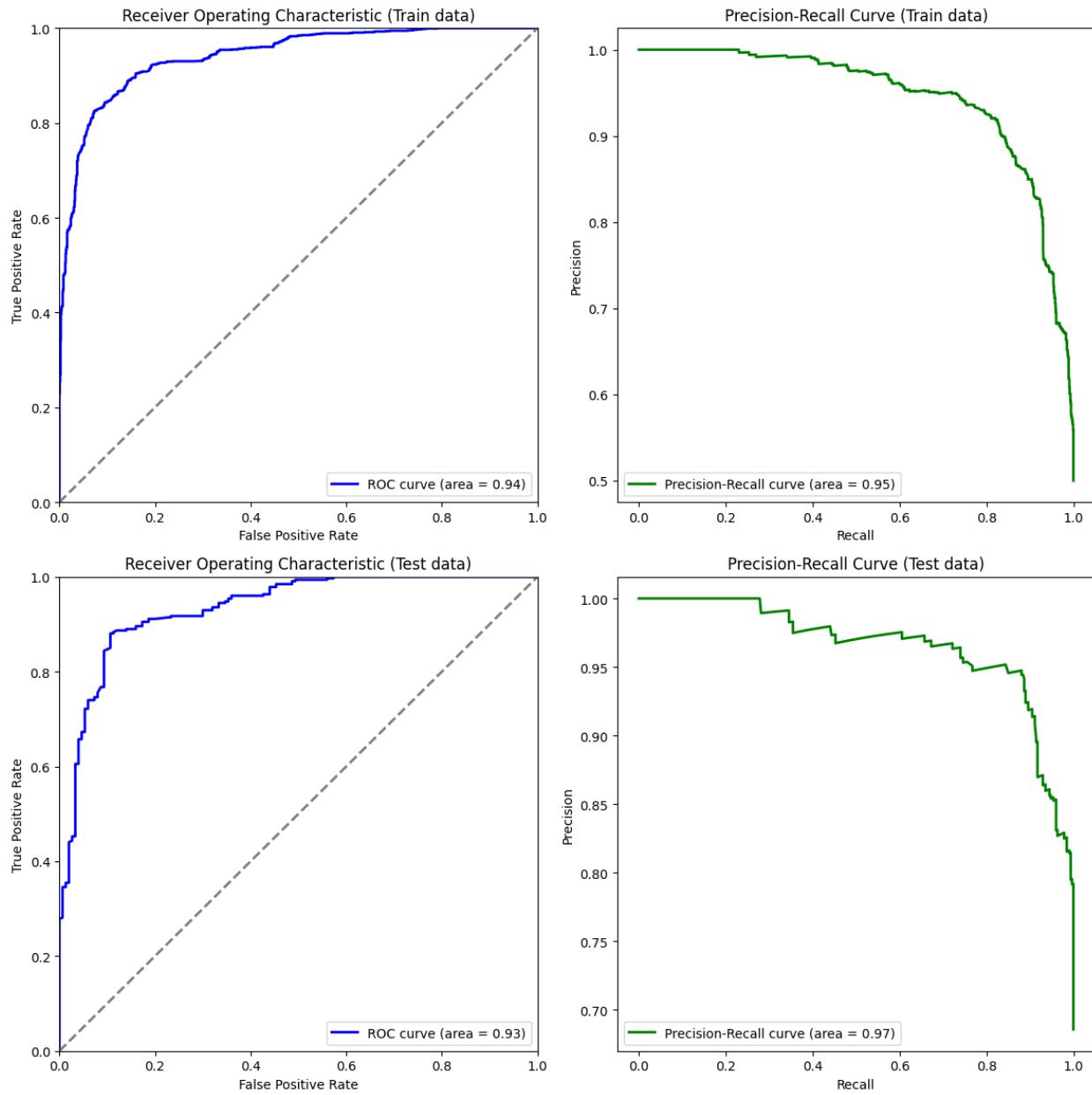
Train Classification Report:

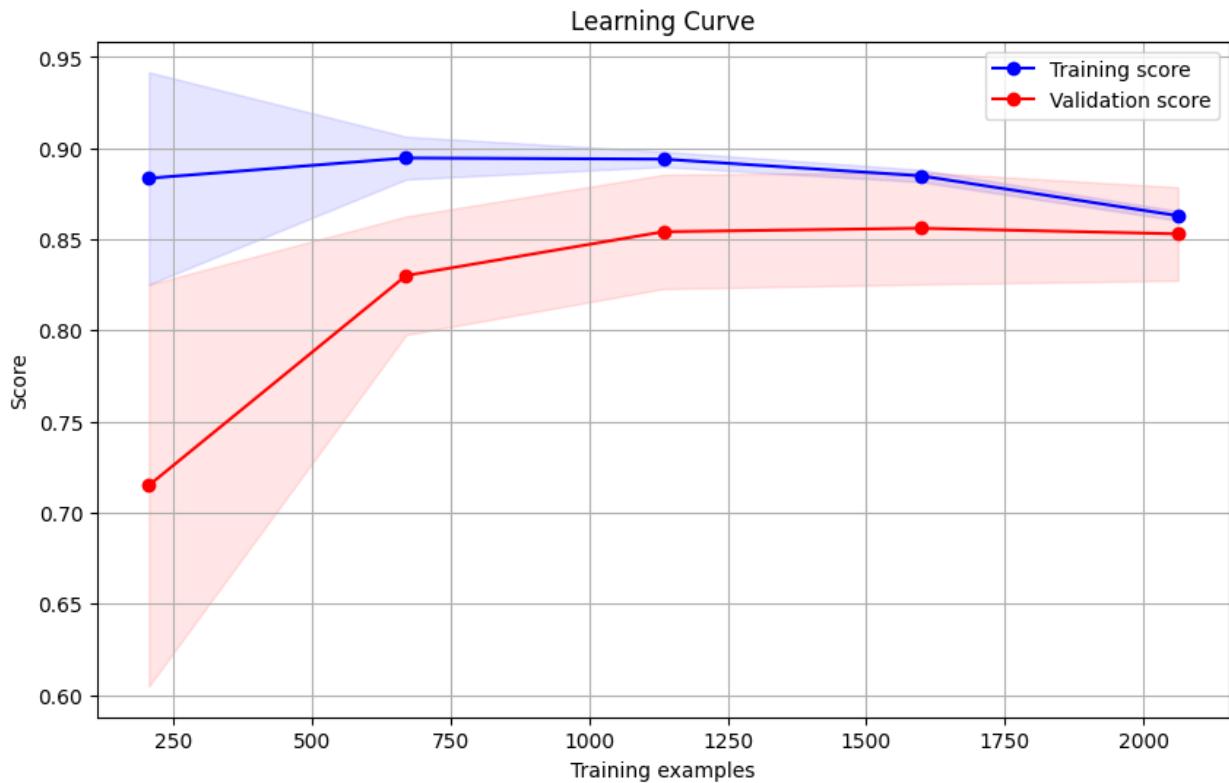
	precision	recall	f1-score	support
0	0.91	0.81	0.86	1289
1	0.83	0.92	0.87	1289
accuracy			0.86	2578
macro avg	0.87	0.86	0.86	2578
weighted avg	0.87	0.86	0.86	2578

Test Classification Report:

	precision	recall	f1-score	support
0	0.81	0.81	0.81	150
1	0.91	0.91	0.91	327
accuracy			0.88	477
macro avg	0.86	0.86	0.86	477
weighted avg	0.88	0.88	0.88	477







MLFL0W Logging is completed

## BAGGING CLASSIFIER ON BEST\_RF MODEL

### Imbalanced Dataset

#### Hyper parameter Tuning

```
rfc_imb

RandomForestClassifier(ccp_alpha=0.028792961647572612,
class_weight='balanced',
                     criterion='log_loss', max_depth=13,
max_leaf_nodes=14,
                     min_impurity_decrease=0.013911619414306187,
                     min_samples_leaf=18,
                     min_weight_fraction_leaf=0.009055091910420254,
                     n_estimators=877, oob_score=True,
random_state=42)

# Base RandomForest model with the provided parameters
base_rf =
RandomForestClassifier(class_weight='balanced', criterion='log_loss',
max_depth=20, max_leaf_nodes=20,
                     n_estimators=5, oob_score=True,
```

```

random_state=42)

# Define the parameter grid for Bagging
start_tune_time = time.time()
param_dist = {
    'n_estimators': stats.randint(10, 500), # Number of base
estimators in the ensemble
    'max_samples': stats.uniform(0.1, 1.0), # Fraction of samples to
draw from X to train each base estimator
    'max_features': stats.uniform(0.1, 1.0), # Fraction of features
to draw from X to train each base estimator
    'bootstrap': [True, False], # Whether samples are drawn with
replacement
    'bootstrap_features': [True, False], # Whether features are drawn
with replacement
    'random_state': [42] # Fixed random state for reproducibility
}

# Initialize the Bagging classifier with the RandomForest as the base
estimator
bagging_clf = BaggingClassifier(base_rf, n_jobs=-1)

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=bagging_clf,
    param_distributions=param_dist,
    n_iter=200, # Number of parameter settings to try
    cv=5, # Number of folds in cross-validation
    verbose=5,
    random_state=42,
    n_jobs=-1 # Use all available cores
)

# Fit RandomizedSearchCV
random_search.fit(X_train_imb, y_train_imb)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning_time", tuning_time)

Fitting 5 folds for each of 200 candidates, totalling 1000 fits
Best parameters found: {'bootstrap': False, 'bootstrap_features':
False, 'max_features': 0.9631385219890038, 'max_samples':
0.9803599686384168, 'n_estimators': 154, 'random_state': 42}
Best score: 0.9043983975687249
Tuning_time 860.3156924247742

```

## Logging Best Bagging RF Model into MLFLOW

```
# Model details
name = "Tuned_Bagging_RF_on_Imbalanced_Dataset"
bal_type = "Imbalanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time
start_train_time = time.time()
model.fit(X_train_imb, y_train_imb)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_imb_train = model.predict(X_train_imb)
y_pred_imb_test = model.predict(X_test_imb)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df,
mlflow_logging_and_metric_printing
time_df,feature_importance_df =
log_time_and_feature_importances_df(time_df,feature_importance_df,name
,training_time,testing_time,model,ola_features,bal_type,tuning_time)
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_imb,y_t
rain_imb,X_test_imb,y_test_imb,y_pred_imb_train,y_pred_imb_test,tuning
_score,**params)

Model: Tuned_Bagging_RF_on_Imbalanced_Dataset
params: {'bootstrap': False, 'bootstrap_features': False,
'max_features': 0.9631385219890038, 'max_samples': 0.9803599686384168,
'n_estimators': 154, 'random_state': 42}
Training Time: 0.8501 seconds
Testing Time: 0.4739 seconds
Tuning Time: 860.3157 seconds
Train Metrics:
Accuracy_train: 0.9181
Precision_train: 0.9381
```

```
Recall_train: 0.9410
F1_score_train: 0.9396
F2_score_train: 0.9405
Roc_auc_train: 0.9724
Pr_auc_train: 0.9863
```

```
Test Metrics:
Accuracy_test: 0.9078
Precision_test: 0.9224
Recall_test: 0.9450
F1_score_test: 0.9335
F2_score_test: 0.9404
Roc_auc_test: 0.9518
Pr_auc_test: 0.9716
```

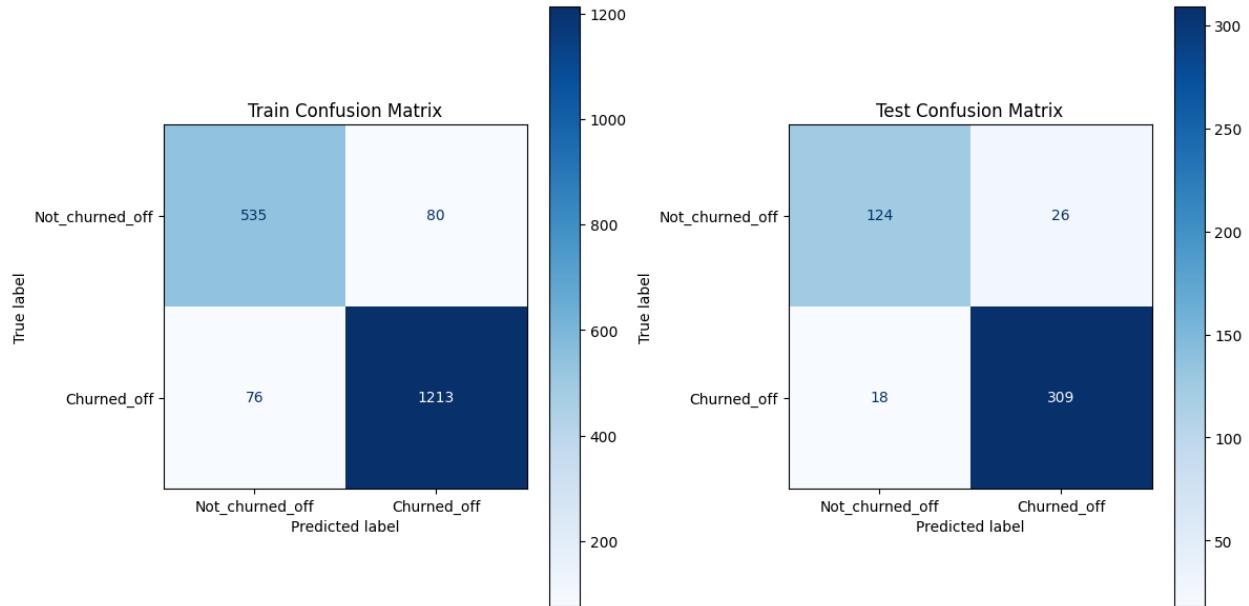
```
Tuning Metrics:
hyper_parameter_tuning_best_est_score: 0.9044
```

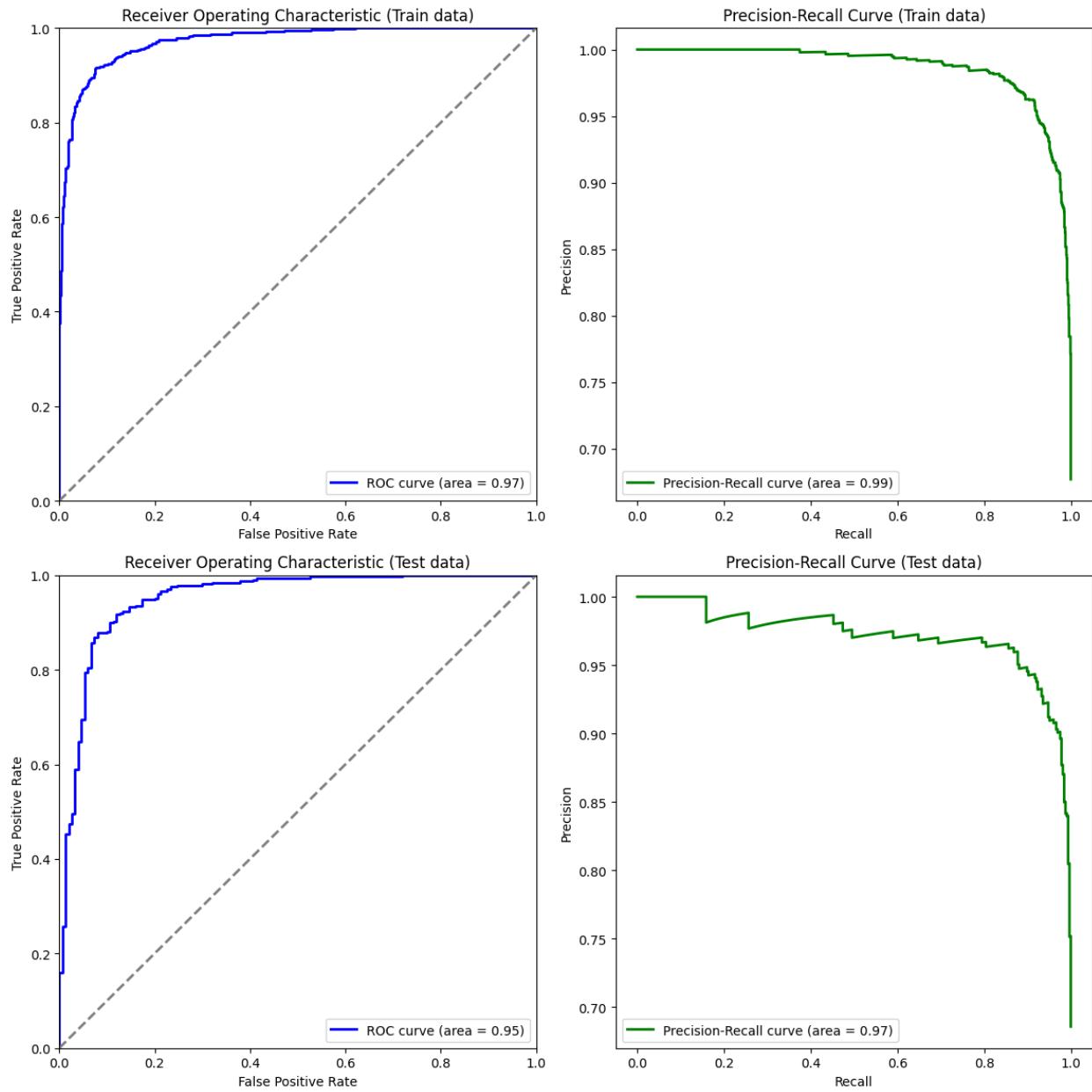
```
Train Classification Report:
```

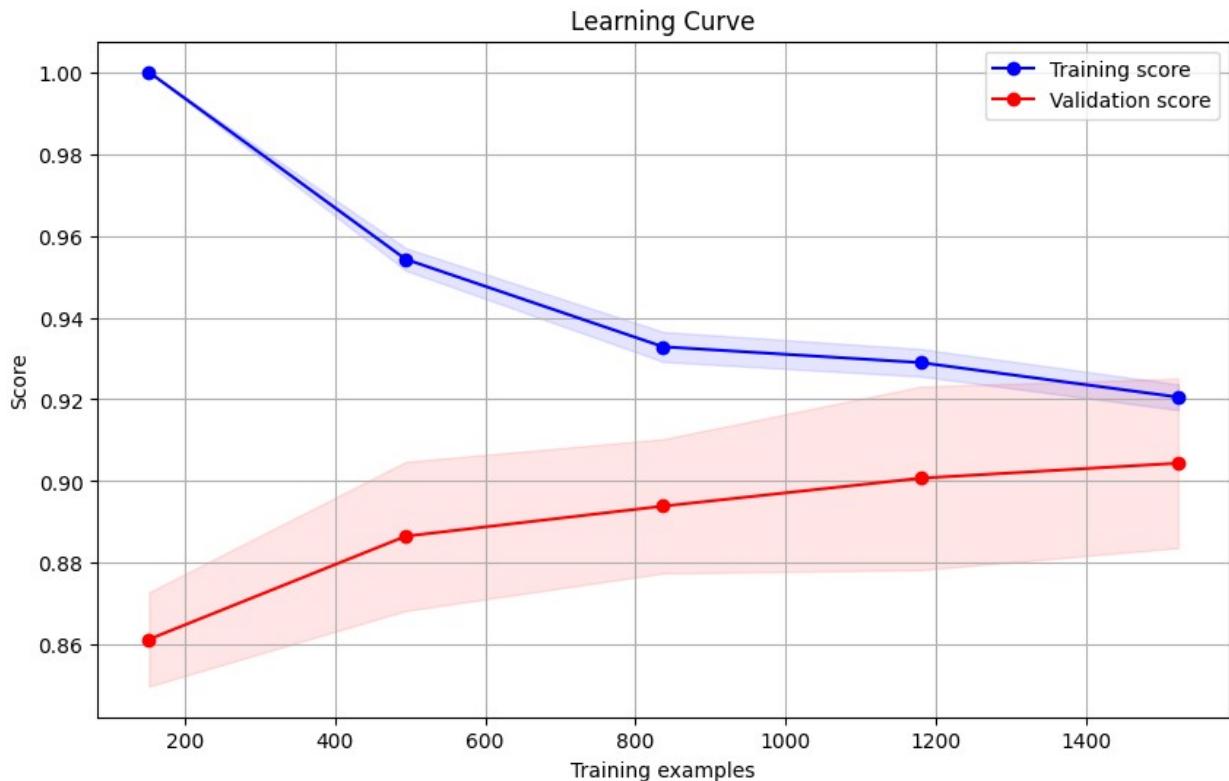
	precision	recall	f1-score	support
0	0.88	0.87	0.87	615
1	0.94	0.94	0.94	1289
accuracy			0.92	1904
macro avg	0.91	0.91	0.91	1904
weighted avg	0.92	0.92	0.92	1904

```
Test Classification Report:
```

	precision	recall	f1-score	support
0	0.87	0.83	0.85	150
1	0.92	0.94	0.93	327
accuracy			0.91	477
macro avg	0.90	0.89	0.89	477
weighted avg	0.91	0.91	0.91	477







MLFLOW Logging is completed

## Balanced Dataset

### Hyper parameter Tuning

```
# Base RandomForest model with the provided parameters
base_rf =
RandomForestClassifier(class_weight='balanced', criterion='log_loss',
max_depth=20, max_leaf_nodes=20,
                      n_estimators=5, oob_score=True,
random_state=42)

# Define the parameter grid for Bagging
start_tune_time = time.time()
param_dist = {
    'n_estimators': stats.randint(10, 500), # Number of base
estimators in the ensemble
    'max_samples': stats.uniform(0.1, 1.0), # Fraction of samples to
draw from X to train each base estimator
    'max_features': stats.uniform(0.1, 1.0), # Fraction of features
to draw from X to train each base estimator
    'bootstrap': [True, False], # Whether samples are drawn with
replacement
    'bootstrap_features': [True, False], # Whether features are drawn
```

```

with replacement
    'random_state': [42] # Fixed random state for reproducibility
}

# Initialize the Bagging classifier with the RandomForest as the base
estimator
bagging_clf = BaggingClassifier(base_estimator=base_rf)

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=bagging_clf,
    param_distributions=param_dist,
    n_iter=200, # Number of parameter settings to try
    cv=5, # Number of folds in cross-validation
    verbose=1,
    random_state=42,
    n_jobs=-1 # Use all available cores
)

# Fit RandomizedSearchCV
random_search.fit(X_train_bal, y_train_bal)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning_time", tuning_time)

Fitting 5 folds for each of 200 candidates, totalling 1000 fits
Best parameters found: {'bootstrap': False, 'bootstrap_features': False, 'max_features': 0.9631385219890038, 'max_samples': 0.9803599686384168, 'n_estimators': 154, 'random_state': 42}
Best score: 0.8824678256942876
Tuning_time 861.6904680728912

```

## Logging Best Bagging RF Model into MLFLOW

```

# Model details
name = "Tuned_Bagging_RF_on_Balanced_Dataset"
bal_type = "Balanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time
start_train_time = time.time()
model.fit(X_train_bal, y_train_bal)
end_train_time = time.time()

```

```

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_bal_train = model.predict(X_train_bal)
y_pred_bal_test = model.predict(X_test_bal)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df,
mlflow_logging_and_metric_printing
time_df,feature_importance_df =
log_time_and_feature_importances_df(time_df,feature_importance_df,name
,training_time,testing_time,model,ola_features,bal_type,tuning_time)
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_bal,y_t
rain_bal,X_test_bal,y_test_bal,y_pred_bal_train,y_pred_bal_test,tuning
_score,**params)

Model: Tuned_Bagging_RF_on_Balanced_Dataset
params: {'bootstrap': False, 'bootstrap_features': False,
'max_features': 0.9631385219890038, 'max_samples': 0.9803599686384168,
'n_estimators': 154, 'random_state': 42}
Training Time: 5.0681 seconds
Testing Time: 0.4171 seconds
Tuning Time: 861.6905 seconds
Train Metrics:
Accuracy_train: 0.9077
Precision_train: 0.8873
Recall_train: 0.9341
F1_score_train: 0.9101
F2_score_train: 0.9243
Roc_auc_train: 0.9681
Pr_auc_train: 0.9690

Test Metrics:
Accuracy_test: 0.9015
Precision_test: 0.9321
Recall_test: 0.9235
F1_score_test: 0.9278
F2_score_test: 0.9252

```

```
Roc_auc_test: 0.9482
```

```
Pr_auc_test: 0.9708
```

Tuning Metrics:

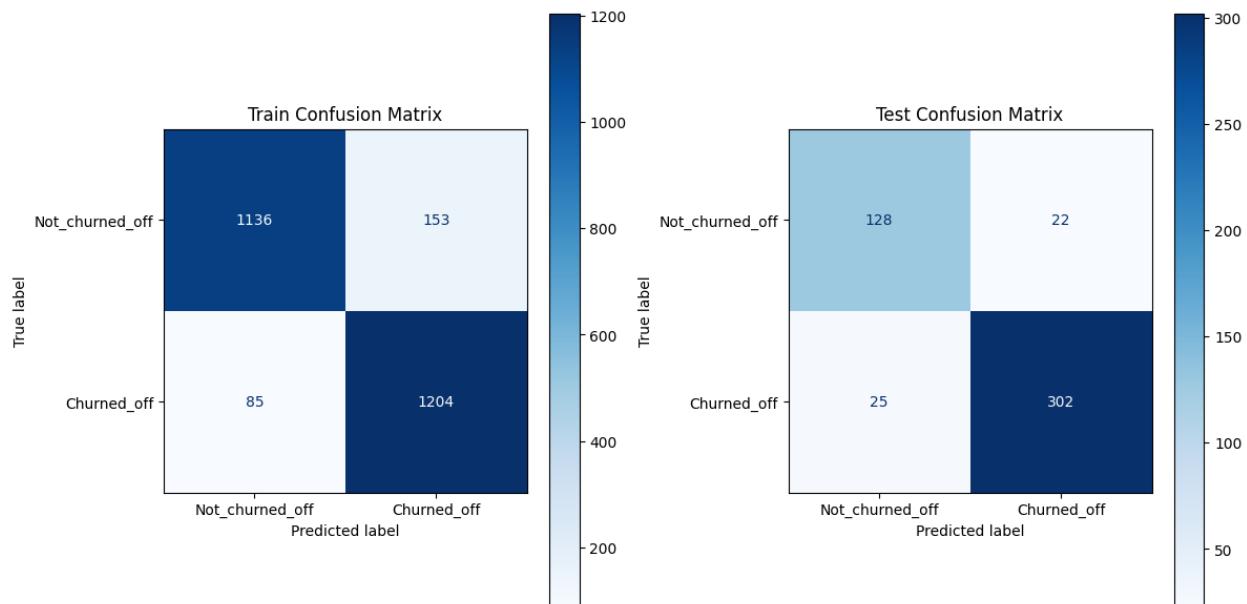
```
hyper_parameter_tuning_best_est_score: 0.8825
```

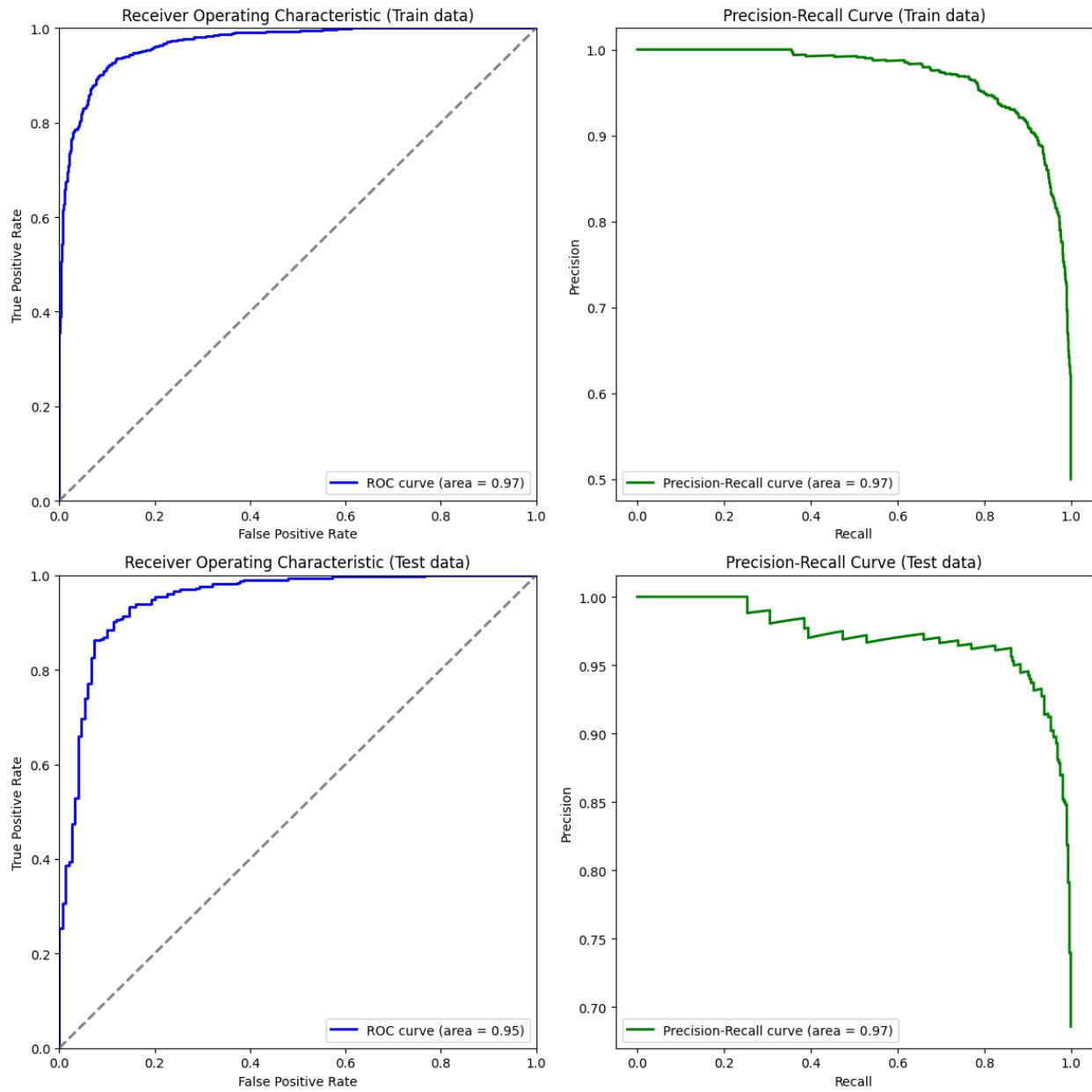
Train Classification Report:

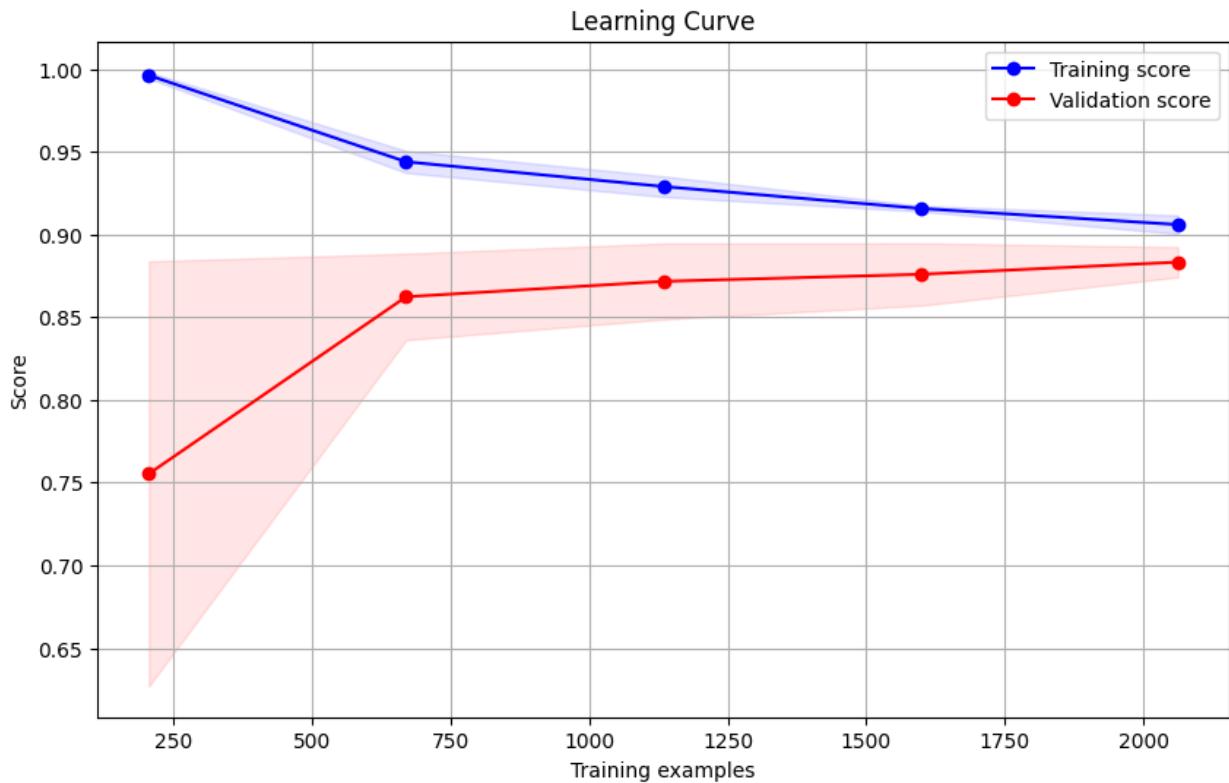
	precision	recall	f1-score	support
0	0.93	0.88	0.91	1289
1	0.89	0.93	0.91	1289
accuracy			0.91	2578
macro avg	0.91	0.91	0.91	2578
weighted avg	0.91	0.91	0.91	2578

Test Classification Report:

	precision	recall	f1-score	support
0	0.84	0.85	0.84	150
1	0.93	0.92	0.93	327
accuracy			0.90	477
macro avg	0.88	0.89	0.89	477
weighted avg	0.90	0.90	0.90	477







MLFL0W Logging is completed

## ADABOOST MODEL

### Imbalanced Dataset

#### Hyper parameter Tuning

```
# Define the base estimator (Decision Stump)
base_stump = DecisionTreeClassifier(max_depth=5)

# Define the parameter grid
start_tune_time = time.time()
param_dist = {
    'n_estimators': stats.randint(50, 1000), # Number of boosting stages
    'learning_rate': stats.uniform(0.01, 1.0), # Step size for boosting
    'algorithm': ['SAMME', 'SAMME.R'], # Algorithm to use for boosting
}

# Initialize the AdaBoost model with the decision stump as base estimator
ada_boost = AdaBoostClassifier(estimator=base_stump, random_state=42)
```

```

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=ada_boost,
    param_distributions=param_dist,
    n_iter=200, # Number of parameter settings to try
    cv=5, # Number of folds in cross-validation
    verbose=1,
    random_state=42,
    n_jobs=-1 # Use all available cores
)

# Fit RandomizedSearchCV
random_search.fit(X_train_imb, y_train_imb)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning_time:", tuning_time)

Fitting 5 folds for each of 200 candidates, totalling 1000 fits
Best parameters found: {'algorithm': 'SAMME', 'learning_rate': 0.1370605126518848, 'n_estimators': 313}
Best score: 0.9154344522724133
Tuning_time: 908.1767725944519

```

## Logging Best Adaboost Model into MLFLOW

```

# Model details
name = "Tuned_Adaboost_on_Imbalanced_Dataset"
bal_type = "Imbalanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time
start_train_time = time.time()
model.fit(X_train_imb, y_train_imb)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_imb_train = model.predict(X_train_imb)
y_pred_imb_test = model.predict(X_test_imb)
end_test_time = time.time()

```

```

# Calculate testing time
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df,
# mlflow_logging_and_metric_printing
time_df,feature_importance_df =
log_time_and_feature_importances_df(time_df,feature_importance_df,name
,training_time,testing_time,model,ola_features,bal_type,tuning_time)
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_imb,y_t
rain_imb,X_test_imb,y_test_imb,y_pred_imb_train,y_pred_imb_test,tuning
_score,**params)

Model: Tuned_Adaboost_on_Imbalanced_Dataset
params: {'algorithm': 'SAMME', 'learning_rate': 0.1370605126518848,
'n_estimators': 313}
Training Time: 3.8815 seconds
Testing Time: 0.2613 seconds
Tuning Time: 908.1768 seconds
Train Metrics:
Accuracy_train: 0.9727
Precision_train: 0.9703
Recall_train: 0.9899
F1_score_train: 0.9800
F2_score_train: 0.9859
Roc_auc_train: 0.9980
Pr_auc_train: 0.9991

Test Metrics:
Accuracy_test: 0.9036
Precision_test: 0.9169
Recall_test: 0.9450
F1_score_test: 0.9307
F2_score_test: 0.9392
Roc_auc_test: 0.9559
Pr_auc_test: 0.9759

Tuning Metrics:
hyper_parameter_tuning_best_est_score: 0.9154

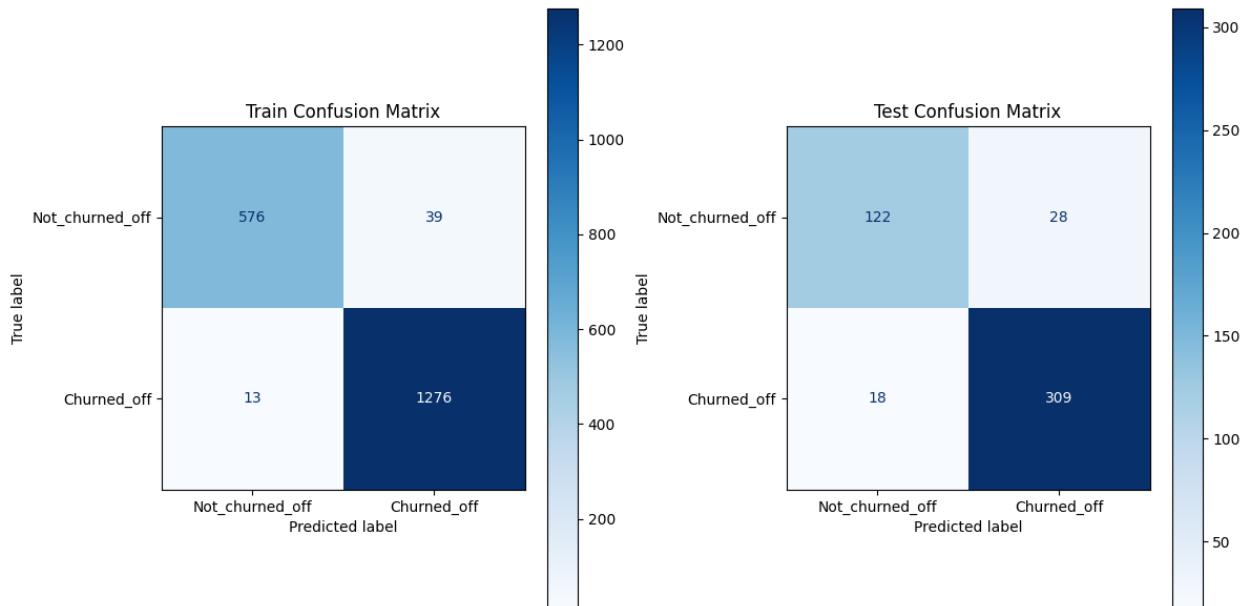
Train Classification Report:
    precision    recall   f1-score   support

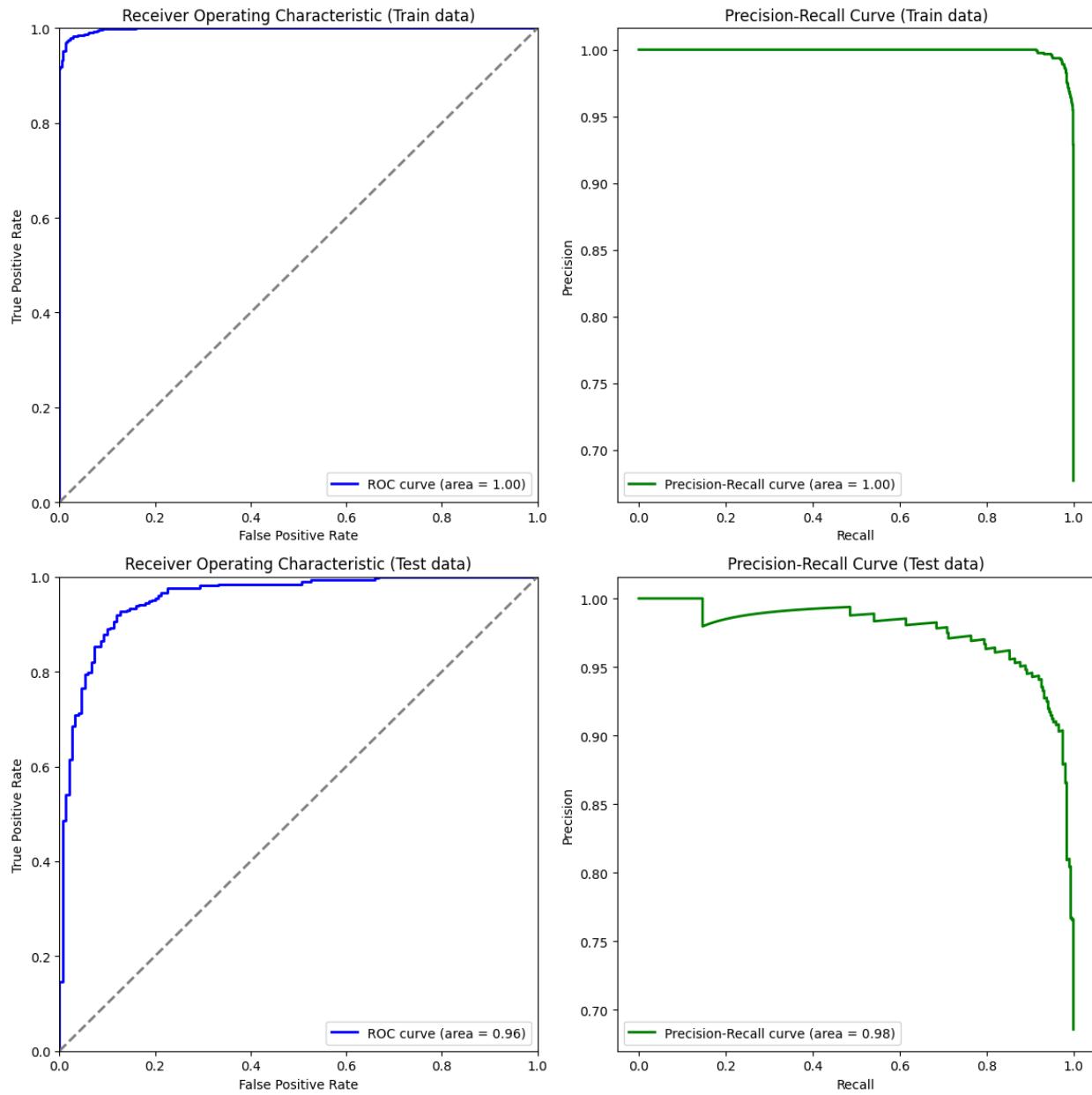
```

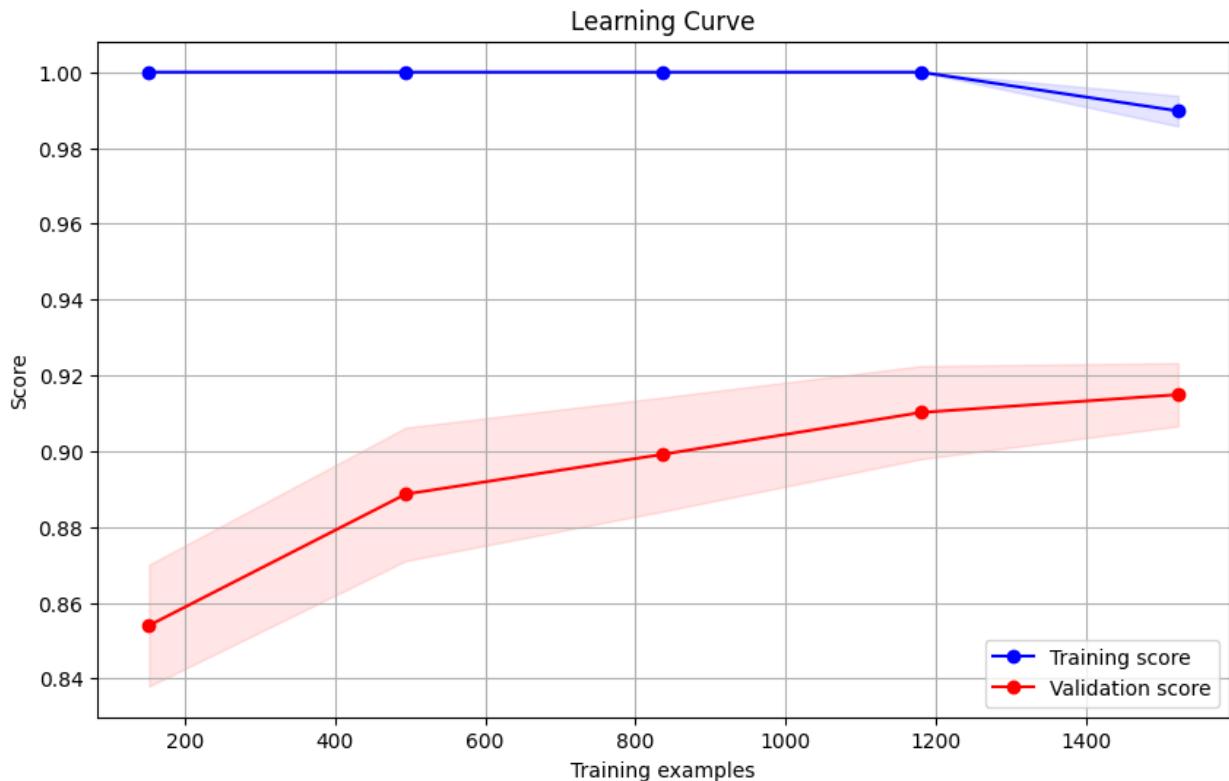
	0	0.98	0.94	0.96	615
	1	0.97	0.99	0.98	1289
accuracy				0.97	1904
macro avg		0.97	0.96	0.97	1904
weighted avg		0.97	0.97	0.97	1904

Test Classification Report:					
	precision	recall	f1-score	support	
0	0.87	0.81	0.84	150	
1	0.92	0.94	0.93	327	
accuracy			0.90	477	
macro avg	0.89	0.88	0.89	477	
weighted avg	0.90	0.90	0.90	477	







