```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
```

**Problem Statement:** Ola, a leading ride-sharing platform, aims to proactively identify drivers who are at risk of attrition (churn). Driver churn impacts operational efficiency, customer satisfaction, and recruitment costs. Given a historical dataset containing driver demographics, performance metrics, and employment history, the objective is to build a predictive model using ensemble learning techniques to forecast whether a driver is likely to churn.

This model should be able to generalize well on unseen data and provide interpretable, actionable insights that can support Ola's driver retention strategy.

**Key Objectives:**

**Predictive Modeling:**

Classify whether a driver will churn (Churn = 1) or stay (Churn = 0) using historical features.

**Ensemble Learning Focus:**

Apply ensemble methods (e.g., Random Forest, Gradient Boosting, XGBoost, etc.) to improve model accuracy and robustness.

**Business Impact:**

Enable early intervention strategies to retain high-risk drivers.

Reduce churn rate and associated costs.

**Model Interpretability:**

Identify key drivers of churn (e.g., income, rating, city, age, etc.).

Support HR and operational teams with explainable metrics.

```
from google.colab import files
uploaded = files.upload()
```

Choose Files ola_driver_scaler.csv

- **ola_driver_scaler.csv**(text/csv) - 1127673 bytes, last modified: 29/7/2025 - 100% done

```
Saving ola_driver_scaler.csv to ola_driver_scaler.csv
```

```
df = pd.read_csv('ola_driver_scaler.csv')
df.head(10)
```

```
/usr/local/lib/python3.11/dist-packages/google/colab/_dataframe_summarizer.py:88: UserWarning: Could not infer format, so each eleme
  cast_date_col = pd.to_datetime(column, errors="coerce")
```

| | Unnamed: 0 | MMM-YY | Driver_ID | Age | Gender | City | Education_Level | Income | Dateofjoining | LastWorkingDate | Joining Designation | Grade | Bu |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 01/01/19 | 1 | 28.0 | 0.0 | C23 | 2 | 57387 | 24/12/18 | NaN | 1 | 1 | 2 |
| 1 | 1 | 02/01/19 | 1 | 28.0 | 0.0 | C23 | 2 | 57387 | 24/12/18 | NaN | 1 | 1 | - |
| 2 | 2 | 03/01/19 | 1 | 28.0 | 0.0 | C23 | 2 | 57387 | 24/12/18 | 03/11/19 | 1 | 1 | |
| 3 | 3 | 11/01/20 | 2 | 31.0 | 0.0 | C7 | 2 | 67016 | 11/06/20 | NaN | 2 | 2 | |
| 4 | 4 | 12/01/20 | 2 | 31.0 | 0.0 | C7 | 2 | 67016 | 11/06/20 | NaN | 2 | 2 | |
| 5 | 5 | 12/01/19 | 4 | 43.0 | 0.0 | C13 | 2 | 65603 | 12/07/19 | NaN | 2 | 2 | |
| 6 | 6 | 01/01/20 | 4 | 43.0 | 0.0 | C13 | 2 | 65603 | 12/07/19 | NaN | 2 | 2 | |
| 7 | 7 | 02/01/20 | 4 | 43.0 | 0.0 | C13 | 2 | 65603 | 12/07/19 | NaN | 2 | 2 | |
| 8 | 8 | 03/01/20 | 4 | 43.0 | 0.0 | C13 | 2 | 65603 | 12/07/19 | NaN | 2 | 2 | |
| 9 | 9 | 04/01/20 | 4 | 43.0 | 0.0 | C13 | 2 | 65603 | 12/07/19 | 27/04/20 | 2 | 2 | |

Next steps: Generate code with df | View recommended plots | New interactive sheet

```
df.shape
```

```
(19104, 14)
```

```
df.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
```

Basic Information About the Dataset • The dataset contains 19104 rows and 14 columns. Data Types Summary:

**Object type (categorical/date) columns:**

1. MMM-YY
2. City
3. Dateofjoining
4. LastWorkingDate

**Numeric type columns:**

o int64: Unnamed: 0 (can be dropped) Driver_ID, Education_Level, Income, Joining Designation, Grade, Total Business Value, Quarterly Rating

o float64: Age, Gender (both have missing values)

**Missing Values Summary:**

• Age: 61 missing

• Gender: 52 missing

• LastWorkingDate: missing for most rows (only 1616 non-null values)

This is expected and will be used to create a target column. but LastWorkingDate can be converted into target column where we can input 0 for missing value and 1 for available value. This means at 1 driver has left the organisation and 0 means still working

Double-click (or enter) to edit

```
df.describe()
```

| | Unnamed: 0 | Driver_ID | Age | Gender | Education_Level | Income | Joining Designation | Grade | Tota Busines Valu |
|---|---|---|---|---|---|---|---|---|---|
| **count** | 19104.000000 | 19104.000000 | 19043.000000 | 19052.000000 | 19104.000000 | 19104.000000 | 19104.000000 | 19104.000000 | 1.910400e+0 |
| **mean** | 9551.500000 | 1415.591133 | 34.668435 | 0.418749 | 1.021671 | 65652.025126 | 1.690536 | 2.252670 | 5.716621e+0 |
| **std** | 5514.994107 | 810.705321 | 6.257912 | 0.493367 | 0.800167 | 30914.515344 | 0.836984 | 1.026512 | 1.128312e+0 |
| **min** | 0.000000 | 1.000000 | 21.000000 | 0.000000 | 0.000000 | 10747.000000 | 1.000000 | 1.000000 | -6.000000e+0 |
| **25%** | 4775.750000 | 710.000000 | 30.000000 | 0.000000 | 0.000000 | 42383.000000 | 1.000000 | 1.000000 | 0.000000e+0 |
| **50%** | 9551.500000 | 1417.000000 | 34.000000 | 0.000000 | 1.000000 | 60087.000000 | 1.000000 | 2.000000 | 2.500000e+0 |
| **75%** | 14327.250000 | 2137.000000 | 39.000000 | 1.000000 | 2.000000 | 83969.000000 | 2.000000 | 3.000000 | 6.997000e+0 |

```
df.columns
```

```
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')
```

```
# converting obejct type into date type columns

df['MMM-YY'] = pd.to_datetime(df['MMM-YY'], errors='coerce')
df['Dateofjoining'] = pd.to_datetime(df['Dateofjoining'], errors='coerce')
df['LastWorkingDate'] = pd.to_datetime(df['LastWorkingDate'], errors='coerce')
```

```
/tmp/ipython-input-4071156479.py:3: UserWarning: Could not infer format, so each element will be parsed individually, falling back t
  df['MMM-YY'] = pd.to_datetime(df['MMM-YY'], errors='coerce')
/tmp/ipython-input-4071156479.py:4: UserWarning: Could not infer format, so each element will be parsed individually, falling back t
  df['Dateofjoining'] = pd.to_datetime(df['Dateofjoining'], errors='coerce')
/tmp/ipython-input-4071156479.py:5: UserWarning: Could not infer format, so each element will be parsed individually, falling back t
  df['LastWorkingDate'] = pd.to_datetime(df['LastWorkingDate'], errors='coerce')
```

```
df.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  datetime64[ns]
 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  datetime64[ns]
 9   LastWorkingDate       1616 non-null   datetime64[ns]
 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: datetime64[ns](3), float64(2), int64(8), object(1)
memory usage: 2.0+ MB
```

```
df.head(5)
```

| | Unnamed: 0 | MMM-YY | Driver_ID | Age | Gender | City | Education_Level | Income | Dateofjoining | LastWorkingDate | Joining Designation | Grade | Tot Busine Val |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0 | 2019-01-01 | 1 | 28.0 | 0.0 | C23 | 2 | 57387 | 2018-12-24 | NaT | 1 | 1 | 2381( |
| **1** | 1 | 2019-02-01 | 1 | 28.0 | 0.0 | C23 | 2 | 57387 | 2018-12-24 | NaT | 1 | 1 | -665 |
| **2** | 2 | 2019-03-01 | 1 | 28.0 | 0.0 | C23 | 2 | 57387 | 2018-12-24 | 2019-03-11 | 1 | 1 | |
| **3** | 3 | 2020-11-01 | 2 | 31.0 | 0.0 | C7 | 2 | 67016 | 2020-11-06 | NaT | 2 | 2 | |
| **4** | 4 | 2020-12-01 | 2 | 31.0 | 0.0 | C7 | 2 | 67016 | 2020-11-06 | NaT | 2 | 2 | |

Next steps: Generate code with df | View recommended plots | New interactive sheet

```
# Create Churn column: 1 if LastWorkingDate is present, else 0
df['Churn'] = df['LastWorkingDate'].notnull().astype(int)
```

```
# for keeping main data frame seperate copied to anotther data frame df1
df1=df
```

```
df1['year'] = df1['LastWorkingDate'].dt.year
df1['month'] = df1['LastWorkingDate'].dt.month
```

```
df1['month'].value_counts()
```

| month | count |
|---|---|
| **7.0** | 189 |
| **5.0** | 161 |
| **2.0** | 155 |
| **1.0** | 152 |
| **9.0** | 145 |
| **11.0** | 142 |
| **6.0** | 138 |
| **3.0** | 133 |
| **10.0** | 132 |
| **12.0** | 121 |
| **4.0** | 91 |
| **8.0** | 57 |

**dtype:** int64

*Most of the drive leaves the company in month July September and November*

```
df1['year'].value_counts()
```

| year | count |
|---|---|
| **2019.0** | 825 |
| **2020.0** | 786 |
| **2018.0** | 5 |

**dtype:** int64

Approximately drive churn rate is same for year 2019 and 2020

```
# Use LastWorkingDate if available, else use last MMM-YY record for each Driver_ID
df1['EndDate'] = df['LastWorkingDate']
df1['EndDate'] = df['EndDate'].fillna(df['MMM-YY'])

# Calculate working duration in years
df1['Tenure_Years'] = (df['EndDate'] - df['Dateofjoining']).dt.days / 365

# Group by Driver_ID to get total income and final tenure
driver_summary = df1.groupby('Driver_ID').agg({
    'Tenure_Years': 'max',           # Max tenure value per driver
    'Income': 'sum'                  # Total income over all months
}).reset_index()

# Round tenure for better readability
driver_summary['Tenure_Years'] = driver_summary['Tenure_Years'].round(2).sort_values(ascending=False)

# Sort by total income in descending order
driver_summary = driver_summary.sort_values(by='Income', ascending=False)

# Display the result
driver_summary.head()
```

| | Driver_ID | Tenure_Years | Income |
|---|---|---|---|
| **257** | 308 | 6.30 | 4522032 |
| **2063** | 2420 | 5.35 | 4069176 |
| **1133** | 1335 | 5.31 | 3770976 |
| **945** | 1111 | 7.50 | 3674616 |
| **481** | 560 | 2.36 | 3653616 |

Next steps: Generate code with driver_summary | View recommended plots | New interactive sheet

**drives which left the company as earn maximum upto 45lakhs to 35lakhs**

```
# check age of drivers leaves the company
df1[df1["Churn"]==1]['Age'].value_counts()
```

| Age | count |
|---|---|
| **31.0** | 126 |
| **32.0** | 123 |
| **34.0** | 114 |
| **33.0** | 103 |
| **30.0** | 98 |
| **28.0** | 90 |
| **29.0** | 84 |
| **36.0** | 84 |
| **35.0** | 82 |
| **27.0** | 79 |
| **37.0** | 68 |
| **38.0** | 68 |
| **25.0** | 60 |
| **39.0** | 56 |
| **26.0** | 53 |
| **41.0** | 49 |
| **42.0** | 35 |
| **24.0** | 34 |
| **40.0** | 31 |
| **44.0** | 27 |
| **23.0** | 27 |
| **43.0** | 26 |
| **46.0** | 20 |
| **45.0** | 20 |
| **22.0** | 13 |
| **47.0** | 9 |
| **49.0** | 7 |
| **48.0** | 7 |
| **52.0** | 6 |
| **51.0** | 6 |
| **21.0** | 2 |
| **50.0** | 2 |
| **58.0** | 1 |
| **53.0** | 1 |

**dtype:** int64

Drivers in the age of 30 to 34 left the company most

```
df.head(5)
```

| | Unnamed: 0 | MMM-YY | Driver_ID | Age | Gender | City | Education_Level | Income | Dateofjoining | LastWorkingDate | Joining Designation | Grade | Tot Busine Val |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | 0 | 2019-01-01 | 1 | 28.0 | 0.0 | C23 | 2 | 57387 | 2018-12-24 | NaT | 1 | 1 | 2381( |
| **1** | 1 | 2019-02-01 | 1 | 28.0 | 0.0 | C23 | 2 | 57387 | 2018-12-24 | NaT | 1 | 1 | -665‹ |
| **2** | 2 | 2019-03-01 | 1 | 28.0 | 0.0 | C23 | 2 | 57387 | 2018-12-24 | 2019-03-11 | 1 | 1 | |
| **3** | 3 | 2020-11-01 | 2 | 31.0 | 0.0 | C7 | 2 | 67016 | 2020-11-06 | NaT | 2 | 2 | |
| **4** | 4 | 2020-12-01 | 2 | 31.0 | 0.0 | C7 | 2 | 67016 | 2020-11-06 | NaT | 2 | 2 | |

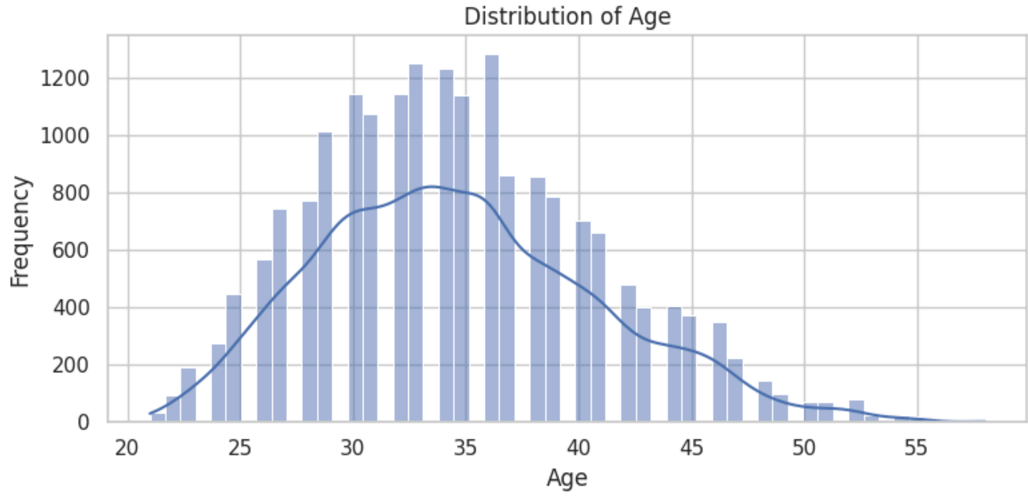
Next steps: Generate code with df | View recommended plots | New interactive sheet

