📃 Business Case: OLA - Ensemble Learning 🚙

Analysed by : KASI

About Ola

Ola is a major player in the ride-hailing industry, known for its vast network of drivers and vehicles, providing convenient and affordable transportation options. However, it faces significant challenges in driver recruitment and retention, which directly impact its operational efficiency and customer satisfaction.

Problem Statement

Ola is struggling with high attrition rates among its drivers. This churn creates multiple challenges:

1. **Operational Challenges**: A constant need to recruit new drivers, which is costly and resource-intensive.
2. **Retention Issues**: Difficulty in retaining existing drivers who might leave due to dissatisfaction, better offers, or other factors.
3. **Business Impact**: Frequent driver exits impact morale, operational consistency, and financial stability since acquiring new drivers is more expensive than retaining current ones.

The goal is to predict whether a driver will leave the company based on historical and demographic data, enabling Ola to take proactive steps to improve retention.

Objective

The primary objective is:

- **Predict driver attrition** using available data (demographics, tenure, performance, etc.).
- This prediction will help Ola identify at-risk drivers and design targeted interventions (e.g., incentives, support programs, or policy changes) to retain them and reduce operational costs.

Concepts Used

This project incorporates the following key data science and machine learning techniques:

1. **Ensemble Learning - Bagging**:

- A technique that combines predictions from multiple models (e.g., Random Forest) to reduce variance and improve prediction stability and accuracy.

2. **Ensemble Learning - Boosting**:
   - An iterative approach that focuses on correcting errors made by weak models in the previous iterations (e.g., Gradient Boosting, XGBoost).
3. **KNN Imputation of Missing Values**:
   - A method to handle missing data by imputing values based on the similarity (proximity) of data points.
4. **Working with an Imbalanced Dataset**:
   - Addressing class imbalance where attrition (leaving drivers) might be a minority class compared to non-attrition. Techniques such as SMOTE (Synthetic Minority Oversampling Technique), under-sampling, or class-weighted algorithms may be used to balance the dataset for better model performance.

Libraries 📚

In [ ]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

from sklearn.preprocessing import MinMaxScaler

from sklearn.model_selection import train_test_split

from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score
from sklearn.metrics import confusion_matrix, ConfusionMatrixDisplay, classification_report
from sklearn.metrics import precision_recall_curve
from sklearn.metrics import roc_curve, auc
from sklearn.metrics import roc_curve, roc_auc_score

from imblearn.over_sampling import SMOTE
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import RandomizedSearchCV
import datetime as dt

import warnings
warnings.filterwarnings('ignore')
```

In [ ]:
```
pd.set_option('display.max_columns', None)
```

Exploring the data... 🔎

In [ ]:
```
data = pd.read_csv('/content/drive/MyDrive/Colab Notebooks/Data Sets/ola_driver_scaler.csv')
```

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

Out[ ]:

	Unnamed: 0	MMM-YY	Driver_ID	Age	Gender	City	Education_Level	Income	Dateofjoining	LastWorkingDate	Joining Designation	Grade	Total Business Value	Quarter Ratin
0	0	01/01/19	1	28.0	0.0	C23	2	57387	24/12/18	NaN	1	1	2381060	
1	1	02/01/19	1	28.0	0.0	C23	2	57387	24/12/18	NaN	1	1	-665480	
2	2	03/01/19	1	28.0	0.0	C23	2	57387	24/12/18	03/11/19	1	1	0	
3	3	11/01/20	2	31.0	0.0	C7	2	67016	11/06/20	NaN	2	2	0	
4	4	12/01/20	2	31.0	0.0	C7	2	67016	11/06/20	NaN	2	2	0	

In [ ]:
```
# Checking the number of rows and columns
print(f"The number of rows: {data.shape[0]:,} \nThe number of columns: {data.shape[1]}")
```

```
The number of rows: 19,104
The number of columns: 14
```

In [ ]:
```
# Check all column names
data.columns
```

Out[ ]:
```
Index(['Unnamed: 0', 'MMM-YY', 'Driver_ID', 'Age', 'Gender', 'City',
       'Education_Level', 'Income', 'Dateofjoining', 'LastWorkingDate',
       'Joining Designation', 'Grade', 'Total Business Value',
       'Quarterly Rating'],
      dtype='object')
```

Observations on Data 👀

In [ ]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 19104 entries, 0 to 19103
Data columns (total 14 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   Unnamed: 0            19104 non-null  int64
 1   MMM-YY                19104 non-null  object
 2   Driver_ID             19104 non-null  int64
 3   Age                   19043 non-null  float64
 4   Gender                19052 non-null  float64
 5   City                  19104 non-null  object
 6   Education_Level       19104 non-null  int64
 7   Income                19104 non-null  int64
 8   Dateofjoining         19104 non-null  object
 9   LastWorkingDate       1616 non-null   object
 10  Joining Designation   19104 non-null  int64
 11  Grade                 19104 non-null  int64
 12  Total Business Value  19104 non-null  int64
 13  Quarterly Rating      19104 non-null  int64
dtypes: float64(2), int64(8), object(4)
memory usage: 2.0+ MB
```

🔍**OBSERVATION**🔎

1. **Data Overview**:
   - Total entries: **19,104 rows**
   - Total columns: **14**
   - Missing data in columns:
     - **Age**: 61 missing values.
     - **Gender**: 52 missing values.
     - **LastWorkingDate**: Majority missing (17,488 missing values, only 1,616 non-null). Likely indicates currently active drivers.
2. **Columns Details**:
   - `MMMM-YY` (object): Monthly reporting date. Time-based feature for trend analysis.
   - `Driver_ID` (int64): Unique identifier for drivers.
   - `Age` (float64): Driver's age, has 61 missing values.
   - `Gender` (float64): Encoded (Male = 0, Female = 1). Missing values need imputation.
   - `City` (object): Categorical variable, encoded city codes.
   - `Education_Level` (int64): Encoded education levels (0 = 10+, 1 = 12+, 2 = graduate).
   - `Income` (int64): Monthly average income of the driver.
   - `DateOfJoining` (object): Date when the driver joined the company.
   - `LastWorkingDate` (object): Date when the driver left (or null if still active).
   - `Joining Designation` (int64): Initial designation of the driver.
   - `Grade` (int64): Grade assigned to the driver during the reporting period.
   - `Total Business Value` (int64): Monthly business value, with potential for negative values indicating cancellations or refunds.
   - `Quarterly Rating` (int64): Driver performance rating (1 to 5).
3. **Data Challenges**:
   - **Missing Values**: Key variables like `Age`, `Gender`, and especially `LastWorkingDate` have missing values requiring appropriate handling.
   - **Imbalanced Dataset**: Drivers with `LastWorkingDate` populated are likely those who have left, potentially creating an imbalance in attrition prediction.
   - **Temporal Features**: `MMMM-YY`, `DateOfJoining`, and `LastWorkingDate` need preprocessing to derive tenure and time-based features.
4. **Potential Derived Features**:
   - **Driver Tenure**: Difference between reporting date (`MMMM-YY`) and joining date (`DateOfJoining`).
   - **Activity Status**: Binary column indicating whether the driver is active (`LastWorkingDate` is null) or has left (`LastWorkingDate` populated).
   - **Income Trends**: Calculate income trends over time for active drivers.
   - **Performance Metrics**: Analyze `Quarterly Rating` and `Total Business Value` trends.

In [ ]:
```python
# Number of unique values in each coluumn and datatype:
print("Number of unique values in each coluumn and datatype:")
print("-" * 55)
for i, elem in (enumerate(data.columns)):
  print(f"{i+1}. {elem}: {data[elem].nunique(), data[elem].dtypes}")
```

```
Number of unique values in each coluumn and datatype:
-----------------------------------------------------
1. Unnamed: 0: (19104, dtype('int64'))
2. MMM-YY: (24, dtype('O'))
3. Driver_ID: (2381, dtype('int64'))
4. Age: (36, dtype('float64'))
5. Gender: (2, dtype('float64'))
6. City: (29, dtype('O'))
7. Education_Level: (3, dtype('int64'))
8. Income: (2383, dtype('int64'))
9. Dateofjoining: (869, dtype('O'))
10. LastWorkingDate: (493, dtype('O'))
11. Joining Designation: (5, dtype('int64'))
12. Grade: (5, dtype('int64'))
13. Total Business Value: (10181, dtype('int64'))
14. Quarterly Rating: (4, dtype('int64'))
```

Processing the data 🧰

In [ ]:
```python
# Creating a deep copy
df1 = data.copy()
```

In [ ]:
```python
# Remove Unnamed column
df1.drop('Unnamed: 0', axis=1,inplace=True)
```

In [ ]:
```python
# Doing the required operations for each columns
df1 = df1.rename(columns={'MMM-YY': 'Reporting_Date'})
df1['Reporting_Date'] = pd.to_datetime(df1['Reporting_Date'], format='mixed',errors='coerce')
df1['Dateofjoining'] = pd.to_datetime(df1['Dateofjoining'], format='mixed',errors='coerce')
df1['LastWorkingDate'] = pd.to_datetime(df1['LastWorkingDate'], format='mixed',errors='coerce')
df1['City'] = df1['City'].astype('category') #OHE
df1["Gender"].replace({0.0:"Male",1.0:"Female"},inplace=True)
df1['Gender'] = df1['Gender'].astype('category') #Convert to int after EDA
df1['Education_Level'] = df1['Education_Level'].astype('category') #Convert to int after EDA
df1['Joining Designation'] = df1['Joining Designation'].astype('category') #Convert to int after EDA
df1['Grade'] = df1['Grade'].astype('category') #Convert to int after EDA
```

In [ ]:
```python
print(df1.dtypes)
```

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

Feature Engineering 🔩

Creating Target feature

In [ ]:
```python
df1.head(15)
```

Out[ ]:

	Reporting_Date	Driver_ID	Age	Gender	City	Education_Level	Income	Dateofjoining	LastWorkingDate	Joining Designation	Grade	Total Business Value	Quarterly Rating
0	2019-01-01	1	28.0	Male	C23	2	57387	2018-12-24	NaT	1	1	2381060	2
1	2019-02-01	1	28.0	Male	C23	2	57387	2018-12-24	NaT	1	1	-665480	2
2	2019-03-01	1	28.0	Male	C23	2	57387	2018-12-24	2019-03-11	1	1	0	2
3	2020-11-01	2	31.0	Male	C7	2	67016	2020-11-06	NaT	2	2	0	1
4	2020-12-01	2	31.0	Male	C7	2	67016	2020-11-06	NaT	2	2	0	1
5	2019-12-01	4	43.0	Male	C13	2	65603	2019-12-07	NaT	2	2	0	1
6	2020-01-01	4	43.0	Male	C13	2	65603	2019-12-07	NaT	2	2	0	1
7	2020-02-01	4	43.0	Male	C13	2	65603	2019-12-07	NaT	2	2	0	1
8	2020-03-01	4	43.0	Male	C13	2	65603	2019-12-07	NaT	2	2	350000	1
9	2020-04-01	4	43.0	Male	C13	2	65603	2019-12-07	2020-04-27	2	2	0	1
10	2019-01-01	5	29.0	Male	C9	0	46368	2019-01-09	NaT	1	1	0	1
11	2019-02-01	5	29.0	Male	C9	0	46368	2019-01-09	NaT	1	1	120360	1
12	2019-03-01	5	29.0	Male	C9	0	46368	2019-01-09	2019-03-07	1	1	0	1
13	2020-08-01	6	31.0	Female	C11	1	78728	2020-07-31	NaT	3	3	0	1
14	2020-09-01	6	31.0	Female	C11	1	78728	2020-07-31	NaT	3	3	0	1

In [ ]:
```
churn =  (df1.groupby('Driver_ID').agg({'LastWorkingDate':'last'})['LastWorkingDate'].isna()).reset_index()
churn['LastWorkingDate'].replace({True:0,False:1},inplace=True) # Churned -> 1, not churned -> 0
churn.rename(columns={'LastWorkingDate':'churn'},inplace=True)
churn.head()
```

Out[ ]:

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

Quarterly Rating

In [ ]:
```
# If Quarterly Rating has increased than value 1 else 0
QR1 =  (df1.groupby('Driver_ID').agg({'Quarterly Rating':'first'})).reset_index()
QR2 = (df1.groupby('Driver_ID').agg({'Quarterly Rating':'last'})).reset_index()
```

In [ ]:
```
QR1.isna().sum(),QR2.isna().sum()
```

Out[ ]:
```
(Driver_ID          0
 Quarterly Rating    0
 dtype: int64,
 Driver_ID          0
 Quarterly Rating    0
 dtype: int64)
```

In [ ]:
```
churn = churn.merge(QR1,on='Driver_ID')
churn = churn.merge(QR2,on='Driver_ID')
churn['Rating_increase']=np.where(churn['Quarterly Rating_x'] < churn['Quarterly Rating_y'], 1,0)
```

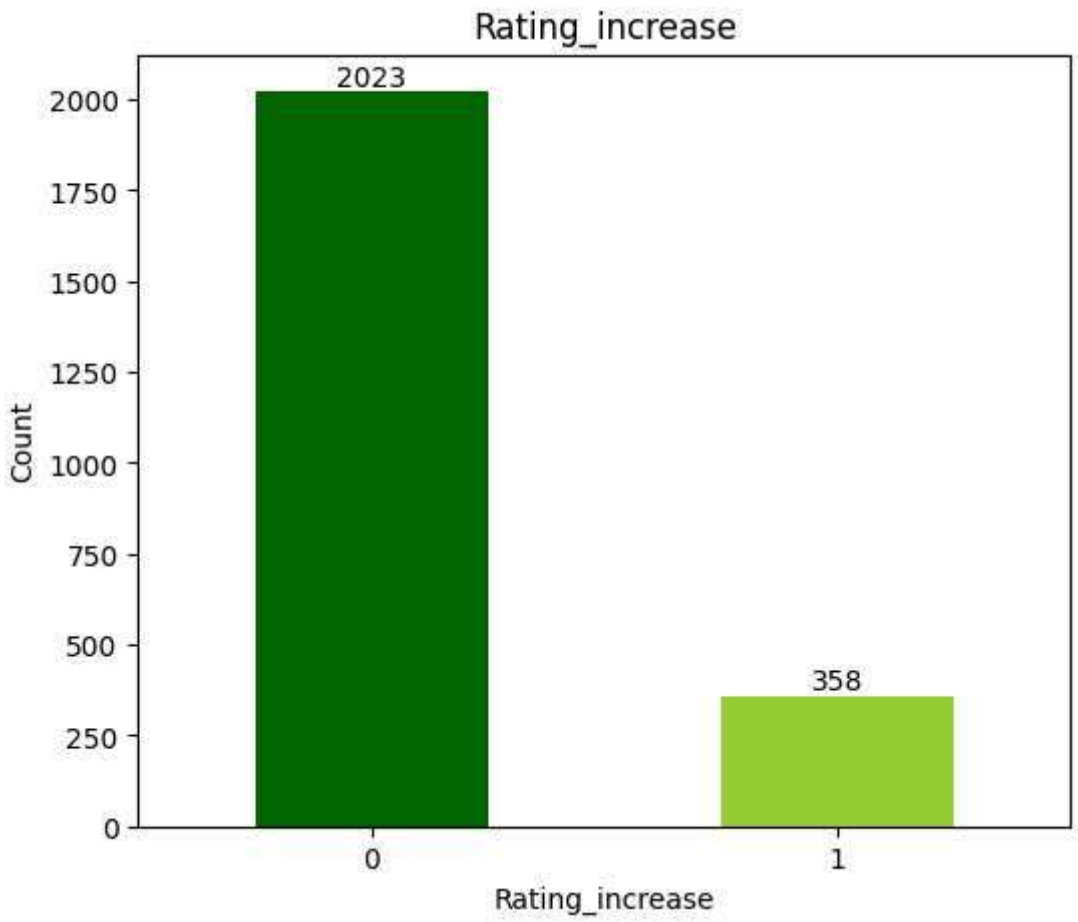
In [ ]:
```
churn.head()
```

Out[ ]:

	Driver_ID	churn	Quarterly Rating_x	Quarterly Rating_y	Rating_increase
0	1	1	2	2	0
1	2	0	1	1	0
2	4	1	1	1	0
3	5	1	1	1	0
4	6	0	1	2	1

In [ ]:
```
plt.figure(figsize=(6,5))
label = churn['Rating_increase'].value_counts().plot(kind="bar", color=['darkgreen', 'yellowgreen'])
for i in label.containers:
    label.bar_label(i)
plt.xticks(rotation=360)
plt.xlabel("Rating_increase")
plt.ylabel("Count")
```

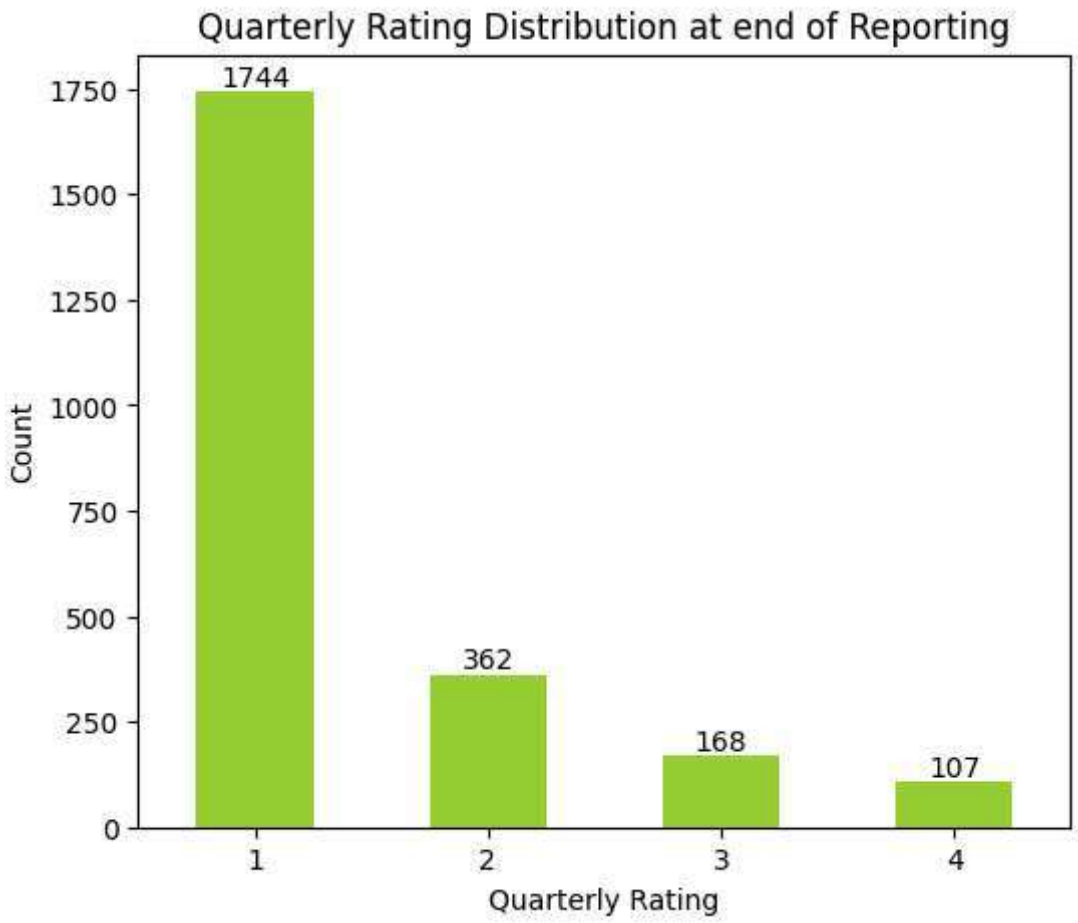
```
plt.title("Rating_increase")
plt.show()
```



🔍OBSERVATION🔎

- This indicates that a significant majority 2023 (approximately 85%) of the drivers are either maintaining the same rating or experiencing a decline.
- Only 358 drivers (approximately 15%) showed an improvement in their ratings during the same period.

In [ ]:
```
plt.figure(figsize=(6,5))
label = churn['Quarterly Rating_y'].value_counts().plot(kind="bar", color='yellowgreen')
for i in label.containers:
    label.bar_label(i)
plt.xticks(rotation=360)
plt.xlabel("Quarterly Rating")
plt.ylabel("Count")
plt.title("Quarterly Rating Distribution at end of Reporting")
plt.show()
```



🔍OBSERVATION🔎

- The majority of drivers (1,744 out of the total) have a Quarterly Rating of 1. This indicates that most drivers are rated poorly on their quarterly performance metrics.
- The small number of drivers with higher ratings (3 or 4) suggests that most drivers are underperforming or the rating system is stringent.