MLFL0W Logging is completed

## Balanced Dataset

### Hyper parameter Tuning

```
# Define the base estimator (Decision Stump)
base_stump = DecisionTreeClassifier(max_depth=5)

# Define the parameter grid
start_tune_time = time.time()
param_dist = {
    'n_estimators': stats.randint(50, 1000), # Number of boosting stages
    'learning_rate': stats.uniform(0.01, 1.0), # Step size for boosting
    'algorithm': ['SAMME', 'SAMME.R'], # Algorithm to use for boosting
}

# Initialize the AdaBoost model with the decision stump as base estimator
ada_boost = AdaBoostClassifier(estimator=base_stump, random_state=42)

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
```

```

estimator=ada_boost,
param_distributions=param_dist,
n_iter=200, # Number of parameter settings to try
cv=5, # Number of folds in cross-validation
verbose=1,
random_state=42,
n_jobs=-1 # Use all available cores
)

# Fit RandomizedSearchCV
random_search.fit(X_train_bal, y_train_bal)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning_time:", tuning_time)

Fitting 5 folds for each of 200 candidates, totalling 1000 fits
Best parameters found: {'algorithm': 'SAMME.R', 'learning_rate': 0.9000053418175663, 'n_estimators': 520}
Best score: 0.9185534733197862
Tuning_time: 1126.033709526062

```

## Logging Best Adaboost Model into MLFLOW

```

# Model details
name = "Tuned_Adaboost_on_Balanced_Dataset"
bal_type = "Balanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time
start_train_time = time.time()
model.fit(X_train_bal, y_train_bal)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_bal_train = model.predict(X_train_bal)
y_pred_bal_test = model.predict(X_test_bal)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time

```

```

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df,
mlflow_logging_and_metric_printing
time_df, feature_importance_df =
log_time_and_feature_importances_df(time_df, feature_importance_df, name
, training_time, testing_time, model, ola_features, bal_type, tuning_time)
mlflow_logging_and_metric_printing(model, name, bal_type, X_train_bal, y_t
rain_bal, X_test_bal, y_test_bal, y_pred_bal_train, y_pred_bal_test, tuning
_score, **params)

Model: Tuned_Adaboost_on_Balanced_Dataset
params: {'algorithm': 'SAMME.R', 'learning_rate': 0.9000053418175663,
'n_estimators': 520}
Training Time: 8.7266 seconds
Testing Time: 0.5456 seconds
Tuning Time: 1126.0337 seconds
Train Metrics:
Accuracy_train: 1.0000
Precision_train: 1.0000
Recall_train: 1.0000
F1_score_train: 1.0000
F2_score_train: 1.0000
Roc_auc_train: 1.0000
Pr_auc_train: 1.0000

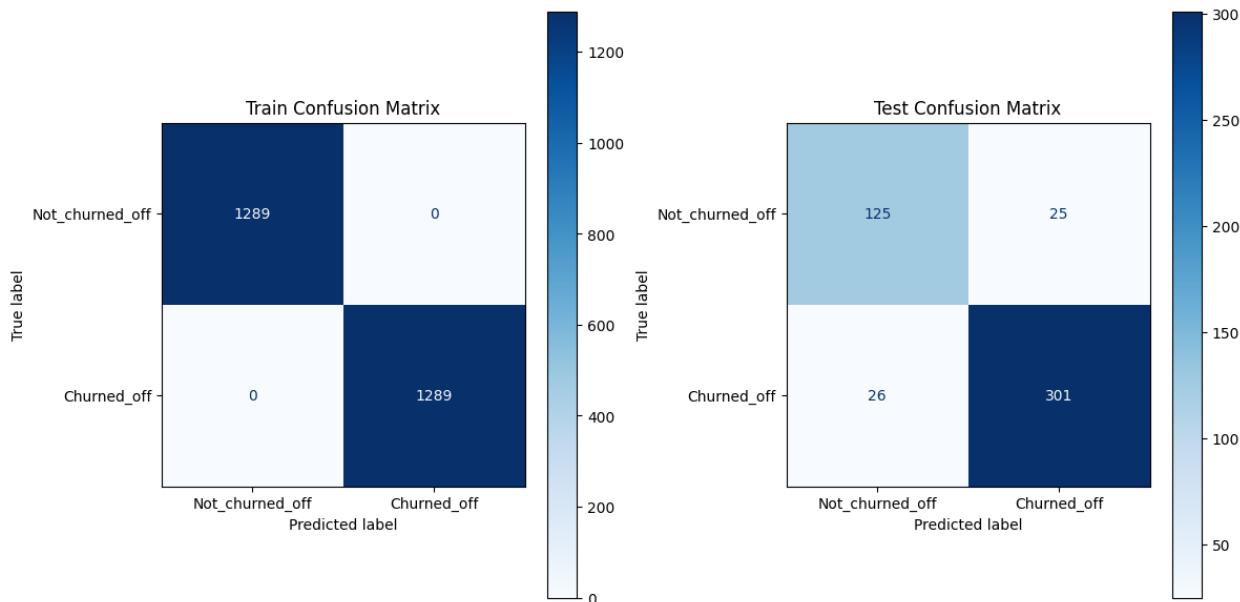
Test Metrics:
Accuracy_test: 0.8931
Precision_test: 0.9233
Recall_test: 0.9205
F1_score_test: 0.9219
F2_score_test: 0.9211
Roc_auc_test: 0.9431
Pr_auc_test: 0.9686

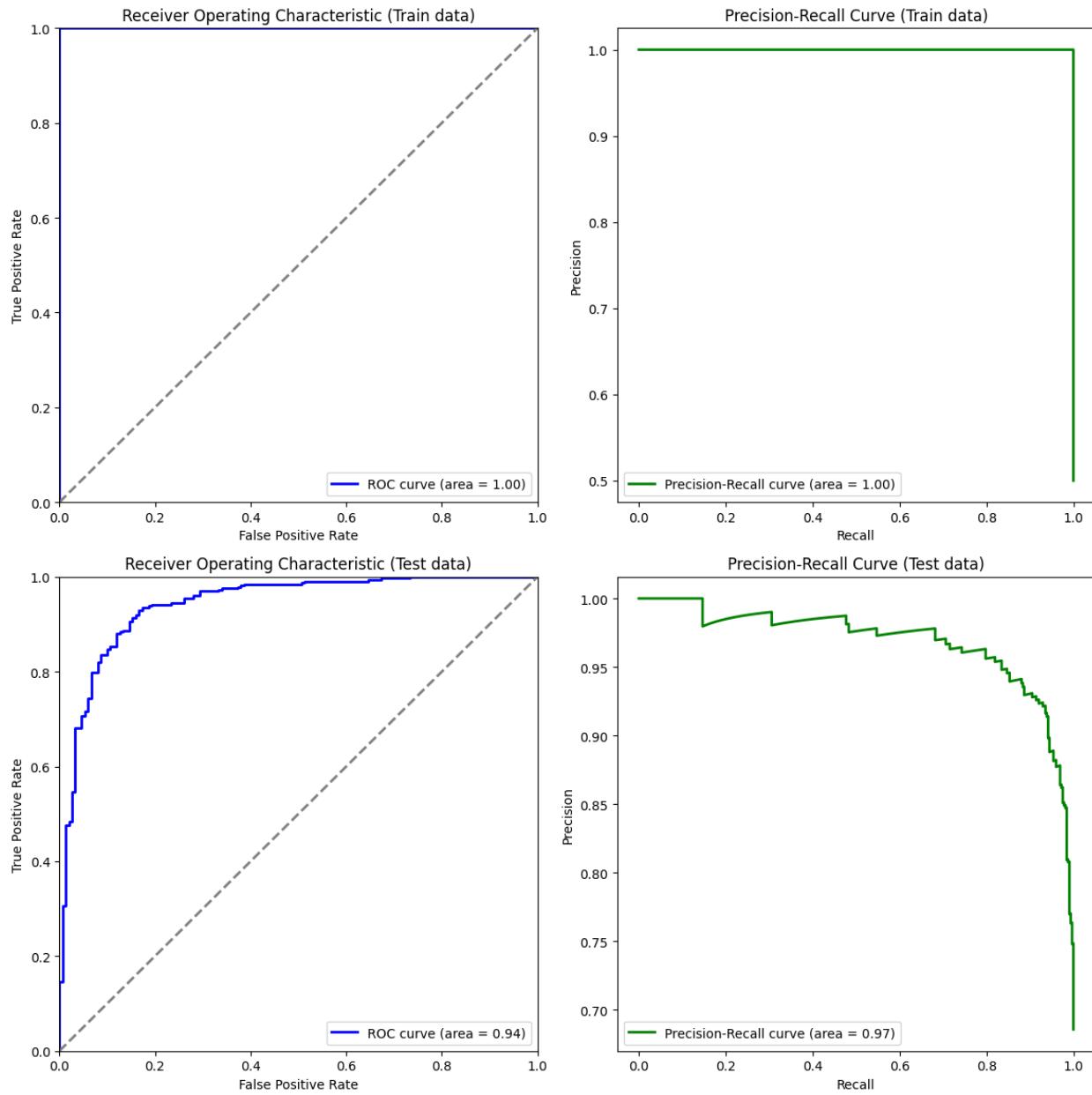
Tuning Metrics:
hyper_parameter_tuning_best_est_score: 0.9186

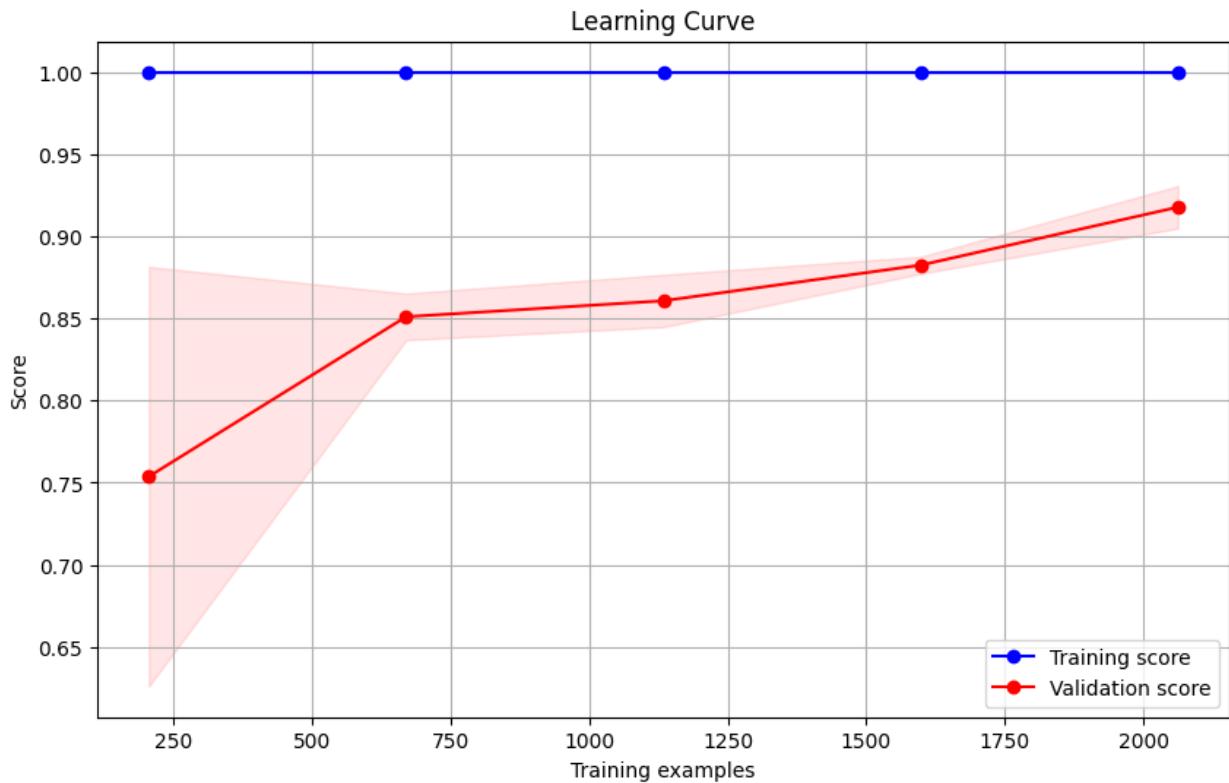
Train Classification Report:
      precision    recall   f1-score   support
          0         1.00      1.00      1.00      1289
          1         1.00      1.00      1.00      1289

```

accuracy		1.00	1.00	1.00	2578
macro avg	1.00	1.00	1.00	2578	
weighted avg	1.00	1.00	1.00	2578	
<b>Test Classification Report:</b>					
	precision	recall	f1-score	support	
0	0.83	0.83	0.83	150	
1	0.92	0.92	0.92	327	
accuracy			0.89	477	
macro avg	0.88	0.88	0.88	477	
weighted avg	0.89	0.89	0.89	477	







MLFL0W Logging is completed

## GRADIENT BOOST MODEL

### Imbalanced Dataset

#### Hyper parameter Tuning

```
# Start timing the tuning process
start_tune_time = time.time()

# Define the parameter grid
param_dist = {
    'n_estimators': stats.randint(100, 1000), # Number of
    # boosting stages to be run
    'learning_rate': stats.uniform(0.01, 0.3), # Learning rate
    # shrinks the contribution of each tree
    'max_depth': stats.randint(3, 15), # Maximum depth
    # of the individual regression estimators
    'min_samples_split': stats.randint(2, 20), # Minimum
    # number of samples required to split an internal node
    'min_samples_leaf': stats.randint(1, 20), # Minimum
    # number of samples required to be at a leaf node
    'max_features': ['auto', 'sqrt', 'log2', None], # Number of
    # features to consider when looking for the best split
```

```

'subsample': stats.uniform(0.7, 0.3),                      # Fraction of
samples used for fitting the individual base learners
'min_impurity_decrease': stats.uniform(0.0, 0.1), # A node will
be split if this split induces a decrease of the impurity
'random_state': [42]                                     # Fixed random
state for reproducibility
}

# Initialize the GradientBoostingClassifier
gbc = GradientBoostingClassifier()

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=gbc,
    param_distributions=param_dist,
    n_iter=200, # Number of parameter settings to try
    cv=5, # Number of folds in cross-validation
    verbose=5,
    random_state=42,
    n_jobs=-1 # Use all available cores
)

# Fit RandomizedSearchCV
random_search.fit(X_train_imb, y_train_imb)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_

# End timing and print tuning time
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning_time:", tuning_time)

Fitting 5 folds for each of 200 candidates, totalling 1000 fits
Best parameters found: {'learning_rate': 0.06676646193550176,
'max_depth': 7, 'max_features': 'log2', 'min_impurity_decrease': 0.09613152881849074, 'min_samples_leaf': 19, 'min_samples_split': 11,
'n_estimators': 493, 'random_state': 42, 'subsample': 0.94598407522243}
Best score: 0.9164815582262744
Tuning_time: 149.77282738685608

```

## Logging Best Gradient boost Model into MLFLOW

```

# Model details
name = "Tuned_Gradient_boost_on_Imbalanced_Dataset"
bal_type = "Imbalanced"
model = random_search.best_estimator_

```

```

params = random_search.best_params_

# Record training time
start_train_time = time.time()
model.fit(X_train_imb, y_train_imb)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_imb_train = model.predict(X_train_imb)
y_pred_imb_test = model.predict(X_test_imb)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df,
mlflow_logging_and_metric_printing
time_df,feature_importance_df =
log_time_and_feature_importances_df(time_df,feature_importance_df,name
,training_time,testing_time,model,ola_features,bal_type,tuning_time)
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_imb,y_t
rain_imb,X_test_imb,y_test_imb,y_pred_imb_train,y_pred_imb_test,tuning
_score,**params)

Model: Tuned_Gradient_boost_on_Imbalanced_Dataset
params: {'learning_rate': 0.06676646193550176, 'max_depth': 7,
'max_features': 'log2', 'min_impurity_decrease': 0.09613152881849074,
'min_samples_leaf': 19, 'min_samples_split': 11, 'n_estimators': 493,
'random_state': 42, 'subsample': 0.94598407522243}
Training Time: 1.0254 seconds
Testing Time: 0.0091 seconds
Tuning Time: 149.7728 seconds
Train Metrics:
Accuracy_train: 0.9496
Precision_train: 0.9529
Recall_train: 0.9736
F1_score_train: 0.9632
F2_score_train: 0.9694
Roc_auc_train: 0.9889

```

Pr\_auc\_train: 0.9945

Test Metrics:

Accuracy\_test: 0.9203  
Precision\_test: 0.9263  
Recall\_test: 0.9602  
F1\_score\_test: 0.9429  
F2\_score\_test: 0.9532  
Roc\_auc\_test: 0.9553  
Pr\_auc\_test: 0.9625

Tuning Metrics:

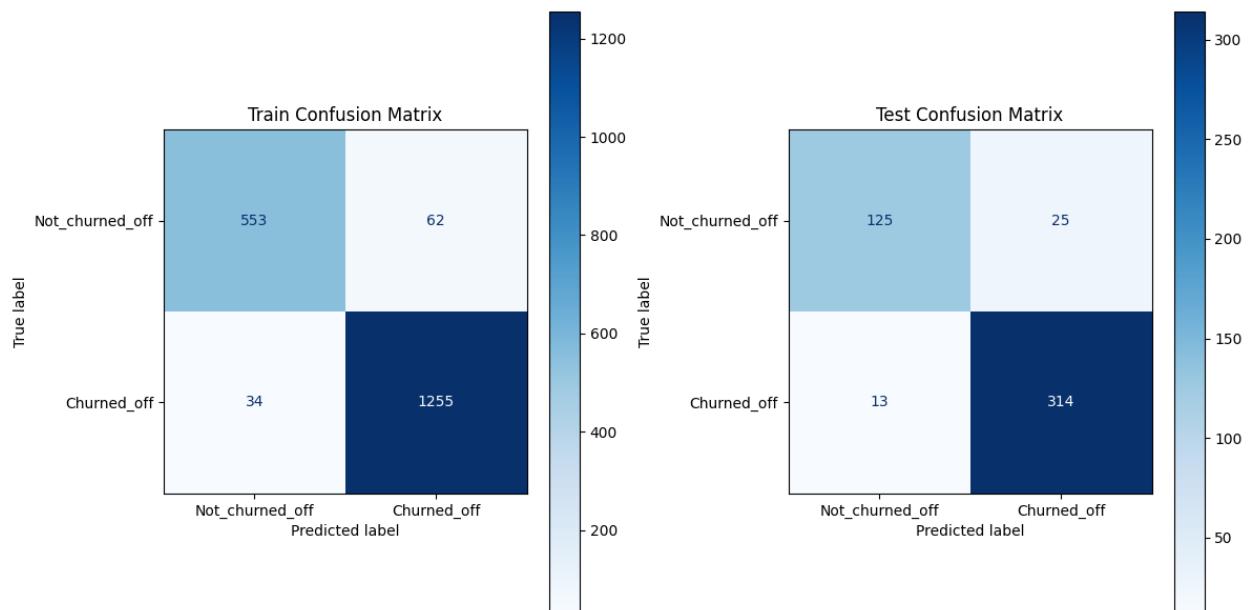
hyper\_parameter\_tuning\_best\_est\_score: 0.9165

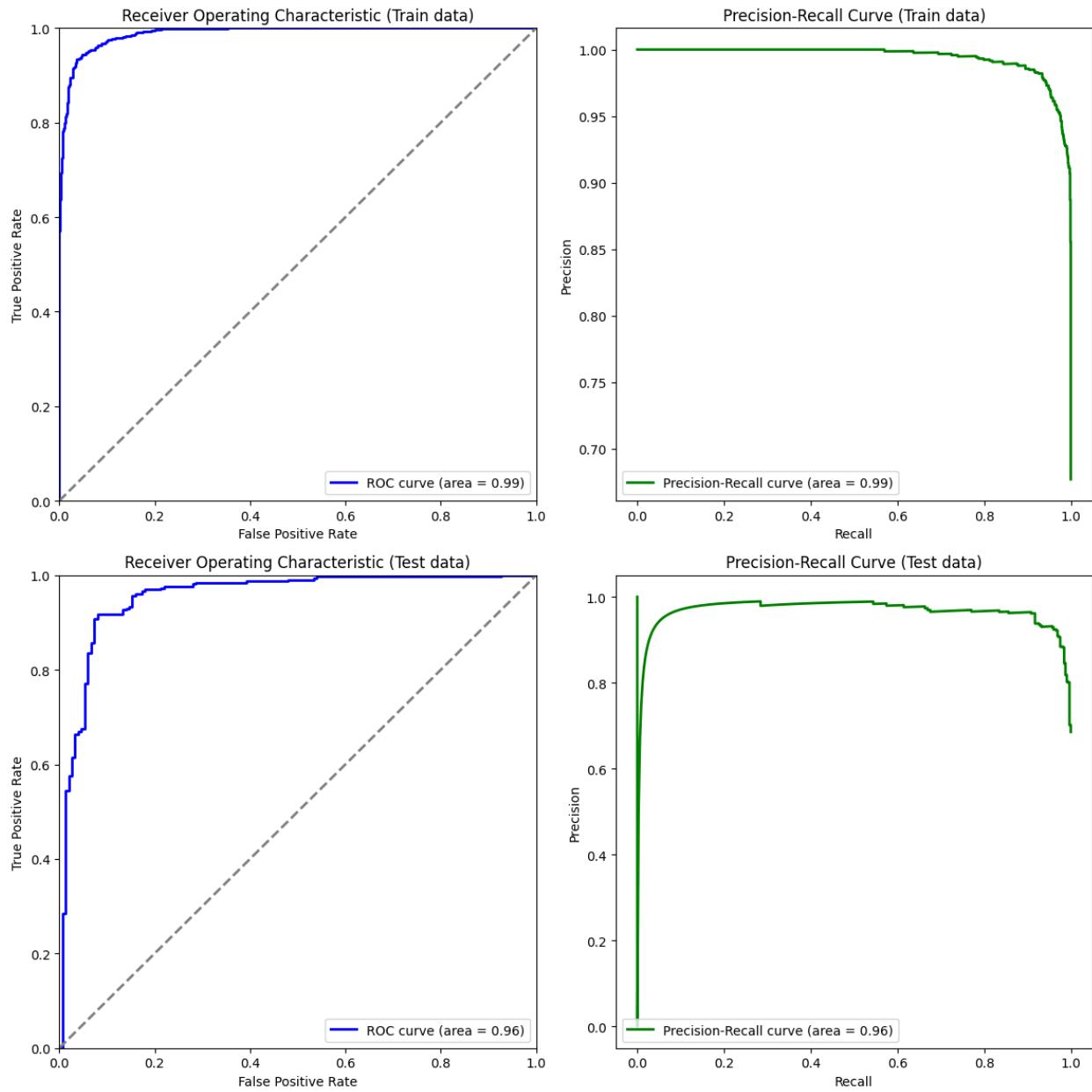
Train Classification Report:

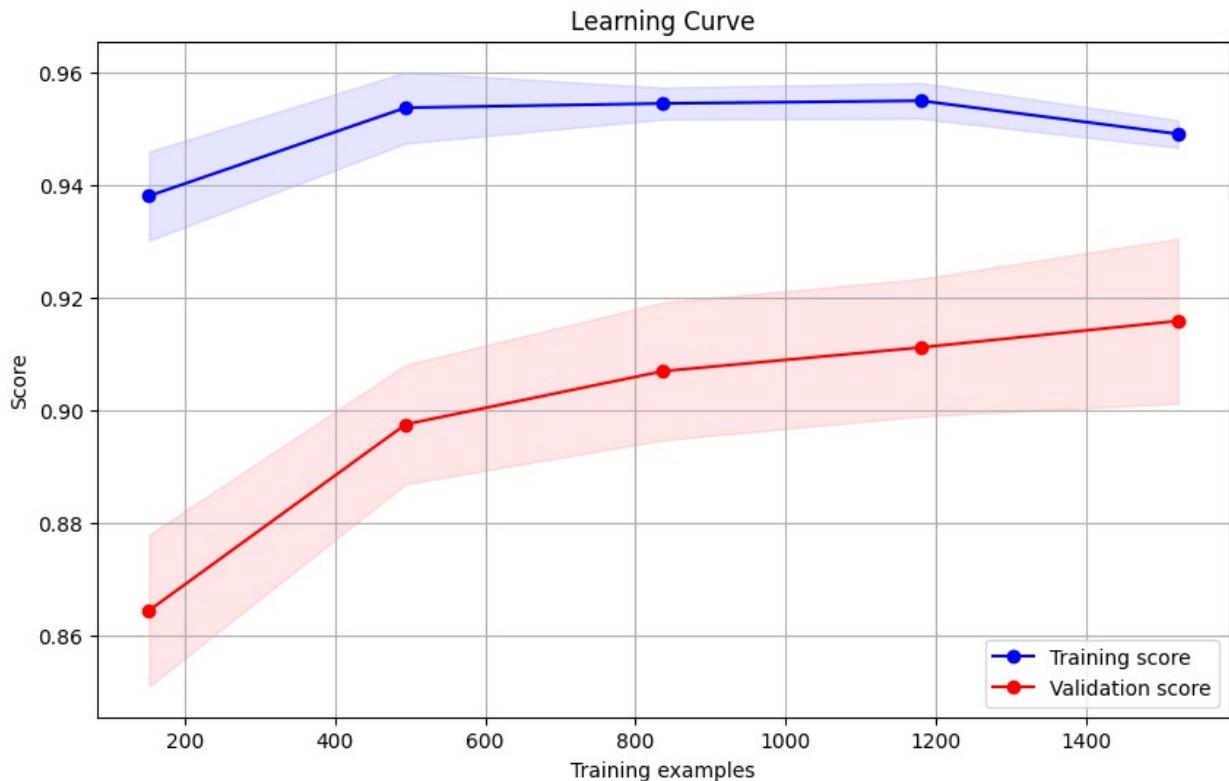
	precision	recall	f1-score	support
0	0.94	0.90	0.92	615
1	0.95	0.97	0.96	1289
accuracy			0.95	1904
macro avg	0.95	0.94	0.94	1904
weighted avg	0.95	0.95	0.95	1904

Test Classification Report:

	precision	recall	f1-score	support
0	0.91	0.83	0.87	150
1	0.93	0.96	0.94	327
accuracy			0.92	477
macro avg	0.92	0.90	0.91	477
weighted avg	0.92	0.92	0.92	477







MLFL0W Logging is completed

## Balanced Dataset

### Hyper parameter Tuning

```
# Start timing the tuning process
start_tune_time = time.time()

# Define the parameter grid
param_dist = {
    'n_estimators': stats.randint(100, 1000),                      # Number of
    boosting stages to be run
    'learning_rate': stats.uniform(0.01, 0.3),                      # Learning rate
    shrinks the contribution of each tree
    'max_depth': stats.randint(3, 15),                                # Maximum depth
    of the individual regression estimators
    'min_samples_split': stats.randint(2, 20),                         # Minimum
    number of samples required to split an internal node
    'min_samples_leaf': stats.randint(1, 20),                          # Minimum
    number of samples required to be at a leaf node
    'max_features': ['auto', 'sqrt', 'log2', None],                  # Number of
    features to consider when looking for the best split
    'subsample': stats.uniform(0.7, 0.3),                            # Fraction of
    samples used for fitting the individual base learners
}
```

```

'min_impurity_decrease': stats.uniform(0.0, 0.1), # A node will
be split if this split induces a decrease of the impurity
'random_state': [42] # Fixed random
state for reproducibility
}

# Initialize the GradientBoostingClassifier
gbc = GradientBoostingClassifier()

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=gbc,
    param_distributions=param_dist,
    n_iter=200, # Number of parameter settings to try
    cv=5, # Number of folds in cross-validation
    verbose=5,
    random_state=42,
    n_jobs=-1 # Use all available cores
)

# Fit RandomizedSearchCV
random_search.fit(X_train_bal, y_train_bal)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_

# End timing and print tuning time
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning_time:", tuning_time)

Fitting 5 folds for each of 200 candidates, totalling 1000 fits
Best parameters found: {'learning_rate': 0.07858754845747705,
'max_depth': 9, 'max_features': 'log2', 'min_impurity_decrease':
0.0001120111480109487, 'min_samples_leaf': 2, 'min_samples_split': 15,
'n_estimators': 688, 'random_state': 42, 'subsample': 0.8522403201282508}
Best score: 0.9204937156619252
Tuning_time: 184.98244452476501

```

## Logging Best Gradient boost Model into MLFLOW

```

# Model details
name = "Tuned_Gradient_boost_on_Balanced_Dataset"
bal_type = "Balanced"
model = random_search.best_estimator_
params = random_search.best_params_

```

```

# Record training time
start_train_time = time.time()
model.fit(X_train_bal, y_train_bal)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_bal_train = model.predict(X_train_bal)
y_pred_bal_test = model.predict(X_test_bal)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df,
mlflow_logging_and_metric_printing
time_df,feature_importance_df =
log_time_and_feature_importances_df(time_df,feature_importance_df,name
,training_time,testing_time,model,ola_features,bal_type,tuning_time)
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_bal,y_t
rain_bal,X_test_bal,y_test_bal,y_pred_bal_train,y_pred_bal_test,tuning
_score,**params)

Model: Tuned_Gradient_boost_on_Balanced_Dataset
params: {'learning_rate': 0.07858754845747705, 'max_depth': 9,
'max_features': 'log2', 'min_impurity_decrease':
0.0001120111480109487, 'min_samples_leaf': 2, 'min_samples_split': 15,
'n_estimators': 688, 'random_state': 42, 'subsample':
0.8522403201282508}
Training Time: 3.1670 seconds
Testing Time: 0.0326 seconds
Tuning Time: 184.9824 seconds
Train Metrics:
Accuracy_train: 1.0000
Precision_train: 1.0000
Recall_train: 1.0000
F1_score_train: 1.0000
F2_score_train: 1.0000
Roc_auc_train: 1.0000
Pr_auc_train: 1.0000

```

```
Test Metrics:  
Accuracy_test: 0.8889  
Precision_test: 0.9281  
Recall_test: 0.9083  
F1_score_test: 0.9181  
F2_score_test: 0.9122  
Roc_auc_test: 0.9405  
Pr_auc_test: 0.9598
```

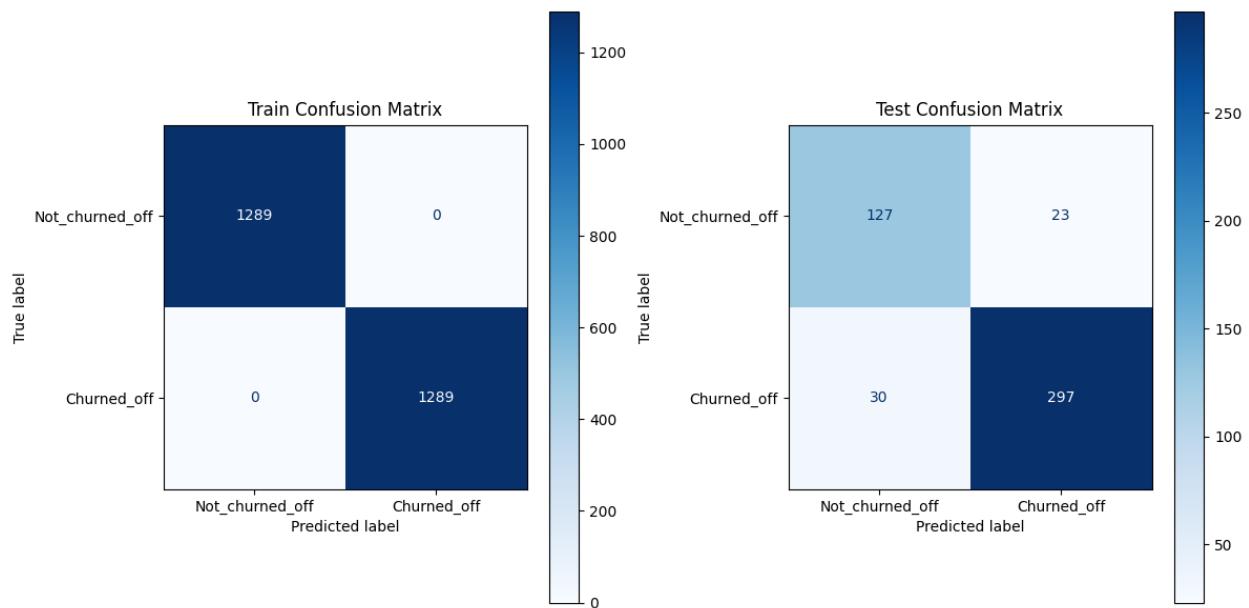
```
Tuning Metrics:  
hyper_parameter_tuning_best_est_score: 0.9205
```

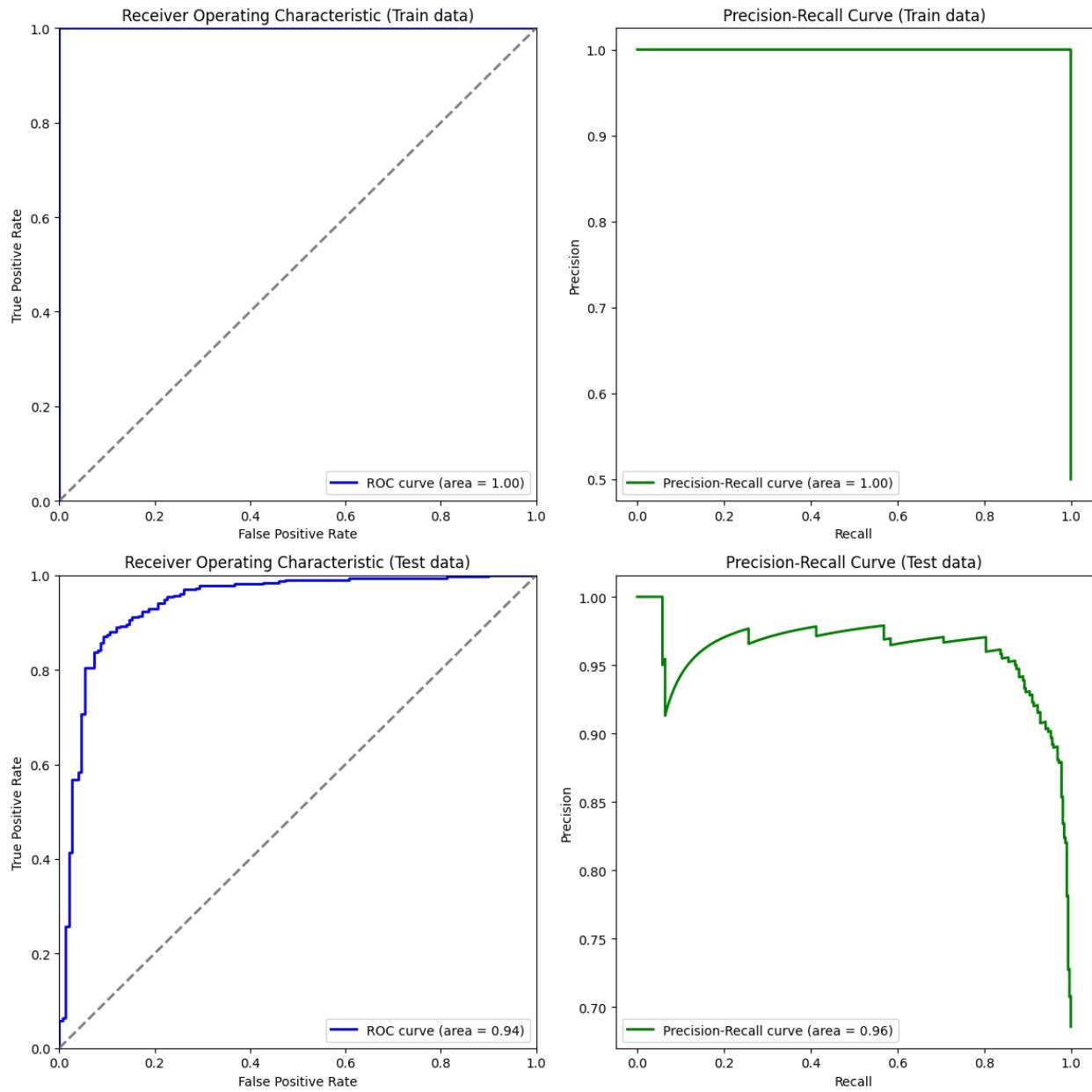
```
Train Classification Report:
```

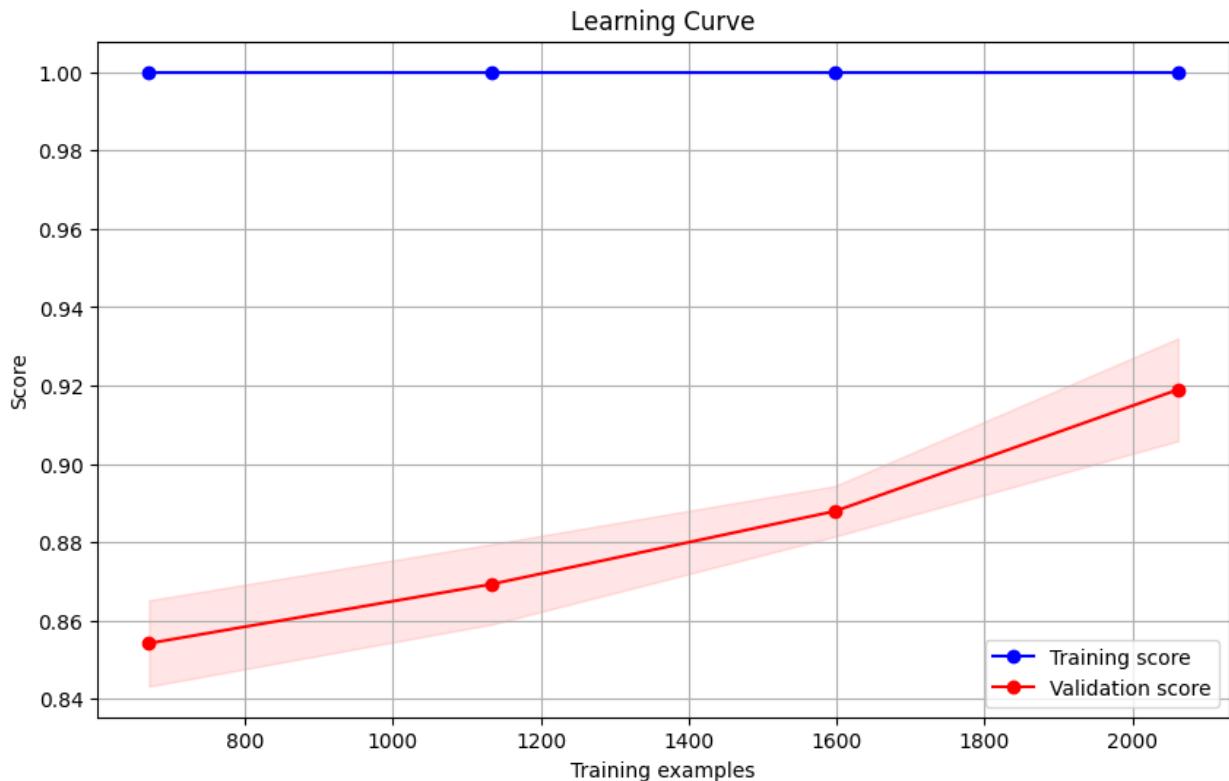
	precision	recall	f1-score	support
0	1.00	1.00	1.00	1289
1	1.00	1.00	1.00	1289
accuracy			1.00	2578
macro avg	1.00	1.00	1.00	2578
weighted avg	1.00	1.00	1.00	2578

```
Test Classification Report:
```

	precision	recall	f1-score	support
0	0.81	0.85	0.83	150
1	0.93	0.91	0.92	327
accuracy			0.89	477
macro avg	0.87	0.88	0.87	477
weighted avg	0.89	0.89	0.89	477







MLFL0W Logging is completed

## XGBOOST MODEL

### Imbalanced Dataset

#### Hyper parameter Tuning

```
# Start timing the tuning process
start_tune_time = time.time()

# Define the parameter grid
param_dist = {
    'n_estimators': stats.randint(100, 1000),           # Number of
    'boosting rounds':                                # Learning rate
    'learning_rate': stats.uniform(0.01, 0.3),          # Maximum depth
    ('shrinkage factor'):                            # Minimum sum
    'max_depth': stats.randint(3, 15),                  # Minimum loss
    ('of a tree'):                                     # Subsample
    'min_child_weight': stats.randint(1, 10),           # Reduction required to make a split
    'gamma': stats.uniform(0, 0.5),                      # Subsample ratio of the training instance
    'subsample': stats.uniform(0.7, 0.3),
```

```

    'colsample_bytree': stats.uniform(0.7, 0.3),           # Subsample
ratio of columns when constructing each tree
    'colsample_bylevel': stats.uniform(0.7, 0.3),          # Subsample
ratio of columns for each level of a tree
    'colsample_bynode': stats.uniform(0.7, 0.3),           # Subsample
ratio of columns for each node (split)
    'reg_alpha': stats.uniform(0, 0.5),                   # L1
regularization term on weights
    'reg_lambda': stats.uniform(0.5, 1.5),                 # L2
regularization term on weights
    'scale_pos_weight': stats.uniform(0.5, 2),            # Balance of
positive and negative weights
    'booster': ['gbtree', 'gblinear', 'dart'],             # Type of
booster to use
    'tree_method': ['auto', 'exact', 'approx', 'hist'],   # Algorithm
used to train trees
    'grow_policy': ['depthwise', 'lossguide'],            # Controls the
way new nodes are added to the tree
    'objective': ['binary:logistic', 'multi:softprob'],   # Learning task
and the corresponding objective function
    'sampling_method': ['uniform', 'gradient_based'],     # Method used
to sample training data
    'random_state': [42]                                  # Fixed random
state for reproducibility
}

# Initialize the XGBClассifier
xgb_clf = XGBClassifier(use_label_encoder=False,
eval_metric='logloss')

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=xgb_clf,
    param_distributions=param_dist,
    n_iter=200, # Number of parameter settings to try
    cv=5, # Number of folds in cross-validation
    verbose=1,
    random_state=42,
    n_jobs=-1 # Use all available cores
)

# Fit RandomizedSearchCV
random_search.fit(X_train_imb, y_train_imb)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_

# End timing and print tuning time

```

```

end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning_time:", tuning_time)

Fitting 5 folds for each of 200 candidates, totalling 1000 fits
Best parameters found: {'booster': 'dart', 'colsample_bylevel':
0.9526356769407125, 'colsample_bynode': 0.9514986114133412,
'colsample_bytree': 0.840607947938491, 'gamma': 0.2074097511688326,
'grow_policy': 'depthwise', 'learning_rate': 0.014101589448099186,
'max_depth': 9, 'min_child_weight': 6, 'n_estimators': 575,
'objective': 'binary:logistic', 'random_state': 42, 'reg_alpha':
0.26717313751473154, 'reg_lambda': 1.6248661242049578,
'sampling_method': 'uniform', 'scale_pos_weight': 1.610863411205255,
'subsample': 0.9306962245541531, 'tree_method': 'auto'}
Best score: 0.9122793203481143
Tuning_time: 845.787605047226

```

### Logging Best XG boost Model into MLFLOW

```

# Model details
name = "Tuned_XG_boost_on_Imbalanced_Dataset"
bal_type = "Imbalanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time
start_train_time = time.time()
model.fit(X_train_imb, y_train_imb)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_imb_train = model.predict(X_train_imb)
y_pred_imb_test = model.predict(X_test_imb)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df,