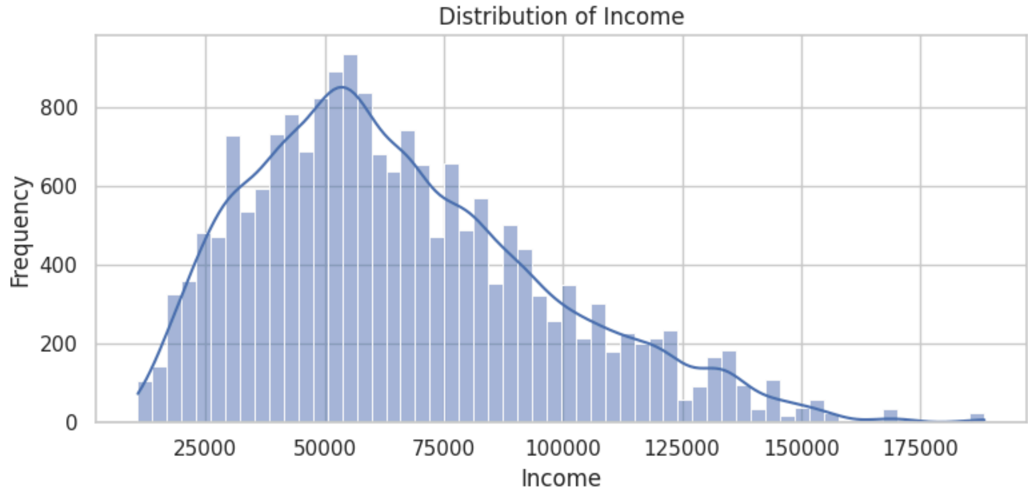
```
df.info()
```

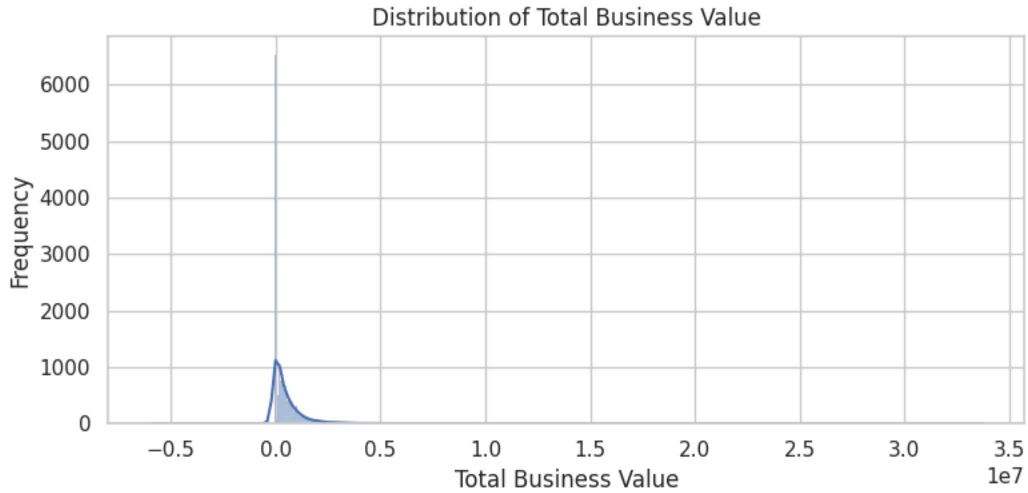
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 19104 entries, 0 to 19103
Data columns (total 19 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   Unnamed: 0            19104 non-null  int64
 1   MMM-YY                19104 non-null  datetime64[ns]
 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  datetime64[ns]
 9   LastWorkingDate       1616 non-null   datetime64[ns]
 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
 14  Churn                 19104 non-null  int64
 15  year                  1616 non-null   float64
 16  month                 1616 non-null   float64
 17  EndDate               19104 non-null  datetime64[ns]
 18  Tenure_Years          19104 non-null  float64
dtypes: datetime64[ns](4), float64(5), int64(9), object(1)
memory usage: 2.8+ MB
```

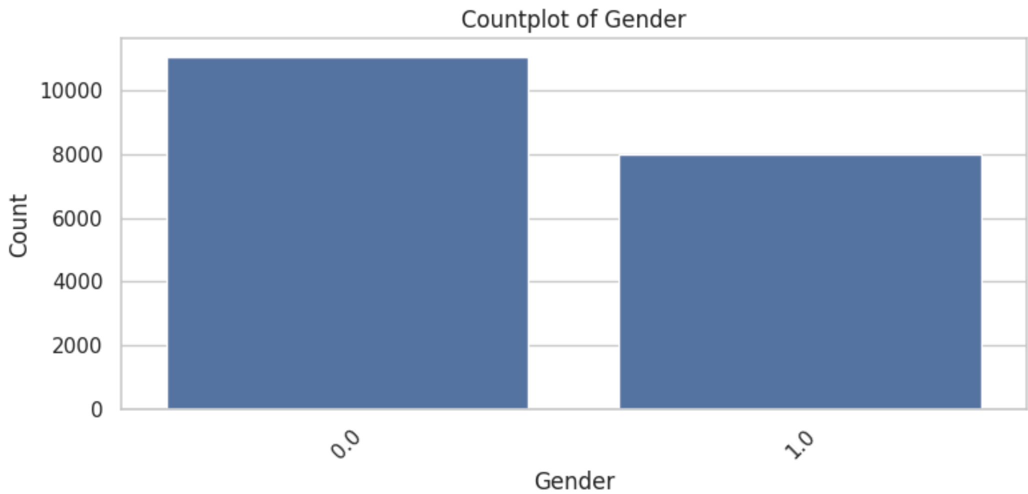
```
# Graphical reprsentation for continous and categorical varibales
import seaborn as sns
import matplotlib.pyplot as plt

# Set style
sns.set(style='whitegrid')

# Continuous columns
continuous_cols = ['Age', 'Income', 'Total Business Value']

# Plot distributions for continuous variables
for col in continuous_cols:
    plt.figure(figsize=(8, 4))
    sns.histplot(df[col].dropna(), kde=True)
    plt.title(f'Distribution of {col}')
    plt.xlabel(col)
    plt.ylabel('Frequency')
    plt.tight_layout()
    plt.show()

# Categorical columns
categorical_cols = ['Gender', 'City', 'Education_Level', 'Joining Designation', 'Grade', 'Quarterly Rating']

# Plot countplots for categorical variables
for col in categorical_cols:
    plt.figure(figsize=(8, 4))
    sns.countplot(data=df, x=col, order=df[col].value_counts().index)
    plt.title(f'Countplot of {col}')
    plt.xlabel(col)
    plt.ylabel('Count')
    plt.xticks(rotation=45)
    plt.tight_layout()
```