Income

In [ ]:
```
income1 =  (df1.groupby('Driver_ID').agg({'Income':'first'})['Income']).reset_index()
income2 = (df1.groupby('Driver_ID').agg({'Income':'last'})['Income']).reset_index()
```

In [ ]:
```
income1.shape,income2.shape
```

Out[ ]:
```
((2381, 2), (2381, 2))
```

In [ ]:
```
income1.isna().sum(),income2.isna().sum()
```

Out[ ]:
```
(Driver_ID    0
 Income       0
 dtype: int64,
 Driver_ID    0
 Income       0
 dtype: int64)
```

In [ ]:
```
churn = churn.merge(income1,on='Driver_ID')
churn = churn.merge(income2,on='Driver_ID')
churn['Income_increase'] = np.where(churn['Income_x'] < churn['Income_y'], 1,0)
```

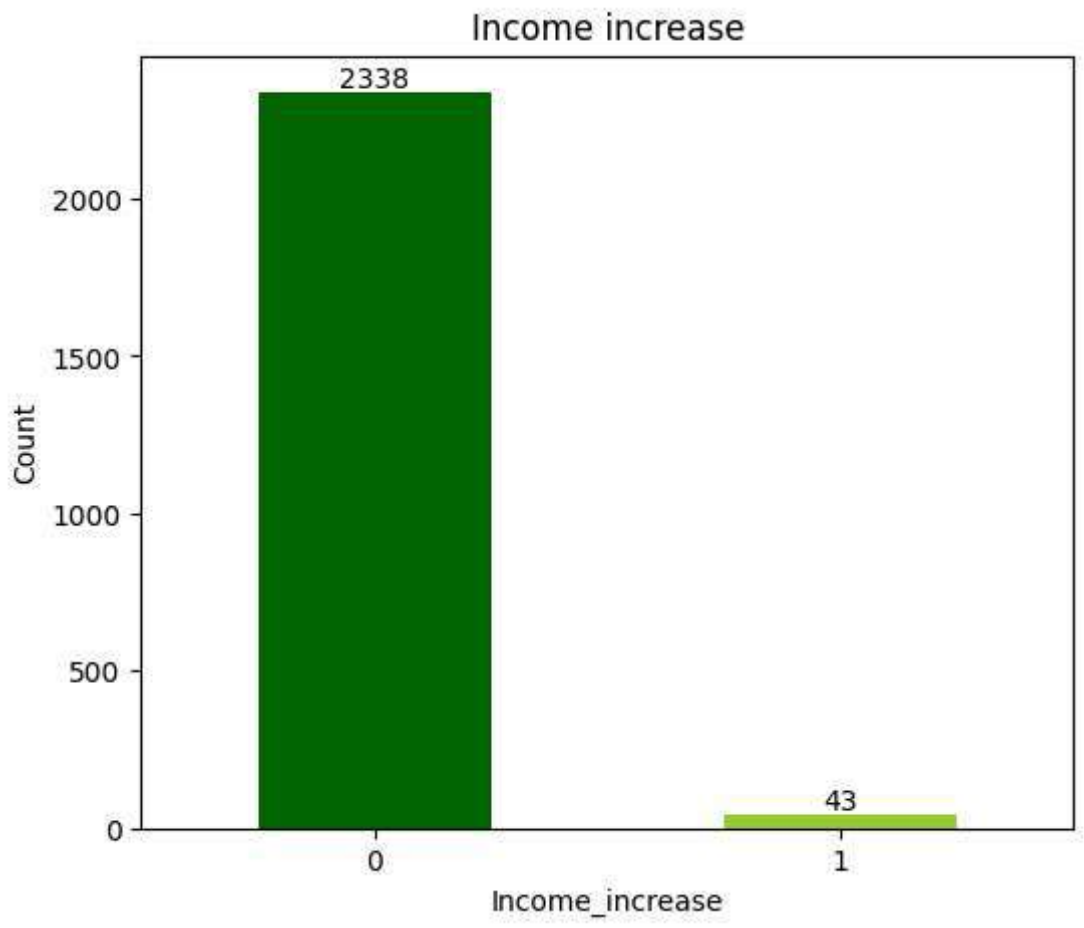
In [ ]:
```
churn['Income_increase'].value_counts()
```

Out[ ]:

Income_increase	count
0	2338
1	43

dtype: int64

In [ ]:
```
plt.figure(figsize=(6,5))
label = churn['Income_increase'].value_counts().plot(kind="bar", color=['darkgreen', 'yellowgreen'])
for i in label.containers:
    label.bar_label(i)
plt.xticks(rotation=360)
plt.xlabel("Income_increase")
plt.ylabel("Count")
plt.title("Income increase")
plt.show()
```



🔍OBSERVATION🔎

- Out of the total drivers, 2,338 drivers (approximately 98%) either experienced no increase or a decrease in their income over the observed period. This reflects a significant challenge in driver satisfaction, as stagnant or declining income can lead to dissatisfaction and potentially higher churn.
- Only 43 drivers (approximately 2%) experienced an increase in income during the period. This extremely low percentage indicates that income progression is rare, which could discourage drivers from staying loyal to Ola.

In [ ]:
```
churn.columns
```

Out[ ]:
```
Index(['Driver_ID', 'churn', 'Quarterly Rating_x', 'Quarterly Rating_y',
       'Rating_increase', 'Income_x', 'Income_y', 'Income_increase'],
      dtype='object')
```

In [ ]:
```
target = churn[['Driver_ID','churn','Rating_increase','Income_increase']]
target.head()
```

Out[ ]:

	Driver_ID	churn	Rating_increase	Income_increase
0	1	1	0	0
1	2	0	0	0
2	4	1	0	0
3	5	1	0	0
4	6	0	1	0

Aggregated dataframe

In [ ]:
```
# Creating a deep copy
df = df1.copy()
```

In [ ]:
```
functions = {'Reporting_Date':'count',
             'Driver_ID':'first',
             'Age':'max',
             'Gender':'last',
             'City':'last',
             'Education_Level':'last',
             'Dateofjoining':'first',
             'LastWorkingDate':'last',
             'Grade':'last',
             'Total Business Value':'sum',
             'Income':'last',
             'Joining Designation':'last',
             'Quarterly Rating':'mean'} #last
df = df.groupby('Driver_ID').aggregate(functions).reset_index(drop=True)
df.rename(columns={'Reporting_Date':'Reportings'},inplace=True)
```

In [ ]:
```
# Merge with target
df = df.merge(target,on='Driver_ID')
```

In [ ]:
```
# Maintaining decimal places
df['Quarterly Rating'] = round(df['Quarterly Rating'],1)
```

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

Out[ ]:

	Reportings	Driver_ID	Age	Gender	City	Education_Level	Dateofjoining	LastWorkingDate	Grade	Total Business Value	Income	Joining Designation	Quarterly Rating	churn
0	3	1	28.0	Male	C23	2	2018-12-24	2019-03-11	1	1715580	57387	1	2.0	1
1	2	2	31.0	Male	C7	2	2020-11-06	NaT	2	0	67016	2	1.0	0
2	5	4	43.0	Male	C13	2	2019-12-07	2020-04-27	2	350000	65603	2	1.0	1
3	3	5	29.0	Male	C9	0	2019-01-09	2019-03-07	1	120360	46368	1	1.0	1
4	5	6	31.0	Female	C11	1	2020-07-31	NaT	3	1265000	78728	3	1.6	0

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

Out[ ]:
```
(2381, 16)
```

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

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2381 entries, 0 to 2380
Data columns (total 16 columns):
 #   Column                Non-Null Count  Dtype         
---  ------                --------------  -----         
 0   Reportings            2381 non-null   int64         
 1   Driver_ID             2381 non-null   int64         
 2   Age                   2381 non-null   float64       
 3   Gender                2381 non-null   category      
 4   City                  2381 non-null   category      
 5   Education_Level       2381 non-null   category      
 6   Dateofjoining         2381 non-null   datetime64[ns]
 7   LastWorkingDate       1616 non-null   datetime64[ns]
 8   Grade                 2381 non-null   category      
 9   Total Business Value  2381 non-null   int64         
 10  Income                2381 non-null   int64         
 11  Joining Designation   2381 non-null   category      
 12  Quarterly Rating      2381 non-null   float64       
 13  churn                 2381 non-null   int64         
 14  Rating_increase       2381 non-null   int64         
 15  Income_increase       2381 non-null   int64         
dtypes: category(5), datetime64[ns](2), float64(2), int64(7)
memory usage: 218.3 KB
```

In [ ]:
```
# Doing the required operations for each columns
#df['Reportings'] = df['Reportings'].astype('category')
df['Age'] = df['Age'].astype('int')
```

```
df['churn'] = df['churn'].astype('category')
#df['Quarterly Rating'] = df['Quarterly Rating'].astype('category')
df['Rating_increase'] = df['Rating_increase'].astype('category')
df['Income_increase'] = df['Income_increase'].astype('category')
```

In [ ]:
```
print(df.dtypes)
```

```
Reportings                      int64
Driver_ID                       int64
Age                             int64
Gender                       category
City                         category
Education_Level              category
Dateofjoining          datetime64[ns]
LastWorkingDate        datetime64[ns]
Grade                        category
Total Business Value            int64
Income                          int64
Joining Designation          category
Quarterly Rating              float64
churn                        category
Rating_increase              category
Income_increase              category
dtype: object
```

In [ ]:
```
# Number of unique values in each column and datatype:
print("Number of unique values in each column and datatype:")
print("-" * 55)
for i, elem in (enumerate(df.columns)):
  print(f"{i+1}. {elem}: {df[elem].nunique(), df[elem].dtypes}")
```

```
Number of unique values in each column and datatype:
-------------------------------------------------------
1. Reportings: (24, dtype('int64'))
2. Driver_ID: (2381, dtype('int64'))
3. Age: (36, dtype('int64'))
4. Gender: (2, CategoricalDtype(categories=['Female', 'Male'], ordered=False, categories_dtype=object))
5. City: (29, CategoricalDtype(categories=['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'],
, ordered=False, categories_dtype=object))
6. Education_Level: (3, CategoricalDtype(categories=[0, 1, 2], ordered=False, categories_dtype=int64))
7. Dateofjoining: (869, dtype('<M8[ns]'))
8. LastWorkingDate: (493, dtype('<M8[ns]'))
9. Grade: (5, CategoricalDtype(categories=[1, 2, 3, 4, 5], ordered=False, categories_dtype=int64))
10. Total Business Value: (1629, dtype('int64'))
11. Income: (2339, dtype('int64'))
12. Joining Designation: (5, CategoricalDtype(categories=[1, 2, 3, 4, 5], ordered=False, categories_dtype=int64))
13. Quarterly Rating: (31, dtype('float64'))
14. churn: (2, CategoricalDtype(categories=[0, 1], ordered=False, categories_dtype=int64))
15. Rating_increase: (2, CategoricalDtype(categories=[0, 1], ordered=False, categories_dtype=int64))
16. Income_increase: (2, CategoricalDtype(categories=[0, 1], ordered=False, categories_dtype=int64))
```

Check for Duplicate 👫

In [ ]:
```
df[df.duplicated]
```

Out[ ]:

Reportings	Driver_ID	Age	Gender	City	Education_Level	Dateofjoining	LastWorkingDate	Grade	Total Business Value	Income	Joining Designation	Quarterly Rating	churn

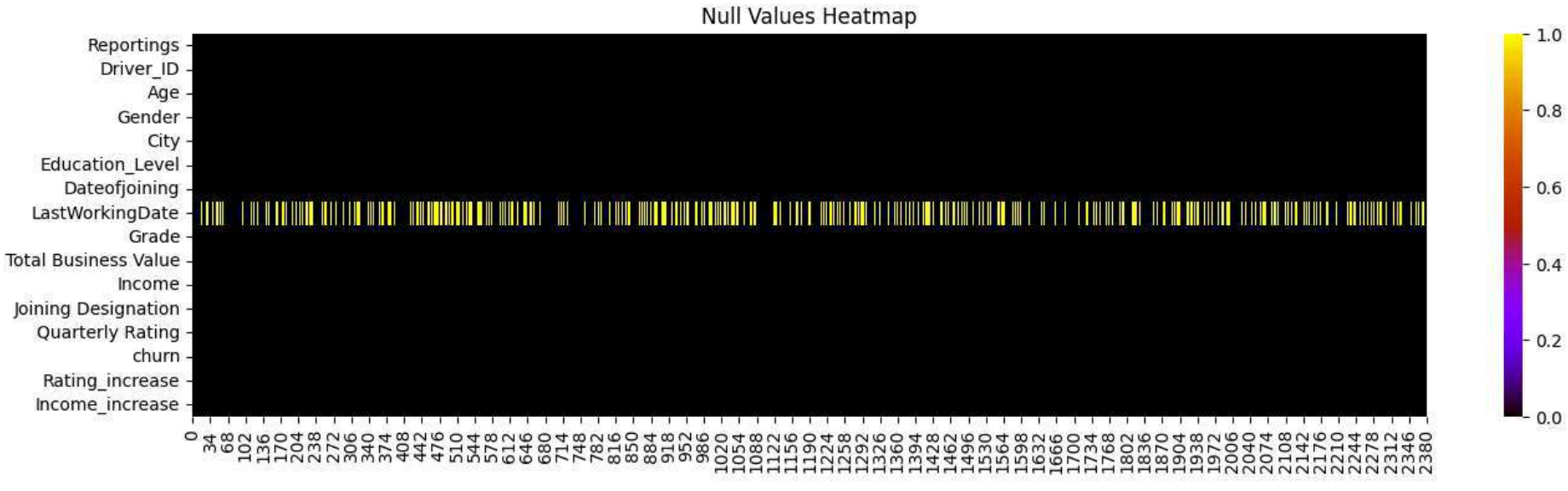
Missing value treatmen 🧹

In [ ]:
```
# How many percentage of data is missing in each column
missing_value = pd.DataFrame({'Missing Value': df.isnull().sum(), 'Percentage': (((df.isnull().sum() / len(df))*100)).round(2)})
missing_value.sort_values(by='Percentage', ascending=False)
```

Out[ ]:

	Missing Value	Percentage
LastWorkingDate	765	32.13
Reportings	0	0.00
Driver_ID	0	0.00
Age	0	0.00
Gender	0	0.00
City	0	0.00
Education_Level	0	0.00
Dateofjoining	0	0.00
Grade	0	0.00
Total Business Value	0	0.00
Income	0	0.00
Joining Designation	0	0.00
Quarterly Rating	0	0.00
churn	0	0.00
Rating_increase	0	0.00
Income_increase	0	0.00

In [ ]:
```
# Null value heatmap:
plt.figure(figsize = (16,4))
sns.heatmap(df.isnull().T, cmap='gnuplot')
plt.title('Null Values Heatmap')
plt.show()
```



🔍 OBSERVATION 🔎

- Lets keep the last working dat as it is because if it is null then the person is still working

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

Out[ ]:

	Reportings	Driver_ID	Age	Gender	City	Education_Level	Dateofjoining	LastWorkingDate	Grade	Total Business Value	Income	Joining Designation	Quarterly Rating	churn
0	3	1	28	Male	C23	2	2018-12-24	2019-03-11	1	1715580	57387	1	2.0	1
1	2	2	31	Male	C7	2	2020-11-06	NaT	2	0	67016	2	1.0	0
2	5	4	43	Male	C13	2	2019-12-07	2020-04-27	2	350000	65603	2	1.0	1
3	3	5	29	Male	C9	0	2019-01-09	2019-03-07	1	120360	46368	1	1.0	1
4	5	6	31	Female	C11	1	2020-07-31	NaT	3	1265000	78728	3	1.6	0

In [ ]:
```
# Display the range of attributes
print("Range of attributes:")
print("-" * 20)
df.describe(include='all').T
```

```
Range of attributes:
--------------------
```

Out[ ]:

	count	unique	top	freq	mean	min	25%	50%	75%	max	std
Reportings	2381.0	NaN	NaN	NaN	8.02352	1.0	3.0	5.0	10.0	24.0	6.78359
Driver_ID	2381.0	NaN	NaN	NaN	1397.559009	1.0	695.0	1400.0	2100.0	2788.0	806.161628
Age	2381.0	NaN	NaN	NaN	33.663167	21.0	29.0	33.0	37.0	58.0	5.983375
Gender	2381	2	Male	1404	NaN	NaN	NaN	NaN	NaN	NaN	NaN
City	2381	29	C20	152	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Education_Level	2381.0	3.0	2.0	802.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Dateofjoining	2381	NaN	NaN	NaN	2019-02-08 07:14:50.550189056	2013-04-01 00:00:00	2018-06-29 00:00:00	2019-07-21 00:00:00	2020-05-02 00:00:00	2020-12-28 00:00:00	NaN
LastWorkingDate	1616	NaN	NaN	NaN	2019-12-21 20:59:06.534653440	2018-12-31 00:00:00	2019-06-06 00:00:00	2019-12-20 12:00:00	2020-07-03 00:00:00	2020-12-28 00:00:00	NaN
Grade	2381.0	5.0	2.0	855.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Total Business Value	2381.0	NaN	NaN	NaN	4586741.822764	-1385530.0	0.0	817680.0	4173650.0	95331060.0	9127115.313446
Income	2381.0	NaN	NaN	NaN	59334.157077	10747.0	39104.0	55315.0	75986.0	188418.0	28383.666384
Joining Designation	2381.0	5.0	1.0	1026.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Quarterly Rating	2381.0	NaN	NaN	NaN	1.567535	1.0	1.0	1.0	2.0	4.0	0.720405
churn	2381.0	2.0	1.0	1616.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Rating_increase	2381.0	2.0	0.0	2023.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN
Income_increase	2381.0	2.0	0.0	2338.0	NaN	NaN	NaN	NaN	NaN	NaN	NaN

In [ ]:
```
# Display the statistical summary
print("statistical summary:")
print("-" * 20)
df.describe().T
```

```
statistical summary:
--------------------
```

Out[ ]:

	count	mean	min	25%	50%	75%	max	std
Reportings	2381.0	8.02352	1.0	3.0	5.0	10.0	24.0	6.78359
Driver_ID	2381.0	1397.559009	1.0	695.0	1400.0	2100.0	2788.0	806.161628
Age	2381.0	33.663167	21.0	29.0	33.0	37.0	58.0	5.983375
Dateofjoining	2381	2019-02-08 07:14:50.550189056	2013-04-01 00:00:00	2018-06-29 00:00:00	2019-07-21 00:00:00	2020-05-02 00:00:00	2020-12-28 00:00:00	NaN
LastWorkingDate	1616	2019-12-21 20:59:06.534653440	2018-12-31 00:00:00	2019-06-06 00:00:00	2019-12-20 12:00:00	2020-07-03 00:00:00	2020-12-28 00:00:00	NaN
Total Business Value	2381.0	4586741.822764	-1385530.0	0.0	817680.0	4173650.0	95331060.0	9127115.313446
Income	2381.0	59334.157077	10747.0	39104.0	55315.0	75986.0	188418.0	28383.666384
Quarterly Rating	2381.0	1.567535	1.0	1.0	1.0	2.0	4.0	0.720405

🔍**OBSERVATION**🔎

1. **Age**:
   - The average age of drivers is approximately 33.66 years.
   - The age ranges from 21 to 58 years, with a standard deviation of around 5.98 years.
2. **Gender**:
   - The dataset is predominantly male, with 1404 male drivers (around 59% of the dataset) and the remaining being female.
3. **City**:
   - The most frequent city is `C20`, with 152 drivers working there, but there are 29 distinct cities, showing a diverse distribution of drivers.
4. **Total Business Value**:
   - The average total business value per driver is around 4.59 million, but the values range from a minimum of -1.39 million to a maximum of 95.33 million, suggesting significant variation in business performance among drivers.
5. **Income**:
   - The average income is approximately 59,334, with a minimum income of 10,747 and a maximum of 188,418, indicating a wide spread of earnings among drivers.

Exploratory data analysis 📊 📈

Univariate Analysis

In [ ]:
```
# Selecting the categorical columns
categorical_cols = df.select_dtypes(include='category').columns
categorical_cols
```

Out[ ]:
```
Index(['Gender', 'City', 'Education_Level', 'Grade', 'Joining Designation',
       'churn', 'Rating_increase', 'Income_increase'],
      dtype='object')
```

In [ ]:
```
# Required colour palette
green_palette = ['#187c19', '#69b41e', '#8dc71e', '#b8d53d'] #'#0d5b11',
#green_palette = ['#F5FAD1', '#D1EBA1', '#A6D577', '#76C352', '#438032', '#189E1E', '#028B22']
```

In [ ]:
```
# Value couts for categorical columns
for elem in categorical_cols:
  print(f"Column Name: {elem}")
  print(df[elem].value_counts())
  print()
  print(round(((df[elem].value_counts(normalize=True)) * 100),2))
  print("_" * 35)
  print()
```

```
Column Name: Gender
Gender
Male      1404
Female     977
Name: count, dtype: int64

Gender
Male      58.97
Female    41.03
Name: proportion, dtype: float64
```

```
Column Name: City
City
C20    152
C15    101
C29     96
C26     93
C8      89
C27     89
C10     86
C16     84
C22     82
C3      82
C28     82
C12     81
C5      80
C1      80
C21     79
C14     79
C6      78
C4      77
C7      76
C9      75
C23     74
C25     74
C24     73
C2      72
C19     72
C17     71
C13     71
C18     69
C11     64
Name: count, dtype: int64