```

```

mlflow_logging_and_metric_printing
time_df,feature_importance_df =
log_time_and_feature_importances_df(time_df,feature_importance_df,name
,training_time,testing_time,model,ola_features,bal_type,tuning_time)
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_imb,y_t
rain_imb,X_test_imb,y_test_imb,y_pred_imb_train,y_pred_imb_test,tuning
_score,**params)

Model: Tuned_XG_boost_on_Imbalanced_Dataset
params: {'booster': 'dart', 'colsample_bylevel': 0.9526356769407125,
'colsample_bynode': 0.9514986114133412, 'colsample_bytree':
0.840607947938491, 'gamma': 0.2074097511688326, 'grow_policy':
'depthwise', 'learning_rate': 0.014101589448099186, 'max_depth': 9,
'min_child_weight': 6, 'n_estimators': 575, 'objective':
'binary:logistic', 'random_state': 42, 'reg_alpha':
0.26717313751473154, 'reg_lambda': 1.6248661242049578,
'sampling_method': 'uniform', 'scale_pos_weight': 1.610863411205255,
'subsample': 0.9306962245541531, 'tree_method': 'auto'}
Training Time: 44.0436 seconds
Testing Time: 0.2345 seconds
Tuning Time: 845.7876 seconds
Train Metrics:
Accuracy_train: 0.9559
Precision_train: 0.9500
Recall_train: 0.9868
F1_score_train: 0.9680
F2_score_train: 0.9792
Roc_auc_train: 0.9919
Pr_auc_train: 0.9960

Test Metrics:
Accuracy_test: 0.9099
Precision_test: 0.9104
Recall_test: 0.9633
F1_score_test: 0.9361
F2_score_test: 0.9522
Roc_auc_test: 0.9569
Pr_auc_test: 0.9689

Tuning Metrics:
hyper_parameter_tuning_best_est_score: 0.9123

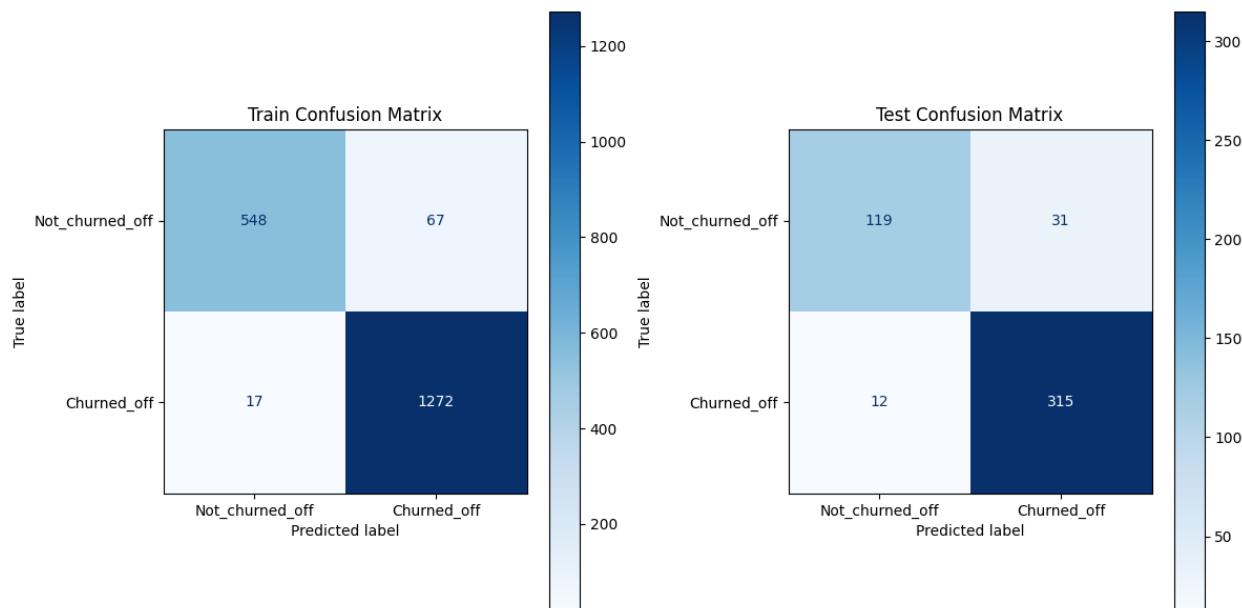
Train Classification Report:
      precision    recall   f1-score   support
          0       0.97     0.89     0.93      615
          1       0.95     0.99     0.97     1289
   accuracy          0.96     0.94     0.95     1904
  macro avg          0.96     0.94     0.95     1904

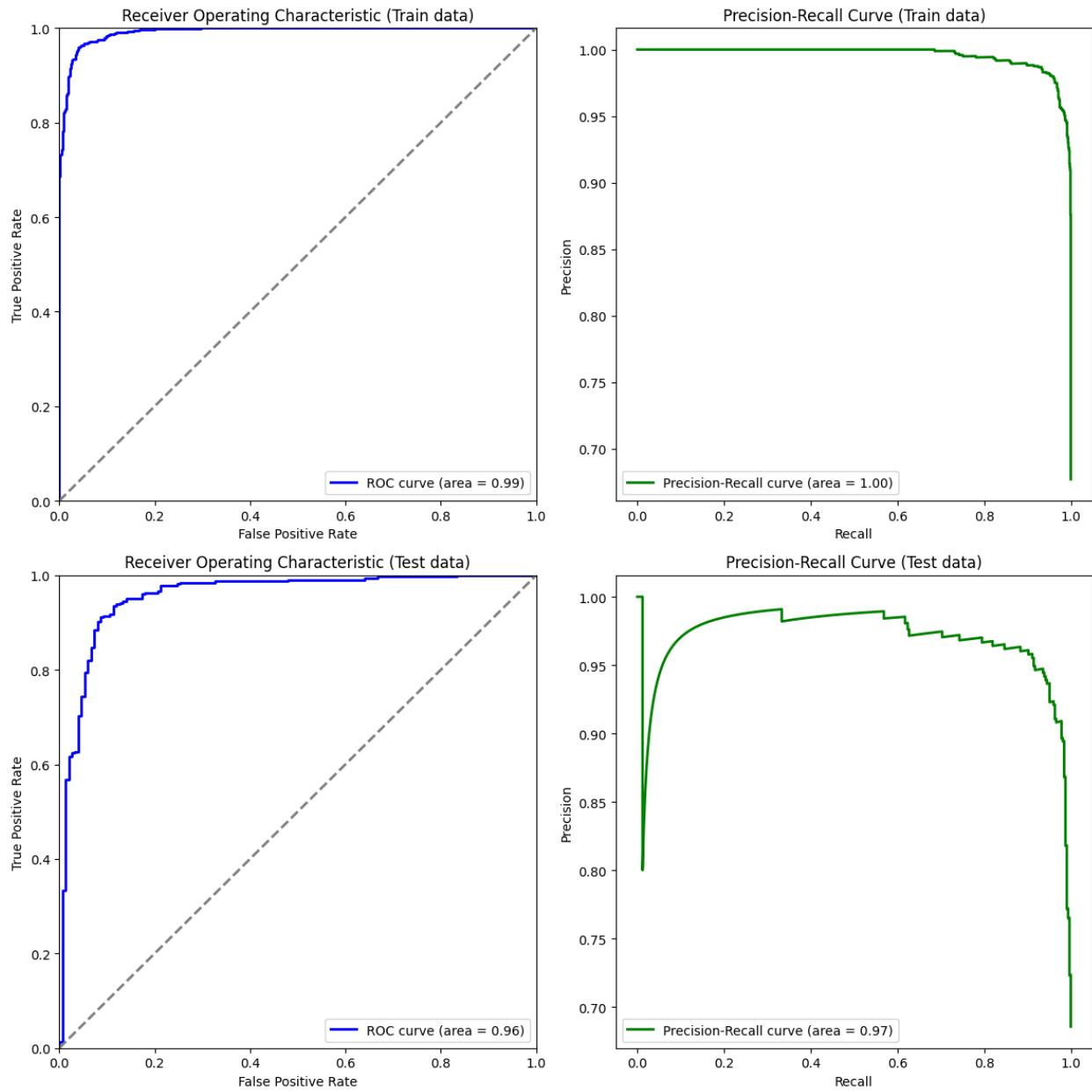
```

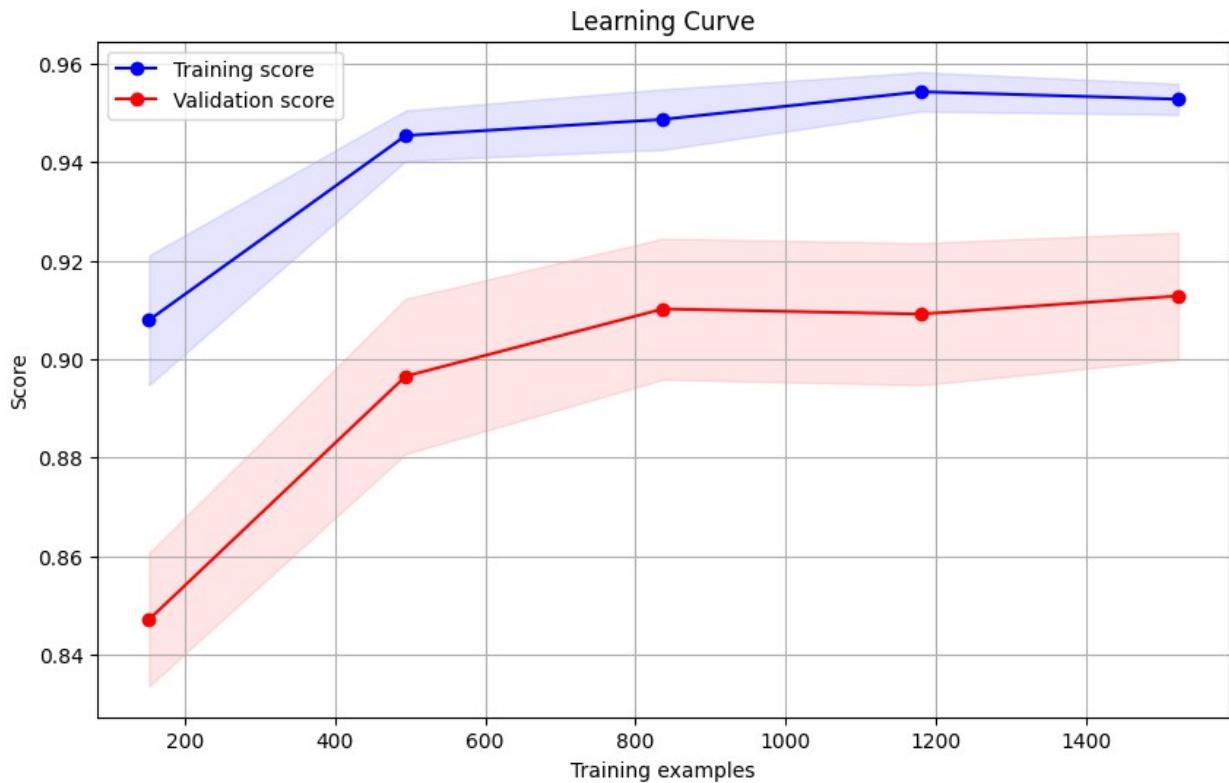
weighted avg	0.96	0.96	0.96	1904
--------------	------	------	------	------

#### Test Classification Report:

	precision	recall	f1-score	support
0	0.91	0.79	0.85	150
1	0.91	0.96	0.94	327
accuracy			0.91	477
macro avg	0.91	0.88	0.89	477
weighted avg	0.91	0.91	0.91	477







MLFL0W Logging is completed

## Balanced Dataset

### Hyper parameter Tuning

```
# Start timing the tuning process
start_tune_time = time.time()

# Define the parameter grid
param_dist = {
    'n_estimators': stats.randint(100, 1000),           # Number of
    'boosting rounds':                                # Learning rate
    'learning_rate': stats.uniform(0.01, 0.3),          # Maximum depth
    ('shrinkage factor'):                            # Minimum sum
    'max_depth': stats.randint(3, 15),                 # Minimum loss
    ('of a tree'):                                     # Subsample
    'min_child_weight': stats.randint(1, 10),           # Subsample
    ('of instance weight (hessian) needed in a child'): # Subsample
    'gamma': stats.uniform(0, 0.5),                     # Subsample
    ('reduction required to make a split'):            # Subsample
    'subsample': stats.uniform(0.7, 0.3),               # Subsample
    ('ratio of the training instance'):                  # Subsample
    'colsample_bytree': stats.uniform(0.7, 0.3),        # Subsample
    ('ratio of columns when constructing each tree'):  # Subsample
```

```

    'colsample_bylevel': stats.uniform(0.7, 0.3),      # Subsample
ratio of columns for each level of a tree
    'colsample_bynode': stats.uniform(0.7, 0.3),       # Subsample
ratio of columns for each node (split)
    'reg_alpha': stats.uniform(0, 0.5),                 # L1
regularization term on weights
    'reg_lambda': stats.uniform(0.5, 1.5),              # L2
regularization term on weights
    'scale_pos_weight': stats.uniform(0.5, 2),          # Balance of
positive and negative weights
    'booster': ['gbtree', 'gblinear', 'dart'],          # Type of
booster to use
    'tree_method': ['auto', 'exact', 'approx', 'hist'],# Algorithm
used to train trees
    'grow_policy': ['depthwise', 'lossguide'],           # Controls the
way new nodes are added to the tree
    'objective': ['binary:logistic', 'multi:softprob'],# Learning task
and the corresponding objective function
    'sampling_method': ['uniform', 'gradient_based'],  # Method used
to sample training data
    'random_state': [42]                                # Fixed random
state for reproducibility
}

# Initialize the XGBClассifier
xgb_clf = XGBClassifier(use_label_encoder=False,
eval_metric='logloss')

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=xgb_clf,
    param_distributions=param_dist,
    n_iter=200, # Number of parameter settings to try
    cv=5, # Number of folds in cross-validation
    verbose=1,
    random_state=42,
    n_jobs=-1 # Use all available cores
)

# Fit RandomizedSearchCV
random_search.fit(X_train_bal, y_train_bal)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_

# End timing and print tuning time
end_tune_time = time.time()

```

```

tuning_time = end_tune_time - start_tune_time
print("Tuning_time:", tuning_time)

Fitting 5 folds for each of 200 candidates, totalling 1000 fits
Best parameters found: {'booster': 'dart', 'colsample_bylevel':
0.8350373297259278, 'colsample_bynode': 0.7680894083096093,
'colsample_bytree': 0.9668172748386532, 'gamma': 0.22530982836487895,
'grow_policy': 'depthwise', 'learning_rate': 0.055050700245112494,
'max_depth': 14, 'min_child_weight': 1, 'n_estimators': 186,
'objective': 'binary:logistic', 'random_state': 42, 'reg_alpha':
0.023915410197052733, 'reg_lambda': 0.5570327723561823,
'sampling_method': 'uniform', 'scale_pos_weight': 1.9326467761253645,
'subsample': 0.721625300799602, 'tree_method': 'hist'}
Best score: 0.9185466997817416
Tuning_time: 1073.087533712387

```

### Logging Best XG boost Model into MLFLOW

```

# Model details
name = "Tuned_XG_boost_on_Balanced_Dataset"
bal_type = "Balanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time
start_train_time = time.time()
model.fit(X_train_bal, y_train_bal)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_bal_train = model.predict(X_train_bal)
y_pred_bal_test = model.predict(X_test_bal)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df,
mlflow_logging_and_metric_printing

```

```

time_df,feature_importance_df =
log_time_and_feature_importances_df(time_df,feature_importance_df,name
,training_time,testing_time,model,ola_features,bal_type,tuning_time)
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_bal,y_t
rain_bal,X_test_bal,y_test_bal,y_pred_bal_train,y_pred_bal_test,tuning
_score,**params)

Model: Tuned_XG_boost_on_Balanced_Dataset
params: {'booster': 'dart', 'colsample_bylevel': 0.8350373297259278,
'colsample_bynode': 0.7680894083096093, 'colsample_bytree':
0.9668172748386532, 'gamma': 0.22530982836487895, 'grow_policy':
'depthwise', 'learning_rate': 0.055050700245112494, 'max_depth': 14,
'min_child_weight': 1, 'n_estimators': 186, 'objective':
'binary:logistic', 'random_state': 42, 'reg_alpha':
0.023915410197052733, 'reg_lambda': 0.5570327723561823,
'sampling_method': 'uniform', 'scale_pos_weight': 1.9326467761253645,
'subsample': 0.721625300799602, 'tree_method': 'hist'}
Training Time: 4.4050 seconds
Testing Time: 0.0725 seconds
Tuning Time: 1073.0875 seconds
Train Metrics:
Accuracy_train: 0.9996
Precision_train: 0.9992
Recall_train: 1.0000
F1_score_train: 0.9996
F2_score_train: 0.9998
Roc_auc_train: 1.0000
Pr_auc_train: 1.0000

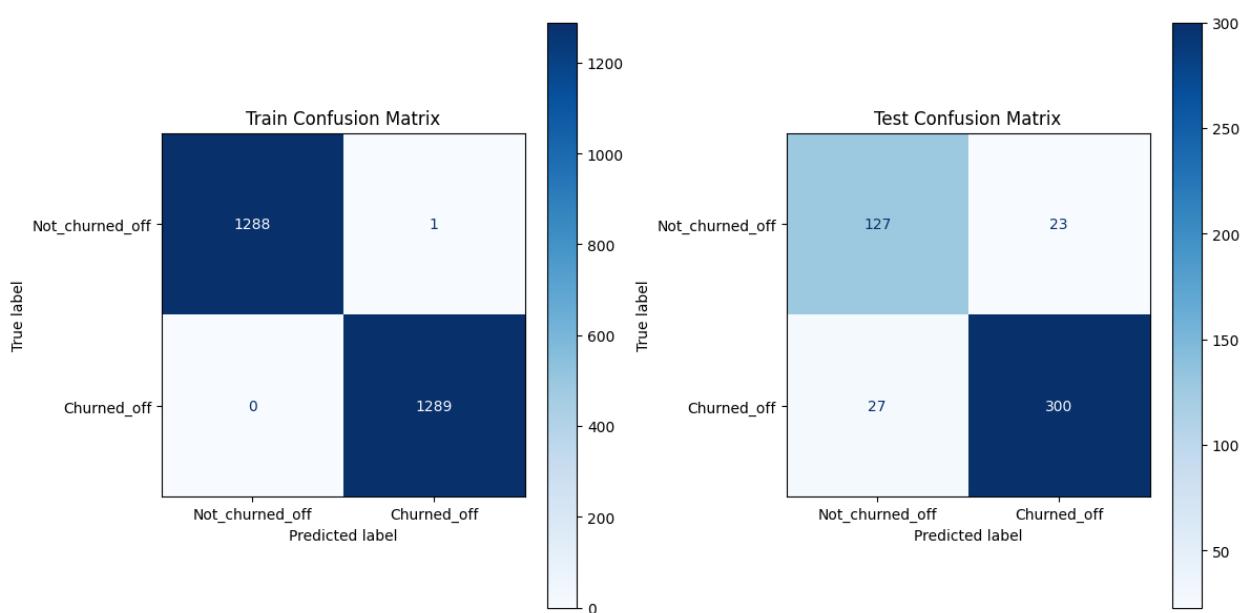
Test Metrics:
Accuracy_test: 0.8952
Precision_test: 0.9288
Recall_test: 0.9174
F1_score_test: 0.9231
F2_score_test: 0.9197
Roc_auc_test: 0.9477
Pr_auc_test: 0.9652

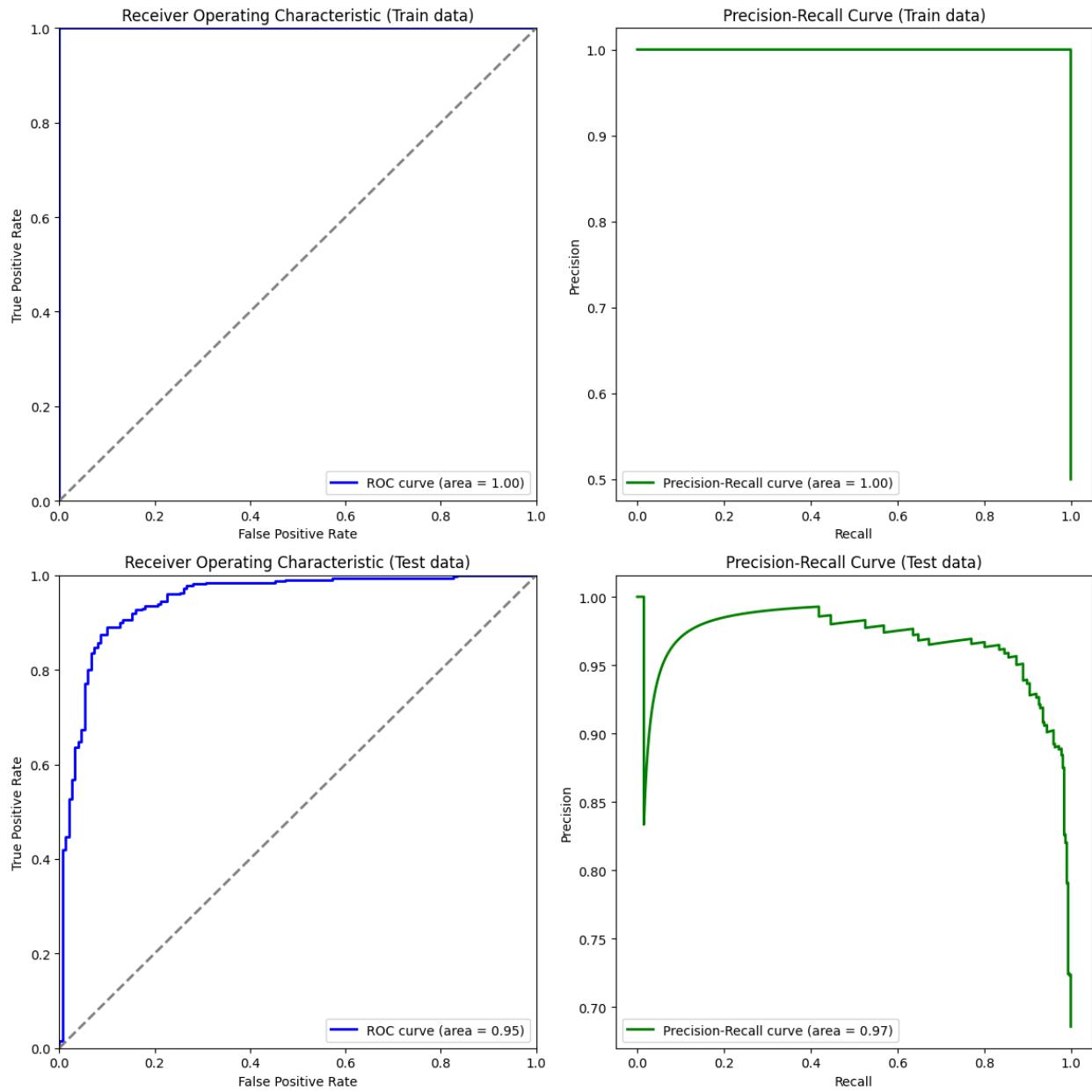
Tuning Metrics:
hyper_parameter_tuning_best_est_score: 0.9185

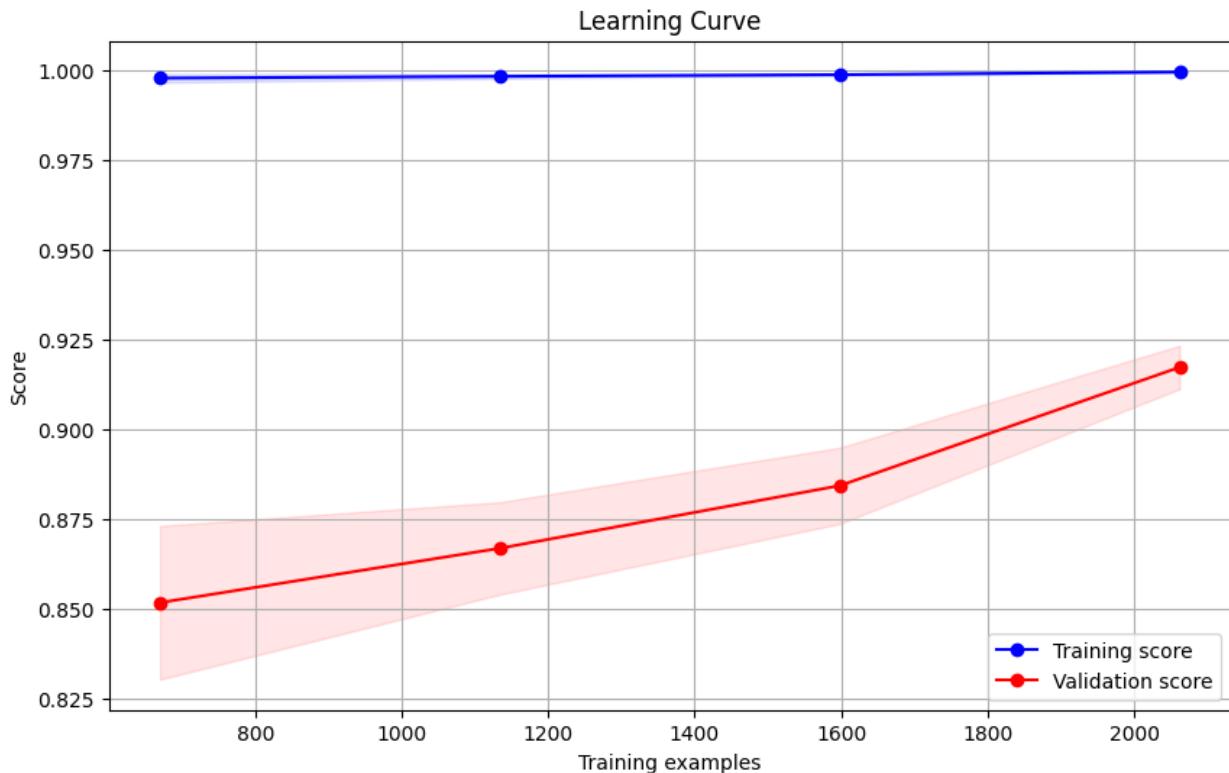
Train Classification Report:
      precision    recall   f1-score   support
          0         1.00     1.00     1.00     1289
          1         1.00     1.00     1.00     1289
          accuracy                           1.00     2578
          macro avg       1.00     1.00     1.00     2578
          weighted avg    1.00     1.00     1.00     2578

```

Test Classification Report:				
	precision	recall	f1-score	support
0	0.82	0.85	0.84	150
1	0.93	0.92	0.92	327
accuracy			0.90	477
macro avg	0.88	0.88	0.88	477
weighted avg	0.90	0.90	0.90	477







MLFLOW Logging is completed

## LIGHT\_GB MODEL

### Imbalanced Dataset

#### Hyper parameter Tuning

```
from lightgbm import LGBMClassifier
from sklearn.model_selection import RandomizedSearchCV
from scipy import stats
import time
import warnings

# Start timing the tuning process
start_tune_time = time.time()

# Define the parameter grid
param_dist = {
    'num_leaves': stats.randint(20, 150),                      # Number of leaves in one tree
    'max_depth': stats.randint(3, 15),                           # Maximum tree depth for base learners
    'learning_rate': stats.uniform(0.01, 0.3),                  # Boosting learning rate
}
```

```

'n_estimators': stats.randint(100, 2000),           # Number of
booster rounds
'min_child_samples': stats.randint(10, 100),         # Minimum
number of data needed in a child (leaf)
'min_child_weight': stats.uniform(1e-3, 1e-1),       # Minimum sum
of instance weight (hessian) needed in a child (leaf)
'subsample': stats.uniform(0.5, 1.0),                 # Subsample
ratio of the training instance
'colsample_bytree': stats.uniform(0.5, 1.0),          # Subsample
ratio of columns when constructing each tree
'reg_alpha': stats.uniform(0, 0.5),                   # L1
regularization term on weights
'reg_lambda': stats.uniform(0.5, 1.5),                # L2
regularization term on weights
'scale_pos_weight': stats.uniform(0.5, 2),            # Control the
balance of positive and negative weights
'boosting_type': ['gbdt', 'dart', 'goss'],             # Boosting type
'objective': ['binary', 'multiclass'],                # Objective
function
'bagging_fraction': stats.uniform(0.5, 1.0),           # Fraction of
data to use for each iteration (randomly selected)
'bagging_freq': stats.randint(1, 10),                  # Frequency of
bagging
'feature_fraction': stats.uniform(0.5, 1.0),           # Fraction of
features to consider at each iteration
'min_split_gain': stats.uniform(0, 0.1),                # Minimum gain
to make a split
'min_data_in_leaf': stats.randint(20, 100),             # Minimum
number of data points in a leaf node
'random_state': [42],                                  # Fixed random
state for reproducibility
}

# Initialize the LGBMClassifier
lgbm_clf = LGBMClassifier()

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=lgbm_clf,
    param_distributions=param_dist,
    n_iter=200,  # Number of parameter settings to try
    cv=5,  # Number of folds in cross-validation
    verbose=False,
    random_state=42,
    n_jobs=-1  # Use all available cores
)

# Fit RandomizedSearchCV
random_search.fit(X_train_imb, y_train_imb)

```

```

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_

# End timing and print tuning time
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning_time:", tuning_time)
warnings.filterwarnings('ignore')

```

Best parameters found: {'bagging\_fraction': 0.7040356346288988, 'bagging\_freq': 5, 'boosting\_type': 'dart', 'colsample\_bytree': 1.1159854502601791, 'feature\_fraction': 0.6880247346154161, 'learning\_rate': 0.11661537060572182, 'max\_depth': 11, 'min\_child\_samples': 45, 'min\_child\_weight': 0.0015229613542912736, 'min\_data\_in\_leaf': 77, 'min\_split\_gain': 0.003531135494023907, 'n\_estimators': 308, 'num\_leaves': 33, 'objective': 'binary', 'random\_state': 42, 'reg\_alpha': 0.4790367400596901, 'reg\_lambda': 1.0519111269531267, 'scale\_pos\_weight': 1.1538632326761582, 'subsample': 0.6488880503324447}

Best score: 0.9149026108578534

Tuning\_time: 31.56225085258484

## Logging Best Light GBM Model into MLFLOW

```

# Model details
name = "Tuned_Light_GBM_on_Imbalanced_Dataset"
bal_type = "Imbalanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time
start_train_time = time.time()
model.fit(X_train_imb, y_train_imb)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_imb_train = model.predict(X_train_imb)
y_pred_imb_test = model.predict(X_test_imb)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")

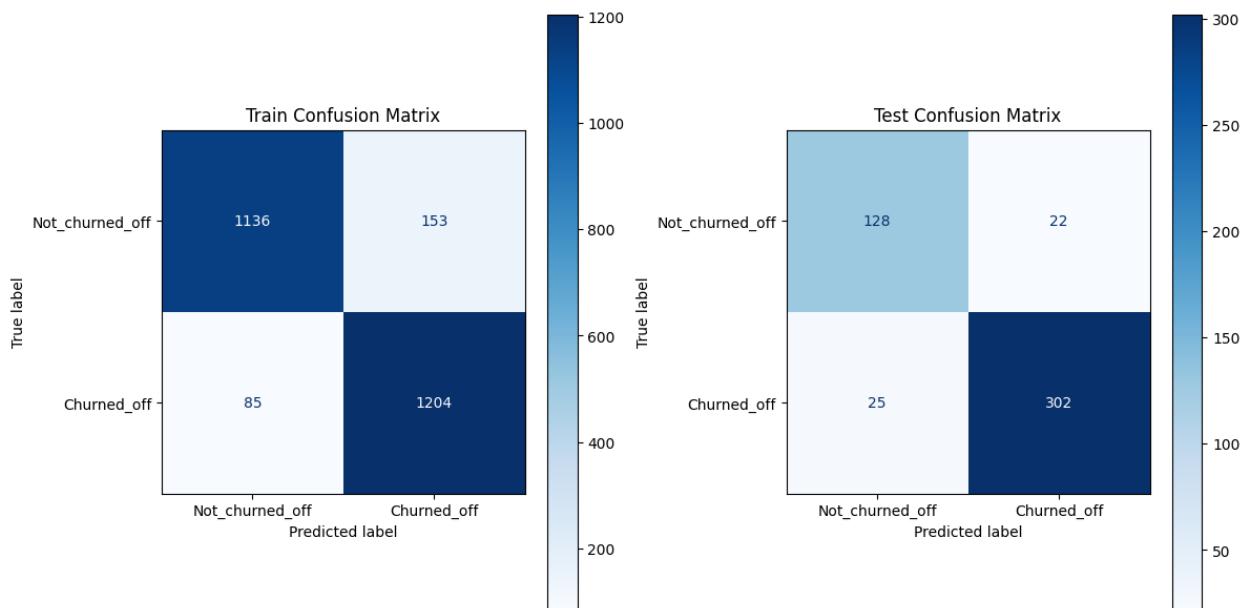
```

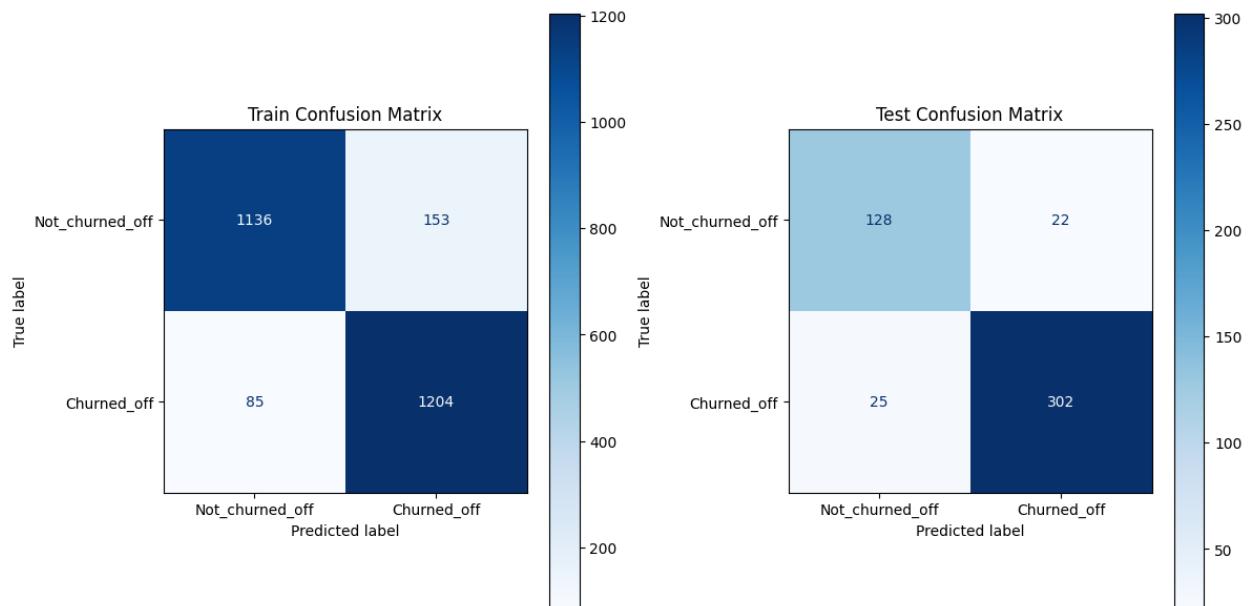
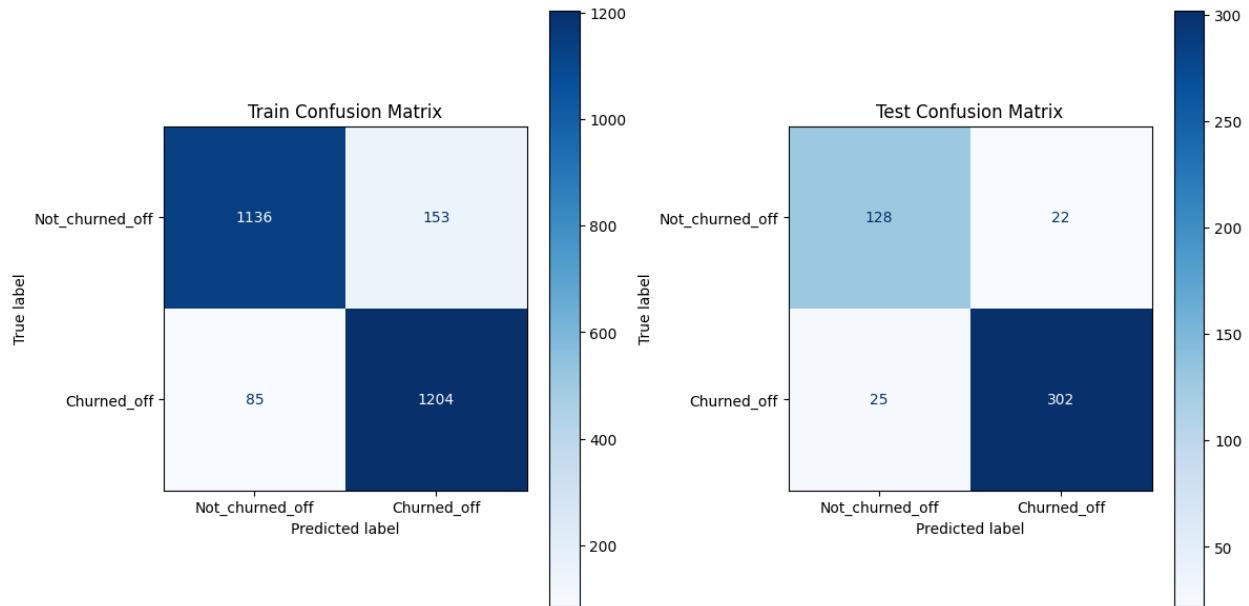
```

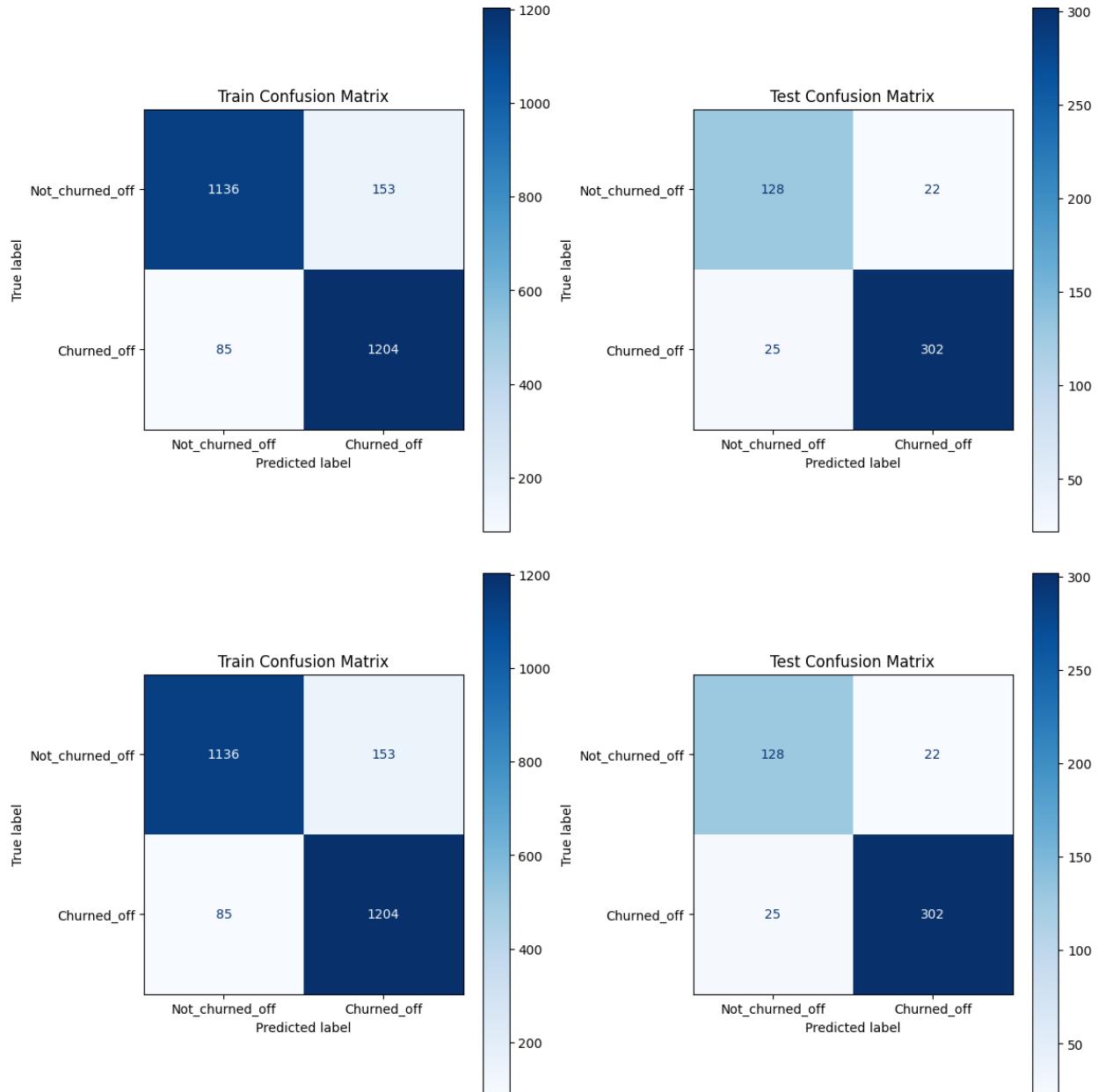
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df,
mlflow_logging_and_metric_printing
time_df,feature_importance_df =
log_time_and_feature_importances_df(time_df,feature_importance_df,name
,training_time,testing_time,model,ola_features,bal_type,tuning_time)
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_imb,y_t
rain_imb,X_test_imb,y_test_imb,y_pred_imb_train,y_pred_imb_test,tuning
_score,**params)

```







## Balanced Dataset

### Hyper parameter Tuning

```
from lightgbm import LGBMClassifier
from sklearn.model_selection import RandomizedSearchCV
from scipy import stats
import time
import warnings

# Start timing the tuning process
start_tune_time = time.time()
```

```

# Define the parameter grid
param_dist = {
    'num_leaves': stats.randint(20, 150), # Number of leaves in one tree
    'max_depth': stats.randint(3, 15), # Maximum tree depth for base learners
    'learning_rate': stats.uniform(0.01, 0.3), # Boosting learning rate
    'n_estimators': stats.randint(100, 2000), # Number of boosting rounds
    'min_child_samples': stats.randint(10, 100), # Minimum number of data needed in a child (leaf)
    'min_child_weight': stats.uniform(1e-3, 1e-1), # Minimum sum of instance weight (hessian) needed in a child (leaf)
    'subsample': stats.uniform(0.5, 1.0), # Subsample ratio of the training instance
    'colsample_bytree': stats.uniform(0.5, 1.0), # Subsample ratio of columns when constructing each tree
    'reg_alpha': stats.uniform(0, 0.5), # L1 regularization term on weights
    'reg_lambda': stats.uniform(0.5, 1.5), # L2 regularization term on weights
    'scale_pos_weight': stats.uniform(0.5, 2), # Control the balance of positive and negative weights
    'boosting_type': ['gbdt', 'dart', 'goss'], # Boosting type
    'objective': ['binary', 'multiclass'], # Objective function
    'bagging_fraction': stats.uniform(0.5, 1.0), # Fraction of data to use for each iteration (randomly selected)
    'bagging_freq': stats.randint(1, 10), # Frequency of bagging
    'feature_fraction': stats.uniform(0.5, 1.0), # Fraction of features to consider at each iteration
    'min_split_gain': stats.uniform(0, 0.1), # Minimum gain to make a split
    'min_data_in_leaf': stats.randint(20, 100), # Minimum number of data points in a leaf node
    'random_state': [42], # Fixed random state for reproducibility
}

# Initialize the LGBMClassifier
lgbm_clf = LGBMClassifier()

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=lgbm_clf,
    param_distributions=param_dist,
    n_iter=200, # Number of parameter settings to try
)

```

```

        cv=5, # Number of folds in cross-validation
        verbose=False,
        random_state=42,
        n_jobs=-1 # Use all available cores
    )

# Fit RandomizedSearchCV
random_search.fit(X_train_bal, y_train_bal)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_

# End timing and print tuning time
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning_time:", tuning_time)
warnings.filterwarnings('ignore')

```

Best parameters found: {'bagging\_fraction': 0.795633685837714, 'bagging\_freq': 5, 'boosting\_type': 'dart', 'colsample\_bytree': 0.9565345704829102, 'feature\_fraction': 0.7184404372168336, 'learning\_rate': 0.13495298436110986, 'max\_depth': 13, 'min\_child\_samples': 98, 'min\_child\_weight': 0.03280034749718639, 'min\_data\_in\_leaf': 20, 'min\_split\_gain': 0.02279351625419417, 'n\_estimators': 732, 'num\_leaves': 135, 'objective': 'binary', 'random\_state': 42, 'reg\_alpha': 0.4303652916281717, 'reg\_lambda': 0.510428195796786, 'scale\_pos\_weight': 1.5214946051551315, 'subsample': 0.917411003148779}

Best score: 0.9104056596673441

Tuning\_time: 41.100813150405884

Logging Best Light GBM boost Model into MLFLOW

```

# Model details
name = "Tuned_Light_GBM_on_Balanced_Dataset"
bal_type = "Balanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time
start_train_time = time.time()
model.fit(X_train_bal, y_train_bal)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()