```
plt.show()
```

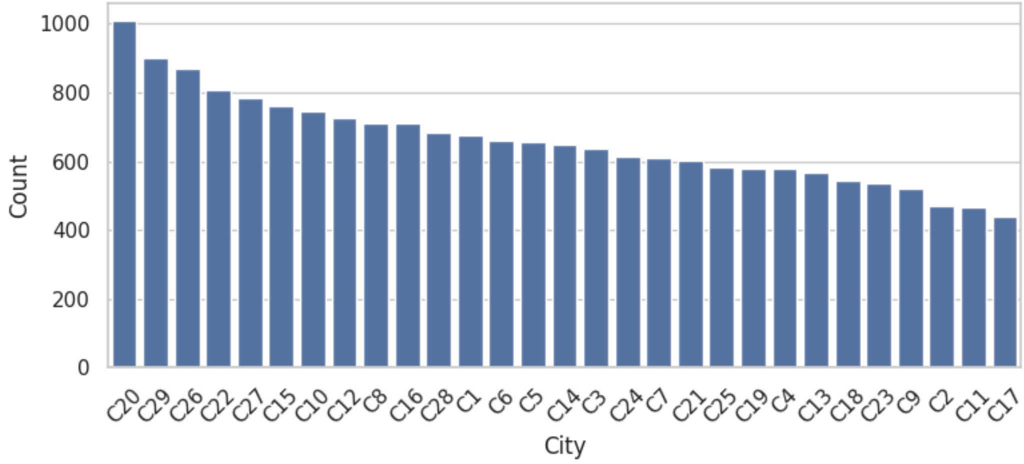


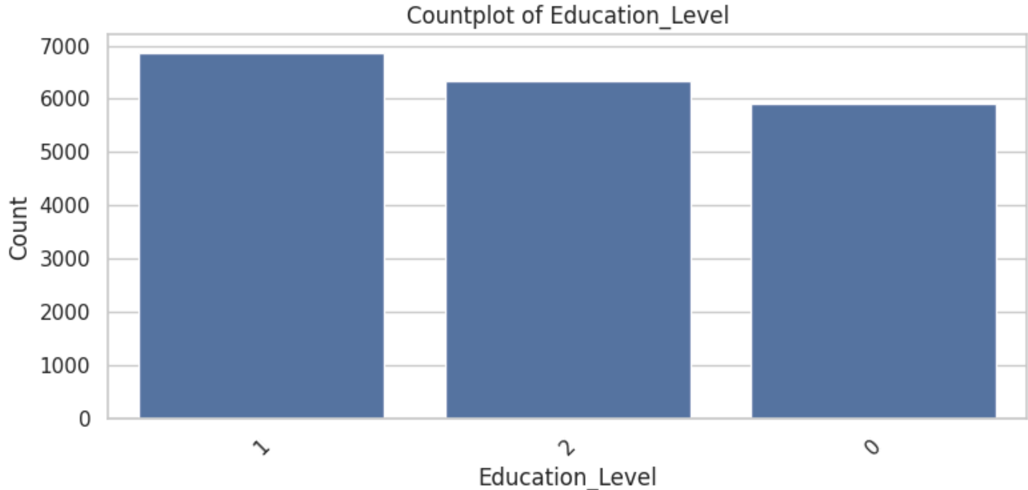



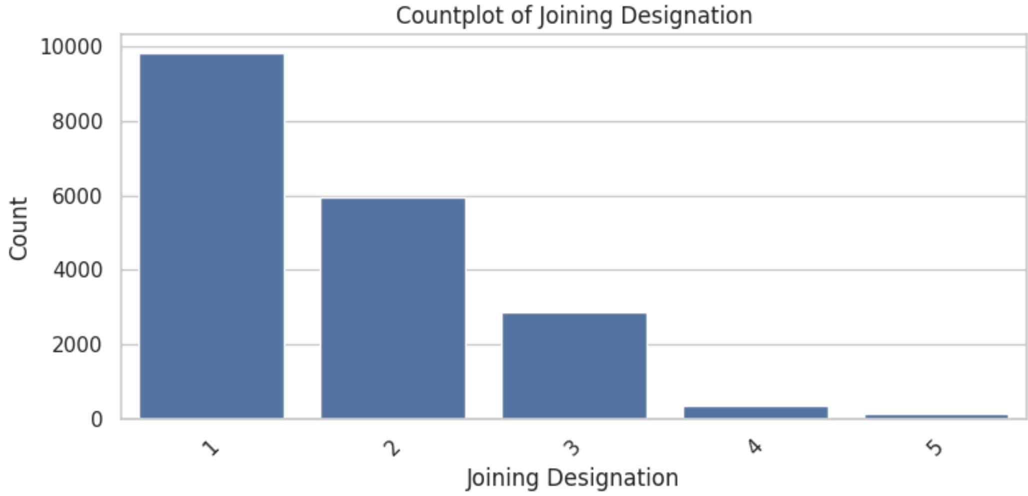



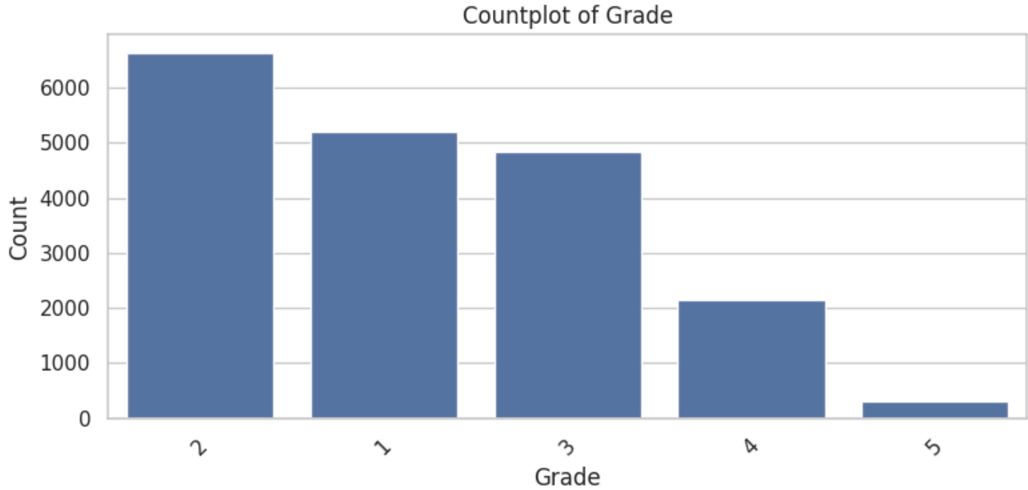



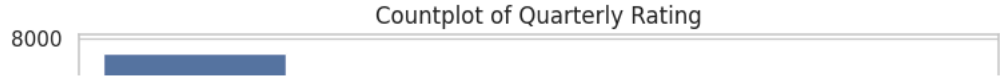

Countplot of City

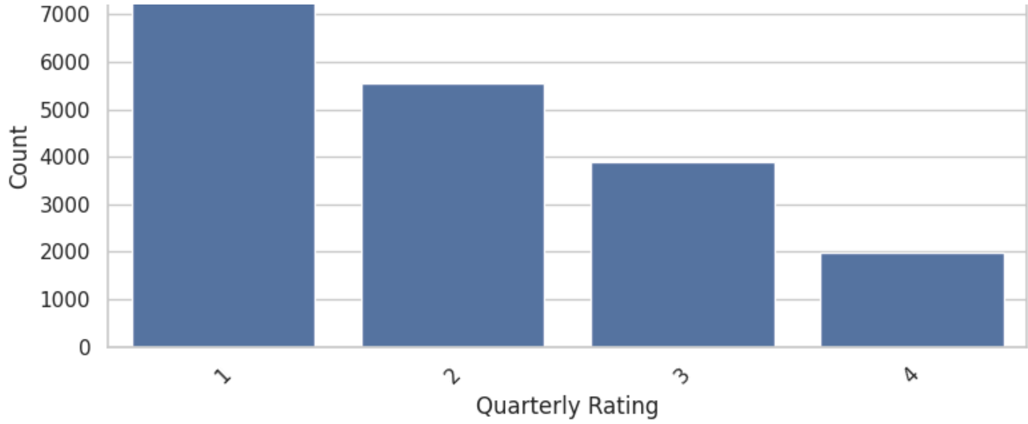

Count
City
C20
C29
C26
C22
C27
C15
C10
C12
C8
C16
C28
C1
C6
C5
C14
C3
C24
C7
C21
C25
C19
C4
C13
C18
C23
C9
C2
C11
C17


Countplot of Education_Level
Count
Education_Level
1
2
0


Countplot of Joining Designation
Count
Joining Designation
1
2
3
4
5


Countplot of Grade
Count
Grade
2
1
3
4
5


Countplot of Quarterly Rating
8000


7000
6000
5000
4000
3000
2000
1000
0
Count
1
2
3
4
Quarterly Rating

🔹 Age Distribution The age distribution is positively skewed, with most drivers aged between 28 and 38 years.

The modal age group is around 32–35 years, highlighting this as the most common age bracket.

Very few drivers are older than 45, and drivers above 50 are rare.

Implication: Ola's driver base is relatively young. Retention strategies should focus on early-career engagement, career progression, and long-term incentives.

🔹 Income Distribution Income is also right-skewed, with a long tail toward higher earnings.

Most drivers earn between ₹35,000 and ₹75,000, with a peak in the ₹50,000–₹60,000 range.

High earners (above ₹1,25,000) are rare.

Implication: While income varies widely, the majority fall in a mid-income bracket. Income could be a strong predictor of churn, with different motivations for low- vs. high-income drivers.

🔹 Total Business Value (TBV) Extremely right-skewed with a sharp spike near zero and a long tail of high performers.

Many drivers show minimal TBV, possibly due to short tenure or low activity.

Implication: Apply log transformation to normalize TBV. Investigate high TBV outliers as potential top performers or data errors.

🔹 Gender Two encoded categories: 0.0 and 1.0.

Category 0.0 is slightly more common (~~11,000 drivers) than 1.0 (~~8,000).

Implication: Mild imbalance—may be usable as-is, but monitor for bias in churn modeling.

🔹 City Drivers are spread across many cities, with C20, C29, and C26 having the highest counts (~1,000 each).

Smaller cities have ~400–500 drivers.

Implication: A few urban centers dominate the driver population. Consider grouping low-volume cities into an "Other" category or applying target encoding.

🔹 Education Level Encoded as 0, 1, 2.

Fairly balanced: Level 1 (~~6,900) slightly ahead of 2 (~~6,300) and 0 (~5,900).

Implication: Drivers come from diverse educational backgrounds, likely not a strong standalone churn predictor but may interact with Grade or Income.

🔹 Joining Designation Highly imbalanced:

Designation 1 dominates (~9,800 drivers).

Designations 2 (~~6,000), 3 (~~2,800) are less common.

Designations 4 and 5 are rare (<500).

Implication: Most drivers join at the lowest level. Consider grouping rare designations or using ordinal encoding.

🔹 Grade Majority in Grade 2 (~~6,600), followed by Grades 1 and 3 (~~5,000 each).

Grades 4 (~2,100) and 5 (<500) are rare.

Implication: Strong central tendency in grading. Merge sparse categories for modeling stability.

🔹 Quarterly Rating Strongly skewed toward lower ratings:

Rating 1 has ~7,600 drivers.

Ratings 2–4 decline sharply.

Implication: Potential link between low performance and churn. A candidate for feature interaction with TBV or Grade.

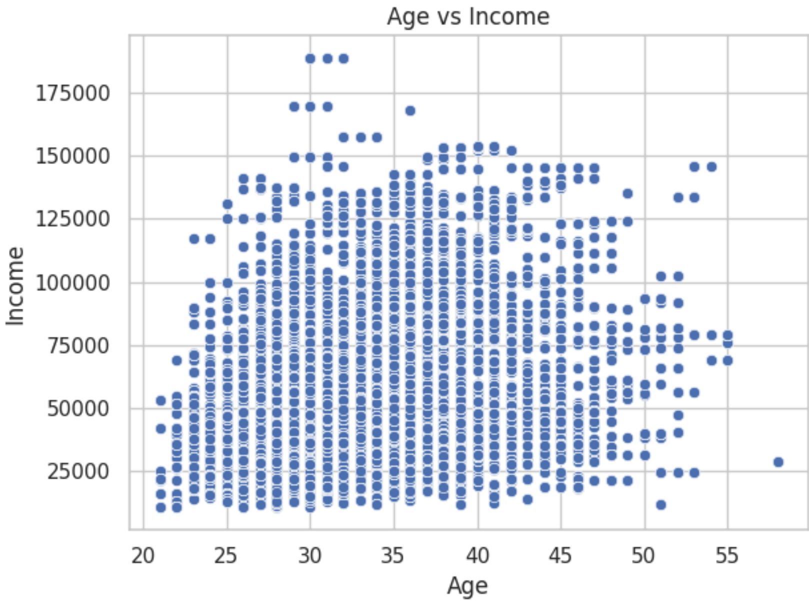
🔑 Key Takeaways for Modeling: Skewed variables (Age, Income, TBV) require transformation or binning.

High cardinality (City) and imbalanced variables (Grade, Designation, Rating) need careful encoding.

Categorical variables show interpretable patterns, especially where performance or tenure may vary (e.g., Ratings, Grades).

Target features like TBV, Income, and Rating may have strong predictive power for churn.

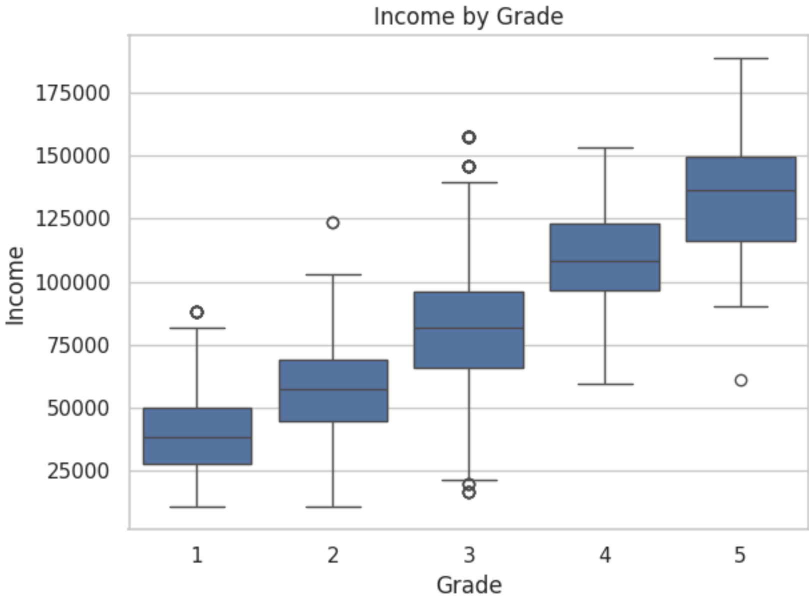
```
# Bivariate graphical checking
# age vs income

# Scatter plot for continuous-continuous relationships
sns.scatterplot(data=df, x='Age', y='Income')
plt.title('Age vs Income')
plt.xlabel('Age')
plt.ylabel('Income')
plt.tight_layout()
plt.show()
```




1. Income Pattern Across Age • Income appears to increase with age until around 35–40 years, after which it starts to plateau or slightly decline. • This suggests that experience contributes to higher income in early to mid-career, but there may be diminishing returns or attrition effects after 45.
2. High-Density Zone • There is a high concentration of points between: o Age: 28 to 40 years o Income: ₹40,000 to ₹100,000 • This indicates that the core workforce of drivers falls within this age-income bracket.
3. Outliers • A few data points show incomes above ₹150,000, particularly between the ages of 28–38. • These may be top performers, special assignments, or anomalies worth investigating.
4. Older Age Group Trends • Post age 45, the density of drivers drops, and their incomes also appear more scattered and lower. • This could indicate: o Lower income opportunities for older drivers o Early retirement or career transitions o Health or performance constraints impacting income
5. Younger Drivers (20–25) • Income levels for drivers aged below 25 are relatively low and varied. • Possibly due to: o Being new to the platform o Fewer hours worked o Limited access to high-earning opportunities

**Insights & Implications** • Workforce Focus: Majority of drivers earning mid-to-high income are in the 30–40 age range. This could be the sweet spot for engagement, retention, and promotion. • Policy Direction: o Offer career growth plans for younger drivers. o Support older drivers with incentives or alternative roles if income tends to decline. • Modeling Note: There may be a non-linear relationship between age and income. Consider polynomial features or binning age when building predictive models.

```
# grade vs income
# Box plot for categorical-continuous relationships
sns.boxplot(data=df, x='Grade', y='Income')
plt.title('Income by Grade')
plt.xlabel('Grade')
plt.ylabel('Income')
plt.tight_layout()
plt.show()
```

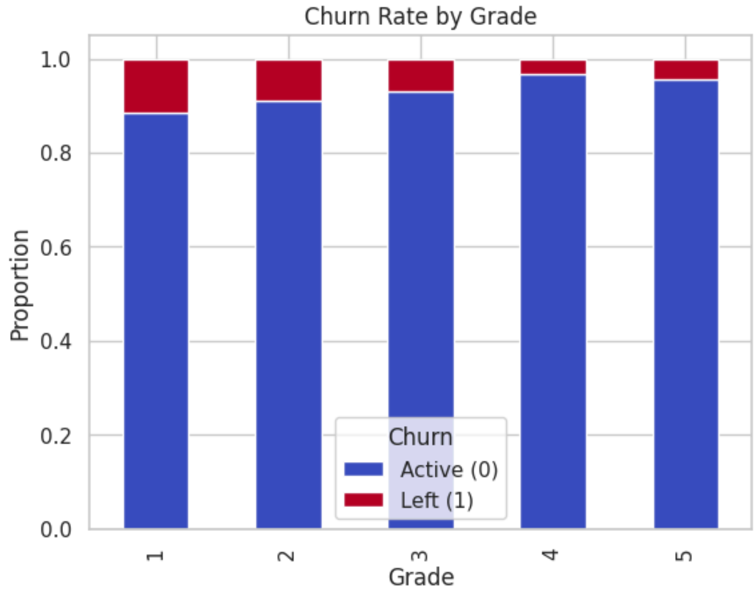


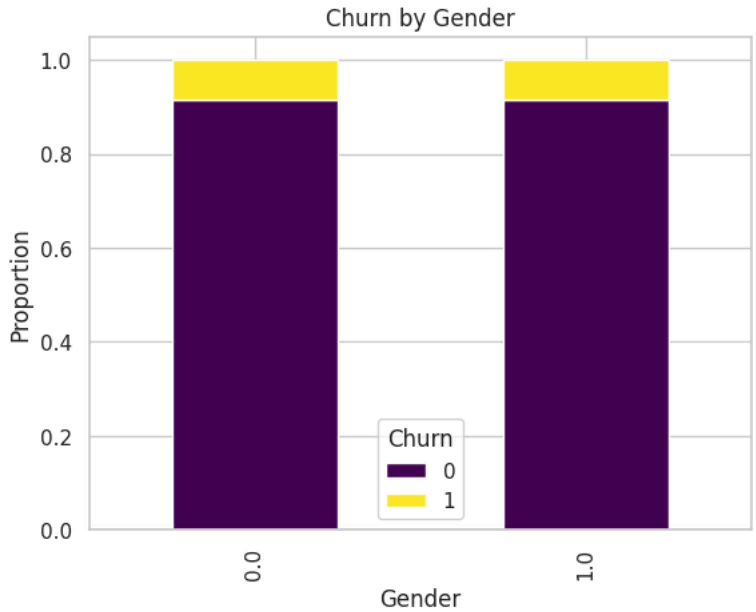

1. Positive Correlation • There is a clear upward trend: As the Grade increases from 1 to 5, the median income rises significantly. • This suggests that Grade is a strong indicator of driver income, and likely tied to experience, performance, or tenure.
2. Median Incomes by Grade • Grade 1: Median income around ₹40,000 • Grade 2: Median near ₹60,000 • Grade 3: Median around ₹85,000 • Grade 4: Median around ₹110,000 • Grade 5: Median around ₹135,000+ The growth is consistent and significant across grades.
3. Interquartile Range (IQR) and Spread • The spread increases with grade, especially for Grades 3 to 5. • This indicates more variability in income at higher grades — likely due to performance-based pay or variable workloads.
4. Outliers • Some outliers are present at all grades, both high and low: o Lower-grade outliers show higher incomes (possibly high performers or bonuses). o Higher-grade outliers with lower incomes may suggest inactivity, part-time work, or data anomalies.
5. Lower Bound Shift • The minimum incomes also shift upward with grade, indicating even the lowest earners in higher grades still make more than most low-grade drivers.

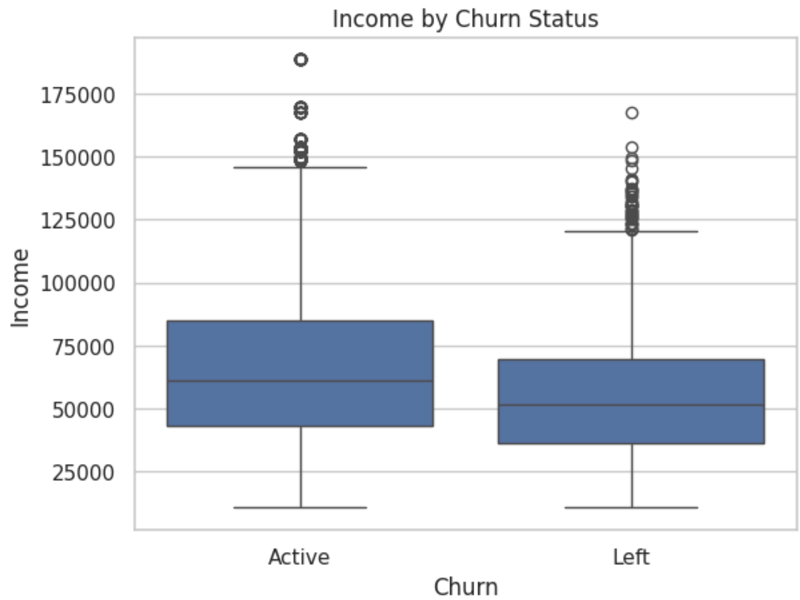
**Insights & Implications** • Career Incentive Structure: The strong income–grade linkage suggests that promoting drivers to higher grades is financially rewarding and can be used to boost retention. • Modeling Suggestion: Grade should be treated as a predictor variable in churn and income models — possibly even ordinal if treated numerically.

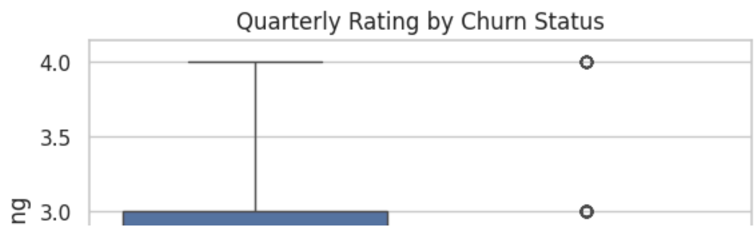
```
# bivariate analysis aganist considering target column churn