City
C20    6.38
C15    4.24
C29    4.03
C26    3.91
C8     3.74
C27    3.74
C10    3.61
C16    3.53
C22    3.44
C3     3.44
C28    3.44
C12    3.40
C5     3.36
C1     3.36
C21    3.32
C14    3.32
C6     3.28
C4     3.23
C7     3.19
C9     3.15
C23    3.11
C25    3.11
C24    3.07
C2     3.02
C19    3.02
C17    2.98
C13    2.98
C18    2.90
C11    2.69
Name: proportion, dtype: float64
```

```
Column Name: Education_Level
Education_Level
2    802
1    795
0    784
Name: count, dtype: int64

Education_Level
2    33.68
1    33.39
0    32.93
Name: proportion, dtype: float64
```

```
Column Name: Grade
```

```
Grade
2    855
1    741
3    623
4    138
5     24
Name: count, dtype: int64

Grade
2    35.91
1    31.12
3    26.17
4     5.80
5     1.01
Name: proportion, dtype: float64
```

```
Column Name: Joining Designation
Joining Designation
1    1026
2     815
3     493
4      36
5      11
Name: count, dtype: int64

Joining Designation
1    43.09
2    34.23
3    20.71
4     1.51
5     0.46
Name: proportion, dtype: float64
```

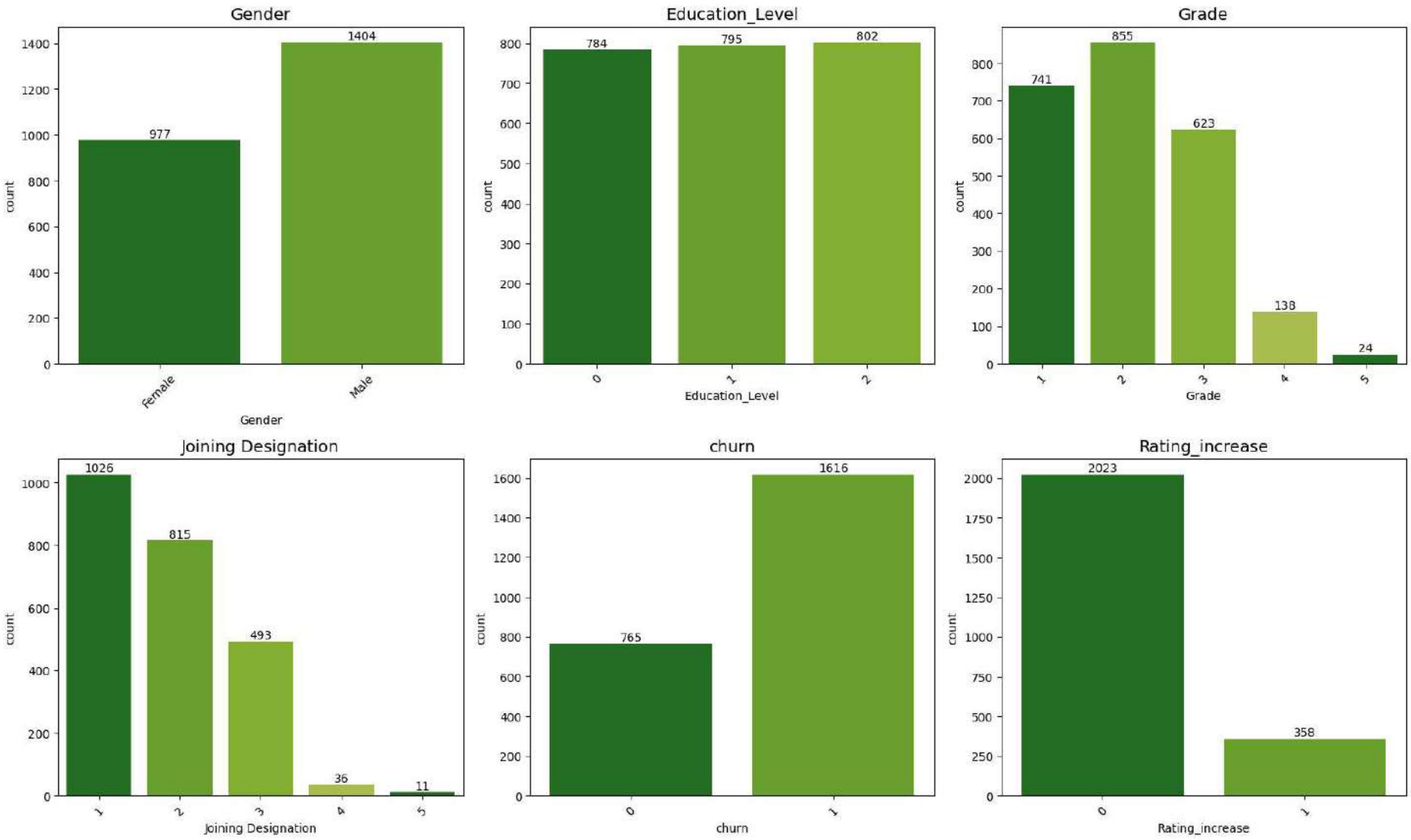
```
Column Name: churn
churn
1    1616
0     765
Name: count, dtype: int64

churn
1    67.87
0    32.13
Name: proportion, dtype: float64
```

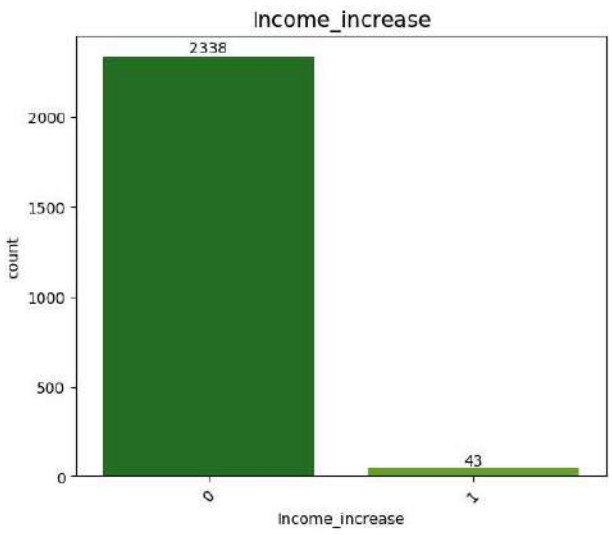
```
Column Name: Rating_increase
Rating_increase
0    2023
1     358
Name: count, dtype: int64

Rating_increase
0    84.96
1    15.04
Name: proportion, dtype: float64
```

```
Column Name: Income_increase
Income_increase
0    2338
1      43
Name: count, dtype: int64

Income_increase
0    98.19
1     1.81
Name: proportion, dtype: float64
```

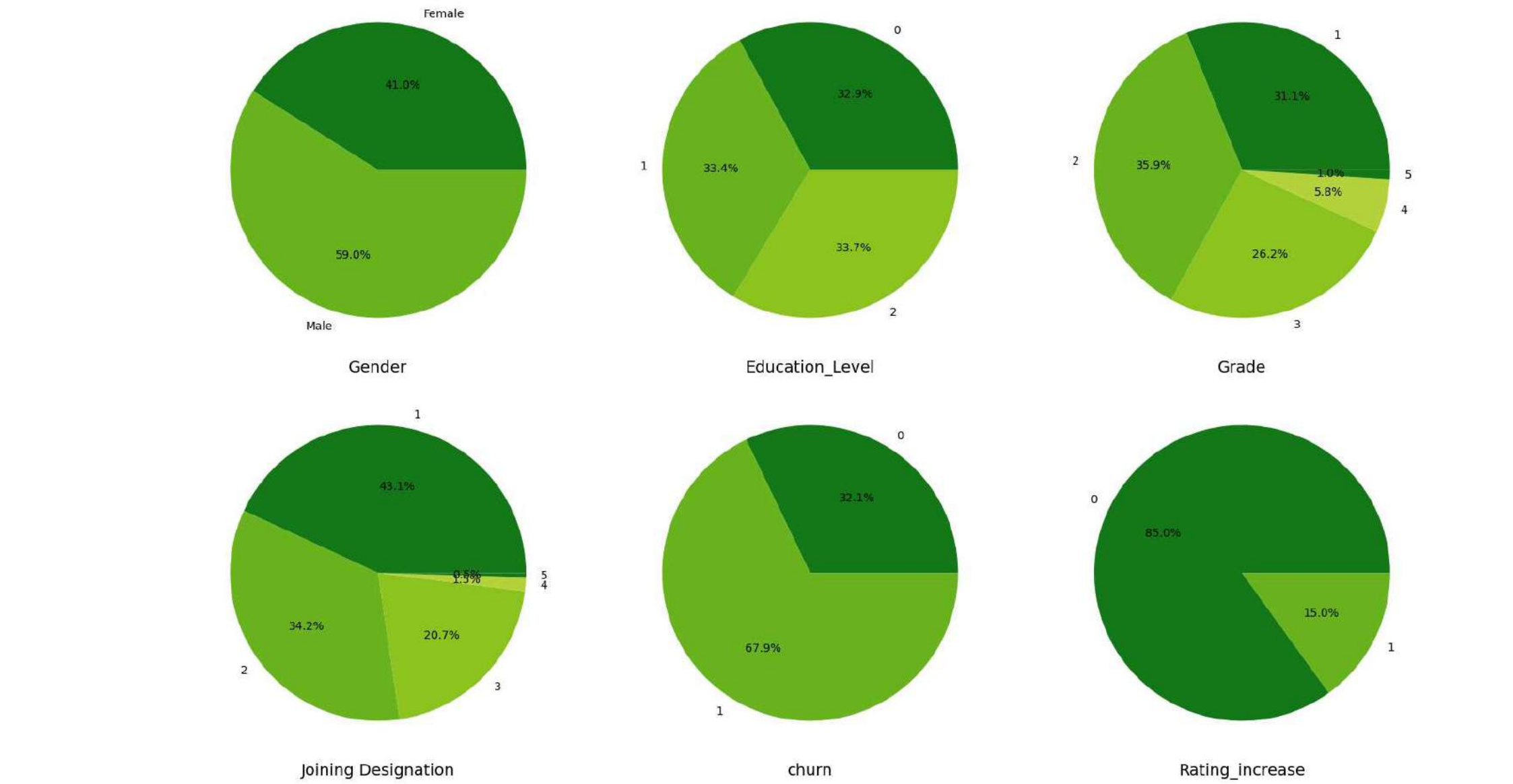
In [ ]:

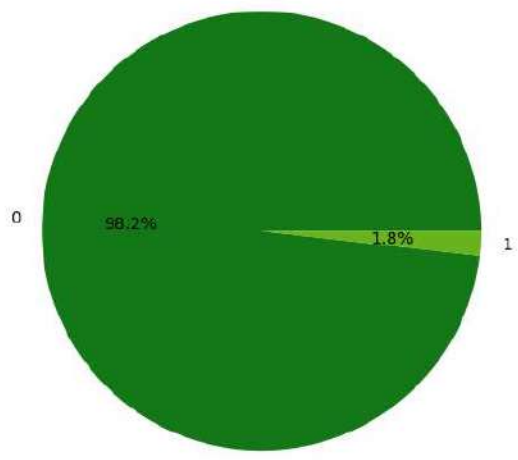
```python
# Count Plots for Categorical features
req_cat_col_plot = ['Gender', 'Education_Level', 'Grade', 'Joining Designation','churn', 'Rating_increase', 'Income_increase']

plt.figure(figsize=(17,20))
for i, elem in enumerate(req_cat_col_plot):
  plt.subplot(4,3,i+1)
  label = sns.countplot(data = df, x = elem, palette = green_palette)
  for i in label.containers:
    label.bar_label(i)

  plt.xticks(rotation = 45)
  plt.ylabel('count')
  plt.title(elem, fontsize=14)

#plt.suptitle("Count Plots for Categorical features", fontsize = 18)
plt.tight_layout()
plt.show()
```





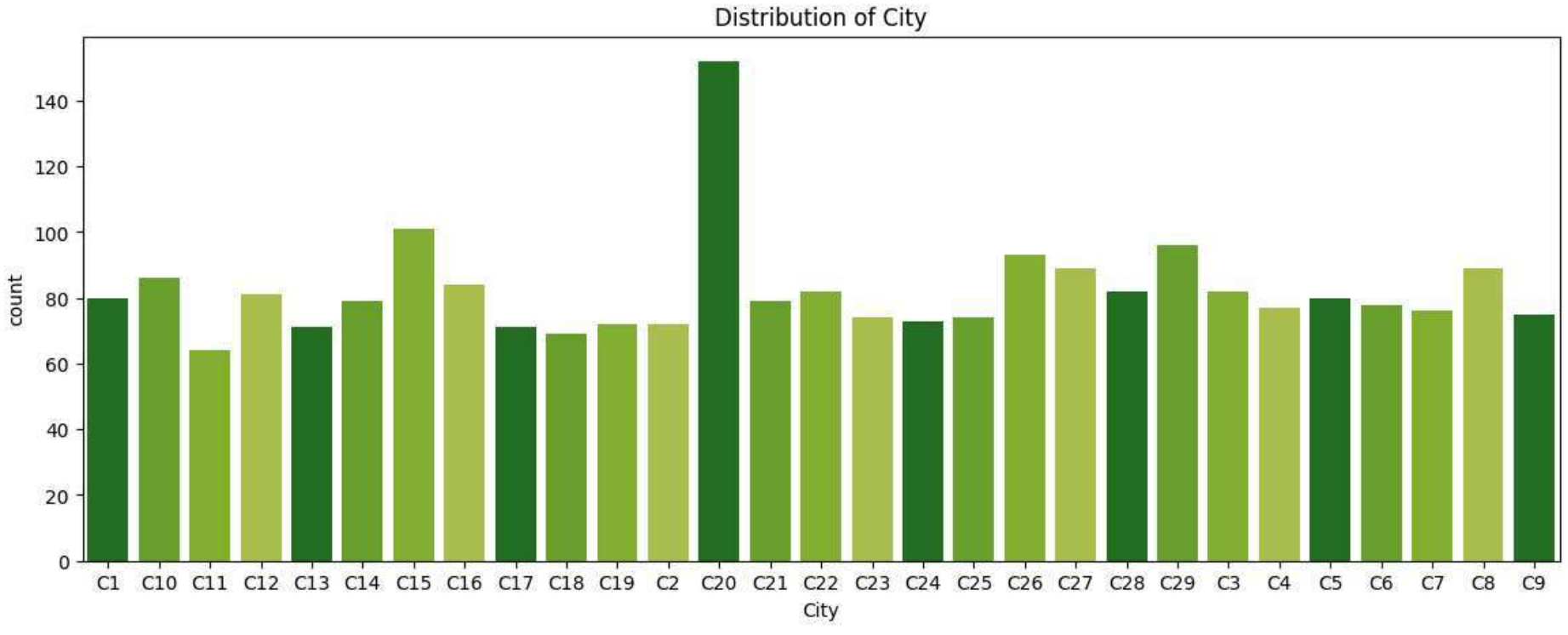
In [ ]:
```
# Pie Plots for Categorical features
plt.figure(figsize=(16,20))
for i, elem in enumerate(req_cat_col_plot):
  plt.subplot(4,3,i+1)
  labels = df.groupby(elem)[elem].count().index.categories
  plt.pie(df.groupby(elem)[elem].count().values, labels = labels, autopct = "%1.1f%%", colors=green_palette)
  plt.xlabel(elem, fontsize=14)

plt.suptitle("Pie Plots for Categorical features", fontsize = 18)
plt.tight_layout()
plt.show()
```





In [ ]:
```
plt.figure(figsize=(14,5))
sns.countplot(x=df['City'], palette = green_palette)
plt.title('Distribution of City')
```

Out[ ]:
```
Text(0.5, 1.0, 'Distribution of City')
```

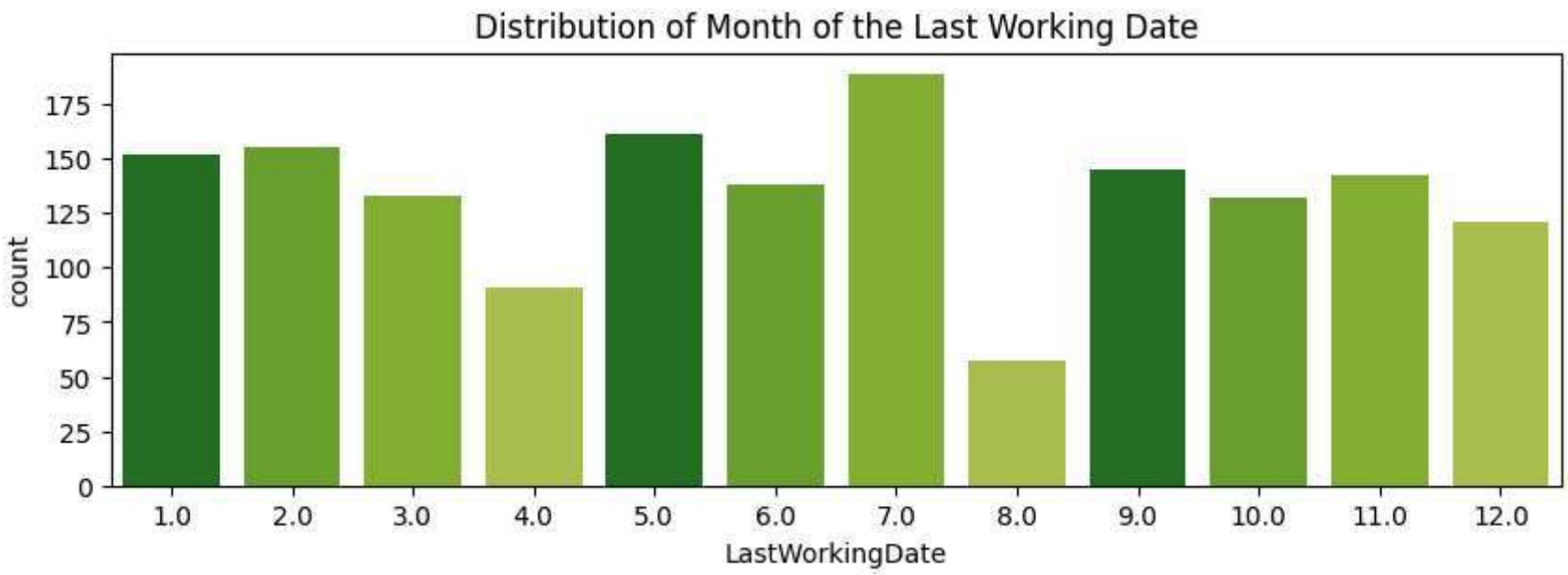


🔍 OBSERVATION 🔎

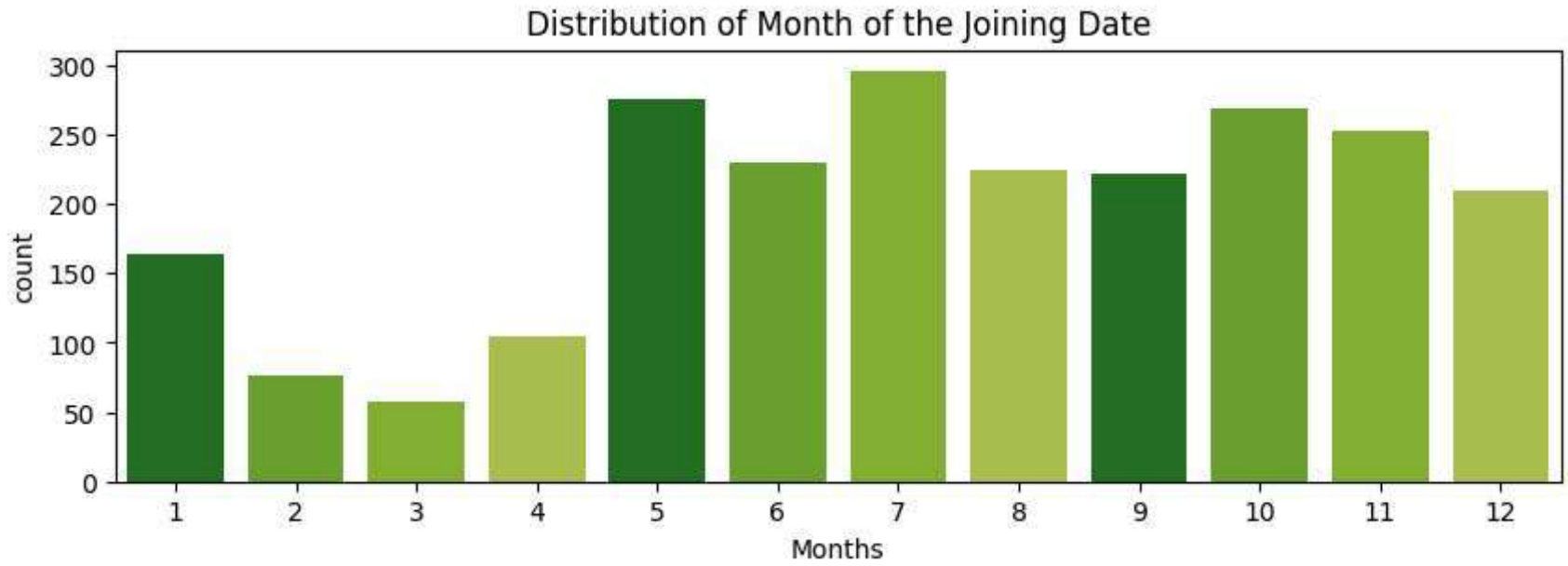
1. **Gender**:
   - The majority of drivers are **Male (58.97%)**.

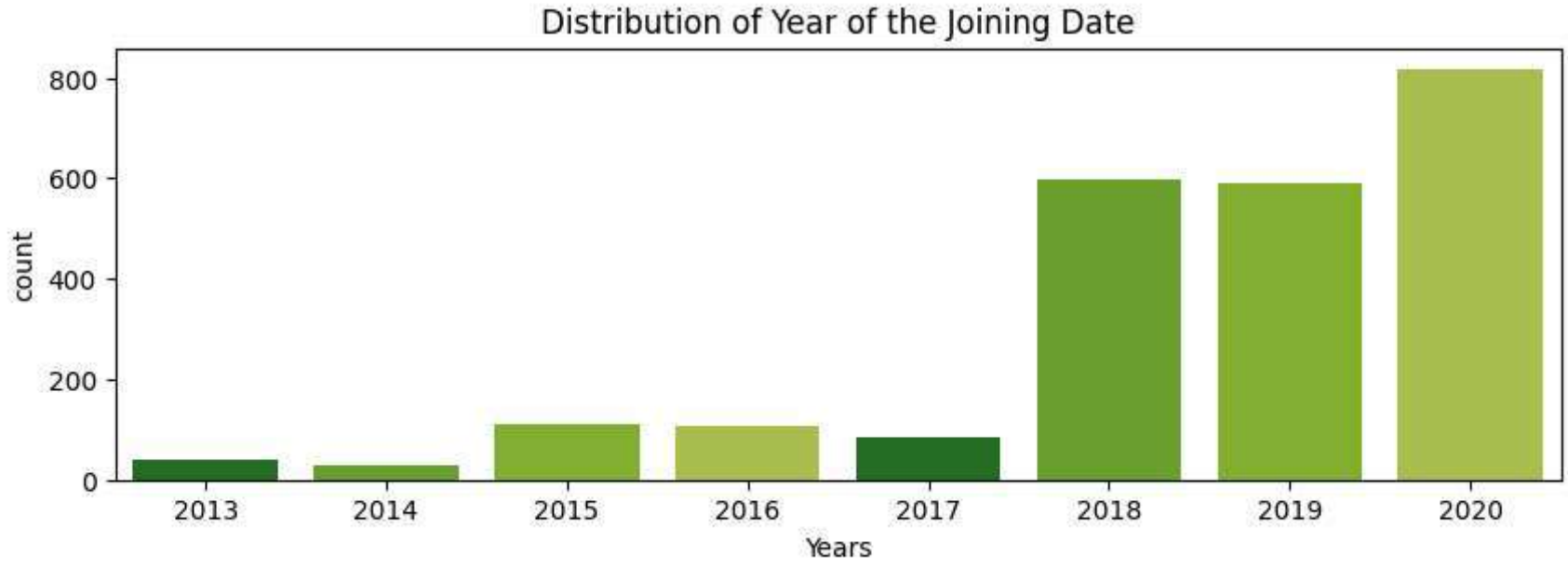
2. **City**:
    - The most common city is **C20 (6.38%)**.
3. **Education Level**:
    - The most frequent education level is **2 (33.68%)**, which could indicate a moderate educational background.
4. **Grade**:
    - The most prevalent grade is **Grade 2 (35.91%)**, representing a significant portion of drivers.
5. **Joining Designation**:
    - Most drivers started with **Joining Designation 1 (43.09%)**.
6. **Churn**:
    - The churn rate is high, with **67.87% of drivers leaving**.
7. **Rating Increase**:
    - A majority of drivers (**84.96%**) did not experience a rating increase.
8. **Income Increase**:
    - Almost all drivers (**98.19%**) did not receive an income increase.

In [ ]:
```
plt.figure(figsize=(10,3))
sns.countplot(x=df['LastWorkingDate'].dt.month, palette = green_palette)
plt.title('Distribution of Month of the Last Working Date')