```

```

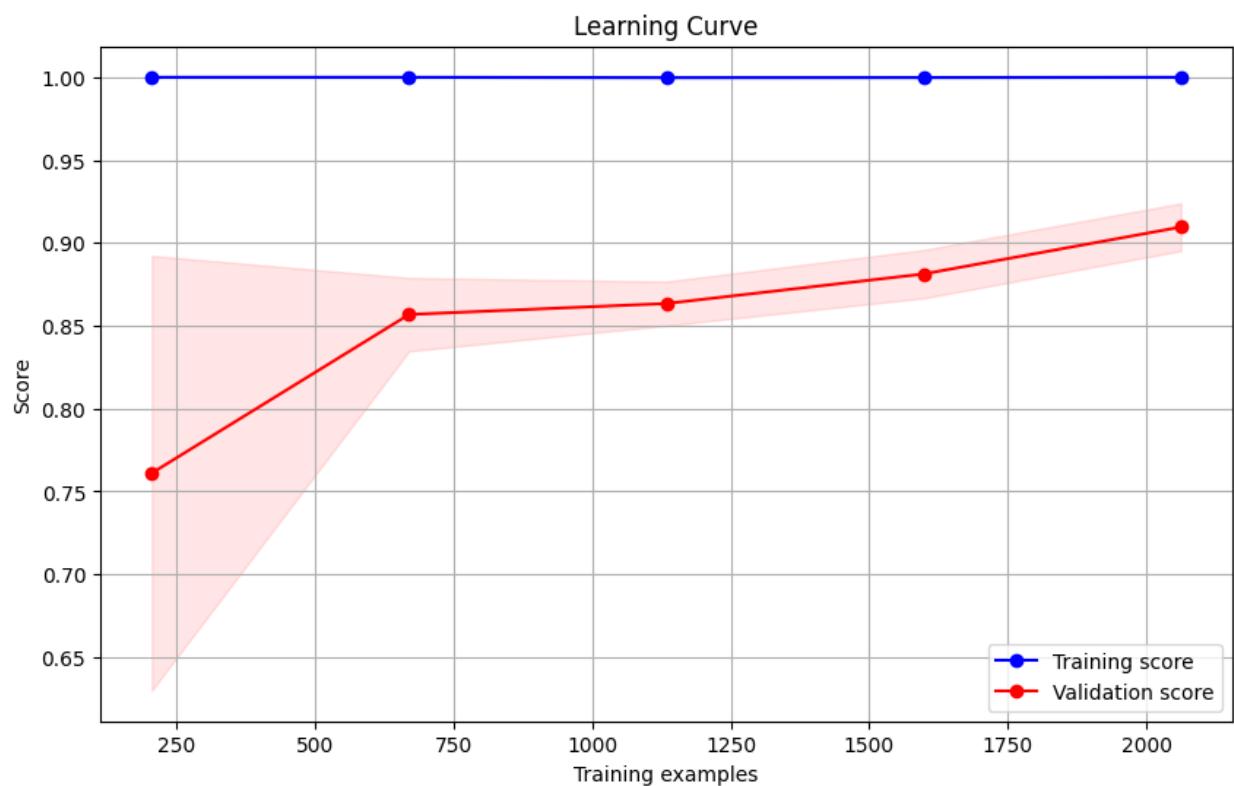
y_pred_bal_train = model.predict(X_train_bal)
y_pred_bal_test = model.predict(X_test_bal)
end_test_time = time.time()

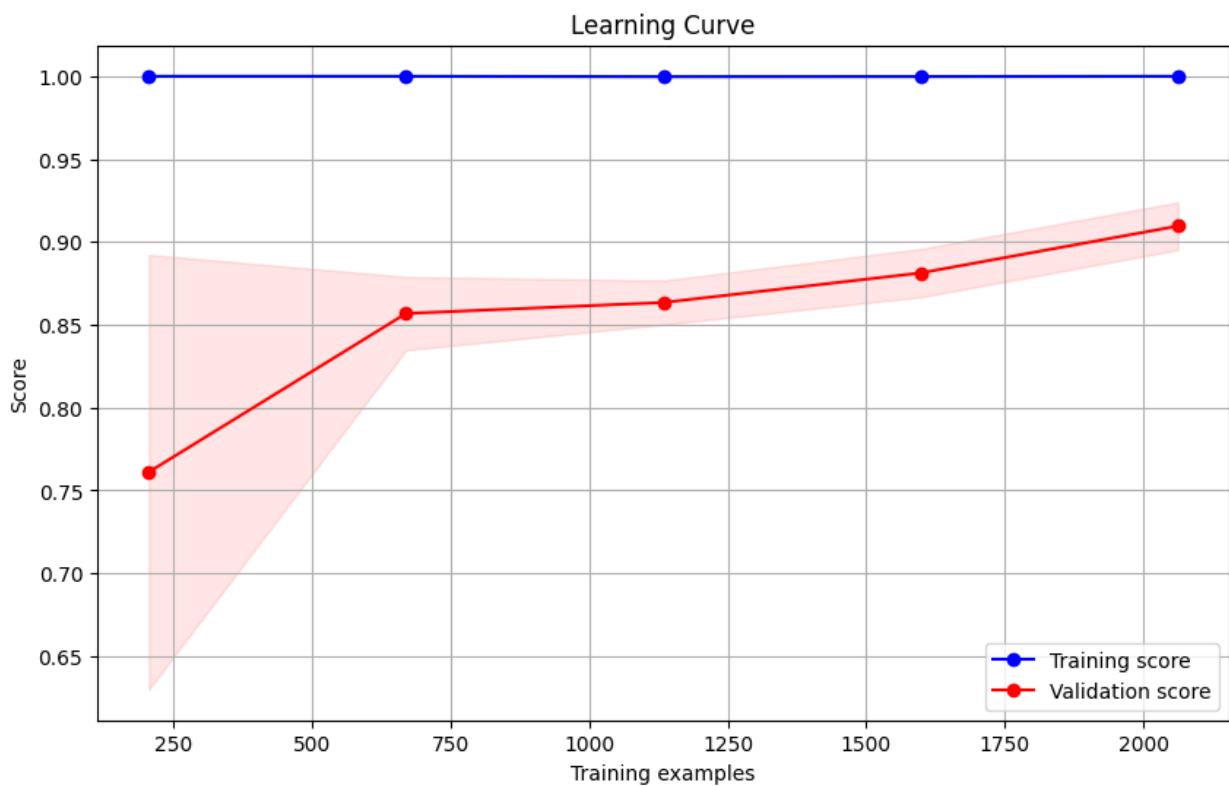
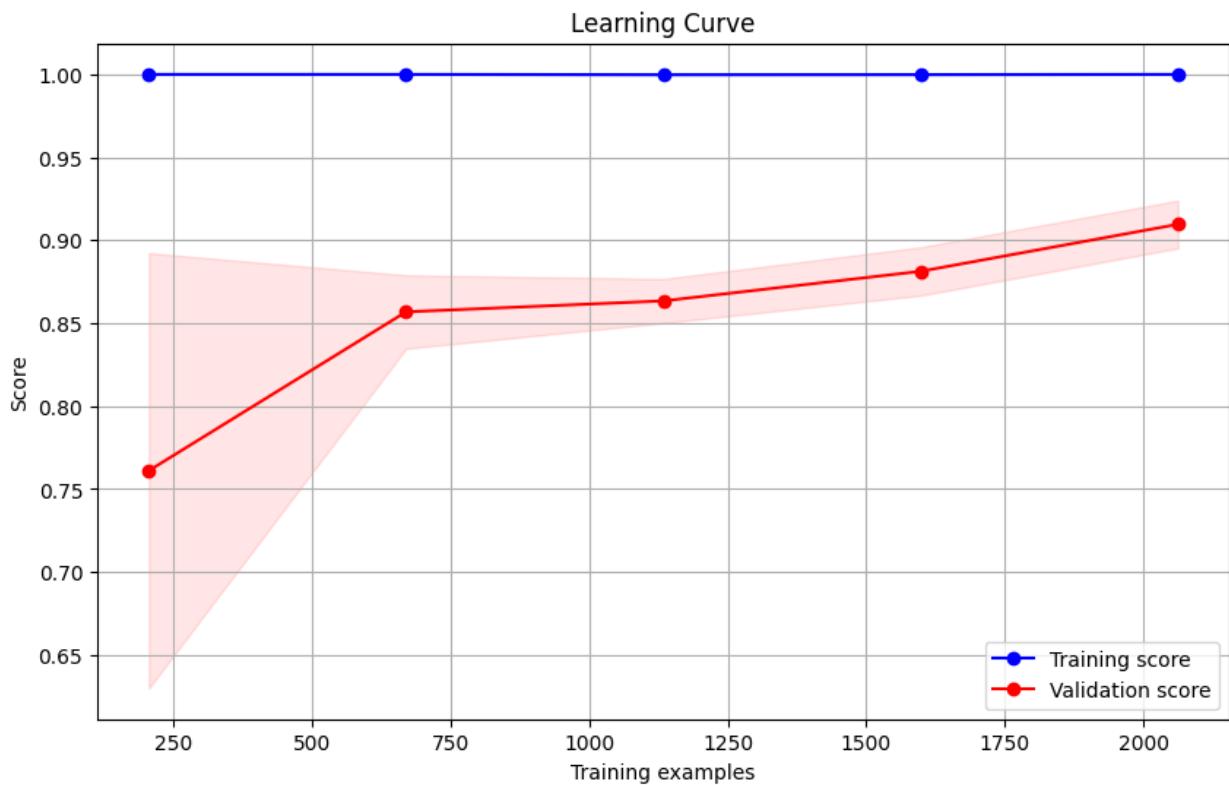
# Calculate testing time
testing_time = end_test_time - start_test_time

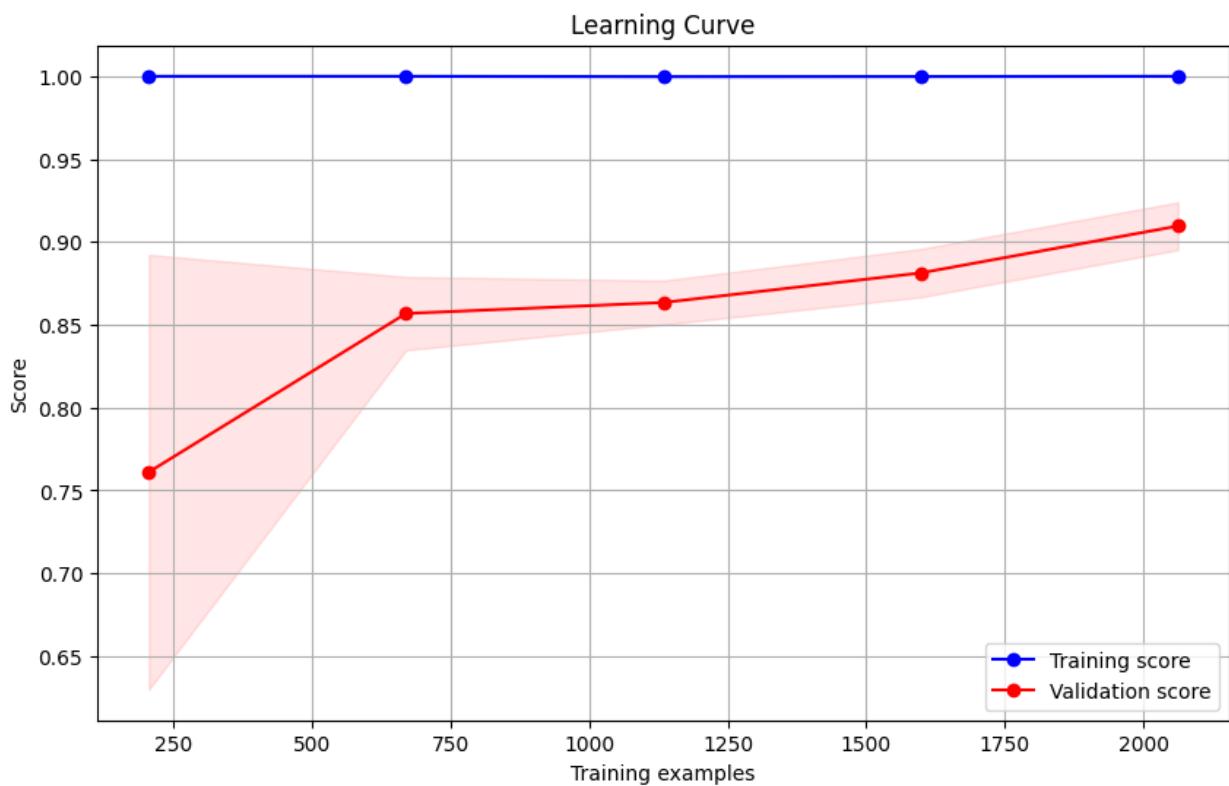
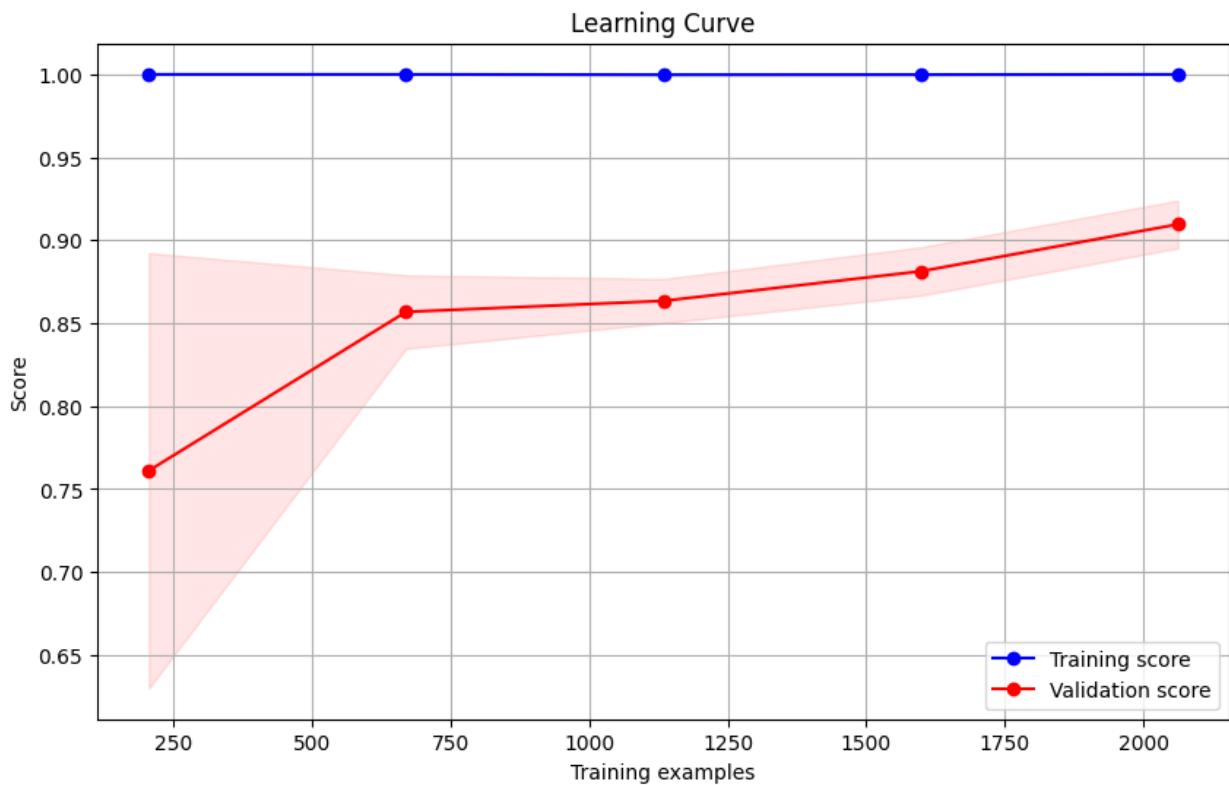
# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df,
mlflow_logging_and_metric_printing
time_df,feature_importance_df =
log_time_and_feature_importances_df(time_df,feature_importance_df,name
,training_time,testing_time,model,ola_features,bal_type,tuning_time)
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_bal,y_t
rain_bal,X_test_bal,y_test_bal,y_pred_bal_train,y_pred_bal_test,tuning
_score,**params)

```







# VOTING CLASSIFIER MODEL

## Imbalanced Dataset

Loading all the models from mlflow

```
def load_model_from_run(run_id, run_name):
    # Construct the model URI
    model_uri = f"runs:{run_id}/{run_name}_model"

    try:
        # Load the model
        model = mlflow.sklearn.load_model(model_uri)
        print(f"Loaded model from run {run_id} with run name {run_name}")
        return model
    except Exception as e:
        print(f"Failed to load model from run {run_id}: {e}")
        return None

# Example usage to load models
experiment = mlflow.get_experiment_by_name("Ola_Ensemble_Learning")
experiment_id = experiment.experiment_id

# Search for runs
runs = mlflow.search_runs(experiment_ids=[experiment_id])
run_ids = runs['run_id'].tolist()

# Load models
models = []
for run_id in run_ids:
    run = mlflow.get_run(run_id)
    run_name = run.info.run_name
    model = load_model_from_run(run_id, run_name)
    if model:
        models.append((f"model_{run_id}", model))
# models = [i[1] for i in models]
print(models)

Loaded model from run eba5fab22ac34697bed53bb627af2c33 with run name Tuned_Light_GBM_on_Balanced_Dataset
Loaded model from run 0c81cb30c10d4f56a99f55b2e9fb2d4c with run name Tuned_Light_GBM_on_Imbalanced_Dataset
Loaded model from run 32171425fbf342ee9f549d502105a681 with run name Tuned_XG_boost_on_Balanced_Dataset
Loaded model from run 2adb1674b77d4b9ba15c0c8978e82d50 with run name Tuned_XG_boost_on_Imbalanced_Dataset
Loaded model from run 2576e8de75f841a5bbfc9aa9108e406d with run name Tuned_Gradient_boost_on_Balanced_Dataset
Loaded model from run aef92e3ec93c485bb2457c0a4bd209d8 with run name
```

```
Tuned_Gradient_boost_on_Imbalanced_Dataset
Loaded model from run 6dce7745cea54262abfdd53e1081031c with run name
Tuned_Adaboost_on_Balanced_Dataset
Loaded model from run 0d740d00e0094d11b582f53aed58a72b with run name
Tuned_Adaboost_on_Imbalanced_Dataset
Loaded model from run 13b64ce985364b61ba859b705db9bac5 with run name
Tuned_Bagging_RF_on_Balanced_Dataset
Loaded model from run 907c9c90e757474bb2335a8d6155d157 with run name
Tuned_Bagging_RF_on_Imbalanced_Dataset
Loaded model from run 396f38da1e9d416bbd82023ff75cf227 with run name
Tuned_Random_Forest_on_Balanced_Dataset
Loaded model from run 60050ff4ee41446da38a6b38fa371520 with run name
Tuned_Random_Forest_on_Imbalanced_Dataset
Loaded model from run b64120f1fb98427ca7cce36345695bbf with run name
Tuned_Decision_Tree_on_Balanced_Dataset
Loaded model from run 43f861f9ac2041e0a2fd0663667b8129 with run name
Tuned_Decision_Tree_on_Imbalanced_Dataset
Loaded model from run 477ac0ff3c144d70af08a5747b5e8e96 with run name
Tuned_SVC_on_Balanced_Dataset
Loaded model from run 112add6166414f00887dcbea7fc6d86d with run name
Tuned_SVC_on_Imbalanced_Dataset
Loaded model from run 2897df0fb90d464c9c9195c03064411e with run name
Tuned_KNN_on_Balanced_Dataset
Loaded model from run 1c152ab6db5a4f2fb1a457823c9f65f with run name
Tuned_KNN_on_Imbalanced_Dataset
Loaded model from run 4caa37eaa4964998b5ff0b94dae599b1 with run name
Tuned_MLPClassifier_on_Balanced_Dataset
Loaded model from run 1d7993e0c3124955a9b383b40b15ea48 with run name
Tuned_MLPClassifier_on_Imbalanced_Dataset
Loaded model from run 62fc0473d24b4f71babf471575f220cf with run name
Tuned_Logistic_Regression_on_Balanced_Dataset
Loaded model from run a2514820d45d41e5883b697be7f14629 with run name
Tuned_Logistic_Regression_on_Imbalanced_Dataset
Loaded model from run efea588e45d8474f83c3b0771dfa5709 with run name
Simple_Logistic_Regresssion_on_balanced_Dataset
Loaded model from run e624cb2d315c46f5a563c2e82653f175 with run name
Simple_Logistic_Regression_on_Imbalanced_Dataset
[('model_eba5fab22ac34697bed53bb627af2c33',
LGBMClassifier(bagging_fraction=0.795633685837714, bagging_freq=5,
               boosting_type='dart',
               colsample_bytree=0.9565345704829102,
               feature_fraction=0.7184404372168336,
               learning_rate=0.13495298436110986, max_depth=13,
               min_child_samples=98,
               min_child_weight=0.03280034749718639,
               min_data_in_leaf=20,
               min_split_gain=0.02279351625419417,
               n_estimators=732, num_leaves=135, objective='binary',
               random_state=42, reg_alpha=0.4303652916281717,
```

```
        reg_lambda=0.510428195796786,
        scale_pos_weight=1.5214946051551315,
        subsample=0.917411003148779)),
('model_0c81cb30c10d4f56a99f55b2e9fb2d4c',
LGBMClassifier(bagging_fraction=0.7040356346288988, bagging_freq=5,
                boosting_type='dart',
colsample_bytree=1.1159854502601791,
                feature_fraction=0.6880247346154161,
                learning_rate=0.11661537060572182, max_depth=11,
                min_child_samples=45,
min_child_weight=0.0015229613542912736,
                min_data_in_leaf=77,
min_split_gain=0.003531135494023907,
                n_estimators=308, num_leaves=33, objective='binary',
random_state=42, reg_alpha=0.4790367400596901,
reg_lambda=1.0519111269531267,
                scale_pos_weight=1.1538632326761582,
                subsample=0.6488880503324447)),
('model_32171425fbf342ee9f549d502105a681',
XGBClassifier(base_score=None, booster='dart', callbacks=None,
                colsample_bylevel=0.8350373297259278,
                colsample_bynode=0.7680894083096093,
                colsample_bytree=0.9668172748386532, device=None,
                early_stopping_rounds=None, enable_categorical=False,
eval_metric='logloss', feature_types=None,
gamma=0.22530982836487895, grow_policy='depthwise',
importance_type=None, interaction_constraints=None,
learning_rate=0.055050700245112494, max_bin=None,
max_cat_threshold=None, max_cat_to_onehot=None,
max_delta_step=None, max_depth=14, max_leaves=None,
min_child_weight=1, missing=nan,
monotone_constraints=None,
                multi_strategy=None, n_estimators=186, n_jobs=None,
num_parallel_tree=None, random_state=42, ...)),
('model_2adb1674b77d4b9ba15c0c8978e82d50',
XGBClassifier(base_score=None, booster='dart', callbacks=None,
                colsample_bylevel=0.9526356769407125,
                colsample_bynode=0.9514986114133412,
                colsample_bytree=0.840607947938491, device=None,
                early_stopping_rounds=None, enable_categorical=False,
eval_metric='logloss', feature_types=None,
gamma=0.2074097511688326, grow_policy='depthwise',
importance_type=None, interaction_constraints=None,
learning_rate=0.014101589448099186, max_bin=None,
max_cat_threshold=None, max_cat_to_onehot=None,
max_delta_step=None, max_depth=9, max_leaves=None,
min_child_weight=6, missing=nan,
monotone_constraints=None,
                multi_strategy=None, n_estimators=575, n_jobs=None,
```

```
        num_parallel_tree=None, random_state=42, ...)),  
('model_2576e8de75f841a5bbfc9aa9108e406d',  
GradientBoostingClassifier(learning_rate=0.07858754845747705,  
max_depth=9,  
    max_features='log2',  
  
min_impurity_decrease=0.0001120111480109487,  
    min_samples_leaf=2, min_samples_split=15,  
    n_estimators=688, random_state=42,  
    subsample=0.8522403201282508)),  
('model_aef92e3ec93c485bb2457c0a4bd209d8',  
GradientBoostingClassifier(learning_rate=0.06676646193550176,  
max_depth=7,  
    max_features='log2',  
    min_impurity_decrease=0.09613152881849074,  
    min_samples_leaf=19, min_samples_split=11,  
    n_estimators=493, random_state=42,  
    subsample=0.94598407522243)),  
('model_6dce7745cea54262abfdd53e1081031c',  
AdaBoostClassifier(estimator=DecisionTreeClassifier(max_depth=5),  
    learning_rate=0.9000053418175663, n_estimators=520,  
    random_state=42)),  
('model_0d740d00e0094d11b582f53aed58a72b',  
AdaBoostClassifier(algorithm='SAMME',  
    estimator=DecisionTreeClassifier(max_depth=5),  
    learning_rate=0.1370605126518848, n_estimators=313,  
    random_state=42)),  
('model_13b64ce985364b61ba859b705db9bac5',  
BaggingClassifier(bootstrap=False,  
  
estimator=RandomForestClassifier(class_weight='balanced',  
  
criterion='log_loss',  
    max_depth=20,  
    max_leaf_nodes=20,  
    n_estimators=5,  
    oob_score=True,  
    random_state=42),  
    max_features=0.9631385219890038,  
    max_samples=0.9803599686384168, n_estimators=154,  
    random_state=42)),  
('model_907c9c90e757474bb2335a8d6155d157',  
BaggingClassifier(bootstrap=False,  
  
estimator=RandomForestClassifier(class_weight='balanced',  
  
criterion='log_loss',  
    max_depth=20,  
    max_leaf_nodes=20,  
    n_estimators=5,
```

```
          oob_score=True,
          random_state=42),
max_features=0.9631385219890038,
max_samples=0.9803599686384168, n_estimators=154,
n_jobs=-1,
random_state=42)),
('model_396f38dale9d416bb82023ff75cf227',
RandomForestClassifier(ccp_alpha=0.028792961647572612,
class_weight='balanced',
criterion='log_loss', max_depth=13,
max_leaf_nodes=14,
min_impurity_decrease=0.013911619414306187,
min_samples_leaf=18,
min_weight_fraction_leaf=0.009055091910420254,
n_estimators=877, oob_score=True,
random_state=42)), ('model_60050ff4ee41446da38a6b38fa371520',
RandomForestClassifier(ccp_alpha=0.028792961647572612,
class_weight='balanced',
criterion='log_loss', max_depth=13,
max_leaf_nodes=14,
min_impurity_decrease=0.013911619414306187,
min_samples_leaf=18,
min_weight_fraction_leaf=0.009055091910420254,
n_estimators=877, oob_score=True,
random_state=42)), ('model_b64120f1fb98427ca7cce36345695bbf',
DecisionTreeClassifier(ccp_alpha=0.011483682473920355,
criterion='entropy',
max_depth=15, max_features='log2',
max_leaf_nodes=39,
min_impurity_decrease=0.05812382214226123,
min_samples_leaf=11, min_samples_split=5,
min_weight_fraction_leaf=0.16032109607975714,
random_state=42)),
('model_43f861f9ac2041e0a2fd0663667b8129',
DecisionTreeClassifier(ccp_alpha=0.01504168911035282, max_depth=18,
max_leaf_nodes=34,
min_impurity_decrease=0.046869315979497034,
min_samples_leaf=3,
min_weight_fraction_leaf=0.0068359824134986424,
random_state=42)),
('model_477ac0ff3c144d70af08a5747b5e8e96', SVC(C=81.5862458136845,
class_weight='balanced', degree=2, gamma='auto',
max_iter=1677, probability=True, random_state=42)),
('model_112add6166414f00887dcbea7fc6d86d', SVC(C=6.436932213118794,
class_weight='balanced', degree=4, gamma='auto',
max_iter=1130, probability=True, random_state=42)),
('model_2897df0fb90d464c9c9195c03064411e',
KNeighborsClassifier(algorithm='ball_tree', leaf_size=49,
n_neighbors=16, p=1,
```

```

        weights='distance'))),
('model_1c152ab6db5a4f2fbcl457823c9f65f',
KNeighborsClassifier(algorithm='kd_tree', leaf_size=28,
metric='manhattan',
           n_neighbors=11, p=3, weights='distance')),
('model_4caa37eaa4964998b5ff0b94dae599b1',
MLPClassifier(activation='logistic', alpha=3.893929205071642,
           hidden_layer_sizes=(50,), learning_rate_init=0.030524224295953774,
           max_iter=221, random_state=42)),
('model_1d7993e0c3124955a9b383b40b15ea48',
MLPClassifier(activation='logistic', alpha=3.893929205071642,
           hidden_layer_sizes=(50,), learning_rate_init=0.030524224295953774,
           max_iter=221, random_state=42)),
('model_62fc0473d24b4f71babf471575f220cf',
LogisticRegression(C=1872.706848835624, class_weight='balanced',
max_iter=10000,
           penalty='l1', random_state=42, solver='saga')),
('model_a2514820d45d41e5883b697be7f14629',
LogisticRegression(C=1872.706848835624, class_weight='balanced',
max_iter=10000,
           penalty='l1', random_state=42, solver='saga')),
('model_efea588e45d8474f83c3b0771dfa5709', LogisticRegression(n_jobs=-1,
random_state=42)), ('model_e624cb2d315c46f5a563c2e82653f175',
LogisticRegression(n_jobs=-1, random_state=42))]
```

## Hyper parameter Tuning

```

# Define the parameter grid for RandomizedSearchCV
start_tune_time = time.time()
param_dist = {
    'voting': ['hard', 'soft'],
    'weights': [
        None, # Default equal weights
        [1] * len(models[1::2]), # Equal weights
        list([1 / (i + 1) for i in range(len(models[1::2]))])
    ],
    'n_jobs': [-1]
}
# Initialize the VotingClassifier
voting_clf = VotingClassifier(estimators=models[1::2])

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=voting_clf,
    param_distributions=param_dist,
    n_iter=6, # Number of parameter settings to try
    cv=5, # Number of folds in cross-validation
    verbose=1,
```

```

        random_state=42,
        n_jobs=-1
    )

# Fit RandomizedSearchCV
random_search.fit(X_train_imb, y_train_imb)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning time:", tuning_time)

Fitting 5 folds for each of 6 candidates, totalling 30 fits
Best parameters found: {'weights': [1.0, 0.5, 0.3333333333333333,
0.25, 0.2, 0.1666666666666666, 0.14285714285714285, 0.125,
0.1111111111111111, 0.1, 0.09090909090909091, 0.0833333333333333],
'voting': 'hard', 'n_jobs': -1}
Best score: 0.9138541234977208
Tuning time: 1044.1496663093567

```

## Logging Best Voting Classifier Model into MLFLOW

```

# Model details
name = "Tuned_Voting_Classifier_on_Imbalanced_Dataset"
bal_type = "Imbalanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time
start_train_time = time.time()
model.fit(X_train_imb, y_train_imb)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_imb_train = model.predict(X_train_imb)
y_pred_imb_test = model.predict(X_test_imb)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")

```

```

print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df,
mlflow_logging_and_metric_printing
time_df,feature_importance_df =
log_time_and_feature_importances_df(time_df,feature_importance_df,name
,training_time,testing_time,model,ola_features,bal_type,tuning_time)
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_imb,y_t
rain_imb,X_test_imb,y_test_imb,y_pred_imb_train,y_pred_imb_test,tuning
_score,**params)

[LightGBM] [Warning] min_data_in_leaf is set=77, min_child_samples=45
will be ignored. Current value: min_data_in_leaf=77
[LightGBM] [Warning] feature_fraction is set=0.6880247346154161,
colsample_bytree=1.1159854502601791 will be ignored. Current value:
feature_fraction=0.6880247346154161
[LightGBM] [Warning] bagging_fraction is set=0.7040356346288988,
subsample=0.6488880503324447 will be ignored. Current value:
bagging_fraction=0.7040356346288988
[LightGBM] [Warning] bagging_freq is set=5, subsample_freq=0 will be
ignored. Current value: bagging_freq=5
[LightGBM] [Warning] min_data_in_leaf is set=77, min_child_samples=45
will be ignored. Current value: min_data_in_leaf=77
[LightGBM] [Warning] feature_fraction is set=0.6880247346154161,
colsample_bytree=1.1159854502601791 will be ignored. Current value:
feature_fraction=0.6880247346154161
[LightGBM] [Warning] bagging_fraction is set=0.7040356346288988,
subsample=0.6488880503324447 will be ignored. Current value:
bagging_fraction=0.7040356346288988
[LightGBM] [Warning] bagging_freq is set=5, subsample_freq=0 will be
ignored. Current value: bagging_freq=5
Model: Tuned_Voting_Classifier_on_Imbalanced_Dataset
params: {'weights': [1.0, 0.5, 0.3333333333333333, 0.25, 0.2,
0.1666666666666666, 0.14285714285714285, 0.125, 0.1111111111111111,
0.1, 0.09090909090909091, 0.0833333333333333], 'voting': 'hard',
'n_jobs': -1}
Training Time: 114.1385 seconds
Testing Time: 1.8519 seconds
Tuning Time: 1044.1497 seconds
Error in auc_plots: This 'VotingClassifier' has no attribute
'predict_proba'
Error in auc_plots: This 'VotingClassifier' has no attribute
'predict_proba'
Train Metrics:
Accuracy_train: 0.9417
Precision_train: 0.9462
Recall_train: 0.9690

```

```
F1_score_train: 0.9575  
F2_score_train: 0.9643
```

```
Test Metrics:  
Accuracy_test: 0.9161  
Precision_test: 0.9208  
Recall_test: 0.9602  
F1_score_test: 0.9401  
F2_score_test: 0.9521
```

```
Tuning Metrics:  
hyper_parameter_tuning_best_est_score: 0.9139
```

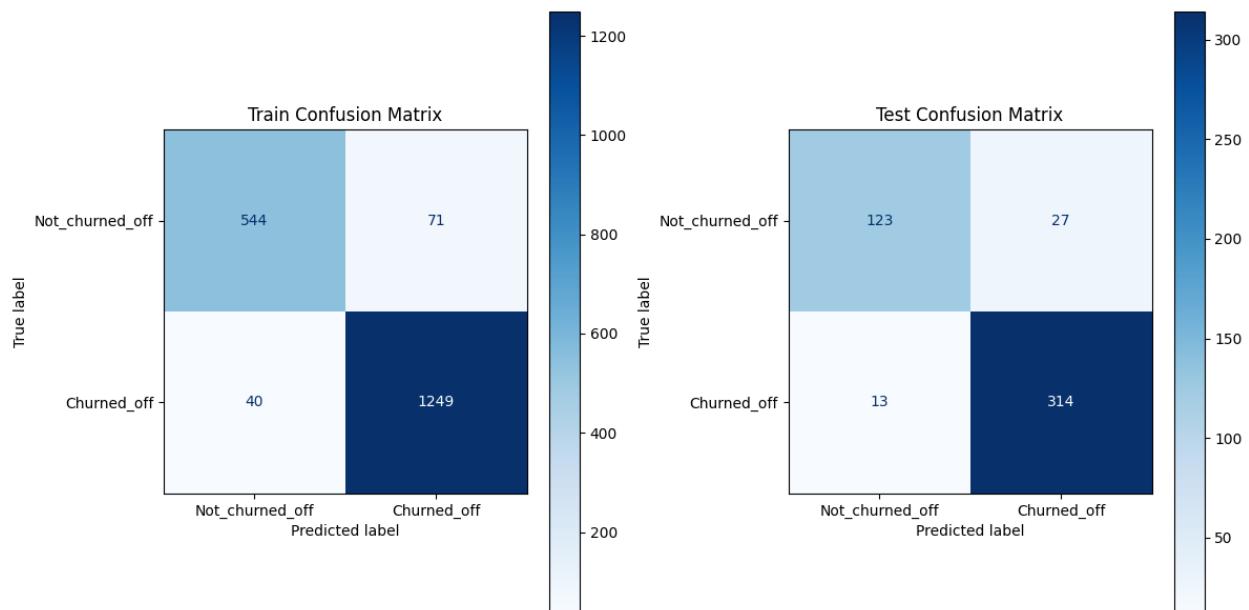
```
Train Classification Report:
```

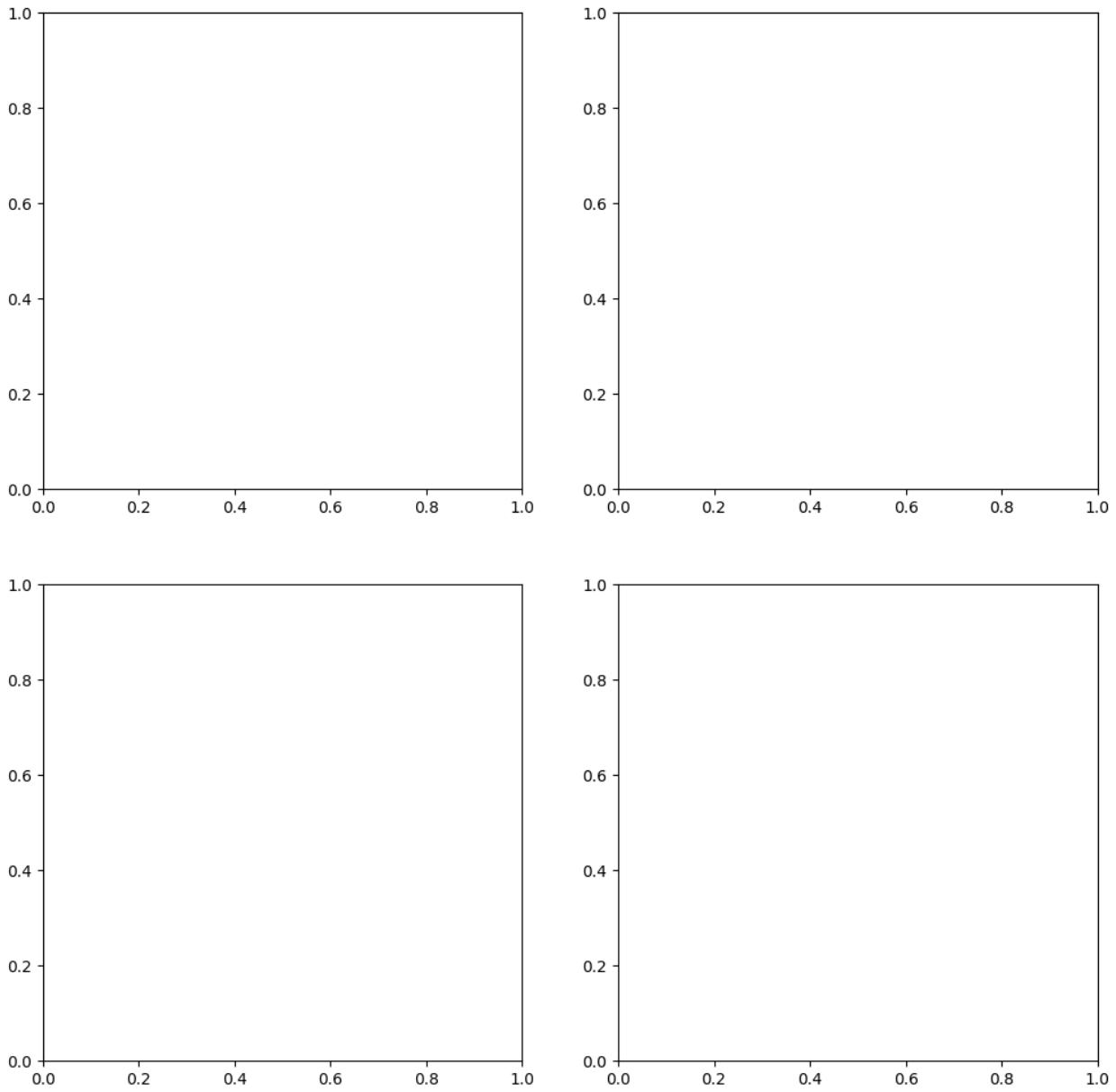
	precision	recall	f1-score	support
0	0.93	0.88	0.91	615
1	0.95	0.97	0.96	1289
accuracy			0.94	1904
macro avg	0.94	0.93	0.93	1904
weighted avg	0.94	0.94	0.94	1904

```
Test Classification Report:
```

	precision	recall	f1-score	support
0	0.90	0.82	0.86	150
1	0.92	0.96	0.94	327
accuracy			0.92	477
macro avg	0.91	0.89	0.90	477
weighted avg	0.92	0.92	0.91	477

```
Error in auc_plots: This 'VotingClassifier' has no attribute  
'predict_proba'
```





## Balanced Dataset

### Hyper parameter Tuning

```
# Define the parameter grid for RandomizedSearchCV
start_tune_time = time.time()
param_dist = {
    'voting': ['hard', 'soft'],
    'weights': [
        None, # Default equal weights
        [1] * len(models[::2]), # Equal weights
        list([1 / (i + 1) for i in range(len(models[::2]))]) # Decreasing
```

```

        ],
        'n_jobs': [-1]
    }
# Initialize the VotingClassifier
voting_clf = VotingClassifier(estimators=models[::2])

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=voting_clf,
    param_distributions=param_dist,
    n_iter=6, # Number of parameter settings to try
    cv=5, # Number of folds in cross-validation
    verbose=1,
    random_state=42,
    n_jobs=-1
)

# Fit RandomizedSearchCV
random_search.fit(X_train_bal, y_train_bal)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning time:", tuning_time)

Fitting 5 folds for each of 6 candidates, totalling 30 fits
Best parameters found: {'weights': [1.0, 0.5, 0.3333333333333333,
0.25, 0.2, 0.1666666666666666, 0.14285714285714285, 0.125,
0.1111111111111111, 0.1, 0.09090909090909091, 0.0833333333333333],
'veting': 'soft', 'n_jobs': -1}
Best score: 0.9169978174155189
Tuning time: 275.3588945865631

```

## Logging Best Voting Classifier Model into MLFLOW

```

# Model details
name = "Tuned_Voting_Classifier_on_Balanced_Dataset"
bal_type = "Balanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time
start_train_time = time.time()
model.fit(X_train_bal, y_train_bal)
end_train_time = time.time()

# Calculate training time

```

```

training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_bal_train = model.predict(X_train_bal)
y_pred_bal_test = model.predict(X_test_bal)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df,
mlflow_logging_and_metric_printing
time_df,feature_importance_df =
log_time_and_feature_importances_df(time_df,feature_importance_df,name
,training_time,testing_time,model,ola_features,bal_type,tuning_time)
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_bal,y_t
rain_bal,X_test_bal,y_test_bal,y_pred_bal_train,y_pred_bal_test,tuning_
score,**params)

[LightGBM] [Warning] min_data_in_leaf is set=20, min_child_samples=98
will be ignored. Current value: min_data_in_leaf=20
[LightGBM] [Warning] feature_fraction is set=0.7184404372168336,
colsample_bytree=0.9565345704829102 will be ignored. Current value:
feature_fraction=0.7184404372168336
[LightGBM] [Warning] bagging_fraction is set=0.795633685837714,
subsample=0.917411003148779 will be ignored. Current value:
bagging_fraction=0.795633685837714
[LightGBM] [Warning] bagging_freq is set=5, subsample_freq=0 will be
ignored. Current value: bagging_freq=5
[LightGBM] [Warning] min_data_in_leaf is set=20, min_child_samples=98
will be ignored. Current value: min_data_in_leaf=20
[LightGBM] [Warning] feature_fraction is set=0.7184404372168336,
colsample_bytree=0.9565345704829102 will be ignored. Current value:
feature_fraction=0.7184404372168336
[LightGBM] [Warning] bagging_fraction is set=0.795633685837714,
subsample=0.917411003148779 will be ignored. Current value:
bagging_fraction=0.795633685837714
[LightGBM] [Warning] bagging_freq is set=5, subsample_freq=0 will be
ignored. Current value: bagging_freq=5
Model: Tuned_Voting_Classifier_on_Balanced_Dataset
params: {'weights': [1.0, 0.5, 0.3333333333333333, 0.25, 0.2,
0.1666666666666666, 0.14285714285714285, 0.125, 0.1111111111111111,
```

```
0.1, 0.09090909090909091, 0.0833333333333333], 'voting': 'soft',
'n_jobs': -1}
Training Time: 20.0931 seconds
Testing Time: 1.7704 seconds
Tuning Time: 275.3589 seconds
[LightGBM] [Warning] min_data_in_leaf is set=20, min_child_samples=98
will be ignored. Current value: min_data_in_leaf=20
[LightGBM] [Warning] feature_fraction is set=0.7184404372168336,
colsample_bytree=0.9565345704829102 will be ignored. Current value:
feature_fraction=0.7184404372168336
[LightGBM] [Warning] bagging_fraction is set=0.795633685837714,
subsample=0.917411003148779 will be ignored. Current value:
bagging_fraction=0.795633685837714
[LightGBM] [Warning] bagging_freq is set=5, subsample_freq=0 will be
ignored. Current value: bagging_freq=5
[LightGBM] [Warning] min_data_in_leaf is set=20, min_child_samples=98
will be ignored. Current value: min_data_in_leaf=20
[LightGBM] [Warning] feature_fraction is set=0.7184404372168336,
colsample_bytree=0.9565345704829102 will be ignored. Current value:
feature_fraction=0.7184404372168336
[LightGBM] [Warning] bagging_fraction is set=0.795633685837714,
subsample=0.917411003148779 will be ignored. Current value:
bagging_fraction=0.795633685837714
[LightGBM] [Warning] bagging_freq is set=5, subsample_freq=0 will be
ignored. Current value: bagging_freq=5
Train Metrics:
Accuracy_train: 1.0000
Precision_train: 1.0000
Recall_train: 1.0000
F1_score_train: 1.0000
F2_score_train: 1.0000
Roc_auc_train: 1.0000
Pr_auc_train: 1.0000

Test Metrics:
Accuracy_test: 0.8994
Precision_test: 0.9346
Recall_test: 0.9174
F1_score_test: 0.9259
F2_score_test: 0.9208
Roc_auc_test: 0.9495
Pr_auc_test: 0.9719

Tuning Metrics:
hyper_parameter_tuning_best_est_score: 0.9170

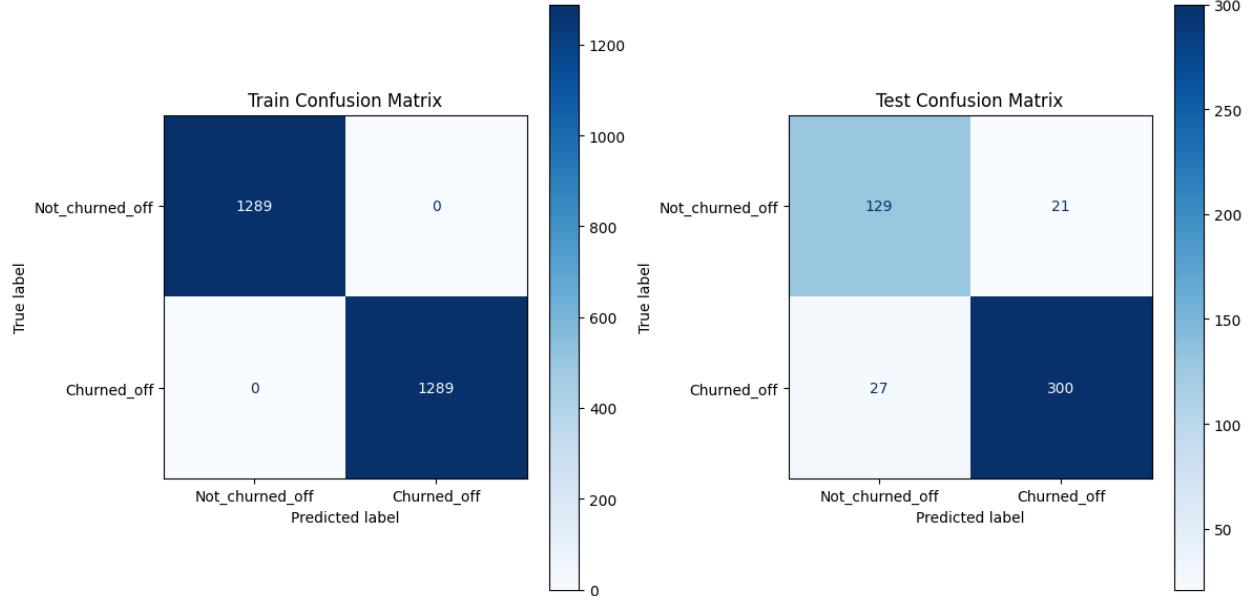
Train Classification Report:
precision    recall   f1-score   support
      0         1.00     1.00     1.00      1289
```

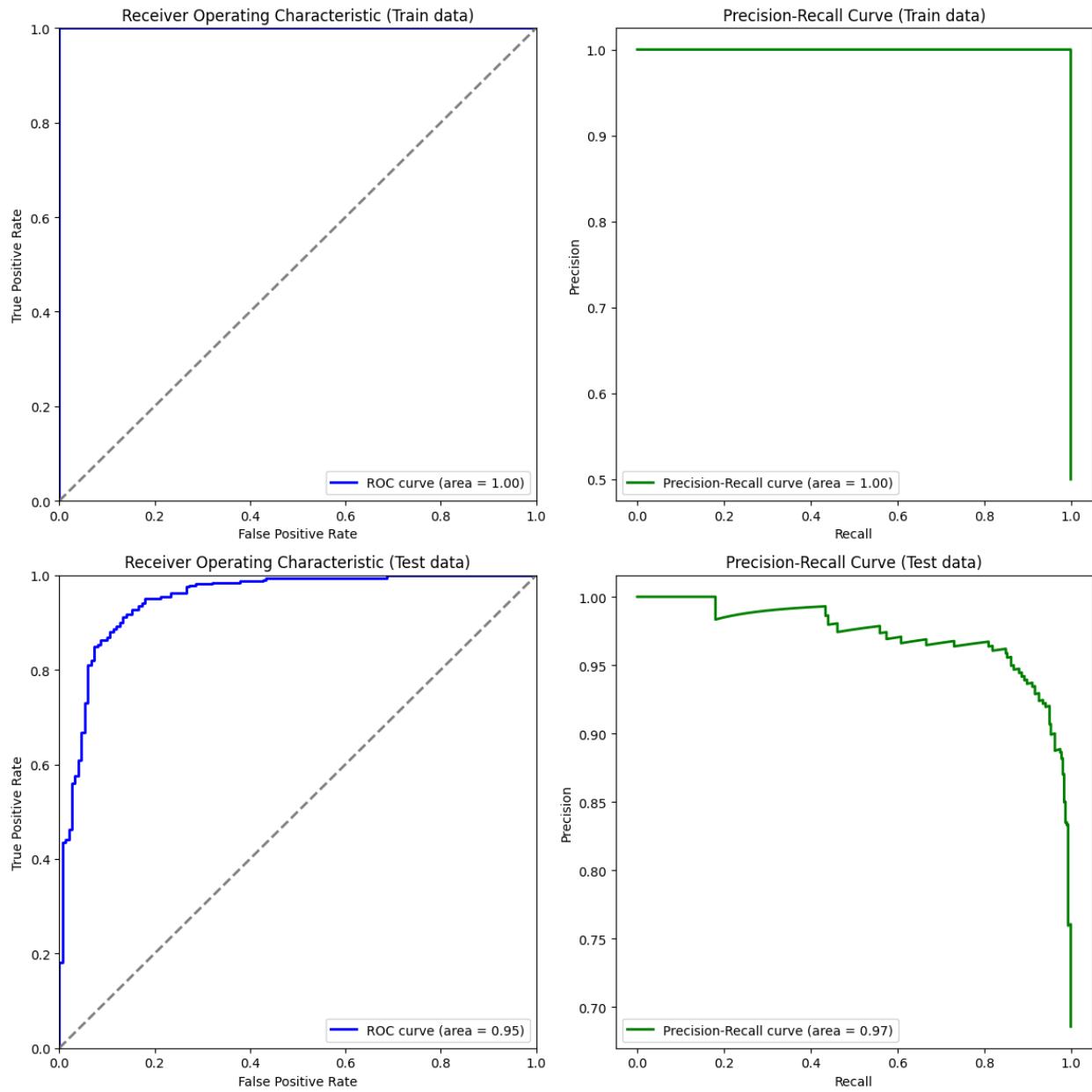
1	1.00	1.00	1.00	1289
accuracy			1.00	2578
macro avg	1.00	1.00	1.00	2578
weighted avg	1.00	1.00	1.00	2578