# Grade vs Churn
# Stacked bar plot
pd.crosstab(df['Grade'], df['Churn'], normalize='index').plot(kind='bar', stacked=True, colormap='coolwarm')
plt.title('Churn Rate by Grade')
plt.ylabel('Proportion')
plt.xlabel('Grade')
plt.legend(title='Churn', labels=['Active (0)', 'Left (1)'])
plt.show()


# Gender vs Churn

pd.crosstab(df['Gender'], df['Churn'], normalize='index').plot(kind='bar', stacked=True, colormap='viridis')
plt.title('Churn by Gender')
plt.ylabel('Proportion')
plt.xlabel('Gender')
plt.show()

# Income vs Churn

sns.boxplot(data=df, x='Churn', y='Income')
plt.title('Income by Churn Status')
plt.xticks([0, 1], ['Active', 'Left'])
plt.show()



# Quarterly Rating vs Churn


sns.boxplot(data=df, x='Churn', y='Quarterly Rating')
plt.title('Quarterly Rating by Churn Status')
plt.xticks([0,1], ['Active', 'Left'])
plt.show()
```

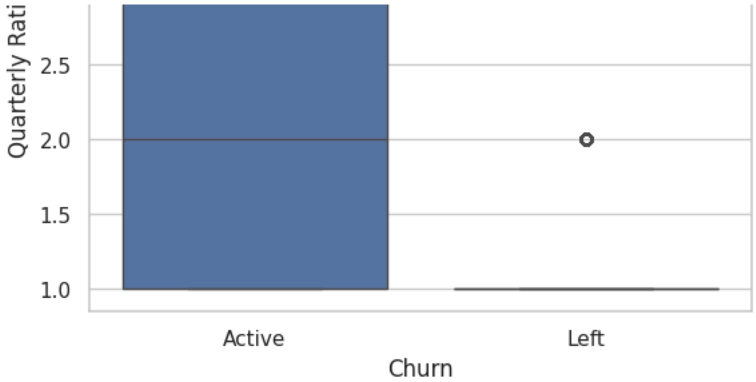
```
# Age vs Churn


sns.boxplot(data=df, x='Churn', y='Age')
plt.title('Age by Churn Status')
plt.xticks([0, 1], ['Active', 'Left'])
plt.show()

# Boxplot of Total Business Value by Churn


sns.boxplot(x='Churn', y='Total Business Value', data=df)
plt.title("Business Value vs Churn Status")
plt.show()
```

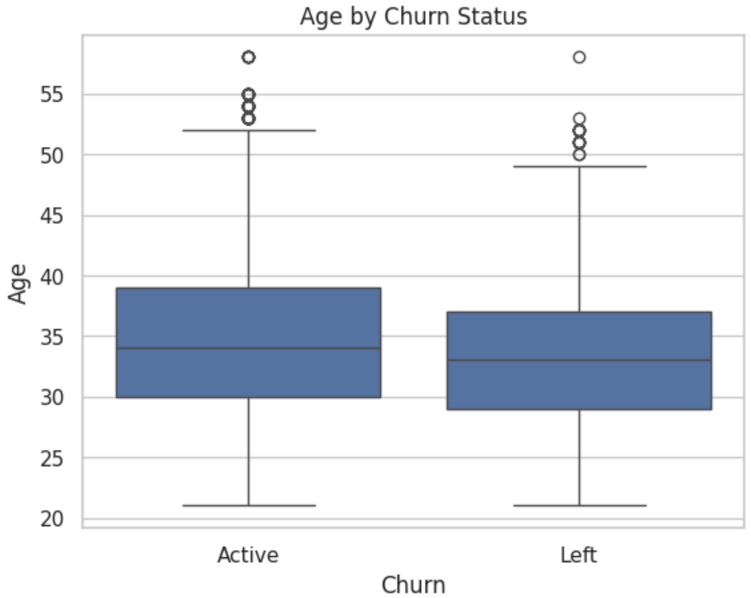

Churn Rate by Grade
Proportion
1.0
0.8
0.6
0.4
0.2
0.0
Churn
Active (0)
Left (1)
1
2
3
4
5
Grade

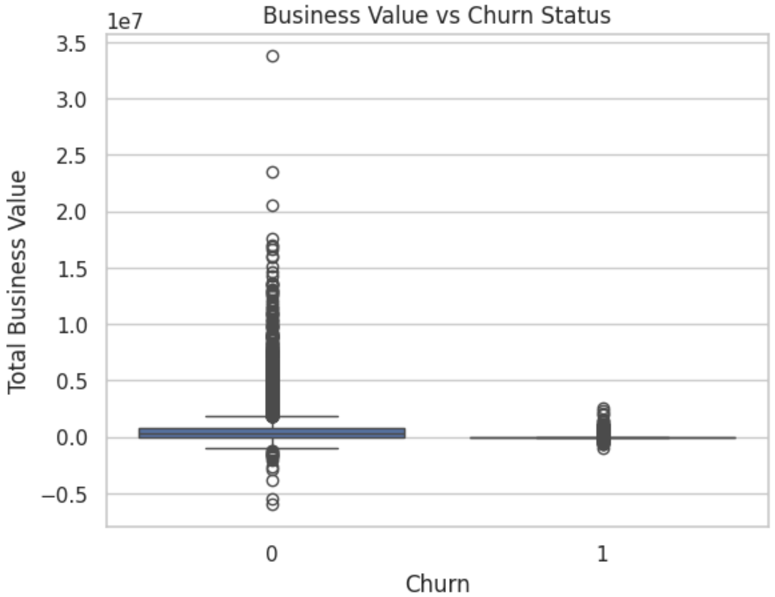

Churn by Gender
Proportion
1.0
0.8
0.6
0.4
0.2
0.0
Churn
0
1
0.0
1.0
Gender


Income by Churn Status
Income
175000
150000
125000
100000
75000
50000
25000
Active
Left
Churn

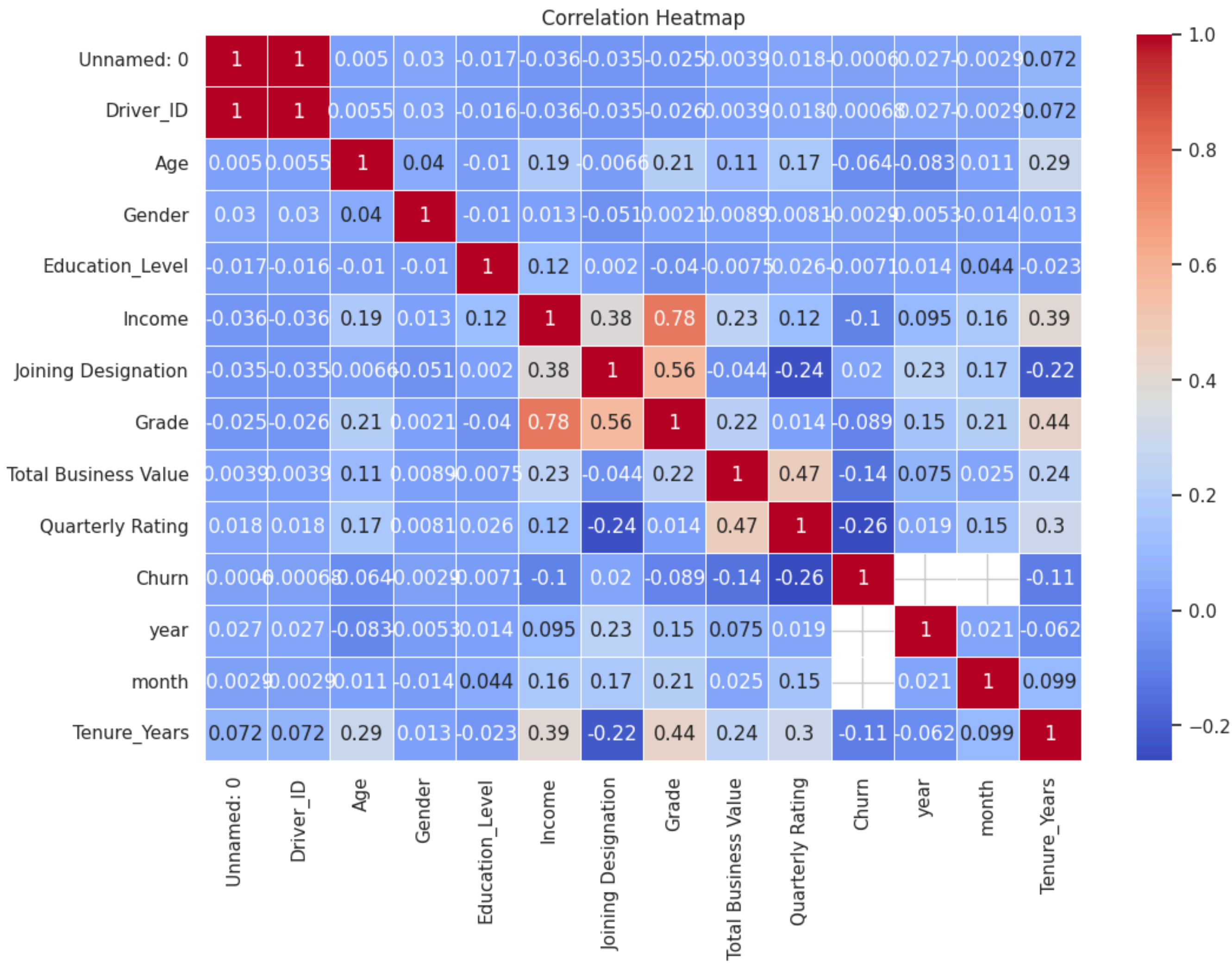

Quarterly Rating by Churn Status
4.0
3.5
3.0


Quarterly Rati
2.5
2.0
1.5
1.0
Active
Left
Churn


Age by Churn Status
Age
55
50
45
40
35
30
25
20
Active
Left
Churn


Business Value vs Churn Status
1e7
Total Business Value
3.5
3.0
2.5
2.0
1.5
1.0
0.5
0.0
−0.5
0
1
Churn

```
# chceking corelation of columns

# Select only numeric columns for correlation analysis
numeric_df = df.select_dtypes(include=['number'])

# Compute correlation matrix
corr_matrix = numeric_df.corr()

# Plot the heatmap
plt.figure(figsize=(12, 8))
sns.heatmap(corr_matrix, annot=True, cmap='coolwarm', linewidths=0.5)
plt.title("Correlation Heatmap")
plt.show()
```




The data strongly suggests that employee performance and seniority are the primary drivers of churn. Employees with low quarterly ratings and those in lower grades are significantly more likely to leave the company. In contrast, demographic factors like gender and age appear to have little to no impact on an employee's decision to leave.

**Performance is a Key Predictor of Churn** The most significant factor related to employee churn is the Quarterly Rating. • Correlation Heatmap: Shows a moderate negative correlation of -0.26 between Quarterly Rating and Churn. This indicates that as an employee's rating goes down, the likelihood of them leaving goes up. • Quarterly Rating Boxplot: This chart provides a stark visual confirmation. The vast majority of employees who left had a quarterly rating of 1. In contrast, active employees have a much higher median rating and a wider distribution of scores. This implies that poor performance is a major reason for attrition.

- Drivers who stayed (Churn=0) generally generated higher business value. This supports the hypothesis that high-performing drivers are more likely to stay, a useful signal for retention modeling."

**Seniority and Grade Matter** An employee's grade and designation, which are linked to seniority and responsibility, also play a crucial role. • Churn Rate by Grade: The stacked bar chart clearly shows that employees in Grade 1 have the highest proportion of churn. This churn rate progressively decreases as the grade level increases up to Grade 4. This suggests that entry-level or junior employees are the most likely to leave. • Correlation Heatmap: This is supported by the negative correlations between Churn and Grade (-0.14) and Joining Designation

(-0.20). Note that Grade and Joining Designation are strongly correlated with each other (0.56) and with Income (0.78), creating a cluster of factors related to seniority.

**Demographics Show Little Impact** Demographic factors do not appear to be significant drivers of churn in this dataset. • Gender: The Churn by Gender chart shows that the proportion of employees leaving is almost identical for both genders. The heatmap confirms this with a near-zero correlation of 0.013. • Age: The Age by Churn Status boxplot shows that the median age of employees who left is only slightly lower than that of active employees. The distributions largely overlap, indicating age is not a strong differentiator. • Income: While the median income for employees who left is slightly lower, the Income by Churn Status boxplot shows a very large overlap between the two groups. The weak correlation of -0.1 in the heatmap confirms that income is not a primary driver of churn on its own.

Double-click (or enter) to edit

```
# removing duplicates

# Find complete duplicate rows
duplicate_rows = df[df.duplicated()]

# Count of complete duplicates
print(f"Total complete duplicate rows: {duplicate_rows.shape[0]}")
```

```
Total complete duplicate rows: 0
```

No duplicates row where found.