plt.figure(figsize=(10,3))
sns.countplot(x=df['LastWorkingDate'].dt.year, palette = green_palette)
plt.title('Distribution of Year of the Last Working Date')

plt.show()
```



In [ ]:
```
plt.figure(figsize=(10,3))
sns.countplot(x=df['Dateofjoining'].dt.month, palette = green_palette)
plt.title('Distribution of Month of the Joining Date')
plt.xlabel('Months')

plt.figure(figsize=(10,3))
sns.countplot(x=df['Dateofjoining'].dt.year, palette = green_palette)
plt.title('Distribution of Year of the Joining Date')
plt.xlabel('Years')

plt.show()
```





In [ ]:
```
df.select_dtypes(include=['int64','float64'])
```

Out[ ]:

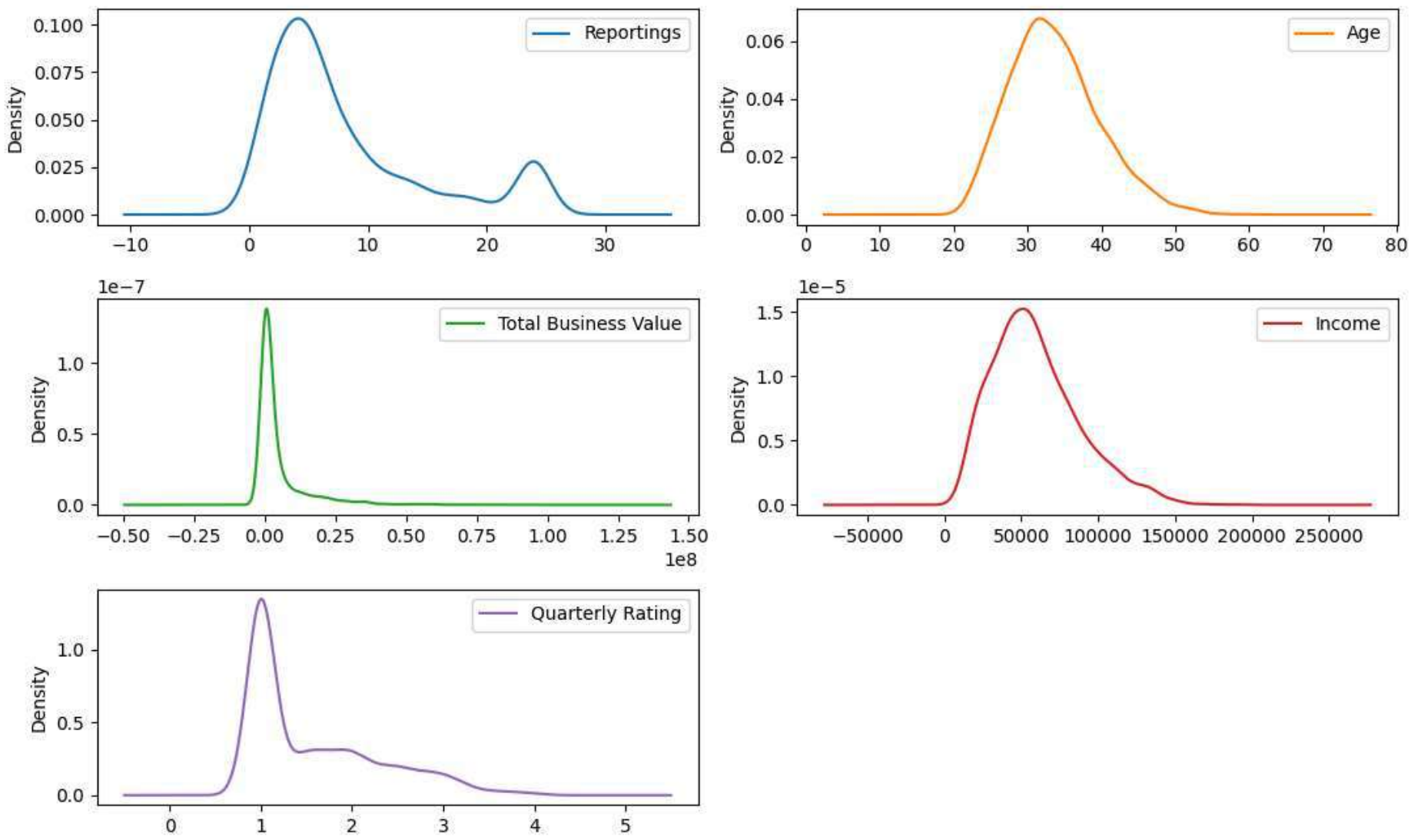
	Reportings	Driver_ID	Age	Total Business Value	Income	Quarterly Rating
0	3	1	28	1715580	57387	2.0
1	2	2	31	0	67016	1.0
2	5	4	43	350000	65603	1.0
3	3	5	29	120360	46368	1.0
4	5	6	31	1265000	78728	1.6
...	...	...	...	...	...	...
2376	24	2784	34	21748820	82815	2.6
2377	3	2785	34	0	12105	1.0
2378	9	2786	45	2815090	35370	1.7
2379	6	2787	28	977830	69498	1.5
2380	7	2788	30	2298240	70254	2.3

2381 rows × 6 columns

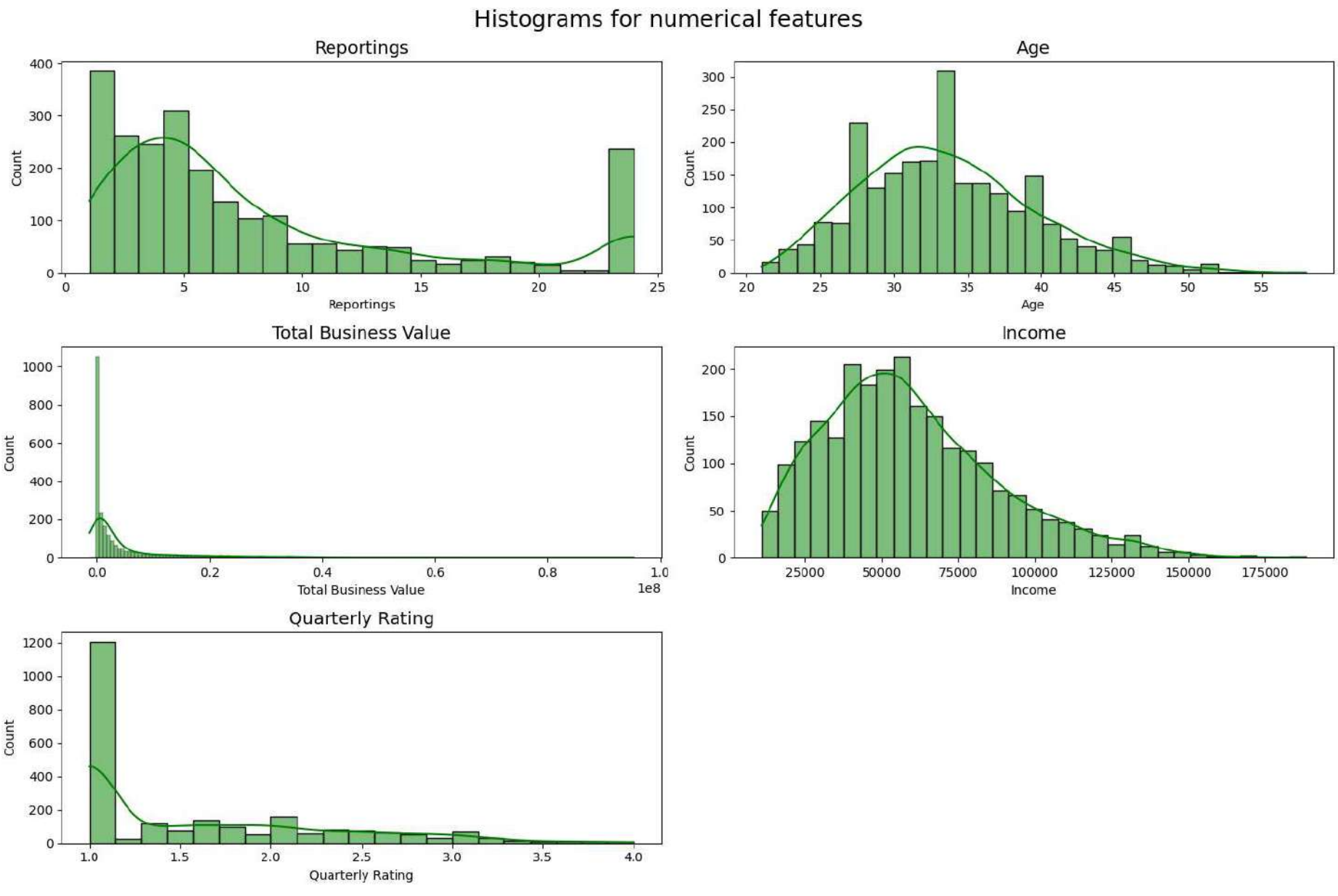
In [ ]:
```
# Selecting the categorical columns
numerical_df = df.select_dtypes(include=['int64','float64'])
numerical_df_req = numerical_df[['Reportings', 'Age', 'Total Business Value', 'Income', 'Quarterly Rating']] #numerical_df[numeri
numerical_df_req.columns
```

Out[ ]:
```
Index(['Reportings', 'Age', 'Total Business Value', 'Income',
       'Quarterly Rating'],
      dtype='object')
```

In [ ]:
```
# Density plot for numerical columns
plt.rcParams["figure.figsize"] = [11,7]
numerical_df_req.plot(kind="density", subplots = True, layout = (3,2), sharex = False)
plt.suptitle("Density plot for numerical features", fontsize = 18)
plt.tight_layout()
plt.show()
```



In [ ]:
```
# Histograms for numerical columns
numerical_cols = numerical_df_req.columns

plt.figure(figsize=(15,10))
for i, elem in enumerate(numerical_cols):
  plt.subplot(3,2,i+1)
  sns.histplot(df[elem], kde=True, color='green')
  plt.title(elem, fontsize=14)

plt.suptitle("Histograms for numerical features", fontsize = 18)
plt.tight_layout()
plt.show()
```



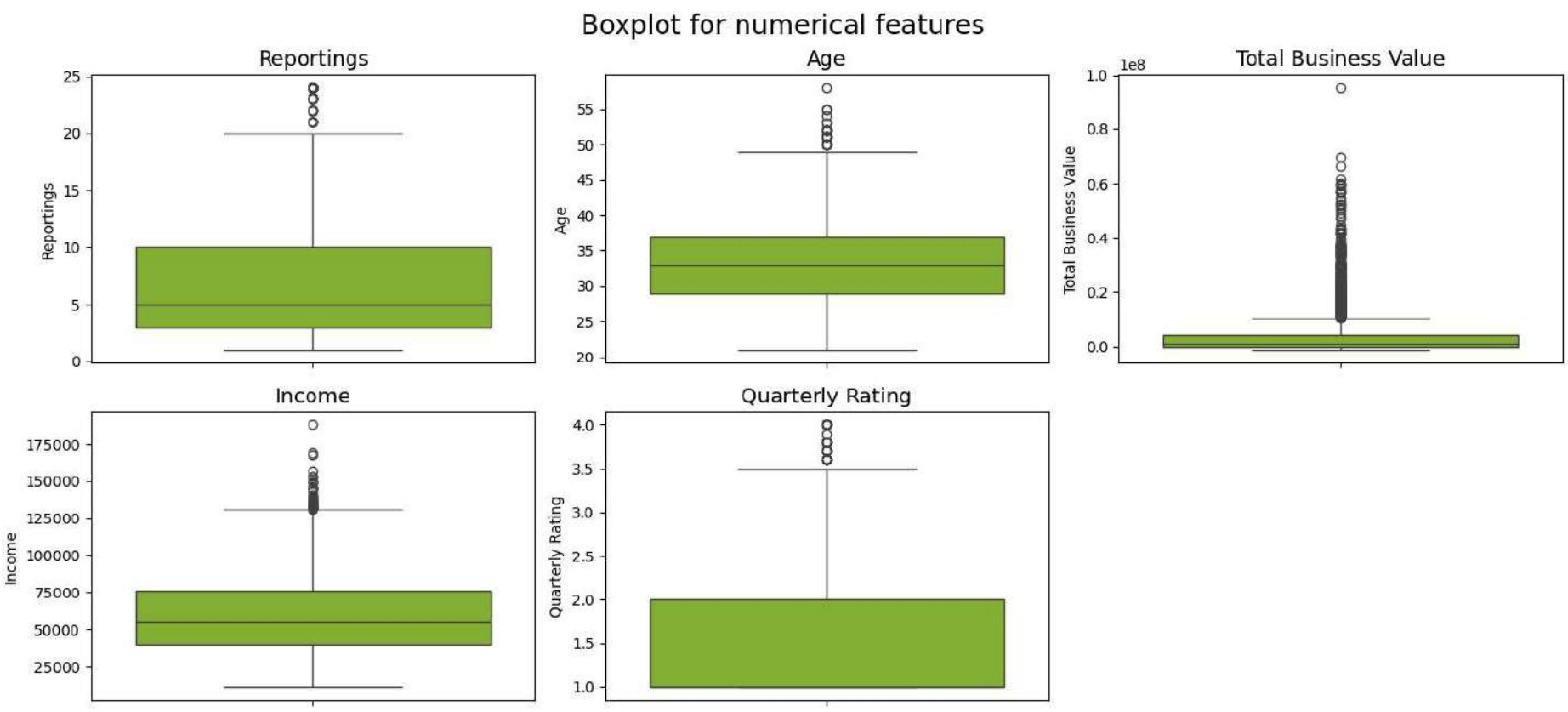
In [ ]:
```
# Skewness Coefficient:
print("Skewness Coefficient")
print("-" * 20)
print(numerical_df_req.skew().round(4))
```

```
Skewness Coefficient
--------------------
Reportings              1.2970
Age                     0.5391
Total Business Value    3.3613
Income                  0.7795
Quarterly Rating        1.0882
dtype: float64
```

🔍 OBSERVATION 🔎

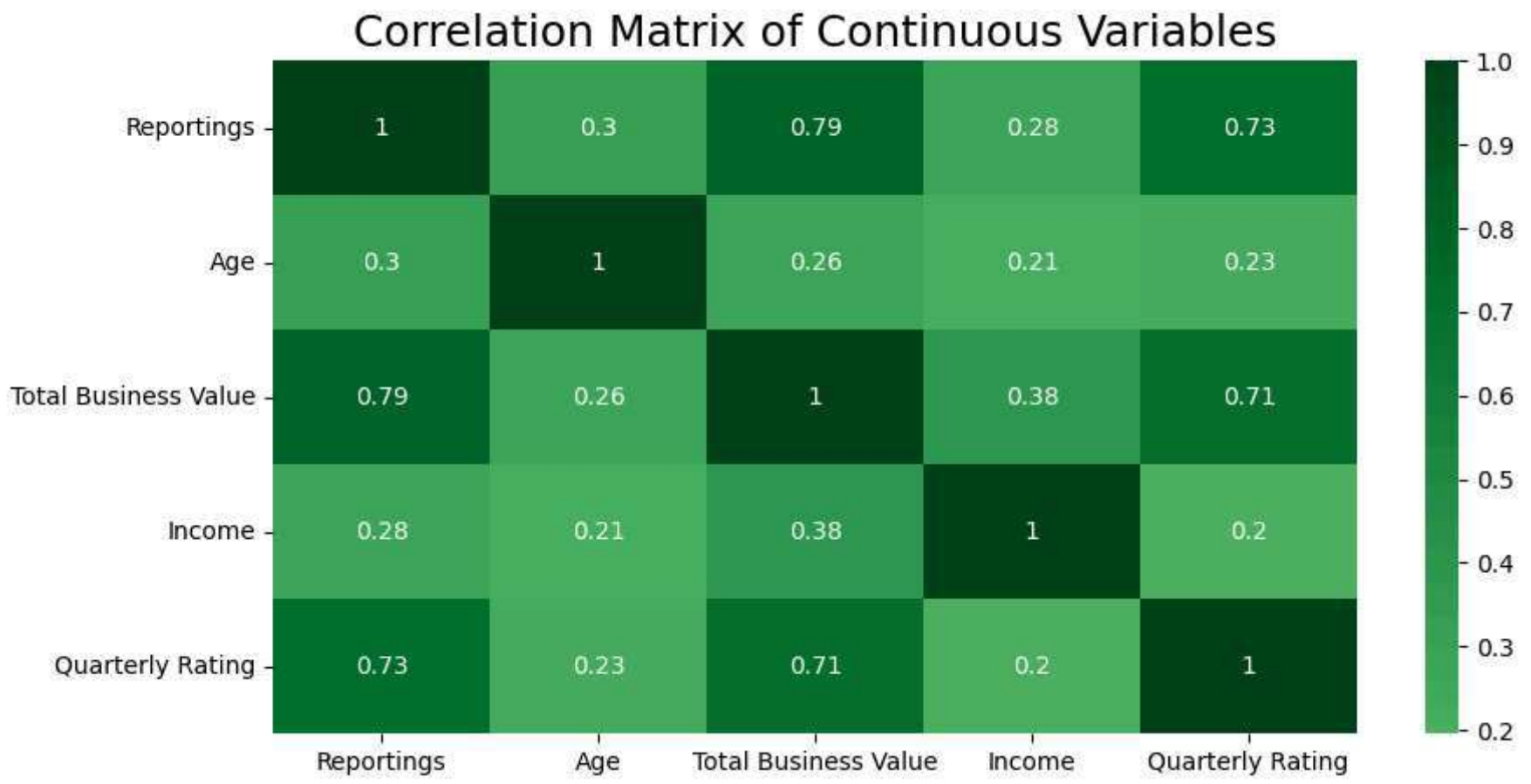
1. **Reportings**: Skewed positively with a skewness of **1.2970**, indicating a longer tail on the right side.
2. **Age**: Slightly positively skewed with a skewness of **0.5391**, showing a near-normal distribution.
3. **Total Business Value**: Highly positively skewed with a skewness of **3.3613**, suggesting a strong concentration of lower values.
4. **Income**: Positively skewed with a skewness of **0.7795**, indicating a moderate tail of higher income values.
5. **Quarterly Rating**: Positively skewed with a skewness of **1.0882**, reflecting more drivers with lower ratings.

In [ ]:
```python
# Box plots for numerical columns
green = ['#8dc71e']