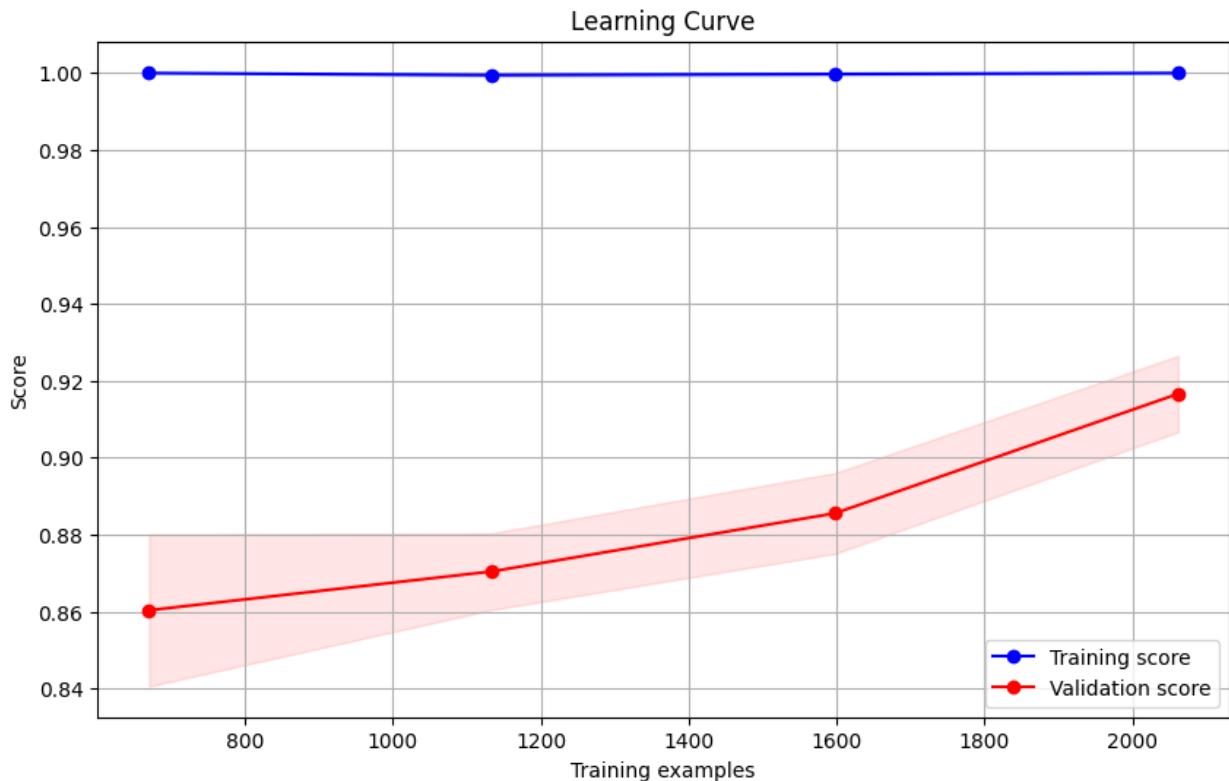
#### Test Classification Report:

	precision	recall	f1-score	support
0	0.83	0.86	0.84	150
1	0.93	0.92	0.93	327
accuracy			0.90	477
macro avg	0.88	0.89	0.88	477
weighted avg	0.90	0.90	0.90	477

```
[LightGBM] [Warning] min_data_in_leaf is set=20, min_child_samples=98
will be ignored. Current value: min_data_in_leaf=20
[LightGBM] [Warning] feature_fraction is set=0.7184404372168336,
colsample_bytree=0.9565345704829102 will be ignored. Current value:
feature_fraction=0.7184404372168336
[LightGBM] [Warning] bagging_fraction is set=0.795633685837714,
subsample=0.917411003148779 will be ignored. Current value:
bagging_fraction=0.795633685837714
[LightGBM] [Warning] bagging_freq is set=5, subsample_freq=0 will be
ignored. Current value: bagging_freq=5
[LightGBM] [Warning] min_data_in_leaf is set=20, min_child_samples=98
will be ignored. Current value: min_data_in_leaf=20
[LightGBM] [Warning] feature_fraction is set=0.7184404372168336,
colsample_bytree=0.9565345704829102 will be ignored. Current value:
feature_fraction=0.7184404372168336
[LightGBM] [Warning] bagging_fraction is set=0.795633685837714,
subsample=0.917411003148779 will be ignored. Current value:
bagging_fraction=0.795633685837714
[LightGBM] [Warning] bagging_freq is set=5, subsample_freq=0 will be
ignored. Current value: bagging_freq=5
```







MLFL0W Logging is completed

## STACKING CLASSIFIER

### Imbalanced Dataset

#### Hyper parameter Tuning

```
# Define the parameter grid for RandomizedSearchCV
start_tune_time = time.time()
param_dist = {
    'stack_method': ['auto', 'predict_proba', 'predict'], # Methods
    'passthrough': [True, False], # Whether to pass the original
    'final_estimator': LogisticRegression(max_iter=10000,
    random_state=42), # The final estimator
    'n_jobs=-1 # Use all available cores
}
```

```

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
    estimator=stacking_clf,
    param_distributions=param_dist,
    n_iter=6, # Number of parameter settings to try
    cv=5, # Number of folds in cross-validation
    verbose=1,
    random_state=42,
    n_jobs=-1
)

# Fit RandomizedSearchCV
random_search.fit(X_train_imb, y_train_imb)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning time:", tuning_time)

Fitting 5 folds for each of 6 candidates, totalling 30 fits
Best parameters found: {'stack_method': 'predict', 'passthrough': True}
Best score: 0.9133305705207901
Tuning time: 3721.9128143787384

```

### Logging Best Stacking Classifier Model into MLFLOW

```

# Model details
name = "Tuned_Stacking_Classifier_on_Imbalanced_Dataset"
bal_type = "Imbalanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time
start_train_time = time.time()
model.fit(X_train_imb, y_train_imb)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_imb_train = model.predict(X_train_imb)
y_pred_imb_test = model.predict(X_test_imb)
end_test_time = time.time()

```

```

# Calculate testing time
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df,
mlflow_logging_and_metric_printing
time_df,feature_importance_df =
log_time_and_feature_importances_df(time_df,feature_importance_df,name
,training_time,testing_time,model,ola_features,bal_type,tuning_time)
mlflow_logging_and_metric_printing(model,name,bal_type,X_train_imb,y_t
rain_imb,X_test_imb,y_test_imb,y_pred_imb_train,y_pred_imb_test,tuning
_score,**params)

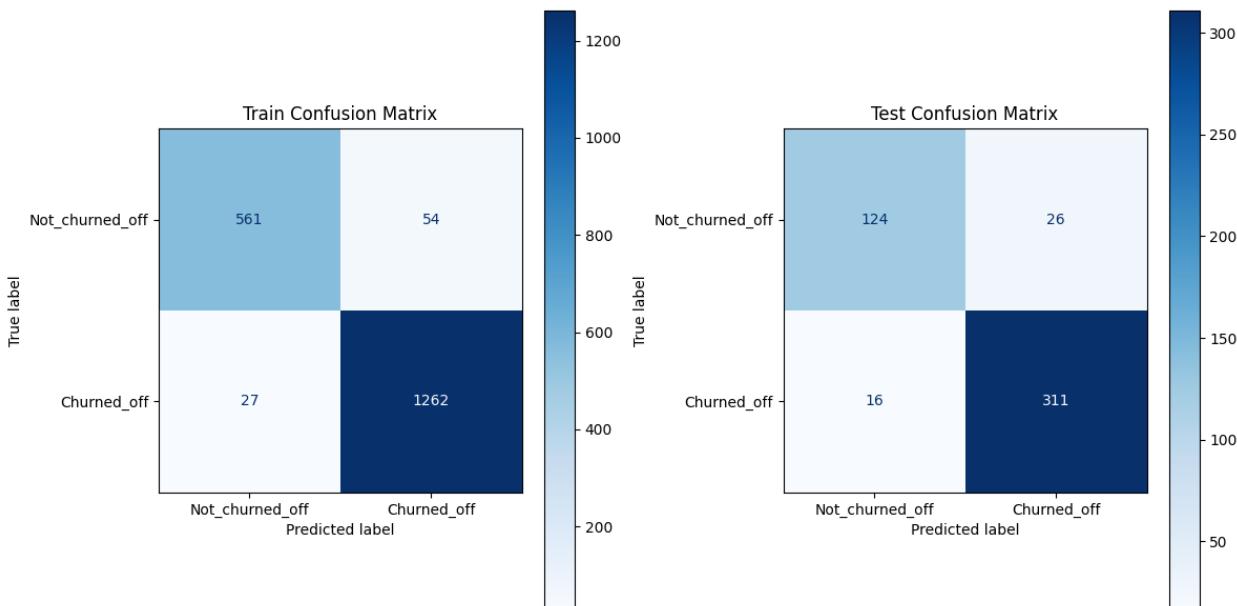
[LightGBM] [Warning] min_data_in_leaf is set=77, min_child_samples=45
will be ignored. Current value: min_data_in_leaf=77
[LightGBM] [Warning] feature_fraction is set=0.6880247346154161,
colsample_bytree=1.1159854502601791 will be ignored. Current value:
feature_fraction=0.6880247346154161
[LightGBM] [Warning] bagging_fraction is set=0.7040356346288988,
subsample=0.6488880503324447 will be ignored. Current value:
bagging_fraction=0.7040356346288988
[LightGBM] [Warning] bagging_freq is set=5, subsample_freq=0 will be
ignored. Current value: bagging_freq=5
[LightGBM] [Warning] min_data_in_leaf is set=77, min_child_samples=45
will be ignored. Current value: min_data_in_leaf=77
[LightGBM] [Warning] feature_fraction is set=0.6880247346154161,
colsample_bytree=1.1159854502601791 will be ignored. Current value:
feature_fraction=0.6880247346154161
[LightGBM] [Warning] bagging_fraction is set=0.7040356346288988,
subsample=0.6488880503324447 will be ignored. Current value:
bagging_fraction=0.7040356346288988
[LightGBM] [Warning] bagging_freq is set=5, subsample_freq=0 will be
ignored. Current value: bagging_freq=5
Model: Tuned_Stacking_Classifier_on_Imbalanced_Dataset
params: {'stack_method': 'predict', 'passthrough': True}
Training Time: 245.9284 seconds
Testing Time: 1.7313 seconds
Tuning Time: 3721.9128 seconds
[LightGBM] [Warning] min_data_in_leaf is set=77, min_child_samples=45
will be ignored. Current value: min_data_in_leaf=77
[LightGBM] [Warning] feature_fraction is set=0.6880247346154161,
colsample_bytree=1.1159854502601791 will be ignored. Current value:
feature_fraction=0.6880247346154161
[LightGBM] [Warning] bagging_fraction is set=0.7040356346288988,

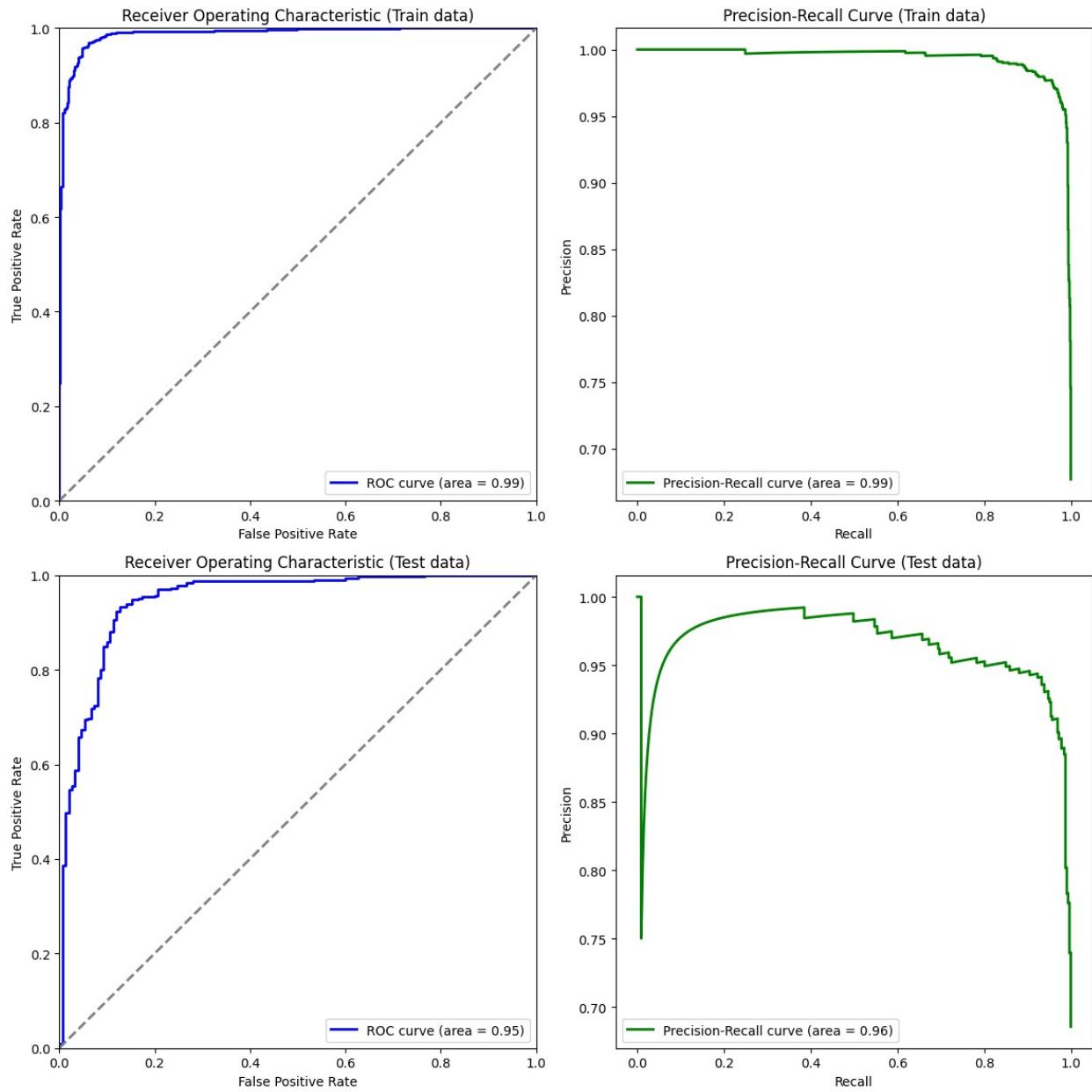
```

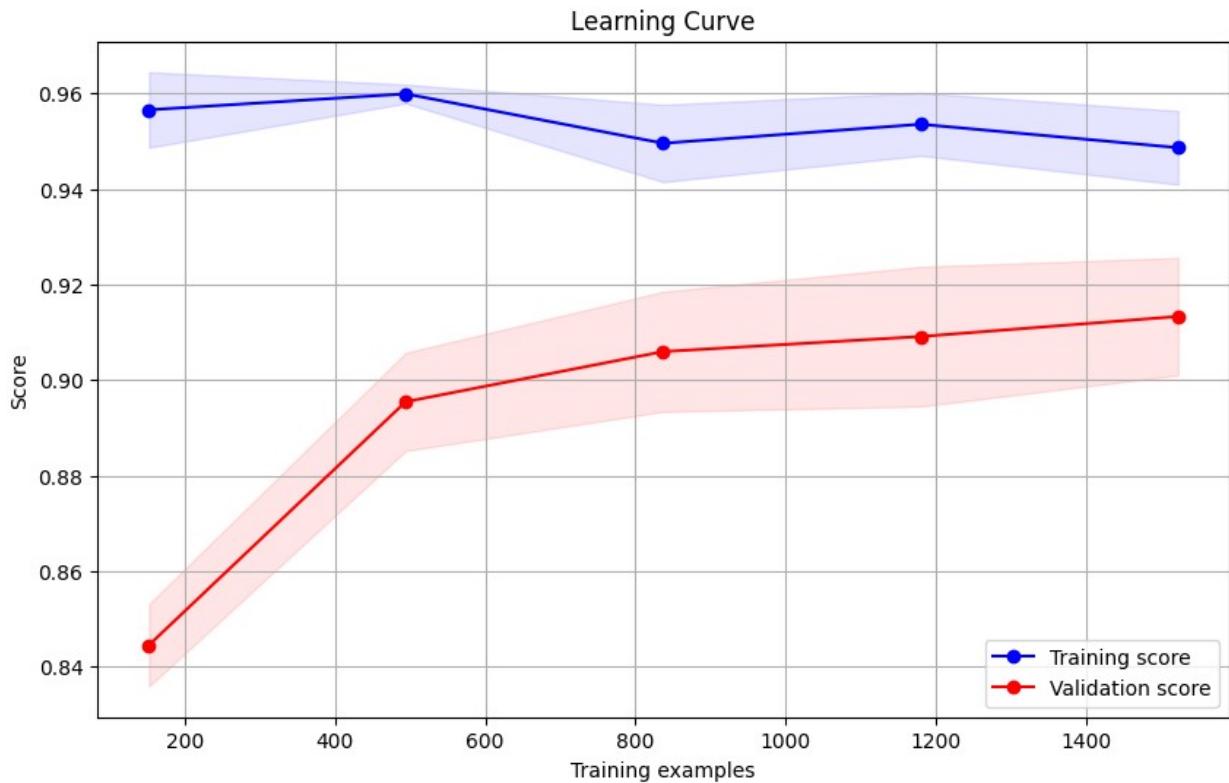
```
subsample=0.6488880503324447 will be ignored. Current value:  
bagging_fraction=0.7040356346288988  
[LightGBM] [Warning] bagging_freq is set=5, subsample_freq=0 will be  
ignored. Current value: bagging_freq=5  
[LightGBM] [Warning] min_data_in_leaf is set=77, min_child_samples=45  
will be ignored. Current value: min_data_in_leaf=77  
[LightGBM] [Warning] feature_fraction is set=0.6880247346154161,  
colsample_bytree=1.1159854502601791 will be ignored. Current value:  
feature_fraction=0.6880247346154161  
[LightGBM] [Warning] bagging_fraction is set=0.7040356346288988,  
subsample=0.6488880503324447 will be ignored. Current value:  
bagging_fraction=0.7040356346288988  
[LightGBM] [Warning] bagging_freq is set=5, subsample_freq=0 will be  
ignored. Current value: bagging_freq=5  
Train Metrics:  
Accuracy_train: 0.9575  
Precision_train: 0.9590  
Recall_train: 0.9791  
F1_score_train: 0.9689  
F2_score_train: 0.9750  
Roc_auc_train: 0.9883  
Pr_auc_train: 0.9939  
  
Test Metrics:  
Accuracy_test: 0.9119  
Precision_test: 0.9228  
Recall_test: 0.9511  
F1_score_test: 0.9367  
F2_score_test: 0.9453  
Roc_auc_test: 0.9480  
Pr_auc_test: 0.9634  
  
Tuning Metrics:  
hyper_parameter_tuning_best_est_score: 0.9133  
  
Train Classification Report:  
precision recall f1-score support  
0 0.95 0.91 0.93 615  
1 0.96 0.98 0.97 1289  
  
accuracy 0.96 0.96 1904  
macro avg 0.96 0.95 0.95 1904  
weighted avg 0.96 0.96 0.96 1904  
  
Test Classification Report:  
precision recall f1-score support  
0 0.89 0.83 0.86 150
```

1	0.92	0.95	0.94	327
accuracy			0.91	477
macro avg	0.90	0.89	0.90	477
weighted avg	0.91	0.91	0.91	477

[LightGBM] [Warning] min\_data\_in\_leaf is set=77, min\_child\_samples=45 will be ignored. Current value: min\_data\_in\_leaf=77  
[LightGBM] [Warning] feature\_fraction is set=0.6880247346154161, colsample\_bytree=1.1159854502601791 will be ignored. Current value: feature\_fraction=0.6880247346154161  
[LightGBM] [Warning] bagging\_fraction is set=0.7040356346288988, subsample=0.6488880503324447 will be ignored. Current value: bagging\_fraction=0.7040356346288988  
[LightGBM] [Warning] bagging\_freq is set=5, subsample\_freq=0 will be ignored. Current value: bagging\_freq=5  
[LightGBM] [Warning] min\_data\_in\_leaf is set=77, min\_child\_samples=45 will be ignored. Current value: min\_data\_in\_leaf=77  
[LightGBM] [Warning] feature\_fraction is set=0.6880247346154161, colsample\_bytree=1.1159854502601791 will be ignored. Current value: feature\_fraction=0.6880247346154161  
[LightGBM] [Warning] bagging\_fraction is set=0.7040356346288988, subsample=0.6488880503324447 will be ignored. Current value: bagging\_fraction=0.7040356346288988  
[LightGBM] [Warning] bagging\_freq is set=5, subsample\_freq=0 will be ignored. Current value: bagging\_freq=5







MLFL0W Logging is completed

## Balanced Dataset

### Hyper parameter Tuning

```
# Define the parameter grid for RandomizedSearchCV
start_tune_time = time.time()
param_dist = {
    'stack_method': ['auto', 'predict_proba', 'predict'], # Methods for stacking
    'passthrough': [True, False], # Whether to pass the original features to the final estimator
}

# Initialize the StackingClassifier
stacking_clf = StackingClassifier(
    estimators=models[::2], # The base models
    final_estimator=LogisticRegression(max_iter=10000,
random_state=42), # The final estimator
    n_jobs=-1 # Use all available cores
)

# Setup RandomizedSearchCV
random_search = RandomizedSearchCV(
```

```

estimator=stacking_clf,
param_distributions=param_dist,
n_iter=6, # Number of parameter settings to try
cv=5, # Number of folds in cross-validation
verbose=1,
random_state=42,
n_jobs=-1
)

# Fit RandomizedSearchCV
random_search.fit(X_train_bal, y_train_bal)

# Best model and hyperparameters
print("Best parameters found:", random_search.best_params_)
print("Best score:", random_search.best_score_)
tuning_score = random_search.best_score_
end_tune_time = time.time()
tuning_time = end_tune_time - start_tune_time
print("Tuning time:", tuning_time)

Fitting 5 folds for each of 6 candidates, totalling 30 fits
Best parameters found: {'stack_method': 'predict', 'passthrough': True}
Best score: 0.9228200496726122
Tuning time: 1385.157784461975

```

## Logging Best Stacking Classifier Model into MLFLOW

```

# Model details
name = "Tuned_Stacking_Classifier_on_Balanced_Dataset"
bal_type = "Balanced"
model = random_search.best_estimator_
params = random_search.best_params_

# Record training time
start_train_time = time.time()
model.fit(X_train_bal, y_train_bal)
end_train_time = time.time()

# Calculate training time
training_time = end_train_time - start_train_time

# Record testing time
start_test_time = time.time()
y_pred_bal_train = model.predict(X_train_bal)
y_pred_bal_test = model.predict(X_test_bal)
end_test_time = time.time()

# Calculate testing time
testing_time = end_test_time - start_test_time

```

```

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")
print(f"Tuning Time: {tuning_time:.4f} seconds")

# logging time_df, feature_importance_df,
mlflow_logging_and_metric_printing
time_df, feature_importance_df =
log_time_and_feature_importances_df(time_df, feature_importance_df, name
, training_time, testing_time, model, ola_features, bal_type, tuning_time)
mlflow_logging_and_metric_printing(model, name, bal_type, X_train_bal, y_train_bal, X_test_bal, y_test_bal, y_pred_bal_train, y_pred_bal_test, tuning_score, **params)

[LightGBM] [Warning] min_data_in_leaf is set=20, min_child_samples=98
will be ignored. Current value: min_data_in_leaf=20
[LightGBM] [Warning] feature_fraction is set=0.7184404372168336,
colsample_bytree=0.9565345704829102 will be ignored. Current value:
feature_fraction=0.7184404372168336
[LightGBM] [Warning] bagging_fraction is set=0.795633685837714,
subsample=0.917411003148779 will be ignored. Current value:
bagging_fraction=0.795633685837714
[LightGBM] [Warning] bagging_freq is set=5, subsample_freq=0 will be
ignored. Current value: bagging_freq=5
[LightGBM] [Warning] min_data_in_leaf is set=20, min_child_samples=98
will be ignored. Current value: min_data_in_leaf=20
[LightGBM] [Warning] feature_fraction is set=0.7184404372168336,
colsample_bytree=0.9565345704829102 will be ignored. Current value:
feature_fraction=0.7184404372168336
[LightGBM] [Warning] bagging_fraction is set=0.795633685837714,
subsample=0.917411003148779 will be ignored. Current value:
bagging_fraction=0.795633685837714
[LightGBM] [Warning] bagging_freq is set=5, subsample_freq=0 will be
ignored. Current value: bagging_freq=5
Model: Tuned_Stacking_Classifier_on_Balanced_Dataset
params: {'stack_method': 'predict', 'passthrough': True}
Training Time: 56.0524 seconds
Testing Time: 1.7386 seconds
Tuning Time: 1385.1578 seconds
[LightGBM] [Warning] min_data_in_leaf is set=20, min_child_samples=98
will be ignored. Current value: min_data_in_leaf=20
[LightGBM] [Warning] feature_fraction is set=0.7184404372168336,
colsample_bytree=0.9565345704829102 will be ignored. Current value:
feature_fraction=0.7184404372168336
[LightGBM] [Warning] bagging_fraction is set=0.795633685837714,
subsample=0.917411003148779 will be ignored. Current value:
bagging_fraction=0.795633685837714

```

```

[LightGBM] [Warning] bagging_freq is set=5, subsample_freq=0 will be
ignored. Current value: bagging_freq=5
[LightGBM] [Warning] min_data_in_leaf is set=20, min_child_samples=98
will be ignored. Current value: min_data_in_leaf=20
[LightGBM] [Warning] feature_fraction is set=0.7184404372168336,
colsample_bytree=0.9565345704829102 will be ignored. Current value:
feature_fraction=0.7184404372168336
[LightGBM] [Warning] bagging_fraction is set=0.795633685837714,
subsample=0.917411003148779 will be ignored. Current value:
bagging_fraction=0.795633685837714
[LightGBM] [Warning] bagging_freq is set=5, subsample_freq=0 will be
ignored. Current value: bagging_freq=5
Train Metrics:
Accuracy_train: 0.9992
Precision_train: 0.9992
Recall_train: 0.9992
F1_score_train: 0.9992
F2_score_train: 0.9992
Roc_auc_train: 1.0000
Pr_auc_train: 1.0000

Test Metrics:
Accuracy_test: 0.8994
Precision_test: 0.9319
Recall_test: 0.9205
F1_score_test: 0.9262
F2_score_test: 0.9227
Roc_auc_test: 0.9389
Pr_auc_test: 0.9639

Tuning Metrics:
hyper_parameter_tuning_best_est_score: 0.9228

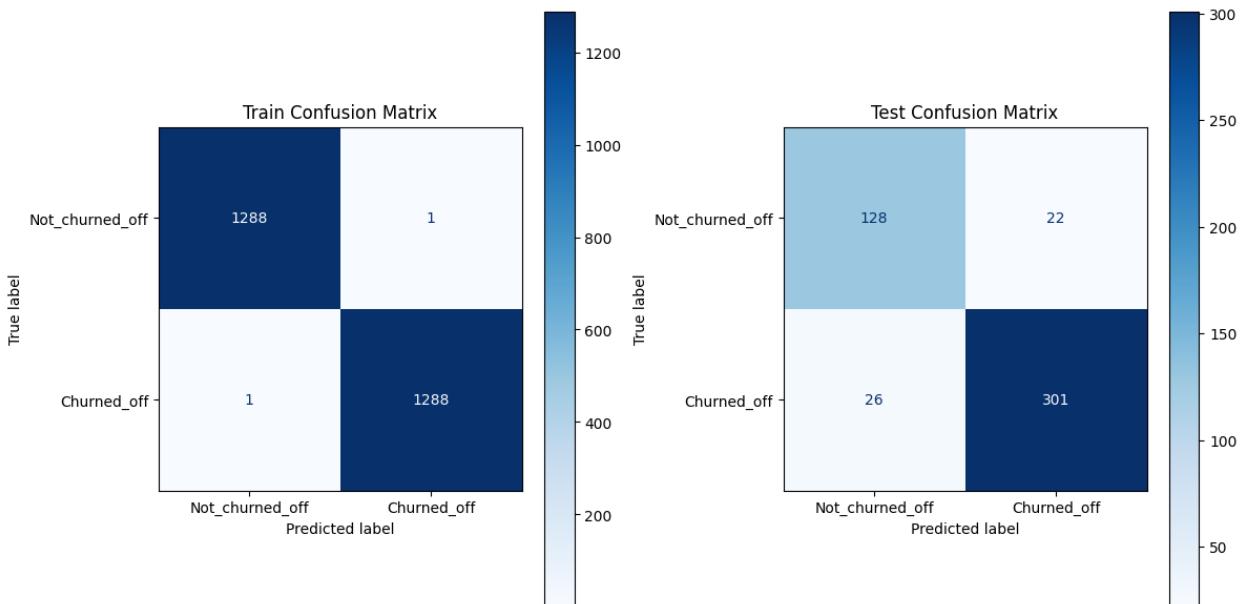
Train Classification Report:
precision    recall   f1-score   support
      0       1.00     1.00     1.00     1289
      1       1.00     1.00     1.00     1289
      accuracy          1.00     1.00     1.00     2578
      macro avg       1.00     1.00     1.00     2578
      weighted avg    1.00     1.00     1.00     2578

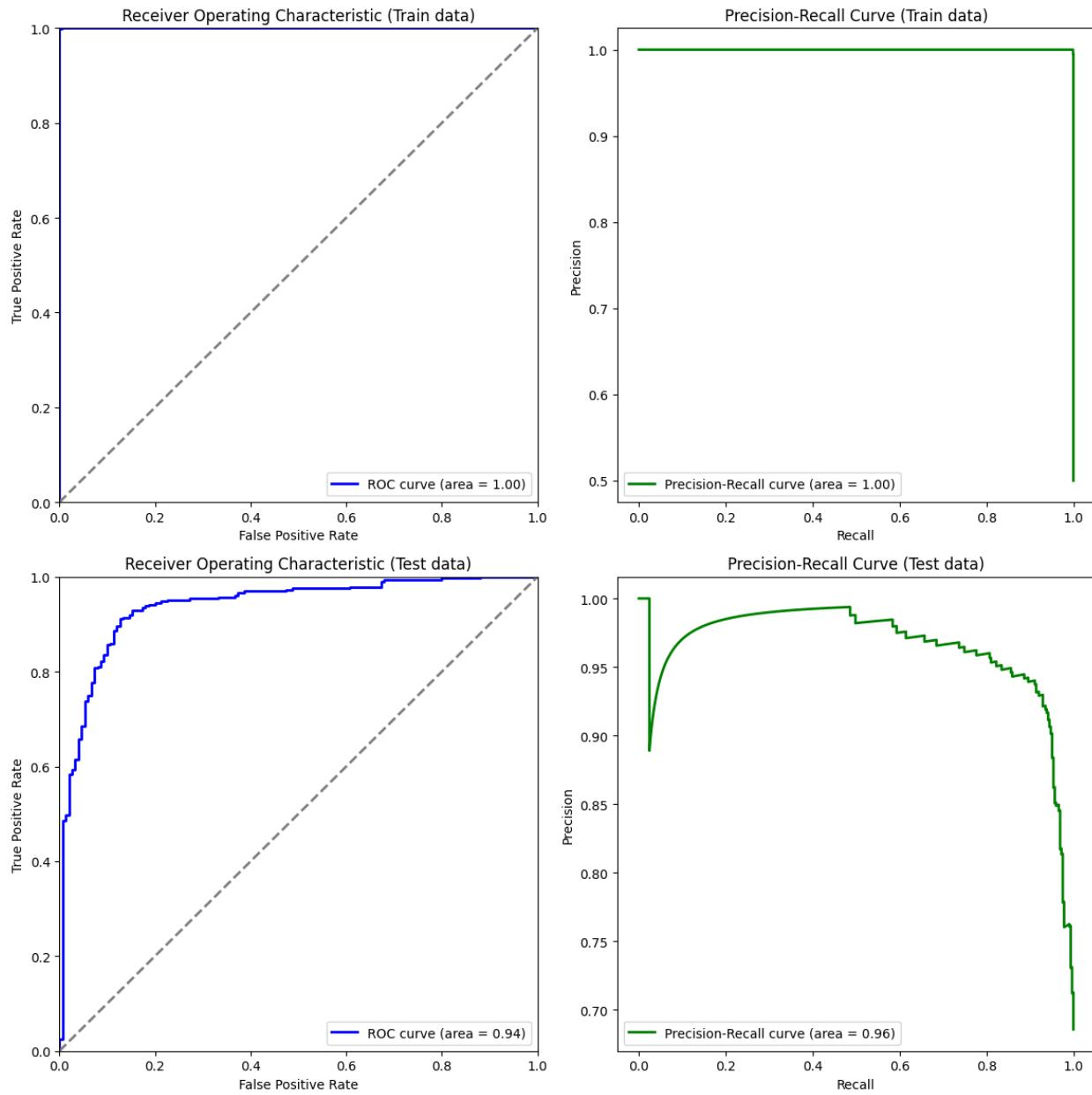
Test Classification Report:
precision    recall   f1-score   support
      0       0.83     0.85     0.84      150
      1       0.93     0.92     0.93      327

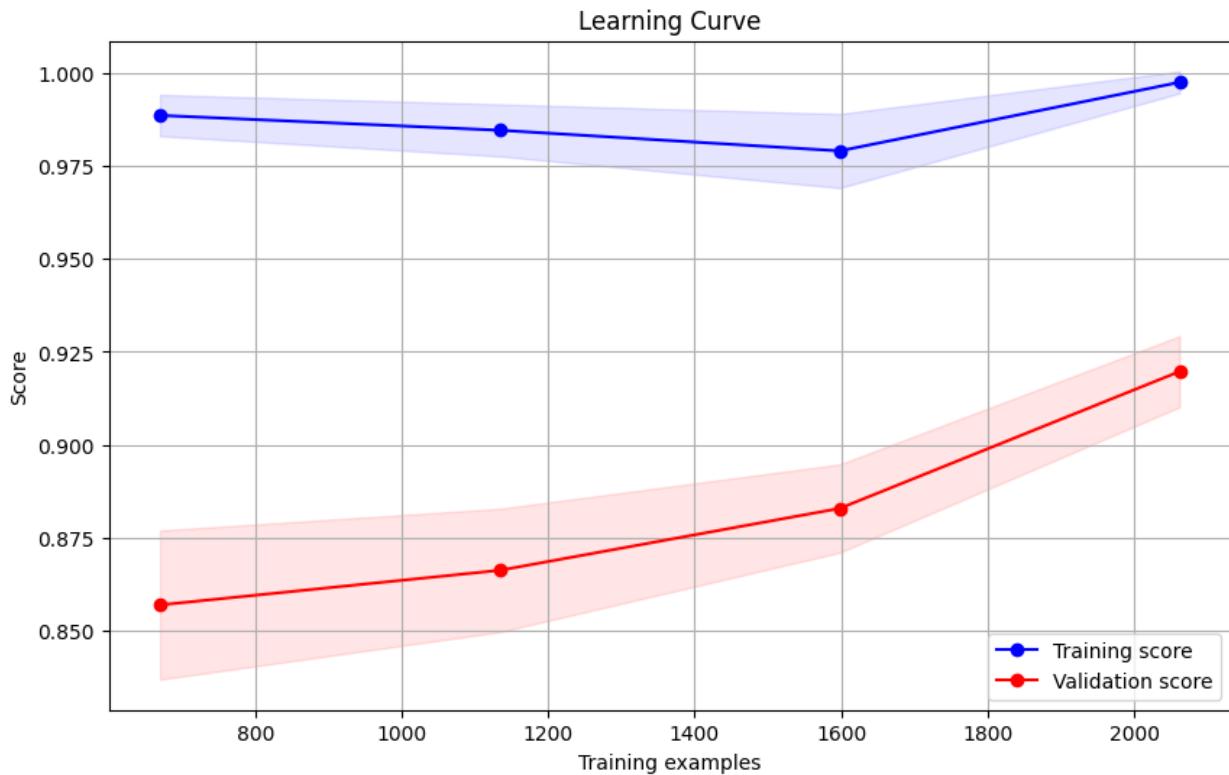
```

	accuracy		0.90	477
macro avg	0.88	0.89	0.88	477
weighted avg	0.90	0.90	0.90	477

[LightGBM] [Warning] min\_data\_in\_leaf is set=20, min\_child\_samples=98 will be ignored. Current value: min\_data\_in\_leaf=20  
[LightGBM] [Warning] feature\_fraction is set=0.7184404372168336, colsample\_bytree=0.9565345704829102 will be ignored. Current value: feature\_fraction=0.7184404372168336  
[LightGBM] [Warning] bagging\_fraction is set=0.795633685837714, subsample=0.917411003148779 will be ignored. Current value: bagging\_fraction=0.795633685837714  
[LightGBM] [Warning] bagging\_freq is set=5, subsample\_freq=0 will be ignored. Current value: bagging\_freq=5  
[LightGBM] [Warning] min\_data\_in\_leaf is set=20, min\_child\_samples=98 will be ignored. Current value: min\_data\_in\_leaf=20  
[LightGBM] [Warning] feature\_fraction is set=0.7184404372168336, colsample\_bytree=0.9565345704829102 will be ignored. Current value: feature\_fraction=0.7184404372168336  
[LightGBM] [Warning] bagging\_fraction is set=0.795633685837714, subsample=0.917411003148779 will be ignored. Current value: bagging\_fraction=0.795633685837714  
[LightGBM] [Warning] bagging\_freq is set=5, subsample\_freq=0 will be ignored. Current value: bagging\_freq=5







MLFLOW Logging is completed

## CASCADING CLASSIFIER MODEL

Imbalanced Dataset (Code from scratch)

```
def load_model_from_run(run_id, run_name):
    # Construct the model URI
    model_uri = f"runs:/{{run_id}}/{{run_name}}_model"

    try:
        # Load the model
        model = mlflow.sklearn.load_model(model_uri)
        print(f"Loaded model from run {run_id} with run name
{run_name}")
        return model
    except Exception as e:
        print(f"Failed to load model from run {run_id}: {e}")
        return None

# Example usage to load models
experiment = mlflow.get_experiment_by_name("Ola_Ensemble_Learning")
experiment_id = experiment.experiment_id

# Search for runs
```

```

runs = mlflow.search_runs(experiment_ids=[experiment_id])
run_ids = runs['run_id'].tolist()

# Load models
models = []
for run_id in run_ids:
    run = mlflow.get_run(run_id)
    run_name = run.info.run_name
    model = load_model_from_run(run_id, run_name)
    if model:
        models.append((f"model_{run_id}", model))
models = [i[1] for i in models]
print(models)

```

Loaded model from run 081f39eda2ee48a0a5b63617f22e9ab3 with run name Tuned\_Stacking\_Classifier\_on\_Balanced\_Dataset

Loaded model from run 54f7b900538248c396eb076c95a1d717 with run name Tuned\_Stacking\_Classifier\_on\_Imbalanced\_Dataset

Loaded model from run ebda8eeaaff44d5abb19a8d48bf35300 with run name Tuned\_Voting\_Classifier\_on\_Balanced\_Dataset

Failed to load model from run 08a2273f534a4a5db7f59e667dc8698c: No such file or directory: 'C:\Users\saina\Desktop\DS\_ML\_AI\Scaler\Module\_14\_ML\_Supervised\_Algorithms\Case\_study\Ola\_Ensemble\_learning\_Case\_study\main\mlruns\165809646674472470\08a2273f534a4a5db7f59e667dc8698c\artifacts\Tuned\_Voting\_Classifier\_on\_Imbalanced\_Dataset\_model'

Loaded model from run eba5fab22ac34697bed53bb627af2c33 with run name Tuned\_Light\_GBM\_on\_Balanced\_Dataset

Loaded model from run 0c81cb30c10d4f56a99f55b2e9fb2d4c with run name Tuned\_Light\_GBM\_on\_Imbalanced\_Dataset

Loaded model from run 32171425fbf342ee9f549d502105a681 with run name Tuned\_XG\_boost\_on\_Balanced\_Dataset

Loaded model from run 2adb1674b77d4b9ba15c0c8978e82d50 with run name Tuned\_XG\_boost\_on\_Imbalanced\_Dataset

Loaded model from run 2576e8de75f841a5bbfc9aa9108e406d with run name Tuned\_Gradient\_boost\_on\_Balanced\_Dataset

Loaded model from run aef92e3ec93c485bb2457c0a4bd209d8 with run name Tuned\_Gradient\_boost\_on\_Imbalanced\_Dataset

Loaded model from run 6dce7745cea54262abfdd53e1081031c with run name Tuned\_Adaboost\_on\_Balanced\_Dataset

Loaded model from run 0d740d00e0094d11b582f53aed58a72b with run name Tuned\_Adaboost\_on\_Imbalanced\_Dataset

Loaded model from run 13b64ce985364b61ba859b705db9bac5 with run name Tuned\_Bagging\_RF\_on\_Balanced\_Dataset

Loaded model from run 907c9c90e757474bb2335a8d6155d157 with run name Tuned\_Bagging\_RF\_on\_Imbalanced\_Dataset

Loaded model from run 396f38da1e9d416bbd82023ff75cf227 with run name Tuned\_Random\_Forest\_on\_Balanced\_Dataset

Loaded model from run 60050ff4ee41446da38a6b38fa371520 with run name Tuned\_Random\_Forest\_on\_Imbalanced\_Dataset

Loaded model from run b64120f1fb98427ca7cce36345695bbf with run name Tuned\_Decision\_Tree\_on\_Balanced\_Dataset

Loaded model from run 43f861f9ac2041e0a2fd0663667b8129 with run name Tuned\_Decision\_Tree\_on\_Imbalanced\_Dataset

Loaded model from run 477ac0ff3c144d70af08a5747b5e8e96 with run name Tuned\_SVC\_on\_Balanced\_Dataset

Loaded model from run 112add6166414f00887dcbea7fc6d86d with run name Tuned\_SVC\_on\_Imbalanced\_Dataset

Loaded model from run 2897df0fb90d464c9c9195c03064411e with run name Tuned\_KNN\_on\_Balanced\_Dataset

Loaded model from run 1c152ab6db5a4f2fbc1a457823c9f65f with run name Tuned\_KNN\_on\_Imbalanced\_Dataset

Loaded model from run 4caa37eaa4964998b5ff0b94dae599b1 with run name Tuned\_MLPClassifier\_on\_Balanced\_Dataset

Loaded model from run 1d7993e0c3124955a9b383b40b15ea48 with run name Tuned\_MLPClassifier\_on\_Imbalanced\_Dataset

Loaded model from run 62fc0473d24b4f71babf471575f220cf with run name Tuned\_Logistic\_Regression\_on\_Balanced\_Dataset

Loaded model from run a2514820d45d41e5883b697be7f14629 with run name Tuned\_Logistic\_Regression\_on\_Imbalanced\_Dataset

Loaded model from run efea588e45d8474f83c3b0771dfa5709 with run name Simple\_Logistic\_Regresssion\_on\_balanced\_Dataset