```
# checking and taking action on missing value

# Check for missing values
missing_summary = df.isnull().sum()
missing_summary = missing_summary[missing_summary > 0].sort_values(ascending=False)

print("Missing values by column:\n", missing_summary)
```

```
Missing values by column:
 LastWorkingDate    17488
month              17488
year               17488
Age                   61
Gender                52
dtype: int64
```

LastWorkingDate , month and year are not going to affect on model training only we required to impute missing values in Age and Gender column

```
# inputing the missing values by KNN imputer to Agr and Generd column


from sklearn.impute import KNNImputer

# Select only the relevant columns for imputation
subset_cols = ['Age', 'Gender']
df_subset = df[subset_cols]

# Apply KNN Imputer
imputer = KNNImputer(n_neighbors=5)
df_imputed = pd.DataFrame(imputer.fit_transform(df_subset), columns=subset_cols)

# Update the original DataFrame
df['Age'] = df_imputed['Age']
df['Gender'] = df_imputed['Gender']
```

```
# onces again Checking for missing values
missing_summary = df.isnull().sum()
missing_summary = missing_summary[missing_summary > 0].sort_values(ascending=False)

print("Missing values by column:\n", missing_summary)
```

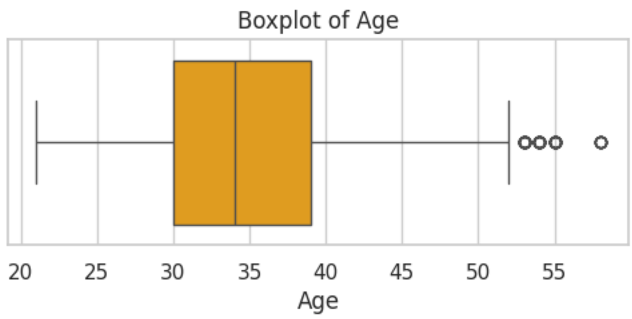
```
Missing values by column:
 LastWorkingDate    17488
year               17488
month              17488
dtype: int64
```

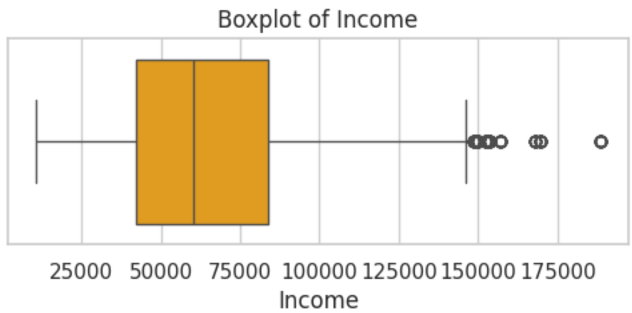
Required result got

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 19104 entries, 0 to 19103
Data columns (total 19 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   Unnamed: 0            19104 non-null  int64
 1   MMM-YY                19104 non-null  datetime64[ns]
 2   Driver_ID             19104 non-null  int64
 3   Age                   19104 non-null  float64
 4   Gender                19104 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  datetime64[ns]
 9   LastWorkingDate       1616 non-null   datetime64[ns]
 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
 14  Churn                 19104 non-null  int64
 15  year                  1616 non-null   float64
 16  month                 1616 non-null   float64
 17  EndDate               19104 non-null  datetime64[ns]
 18  Tenure_Years          19104 non-null  float64
dtypes: datetime64[ns](4), float64(5), int64(9), object(1)
memory usage: 2.8+ MB
```

```
# checking outliers of data set of Box plot

cols = ['Age','Income']

for col in cols:
    plt.figure(figsize=(6, 2))
    sns.boxplot(x=df[col], color='orange')
    plt.title(f'Boxplot of {col}')
    plt.show()
```



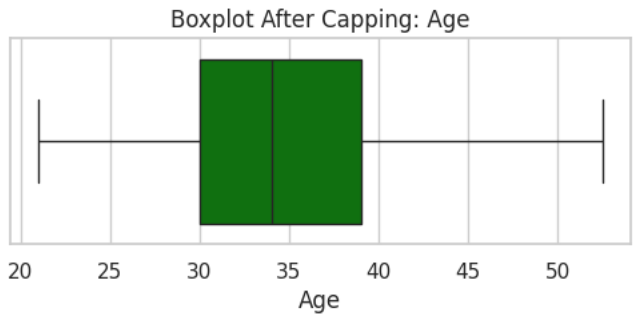



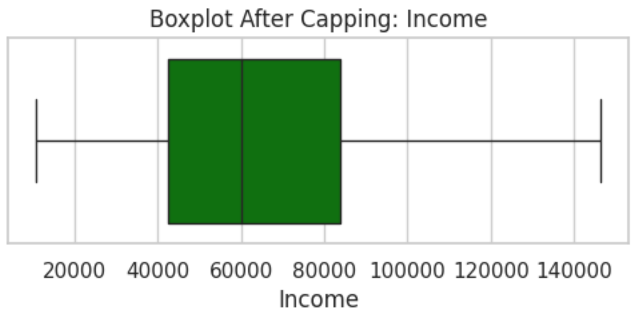

```
# checking by IQR method

def cap_outliers_iqr(df, col):
    Q1 = df[col].quantile(0.25)
    Q3 = df[col].quantile(0.75)
    IQR = Q3 - Q1
    lower = Q1 - 1.5 * IQR
    upper = Q3 + 1.5 * IQR
    print(f'{col} - Lower Bound: {lower:.2f}, Upper Bound: {upper:.2f}')
    df[col] = df[col].clip(lower, upper)
    return df

# applying clipping
```

```
for col in cols:
    df = cap_outliers_iqr(df, col)
```

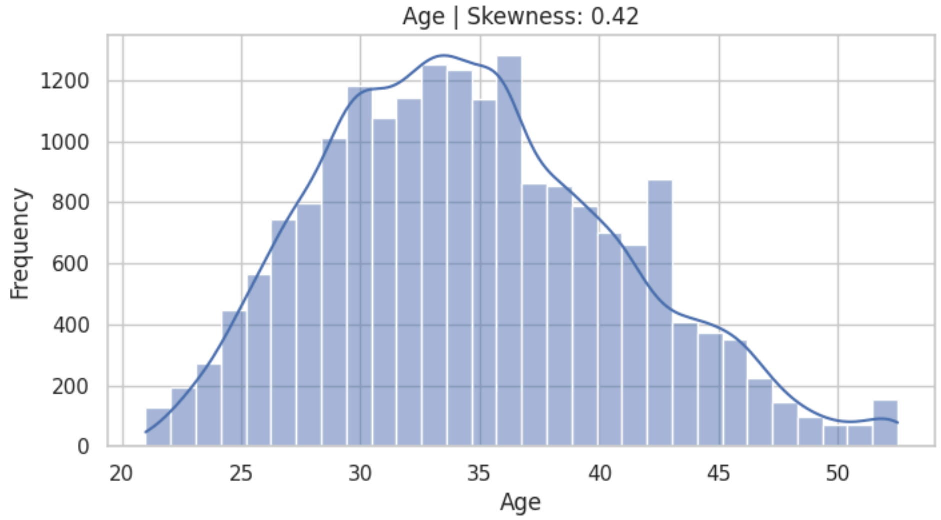
```
Age - Lower Bound: 16.50, Upper Bound: 52.50
Income - Lower Bound: -19996.00, Upper Bound: 146348.00
```

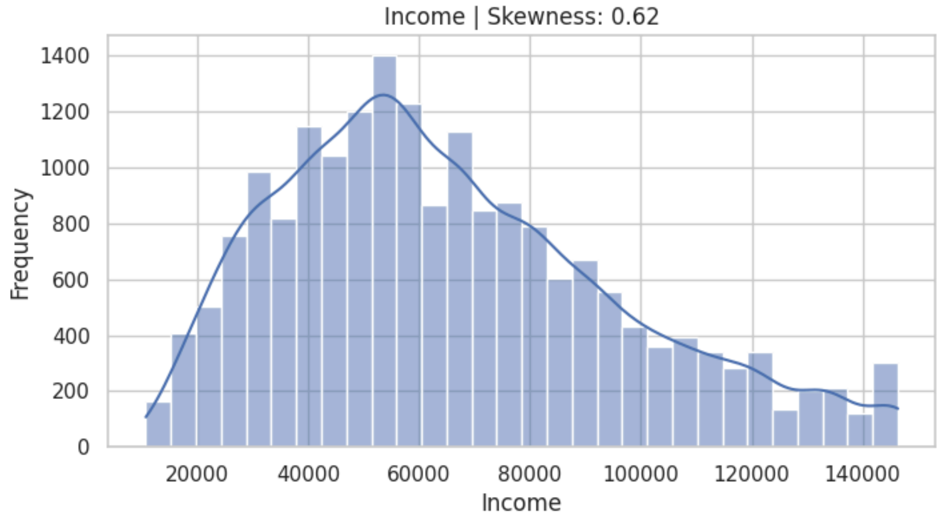
```
# re checking by box plot
for col in cols:
    plt.figure(figsize=(6, 2))
    sns.boxplot(x=df[col], color='green')
    plt.title(f'Boxplot After Capping: {col}')
    plt.show()
```

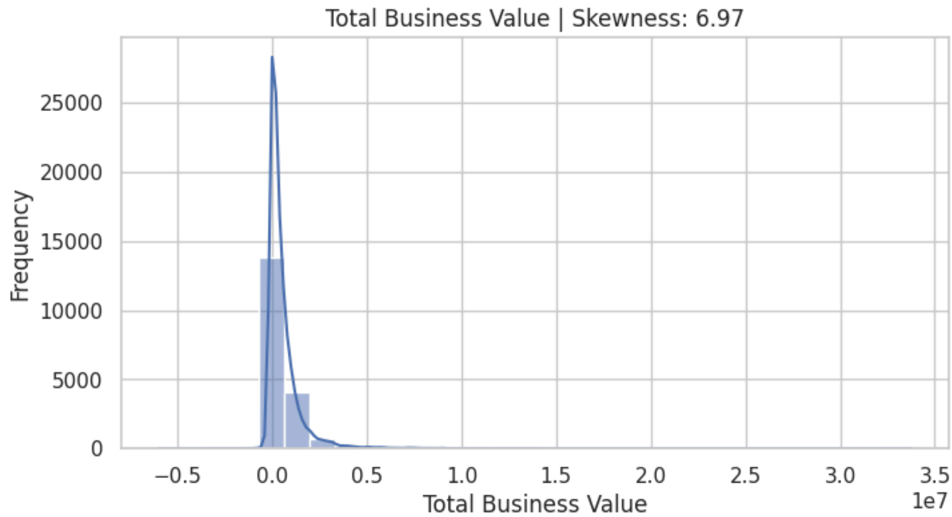


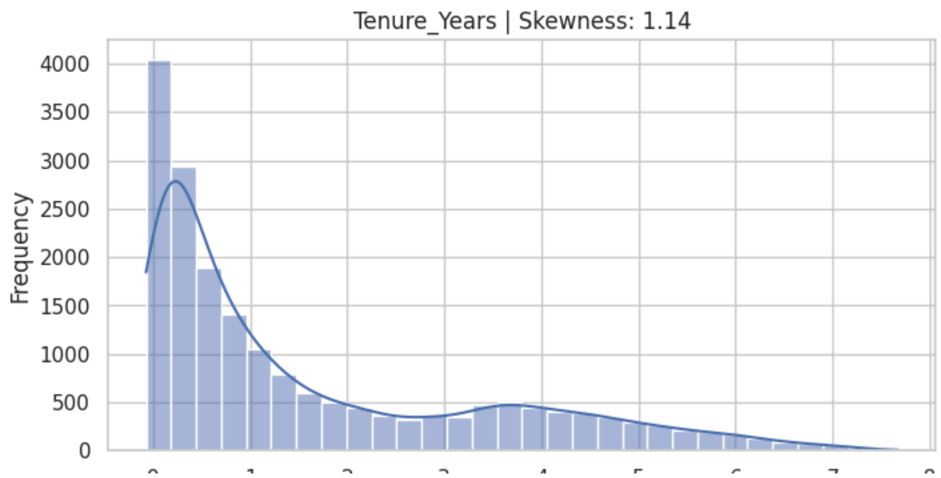



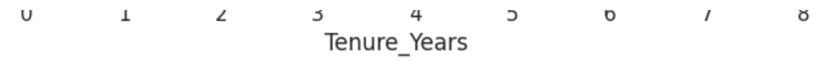

```
# cehcking the sckewness of colums
from scipy.stats import skew

continuous_vars = ['Age', 'Income', 'Total Business Value', 'Tenure_Years']

for col in continuous_vars:
    plt.figure(figsize=(8, 4))
    sns.histplot(df[col], kde=True, bins=30)
    plt.title(f'{col} | Skewness: {round(df[col].skew(), 2)}')
    plt.xlabel(col)
    plt.ylabel("Frequency")
    plt.show()
```