plt.figure(figsize=(15, 10))
for i, col in enumerate(numerical_cols):
    plt.subplot(3, 3, i+1)
    sns.boxplot(df[col],palette = green)
    plt.title(col, fontsize=14)

plt.suptitle("Boxplot for numerical features", fontsize = 18)
plt.tight_layout()
plt.show()
```

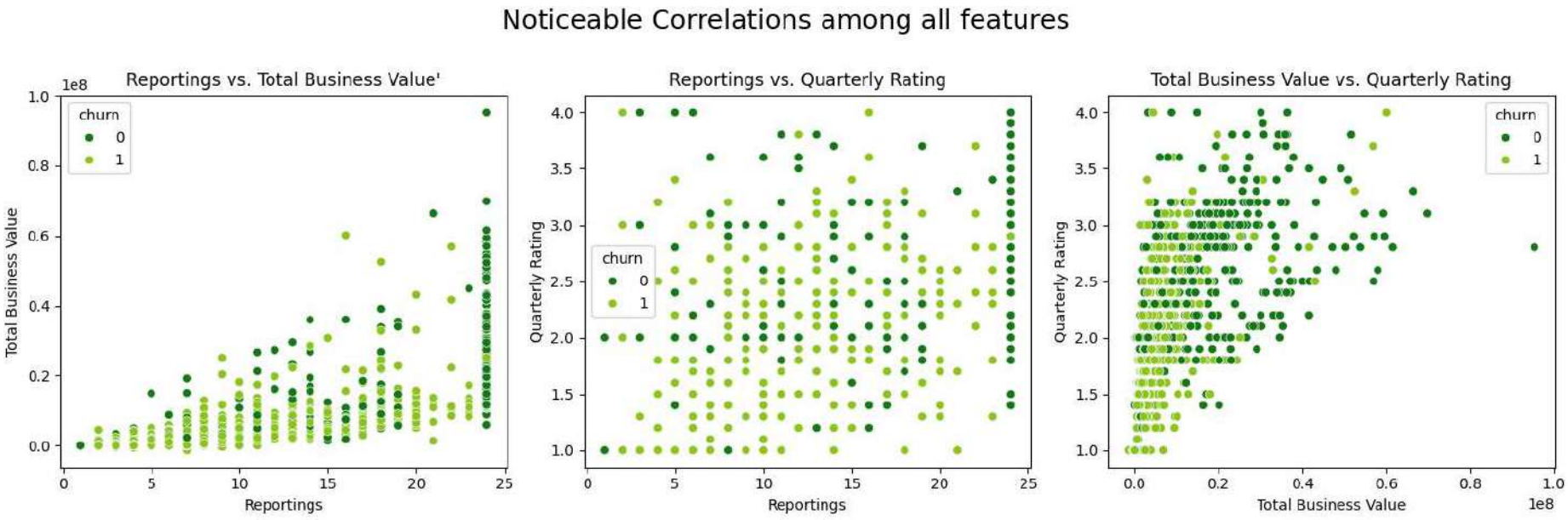


Multivariate Analysis

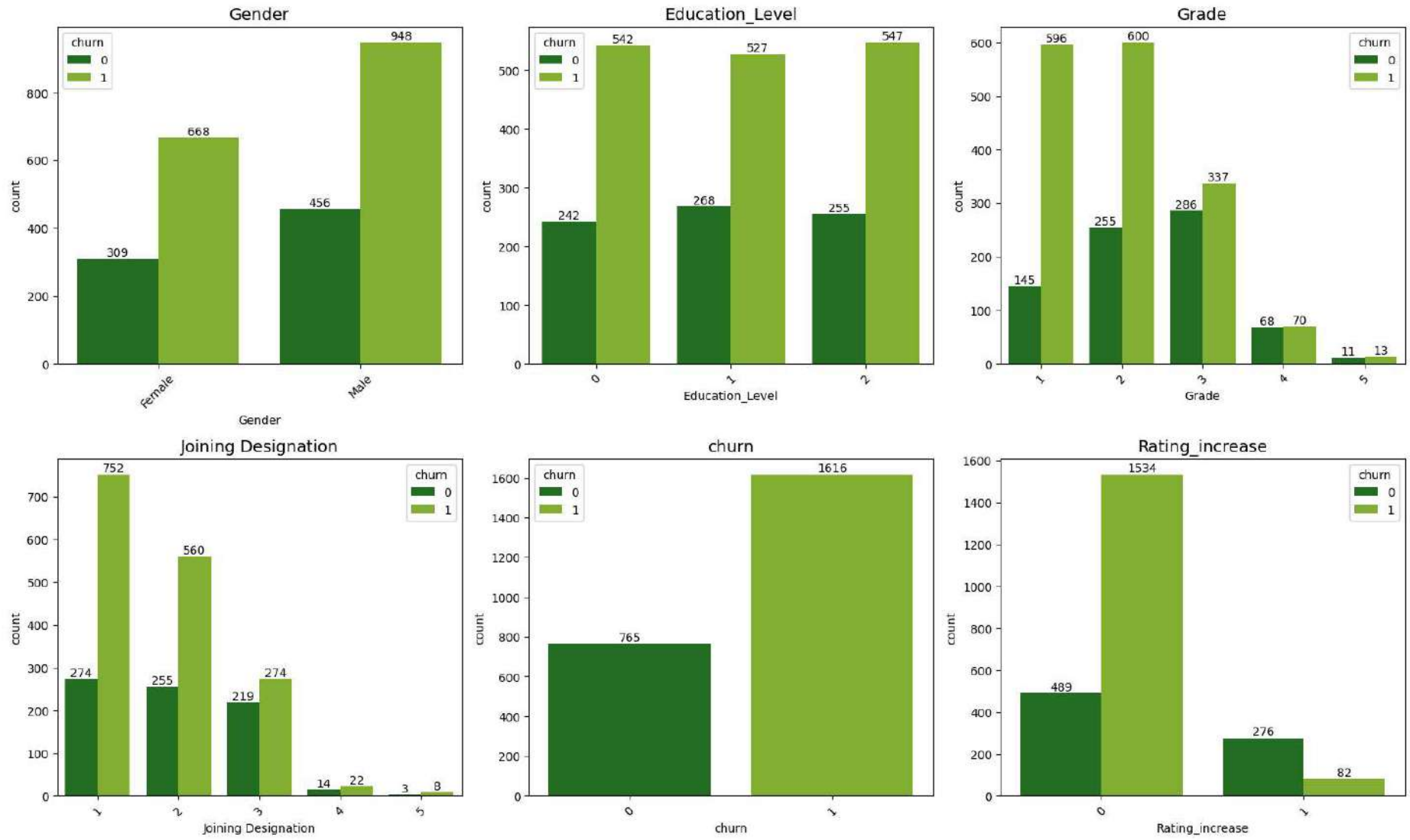
In [ ]:
```python
# Correlation Matrix of Continuous Variables
corr_matrix = numerical_df[numerical_cols].corr()
plt.figure(figsize=(10, 5))
sns.heatmap(corr_matrix, annot=True, cmap='Greens', center=0)
plt.title('Correlation Matrix of Continuous Variables', fontsize = 18)
plt.yticks(rotation=360)
plt.show()
```

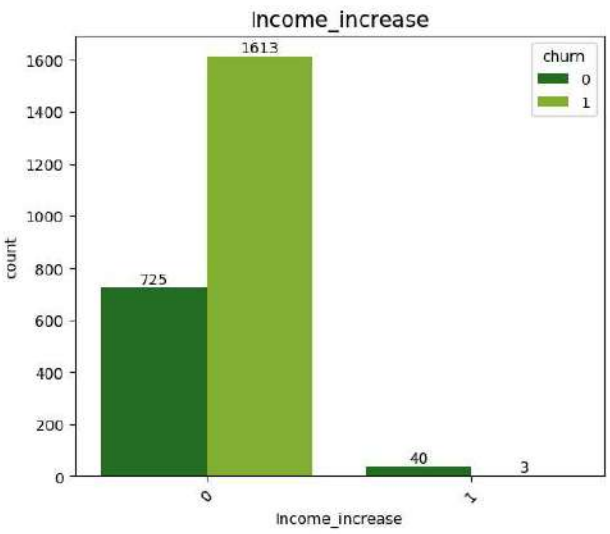


In [ ]:
```
req_palette = ['#187c19', '#8dc71e']
plt.figure(figsize=(15, 5))

plt.subplot(1, 3, 1)
sns.scatterplot(x='Reportings', y='Total Business Value', hue='churn', data=df, palette=req_palette)
plt.title("Reportings vs. Total Business Value'")

plt.subplot(1, 3, 2)
sns.scatterplot(x='Reportings', y='Quarterly Rating', hue='churn', data=df, palette=req_palette)
plt.title("Reportings vs. Quarterly Rating")

plt.subplot(1, 3, 3)
sns.scatterplot(x='Total Business Value', y='Quarterly Rating', hue='churn', data=df, palette=req_palette)
plt.title("Total Business Value vs. Quarterly Rating")

plt.suptitle("Noticeable Correlations among all features", fontsize = 18)
plt.tight_layout()
plt.show()
```



In [ ]:
```
# Count Plots for Categorical features
req_cat_col_plot = ['Gender', 'Education_Level', 'Grade', 'Joining Designation','churn', 'Rating_increase', 'Income_increase']

plt.figure(figsize=(17,20))
for i, elem in enumerate(req_cat_col_plot):
  plt.subplot(4,3,i+1)
  label = sns.countplot(data = df, x = elem, hue='churn', palette = req_palette)
  for i in label.containers:
    label.bar_label(i)

  plt.xticks(rotation = 45)
  plt.ylabel('count')
  plt.title(elem, fontsize=14)

#plt.suptitle("Count Plots for Categorical features", fontsize = 18)
plt.tight_layout()
plt.show()
```

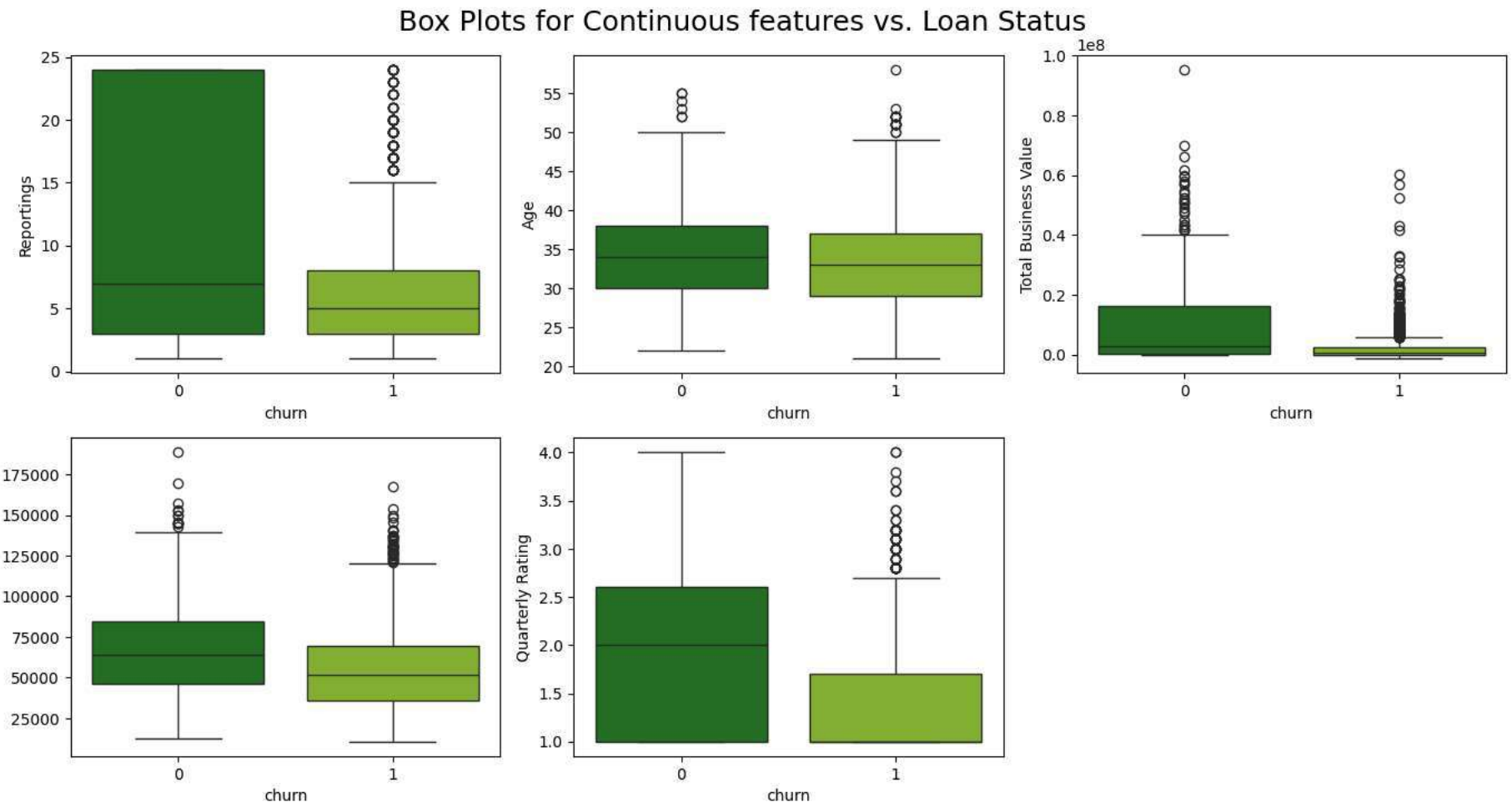




In [ ]:
```
# Comparisons of Median values of all numerical features by loan status
print("Comparisons of Median values of all numerical features by loan status")
print("-" * 72)

for elem in numerical_cols:
  print(f"Column Name: {elem}")
  print(df.groupby('churn')[elem].median())
  print("_" * 35)
  print()
```

```
Comparisons of Median values of all numerical features by loan status
----------------------------------------------------------------------
Column Name: Reportings
churn
0    7.0
1    5.0
Name: Reportings, dtype: float64
_____________________________________

Column Name: Age
churn
0    34.0
1    33.0
Name: Age, dtype: float64
_____________________________________

Column Name: Total Business Value
churn
0    2636210.0
1     465025.0
Name: Total Business Value, dtype: float64
_____________________________________

Column Name: Income
churn
0    64154.0
1    51630.0
Name: Income, dtype: float64
_____________________________________

Column Name: Quarterly Rating
churn
0    2.0
1    1.0
Name: Quarterly Rating, dtype: float64
_____________________________________
```

In [ ]:
```python
# Box Plots for Continuous features vs. Loan Status

plt.figure(figsize=(14, 14))
for i, col in enumerate(numerical_cols):
    plt.subplot(4, 3, i+1)
    sns.boxplot(x='churn', y=col, data=df, palette = req_palette)

plt.suptitle("Box Plots for Continuous features vs. Loan Status", fontsize = 18)
plt.tight_layout()
plt.show()
```



🔍 OBSERVATION 🔎

1. **Reportings**: Non-churned drivers have a higher median value (**7.0**) compared to churned drivers (**5.0**), indicating more reportings are linked to retention.
2. **Age**: Non-churned drivers have a slightly higher median age (**34**) compared to churned drivers (**33**), suggesting age might have a minor influence.
3. **Total Business Value**: Non-churned drivers have a significantly higher median total business value (**2,636,210**) than churned drivers (**465,025**), showing a strong correlation with retention.
4. **Income**: Non-churned drivers have a higher median income (**64,154**) compared to churned drivers (**51,630**), suggesting better pay may reduce churn.

5. **Quarterly Rating**: Non-churned drivers have a higher median quarterly rating (**2.0**) compared to churned drivers (**1.0**), indicating performance rating impacts churn behavior.

Churn Analysis

In [ ]:
```
# Feature Engineering
df_churned = df[df['churn'] == 1]
```

In [ ]:
```
df_churned['tenure_days'] = (df_churned['LastWorkingDate'] - df_churned['Dateofjoining']).apply(lambda x: x.days)
```

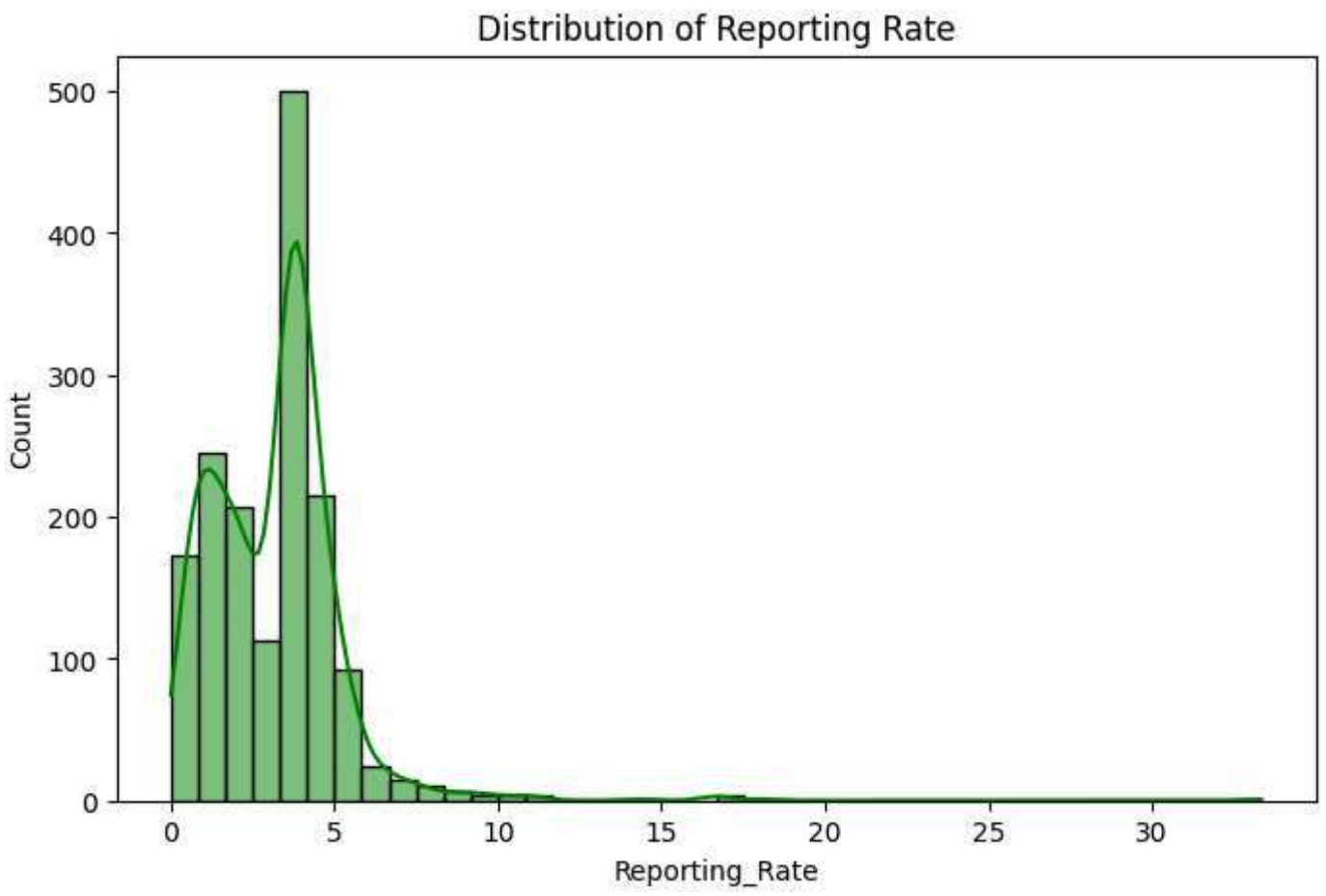
In [ ]:
```
df_churned['Reporting_Rate'] = (df_churned['Reportings'] / df_churned['tenure_days'].replace(0, pd.NA)) * 100
df_churned['Reporting_Rate'] = df_churned['Reporting_Rate'].fillna(0)
df_churned['Reporting_Rate'] = df_churned['Reporting_Rate'].round(2)
```

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

Out[ ]:

	Reportings	Driver_ID	Age	Gender	City	Education_Level	Dateofjoining	LastWorkingDate	Grade	Total Business Value	Income	Joining Designation	Quarterly Rating	churn
0	3	1	28	Male	C23	2	2018-12-24	2019-03-11	1	1715580	57387	1	2.0	1
2	5	4	43	Male	C13	2	2019-12-07	2020-04-27	2	350000	65603	2	1.0	1
3	3	5	29	Male	C9	0	2019-01-09	2019-03-07	1	120360	46368	1	1.0	1
5	3	8	34	Male	C2	0	2020-09-19	2020-11-15	3	0	70656	3	1.0	1
7	6	12	35	Male	C23	2	2019-06-29	2019-12-21	1	2607180	28116	1	2.5	1

In [ ]:
```
# Histogram
plt.figure(figsize=(8, 5))
sns.histplot(data=df_churned, x ='Reporting_Rate', bins=40, kde=True, color='green')
plt.title("Distribution of Reporting Rate")
plt.show()
```



In [ ]:
```
df_churned['Reporting_Rate'].describe()
```

Out[ ]:

	Reporting_Rate
count	1616.000000
mean	3.155736
std	2.048271
min	0.000000
25%	1.600000
50%	3.470000
75%	4.092500
max	33.330000

dtype: float64

🔍 **OBSERVATION** 🔎

- The mean reporting rate for churned drivers is 3.16, indicating that, on average, churned drivers had moderate reporting rates before leaving.
- The median reporting rate is 3.47, which is close to the mean, indicating a relatively symmetric distribution for most churned drivers.
- The reporting rates vary widely, from a minimum of 0.00 to a maximum of 33.33, suggesting that while some churned drivers had no reports, a few had exceptionally high reporting rates.

Data preparation for modeling 🔢

In [ ]:
```
# Creating a deep copy
df_model = df.copy()
```

In [ ]:
```
df_model.drop(columns=['Driver_ID', 'LastWorkingDate'], inplace=True)
df_model['Month_of_joining'] = df_model['Dateofjoining'].dt.month
df_model['Year_of_joining']= df_model['Dateofjoining'].dt.year
df_model.drop(columns='Dateofjoining', inplace=True)
```

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

Out[ ]:

	Reportings	Age	Gender	City	Education_Level	Grade	Total Business Value	Income	Joining Designation	Quarterly Rating	churn	Rating_increase	Income_increase	Month_o
0	3	28	Male	C23	2	1	1715580	57387	1	2.0	1	0	0	
1	2	31	Male	C7	2	2	0	67016	2	1.0	0	0	0	
2	5	43	Male	C13	2	2	350000	65603	2	1.0	1	0	0	
3	3	29	Male	C9	0	1	120360	46368	1	1.0	1	0	0	
4	5	31	Female	C11	1	3	1265000	78728	3	1.6	0	1	0	

In [ ]:
```
df_model.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2381 entries, 0 to 2380
Data columns (total 15 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   Reportings            2381 non-null   int64
 1   Age                   2381 non-null   int64
 2   Gender                2381 non-null   category
 3   City                  2381 non-null   category
 4   Education_Level       2381 non-null   category
 5   Grade                 2381 non-null   category
 6   Total Business Value  2381 non-null   int64
 7   Income                2381 non-null   int64
 8   Joining Designation   2381 non-null   category
 9   Quarterly Rating      2381 non-null   float64
 10  churn                 2381 non-null   category
 11  Rating_increase       2381 non-null   category
 12  Income_increase       2381 non-null   category
 13  Month_of_joining      2381 non-null   int32
 14  Year_of_joining       2381 non-null   int32
dtypes: category(8), float64(1), int32(2), int64(4)
memory usage: 132.6 KB
```

In [ ]:
```
df_model_category_columns = df_model.select_dtypes(include='category')
df_model_category_columns.head()
```

Out[ ]:

	Gender	City	Education_Level	Grade	Joining Designation	churn	Rating_increase	Income_increase
0	Male	C23	2	1	1	1	0	0
1	Male	C7	2	2	2	0	0	0
2	Male	C13	2	2	2	1	0	0
3	Male	C9	0	1	1	1	0	0
4	Female	C11	1	3	3	0	1	0

Label Encoding

In [ ]:
```
# Convert the following columns directly to int since they are ordinal
columns = ['Education_Level', 'Grade', 'Joining Designation', 'churn', 'Rating_increase', 'Income_increase']