Loaded model from run e624cb2d315c46f5a563c2e82653f175 with run name Simple\_Logistic\_Regression\_on\_Imbalanced\_Dataset

```
[StackingClassifier(estimators=[('model_eba5fab22ac34697bed53bb627af2c33', LGBMClassifier(bagging_fraction=0.795633685837714, bagging_freq=5, boosting_type='dart', colsample_bytree=0.9565345704829102, feature_fraction=0.7184404372168336, learning_rate=0.13495298436110986, max_depth=13, min_child_samples=98, min_child_weight=0.03280034749718639, min_data_in_le... ('model_62fc0473d24b4f71babf471575f220cf', LogisticRegression(C=1872.706848835624, class_weight='balanced', max_iter=10000, penalty='l1', random_state=42, solver='saga')), ('model_efea588e45d8474f83c3b0771dfa5709', LogisticRegression(n_jobs=-1, random_state=42))], final_estimator=LogisticRegression(max_iter=10000, random_state=42), n_jobs=-1, passthrough=True, stack_method='predict'), StackingClassifier(estimators=[('model_0c81cb30c10d4f56a99f55b2e9fb2d4c', LGBMClassifier(bagging_fraction=0.7040356346288988, bagging_freq=5,
```

```
boosting_type='dart', colsample_bytree=1.1159854502601791,
feature_fraction=0.6880247346154161, learning_rate=0.11661537060572182,
max_depth=11, min_child_samples=45, min_child_weight=0.0015229613542912736,
min_data_in... ('model_a2514820d45d41e5883b697be7f14629',
LogisticRegression(C=1872.706848835624, class_weight='balanced',
max_iter=10000, penalty='l1', random_state=42, solver='saga')),
('model_e624cb2d315c46f5a563c2e82653f175', LogisticRegression(n_jobs=-1,
random_state=42))], final_estimator=LogisticRegression(max_iter=10000,
random_state=42), n_jobs=-1, passthrough=True, stack_method='predict'),
VotingClassifier(estimators=[('model_eba5fab22ac34697bed53bb627af2c33',
LGBMClassifier(bagging_fraction=0.795633685837714, bagging_freq=5,
boosting_type='dart', colsample_bytree=0.9565345704829102,
feature_fraction=0.7184404372168336, learning_rate=0.13495298436110986,
max_depth=13, min_child_samples=98, min_child_weight=0.03280034749718639,
min_data_in_leaf... class_weight='balanced', max_iter=10000, penalty='l1',
random_state=42, solver='saga')), ('model_efea588e45d8474f83c3b0771dfa5709',
LogisticRegression(n_jobs=-1, random_state=42))], n_jobs=-1, voting='soft',
weights=[1.0, 0.5, 0.3333333333333333, 0.25, 0.2, 0.1666666666666666,
0.14285714285714285, 0.125, 0.1111111111111111, 0.1, 0.09090909090909091,
0.0833333333333333]), LGBMClassifier(bagging_fraction=0.795633685837714,
bagging_freq=5, boosting_type='dart', colsample_bytree=0.9565345704829102,
feature_fraction=0.7184404372168336, learning_rate=0.13495298436110986,
max_depth=13, min_child_samples=98, min_child_weight=0.03280034749718639,
min_data_in_leaf=20, min_split_gain=0.02279351625419417, n_estimators=732,
num_leaves=135, objective='binary', random_state=42,
reg_alpha=0.4303652916281717, reg_lambda=0.510428195796786,
scale_pos_weight=1.5214946051551315, subsample=0.917411003148779),
LGBMClassifier(bagging_fraction=0.7040356346288988, bagging_freq=5,
boosting_type='dart', colsample_bytree=1.1159854502601791,
feature_fraction=0.6880247346154161, learning_rate=0.11661537060572182,
max_depth=11, min_child_samples=45, min_child_weight=0.0015229613542912736,
min_data_in_leaf=77, min_split_gain=0.003531135494023907, n_estimators=308,
num_leaves=33, objective='binary', random_state=42,
reg_alpha=0.4790367400596901, reg_lambda=1.0519111269531267,
scale_pos_weight=1.1538632326761582, subsample=0.6488880503324447),
XGBClassifier(base_score=None, booster='dart', callbacks=None,
colsample_bylevel=0.8350373297259278,
colsample_bynode=0.7680894083096093, colsample_bytree=0.9668172748386532,
device=None, early_stopping_rounds=None, enable_categorical=False,
eval_metric='logloss', feature_types=None, gamma=0.22530982836487895,
grow_policy='depthwise', importance_type=None, interaction_constraints=None,
learning_rate=0.055050700245112494, max_bin=None, max_cat_threshold=None,
max_cat_to_onehot=None, max_delta_step=None, max_depth=14,
max_leaves=None, min_child_weight=1, missing=nan, monotone_constraints=None,
multi_strategy=None, n_estimators=186, n_jobs=None, num_parallel_tree=None,
random_state=42, ...), XGBClassifier(base_score=None, booster='dart',
callbacks=None, colsample_bylevel=0.9526356769407125,
colsample_bynode=0.9514986114133412, colsample_bytree=0.840607947938491,
device=None, early_stopping_rounds=None, enable_categorical=False,
```

```
eval_metric='logloss', feature_types=None, gamma=0.2074097511688326,
grow_policy='depthwise', importance_type=None, interaction_constraints=None,
learning_rate=0.014101589448099186, max_bin=None, max_cat_threshold=None,
max_cat_to_onehot=None, max_delta_step=None, max_depth=9, max_leaves=None,
min_child_weight=6, missing=nan, monotone_constraints=None,
multi_strategy=None, n_estimators=575, n_jobs=None, num_parallel_tree=None,
random_state=42, ...),
GradientBoostingClassifier(learning_rate=0.07858754845747705, max_depth=9,
max_features='log2', min_impurity_decrease=0.0001120111480109487,
min_samples_leaf=2, min_samples_split=15, n_estimators=688, random_state=42,
subsample=0.8522403201282508),
GradientBoostingClassifier(learning_rate=0.06676646193550176, max_depth=7,
max_features='log2', min_impurity_decrease=0.09613152881849074,
min_samples_leaf=19, min_samples_split=11, n_estimators=493, random_state=42,
subsample=0.94598407522243),
AdaBoostClassifier(estimator=DecisionTreeClassifier(max_depth=5),
learning_rate=0.9000053418175663, n_estimators=520, random_state=42),
AdaBoostClassifier(algorithm='SAMME',
estimator=DecisionTreeClassifier(max_depth=5),
learning_rate=0.1370605126518848, n_estimators=313, random_state=42),
BaggingClassifier(bootstrap=False,
estimator=RandomForestClassifier(class_weight='balanced', criterion='log_loss',
max_depth=20, max_leaf_nodes=20, n_estimators=5, oob_score=True,
random_state=42), max_features=0.9631385219890038,
max_samples=0.9803599686384168, n_estimators=154, random_state=42),
BaggingClassifier(bootstrap=False,
estimator=RandomForestClassifier(class_weight='balanced', criterion='log_loss',
max_depth=20, max_leaf_nodes=20, n_estimators=5, oob_score=True,
random_state=42), max_features=0.9631385219890038,
max_samples=0.9803599686384168, n_estimators=154, n_jobs=-1,
random_state=42), RandomForestClassifier(ccp_alpha=0.028792961647572612,
class_weight='balanced', criterion='log_loss', max_depth=13, max_leaf_nodes=14,
min_impurity_decrease=0.013911619414306187, min_samples_leaf=18,
min_weight_fraction_leaf=0.009055091910420254, n_estimators=877,
oob_score=True, random_state=42),
RandomForestClassifier(ccp_alpha=0.028792961647572612,
class_weight='balanced', criterion='log_loss', max_depth=13, max_leaf_nodes=14,
min_impurity_decrease=0.013911619414306187, min_samples_leaf=18,
min_weight_fraction_leaf=0.009055091910420254, n_estimators=877,
oob_score=True, random_state=42),
DecisionTreeClassifier(ccp_alpha=0.011483682473920355, criterion='entropy',
max_depth=15, max_features='log2', max_leaf_nodes=39,
min_impurity_decrease=0.05812382214226123, min_samples_leaf=11,
min_samples_split=5, min_weight_fraction_leaf=0.16032109607975714,
random_state=42), DecisionTreeClassifier(ccp_alpha=0.01504168911035282,
max_depth=18, max_leaf_nodes=34,
min_impurity_decrease=0.046869315979497034, min_samples_leaf=3,
min_weight_fraction_leaf=0.0068359824134986424, random_state=42),
SVC(C=81.5862458136845, class_weight='balanced', degree=2, gamma='auto',
```

```

max_iter=1677, probability=True, random_state=42), SVC(C=6.436932213118794,
class_weight='balanced', degree=4, gamma='auto', max_iter=1130, probability=True,
random_state=42), KNeighborsClassifier(algorithm='ball_tree', leaf_size=49,
n_neighbors=16, p=1, weights='distance'), KNeighborsClassifier(algorithm='kd_tree',
leaf_size=28, metric='manhattan', n_neighbors=11, p=3, weights='distance'),
MLPClassifier(activation='logistic', alpha=3.893929205071642,
hidden_layer_sizes=(50,), learning_rate_init=0.030524224295953774, max_iter=221,
random_state=42), MLPClassifier(activation='logistic', alpha=3.893929205071642,
hidden_layer_sizes=(50,), learning_rate_init=0.030524224295953774, max_iter=221,
random_state=42), LogisticRegression(C=1872.706848835624,
class_weight='balanced', max_iter=10000, penalty='l1', random_state=42,
solver='saga'), LogisticRegression(C=1872.706848835624, class_weight='balanced',
max_iter=10000, penalty='l1', random_state=42, solver='saga'),
LogisticRegression(n_jobs=-1, random_state=42), LogisticRegression(n_jobs=-1,
random_state=42)]

```

```

# Model details
name = "Two_Level_Cascading_Classifier_on_Imbalanced_Dataset"
bal_type = "Imbalanced"
params = {}
tuning_score = 0

# Models are selected from MLFLOW and Refined to avoid any errors
cascade_models = models[6::2]

# Train first-level models and collect predictions
def train_first_level_models(models, X_train, y_train):
    predictions_train = []
    for model in models:
        model.fit(X_train, y_train)
        preds = model.predict(X_train).reshape(-1, 1)
        probas = model.predict_proba(X_train)
        predictions_train.append(np.hstack((preds, probas)))
    return predictions_train

def transform_features(preds_list, X):
    transformed_features = []
    for model, preds in zip(cascade_models, preds_list):
        preds_test = model.predict(X).reshape(-1, 1)
        probas_test = model.predict_proba(X)
        transformed_features.append(np.hstack((preds_test, probas_test)))
    return np.hstack(transformed_features)

# Collect predictions from the first-level models
start_train_time = time.time()
first_level_train_preds = train_first_level_models(cascade_models,
X_train_imb, y_train_imb)
X_train_level2 = np.hstack(first_level_train_preds)

```

```

# Define and train the second-level model
second_level_model = LGBMClassifier(random_state=42)
second_level_model.fit(X_train_level2, y_train_imb)
end_train_time = time.time()

# training time
training_time = end_train_time - start_train_time

# Prepare test data for the second level
start_test_time = time.time()
first_level_train_preds = [model.predict(X_train_imb).reshape(-1, 1) for model in cascade_models]
first_level_test_preds = [model.predict(X_test_imb).reshape(-1, 1) for model in cascade_models]

first_level_train_probas = [model.predict_proba(X_train_imb) for model in cascade_models]
first_level_test_probas = [model.predict_proba(X_test_imb) for model in cascade_models]

X_train_level2 = np.hstack(first_level_train_preds + first_level_train_probas)
X_test_level2 = np.hstack(first_level_test_preds + first_level_test_probas)

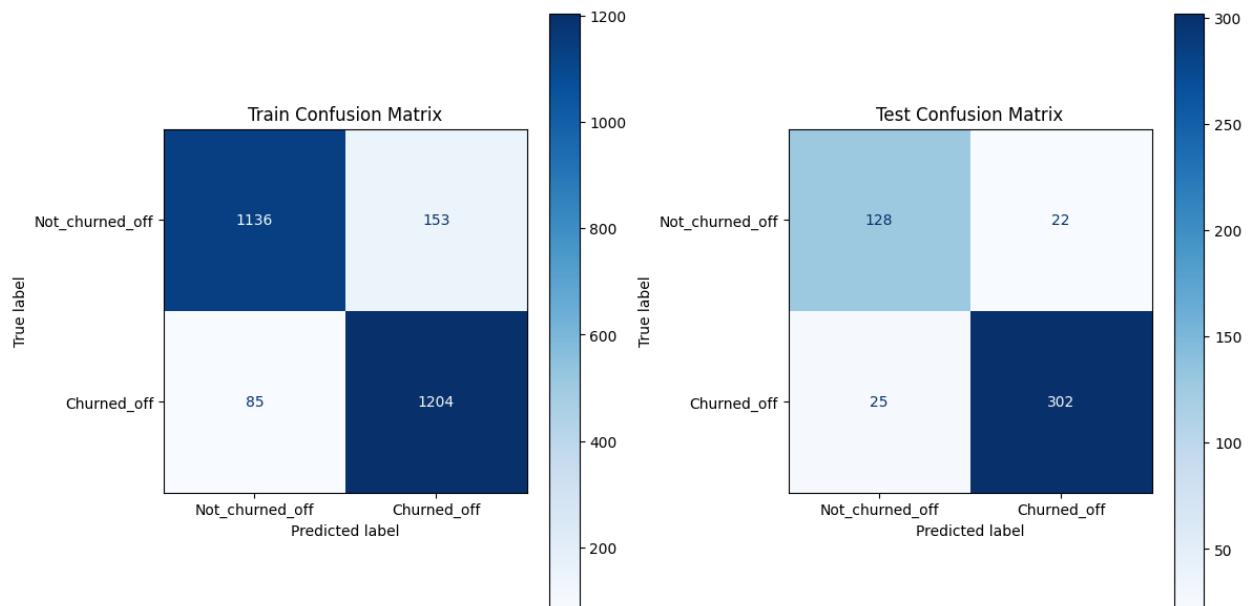
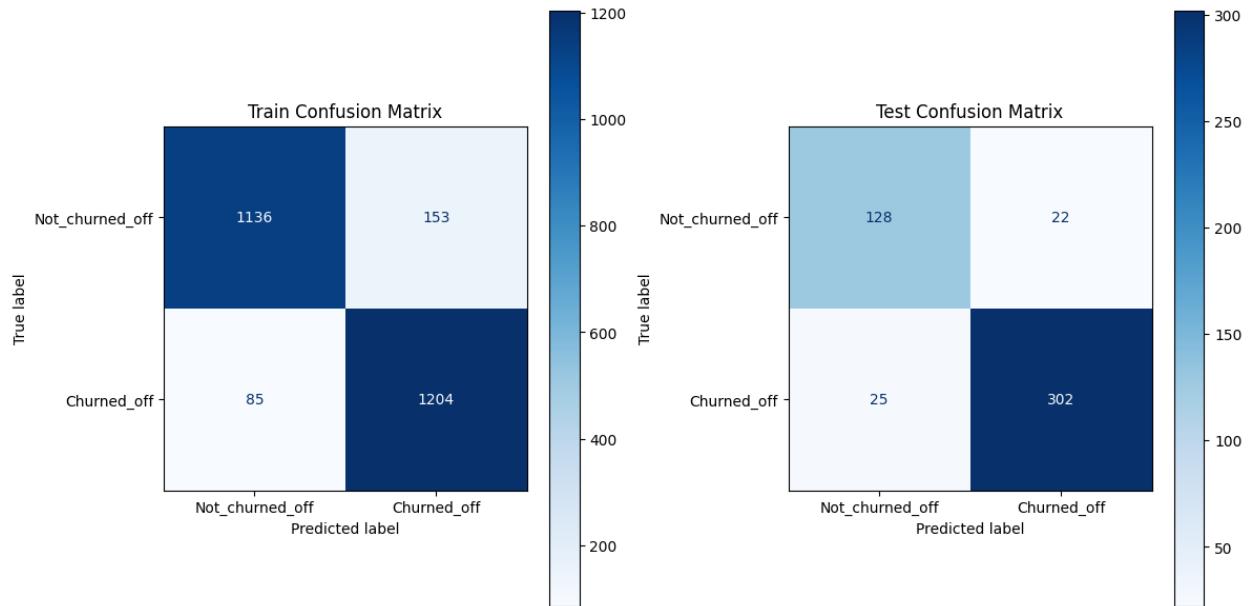
# Make final predictions
y_pred_imb_train = second_level_model.predict(X_train_level2)
y_pred_imb_test = second_level_model.predict(X_test_level2)
end_test_time = time.time()

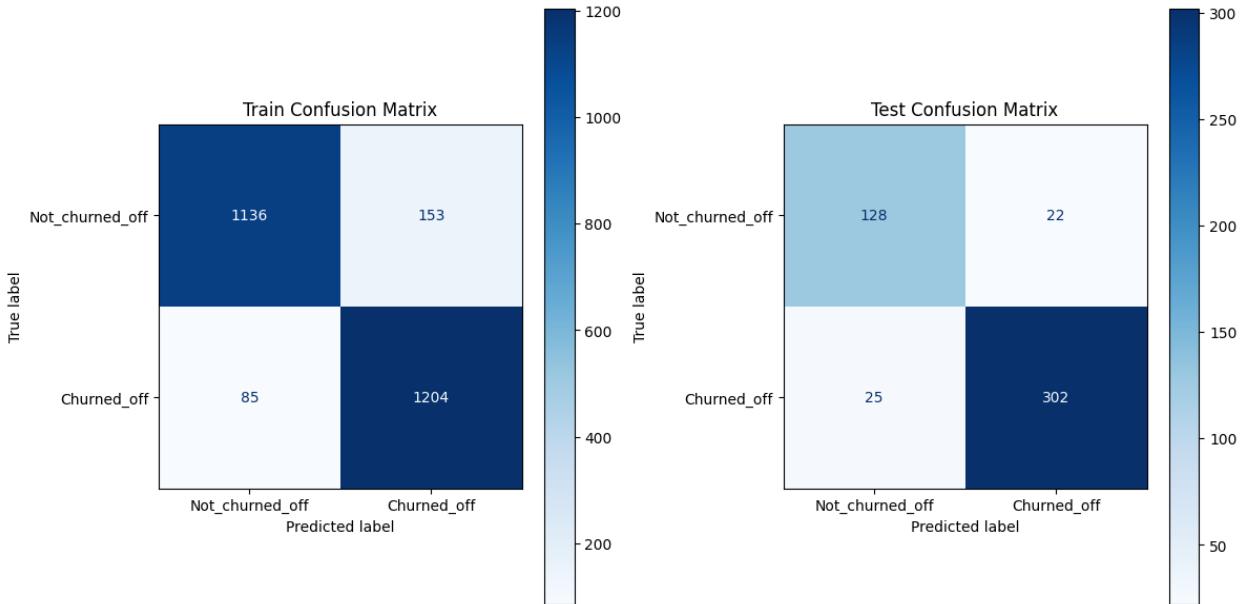
# testing time
testing_time = end_test_time - start_test_time

# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")

# logging time_df, feature_importance_df,
mlflow_logging_and_metric_printing
time_df,feature_importance_df =
log_time_and_feature_importances_df(time_df,feature_importance_df,name ,training_time,testing_time,cascade_models,ola_features,bal_type)
mlflow_logging_and_metric_printing(cascade_models,name,bal_type,X_trai n_imb,y_train_imb,X_test_imb,y_test_imb,y_pred_imb_train,y_pred_imb_te st,tuning_score,**params)

```





## Balanced Dataset

```

# Model details
name = "Two_Level_Cascading_Classifier_on_Balanced_Dataset"
bal_type = "Balanced"
params = {}
tuning_score = 0

# Models are selected from MLFLOW and Refined to avoid any errors
cascade_models = models[5::2]

# Train first-level models and collect predictions
def train_first_level_models(models, X_train, y_train):
    predictions_train = []
    for model in models:
        model.fit(X_train, y_train)
        preds = model.predict(X_train).reshape(-1, 1)
        probas = model.predict_proba(X_train)
        predictions_train.append(np.hstack((preds, probas)))
    return predictions_train

def transform_features(preds_list, X):
    transformed_features = []
    for model, preds in zip(cascade_models, preds_list):
        preds_test = model.predict(X).reshape(-1, 1)
        probas_test = model.predict_proba(X)
        transformed_features.append(np.hstack((preds_test, probas_test)))
    return np.hstack(transformed_features)

# Collect predictions from the first-level models

```

```

start_train_time = time.time()
first_level_train_preds = train_first_level_models(cascade_models,
X_train_bal, y_train_bal)
X_train_level2 = np.hstack(first_level_train_preds)

# Define and train the second-level model
second_level_model = LGBMClassifier(random_state=42)
second_level_model.fit(X_train_level2, y_train_bal)
end_train_time = time.time()

# training time
training_time = end_train_time - start_train_time

# Prepare test data for the second level
start_test_time = time.time()
first_level_train_preds = [model.predict(X_train_bal).reshape(-1, 1)
for model in cascade_models]
first_level_test_preds = [model.predict(X_test_bal).reshape(-1, 1) for
model in cascade_models]

first_level_train_probas = [model.predict_proba(X_train_bal) for model
in cascade_models]
first_level_test_probas = [model.predict_proba(X_test_bal) for model
in cascade_models]

X_train_level2 = np.hstack(first_level_train_preds +
first_level_train_probas)
X_test_level2 = np.hstack(first_level_test_preds +
first_level_test_probas)

# Make final predictions
y_pred_bal_train = second_level_model.predict(X_train_level2)
y_pred_bal_test = second_level_model.predict(X_test_level2)
end_test_time = time.time()

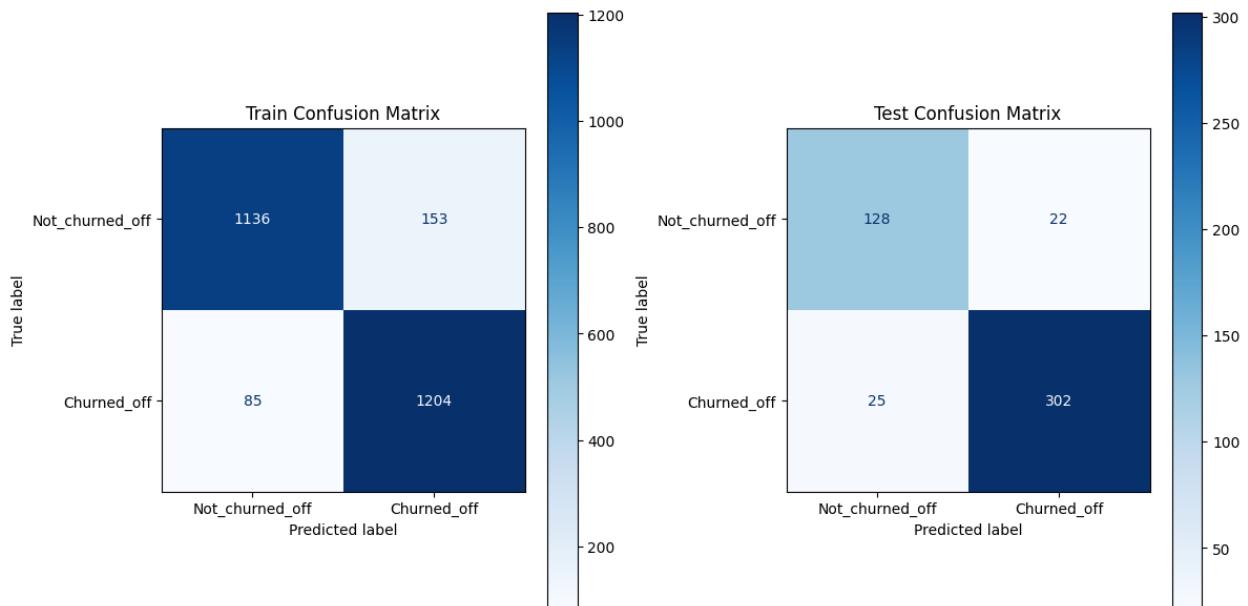
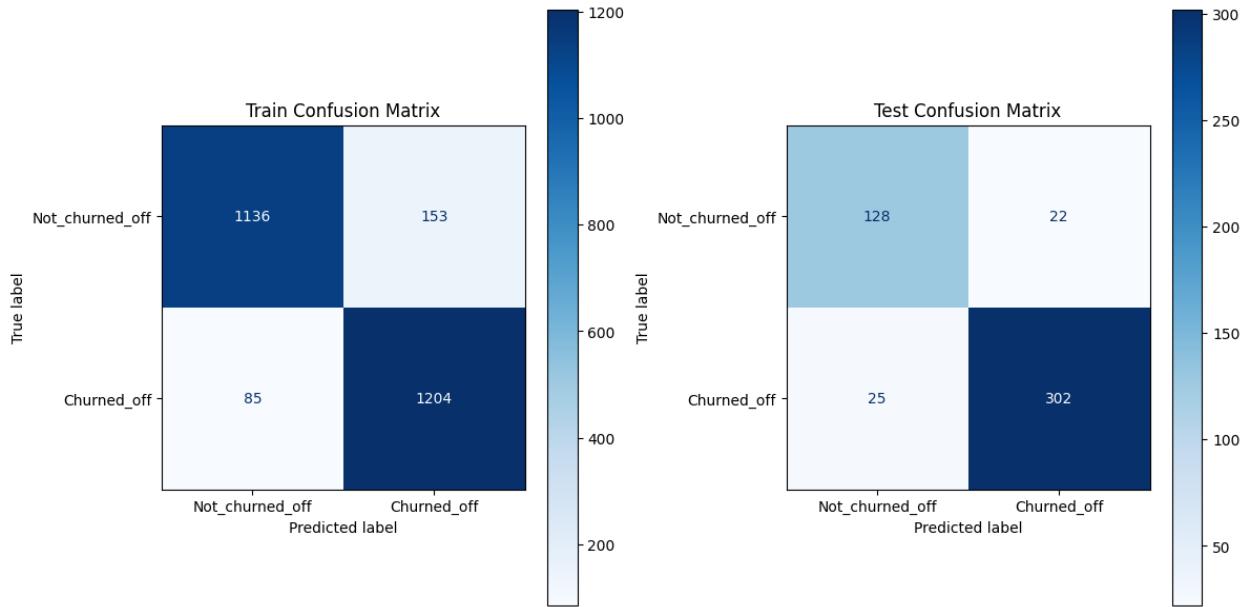
# testing time
testing_time = end_test_time - start_test_time

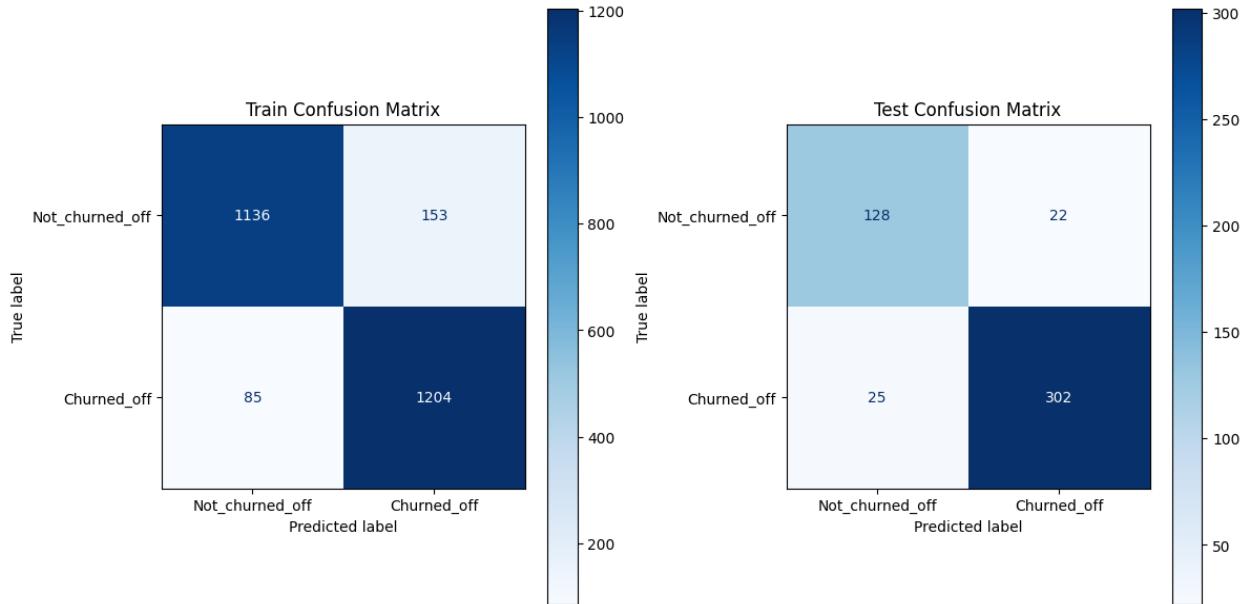
# Print model name and times
print(f"Model: {name}")
print(f"params: {params}")
print(f"Training Time: {training_time:.4f} seconds")
print(f"Testing Time: {testing_time:.4f} seconds")

# logging time_df, feature_importance_df,
mlflow_logging_and_metric_printing
time_df,feature_importance_df =
log_time_and_feature_importances_df(time_df,feature_importance_df,name
,training_time,testing_time,cascade_models,ola_features,bal_type)
mlflow_logging_and_metric_printing(cascade_models,name,bal_type,X_trai

```

```
n_bal,y_train_bal,X_test_bal,y_test_bal,y_pred_bal_train,y_pred_bal_te  
st,tuning_score,**params)
```





## CHAPTER 6 RESULTS EVALUATION

### ALL\_LOGGED\_METRICS

```
all_logged_metrics_df = all_logged_metrics()
all_logged_metrics_df
```

		run_name	bal_type	\
0	Two_Level_Cascading_Classifier_on_Balanced_Dat...		Balanced	
1	Two_Level_Cascading_Classifier_on_Imbalanced_D...		Imbalanced	
2	Tuned_Stacking_Classifier_on_Balanced_Dataset		Balanced	
3	Tuned_Stacking_Classifier_on_Imbalanced_Dataset		Imbalanced	
4	Tuned_Voting_Classifier_on_Balanced_Dataset		Balanced	
5	Tuned_Voting_Classifier_on_Imbalanced_Dataset		Imbalanced	
6	Tuned_Light_GBM_on_Balanced_Dataset		Balanced	
7	Tuned_Light_GBM_on_Imbalanced_Dataset		Imbalanced	
8	Tuned_XG_boost_on_Balanced_Dataset		Balanced	
9	Tuned_XG_boost_on_Imbalanced_Dataset		Imbalanced	
10	Tuned_Gradient_boost_on_Balanced_Dataset		Balanced	
11	Tuned_Gradient_boost_on_Imbalanced_Dataset		Imbalanced	
12	Tuned_Adaboost_on_Balanced_Dataset		Balanced	
13	Tuned_Adaboost_on_Imbalanced_Dataset		Imbalanced	
14	Tuned_Bagging_RF_on_Balanced_Dataset		Balanced	
15	Tuned_Bagging_RF_on_Imbalanced_Dataset		Imbalanced	
16	Tuned_Random_Forest_on_Balanced_Dataset		Balanced	
17	Tuned_Random_Forest_on_Imbalanced_Dataset		Imbalanced	
18	Tuned_Decision_Tree_on_Balanced_Dataset		Balanced	
19	Tuned_Decision_Tree_on_Imbalanced_Dataset		Imbalanced	
20	Tuned_SVC_on_Balanced_Dataset		Balanced	

```
21             Tuned_SVC_on_Imbalanced_Dataset  Imbalanced
22             Tuned_KNN_on_Balanced_Dataset   Balanced
23             Tuned_KNN_on_Imbalanced_Dataset Imbalanced
24             Tuned_MLPClassifier_on_Balanced_Dataset  Balanced
25             Tuned_MLPClassifier_on_Imbalanced_Dataset Imbalanced
26             Tuned_Logistic_Regression_on_Balanced_Dataset  Balanced
27             Tuned_Logistic_Regression_on_Imbalanced_Dataset Imbalanced
28             Simple_Logistic_Regresssion_on_balanced_Dataset  Balanced
29             Simple_Logistic_Regression_on_Imbalanced_Dataset Imbalanced

                                         params_dict
metrics.Accuracy_test \
0                         {'params.bal_type': 'Balanced'}
0.901468
1                         {'params.bal_type': 'Imbalanced'}
0.685535
2   {'params.bal_type': 'Balanced', 'params.passth...
0.899371
3   {'params.bal_type': 'Imbalanced', 'params.pass...
0.911950
4   {'params.bal_type': 'Balanced', 'params.votin...
0.899371
5   {'params.bal_type': 'Imbalanced', 'params.voti...
0.916143
6   {'params.bal_type': 'Balanced', 'params.learni...
0.886792
7   {'params.bal_type': 'Imbalanced', 'params.lear...
0.916143
8   {'params.bal_type': 'Balanced', 'params.learni...
0.895178
9   {'params.bal_type': 'Imbalanced', 'params.lear...
0.909853
10  {'params.bal_type': 'Balanced', 'params.learni...
0.888889
11  {'params.bal_type': 'Imbalanced', 'params.lear...
0.920335
12  {'params.bal_type': 'Balanced', 'params.learni...
0.893082
13  {'params.bal_type': 'Imbalanced', 'params.lear...
0.903564
14  {'params.bal_type': 'Balanced', 'params.random...
0.901468
15  {'params.bal_type': 'Imbalanced', 'params.rand...
0.907757
16  {'params.bal_type': 'Balanced', 'params.random...
0.880503
17  {'params.bal_type': 'Imbalanced', 'params.rand...
0.884696
18  {'params.bal_type': 'Balanced', 'params.random...
```

```
0.750524
19 {'params.bal_type': 'Imbalanced', 'params.rand...
0.865828
20 {'params.bal_type': 'Balanced', 'params.random...
0.836478
21 {'params.bal_type': 'Imbalanced', 'params.rand...
0.884696
22 {'params.bal_type': 'Balanced', 'params.weight...
0.823899
23 {'params.bal_type': 'Imbalanced', 'params.weig...
0.870021
24 {'params.bal_type': 'Balanced', 'params.learni...
0.811321
25 {'params.bal_type': 'Imbalanced', 'params.lear...
0.828092
26 {'params.bal_type': 'Balanced', 'params.C': '1...
0.870021
27 {'params.bal_type': 'Imbalanced', 'params.C': ...
0.890985
28 {'params.bal_type': 'Balanced', 'params.n_jobs...
0.863732
29 {'params.bal_type': 'Imbalanced', 'params.n_jo...
0.909853
```

	metrics.Accuracy_train	metrics.F1_score_test
metrics.F1_score_train \		
0	0.907680	0.927803
0.910053		
1	0.676996	0.813433
0.807391		
2	0.999224	0.926154
0.999224		
3	0.957458	0.936747
0.968906		
4	1.000000	0.925926
1.000000		
5	0.941702	0.940120
0.957455		
6	1.000000	0.916667
1.000000		
7	0.938550	0.939940
0.955121		
8	0.999612	0.923077
0.999612		
9	0.955882	0.936107
0.968037		
10	1.000000	0.918083
1.000000		
11	0.949580	0.942943

0.963162			
12	1.000000	0.921899	
1.000000			
13	0.972689	0.930723	
0.980031			
14	0.907680	0.927803	
0.910053			
15	0.918067	0.933535	
0.939582			
16	0.863848	0.912711	
0.871098			
17	0.892857	0.915254	
0.920561			
18	0.816912	0.785586	
0.787196			
19	0.845063	0.906706	
0.889554			
20	0.972847	0.880000	
0.972741			
21	0.933824	0.914197	
0.950040			
22	1.000000	0.864952	
1.000000			
23	1.000000	0.908555	
1.000000			
24	0.799457	0.860248	
0.806149			
25	0.830882	0.883191	
0.883079			
26	0.869279	0.902208	
0.871423			
27	0.879727	0.919255	
0.908801			
28	0.865787	0.897959	
0.868241			
29	0.886555	0.936107	
0.918491			
metrics.F2_score_test metrics.F2_score_train metrics.Pr_auc_test			
\			
0	0.925245	0.924305	NaN
1	0.915966	0.912890	NaN
2	0.922747	0.999224	0.963908
3	0.945289	0.974969	0.963391
4	0.920810	1.000000	0.971881

	metrics.Pr_auc_train	metrics.Precision_test	
5	0.952092	0.964330	NaN
6	0.911602	1.000000	0.966628
7	0.950213	0.961538	0.968791
8	0.919681	0.999845	0.965202
9	0.952237	0.979215	0.968922
10	0.912162	1.000000	0.959753
11	0.953248	0.969411	0.962534
12	0.921053	1.000000	0.968593
13	0.939210	0.985937	0.975871
14	0.925245	0.924305	0.970759
15	0.940353	0.940456	0.971556
16	0.911873	0.899848	0.965255
17	0.911043	0.918415	0.968269
18	0.709635	0.717338	0.940238
19	0.932813	0.908535	0.926051
20	0.876763	0.970474	0.959062
21	0.903206	0.937549	0.970825
22	0.839052	1.000000	0.941121
23	0.928270	1.000000	0.952205
24	0.852308	0.822620	0.931503
25	0.920974	0.918290	0.934167
26	0.885449	0.880086	0.954531
27	0.910769	0.894481	0.959843
28	0.883807	0.877868	0.955210
29	0.952237	0.933712	0.959764
metrics.Pr_auc_train metrics.Precision_test			

```
metrics.Precision_train \
0           NaN      0.932099
0.887251
1           NaN      0.685535
0.676996
2           0.999996  0.931889
0.999224
3           0.993925  0.922849
0.958967
4           1.000000  0.934579
1.000000
5           NaN      0.920821
0.946212
6           1.000000  0.925234
1.000000
7           0.992846  0.923304
0.944613
8           1.000000  0.928793
0.999225
9           0.995976  0.910405
0.949963
10          1.000000  0.928125
1.000000
11          0.994461  0.926254
0.952923
12          1.000000  0.923313
1.000000
13          0.999066  0.916914
0.970342
14          0.969030  0.932099
0.887251
15          0.986321  0.922388
0.938128
16          0.945343  0.914110
0.827057
17          0.973503  0.922360
0.924159
18          0.902706  0.956140
0.939720
19          0.917734  0.866295
0.859624
20          0.994054  0.885449
0.976544
21          0.991506  0.933121
0.971614
22          1.000000  0.911864
1.000000
23          1.000000  0.877493
1.000000
```

24	0.901928	0.873817	
0.780116			
25	0.948730	0.826667	
0.830034			
26	0.946021	0.931596	
0.857357			
27	0.973839	0.933754	
0.933715			
28	0.946333	0.922581	
0.852655			
29	0.972315	0.910405	
0.894195			
	metrics.Recall_test	metrics.Recall_train	metrics.Roc_auc_test \
0	0.923547	0.934057	NaN
1	1.000000	1.000000	NaN
2	0.920489	0.999224	0.938899
3	0.951070	0.979054	0.947971
4	0.917431	1.000000	0.949541
5	0.960245	0.968968	NaN
6	0.908257	1.000000	0.942712
7	0.957187	0.965865	0.956962
8	0.917431	1.000000	0.947706
9	0.963303	0.986811	0.956901
10	0.908257	1.000000	0.940510
11	0.960245	0.973623	0.955270
12	0.920489	1.000000	0.943099
13	0.944954	0.989915	0.955923
14	0.923547	0.934057	0.948196
15	0.944954	0.941040	0.951825
16	0.911315	0.920093	0.933527
17	0.908257	0.916990	0.942385
18	0.666667	0.677269	0.865443
19	0.951070	0.921645	0.818410
20	0.874618	0.968968	0.911213
21	0.896024	0.929403	0.945545
22	0.822630	1.000000	0.898716
23	0.941896	1.000000	0.912273
24	0.847095	0.833980	0.872701
25	0.948012	0.943367	0.885770
26	0.874618	0.885958	0.929766
27	0.905199	0.885182	0.942120
28	0.874618	0.884407	0.929154
29	0.963303	0.944143	0.942650
	metrics.Roc_auc_train		
	metrics.hyper_parameter_tuning_best_est_score		
0		NaN	
0.000000			

1	NaN
0.000000	
2	0.999996
0.922820	
3	0.988263
0.913331	
4	1.000000
0.916998	
5	NaN
0.913854	
6	1.000000
0.910406	
7	0.985067
0.914903	
8	1.000000
0.918547	
9	0.991914
0.912279	
10	1.000000
0.920494	
11	0.988905
0.916482	
12	1.000000
0.918553	
13	0.998031
0.915434	
14	0.968118
0.882468	
15	0.972392
0.904398	
16	0.941240
0.852980	
17	0.948374
0.883395	
18	0.878143
0.815365	
19	0.806198
0.834027	
20	0.992869
0.897610	
21	0.982138
0.883922	
22	1.000000
0.875879	
23	1.000000
0.874990	
24	0.894488
0.787817	
25	0.899079

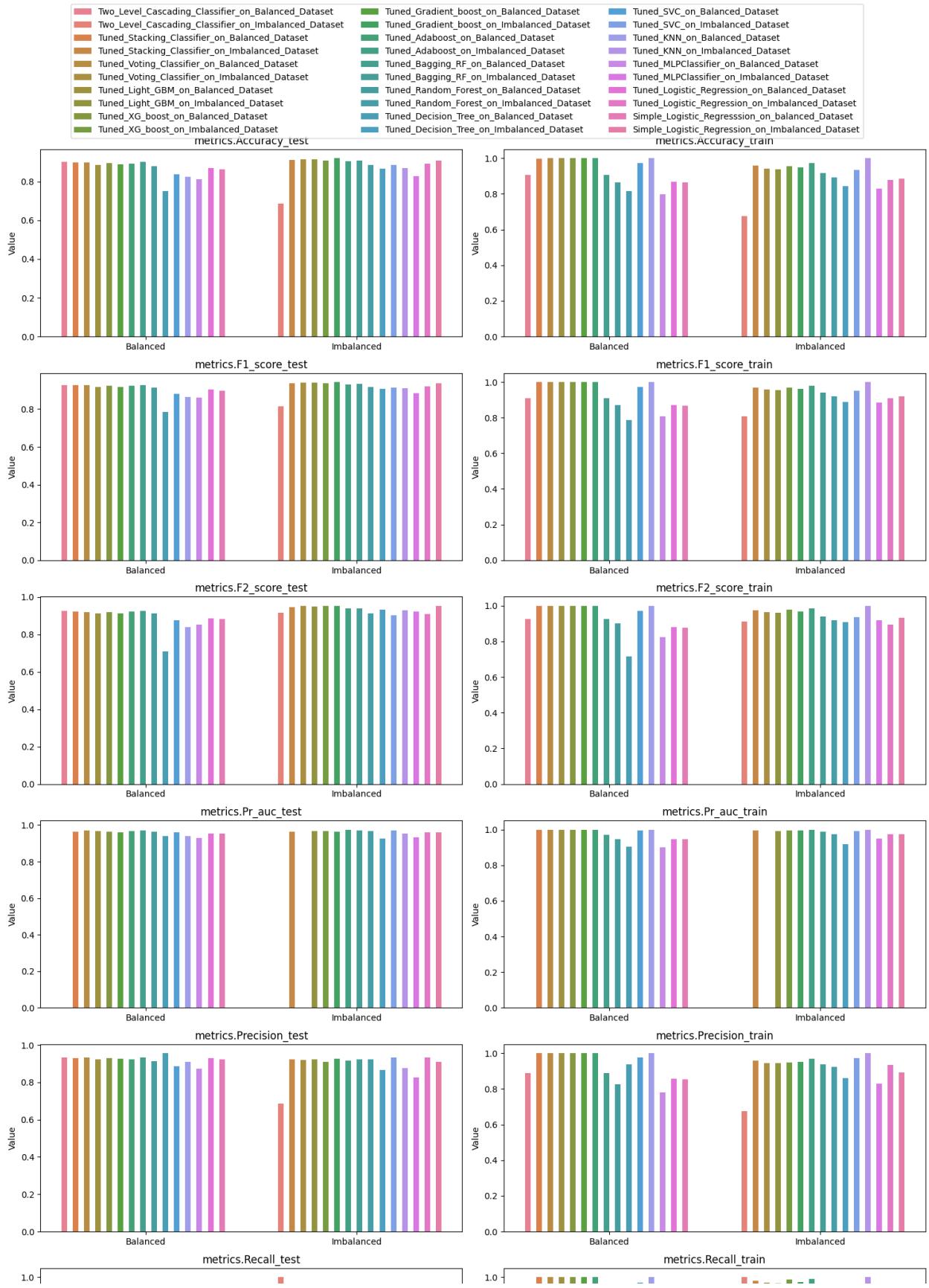
```
0.820902  
26          0.942694  
0.862297  
27          0.948177  
0.877620  
28          0.942393  
0.000000  
29          0.945990  
0.000000
```

Fill zero inplace of Null values

```
all_logged_metrics_df.fillna(value = 0,inplace=True)
```

```
all_logged_metrics_df_plots
```

```
all_logged_metrics_df_plots(all_logged_metrics_df)
```



printing best models according to each metric

```
# Assuming your DataFrame is named 'all_logged_metrics_df'
metrics_columns =
all_logged_metrics_df.columns[all_logged_metrics_df.columns.str.starts
with('metrics.')]

for metric in metrics_columns:
    best_model =
all_logged_metrics_df.loc[all_logged_metrics_df[metric].idxmax(),
'run_name']
    best_params =
all_logged_metrics_df.loc[all_logged_metrics_df[metric].idxmax(),
'params_dict']
    print(f'{metric} -> best_model -> \'{best_model}\', best_params ->
{best_params}')

metrics.Accuracy_test -> best_model ->
"Tuned_Gradient_boost_on_Imbalanced_Dataset", best_params ->
{'params.bal_type': 'Imbalanced', 'params.learning_rate':
'0.06676646193550176', 'params.random_state': '42',
'params.subsample': '0.94598407522243', 'params.max_depth': '7',
'params.n_estimators': '493', 'params.min_impurity_decrease':
'0.09613152881849074', 'params.min_samples_leaf': '19',
'params.max_features': 'log2', 'params.min_samples_split': '11'}
metrics.Accuracy_train -> best_model ->
"Tuned_Voting_Classifier_on_Balanced_Dataset", best_params ->
{'params.bal_type': 'Balanced', 'params.voting': 'soft',
'params.weights': '[1.0, 0.5, 0.3333333333333333, 0.25, 0.2,
0.1666666666666666, 0.14285714285714285, 0.125, 0.1111111111111111,
0.1, 0.09090909090909091, 0.0833333333333333]', 'params.n_jobs': '-1'}
metrics.F1_score_test -> best_model ->
"Tuned_Gradient_boost_on_Imbalanced_Dataset", best_params ->
{'params.bal_type': 'Imbalanced', 'params.learning_rate':
'0.06676646193550176', 'params.random_state': '42',
'params.subsample': '0.94598407522243', 'params.max_depth': '7',
'params.n_estimators': '493', 'params.min_impurity_decrease':
'0.09613152881849074', 'params.min_samples_leaf': '19',
'params.max_features': 'log2', 'params.min_samples_split': '11'}
metrics.F1_score_train -> best_model ->
"Tuned_Voting_Classifier_on_Balanced_Dataset", best_params ->
{'params.bal_type': 'Balanced', 'params.voting': 'soft',
'params.weights': '[1.0, 0.5, 0.3333333333333333, 0.25, 0.2,
0.1666666666666666, 0.14285714285714285, 0.125, 0.1111111111111111,
0.1, 0.09090909090909091, 0.0833333333333333]', 'params.n_jobs': '-1'}
metrics.F2_score_test -> best_model ->
"Tuned_Gradient_boost_on_Imbalanced_Dataset", best_params ->
{'params.bal_type': 'Imbalanced', 'params.learning_rate':
```

```

'0.06676646193550176', 'params.random_state': '42',
'params.subsample': '0.94598407522243', 'params.max_depth': '7',
'params.n_estimators': '493', 'params.min_impurity_decrease':
'0.09613152881849074', 'params.min_samples_leaf': '19',
'params.max_features': 'log2', 'params.min_samples_split': '11'}
metrics.F2_score_train -> best_model ->
"Tuned_Voting_Classifier_on_Balanced_Dataset", best_params ->
{'params.bal_type': 'Balanced', 'params.voting': 'soft',
'params.weights': '[1.0, 0.5, 0.3333333333333333, 0.25, 0.2,
0.1666666666666666, 0.14285714285714285, 0.125, 0.1111111111111111,
0.1, 0.09090909090909091, 0.0833333333333333]', 'params.n_jobs': '-1'}
metrics.Pr_auc_test -> best_model ->
"Tuned_Adaboost_on_Imbalanced_Dataset", best_params ->
{'params.bal_type': 'Imbalanced', 'params.learning_rate':
'0.1370605126518848', 'params.n_estimators': '313',
'params.algorithm': 'SAMME'}
metrics.Pr_auc_train -> best_model ->
"Tuned_Voting_Classifier_on_Balanced_Dataset", best_params ->
{'params.bal_type': 'Balanced', 'params.voting': 'soft',
'params.weights': '[1.0, 0.5, 0.3333333333333333, 0.25, 0.2,
0.1666666666666666, 0.14285714285714285, 0.125, 0.1111111111111111,
0.1, 0.09090909090909091, 0.0833333333333333]', 'params.n_jobs': '-1'}
metrics.Precision_test -> best_model ->
"Tuned_Decision_Tree_on_Balanced_Dataset", best_params ->
{'params.bal_type': 'Balanced', 'params.random_state': '42',
'params.max_depth': '15', 'params.min_impurity_decrease':
'0.05812382214226123', 'params.min_samples_leaf': '11',
'params.max_features': 'log2', 'params.min_samples_split': '5',
'params.max_leaf_nodes': '39', 'params ccp_alpha':
'0.011483682473920355', 'params.min_weight_fraction_leaf':
'0.16032109607975714', 'params.criterion': 'entropy',
'params.splitter': 'best'}
metrics.Precision_train -> best_model ->
"Tuned_Voting_Classifier_on_Balanced_Dataset", best_params ->
{'params.bal_type': 'Balanced', 'params.voting': 'soft',
'params.weights': '[1.0, 0.5, 0.3333333333333333, 0.25, 0.2,
0.1666666666666666, 0.14285714285714285, 0.125, 0.1111111111111111,
0.1, 0.09090909090909091, 0.0833333333333333]', 'params.n_jobs': '-1'}
metrics.Recall_test -> best_model ->
"Two_Level_Cascading_Classifier_on_Imbalanced_Dataset", best_params ->
{'params.bal_type': 'Imbalanced'}
metrics.Recall_train -> best_model ->
"Two_Level_Cascading_Classifier_on_Imbalanced_Dataset", best_params ->
{'params.bal_type': 'Imbalanced'}
metrics.Roc_auc_test -> best_model ->
"Tuned_Light_GBM_on_Imbalanced_Dataset", best_params ->

```

```

{'params.bal_type': 'Imbalanced', 'params.learning_rate': '0.11661537060572182', 'params.min_data_in_leaf': '77', 'params.bagging_fraction': '0.7040356346288988', 'params.num_leaves': '33', 'params.colsample_bytree': '1.1159854502601791', 'params.min_split_gain': '0.003531135494023907', 'params.random_state': '42', 'params.scale_pos_weight': '1.1538632326761582', 'params.subsample': '0.6488880503324447', 'params.bagging_freq': '5', 'params.max_depth': '11', 'params.min_child_weight': '0.0015229613542912736', 'params.n_estimators': '308', 'params.reg_alpha': '0.4790367400596901', 'params.feature_fraction': '0.6880247346154161', 'params.reg_lambda': '1.0519111269531267', 'params.min_child_samples': '45', 'params.boosting_type': 'dart', 'params.objective': 'binary'}
metrics.Roc_auc_train -> best_model ->
"Tuned_Voting_Classifier_on_Balanced_Dataset", best_params ->
{'params.bal_type': 'Balanced', 'params.voting': 'soft', 'params.weights': '[1.0, 0.5, 0.3333333333333333, 0.25, 0.2, 0.1666666666666666, 0.14285714285714285, 0.125, 0.1111111111111111, 0.1, 0.09090909090909091, 0.0833333333333333]', 'params.n_jobs': '-1'}
metrics.hyper_parameter_tuning_best_est_score -> best_model ->
"Tuned_Stacking_Classifier_on_Balanced_Dataset", best_params ->
{'params.bal_type': 'Balanced', 'params.passthrough': 'True', 'params.stack_method': 'predict'}

```

## Observation

On train dataset, KNN is dominating in on trained models. Because KNN has no training time. It simply stores the entire training dataset in memory. Voting Classifier on Balanced Dataset has 100% on most metrics of train dataset

On test dataset, Boosting algorithms like GBDT, Adaboost, XGBoost, Light\_GBM are performing better than Bagging and Stacking.