Age | Skewness: 0.42
Frequency
Age


Income | Skewness: 0.62
Frequency
Income


Total Business Value | Skewness: 6.97
Frequency
Total Business Value
1e7


Tenure_Years | Skewness: 1.14
Frequency




**Feature Skewness**

Age 0.42 Mildly right-skewed

Income 0.62 Moderately right-skewed

Total Business Value 6.97 Highly right-skewed

Tenure_Years 1.14 Significantly right-skewed

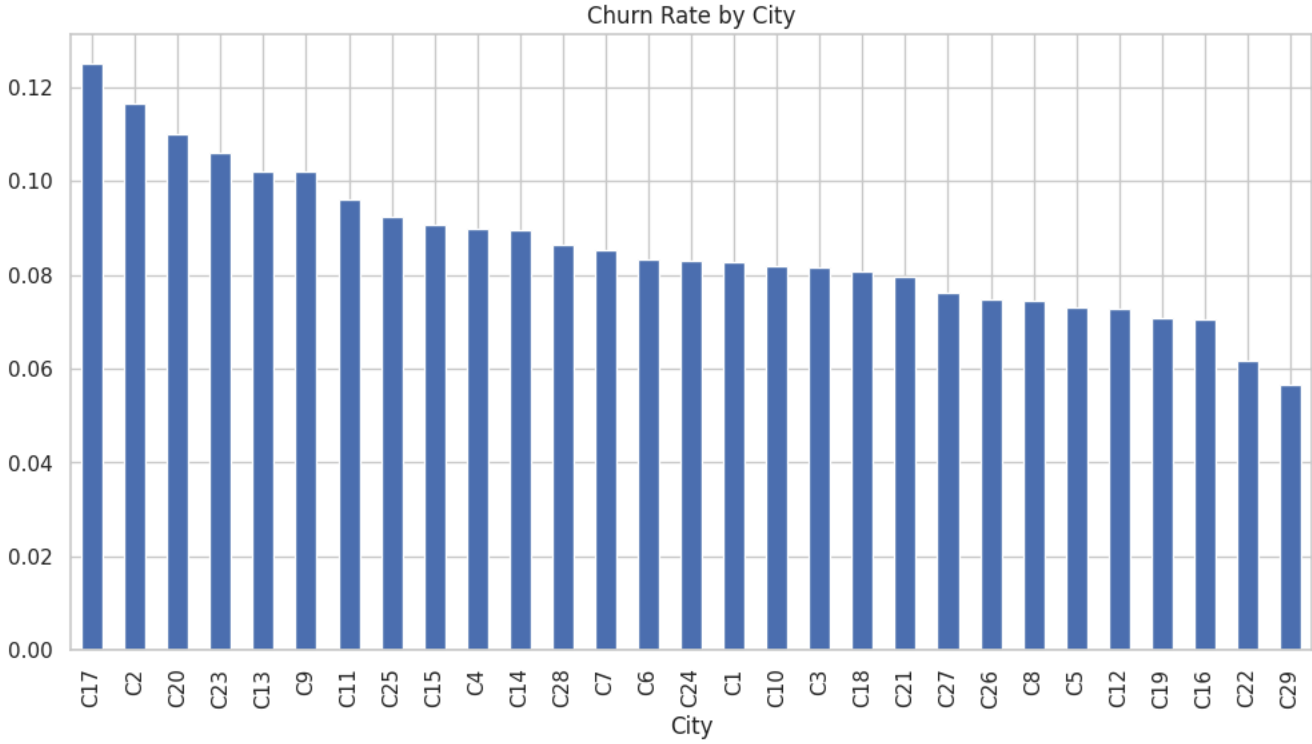
```
# flage creation
# High Business Value Driver
threshold_bv = df['Total Business Value'].quantile(0.90)
df['High_Business_Value_Flag'] = (df['Total Business Value'] >= threshold_bv).astype(int)

# Low Income Driver
threshold_income = df['Income'].quantile(0.10)
df['Low_Income_Flag'] = (df['Income'] <= threshold_income).astype(int)

# Senior Age Group Flag
df['Senior_Driver_Flag'] = (df['Age'] > 50).astype(int)

# Recent Joiner Flag
df['Recent_Joiner_Flag'] = (df['Tenure_Years'] < 1).astype(int)

# Low Rating Flag
df['Low_Rating_Flag'] = (df['Quarterly Rating'] <= 2).astype(int)
```

```
df.head(5)
```

| Gender | City | Education_Level | Income | Dateofjoining | LastWorkingDate | ... | Churn | year | month |
|---|---|---|---|---|---|---|---|---|---|
| 0.0 | C23 | 2 | 57387 | 2018-12-24 | NaT | ... | 0 | NaN | NaN |
| 0.0 | C23 | 2 | 57387 | 2018-12-24 | NaT | ... | 0 | NaN | NaN |
| 0.0 | C23 | 2 | 57387 | 2018-12-24 | 2019-03-11 | ... | 1 | 2019.0 | 3.0 |
| 0.0 | C7 | 2 | 67016 | 2020-11-06 | NaT | ... | 0 | NaN | NaN |
| 0.0 | C7 | 2 | 67016 | 2020-11-06 | NaT | ... | 0 | NaN | NaN |

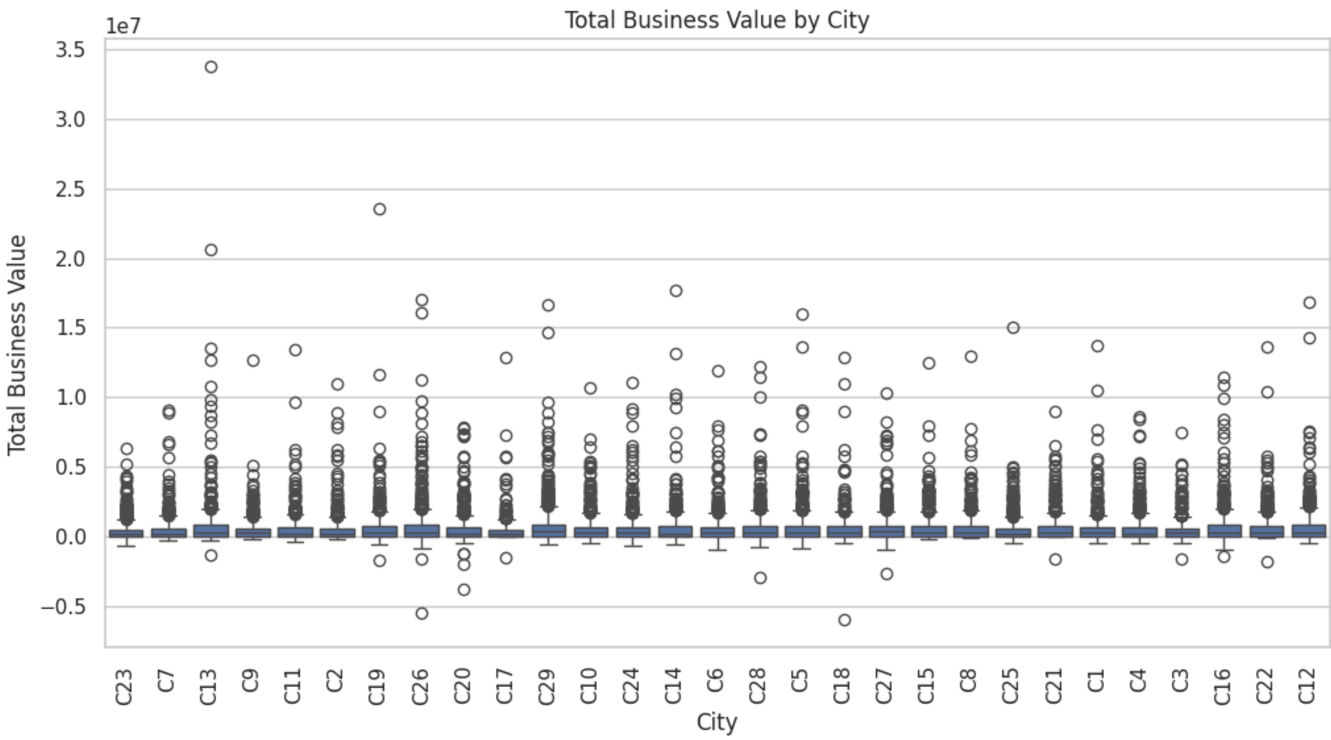
```
# Churn Rate by City

city_churn_rate = df.groupby('City')['Churn'].mean().sort_values(ascending=False)
city_churn_rate.plot(kind='bar', figsize=(12,6), title='Churn Rate by City')
```

```
<Axes: title={'center': 'Churn Rate by City'}, xlabel='City'>
```



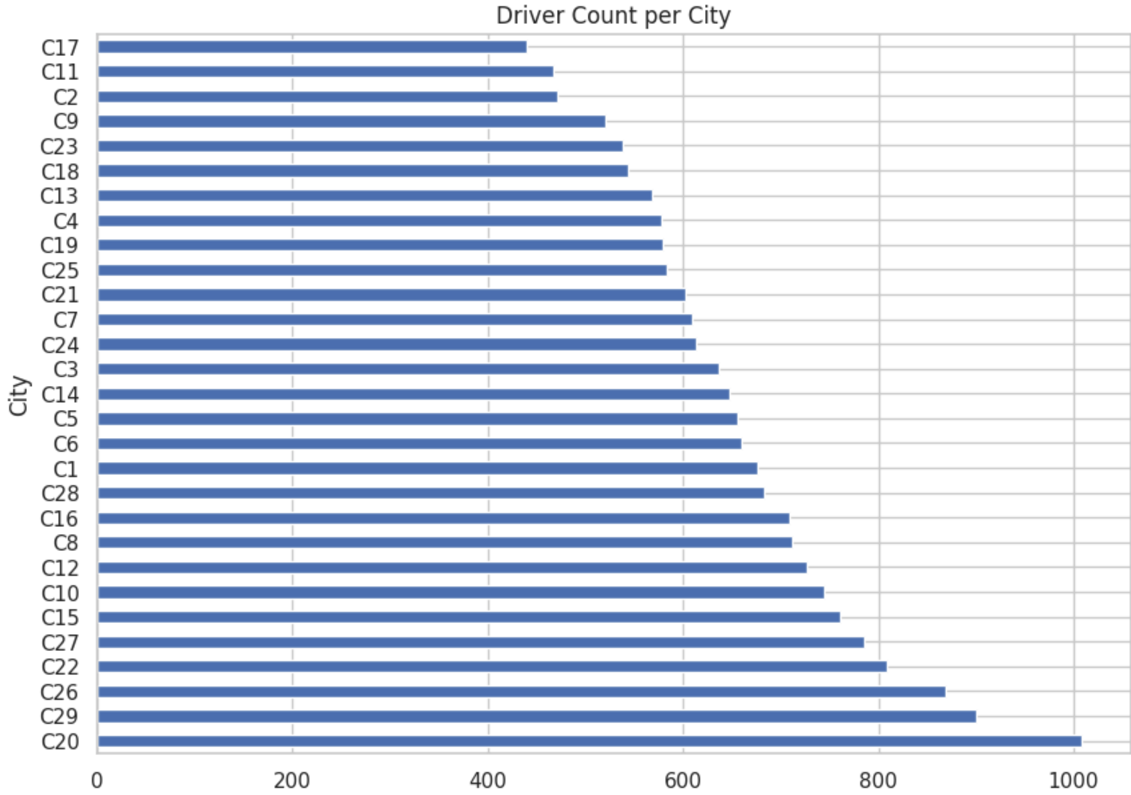

```
# Business Value Distribution by City

import seaborn as sns
plt.figure(figsize=(12,6))
sns.boxplot(x='City', y='Total Business Value', data=df)
plt.xticks(rotation=90)
plt.title('Total Business Value by City')
```

```
Text(0.5, 1.0, 'Total Business Value by City')
```



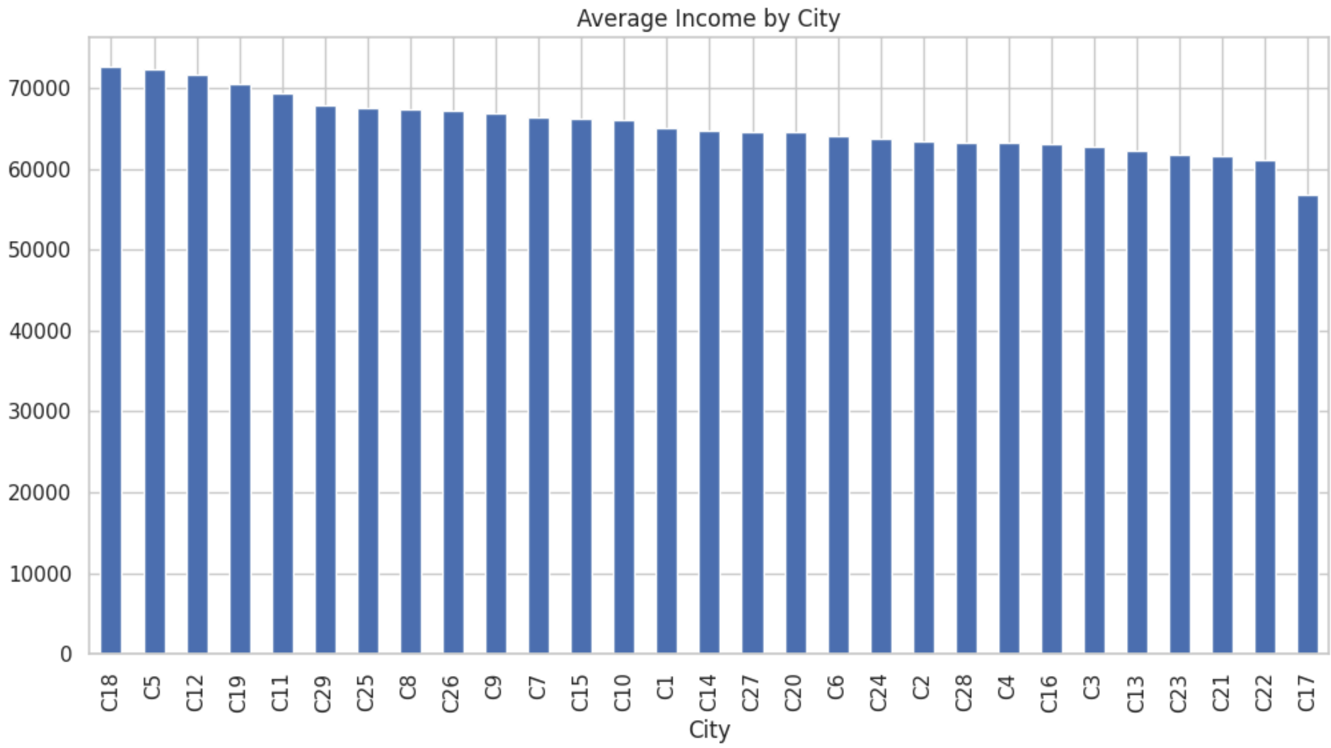

```
# Driver Count per City

df['City'].value_counts().plot(kind='barh', figsize=(10,7), title='Driver Count per City')
```

```
<Axes: title={'center': 'Driver Count per City'}, ylabel='City'>
```




```
# Average Income by City

df.groupby('City')['Income'].mean().sort_values(ascending=False).plot(kind='bar', figsize=(12,6), title='Average Income by City')
```

```
<Axes: title={'center': 'Average Income by City'}, xlabel='City'>
```