for cols in columns:
  df_model[cols] = df_model[cols].astype('int')
```

In [ ]:
```
# Encode (Male = 0, Female = 1)
df_model['Gender'] = df_model['Gender'].map({"Male": 0, "Female": 1})
df_model['Gender'] = df_model['Gender'].astype('int')
```

In [ ]:
```
df_model['City'].nunique(), df_model['City'].unique()
```

Out[ ]:
```
(29,
 ['C23', 'C7', 'C13', 'C9', 'C11', ..., 'C4', 'C3', 'C16', 'C22', 'C12']
 Length: 29
 Categories (29, object): ['C1', 'C10', 'C11', 'C12', ..., 'C6', 'C7', 'C8', 'C9'])
```

One Hot Encodeing

In [ ]:
```
# Perform one-hot encoding for 'City' column and drop the first category
df_encoded = pd.get_dummies(df_model, columns=['City'], prefix='City', drop_first=True)*1
```

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

Out[ ]:

	Reportings	Age	Gender	Education_Level	Grade	Total Business Value	Income	Joining Designation	Quarterly Rating	churn	Rating_increase	Income_increase	Month_of_joini
0	3	28	0	2	1	1715580	57387	1	2.0	1	0	0	
1	2	31	0	2	2	0	67016	2	1.0	0	0	0	
2	5	43	0	2	2	350000	65603	2	1.0	1	0	0	
3	3	29	0	0	1	120360	46368	1	1.0	1	0	0	
4	5	31	1	1	3	1265000	78728	3	1.6	0	1	0	

In [ ]:
```
df_encoded.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2381 entries, 0 to 2380
Data columns (total 42 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   Reportings            2381 non-null   int64
 1   Age                   2381 non-null   int64
 2   Gender                2381 non-null   int64
 3   Education_Level       2381 non-null   int64
 4   Grade                 2381 non-null   int64
 5   Total Business Value  2381 non-null   int64
 6   Income                2381 non-null   int64
 7   Joining Designation   2381 non-null   int64
 8   Quarterly Rating      2381 non-null   float64
 9   churn                 2381 non-null   int64
 10  Rating_increase       2381 non-null   int64
 11  Income_increase       2381 non-null   int64
 12  Month_of_joining      2381 non-null   int32
 13  Year_of_joining       2381 non-null   int32
 14  City_C10              2381 non-null   int64
 15  City_C11              2381 non-null   int64
 16  City_C12              2381 non-null   int64
 17  City_C13              2381 non-null   int64
 18  City_C14              2381 non-null   int64
 19  City_C15              2381 non-null   int64
 20  City_C16              2381 non-null   int64
 21  City_C17              2381 non-null   int64
 22  City_C18              2381 non-null   int64
 23  City_C19              2381 non-null   int64
 24  City_C2               2381 non-null   int64
 25  City_C20              2381 non-null   int64
 26  City_C21              2381 non-null   int64
 27  City_C22              2381 non-null   int64
 28  City_C23              2381 non-null   int64
 29  City_C24              2381 non-null   int64
 30  City_C25              2381 non-null   int64
 31  City_C26              2381 non-null   int64
 32  City_C27              2381 non-null   int64
 33  City_C28              2381 non-null   int64
 34  City_C29              2381 non-null   int64
 35  City_C3               2381 non-null   int64
 36  City_C4               2381 non-null   int64
 37  City_C5               2381 non-null   int64
 38  City_C6               2381 non-null   int64
 39  City_C7               2381 non-null   int64
 40  City_C8               2381 non-null   int64
 41  City_C9               2381 non-null   int64
dtypes: float64(1), int32(2), int64(39)
memory usage: 762.8 KB
```

Train Test Split

In [ ]:
```
# Lets split the data into Independent feature and dependent feature
y = df_encoded['churn']
X = df_encoded.drop('churn', axis = 1)
X.shape, y.shape
```

Out[ ]:
```
((2381, 41), (2381,))
```

In [ ]:
```
# Lets split the data into train and test
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)
```

In [ ]:
```
# After_train_test_split
# X_train | y_train
# X_test | y_test
```

Scaling

In general:

- Bagging and Boosting with Decision Trees: No scaling required.
- Other weak learners (KNN, SVM, etc.) in Bagging/Boosting: Scaling is needed.

Best Practices: For Tree-Based Methods (Bagging/Boosting):

- Scaling is not necessary, but you can still standardize or normalize if you are preprocessing the dataset for consistency across models.

In [ ]:
```
from sklearn.preprocessing import MinMaxScaler
scaler = MinMaxScaler()
X_train_scaled = scaler.fit_transform(X_train)
X_test_scaled = scaler.transform(X_test)

X_train_scaled = pd.DataFrame(X_train_scaled, columns=X.columns)
X_test_scaled = pd.DataFrame(X_test_scaled, columns=X.columns)
```

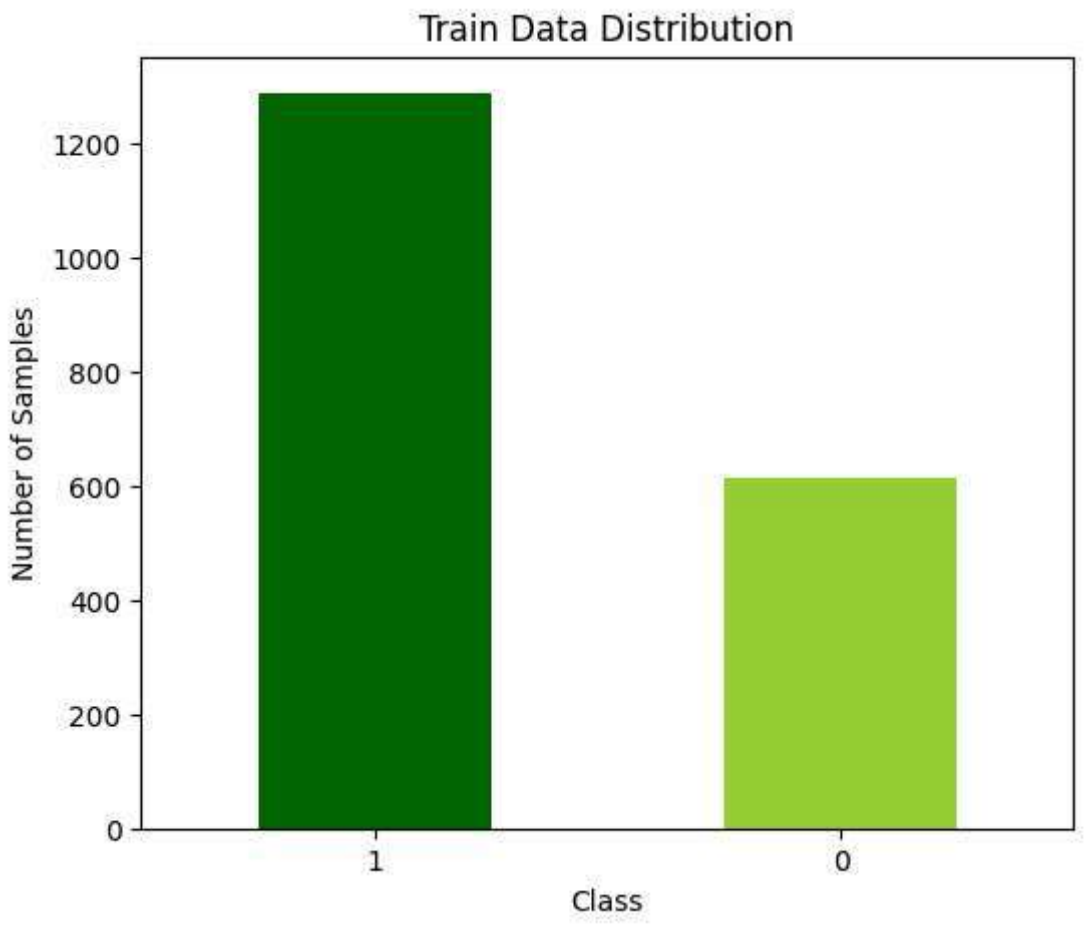
In [ ]:
```
# After_train_test_split       After Scaling
# X_train | y_train  -->   X_train_scaled | y_train
# X_test | y_test    -->   X_test_scaled | y_test
```

Handling Class Imbalance

In [ ]:
```
print(((y_train.value_counts(normalize=True)) * 100).round(2))
```

```
churn
1    67.7
0    32.3
Name: proportion, dtype: float64
```

In [ ]:
```
plt.figure(figsize=(6, 5))
y_train.value_counts().plot(kind='bar', color=['darkgreen', 'yellowgreen'])
plt.xlabel('Class')
plt.ylabel('Number of Samples')
plt.title('Train Data Distribution')
plt.xticks(rotation=0)
plt.show()
```



SMOTE

In [ ]:
```
from imblearn.over_sampling import SMOTE

# Create an instance of SMOTE
smote = SMOTE()

# Perform SMOTE on the training data
```

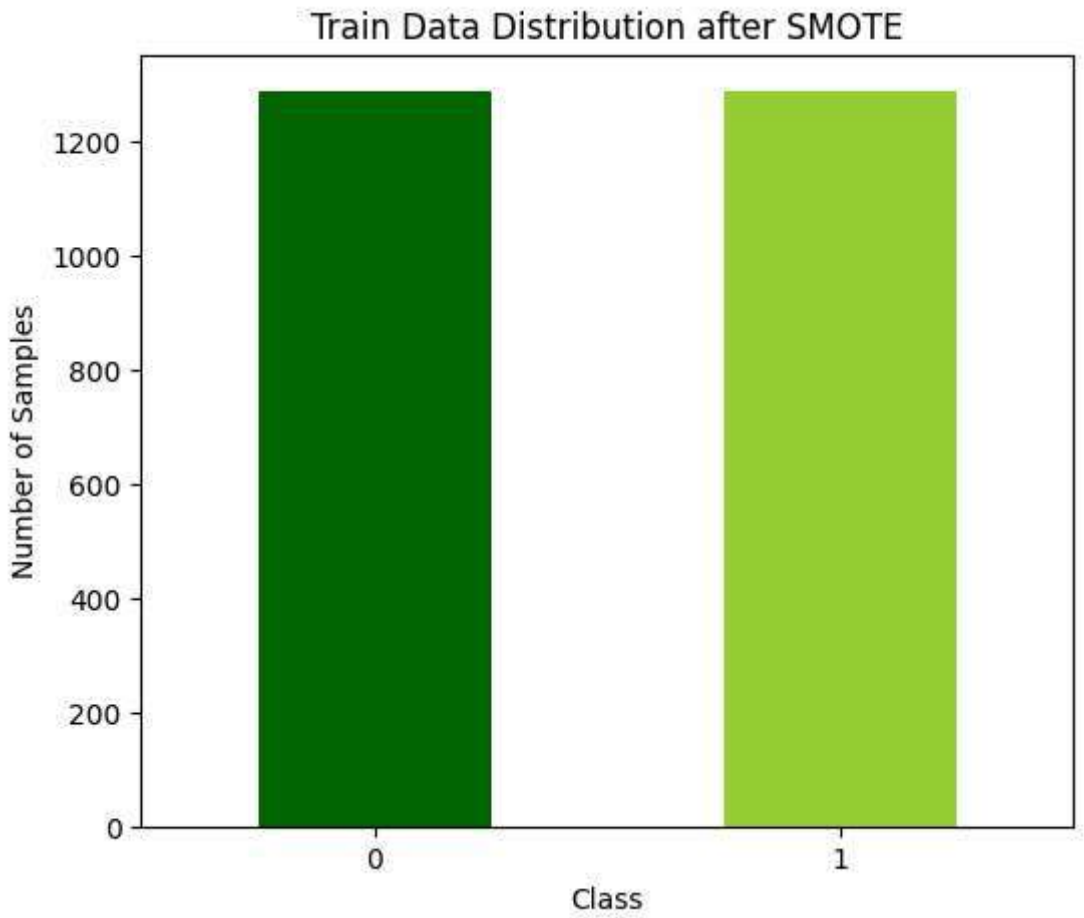
```
# Oversample the training data using SMOTE
X_train_balanced, y_train_balanced  = smote.fit_resample(X_train_scaled, y_train)
```

In [ ]:
```
# After_train_test_split       After Scaling                    After SMOTE
# X_train | y_train  -->   X_train_scaled | y_train   -->   X_train_balanced | y_train_balanced
# X_test | y_test    -->   X_test_scaled | y_test     -->   X_test_scaled | y_test
```

In [ ]:
```
print(((y_train_balanced.value_counts(normalize=True)) * 100).round(2))
```

```
churn
0    50.0
1    50.0
Name: proportion, dtype: float64
```

In [ ]:
```
plt.figure(figsize=(6, 5))
y_train_balanced.value_counts().plot(kind='bar', color=['darkgreen', 'yellowgreen'])
plt.xlabel('Class')
plt.ylabel('Number of Samples')
plt.title('Train Data Distribution after SMOTE')
plt.xticks(rotation=0)
plt.show()
```



Model Building - Ensemble Learning 💻

Bagging Random Forest Classifier

What is Bagging in Ensemble Learning?

Bagging (Bootstrap Aggregating) is an ensemble learning technique that combines predictions from multiple models (base estimators) to improve overall performance and reduce variance. The key idea is:

- **Bootstrap Sampling:** Random subsets of data are sampled with replacement to train each model.
- **Aggregation:** Predictions are averaged (for regression) or voted (for classification) to create the final prediction. Example: Random Forest is a popular bagging algorithm that uses decision trees as base estimators.

Advantages:

- Reduces overfitting by combining weak learners.
- Improves model stability and accuracy.

Hyperparameter Tuning Using RandomizedSearchCV

Define hyperparameters to tune a bagging model (e.g., Random Forest).

These are the hyperparameters for a **Random Forest** model, which is a commonly used **Bagging algorithm**. Here's what each hyperparameter represents:

- `n_estimators` : The number of trees in the forest.
- `max_depth` : The maximum depth of each decision tree.
- `min_samples_split` : The minimum number of samples required to split an internal node.

- **`min_samples_leaf`** : The minimum number of samples required to be at a leaf node.
- **`bootstrap`** : Whether to use bootstrap sampling (random sampling with replacement) when building trees.

In [ ]:
```
# Hyperparameters for RandomizedSearchCV
param_dist = {
    'n_estimators': [50, 100, 200, 500],
    'max_depth': [None, 10, 20, 30],
    'min_samples_split': [2, 5, 10],
    'min_samples_leaf': [1, 2, 4],
    'bootstrap': [True, False]
}
```

In [ ]:
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import RandomizedSearchCV
import datetime as dt

# Define the model
rf = RandomForestClassifier(random_state=42)

# Perform RandomizedSearchCV
random_search = RandomizedSearchCV(
                                   estimator=rf,
                                   param_distributions=param_dist,
                                   n_iter=50,
                                   scoring='roc_auc',
                                   cv=5,
                                   verbose=2,
                                   random_state=42,
                                   n_jobs=-1
                                  )

# Fit the model
start = dt.datetime.now()
random_search.fit(X_train_balanced, y_train_balanced)
end = dt.datetime.now()

# Best parameters
print("Best Parameters:", random_search.best_params_)
print("Best cross-validation score achieved: ", random_search.best_score_)
print(f"Time taken for RandomizedSearchCV(fits) : {end - start}")
```

```
Fitting 5 folds for each of 50 candidates, totalling 250 fits
Best Parameters: {'n_estimators': 500, 'min_samples_split': 2, 'min_samples_leaf': 1, 'max_depth': None, 'bootstrap': False}
Best cross-validation score achieved:  0.9837144065674368
Time taken for RandomizedSearchCV(fits) : 0:03:10.553290
```

Train the Model with Best Hyperparameters

Use the best parameters from `RandomizedSearchCV` to train the model.

In [ ]:
```
# Train the best model
best_rf = random_search.best_estimator_
best_rf.fit(X_train_balanced, y_train_balanced)
```

Out[ ]:
```
RandomForestClassifier(bootstrap=False, n_estimators=500, random_state=42)
```

Model Score / Accuracy Measurement

Evaluate the model's accuracy on the training and testing datasets.