Cascading and Stacking ensemble learning techniques provide best models but so slow

Boosting models are provide next best models but very fast. Sufficient for this dataset  
Balanced Dataset has more generalization than Imbalanced Datasets.

Algorithms like Adaboost, LightGBM works better on Imbalanced Datsets also.

Recall is important in this Case Study because in a churn prediction scenario, missing a churned driver (false negative) can be costly for a business. If the model fails to identify a driver who is likely to churn, the company loses the opportunity to intervene (e.g., offering incentives or addressing issues) to retain the driver.

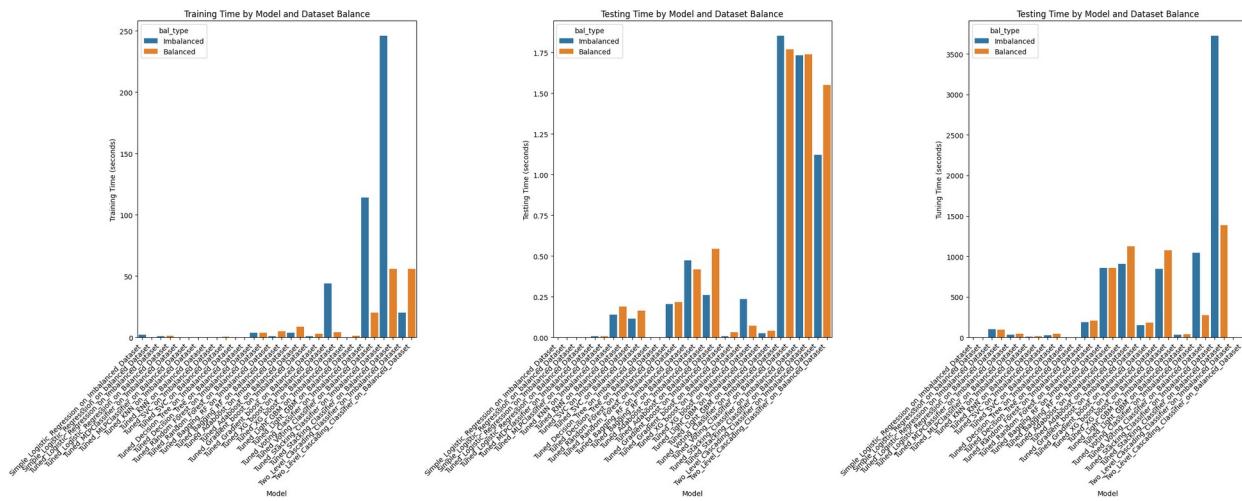
So on Recall\_test --> Cascading on Imbalanced , F2 Score\_test --> Gradient boost on balanced, Pr\_auc\_test --> Adaboost on Imbalanced Models are dominating. Clearly Boosting is effective.

But because of lack of much data, The max\_accuracy\_test is limited to 92%,  
max\_recall\_test is limited to 96%

# TIME\_DF AND ITS PLOT

## time\_df\_plots

```
time_df_plots(time_df)
```



## Observation

Training\_time and testing\_time is more for ensemble models because of combination of individual models

clearly balanced dataset has more training time and testing time compared to imbalanced dataset because of oversampling

Stacking hyper parametric tuning has more time taken.

Hyper parametric tuning of ensemble models are significantly higher than individual models

# FEATURE IMPORTANCE DF

```
feature_importance_df
```

```
Simple_Logistic_Regression_on_Imbalanced_Dataset \ 
Age
0.109896
City
0.069281
Education_Level
0.037378
Gender
0.047752
Grade
0.778416
Grade_raise
```

0.200566  
Income\_raise -  
0.200566  
Income\_range -  
0.722095  
Join\_month -  
0.365371  
Join\_year -  
2.013868  
Joining Designation -  
0.130488  
QuarterlyRating\_last -  
0.720011  
QuarterlyRating\_mean -  
1.805258  
QuarterlyRating\_range -  
0.145085  
QuarterlyRating\_std -  
0.067082  
Rating\_Promotion -  
0.148453  
Reportings -  
3.542816  
TBV\_raise -  
0.966004  
TBV\_sum\_positive -  
2.495274  
TBV\_sum\_negative -  
0.039156  
TBV\_mean\_positive -  
1.608817  
TBV\_mean\_negative -  
0.059553  
Income\_sum\_log -  
0.326551  
QuarterlyRating\_sum\_log -  
3.744988  
TBV\_mean\_positive\_log -  
0.372347  
TBV\_mean\_negative\_log -  
0.145310  
TBV\_range\_log -  
0.114218  
TBV\_std\_log -  
0.306035  
TBV\_sum\_positive\_log -  
0.113332  
TBV\_sum\_negative\_log -  
0.143163

```
Simple_Logistic_Regressson_on_balanced_Dataset \n
Age\n
0.095305\n
City\n
0.098565\n
Education_Level\n
0.256508\n
Gender\n
0.335353\n
Grade\n
0.293402\n
Grade_raise\n
0.095718\n
Income_raise\n
0.095718\n
Income_range\n
0.239048\n
Join_month\n
0.312485\n
Join_year\n
2.256950\n
Joining_Designation\n
0.143158\n
QuarterlyRating_last\n
0.178099\n
QuarterlyRating_mean\n
2.392407\n
QuarterlyRating_range\n
1.534083\n
QuarterlyRating_std\n
1.242552\n
Rating_Promotion\n
0.440989\n
Reportings\n
3.473494\n
TBV_raise\n
1.014269\n
TBV_sum_positive\n
3.752669\n
TBV_sum_negative\n
0.052252\n
TBV_mean_positive\n
1.715850\n
TBV_mean_negative\n
0.065882\n
Income_sum_log\n
0.345360
```

```
QuarterlyRating_sum_log  
4.377599  
TBV_mean_positive_log  
0.046581  
TBV_mean_negative_log  
0.128259  
TBV_range_log  
0.083093  
TBV_std_log  
0.225809  
TBV_sum_positive_log  
0.090800  
TBV_sum_negative_log  
0.127302
```

```
Tuned_Logistic_Regression_on_Imbalanced_Dataset \
```

```
Age  
0.135382  
City  
0.083032  
Education_Level  
0.088890  
Gender  
0.051474  
Grade  
0.640852  
Grade_raise  
0.364203  
Income_raise  
0.364203  
Income_range  
1.126104  
Join_month  
0.389475  
Join_year  
2.588902  
Joining_Designation  
0.016504  
QuarterlyRating_last  
0.704083  
QuarterlyRating_mean  
2.388509  
QuarterlyRating_range  
0.409002  
QuarterlyRating_std  
0.347699  
Rating_Promotion  
0.193882
```

Reportings	-
4.716596	-
TBV_raise	-
1.039046	-
TBV_sum_positive	-
3.309904	-
TBV_sum_negative	-
0.046937	-
TBV_mean_positive	-
1.933124	-
TBV_mean_negative	-
0.082407	-
Income_sum_log	-
0.109277	-
QuarterlyRating_sum_log	-
5.246220	-
TBV_mean_positive_log	-
0.372802	-
TBV_mean_negative_log	-
0.205068	-
TBV_range_log	-
0.053926	-
TBV_std_log	-
0.692717	-
TBV_sum_positive_log	-
0.493279	-
TBV_sum_negative_log	-
0.201487	-
Tuned_Logistic_Regression_on_Balanced_Dataset	
\	
Age	0.114213
City	0.091471
Education_Level	0.288007
Gender	0.345657
Grade	-0.154351
Grade_raise	0.077249
Income_raise	0.077249
Income_range	0.368688
Join_month	-0.317873
Join_year	-2.569129

Joining_Designation	0.048449
QuarterlyRating_last	-0.112876
QuarterlyRating_mean	-2.984543
QuarterlyRating_range	1.970177
QuarterlyRating_std	-1.651725
Rating_Promotion	-0.504848
Reportings	-4.345153
TBV_raise	-1.058345
TBV_sum_positive	-4.458049
TBV_sum_negative	0.058624
TBV_mean_positive	2.166977
TBV_mean_negative	0.082863
Income_sum_log	-0.556826
QuarterlyRating_sum_log	5.676545
TBV_mean_positive_log	-0.090701
TBV_mean_negative_log	0.181600
TBV_range_log	-0.075514
TBV_std_log	-0.359542
TBV_sum_positive_log	-0.040271
TBV_sum_negative_log	0.179480

	Tuned_Decision_Tree_on_Imbalanced_Dataset \
Age	0.00000
City	0.00000
Education_Level	0.00000
Gender	0.00000
Grade	0.00000
Grade_raise	0.00000
Income_raise	0.00000
Income_range	0.00000

Join_month	0.00000
Join_year	0.00000
Joining_Designation	0.00000
QuarterlyRating_last	0.00000
QuarterlyRating_mean	0.00000
QuarterlyRating_range	0.00000
QuarterlyRating_std	0.00000
Rating_Promotion	0.00000
Reportings	0.29645
TBV_raise	0.70355
TBV_sum_positive	0.00000
TBV_sum_negative	0.00000
TBV_mean_positive	0.00000
TBV_mean_negative	0.00000
Income_sum_log	0.00000
QuarterlyRating_sum_log	0.00000
TBV_mean_positive_log	0.00000
TBV_mean_negative_log	0.00000
TBV_range_log	0.00000
TBV_std_log	0.00000
TBV_sum_positive_log	0.00000
TBV_sum_negative_log	0.00000

#### Tuned\_Decision\_Tree\_on\_Balanced\_Dataset \

Age	0.000000
City	0.000000
Education_Level	0.000000
Gender	0.000000
Grade	0.000000
Grade_raise	0.000000
Income_raise	0.000000
Income_range	0.000000
Join_month	0.000000
Join_year	0.377202
Joining_Designation	0.000000
QuarterlyRating_last	0.000000
QuarterlyRating_mean	0.000000
QuarterlyRating_range	0.000000
QuarterlyRating_std	0.000000
Rating_Promotion	0.000000
Reportings	0.000000
TBV_raise	0.332372
TBV_sum_positive	0.000000
TBV_sum_negative	0.000000
TBV_mean_positive	0.000000
TBV_mean_negative	0.000000
Income_sum_log	0.000000
QuarterlyRating_sum_log	0.000000
TBV_mean_positive_log	0.000000

TBV_mean_negative_log	0.000000
TBV_range_log	0.000000
TBV_std_log	0.000000
TBV_sum_positive_log	0.290426
TBV_sum_negative_log	0.000000
Tuned_Random_Forest_on_Imbalanced_Dataset \	
Age	0.000247
City	0.000000
Education_Level	0.000000
Gender	0.000000
Grade	0.005073
Grade_raise	0.000139
Income_raise	0.000059
Income_range	0.000214
Join_month	0.006255
Join_year	0.186109
Joining_Designation	0.012207
QuarterlyRating_last	0.129287
QuarterlyRating_mean	0.036967
QuarterlyRating_range	0.003579
QuarterlyRating_std	0.002481
Rating_Promotion	0.057482
Reportings	0.103556
TBV_raise	0.174634
TBV_sum_positive	0.061848
TBV_sum_negative	0.000000
TBV_mean_positive	0.022343
TBV_mean_negative	0.000000
Income_sum_log	0.024478
QuarterlyRating_sum_log	0.071582
TBV_mean_positive_log	0.021088
TBV_mean_negative_log	0.000000
TBV_range_log	0.008869
TBV_std_log	0.005642
TBV_sum_positive_log	0.065862
TBV_sum_negative_log	0.000000
Tuned_Random_Forest_on_Balanced_Dataset \	
Age	0.000588
City	0.000000
Education_Level	0.000000
Gender	0.001334
Grade	0.001030
Grade_raise	0.000046
Income_raise	0.000270
Income_range	0.000801
Join_month	0.001465
Join_year	0.170910

Joining_Designation	0.005227
QuarterlyRating_last	0.156506
QuarterlyRating_mean	0.048912
QuarterlyRating_range	0.001464
QuarterlyRating_std	0.005065
Rating_Promotion	0.041135
Reportings	0.097673
TBV_raise	0.175896
TBV_sum_positive	0.062365
TBV_sum_negative	0.000000
TBV_mean_positive	0.020651
TBV_mean_negative	0.000000
Income_sum_log	0.029671
QuarterlyRating_sum_log	0.084899
TBV_mean_positive_log	0.017868
TBV_mean_negative_log	0.000000
TBV_range_log	0.005117
TBV_std_log	0.004314
TBV_sum_positive_log	0.066793
TBV_sum_negative_log	0.000000

	Tuned_Adaboost_on_Imbalanced_Dataset \
Age	6.314434e-02
City	4.760089e-02
Education_Level	1.760755e-02
Gender	4.308098e-03
Grade	3.782188e-03
Grade_raise	6.737049e-18
Income_raise	5.466737e-18
Income_range	1.180588e-04
Join_month	7.181951e-02
Join_year	1.735583e-01
Joining_Designation	1.841199e-02
QuarterlyRating_last	1.960056e-02
QuarterlyRating_mean	1.667176e-02
QuarterlyRating_range	1.857619e-03
QuarterlyRating_std	4.960793e-02
Rating_Promotion	7.182531e-03
Reportings	1.873634e-01
TBV_raise	5.553677e-02
TBV_sum_positive	2.158039e-02
TBV_sum_negative	3.278356e-18
TBV_mean_positive	1.526405e-02
TBV_mean_negative	7.569407e-18
Income_sum_log	1.058234e-01
QuarterlyRating_sum_log	1.964270e-02
TBV_mean_positive_log	1.498850e-02
TBV_mean_negative_log	5.588173e-17
TBV_range_log	3.525597e-02

TBV_std_log	3.173435e-02
TBV_sum_positive_log	1.753909e-02
TBV_sum_negative_log	1.940480e-18

	Tuned_Adaboost_on_Balanced_Dataset \
Age	1.072546e-01
City	1.298196e-01
Education_Level	3.317097e-02
Gender	1.871191e-02
Grade	1.633346e-02
Grade_raise	1.590947e-13
Income_raise	0.000000e+00
Income_range	2.689280e-06
Join_month	1.074560e-01
Join_year	2.629639e-02
Joining_Designation	1.400623e-02
QuarterlyRating_last	6.216727e-03
QuarterlyRating_mean	2.405106e-02
QuarterlyRating_range	3.527117e-03
QuarterlyRating_std	2.850540e-02
Rating_Promotion	9.024465e-03
Reportings	4.781582e-02
TBV_raise	2.024027e-02
TBV_sum_positive	1.993274e-02
TBV_sum_negative	7.484432e-19
TBV_mean_positive	2.430918e-02
TBV_mean_negative	9.397426e-06
Income_sum_log	1.847530e-01
QuarterlyRating_sum_log	6.253923e-02
TBV_mean_positive_log	1.513308e-02
TBV_mean_negative_log	0.000000e+00
TBV_range_log	3.753666e-02
TBV_std_log	4.520532e-02
TBV_sum_positive_log	1.814872e-02
TBV_sum_negative_log	3.484355e-19

	Tuned_Gradient_boost_on_Imbalanced_Dataset \
Age	0.010894
City	0.010858
Education_Level	0.002599
Gender	0.001942
Grade	0.007243
Grade_raise	0.000000
Income_raise	0.000000
Income_range	0.000000
Join_month	0.031496
Join_year	0.188310
Joining_Designation	0.023740
QuarterlyRating_last	0.059765

QuarterlyRating_mean	0.032963
QuarterlyRating_range	0.004235
QuarterlyRating_std	0.011115
Rating_Promotion	0.048599
Reportings	0.078377
TBV_raise	0.185037
TBV_sum_positive	0.053515
TBV_sum_negative	0.000000
TBV_mean_positive	0.036183
TBV_mean_negative	0.000000
Income_sum_log	0.060696
QuarterlyRating_sum_log	0.065027
TBV_mean_positive_log	0.019968
TBV_mean_negative_log	0.000000
TBV_range_log	0.020995
TBV_std_log	0.016268
TBV_sum_positive_log	0.030175
TBV_sum_negative_log	0.000000

	Tuned_Gradient_boost_on_Balanced_Dataset \
Age	2.233592e-02
City	2.664572e-02
Education_Level	9.719161e-03
Gender	8.120786e-03
Grade	8.943055e-03
Grade_raise	7.419100e-05
Income_raise	2.278973e-03
Income_range	6.836883e-03
Join_month	4.425638e-02
Join_year	1.814671e-01
Joining_Designation	1.281694e-02
QuarterlyRating_last	6.392784e-02
QuarterlyRating_mean	2.033221e-02
QuarterlyRating_range	5.233838e-03
QuarterlyRating_std	1.615002e-02
Rating_Promotion	2.278207e-02
Reportings	1.090421e-01
TBV_raise	1.242922e-01
TBV_sum_positive	6.451764e-02
TBV_sum_negative	6.356788e-06
TBV_mean_positive	2.562671e-02
TBV_mean_negative	1.426592e-04
Income_sum_log	4.553023e-02
QuarterlyRating_sum_log	4.795282e-02
TBV_mean_positive_log	2.714039e-02
TBV_mean_negative_log	2.312832e-07
TBV_range_log	2.161685e-02
TBV_std_log	1.991877e-02
TBV_sum_positive_log	6.201021e-02
TBV_sum_negative_log	2.817907e-04

	Tuned_XG_boost_on_Imbalanced_Dataset \
Age	0.010045
City	0.009428
Education_Level	0.009156
Gender	0.008312
Grade	0.011849
Grade_raise	0.000000
Income_raise	0.000000
Income_range	0.000000
Join_month	0.024300
Join_year	0.110274
Joining Designation	0.017069
QuarterlyRating_last	0.186792
QuarterlyRating_mean	0.018081
QuarterlyRating_range	0.012241
QuarterlyRating_std	0.011256
Rating_Promotion	0.021871
Reportings	0.071407
TBV_raise	0.309934
TBV_sum_positive	0.020583
TBV_sum_negative	0.000000
TBV_mean_positive	0.011966
TBV_mean_negative	0.000000
Income_sum_log	0.012528
QuarterlyRating_sum_log	0.053771
TBV_mean_positive_log	0.013205
TBV_mean_negative_log	0.000000
TBV_range_log	0.012925
TBV_std_log	0.011131
TBV_sum_positive_log	0.031876
TBV_sum_negative_log	0.000000

	Tuned_XG_boost_on_Balanced_Dataset \
Age	0.010969
City	0.012143
Education_Level	0.012719
Gender	0.015871
Grade	0.012438
Grade_raise	0.006970
Income_raise	0.000000
Income_range	0.005333
Join_month	0.017636
Join_year	0.126778
Joining Designation	0.011477
QuarterlyRating_last	0.357724
QuarterlyRating_mean	0.017301
QuarterlyRating_range	0.015010
QuarterlyRating_std	0.015181

Rating_Promotion	0.027962
Reportings	0.047915
TBV_raise	0.125673
TBV_sum_positive	0.025497
TBV_sum_negative	0.000000
TBV_mean_positive	0.017057
TBV_mean_negative	0.000000
Income_sum_log	0.013909
QuarterlyRating_sum_log	0.034673
TBV_mean_positive_log	0.020206
TBV_mean_negative_log	0.000000
TBV_range_log	0.014274
TBV_std_log	0.014405
TBV_sum_positive_log	0.020876
TBV_sum_negative_log	0.000000

#### Tuned\_Light\_GBM\_on\_Imbalanced\_Dataset \

Age	141
City	221
Education_Level	24
Gender	25
Grade	39
Grade_raise	0
Income_raise	0
Income_range	0
Join_month	264
Join_year	320
Joining_Designation	68
QuarterlyRating_last	75
QuarterlyRating_mean	63
QuarterlyRating_range	5
QuarterlyRating_std	89
Rating_Promotion	26
Reportings	372
TBV_raise	183
TBV_sum_positive	91
TBV_sum_negative	0
TBV_mean_positive	56
TBV_mean_negative	0
Income_sum_log	263
QuarterlyRating_sum_log	225
TBV_mean_positive_log	43
TBV_mean_negative_log	0
TBV_range_log	103
TBV_std_log	61
TBV_sum_positive_log	61
TBV_sum_negative_log	0

#### Tuned\_Light\_GBM\_on\_Balanced\_Dataset

Age	2362
City	3068
Education_Level	542
Gender	366
Grade	274
Grade_raise	0
Income_raise	0
Income_range	0
Join_month	2530
Join_year	1067
Joining_Designation	264
QuarterlyRating_last	289
QuarterlyRating_mean	937
QuarterlyRating_range	115
QuarterlyRating_std	1410
Rating_Promotion	360
Reportings	1661
TBV_raise	674
TBV_sum_positive	1188
TBV_sum_negative	0
TBV_mean_positive	994
TBV_mean_negative	0
Income_sum_log	4337
QuarterlyRating_sum_log	2506
TBV_mean_positive_log	766
TBV_mean_negative_log	0
TBV_range_log	1332
TBV_std_log	1183
TBV_sum_positive_log	822
TBV_sum_negative_log	0

```
# normalizing the values
scaler = StandardScaler()
feature_importance_scaled =
pd.DataFrame(scaler.fit_transform(feature_importance_df),
```

```
index=feature_importance_df.index,
```

```
columns=feature_importance_df.columns)
feature_importance_scaled
```

```
Simple_Logistic_Regression_on_Imbalanced_Dataset \
```

```
Age
```

```
0.265722
```

```
City
```

```
0.232419
```

```
Education_Level
```

```
0.206260
```

```
Gender
```

0.214766  
Grade -  
0.462668  
Grade\_raise -  
0.011152  
Income\_raise -  
0.011152  
Income\_range -  
0.767709  
Join\_month -  
0.123984  
Join\_year -  
1.475705  
Joining\_Designation -  
0.282607  
QuarterlyRating\_last -  
0.414778  
QuarterlyRating\_mean -  
1.304650  
QuarterlyRating\_range -  
0.294577  
QuarterlyRating\_std -  
0.120605  
Rating\_Promotion -  
0.053884  
Reportings -  
2.729400  
TBV\_raise -  
0.616485  
TBV\_sum\_positive -  
1.870444  
TBV\_sum\_negative -  
0.207718  
TBV\_mean\_positive -  
1.494796  
TBV\_mean\_negative -  
0.224442  
Income\_sum\_log -  
0.443374  
QuarterlyRating\_sum\_log  
3.246397  
TBV\_mean\_positive\_log -  
0.129703  
TBV\_mean\_negative\_log -  
0.294761  
TBV\_range\_log -  
0.269266  
TBV\_std\_log -  
0.075330

```
TBV_sum_positive_log  
0.268540  
TBV_sum_negative_log  
0.293000
```

```
Simple_Logistic_Regresstion_on_balanced_Dataset \
```

```
Age  
0.221225  
City  
0.223468  
Education_Level  
0.332120  
Gender  
0.386360  
Grade  
0.046176  
Grade_raise  
0.221509  
Income_raise  
0.221509  
Income_range  
0.320109  
Join_month  
0.059304  
Join_year  
1.396947  
Joining_Designation  
0.254144  
QuarterlyRating_last  
0.033144  
QuarterlyRating_mean  
1.490131  
QuarterlyRating_range  
1.210995  
QuarterlyRating_std  
0.699119  
Rating_Promotion  
0.147704  
Reportings  
2.233836  
TBV_raise  
0.542077  
TBV_sum_positive  
2.425888  
TBV_sum_negative  
0.191608  
TBV_mean_positive  
1.336036
```

```
TBV_mean_negative  
0.200984  
Income_sum_log  
0.081919  
QuarterlyRating_sum_log  
3.167117  
TBV_mean_positive_log  
0.123618  
TBV_mean_negative_log  
0.243895  
TBV_range_log  
0.098501  
TBV_std_log  
0.000323  
TBV_sum_positive_log  
0.093199  
TBV_sum_negative_log  
0.243236
```

```
Tuned_Logistic_Regression_on_Imbalanced_Dataset \
```

```
Age  
0.243013  
City  
0.210920  
Education_Level  
0.214511  
Gender  
0.191574  
Grade  
0.232848  
Grade_raise  
0.063252  
Income_raise  
0.063252  
Income_range  
0.850363  
Join_month  
0.078744  
Join_year  
1.427076  
Joining_Designation  
0.170136  
QuarterlyRating_last  
0.271611  
QuarterlyRating_mean  
1.304228  
QuarterlyRating_range  
0.410752
```

QuarterlyRating\_std -  
0.053134  
Rating\_Promotion -  
0.041161  
Reportings -  
2.731432  
TBV\_raise -  
0.476956  
TBV\_sum\_positive -  
1.869077  
TBV\_sum\_negative -  
0.188792  
TBV\_mean\_positive -  
1.345096  
TBV\_mean\_negative -  
0.210537  
Income\_sum\_log -  
0.227009  
QuarterlyRating\_sum\_log -  
3.376148  
TBV\_mean\_positive\_log -  
0.068523  
TBV\_mean\_negative\_log -  
0.285732  
TBV\_range\_log -  
0.193077  
TBV\_std\_log -  
0.264643  
TBV\_sum\_positive\_log -  
0.462417  
TBV\_sum\_negative\_log -  
0.283537

#### Tuned\_Logistic\_Regression\_on\_Balanced\_Dataset

\	
Age	0.202165
City	0.189602
Education_Level	0.298172
Gender	0.330019
Grade	0.053805
Grade_raise	0.181746
Income_raise	0.181746
Income_range	0.342742

Join_month	-0.036528
Join_year	-1.280165
Joining Designation	0.165835
QuarterlyRating_last	0.076717
QuarterlyRating_mean	-1.509647
QuarterlyRating_range	1.227435
QuarterlyRating_std	-0.773373
Rating_Promotion	-0.139816
Reportings	-2.261274
TBV_raise	-0.445578
TBV_sum_positive	-2.323640
TBV_sum_negative	0.171457
TBV_mean_positive	1.336151
TBV_mean_negative	0.184846
Income_sum_log	-0.168530
QuarterlyRating_sum_log	3.274903
TBV_mean_positive_log	0.088966
TBV_mean_negative_log	0.239391
TBV_range_log	0.097356
TBV_std_log	-0.059546
TBV_sum_positive_log	0.116825
TBV_sum_negative_log	0.238220
Tuned_Decision_Tree_on_Imbalanced_Dataset \	
Age	-0.246288
City	-0.246288
Education_Level	-0.246288
Gender	-0.246288

Grade	-0.246288
Grade_raise	-0.246288
Income_raise	-0.246288
Income_range	-0.246288
Join_month	-0.246288
Join_year	-0.246288
Joining Designation	-0.246288
QuarterlyRating_last	-0.246288
QuarterlyRating_mean	-0.246288
QuarterlyRating_range	-0.246288
QuarterlyRating_std	-0.246288
Rating_Promotion	-0.246288
Reportings	1.944072
TBV_raise	4.951987
TBV_sum_positive	-0.246288
TBV_sum_negative	-0.246288
TBV_mean_positive	-0.246288
TBV_mean_negative	-0.246288
Income_sum_log	-0.246288
QuarterlyRating_sum_log	-0.246288
TBV_mean_positive_log	-0.246288
TBV_mean_negative_log	-0.246288
TBV_range_log	-0.246288
TBV_std_log	-0.246288
TBV_sum_positive_log	-0.246288
TBV_sum_negative_log	-0.246288

#### Tuned\_Decimal\_Tree\_on\_Balanced\_Dataset \

Age	-0.331260
City	-0.331260
Education_Level	-0.331260
Gender	-0.331260
Grade	-0.331260
Grade_raise	-0.331260
Income_raise	-0.331260
Income_range	-0.331260
Join_month	-0.331260
Join_year	3.417304
Joining Designation	-0.331260
QuarterlyRating_last	-0.331260
QuarterlyRating_mean	-0.331260
QuarterlyRating_range	-0.331260
QuarterlyRating_std	-0.331260
Rating_Promotion	-0.331260
Reportings	-0.331260
TBV_raise	2.971786
TBV_sum_positive	-0.331260
TBV_sum_negative	-0.331260
TBV_mean_positive	-0.331260

	Tuned_Random_Forest_on_Imbalanced_Dataset \
TBV_mean_negative	-0.331260
Income_sum_log	-0.331260
QuarterlyRating_sum_log	-0.331260
TBV_mean_positive_log	-0.331260
TBV_mean_negative_log	-0.331260
TBV_range_log	-0.331260
TBV_std_log	-0.331260
TBV_sum_positive_log	2.554940
TBV_sum_negative_log	-0.331260
Age	-0.644419
City	-0.649228
Education_Level	-0.649228
Gender	-0.649228
Grade	-0.550416
Grade_raise	-0.646519
Income_raise	-0.648080
Income_range	-0.645063
Join_month	-0.527406
Join_year	2.975579
Joining_Designation	-0.411481
QuarterlyRating_last	1.868872
QuarterlyRating_mean	0.070776
QuarterlyRating_range	-0.579511
QuarterlyRating_std	-0.600907
Rating_Promotion	0.470345
Reportings	1.367708
TBV_raise	2.752090
TBV_sum_positive	0.555378
TBV_sum_negative	-0.649228
TBV_mean_positive	-0.214066
TBV_mean_negative	-0.649228
Income_sum_log	-0.172474
QuarterlyRating_sum_log	0.744959
TBV_mean_positive_log	-0.238492
TBV_mean_negative_log	-0.649228
TBV_range_log	-0.476494
TBV_std_log	-0.539336
TBV_sum_positive_log	0.633551
TBV_sum_negative_log	-0.649228
	Tuned_Random_Forest_on_Balanced_Dataset \
Age	-0.625474
City	-0.636701
Education_Level	-0.636701
Gender	-0.611229
Grade	-0.617031
Grade_raise	-0.635821

Income_raise	-0.631549
Income_range	-0.621397
Join_month	-0.608726
Join_year	2.627866
Joining Designation	-0.536853
QuarterlyRating_last	2.352721
QuarterlyRating_mean	0.297564
QuarterlyRating_range	-0.608740
QuarterlyRating_std	-0.539959
Rating_Promotion	0.149023
Reportings	1.228955
TBV_raise	2.723099
TBV_sum_positive	0.554530
TBV_sum_negative	-0.636701
TBV_mean_positive	-0.242237
TBV_mean_negative	-0.636701
Income_sum_log	-0.069959
QuarterlyRating_sum_log	0.984951
TBV_mean_positive_log	-0.295404
TBV_mean_negative_log	-0.636701
TBV_range_log	-0.538952
TBV_std_log	-0.554295
TBV_sum_positive_log	0.639124
TBV_sum_negative_log	-0.636701
Tuned_Adaboost_on_Imbalanced_Dataset \	
Age	0.638896
City	0.305776
Education_Level	-0.337028
Gender	-0.622056
Grade	-0.633327
Grade_raise	-0.714385
Income_raise	-0.714385
Income_range	-0.711855
Join_month	0.824819
Join_year	3.005240
Joining Designation	-0.319788
QuarterlyRating_last	-0.294315
QuarterlyRating_mean	-0.357083
QuarterlyRating_range	-0.674574
QuarterlyRating_std	0.348790
Rating_Promotion	-0.560452
Reportings	3.301105
TBV_raise	0.475854
TBV_sum_positive	-0.251884
TBV_sum_negative	-0.714385
TBV_mean_positive	-0.387253
TBV_mean_negative	-0.714385
Income_sum_log	1.553575

QuarterlyRating_sum_log	-0.293412
TBV_mean_positive_log	-0.393158
TBV_mean_negative_log	-0.714385
TBV_range_log	0.041205
TBV_std_log	-0.034269
TBV_sum_positive_log	-0.338495
TBV_sum_negative_log	-0.714385

	Tuned_Adaboost_on_Balanced_Dataset \
Age	1.704283
City	2.224529
Education_Level	-0.003743
Gender	-0.337103
Grade	-0.391939
Grade_raise	-0.768513
Income_raise	-0.768513
Income_range	-0.768451
Join_month	1.708928
Join_year	-0.162240
Joining_Designation	-0.445594
QuarterlyRating_last	-0.625184
QuarterlyRating_mean	-0.214007
QuarterlyRating_range	-0.687194
QuarterlyRating_std	-0.111310
Rating_Promotion	-0.560451
Reportings	0.333900
TBV_raise	-0.301866
TBV_sum_positive	-0.308956
TBV_sum_negative	-0.768513
TBV_mean_positive	-0.208055
TBV_mean_negative	-0.768297
Income_sum_log	3.491040
QuarterlyRating_sum_log	0.673354
TBV_mean_positive_log	-0.419614
TBV_mean_negative_log	-0.768513
TBV_range_log	0.096909
TBV_std_log	0.273713
TBV_sum_positive_log	-0.350087
TBV_sum_negative_log	-0.768513

	Tuned_Gradient_boost_on_Imbalanced_Dataset \
Age	-0.479774
City	-0.480545
Education_Level	-0.657126
Gender	-0.671160
Grade	-0.557831
Grade_raise	-0.712690
Income_raise	-0.712690
Income_range	-0.712690

Join_month	-0.039280
Join_year	3.313511
Joining_Designation	-0.205106
QuarterlyRating_last	0.565134
QuarterlyRating_mean	-0.007909
QuarterlyRating_range	-0.622149
QuarterlyRating_std	-0.475052
Rating_Promotion	0.326395
Reportings	0.963075
TBV_raise	3.243523
TBV_sum_positive	0.431504
TBV_sum_negative	-0.712690
TBV_mean_positive	0.060936
TBV_mean_negative	-0.712690
Income_sum_log	0.585024
QuarterlyRating_sum_log	0.677642
TBV_mean_positive_log	-0.285760
TBV_mean_negative_log	-0.712690
TBV_range_log	-0.263812
TBV_std_log	-0.364878
TBV_sum_positive_log	-0.067531
TBV_sum_negative_log	-0.712690

Tuned\_Gradient\_boost\_on\_Balanced\_Dataset \

Age	-0.268031
City	-0.162992
Education_Level	-0.575528
Gender	-0.614484
Grade	-0.594443
Grade_raise	-0.810596
Income_raise	-0.756861
Income_range	-0.645775
Join_month	0.266218
Join_year	3.610337
Joining_Designation	-0.500028
QuarterlyRating_last	0.745653
QuarterlyRating_mean	-0.316865
QuarterlyRating_range	-0.684845
QuarterlyRating_std	-0.418794
Rating_Promotion	-0.257157
Reportings	1.845183
TBV_raise	2.216861
TBV_sum_positive	0.760028
TBV_sum_negative	-0.812250
TBV_mean_positive	-0.187827
TBV_mean_negative	-0.808928
Income_sum_log	0.297264
QuarterlyRating_sum_log	0.356308
TBV_mean_positive_log	-0.150935
TBV_mean_negative_log	-0.812399

TBV_range_log	-0.285556
TBV_std_log	-0.326942
TBV_sum_positive_log	0.698917
TBV_sum_negative_log	-0.805537

	Tuned_XG_boost_on_Imbalanced_Dataset \
Age	-0.364567
City	-0.374230
Education_Level	-0.378478
Gender	-0.391687
Grade	-0.336328
Grade_raise	-0.521812
Income_raise	-0.521812
Income_range	-0.521812
Join_month	-0.141410
Join_year	1.204455
Joining Designation	-0.254610
QuarterlyRating_last	2.402304
QuarterlyRating_mean	-0.238759
QuarterlyRating_range	-0.330187
QuarterlyRating_std	-0.345611
Rating_Promotion	-0.179433
Reportings	0.596011
TBV_raise	4.330003
TBV_sum_positive	-0.199602
TBV_sum_negative	-0.521812
TBV_mean_positive	-0.334488
TBV_mean_negative	-0.521812
Income_sum_log	-0.325689
QuarterlyRating_sum_log	0.319931
TBV_mean_positive_log	-0.315089
TBV_mean_negative_log	-0.521812
TBV_range_log	-0.319478
TBV_std_log	-0.347559
TBV_sum_positive_log	-0.022819
TBV_sum_negative_log	-0.521812

	Tuned_XG_boost_on_Balanced_Dataset \
Age	-0.333042
City	-0.315556
Education_Level	-0.306986
Gender	-0.260038
Grade	-0.311162
Grade_raise	-0.392591
Income_raise	-0.496388
Income_range	-0.416974
Join_month	-0.233752
Join_year	1.391542
Joining Designation	-0.325474

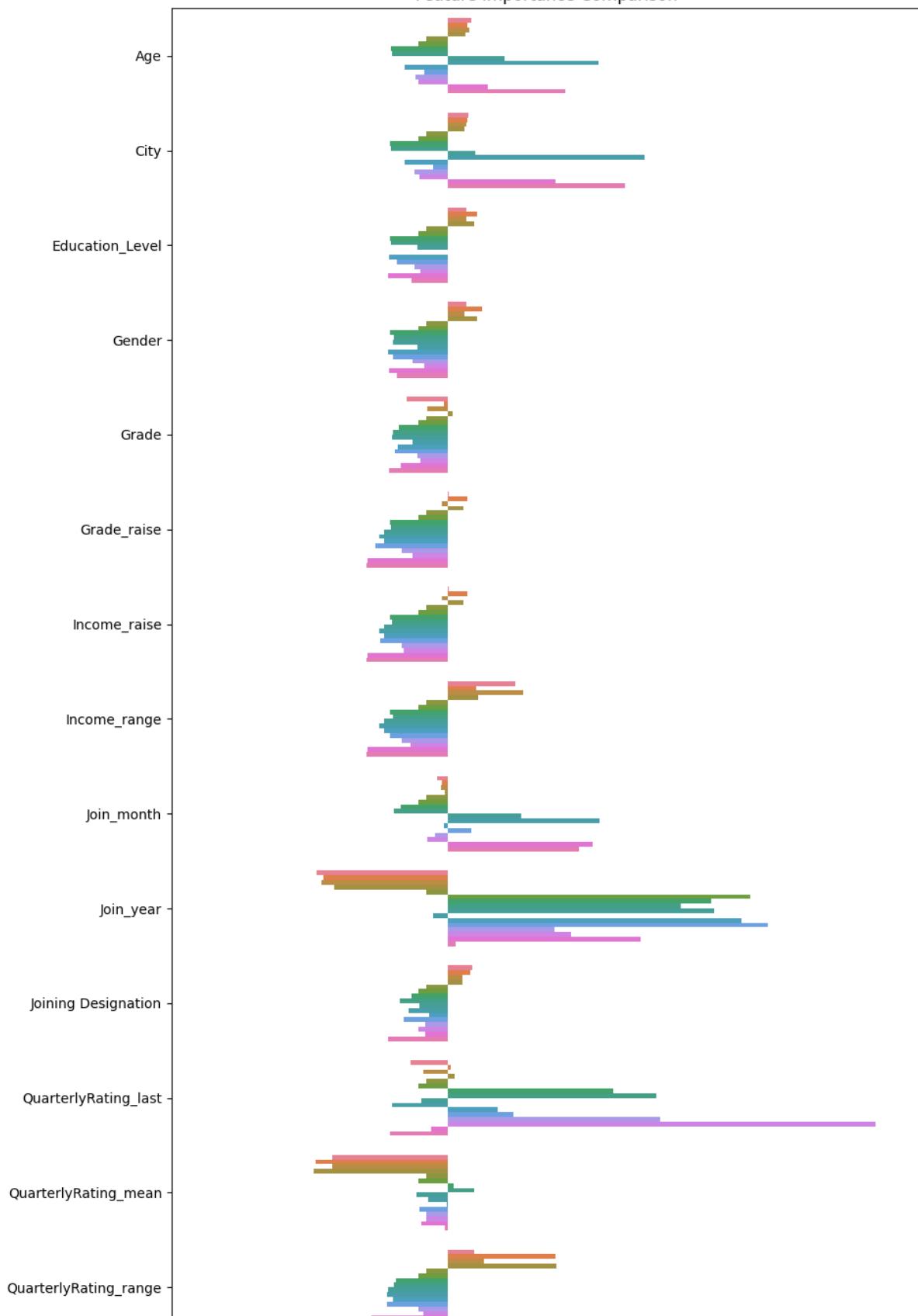
QuarterlyRating_last	4.830718
QuarterlyRating_mean	-0.238751
QuarterlyRating_range	-0.272864
QuarterlyRating_std	-0.270314
Rating_Promotion	-0.079987
Reportings	0.217141
TBV_raise	1.375093
TBV_sum_positive	-0.116697
TBV_sum_negative	-0.496388
TBV_mean_positive	-0.242375
TBV_mean_negative	-0.496388
Income_sum_log	-0.289260
QuarterlyRating_sum_log	0.019957
TBV_mean_positive_log	-0.195481
TBV_mean_negative_log	-0.496388
TBV_range_log	-0.283821
TBV_std_log	-0.281868
TBV_sum_positive_log	-0.185517
TBV_sum_negative_log	-0.496388
Tuned_Light_GBM_on_Imbalanced_Dataset \	
Age	0.452439
City	1.221456
Education_Level	-0.672250
Gender	-0.662637
Grade	-0.528059
Grade_raise	-0.902955
Income_raise	-0.902955
Income_range	-0.902955
Join_month	1.634803
Join_year	2.173116
Joining_Designation	-0.249290
QuarterlyRating_last	-0.182001
QuarterlyRating_mean	-0.297353
QuarterlyRating_range	-0.854891
QuarterlyRating_std	-0.047423
Rating_Promotion	-0.653024
Reportings	2.672977
TBV_raise	0.856173
TBV_sum_positive	-0.028197
TBV_sum_negative	-0.902955
TBV_mean_positive	-0.364643
TBV_mean_negative	-0.902955
Income_sum_log	1.625191
QuarterlyRating_sum_log	1.259907
TBV_mean_positive_log	-0.489608
TBV_mean_negative_log	-0.902955
TBV_range_log	0.087155
TBV_std_log	-0.316579

TBV_sum_positive_log	-0.316579
TBV_sum_negative_log	-0.902955
	Tuned_Light_GBM_on_Balanced_Dataset
Age	1.324696
City	1.995708
Education_Level	-0.405110
Gender	-0.572388
Grade	-0.659829
Grade_raise	-0.920250
Income_raise	-0.920250
Income_range	-0.920250
Join_month	1.484370
Join_year	0.093872
Joining Designation	-0.669334
QuarterlyRating_last	-0.645572
QuarterlyRating_mean	-0.029685
QuarterlyRating_range	-0.810950
QuarterlyRating_std	0.419874
Rating_Promotion	-0.578091
Reportings	0.658435
TBV_raise	-0.279652
TBV_sum_positive	0.208876
TBV_sum_negative	-0.920250
TBV_mean_positive	0.024490
TBV_mean_negative	-0.920250
Income_sum_log	3.201820
QuarterlyRating_sum_log	1.461559
TBV_mean_positive_log	-0.192211
TBV_mean_negative_log	-0.920250
TBV_range_log	0.345739
TBV_std_log	0.204123
TBV_sum_positive_log	-0.138986
TBV_sum_negative_log	-0.920250

## Feature\_importance\_df\_plots

```
feature_importance_plots(feature_importance_scaled)
```

Feature Importance Comparison



sum of all the feature\_importances and normalizing

```
combined_feature_importance_scaled =
feature_importance_scaled.sum(axis = 1)

combined_feature_importance_scaled =
pd.DataFrame(combined_feature_importance_scaled,index=combined_feature
_importance_scaled.index,columns = ["combined_feature_importances"])

combined_feature_importance_scaled
```

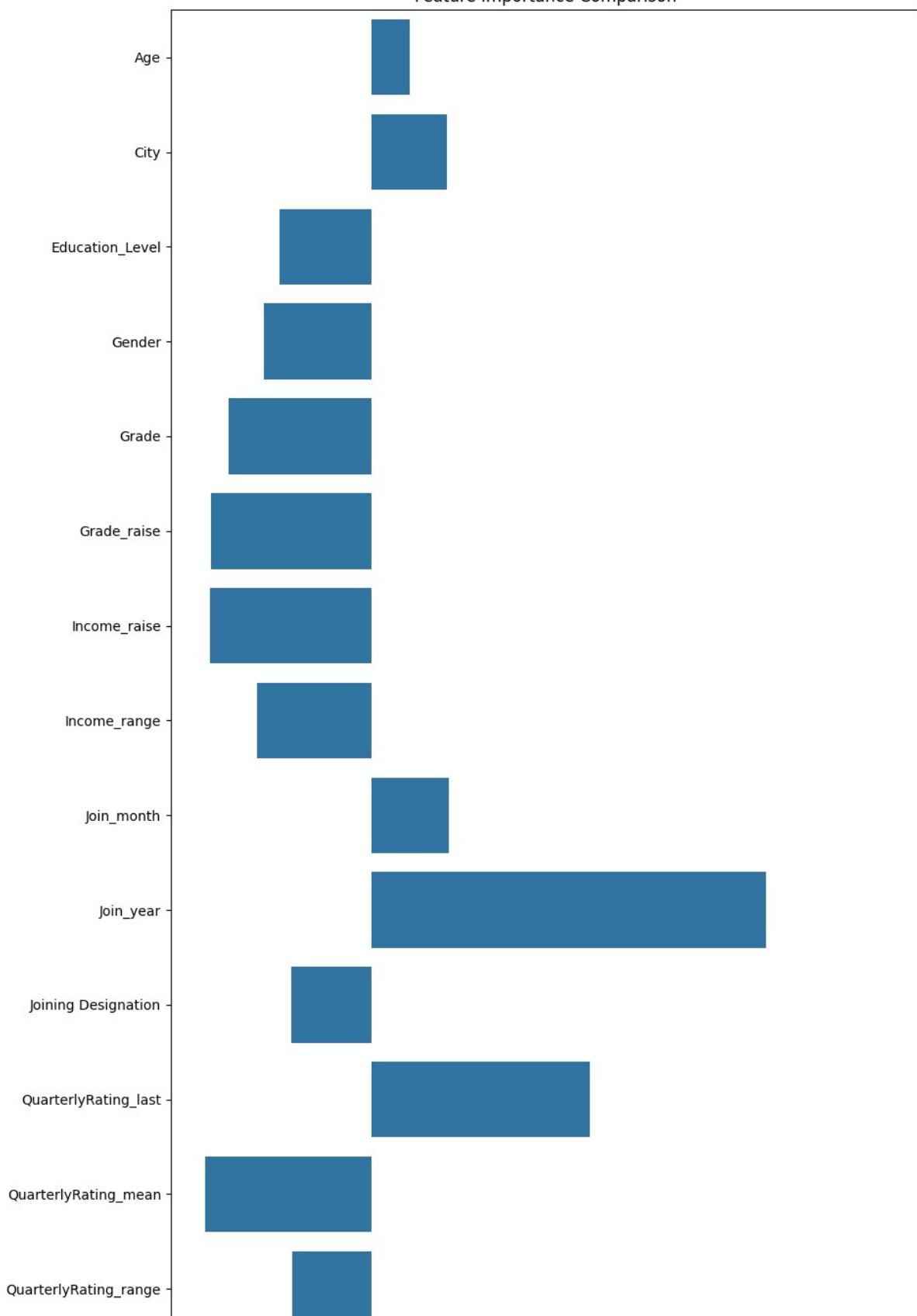
	combined_feature_importances
Age	1.759584
City	3.407079
Education_Level	-4.148663
Gender	-4.846839
Grade	-6.445801
Grade_raise	-7.252526
Income_raise	-7.299877
Income_range	-5.163849
Join_month	3.492457
Join_year	17.824404
Joining_Designation	-3.622382
QuarterlyRating_last	9.864255
QuarterlyRating_mean	-7.518275
QuarterlyRating_range	-3.559694
QuarterlyRating_std	-4.023275
Rating_Promotion	-2.692855
Reportings	4.841360
TBV_raise	23.233855
TBV_sum_positive	-7.461618
TBV_sum_negative	-6.953147
TBV_mean_positive	2.839014
TBV_mean_negative	-6.888373
Income_sum_log	9.738919
QuarterlyRating_sum_log	18.692173
TBV_mean_positive_log	-3.538943
TBV_mean_negative_log	-6.649092
TBV_range_log	-1.516452
TBV_std_log	-3.264633
TBV_sum_positive_log	3.801209
TBV_sum_negative_log	-6.648015

```
scaler = StandardScaler()
combined_feature_importance_scaled =
pd.DataFrame(scaler.fit_transform(combined_feature_importance_scaled),
index=combined_feature_importance_scaled.index,
columns =
combined_feature_importance_scaled.columns)
combined_feature_importance_scaled
```

	combined_feature_importances
Age	0.212454
City	0.411374
Education_Level	-0.500914
Gender	-0.585213
Grade	-0.778273
Grade_raise	-0.875678
Income_raise	-0.881395
Income_range	-0.623489
Join_month	0.421683
Join_year	2.152138
Joining_Designation	-0.437370
QuarterlyRating_last	1.191021
QuarterlyRating_mean	-0.907765
QuarterlyRating_range	-0.429801
QuarterlyRating_std	-0.485775
Rating_Promotion	-0.325138
Reportings	0.584551
TBV_raise	2.805281
TBV_sum_positive	-0.900924
TBV_sum_negative	-0.839530
TBV_mean_positive	0.342786
TBV_mean_negative	-0.831710
Income_sum_log	1.175887
QuarterlyRating_sum_log	2.256913
TBV_mean_positive_log	-0.427296
TBV_mean_negative_log	-0.802818
TBV_range_log	-0.183098
TBV_std_log	-0.394175
TBV_sum_positive_log	0.458962
TBV_sum_negative_log	-0.802688

```
feature_importance_plots(combined_feature_importance_scaled)
```

Feature Importance Comparison



	combined_feature_importances
Grade_raise	-0.875678
Income_raise	-0.881395
Join_year	2.152138
QuarterlyRating_last	1.191021
QuarterlyRating_mean	-0.907765
TBV_raise	2.805281
TBV_sum_positive	-0.900924
TBV_sum_negative	-0.839530
TBV_mean_negative	-0.831710
Income_sum_log	1.175887
QuarterlyRating_sum_log	2.256913
TBV_mean_negative_log	-0.802818
TBV_sum_negative_log	-0.802688

## Observation

### Most Important Features:

- QuarterlyRating\_sum\_log: Another critical feature, suggesting that the cumulative log of quarterly ratings significantly impacts the outcome.
- TBV\_raise: This feature also shows high importance, meaning that changes or raises in Total Business Value (TBV) are crucial.
- Join\_year: The year of joining appears to be a highly relevant factor, potentially affecting performance or outcomes over time.