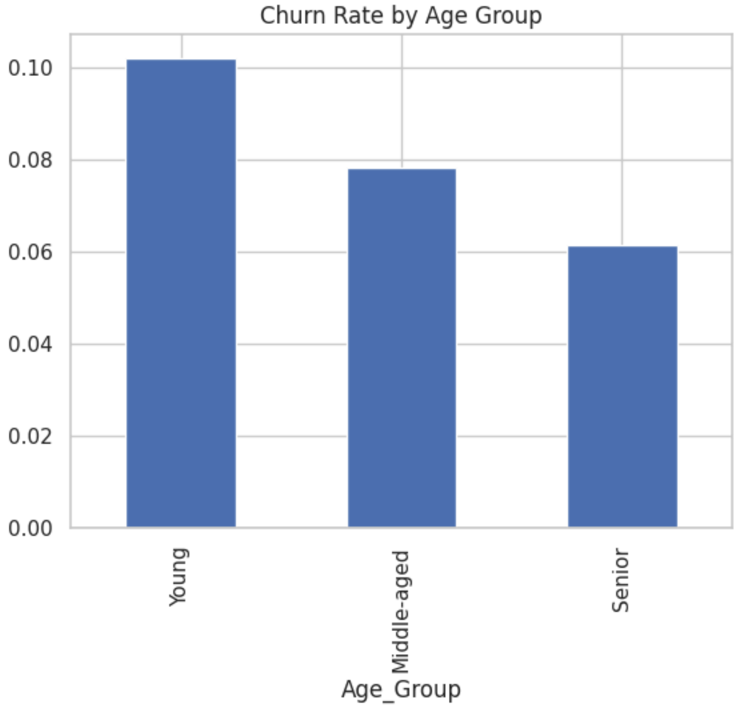

```
# binning on ages

# Define bins and labels
bins = [0, 30, 50, df['Age'].max()]  # You can use 0 to safely catch any invalid low ages
labels = ['Young', 'Middle-aged', 'Senior']

# Create the Age_Group column
df['Age_Group'] = pd.cut(df['Age'], bins=bins, labels=labels, include_lowest=True)

# Check distribution
```

```
print(df['Age_Group'].value_counts())
```

```
Age_Group
Middle-aged    13531
Young           5345
Senior           228
Name: count, dtype: int64
```

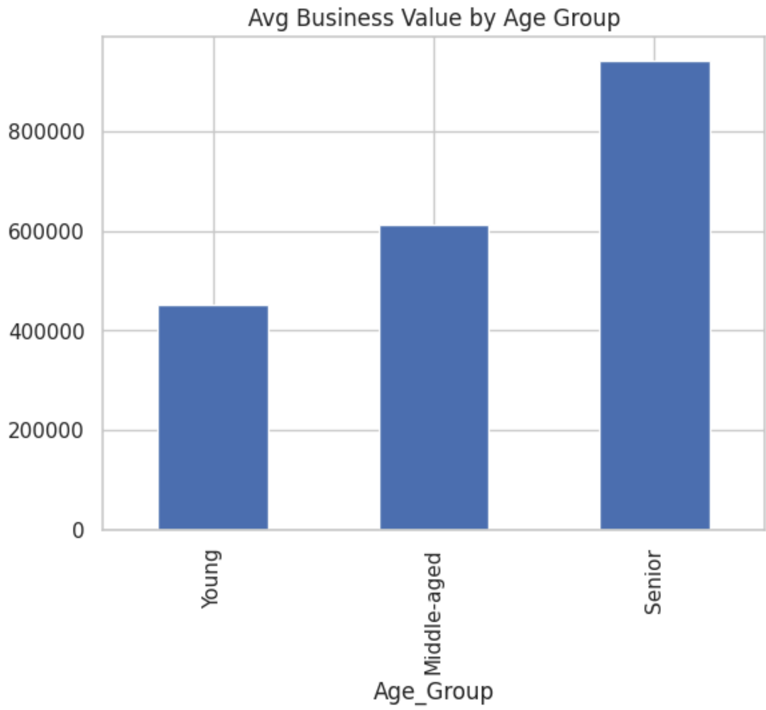
```
# Churn Rate by Age Group
df.groupby('Age_Group')['Churn'].mean().plot(kind='bar', title='Churn Rate by Age Group')
```

```
/tmp/ipython-input-1991971540.py:2: FutureWarning: The default of observed=False is deprecated and will be changed to True in a futu
  df.groupby('Age_Group')['Churn'].mean().plot(kind='bar', title='Churn Rate by Age Group')
<Axes: title={'center': 'Churn Rate by Age Group'}, xlabel='Age_Group'>
```




```
# Business Value by Age Group:

df.groupby('Age_Group')['Total Business Value'].mean().plot(kind='bar', title='Avg Business Value by Age Group')
```

```
/tmp/ipython-input-243265430.py:3: FutureWarning: The default of observed=False is deprecated and will be changed to True in a futur
  df.groupby('Age_Group')['Total Business Value'].mean().plot(kind='bar', title='Avg Business Value by Age Group')
<Axes: title={'center': 'Avg Business Value by Age Group'}, xlabel='Age_Group'>
```




Columns to Drop with Reasoning

**Unnamed: 0**-Just a row index, not useful for modeling

**Driver_ID**- Unique identifier, not predictive

**MMM-YY** - Monthly indicator, temporal leakage possible, already captured in Tenure

**Dateofjoining**- Date field, not model-friendly — already encoded in Tenure_Years **LastWorkingDate**- Missing for most active drivers; can leak churn info

**EndDate**- Could overlap with churn or tenure — may lead to data leakage

**year, month**- Mostly missing, derived from LastWorkingDate; redundant

These are potentially useful features:

Age

Gender

City

Education_Level

Income

Joining Designation

Grade

Total Business Value

Quarterly Rating

Tenure_Years

All flag variables:

High_Business_Value_Flag

Low_Income_Flag

Senior_Driver_Flag

Recent_Joiner_Flag

Low_Rating_Flag

Age_Group (as categorical)

Churn (target)

```
cols_to_drop = [
    'Unnamed: 0', 'Driver_ID', 'MMM-YY', 'Dateofjoining',
    'LastWorkingDate', 'year', 'month', 'EndDate'
]

df_model = df.drop(columns=cols_to_drop)
```

```
# for model preparation we need to split the data set

from sklearn.model_selection import train_test_split

# Features and target
X = df_model.drop(columns=['Churn'])
y = df['Churn']

# 80-20 stratified split
X_train, X_test, y_train, y_test = train_test_split(
    X, y,
    test_size=0.2,
    stratify=y,
    random_state=42
)
```

```
X.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 19104 entries, 0 to 19103
Data columns (total 16 columns):
 #   Column                   Non-Null Count  Dtype   
---  ------                   --------------  -----   
 0   Age                      19104 non-null  float64 
 1   Gender                   19104 non-null  float64 
 2   City                     19104 non-null  object  
 3   Education_Level          19104 non-null  int64   
 4   Income                   19104 non-null  int64   
 5   Joining Designation      19104 non-null  int64   
 6   Grade                    19104 non-null  int64   
 7   Total Business Value     19104 non-null  int64   
 8   Quarterly Rating         19104 non-null  int64   
 9   Tenure_Years             19104 non-null  float64 
 10  High_Business_Value_Flag 19104 non-null  int64   
 11  Low_Income_Flag          19104 non-null  int64   
 12  Senior_Driver_Flag       19104 non-null  int64   
 13  Recent_Joiner_Flag       19104 non-null  int64   
 14  Low_Rating_Flag          19104 non-null  int64   
 15  Age_Group                19104 non-null  category
dtypes: category(1), float64(3), int64(11), object(1)
memory usage: 2.2+ MB
```

```
df_model["City"].nunique()
```

```
29
```

Best Choice for Gradient Boosting: Target Encoding Since we are using GradientBoostingClassifier (tree-based model), Target Encoding is preferred when:

Cardinality is high (like your 29-city case)

we can control leakage via fit on train, transform on train and test

```
!pip install category_encoders
```

```
Collecting category_encoders
  Downloading category_encoders-2.8.1-py3-none-any.whl.metadata (7.9 kB)
Requirement already satisfied: numpy>=1.14.0 in /usr/local/lib/python3.11/dist-packages (from category_encoders) (2.0.2)
Requirement already satisfied: pandas>=1.0.5 in /usr/local/lib/python3.11/dist-packages (from category_encoders) (2.2.2)
Requirement already satisfied: patsy>=0.5.1 in /usr/local/lib/python3.11/dist-packages (from category_encoders) (1.0.1)
Requirement already satisfied: scikit-learn>=1.6.0 in /usr/local/lib/python3.11/dist-packages (from category_encoders) (1.6.1)
Requirement already satisfied: scipy>=1.0.0 in /usr/local/lib/python3.11/dist-packages (from category_encoders) (1.16.1)
Requirement already satisfied: statsmodels>=0.9.0 in /usr/local/lib/python3.11/dist-packages (from category_encoders) (0.14.5)
Requirement already satisfied: python-dateutil>=2.8.2 in /usr/local/lib/python3.11/dist-packages (from pandas>=1.0.5->category_encod
Requirement already satisfied: pytz>=2020.1 in /usr/local/lib/python3.11/dist-packages (from pandas>=1.0.5->category_encoders) (2025
Requirement already satisfied: tzdata>=2022.7 in /usr/local/lib/python3.11/dist-packages (from pandas>=1.0.5->category_encoders) (20
Requirement already satisfied: joblib>=1.2.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn>=1.6.0->category_encoders
Requirement already satisfied: threadpoolctl>=3.1.0 in /usr/local/lib/python3.11/dist-packages (from scikit-learn>=1.6.0->category_e
Requirement already satisfied: packaging>=21.3 in /usr/local/lib/python3.11/dist-packages (from statsmodels>=0.9.0->category_encoder
```

```
Requirement already satisfied: six>=1.5 in /usr/local/lib/python3.11/dist-packages (from python-dateutil>=2.8.2->pandas>=1.0.5->cate
Downloading category_encoders-2.8.1-py3-none-any.whl (85 kB)
   ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 85.7/85.7 kB 2.0 MB/s eta 0:00:00
Installing collected packages: category_encoders
Successfully installed category_encoders-2.8.1
```