In [ ]:
```
# After_train_test_split       After Scaling                    After SMOTE
# X_train | y_train  -->   X_train_scaled | y_train   -->   X_train_balanced | y_train_balanced
# X_test | y_test    -->   X_test_scaled | y_test     -->   X_test_scaled | y_test
```

In [ ]:
```
# Accuracy on train and test sets
train_accuracy = best_rf.score(X_train_balanced, y_train_balanced)
test_accuracy = best_rf.score(X_test_scaled, y_test)

print(f"Train Accuracy: {train_accuracy:.2f}")
print(f"Test Accuracy: {test_accuracy:.2f}")
```

```
Train Accuracy: 1.00
Test Accuracy: 0.91
```

In [ ]:
```
# Make predictions on the test set
y_train_pred = best_rf.predict(X_train_balanced)
y_test_pred = best_rf.predict(X_test_scaled)

# Evaluate the model
# Accuracy on predictions of train and test sets
train_accuracy = accuracy_score(y_train_balanced, y_train_pred)
print(f"Training Accuracy: {train_accuracy:.2f}")

test_accuracy = accuracy_score(y_test, y_test_pred)
print(f"Test Accuracy: {test_accuracy:.2f}")
```

```
Training Accuracy: 1.00
Test Accuracy: 0.91
```

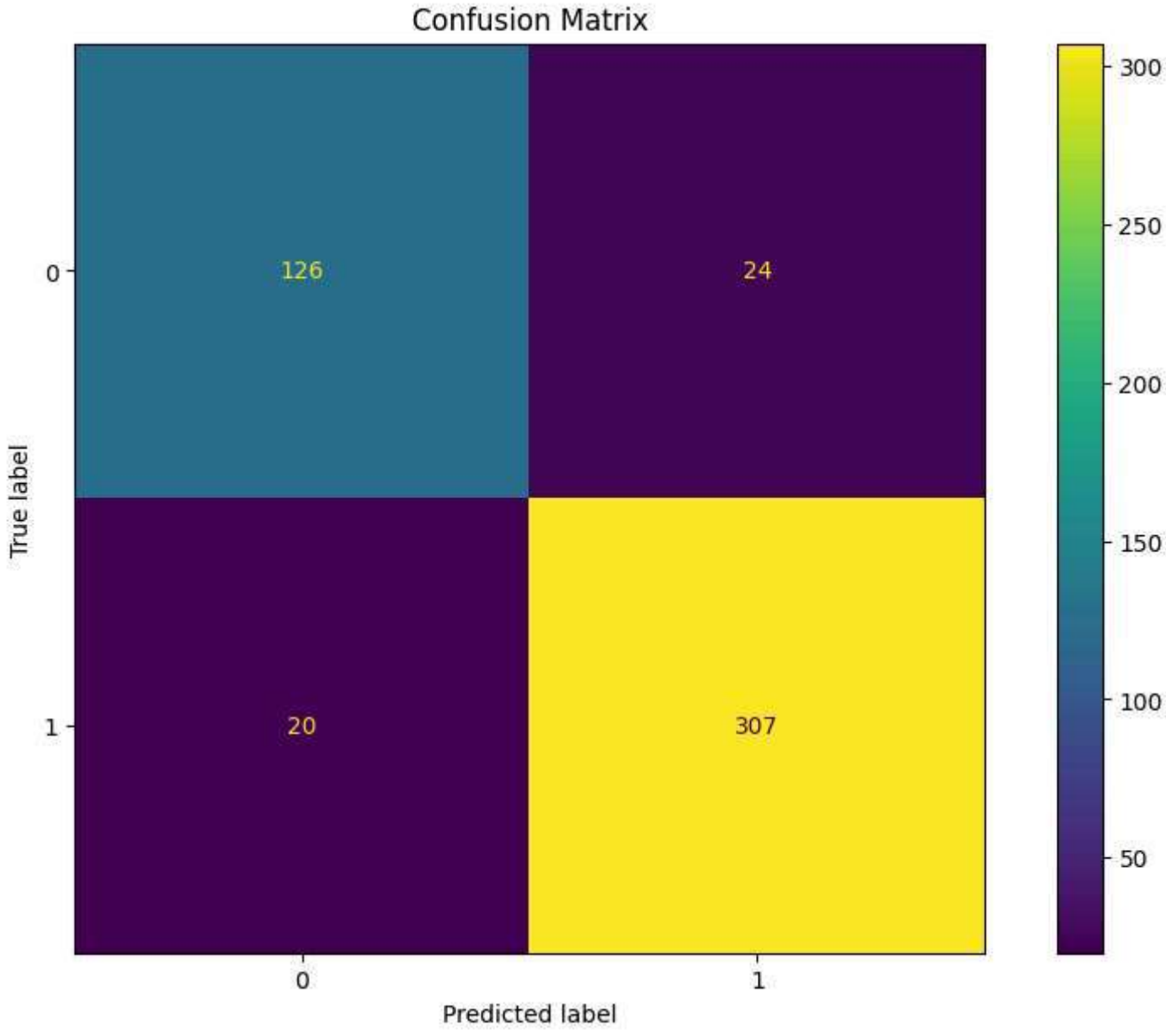
Confusion Matrix

Use the confusion matrix to evaluate classification performance.

In [ ]:
```
# After_train_test_split       After Scaling                    After SMOTE
# X_train | y_train  -->   X_train_scaled | y_train   -->   X_train_balanced | y_train_balanced
# X_test | y_test    -->   X_test_scaled | y_test     -->   X_test_scaled | y_test
```

In [ ]:
```
from sklearn.metrics import confusion_matrix, classification_report

# Predictions on test set
y_pred = best_rf.predict(X_test_scaled)

# Confusion Matrix
cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:\n", cm)

#plt.figure(figsize=(5, 4))
ConfusionMatrixDisplay(cm).plot()
plt.title('Confusion Matrix')
plt.show()

# Classification Report
print("Classification Report:\n", classification_report(y_test, y_pred))
```

```
Confusion Matrix:
[[126  24]
 [ 20 307]]
```



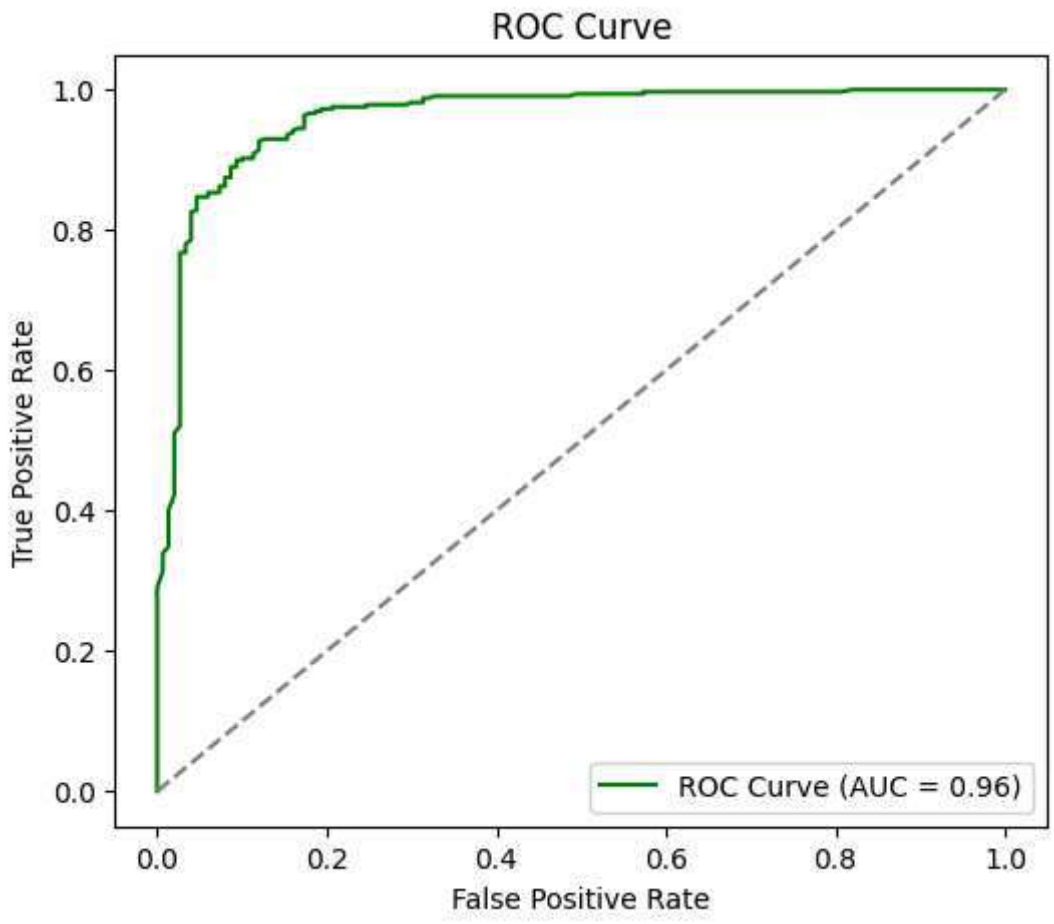

```
Classification Report:
              precision    recall  f1-score   support

           0       0.86      0.84      0.85       150
           1       0.93      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
```

ROC Curve & AUC

Evaluate the ROC Curve and calculate the AUC score.

In [ ]:
```
from sklearn.metrics import roc_curve, auc, roc_auc_score
import matplotlib.pyplot as plt

# Predict probabilities for ROC curve
y_prob = best_rf.predict_proba(X_test_scaled)[:, 1]
```

```
# ROC Curve
fpr, tpr, thresholds = roc_curve(y_test, y_prob)
roc_auc = auc(fpr, tpr) # roc_auc_score(y_test, y_prob) also works

# Plot the ROC Curve
plt.figure(figsize=(6, 5))
plt.plot(fpr, tpr, color='green', label=f'ROC Curve (AUC = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='grey', linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend(loc='lower right')
plt.show()
```

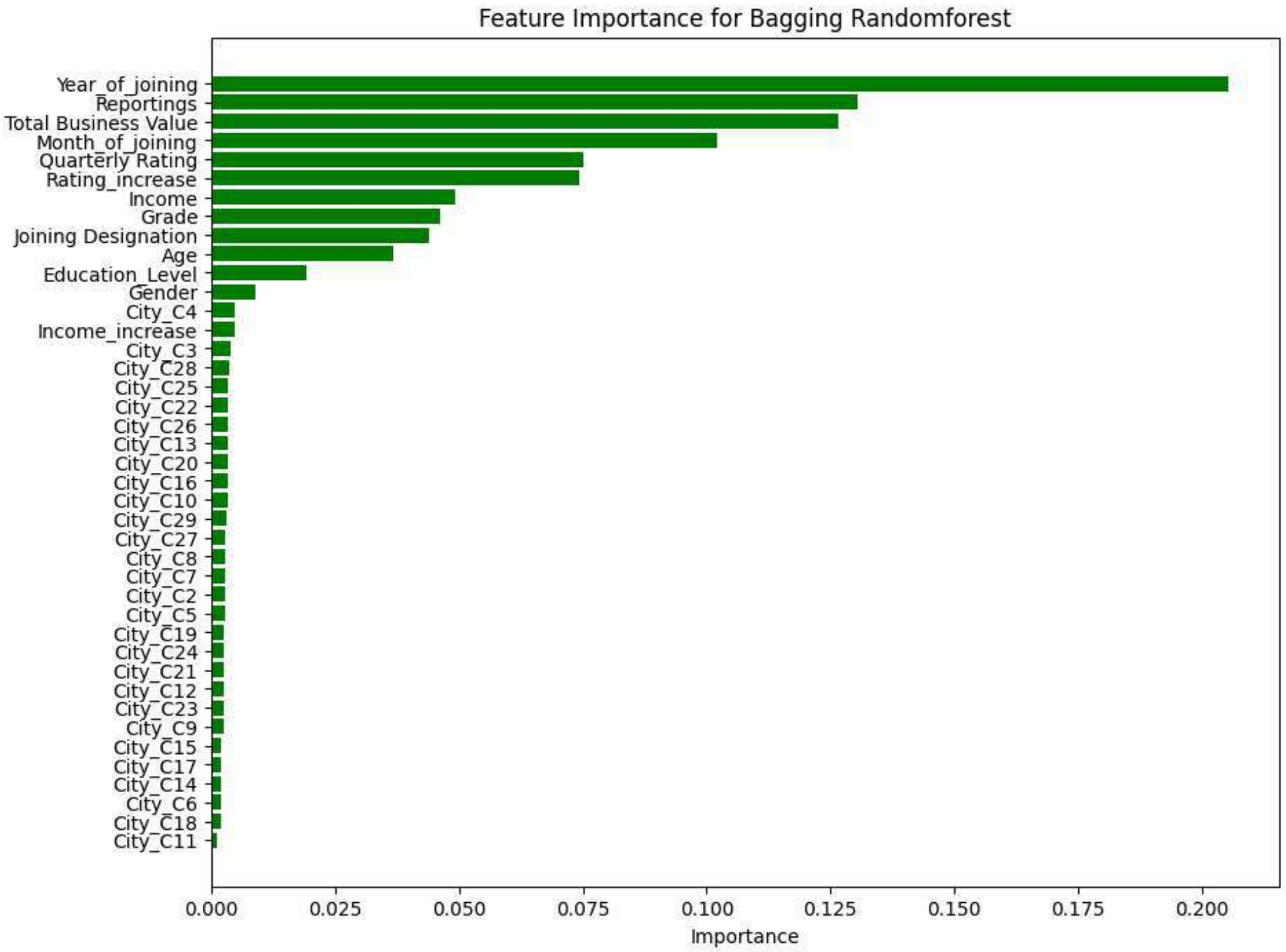


Feature Importance

Identify the features that contribute most to the model's predictions.

In [ ]:
```
# Feature Importance
importances = best_rf.feature_importances_
features = X.columns

# Sort feature importances
importance_df = pd.DataFrame({'Feature': features, 'Importance': importances})
importance_df = importance_df.sort_values(by='Importance', ascending=False)

print("Top Features:\n", importance_df.head())

# Plot feature importances
plt.figure(figsize=(10, 8))
plt.barh(importance_df['Feature'], importance_df['Importance'], color='green') # teal
plt.gca().invert_yaxis()
plt.xlabel('Importance')
plt.title('Feature Importance for Bagging Randomforest')
plt.show()
```

```
Top Features:
                  Feature  Importance
12       Year_of_joining    0.205368
0              Reportings    0.130343
5   Total Business Value    0.126613
11      Month_of_joining    0.102059
8       Quarterly Rating    0.075118
```



Observations 🔎

1. **Model Accuracy:**
   - **Observation:** Train accuracy is 1.00, and test accuracy is 0.92.
   - **Insight:** The model performs excellently on both train and test data. However, perfect training accuracy could indicate slight overfitting. Despite this, the high test accuracy demonstrates that the model generalizes well to unseen data.

1. **Class Balance Performance:**
   - **Observation:** The recall for churned drivers (class 1) is 0.94, while the precision is 0.93. The false negatives (18) are low, which means the model is not missing many actual churned drivers.
   - **Insight:** High recall ensures most churned drivers are identified, aligning with the business objective to capture churn risks. Slightly lower precision indicates occasional over-targeting, but this is acceptable given the priority to minimize churn.

1. **AUC-ROC Curve:**
   - **Observation:** AUC of 0.96 signifies excellent discriminatory power between churned and non-churned drivers.
   - **Insight:** This allows confident ranking of drivers based on their churn probability. High AUC ensures the model can effectively prioritize drivers for retention actions with minimal misclassification.

1. **Feature Importance:**
   - **Observation:** The top 5 features contributing to churn predictions are:
     A. **Year of Joining (0.207):** Recent joiners are at a higher churn risk.
     B. **Reportings (0.129):** Management structure significantly impacts churn.
     C. **Total Business Value (0.125):** Drivers with lower business contributions are more likely to churn.
     D. **Month of Joining (0.109):** Seasonal or onboarding timing affects churn.
     E. **Quarterly Rating (0.074):** Drivers with poor performance ratings are more likely to leave.
   - **Insight:** These features provide actionable levers for improving retention strategies. For instance, better onboarding for recent joiners or addressing low ratings can mitigate churn risks.

1. **Misclassification Insights:**
   - **Observation:** The confusion matrix shows 22 false positives (non-churned misclassified as churned) and 18 false negatives (churned misclassified as non-churned).
   - **Insight:** False negatives (churned drivers missed by the model) are more critical in this context as they represent missed opportunities to prevent churn. The low number of false negatives indicates the model is well-suited for identifying high-risk drivers for targeted interventions.

Overall Insight:

The Bagging Random Forest model is highly effective for this business problem, with strong recall and precision for churned drivers. It identifies critical factors influencing churn, offering actionable insights to design retention strategies while maintaining excellent predictive performance.

Boosting (XGBoost)

What is Boosting in Ensemble Learning?

Boosting is an ensemble learning technique that combines multiple weak learners (e.g., decision trees) to create a strong learner. It focuses on improving the performance iteratively:

- Initially, a weak learner is trained.
- In subsequent iterations, the model gives higher weight to misclassified instances, allowing the next learner to focus on correcting these errors.
- Popular algorithms include XGBoost, LightGBM, and AdaBoost.

Hyperparameter Tuning Using RandomizedSearchCV

Let's use XGBoost as the boosting algorithm. We'll tune the following hyperparameters:

- `n_estimators` : Number of trees.
- `max_depth` : Maximum depth of trees.
- `learning_rate` : How quickly the model learns (shrinkage rate).
- `subsample` : Fraction of data to sample for each tree.
- `colsample_bytree` : Fraction of features to consider per split.

In [ ]:
```
# Define the parameter grid
param_grid = {
    'n_estimators': [100, 200, 300, 400],
    'max_depth': [3, 5, 7, 10],
    'learning_rate': [0.01, 0.05, 0.1, 0.2],
    'subsample': [0.7, 0.8, 0.9, 1],
    'colsample_bytree': [0.7, 0.8, 0.9, 1]
}
```

In [ ]:
```
from xgboost import XGBClassifier
from sklearn.model_selection import RandomizedSearchCV

# Define the model
xgb_model = XGBClassifier(random_state=42, use_label_encoder=False, eval_metric='logloss')

# Randomized Search with Cross Validation
random_search = RandomizedSearchCV(
                                   estimator=xgb_model,
                                   param_distributions=param_grid,
                                   n_iter=50,
                                   scoring='roc_auc',
                                   cv=3,
                                   verbose=2,
                                   random_state=42,
                                   n_jobs=-1
                                  )

# Fit the model
start = dt.datetime.now()
random_search.fit(X_train_balanced, y_train_balanced)
end = dt.datetime.now()

# Best parameters
best_params = random_search.best_params_
print("Best Hyperparameters:", best_params)
print("Best cross-validation score achieved: ", random_search.best_score_)
print(f"Time taken for RandomizedSearchCV(fits) : {end - start}")
```

```
Fitting 3 folds for each of 50 candidates, totalling 150 fits
Best Hyperparameters: {'subsample': 0.7, 'n_estimators': 400, 'max_depth': 10, 'learning_rate': 0.1, 'colsample_bytree': 0.8}
Best cross-validation score achieved:  0.9823631539423859
Time taken for RandomizedSearchCV(fits) : 0:01:04.358362
```

Train the Model with Best Hyperparameters

Once the best parameters are found, train the model on the training set.

In [ ]:
```
# Train the model with the best parameters
best_xgb_model = XGBClassifier(**best_params, random_state=42, use_label_encoder=False, eval_metric='logloss')
best_xgb_model.fit(X_train_balanced, y_train_balanced)
```

Out[ ]:

XGBClassifier

```
XGBClassifier(base_score=None, booster=None, callbacks=None,
              colsample_bylevel=None, colsample_bynode=None,
              colsample_bytree=0.8, device=None, early_stopping_rounds=None,
              enable_categorical=False, eval_metric='logloss',
              feature_types=None, gamma=None, grow_policy=None,
              importance_type=None, interaction_constraints=None,
              learning_rate=0.1, max_bin=None, max_cat_threshold=None,
              max_cat_to_onehot=None, max_delta_step=None, max_depth=10,
              max_leaves=None, min_child_weight=None, missing=nan,
              monotone_constraints=None, multi_strategy=None, n_estimators=400,
              n_jobs=None, num_parallel_tree=None, random_state=42, ...)
```

Model Score / Accuracy Measurement

Evaluate the model's accuracy on the training and testing datasets.