### Moderately Important Features:

- Income\_sum\_log: The log-transformed sum of income is moderately important, indicating that income trends play a role.
- TBV\_sum\_positive\_log: This suggests that positive TBV values, when log-transformed, also have a moderate impact.
- QuarterlyRating\_last, QuarterlyRating\_mean: Both the last and mean quarterly ratings are moderately important, highlighting the significance of periodic performance evaluations.

### Less Important Features:

- Age, City: These demographic features have relatively lower importance, indicating they have less influence on the target variable.
- Gender, Education\_Level: These features also show lower importance, suggesting that gender and education level may not be as critical in this context.
- Various other TBV and Rating features: Some TBV and rating-related features (like TBV\_mean\_positive\_log, TBV\_mean\_negative, TBV\_std\_log) have lesser importance, but they still contribute to the model.

### Negative Importance Values:

- Some features have negative importance values, such as QuarterlyRating\_std, TBV\_sum\_positive, TBV\_mean\_negative\_log. This might indicate that these features

negatively correlate with the target. So Increase in These features makes the driver churn off

## PLOT ALL LEARNING CURVES

```
def get_experiment_id(experiment_name):
    experiment = mlflow.get_experiment_by_name(experiment_name)
    if experiment:
        return experiment.experiment_id
    else:
        print(f"Experiment '{experiment_name}' not found.")
        return None

def get_run_ids(experiment_id):
    client = mlflow.tracking.MlflowClient()
    runs = client.search_runs(experiment_id)
    return [run.info.run_id for run in runs]

# Example usage
experiment_id = get_experiment_id("Ola_Ensemble_Learning")
run_ids = get_run_ids(experiment_id)

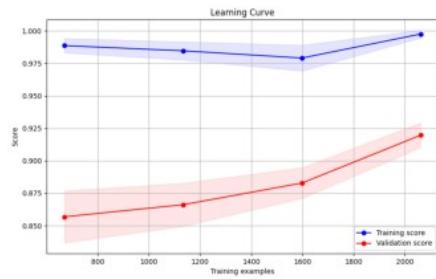
def plot_all_learning_curves(experiment_id, run_ids):
    plt.figure(figsize=(10, 40))
    plot_count = 1

    for run_id in run_ids:
        run_path = f"mlruns/{experiment_id}/{run_id}/artifacts/"
        for file in os.listdir(run_path):
            if file.endswith("_learning_curve.png"):
                img = plt.imread(os.path.join(run_path, file))
                plt.subplot(len(run_ids)//2, 2, plot_count)
                plt.imshow(img)
                plt.axis('off')
                plt.title(f"{file.replace('_learning_curve.png', '')}")
        plot_count += 1

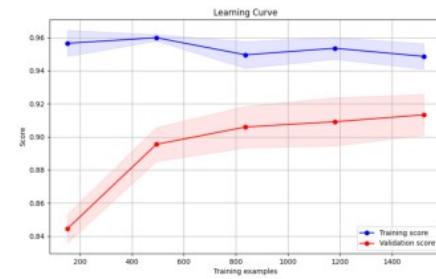
    plt.tight_layout()
    plt.show()

# Example usage
plot_all_learning_curves(experiment_id, run_ids)
```

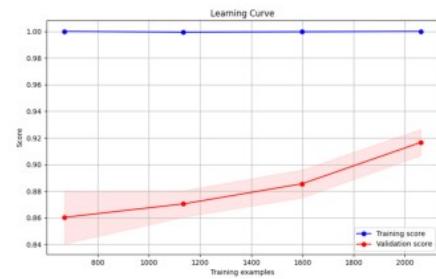
Tuned\_Stacking\_Classifier\_on\_Balanced\_Dataset



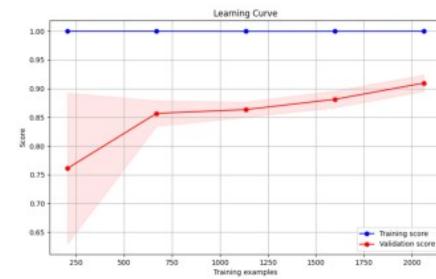
Tuned\_Stacking\_Classifier\_on\_Imbalanced\_Dataset



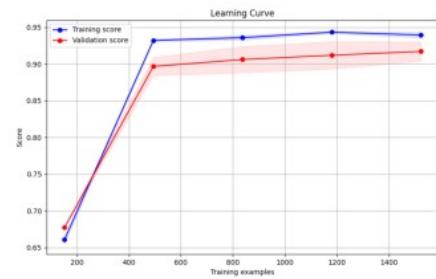
Tuned\_Voting\_Classifier\_on\_Balanced\_Dataset



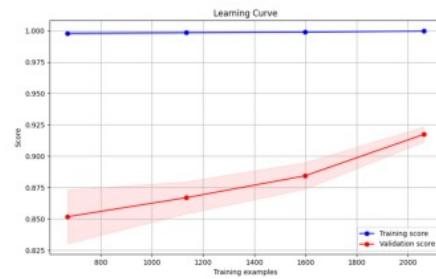
Tuned\_Light\_GBM\_on\_Balanced\_Dataset



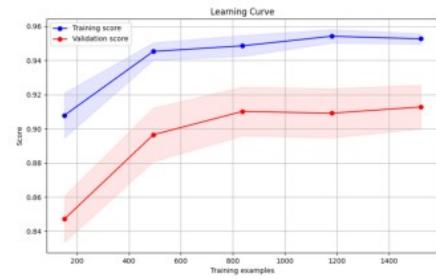
Tuned\_Light\_GBM\_on\_Imbalanced\_Dataset



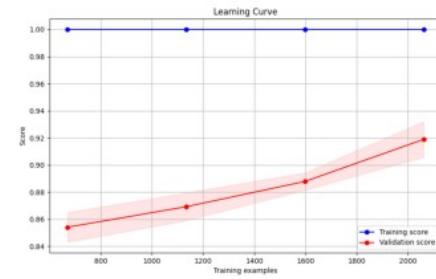
Tuned\_XG\_boost\_on\_Balanced\_Dataset



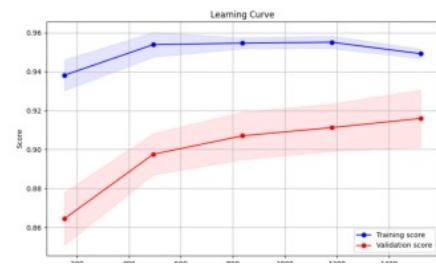
Tuned\_XG\_boost\_on\_Imbalanced\_Dataset



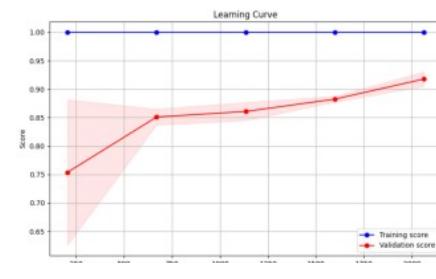
Tuned\_Gradient\_boost\_on\_Balanced\_Dataset



Tuned\_Gradient\_boost\_on\_Imbalanced\_Dataset



Tuned\_Adaboost\_on\_Balanced\_Dataset



## Observation

Stacking classifier has Good Training scores and Validation scores are steadily improving with increase in training examples. Both training and test score crossed 90%. So increase in dataset results in better accuracy by using stacking classifier

Voting Classifier for Imbalanced dataset was not logged

LightGBM has Best generalization in case of Imbalanced Dataset

All the Boosting models have very less scores for less training samples, But Drastically improved with number of training samples. So increase in dataset definitely improves better generalization in Boosting with less training and testing times

Bagging and Random forest are not near 100 %. So to improve the accuracy in Bagging or RF, We need very big datasets. But generalization is very good in these techniques.

All Basic models, Logistic Regression, MLPClassifier, DT cannot provide better accuracy but provide better generalisation.

KNN and SVC suffers in both accuracy and generalization

## PLOT ALL AUC PLOTS

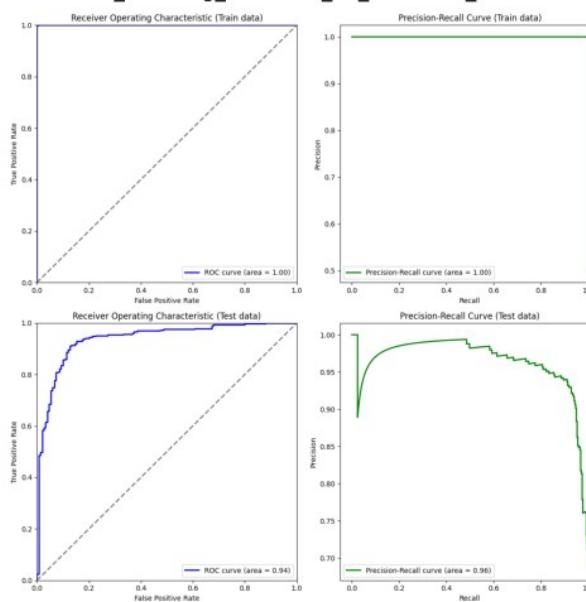
```
def plot_all_roc_auc_plots(experiment_id, run_ids):
    plt.figure(figsize=(10, 80))
    plot_count = 1

    for run_id in run_ids:
        run_path = f"mlruns/{experiment_id}/{run_id}/artifacts/"
        for file in os.listdir(run_path):
            if file.endswith("_auc_plots.png"):
                img = plt.imread(os.path.join(run_path, file))
                plt.subplot(len(run_ids)//2, 2, plot_count)
                plt.imshow(img)
                plt.axis('off')
                plt.title(f"{file.replace('_auc_plots.png', '')}")
                plot_count += 1

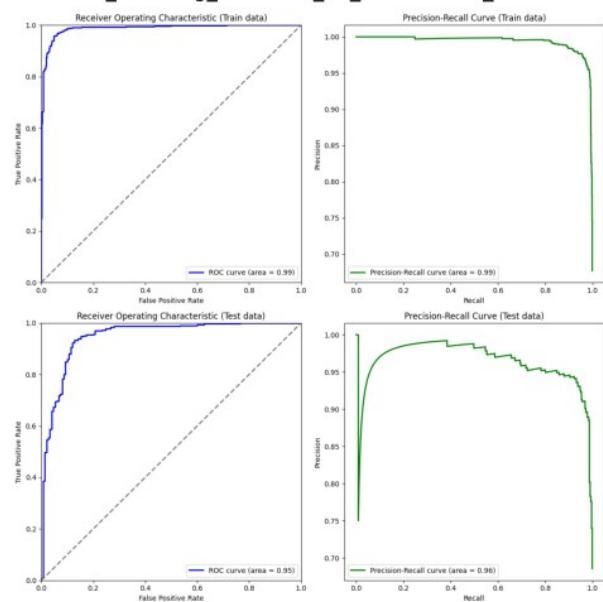
    plt.tight_layout()
    plt.show()

# Example usage
plot_all_roc_auc_plots(experiment_id, run_ids)
```

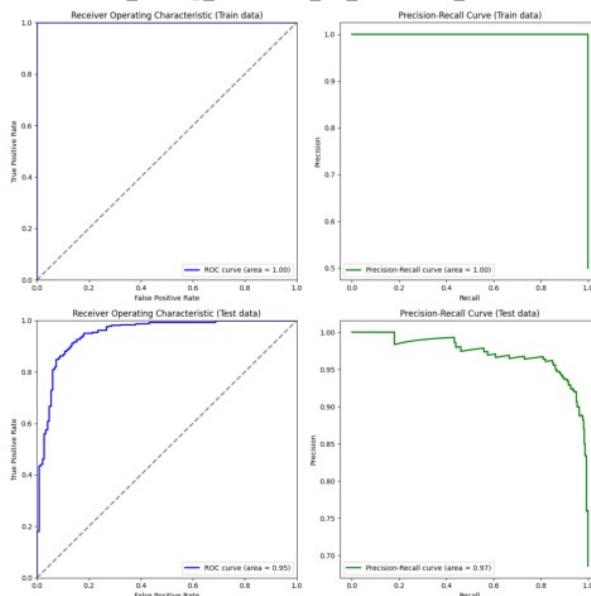
Tuned\_Stacking\_Classifier\_on\_Balanced\_Dataset



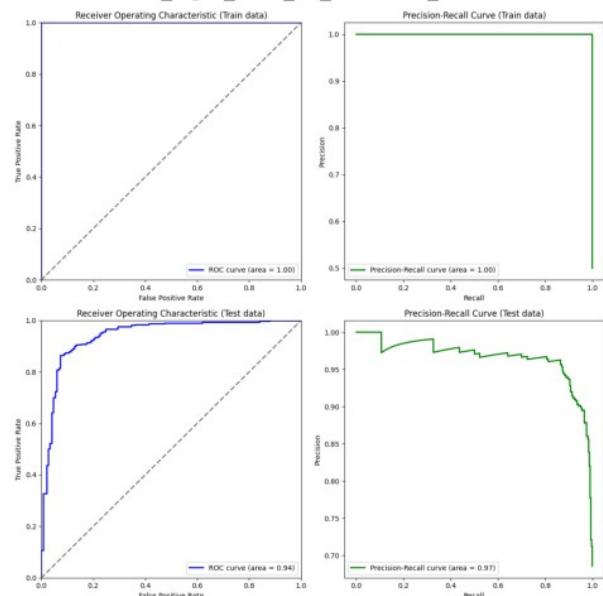
Tuned\_Stacking\_Classifier\_on\_Imbalanced\_Dataset



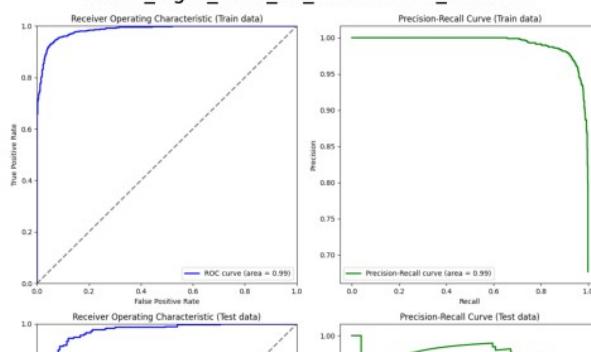
Tuned\_Voting\_Classifier\_on\_Balanced\_Dataset



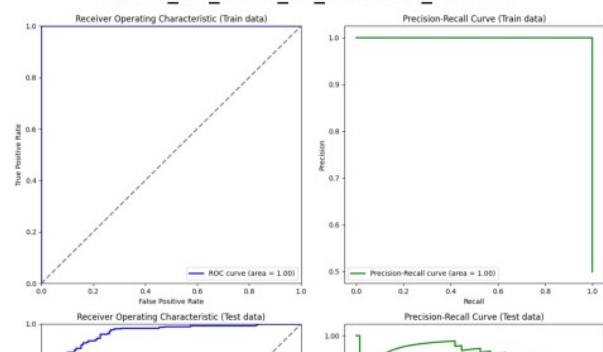
Tuned\_Light\_GBM\_on\_Balanced\_Dataset



Tuned\_Light\_GBM\_on\_Imbalanced\_Dataset



Tuned\_XG\_boost\_on\_Balanced\_Dataset



## Observation

PR\_AUC\_test achieved is 0.97 on Boosting , Voting and Stacking Classifiers

AU\_ROC test achieved is 0.96 on Adaboost, Gradient\_Boost, XGBoost and LightGBM models

PR\_AUC is used for comparing imbalanced datasets

Au\_roc is used for comparing balanced datasets

PR\_AUC reached 1 for Stacking, Voting, LightGBM, XGBOOST, Adaboost for balanced training dataset

## PRINT ALL CLASSIFICATION REPORTS

```
def print_all_classification_reports(experiment_id, run_ids):
    for run_id in run_ids:
        run_path = f"mlruns/{experiment_id}/{run_id}/artifacts/"
        for file in os.listdir(run_path):
            if file.endswith("_classification_report.csv"):
                report_df = pd.read_csv(os.path.join(run_path, file),
index_col=0)
                print(f"\n{file.replace('_classification_report.csv',
'')}:\n")
                print(report_df)
                print("\n" + "-"*80 + "\n")

# Example usage
print_all_classification_reports(experiment_id, run_ids)
```

### Two\_Level\_Cascading\_Classifier\_on\_Balanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.831169	0.853333	0.842105	150.000000
1	0.931889	0.920489	0.926154	327.000000
accuracy	0.899371	0.899371	0.899371	0.899371
macro avg	0.881529	0.886911	0.884130	477.000000
weighted avg	0.900216	0.899371	0.899723	477.000000

---

---

### Two\_Level\_Cascading\_Classifier\_on\_Balanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.999224	0.999224	0.999224	1289.000000
1	0.999224	0.999224	0.999224	1289.000000
accuracy	0.999224	0.999224	0.999224	0.999224
macro avg	0.999224	0.999224	0.999224	2578.000000
weighted avg	0.999224	0.999224	0.999224	2578.000000

Two\_Level\_Cascading\_Classifier\_on\_Imbalanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.885714	0.826667	0.855172	150.00000
1	0.922849	0.951070	0.936747	327.00000
accuracy	0.911950	0.911950	0.911950	0.91195
macro avg	0.904281	0.888869	0.895960	477.00000
weighted avg	0.911171	0.911950	0.911095	477.00000

Two\_Level\_Cascading\_Classifier\_on\_Imbalanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.954082	0.912195	0.932668	615.000000
1	0.958967	0.979054	0.968906	1289.000000
accuracy	0.957458	0.957458	0.957458	0.957458
macro avg	0.956524	0.945624	0.950787	1904.000000
weighted avg	0.957389	0.957458	0.957201	1904.000000

Tuned\_Stacking\_Classifier\_on\_Balanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.831169	0.853333	0.842105	150.000000
1	0.931889	0.920489	0.926154	327.000000
accuracy	0.899371	0.899371	0.899371	0.899371
macro avg	0.881529	0.886911	0.884130	477.000000
weighted avg	0.900216	0.899371	0.899723	477.000000

Tuned\_Stacking\_Classifier\_on\_Balanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.999224	0.999224	0.999224	1289.000000
1	0.999224	0.999224	0.999224	1289.000000
accuracy	0.999224	0.999224	0.999224	0.999224
macro avg	0.999224	0.999224	0.999224	2578.000000

```
weighted avg    0.999224  0.999224  0.999224  2578.000000
```

```
Tuned_Stacking_Classifier_on_Imbalanced_Dataset_test:
```

	precision	recall	f1-score	support
0	0.885714	0.826667	0.855172	150.000000
1	0.922849	0.951070	0.936747	327.000000
accuracy	0.911950	0.911950	0.911950	0.91195
macro avg	0.904281	0.888869	0.895960	477.00000
weighted avg	0.911171	0.911950	0.911095	477.000000

```
Tuned_Stacking_Classifier_on_Imbalanced_Dataset_train:
```

	precision	recall	f1-score	support
0	0.954082	0.912195	0.932668	615.000000
1	0.958967	0.979054	0.968906	1289.000000
accuracy	0.957458	0.957458	0.957458	0.957458
macro avg	0.956524	0.945624	0.950787	1904.000000
weighted avg	0.957389	0.957458	0.957201	1904.000000

```
Tuned_Voting_Classifier_on_Balanced_Dataset_test:
```

	precision	recall	f1-score	support
0	0.826923	0.860000	0.843137	150.000000
1	0.934579	0.917431	0.925926	327.000000
accuracy	0.899371	0.899371	0.899371	0.899371
macro avg	0.880751	0.888716	0.884532	477.000000
weighted avg	0.900725	0.899371	0.899892	477.000000

```
Tuned_Voting_Classifier_on_Balanced_Dataset_train:
```

	precision	recall	f1-score	support
0	1.0	1.0	1.0	1289.0
1	1.0	1.0	1.0	1289.0
accuracy	1.0	1.0	1.0	1.0

macro avg	1.0	1.0	1.0	2578.0
weighted avg	1.0	1.0	1.0	2578.0

---

Tuned\_Voting\_Classifier\_on\_Imbalanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.904412	0.820000	0.860140	150.000000
1	0.920821	0.960245	0.940120	327.000000
accuracy	0.916143	0.916143	0.916143	0.916143
macro avg	0.912616	0.890122	0.900130	477.000000
weighted avg	0.915661	0.916143	0.914969	477.000000

---

Tuned\_Voting\_Classifier\_on\_Imbalanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.931507	0.884553	0.907423	615.000000
1	0.946212	0.968968	0.957455	1289.000000
accuracy	0.941702	0.941702	0.941702	0.941702
macro avg	0.938859	0.926761	0.932439	1904.000000
weighted avg	0.941462	0.941702	0.941294	1904.000000

---

Tuned\_Light\_GBM\_on\_Balanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.807692	0.840000	0.823529	150.000000
1	0.925234	0.908257	0.916667	327.000000
accuracy	0.886792	0.886792	0.886792	0.886792
macro avg	0.866463	0.874128	0.870098	477.000000
weighted avg	0.888271	0.886792	0.887378	477.000000

---

Tuned\_Light\_GBM\_on\_Balanced\_Dataset\_train:

	precision	recall	f1-score	support
0	1.0	1.0	1.0	1289.0
1	1.0	1.0	1.0	1289.0

accuracy	1.0	1.0	1.0	1.0
macro avg	1.0	1.0	1.0	2578.0
weighted avg	1.0	1.0	1.0	2578.0

-----  
-----

Tuned\_Light\_GBM\_on\_Imbalanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.898551	0.826667	0.861111	150.000000
1	0.923304	0.957187	0.939940	327.000000
accuracy	0.916143	0.916143	0.916143	0.916143
macro avg	0.910927	0.891927	0.900526	477.000000
weighted avg	0.915520	0.916143	0.915151	477.000000

-----  
-----

Tuned\_Light\_GBM\_on\_Imbalanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.924915	0.881301	0.902581	615.000000
1	0.944613	0.965865	0.955121	1289.000000
accuracy	0.938550	0.938550	0.938550	0.93855
macro avg	0.934764	0.923583	0.928851	1904.00000
weighted avg	0.938250	0.938550	0.938150	1904.00000

-----  
-----

Tuned\_XG\_boost\_on\_Balanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.824675	0.846667	0.835526	150.000000
1	0.928793	0.917431	0.923077	327.000000
accuracy	0.895178	0.895178	0.895178	0.895178
macro avg	0.876734	0.882049	0.879302	477.000000
weighted avg	0.896051	0.895178	0.895545	477.000000

-----  
-----

Tuned\_XG\_boost\_on\_Balanced\_Dataset\_train:

	precision	recall	f1-score	support
0	1.000000	0.999224	0.999612	1289.000000

1	0.999225	1.000000	0.999612	1289.000000
accuracy	0.999612	0.999612	0.999612	0.999612
macro avg	0.999612	0.999612	0.999612	2578.000000
weighted avg	0.999612	0.999612	0.999612	2578.000000

---

---

Tuned\_XG\_boost\_on\_Imbalanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.908397	0.793333	0.846975	150.000000
1	0.910405	0.963303	0.936107	327.000000
accuracy	0.909853	0.909853	0.909853	0.909853
macro avg	0.909401	0.878318	0.891541	477.000000
weighted avg	0.909773	0.909853	0.908078	477.000000

---

---

Tuned\_XG\_boost\_on\_Imbalanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.969912	0.891057	0.928814	615.000000
1	0.949963	0.986811	0.968037	1289.000000
accuracy	0.955882	0.955882	0.955882	0.955882
macro avg	0.959937	0.938934	0.948425	1904.000000
weighted avg	0.956406	0.955882	0.955367	1904.000000

---

---

Tuned\_Gradient\_boost\_on\_Balanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.808917	0.846667	0.827362	150.000000
1	0.928125	0.908257	0.918083	327.000000
accuracy	0.888889	0.888889	0.888889	0.888889
macro avg	0.868521	0.877462	0.872723	477.000000
weighted avg	0.890638	0.888889	0.889555	477.000000

---

---

Tuned\_Gradient\_boost\_on\_Balanced\_Dataset\_train:

	precision	recall	f1-score	support
--	-----------	--------	----------	---------

0	1.0	1.0	1.0	1289.0
1	1.0	1.0	1.0	1289.0
accuracy	1.0	1.0	1.0	1.0
macro avg	1.0	1.0	1.0	2578.0
weighted avg	1.0	1.0	1.0	2578.0

-----  
-----

Tuned\_Gradient\_boost\_on\_Imbalanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.905797	0.833333	0.868056	150.000000
1	0.926254	0.960245	0.942943	327.000000
accuracy	0.920335	0.920335	0.920335	0.920335
macro avg	0.916025	0.896789	0.905499	477.000000
weighted avg	0.919821	0.920335	0.919393	477.000000

-----  
-----

Tuned\_Gradient\_boost\_on\_Imbalanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.942078	0.899187	0.920133	615.00000
1	0.952923	0.973623	0.963162	1289.00000
accuracy	0.949580	0.949580	0.949580	0.94958
macro avg	0.947501	0.936405	0.941648	1904.00000
weighted avg	0.949420	0.949580	0.949263	1904.00000

-----  
-----

Tuned\_Adaboost\_on\_Balanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.827815	0.833333	0.830565	150.000000
1	0.923313	0.920489	0.921899	327.000000
accuracy	0.893082	0.893082	0.893082	0.893082
macro avg	0.875564	0.876911	0.876232	477.000000
weighted avg	0.893282	0.893082	0.893177	477.000000

-----  
-----

Tuned\_Adaboost\_on\_Balanced\_Dataset\_train:

	precision	recall	f1-score	support
0	1.0	1.0	1.0	1289.0
1	1.0	1.0	1.0	1289.0
accuracy	1.0	1.0	1.0	1.0
macro avg	1.0	1.0	1.0	2578.0
weighted avg	1.0	1.0	1.0	2578.0

Tuned\_Adaboost\_on\_Imbalanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.871429	0.813333	0.841379	150.000000
1	0.916914	0.944954	0.930723	327.000000
accuracy	0.903564	0.903564	0.903564	0.903564
macro avg	0.894171	0.879144	0.886051	477.000000
weighted avg	0.902610	0.903564	0.902627	477.000000

Tuned\_Adaboost\_on\_Imbalanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.977929	0.936585	0.956811	615.000000
1	0.970342	0.989915	0.980031	1289.000000
accuracy	0.972689	0.972689	0.972689	0.972689
macro avg	0.974135	0.963250	0.968421	1904.000000
weighted avg	0.972793	0.972689	0.972531	1904.000000

Tuned\_Bagging\_RF\_on\_Balanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.836601	0.853333	0.844884	150.000000
1	0.932099	0.923547	0.927803	327.000000
accuracy	0.901468	0.901468	0.901468	0.901468
macro avg	0.884350	0.888440	0.886344	477.000000
weighted avg	0.902068	0.901468	0.901728	477.000000

Tuned\_Bagging\_RF\_on\_Balanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.930385	0.881303	0.905179	1289.00000
1	0.887251	0.934057	0.910053	1289.00000
accuracy	0.907680	0.907680	0.907680	0.90768
macro avg	0.908818	0.907680	0.907616	2578.00000
weighted avg	0.908818	0.907680	0.907616	2578.00000

-----  
-----

Tuned\_Bagging\_RF\_on\_Imbalanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.873239	0.826667	0.849315	150.000000
1	0.922388	0.944954	0.933535	327.000000
accuracy	0.907757	0.907757	0.907757	0.907757
macro avg	0.897814	0.885810	0.891425	477.000000
weighted avg	0.906933	0.907757	0.907051	477.000000

-----  
-----

Tuned\_Bagging\_RF\_on\_Imbalanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.875614	0.869919	0.872757	615.000000
1	0.938128	0.941040	0.939582	1289.000000
accuracy	0.918067	0.918067	0.918067	0.918067
macro avg	0.906871	0.905479	0.906169	1904.000000
weighted avg	0.917936	0.918067	0.917997	1904.000000

-----  
-----

Tuned\_Random\_Forest\_on\_Balanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.807947	0.813333	0.810631	150.000000
1	0.914110	0.911315	0.912711	327.000000
accuracy	0.880503	0.880503	0.880503	0.880503
macro avg	0.861029	0.862324	0.861671	477.000000
weighted avg	0.880726	0.880503	0.880610	477.000000

-----  
-----

Tuned\_Random\_Forest\_on\_Balanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.909965	0.807603	0.855734	1289.000000
1	0.827057	0.920093	0.871098	1289.000000
accuracy	0.863848	0.863848	0.863848	0.863848
macro avg	0.868511	0.863848	0.863416	2578.000000
weighted avg	0.868511	0.863848	0.863416	2578.000000

-----  
-----

Tuned\_Random\_Forest\_on\_Imbalanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.806452	0.833333	0.819672	150.000000
1	0.922360	0.908257	0.915254	327.000000
accuracy	0.884696	0.884696	0.884696	0.884696
macro avg	0.864406	0.870795	0.867463	477.000000
weighted avg	0.885911	0.884696	0.885197	477.000000

-----  
-----

Tuned\_Random\_Forest\_on\_Imbalanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.828800	0.842276	0.835484	615.000000
1	0.924159	0.916990	0.920561	1289.000000
accuracy	0.892857	0.892857	0.892857	0.892857
macro avg	0.876480	0.879633	0.878022	1904.000000
weighted avg	0.893358	0.892857	0.893081	1904.000000

-----  
-----

Tuned\_Decision\_Tree\_on\_Balanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.562249	0.933333	0.701754	150.000000
1	0.956140	0.666667	0.785586	327.000000
accuracy	0.750524	0.750524	0.750524	0.750524
macro avg	0.759195	0.800000	0.743670	477.000000
weighted avg	0.832275	0.750524	0.759224	477.000000

-----  
-----

Tuned\_Decision\_Tree\_on\_Balanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.747726	0.956555	0.839346	1289.000000
1	0.939720	0.677269	0.787196	1289.000000
accuracy	0.816912	0.816912	0.816912	0.816912
macro avg	0.843723	0.816912	0.813271	2578.000000
weighted avg	0.843723	0.816912	0.813271	2578.000000

Tuned\_Decision\_Tree\_on\_Imbalanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.864407	0.680000	0.761194	150.000000
1	0.866295	0.951070	0.906706	327.000000
accuracy	0.865828	0.865828	0.865828	0.865828
macro avg	0.865351	0.815535	0.833950	477.000000
weighted avg	0.865701	0.865828	0.860947	477.000000

Tuned\_Decision\_Tree\_on\_Imbalanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.806513	0.684553	0.740545	615.000000
1	0.859624	0.921645	0.889554	1289.000000
accuracy	0.845063	0.845063	0.845063	0.845063
macro avg	0.833069	0.803099	0.815050	1904.000000
weighted avg	0.842469	0.845063	0.841424	1904.000000

Tuned\_SVC\_on\_Balanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.733766	0.753333	0.743421	150.000000
1	0.885449	0.874618	0.880000	327.000000
accuracy	0.836478	0.836478	0.836478	0.836478
macro avg	0.809608	0.813976	0.811711	477.000000
weighted avg	0.837750	0.836478	0.837051	477.000000

Tuned\_SVC\_on\_Balanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.969207	0.976726	0.972952	1289.000000
1	0.976544	0.968968	0.972741	1289.000000
accuracy	0.972847	0.972847	0.972847	0.972847
macro avg	0.972876	0.972847	0.972847	2578.000000
weighted avg	0.972876	0.972847	0.972847	2578.000000

Tuned\_SVC\_on\_Imbalanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.791411	0.860000	0.824281	150.000000
1	0.933121	0.896024	0.914197	327.000000
accuracy	0.884696	0.884696	0.884696	0.884696
macro avg	0.862266	0.878012	0.869239	477.000000
weighted avg	0.888558	0.884696	0.885921	477.000000

Tuned\_SVC\_on\_Imbalanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.864382	0.943089	0.902022	615.000000
1	0.971614	0.929403	0.950040	1289.000000
accuracy	0.933824	0.933824	0.933824	0.933824
macro avg	0.917998	0.936246	0.926031	1904.000000
weighted avg	0.936977	0.933824	0.934530	1904.000000

Tuned\_KNN\_on\_Balanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.681319	0.826667	0.746988	150.000000
1	0.911864	0.822630	0.864952	327.000000
accuracy	0.823899	0.823899	0.823899	0.823899
macro avg	0.796592	0.824648	0.805970	477.000000
weighted avg	0.839366	0.823899	0.827856	477.000000

Tuned\_KNN\_on\_Balanced\_Dataset\_train:

	precision	recall	f1-score	support
0	1.0	1.0	1.0	1289.0
1	1.0	1.0	1.0	1289.0
accuracy	1.0	1.0	1.0	1.0
macro avg	1.0	1.0	1.0	2578.0
weighted avg	1.0	1.0	1.0	2578.0

Tuned\_KNN\_on\_Imbalanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.849206	0.713333	0.775362	150.000000
1	0.877493	0.941896	0.908555	327.000000
accuracy	0.870021	0.870021	0.870021	0.870021
macro avg	0.863350	0.827615	0.841958	477.000000
weighted avg	0.868598	0.870021	0.866670	477.000000

Tuned\_KNN\_on\_Imbalanced\_Dataset\_train:

	precision	recall	f1-score	support
0	1.0	1.0	1.0	615.0
1	1.0	1.0	1.0	1289.0
accuracy	1.0	1.0	1.0	1.0
macro avg	1.0	1.0	1.0	1904.0
weighted avg	1.0	1.0	1.0	1904.0

Tuned\_MLPClassifier\_on\_Balanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.687500	0.733333	0.709677	150.000000
1	0.873817	0.847095	0.860248	327.000000
accuracy	0.811321	0.811321	0.811321	0.811321
macro avg	0.780659	0.790214	0.784963	477.000000
weighted avg	0.815227	0.811321	0.812899	477.000000

Tuned\_MLPClassifier\_on\_Balanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.821667	0.764934	0.792286	1289.000000
1	0.780116	0.833980	0.806149	1289.000000
accuracy	0.799457	0.799457	0.799457	0.799457
macro avg	0.800891	0.799457	0.799218	2578.000000
weighted avg	0.800891	0.799457	0.799218	2578.000000

Tuned\_MLPClassifier\_on\_Imbalanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.833333	0.566667	0.674603	150.000000
1	0.826667	0.948012	0.883191	327.000000
accuracy	0.828092	0.828092	0.828092	0.828092
macro avg	0.830000	0.757339	0.778897	477.000000
weighted avg	0.828763	0.828092	0.817597	477.000000

Tuned\_MLPClassifier\_on\_Imbalanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.833713	0.595122	0.694497	615.000000
1	0.830034	0.943367	0.883079	1289.000000
accuracy	0.830882	0.830882	0.830882	0.830882
macro avg	0.831874	0.769244	0.788788	1904.000000
weighted avg	0.831222	0.830882	0.822166	1904.000000

Tuned\_Logistic\_Regression\_on\_Balanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.758824	0.860000	0.806250	150.000000
1	0.931596	0.874618	0.902208	327.000000
accuracy	0.870021	0.870021	0.870021	0.870021
macro avg	0.845210	0.867309	0.854229	477.000000
weighted avg	0.877265	0.870021	0.872033	477.000000

Tuned\_Logistic\_Regression\_on\_Balanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.882022	0.852599	0.867061	1289.000000
1	0.857357	0.885958	0.871423	1289.000000
accuracy	0.869279	0.869279	0.869279	0.869279
macro avg	0.869690	0.869279	0.869242	2578.000000
weighted avg	0.869690	0.869279	0.869242	2578.000000

Tuned\_Logistic\_Regression\_on\_Imbalanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.806250	0.860000	0.832258	150.000000
1	0.933754	0.905199	0.919255	327.000000
accuracy	0.890985	0.890985	0.890985	0.890985
macro avg	0.870002	0.882599	0.875756	477.000000
weighted avg	0.893658	0.890985	0.891897	477.000000

Tuned\_Logistic\_Regression\_on\_Imbalanced\_Dataset\_train:

	precision	recall	f1-score	support
0	0.782991	0.868293	0.823439	615.000000
1	0.933715	0.885182	0.908801	1289.000000
accuracy	0.879727	0.879727	0.879727	0.879727
macro avg	0.858353	0.876737	0.866120	1904.000000
weighted avg	0.885031	0.879727	0.881229	1904.000000

Simple\_Logistic\_Regesssion\_on\_balanced\_Dataset\_test:

	precision	recall	f1-score	support
0	0.754491	0.840000	0.794953	150.000000
1	0.922581	0.874618	0.897959	327.000000
accuracy	0.863732	0.863732	0.863732	0.863732
macro avg	0.838536	0.857309	0.846456	477.000000

```
weighted avg    0.869722  0.863732  0.865567  477.000000
```

```
-----  
-----
```

```
Simple_Logistic_Regresssion_on_balanced_Dataset_train:
```

	precision	recall	f1-score	support
0	0.879936	0.847168	0.863241	1289.000000
1	0.852655	0.884407	0.868241	1289.000000
accuracy	0.865787	0.865787	0.865787	0.865787
macro avg	0.866295	0.865787	0.865741	2578.000000
weighted avg	0.866295	0.865787	0.865741	2578.000000

```
-----  
-----
```

```
Simple_Logistic_Regression_on_Imbalanced_Dataset_test:
```

	precision	recall	f1-score	support
0	0.908397	0.793333	0.846975	150.000000
1	0.910405	0.963303	0.936107	327.000000
accuracy	0.909853	0.909853	0.909853	0.909853
macro avg	0.909401	0.878318	0.891541	477.000000
weighted avg	0.909773	0.909853	0.908078	477.000000

```
-----  
-----
```

```
Simple_Logistic_Regression_on_Imbalanced_Dataset_train:
```

	precision	recall	f1-score	support
0	0.867403	0.765854	0.813472	615.000000
1	0.894195	0.944143	0.918491	1289.000000
accuracy	0.886555	0.886555	0.886555	0.886555
macro avg	0.880799	0.854998	0.865981	1904.000000
weighted avg	0.885541	0.886555	0.884569	1904.000000

```
-----  
-----
```

# CHAPTER 7 ACTIONABLE INSIGHTS AND RECOMMENDATIONS

## INSIGHTS

The percentages of employees with different education levels are almost same

Majority (35%) of the employees currently are at designation level 2, followed by designation level 1 (31%) and 3 (26%). Less than 5% of the employees are currently in higher designations.

Almost 43% of the employees joined at lowest designation (1). 34% joined at level 2, 20% at level 3 and below 2% joined at higher levels.

The median of the Income for employees having higher Grades is greater.

Top reporting days is 24 days.

Recall and F1 or F2 Score should be considered important in driver churn prediction case.

Initial Dataset contains 12 Features. But after Feature Engineering feature space was increased to 31 features and 1 target

Models under perform with outliers in Age and Gender. So they are handled by using neighboring value in ordered data.

For basic models Imbalanced models may dominate balanced models in terms of metrics, but ensemble models definitely Balanced models provide better accuracy and generalization. It can be observed in learning curves and classification reports.

To deploy the model, It is better to use Balanced Dataset and Voting/Stacking ensembling on Boosted Models.

Join\_year, TBV\_raise, QuarterlyRating\_sum\_log are most important features positively

TBV\_mean\_negative\_log, TBV\_sum\_positive, Quarerly\_rating\_std, Grade raise, income raise are most important features negatively.

Models tend to perform better on balanced datasets, showing higher validation scores and narrower gaps between training and validation scores.

Overfitting is more pronounced on imbalanced datasets, with larger gaps between training and validation scores.

Ensemble models like Stacking, Voting, LightGBM, and XGBoost generally perform well but still show signs of overfitting on imbalanced data.

Simpler models like Logistic Regression and SVC demonstrate better generalization but may have lower overall performance compared to ensemble methods.

Max PR\_AUC\_test achieved is 0.97 using Boosting, Voting and Stacking

Max AU\_ROC\_test achieved is 0.96 using Boosting models

## RECOMMENDATIONS

Increase the Dataset size for better accuracy and generalization. Learning curves shows that there is monotonic increase in validation scores with increase in training samples.

Cascading with two level may not provide better accuracy. So Better go for four or five level cascade models.

Hyper parameter tuning can be done by using Grid search cv for better tuning parameters. But it may cost more time.

Robust performance can be achieved by Balancing dataset and using Ensemble methods like Voting and Stacking.

Regularization can be done to reduce the overfitting scenarios in ensemble methods

Deploying can be done by integrating MLFlow with AWS Sagemaker/Streamlit