In [ ]:
```
# After_train_test_split       After Scaling                    After SMOTE
# X_train | y_train  -->   X_train_scaled | y_train   -->   X_train_balanced | y_train_balanced
# X_test | y_test    -->   X_test_scaled | y_test     -->   X_test_scaled | y_test
```

In [ ]:
```
# Accuracy on train and test sets
train_accuracy = best_xgb_model.score(X_train_balanced, y_train_balanced)
test_accuracy = best_xgb_model.score(X_test_scaled, y_test)

print(f"Train Accuracy: {train_accuracy:.2f}")
print(f"Test Accuracy: {test_accuracy:.2f}")
```

```
Train Accuracy: 1.00
Test Accuracy: 0.93
```

In [ ]:
```
# Make predictions on the test set
y_train_pred = best_xgb_model.predict(X_train_balanced)
y_test_pred = best_xgb_model.predict(X_test_scaled)

# Evaluate the model
# Accuracy on predictions of train and test sets
train_accuracy = accuracy_score(y_train_balanced, y_train_pred)
print(f"Training Accuracy: {train_accuracy:.2f}")

test_accuracy = accuracy_score(y_test, y_test_pred)
print(f"Test Accuracy: {test_accuracy:.2f}")
```

```
Training Accuracy: 1.00
Test Accuracy: 0.93
```

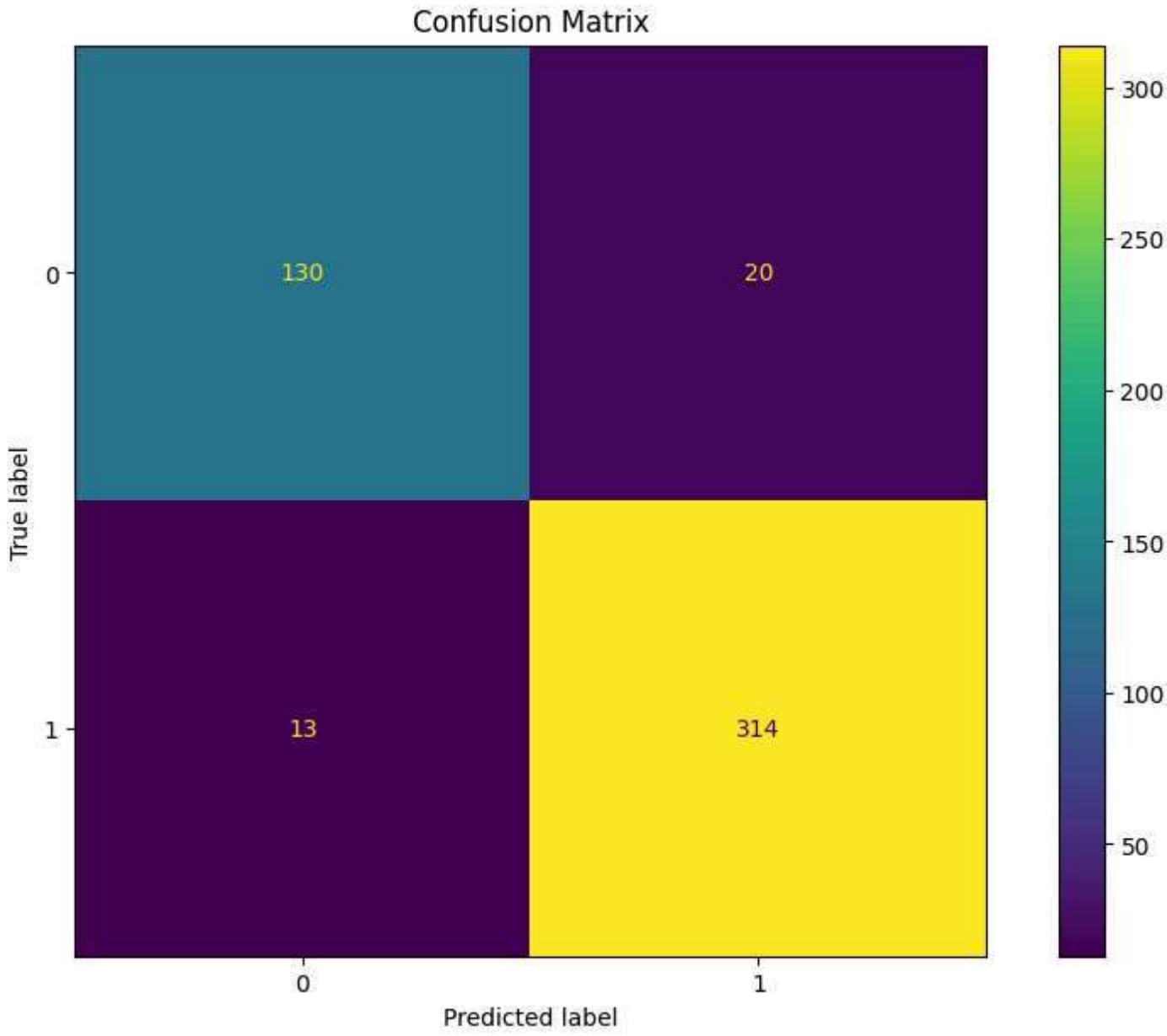
Confusion Matrix

Understand how well the model classifies the positive and negative classes.

In [ ]:
```
# After_train_test_split       After Scaling                    After SMOTE
# X_train | y_train  -->   X_train_scaled | y_train   -->   X_train_balanced | y_train_balanced
# X_test | y_test    -->   X_test_scaled | y_test     -->   X_test_scaled | y_test
```

In [ ]:
```
from sklearn.metrics import confusion_matrix, classification_report

# Predictions on test set
y_pred = best_xgb_model.predict(X_test_scaled)

# Confusion Matrix
cm = confusion_matrix(y_test, y_pred)
print("Confusion Matrix:\n", cm)

#plt.figure(figsize=(5, 4))
ConfusionMatrixDisplay(cm).plot()
plt.title('Confusion Matrix')
plt.show()

# Classification Report
print("Classification Report:\n", classification_report(y_test, y_pred))
```

```
Confusion Matrix:
 [[130  20]
 [ 13 314]]
```

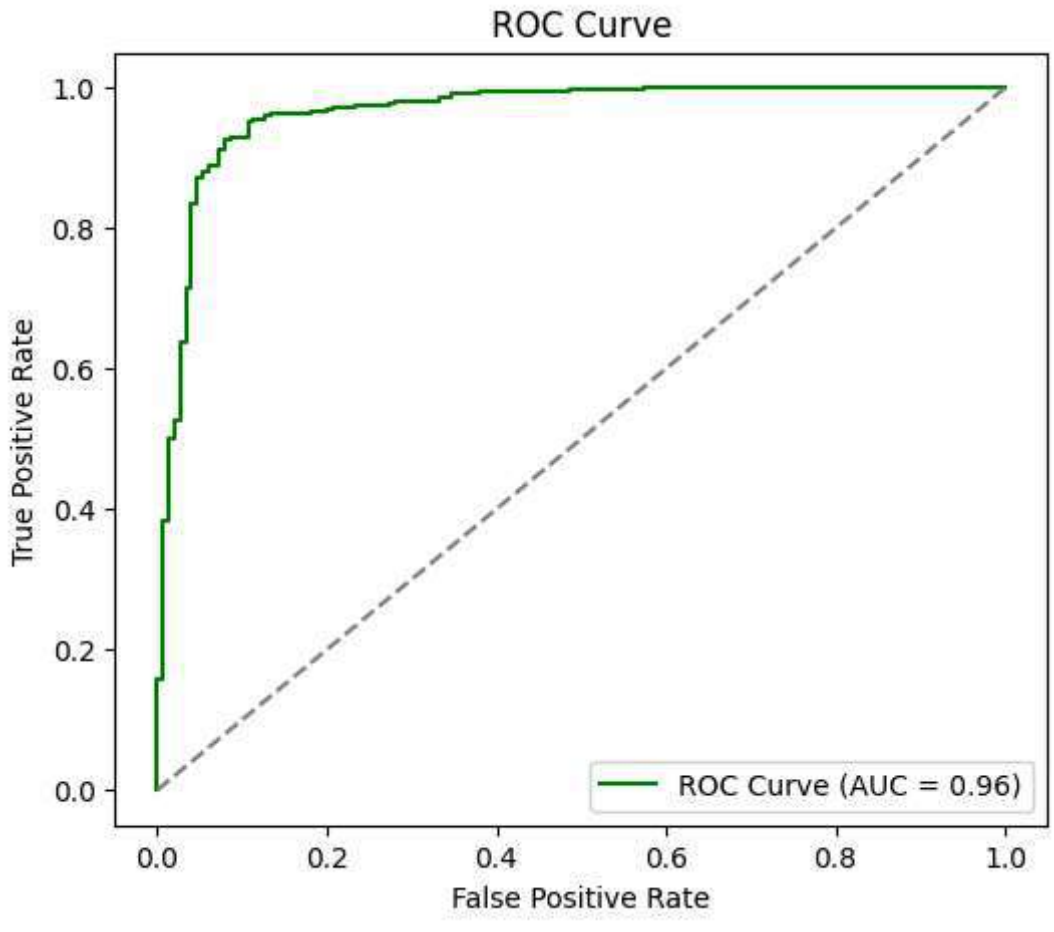



```
Classification Report:
              precision    recall  f1-score   support

           0       0.91      0.87      0.89       150
           1       0.94      0.96      0.95       327

    accuracy                           0.93       477
   macro avg       0.92      0.91      0.92       477
weighted avg       0.93      0.93      0.93       477
```

ROC Curve & AUC

Evaluate the ROC Curve and calculate the AUC score.

In [ ]:
```python
from sklearn.metrics import roc_curve, auc, roc_auc_score
import matplotlib.pyplot as plt

# Predict probabilities for ROC curve
y_prob = best_xgb_model.predict_proba(X_test_scaled)[:, 1]

# ROC Curve
fpr, tpr, thresholds = roc_curve(y_test, y_prob)
roc_auc = auc(fpr, tpr) # roc_auc_score(y_test, y_prob) also works

# Plot the ROC Curve
plt.figure(figsize=(6, 5))
plt.plot(fpr, tpr, color='green', label=f'ROC Curve (AUC = {roc_auc:.2f})')
plt.plot([0, 1], [0, 1], color='grey', linestyle='--')
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve')
plt.legend(loc='lower right')
plt.show()
```

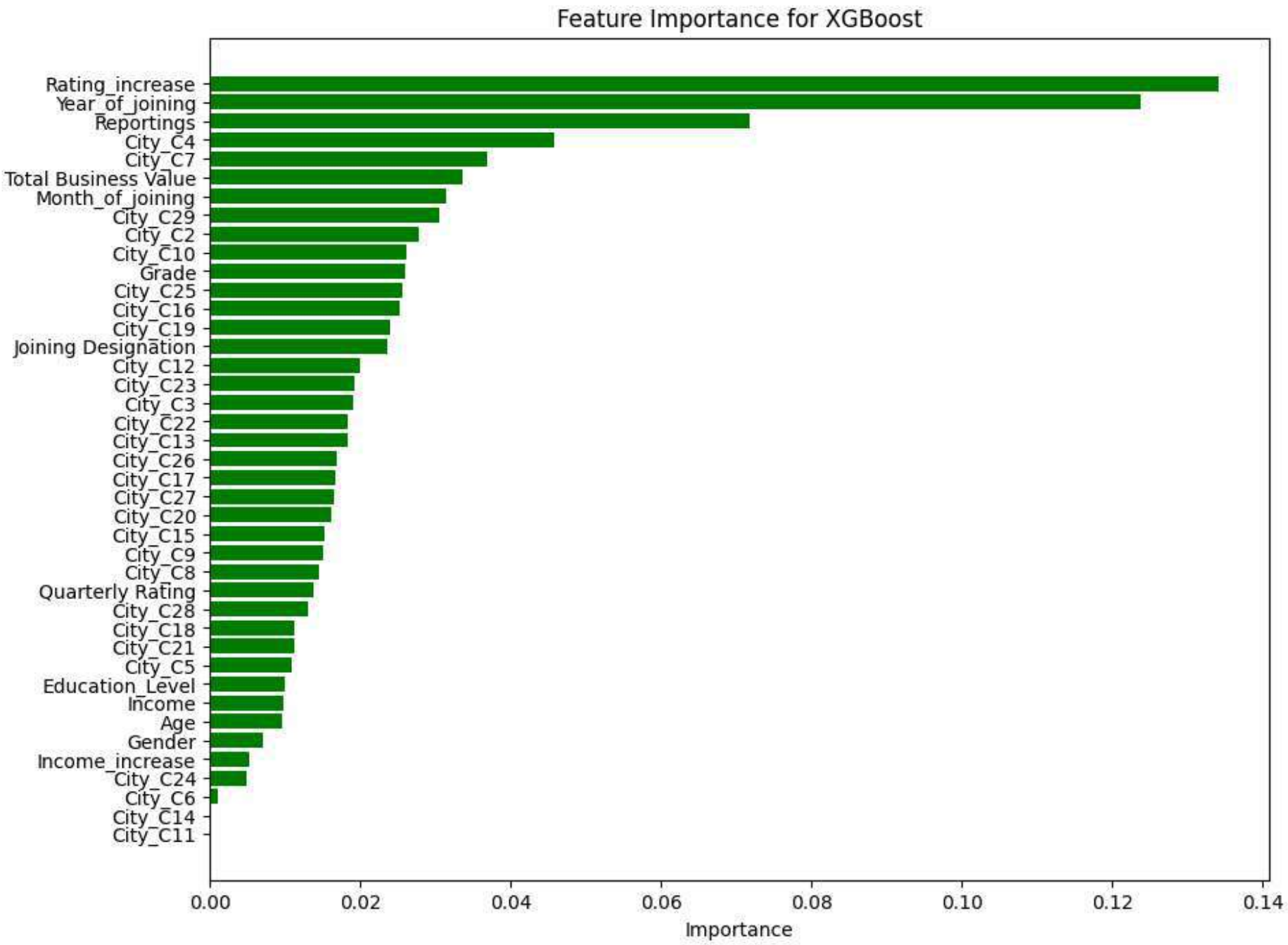


Feature Importance

Identify the features that contribute most to the model's predictions.

In [ ]:
```
# Feature Importance
importances = best_xgb_model.feature_importances_
features = X.columns

# Sort feature importances
importance_df_XGBoost = pd.DataFrame({'Feature': features, 'Importance': importances})
importance_df_XGBoost = importance_df_XGBoost.sort_values(by='Importance', ascending=False)

print("Top Features:\n", importance_df_XGBoost.head())

# Plot feature importances
plt.figure(figsize=(10, 8))
plt.barh(importance_df_XGBoost['Feature'], importance_df_XGBoost['Importance'], color='green') # teal
plt.gca().invert_yaxis()
plt.xlabel('Importance')
plt.title('Feature Importance for XGBoost')
plt.show()
```

```
Top Features:
             Feature  Importance
9   Rating_increase    0.134191
12  Year_of_joining    0.123809
0         Reportings    0.071774
35          City_C4    0.045765
38          City_C7    0.036919
```



Observations🔎

1. **High Predictive Power**:

   - **Train Accuracy**: 100% (perfect fit).
   - **Test Accuracy**: 93%, indicating excellent generalization to unseen data.
   - **ROC AUC = 0.96**, reflecting outstanding model performance in distinguishing between churned and non-churned drivers.

   Insight: The model is highly reliable for operational use, striking a balance between learning from training data and making accurate predictions on test data.

1. **Strong Recall for Churned Drivers**:

   - **Recall for Churned Drivers (Class 1)**: 96%, ensuring that almost all churned drivers are identified.
   - **Low False Negatives (13 cases)**: Indicates minimal missed opportunities to address driver churn.

   Insight: High recall is crucial for the business goal of retention. Identifying almost all churned drivers allows targeted intervention to reduce attrition effectively.

1. **Balanced Precision and Recall**:

   - **Precision for Churned Drivers (Class 1)**: 94%, meaning few false alarms in churn predictions.
   - **F1-Score for Churned Drivers**: 95%, confirming a good trade-off between precision and recall.

   Insight: The model minimizes unnecessary actions on non-churned drivers while ensuring that churn cases are adequately addressed, optimizing resource utilization.

1. **Feature Importance Highlights Critical Factors**:

   - Top features:
     - `Rating_increase` (13.4%): A significant predictor, indicating performance-based retention strategies.
     - `Year_of_joining` (12.3%): Highlights the influence of tenure on churn behavior.
     - `Reportings` (7.2%): Behavioral metrics like reporting frequency are crucial.
     - City-specific factors (`City_C4` and `City_C7`): Reflect localized dynamics affecting churn.

   Insight: These findings enable actionable insights, such as addressing drivers with performance dips or focusing on tenure-based and city-specific retention strategies.

1. **Confusion Matrix Analysis**:

   - **True Positives (314)**: The majority of churn cases were correctly predicted.
   - **False Positives (20)**: A relatively low number of non-churn drivers misclassified as churned.
   - **Overall Accuracy**: 93%, indicating a robust model with balanced performance across all metrics.

Insight: The low false negative rate (13 missed churn cases) ensures effective retention efforts, as most churned drivers are accurately identified.

Summary of Insights:

1. The model is highly effective for churn prediction, achieving a fine balance between recall (minimizing missed churn cases) and precision (reducing false alarms).
2. High-performing features like `Rating_increase` and `Year_of_joining` offer actionable strategies for retention.
3. City-level insights enable localized action plans to address churn in specific regions.
4. The model is ready for deployment, providing reliable and actionable predictions aligned with the business goal of reducing driver churn.

Comparison of Bagging (Random Forest) and Boosting (XGBoost)

Metric	Bagging (Random Forest)	Boosting (XGBoost)	Comparison
Train Accuracy	1.00	1.00	Both models fit the training data perfectly, indicating no underfitting.
Test Accuracy	0.92	0.93	XGBoost performs slightly better, showing better generalization.
Recall for Churned	0.94	0.96	XGBoost has higher recall, critical for identifying churned drivers.
Precision for Churned	0.93	0.94	XGBoost slightly outperforms Random Forest in minimizing false positives.
F1-Score (Churned)	0.94	0.95	XGBoost achieves a better balance between precision and recall.
ROC AUC	0.96	0.96	Both models are equally effective in distinguishing between classes.
Confusion Matrix	More false negatives (18)	Fewer false negatives (13)	XGBoost reduces false negatives, crucial for preventing missed churn cases.
Top Features	`Year_of_joining`, `Reportings`	`Rating_increase`, `Year_of_joining`	Feature importance insights vary, showing distinct value in different aspects.

Key Insights:

1. **Generalization**:
   - Both models generalize well, but XGBoost (93%) slightly outperforms Random Forest (92%) on the test set.
2. **Class Imbalance Handling**:
   - XGBoost handles class imbalance better, as indicated by higher recall for churned drivers (96% vs. 94%). This makes it more effective for reducing churn by identifying most at-risk drivers.
3. **False Negatives**:
   - XGBoost predicts fewer false negatives (13 vs. 18). Given the business objective to minimize driver churn, this is a critical advantage.
4. **Feature Interpretability**:
   - Random Forest highlights `Year_of_joining` and `Total Business Value` as key factors, whereas XGBoost emphasizes `Rating_increase` and `Year_of_joining`. Both offer actionable insights but from different perspectives.
5. **Performance Stability**:
   - XGBoost shows marginally better stability across metrics like recall, F1-score, and precision, making it more reliable for this business case.

Recommendation:

- **XGBoost is the preferred model** for this problem because:
  - It achieves slightly better recall and precision for the churned class, directly addressing the business goal of identifying at-risk drivers.
  - It reduces false negatives, ensuring fewer missed opportunities for retention efforts.
  - While both models perform similarly in AUC and generalization, XGBoost's edge in churn-related metrics aligns better with the business objective.

Random Forest remains a strong alternative, particularly if simplicity or interpretability is prioritized over slight performance gains.

Business Insights and Recommendations 🤝

Business Insights

1. **High Churn Risk Among Drivers with Low Quarterly Ratings and Business Value:**
   - Drivers with lower quarterly ratings and business contributions are more likely to churn, as evidenced by the median `Quarterly Rating` of 1 and `Total Business Value` of approximately 465,000 for churned drivers compared to significantly higher values for retained drivers.

2. **Impact of Driver Age and Tenure on Churn:**
    - Younger drivers (median age of 33) and those who joined recently (notably higher churn among drivers with `Year_of_Joining` closer to the current year) are more prone to leave. This could be due to insufficient work experience, dissatisfaction with early job roles, or lack of adequate support during their initial tenure.
3. **City-Specific Trends in Churn:**
    - Some cities, such as `C20`, exhibit a higher proportion of drivers, potentially with specific churn-related trends. Differences in competition, local demand, or operational challenges might be contributing to these regional disparities.
4. **Gender Balance in Driver Churn:**
    - With a male-to-female driver ratio of approximately 59:41, the churn analysis highlights no stark gender differences, suggesting that both groups face similar challenges. This opens up opportunities to explore targeted gender-neutral retention strategies.
5. **Income and Incentive Influence:**
    - A significant proportion of drivers (98%) reported no income increase, and most churned drivers had relatively lower incomes (median `Income` of ~51,630). This indicates dissatisfaction with pay structures and incentive mechanisms, which might not be competitive enough to retain top talent.

Recommendations

1. **Enhance Driver Performance Support:**
    - Provide personalized training programs and feedback mechanisms for drivers with low quarterly ratings. Partner them with mentors to improve their work experience and performance.
2. **Targeted Retention Strategies for Younger Drivers:**
    - Introduce tailored onboarding programs, periodic role evaluations, and growth opportunities to engage and retain younger drivers. Special initiatives for drivers in their first year can significantly improve retention.
3. **Optimize Pay and Incentive Structures:**
    - Design tiered incentives linked to `Total Business Value`, `Quarterly Ratings`, and reporting consistency. Offering bonuses or incremental pay increases to drivers exceeding performance benchmarks can mitigate churn risk.
4. **Regional Customization of Operations:**
    - Conduct city-level operational studies to address unique challenges faced in high-churn areas such as `C20`. Modify policies, shift timings, or demand management strategies to cater to regional needs effectively.
5. **Improve Reporting and Grievance Systems:**
    - Drivers with higher reporting rates exhibit a higher churn tendency. Implement real-time issue resolution and acknowledgment systems to ensure that drivers feel heard and supported.

😀Conclusion😀

The analysis underscores key factors influencing driver churn, including performance metrics, age, income levels, and regional disparities. By addressing these areas through tailored training, incentive structures, and enhanced support mechanisms, Ola can significantly improve driver retention, operational efficiency, and long-term profitability.