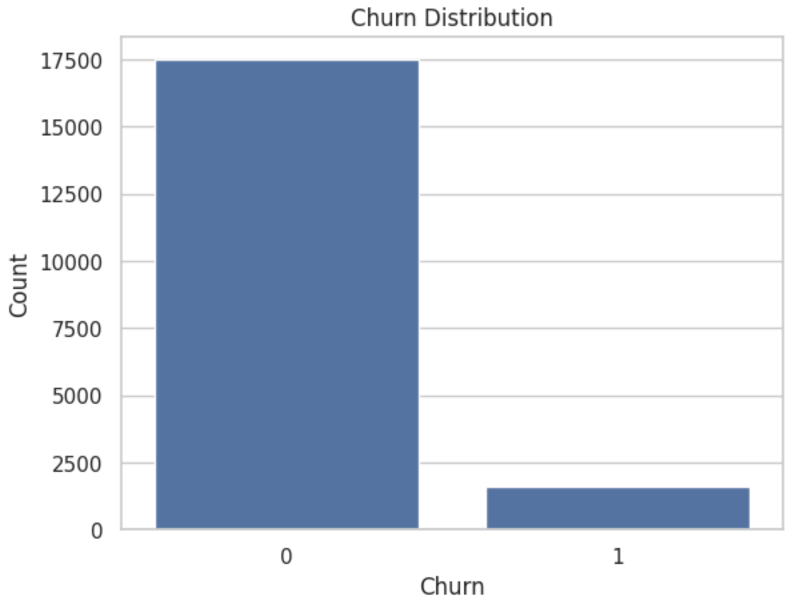
```
from category_encoders import TargetEncoder
from sklearn.model_selection import train_test_split

# Initialize encoder
target_enc = TargetEncoder(cols=['City'])

# Fit and transform on training set
X_train['City'] = target_enc.fit_transform(X_train['City'], y_train)

# Transform test set only
X_test['City'] = target_enc.transform(X_test['City'])
```

```
# encoding for Age_group

df_model['Age_Group'].unique()
# ['Young', 'Middle-aged', 'Senior']
```

```
['Young', 'Middle-aged', 'Senior']
Categories (3, object): ['Young' < 'Middle-aged' < 'Senior']
```

```
from sklearn.preprocessing import LabelEncoder

le = LabelEncoder()
X_train['Age_Group'] = le.fit_transform(X_train['Age_Group'])
X_test['Age_Group'] = le.transform(X_test['Age_Group'])
```

```
# EDA: Check Class Imbalance

# Plot churn distribution
sns.countplot(x=y)
plt.title("Churn Distribution")
plt.xlabel("Churn")
plt.ylabel("Count")
plt.show()

# Print class ratio
print("Class distribution:")
print(y.value_counts(normalize=True))
```




```
Class distribution:
Churn
0    0.91541
1    0.08459
Name: proportion, dtype: float64
```

```
# handling the imbalanced data by SMOTE
```

```
smote = SMOTE(random_state=42)
X_resampled, y_resampled = smote.fit_resample(X_train, y_train)
```

```
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.ensemble import RandomForestClassifier, BaggingClassifier, GradientBoostingClassifier
from sklearn.metrics import classification_report, confusion_matrix
from imblearn.over_sampling import SMOTE
import xgboost as xgb
import lightgbm as lgb
```

```
# RandomForestClassifier with Class Weights

from sklearn.ensemble import RandomForestClassifier
from sklearn.metrics import classification_report, roc_auc_score


rf_params = {
    'n_estimators': [100, 200],
    'max_depth': [10, 20],
    'max_features': ['sqrt', 'log2']
}

rf_clf = GridSearchCV(
    RandomForestClassifier(class_weight='balanced', random_state=42),
    rf_params,
    scoring='f1',
    cv=5,
    n_jobs=-1,
    verbose=2
)

rf_clf.fit(X_resampled, y_resampled)
print("Best RF Params:", rf_clf.best_params_)

rf_final = rf_clf.best_estimator_
rf_pred = rf_final.predict(X_test)
print(" Random Forest:\n", classification_report(y_test, rf_pred))
```

```
Fitting 5 folds for each of 8 candidates, totalling 40 fits
Best RF Params: {'max_depth': 20, 'max_features': 'sqrt', 'n_estimators': 200}
 Random Forest:
               precision    recall  f1-score   support

           0       0.95      0.94      0.95      3498
           1       0.43      0.47      0.45       323

    accuracy                           0.90      3821
   macro avg       0.69      0.71      0.70      3821
weighted avg       0.91      0.90      0.90      3821
```

```
# BaggingClassifier with base estimator = DecisionTreeClassifier

from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import GridSearchCV

# Define base estimator (with class_weight='balanced' to handle imbalance)
base_tree = DecisionTreeClassifier(class_weight='balanced', random_state=42)

# Define hyperparameter grid
bag_params = {
    'n_estimators': [50, 100],
    'max_samples': [0.5, 1.0],
    'max_features': [0.5, 1.0],
    'estimator__max_depth': [5, 10, None],   # tuning DecisionTree
    'estimator__min_samples_split': [2, 5]
}

# Create the bagging classifier with base estimator
bag_clf = GridSearchCV(
    BaggingClassifier(estimator=base_tree, random_state=42),
    bag_params,
    scoring='f1',         # or 'f1_macro' for multi-class
    cv=5,
    n_jobs=-1,
    verbose=1
)
```

```
# Fit on SMOTE-resampled data
bag_clf.fit(X_resampled, y_resampled)

# Best params
print(" Best Bagging Params:", bag_clf.best_params_)

# Evaluate on test set
from sklearn.metrics import classification_report
y_pred = bag_clf.predict(X_test)
print(classification_report(y_test, y_pred))
```

```
Fitting 5 folds for each of 48 candidates, totalling 240 fits
✅ Best Bagging Params: {'estimator__max_depth': None, 'estimator__min_samples_split': 5, 'max_features': 0.5, 'max_samples': 1.0,
              precision    recall  f1-score   support

           0       0.92      0.99      0.95      3498
           1       0.21      0.03      0.05       323

    accuracy                           0.91      3821
   macro avg       0.56      0.51      0.50      3821
weighted avg       0.86      0.91      0.88      3821
```

```
# Grid Search for Gradient Boosting

gb_params = {
    'n_estimators': [100, 200],
    'learning_rate': [0.05, 0.1],
    'max_depth': [3, 5]
}

gb_clf = GridSearchCV(
    GradientBoostingClassifier(random_state=42),
    gb_params,
    scoring='f1',
    cv=5,
    n_jobs=-1
)

gb_clf.fit(X_resampled, y_resampled)
print("Best GB Params:", gb_clf.best_params_)

gb_final = gb_clf.best_estimator_
gb_pred = gb_final.predict(X_test)
print("Gradient Boosting:\n", classification_report(y_test, gb_pred))
```

```
Best GB Params: {'learning_rate': 0.05, 'max_depth': 5, 'n_estimators': 100}
Gradient Boosting:
               precision    recall  f1-score   support

           0       0.97      0.91      0.94      3498
           1       0.42      0.67      0.51       323

    accuracy                           0.89      3821
   macro avg       0.69      0.79      0.73      3821
weighted avg       0.92      0.89      0.90      3821
```

```
# Grid Search for XGBoost

xgb_clf = xgb.XGBClassifier(random_state=42, use_label_encoder=False, eval_metric='logloss')

xgb_params = {
    'n_estimators': [100, 200],
    'learning_rate': [0.05, 0.1],
    'max_depth': [3, 5],
    'scale_pos_weight': [1, 2]  # for imbalance handling
}

xgb_grid = GridSearchCV(
    xgb_clf,
    xgb_params,
    scoring='f1',
    cv=5,
    n_jobs=-1
)

xgb_grid.fit(X_resampled, y_resampled)
```

```
print("Best XGB Params:", xgb_grid.best_params_)

xgb_final = xgb_grid.best_estimator_
xgb_pred = xgb_final.predict(X_test)
print(" XGBoost:\n", classification_report(y_test, xgb_pred))
```

```
/usr/local/lib/python3.11/dist-packages/xgboost/training.py:183: UserWarning: [20:02:26] WARNING: /workspace/src/learner.cc:738:
Parameters: { "use_label_encoder" } are not used.

  bst.update(dtrain, iteration=i, fobj=obj)
Best XGB Params: {'learning_rate': 0.1, 'max_depth': 5, 'n_estimators': 100, 'scale_pos_weight': 1}
 XGBoost:
               precision    recall  f1-score   support

           0       0.96      0.93      0.95      3498
           1       0.45      0.60      0.51       323

    accuracy                           0.90      3821
   macro avg       0.71      0.77      0.73      3821
weighted avg       0.92      0.90      0.91      3821
```

```
# Required Imports
import lightgbm as lgb
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import classification_report

# Define base LightGBM classifier with balanced class weight
lgb_clf = lgb.LGBMClassifier(class_weight='balanced', random_state=42)

# Define hyperparameter grid
lgb_params = {
    'n_estimators': [100, 200],
    'learning_rate': [0.05, 0.1],
    'max_depth': [3, 5]
}

#  Grid Search with F1 Score (best for imbalanced classification)
lgb_grid = GridSearchCV(
    estimator=lgb_clf,
    param_grid=lgb_params,
    scoring='f1',
    cv=5,
    n_jobs=-1,
    verbose=1
)

#  Fit on resampled (balanced) training data
lgb_grid.fit(X_resampled, y_resampled)

#  Best hyperparameters
print("Best LGBM Params:", lgb_grid.best_params_)

#  Final model
lgb_final = lgb_grid.best_estimator_

#  Predict on test set
lgb_pred = lgb_final.predict(X_test)

#  Classification report
from sklearn.metrics import classification_report
print("LightGBM Classification Report:\n")
print(classification_report(y_test, lgb_pred, digits=4))
```

```
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
```

```
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
[LightGBM] [Warning] No further splits with positive gain, best gain: -inf
Best LGBM Params: {'learning_rate': 0.05, 'max_depth': 5, 'n_estimators': 100}
LightGBM Classification Report:

              precision    recall  f1-score   support

           0     0.9675    0.9099    0.9378      3498
           1     0.4068    0.6687    0.5059       323

    accuracy                         0.8896      3821
   macro avg     0.6871    0.7893    0.7218      3821
weighted avg     0.9201    0.8896    0.9013      3821
```

```
# Evaluate All Models

from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

def evaluate_model(name, y_true, y_pred):
    return {
        'Model': name,
        'Accuracy': accuracy_score(y_true, y_pred),
        'Precision': precision_score(y_true, y_pred, zero_division=0),
        'Recall': recall_score(y_true, y_pred),
        'F1 Score': f1_score(y_true, y_pred)
    }



# Collect All Results
results = []

results.append(evaluate_model("Random Forest", y_test, rf_pred))
results.append(evaluate_model("Bagging Classifier", y_test, y_pred))
results.append(evaluate_model("Gradient Boosting", y_test, gb_pred))
results.append(evaluate_model("XGBoost", y_test, xgb_pred))
results.append(evaluate_model("LightGBM", y_test, lgb_pred))



# Create Comparison Table

import pandas as pd

results_df = pd.DataFrame(results)
results_df = results_df.sort_values(by='F1 Score', ascending=False).reset_index(drop=True)

print("📈 Model Comparison:")
print(results_df)
```

```
📈 Model Comparison:
                Model  Accuracy  Precision    Recall  F1 Score
0             XGBoost  0.904214   0.450116  0.600619  0.514589
1   Gradient Boosting  0.892698   0.415861  0.665635  0.511905
2            LightGBM  0.889558   0.406780  0.668731  0.505855
3       Random Forest  0.902643   0.430199  0.467492  0.448071
4  Bagging Classifier  0.908924   0.209302  0.027864  0.049180
```

```
#  Recommend the Best Model

best_model_name = results_df.iloc[0]['Model']
print(f"\n Recommended Best Model: {best_model_name}")
```

```
 Recommended Best Model: XGBoost
```

```
# What percentage of drivers have received a quarterly rating of 5?

rating_5_pct = (df['Quarterly Rating'] == 5).mean() * 100
print(f"Percentage of drivers with rating 5: {rating_5_pct:.2f}%")
```

```
Percentage of drivers with rating 5: 0.00%
```

```
# Comment on the correlation between Age and Quarterly Rating.

correlation = df[['Age', 'Quarterly Rating']].corr().loc['Age', 'Quarterly Rating']
print(f"Correlation between Age and Rating: {correlation:.2f}")
```

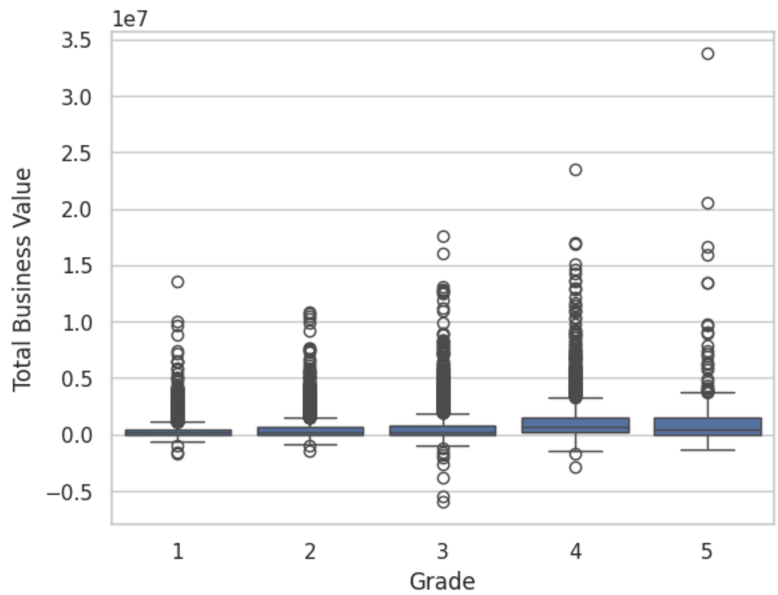
```
Correlation between Age and Rating: 0.17
```

```
# Name the city which showed the most improvement in Quarterly Rating over the past year

df['Year'] = df['MMM-YY'].dt.year
city_year_rating = df.groupby(['City', 'Year'])['Quarterly Rating'].mean().unstack()

improvement = city_year_rating.diff(axis=1).iloc[:, -1]  # Difference from previous year
most_improved_city = improvement.idxmax()
print(f"Most improved city: {most_improved_city}")
```

```
Most improved city: C29
```

```
# Drivers with a Grade of ‘A’ are more likely to have a higher Total Business Value. (T/F)


sns.boxplot(data=df, x='Grade', y='Total Business Value')

grade_A_mean = df[df['Grade'] == 'A']['Total Business Value'].mean()
others_mean = df[df['Grade'] != 'A']['Total Business Value'].mean()
print(f"Grade A Mean: {grade_A_mean}, Others: {others_mean}")
```

```
Grade A Mean: nan, Others: 571662.074958124
```

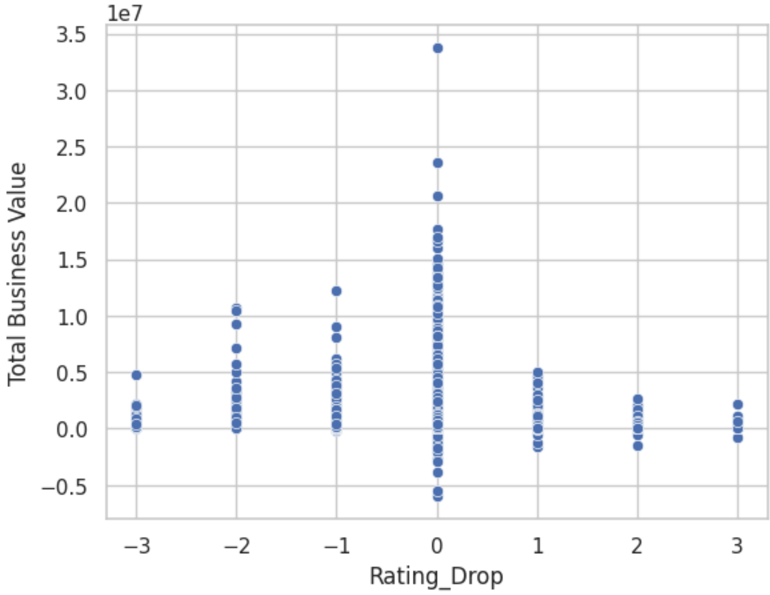



If Grade A mean is significantly higher, then True.

```
# If a driver's Quarterly Rating drops significantly, how does it impact their Total Business Value in the subsequent period?
```

```
df = df.sort_values(by=['Driver_ID', 'MMM-YY'])
df['Prev_Rating'] = df.groupby('Driver_ID')['Quarterly Rating'].shift(1)
df['Rating_Drop'] = df['Prev_Rating'] - df['Quarterly Rating']

# Impact on business value
import matplotlib.pyplot as plt
sns.scatterplot(data=df, x='Rating_Drop', y='Total Business Value')
```

```
<Axes: xlabel='Rating_Drop', ylabel='Total Business Value'>
```




Look for trend: big drop → lower business value?

Which metric should Ola focus on for driver retention? If retaining true churners is critical: use Recall

If you want to avoid false alarms (non-churners wrongly flagged): focus on Precision

If balanced concern: use F1 Score

ROC AUC: good for overall ranking but not threshold-based

Recommendation: Recall or F1 Score

How does the gap in precision and recall affect Ola's relationship with drivers/customers? High Precision, Low Recall: Missing many drivers who might churn → Late retention efforts

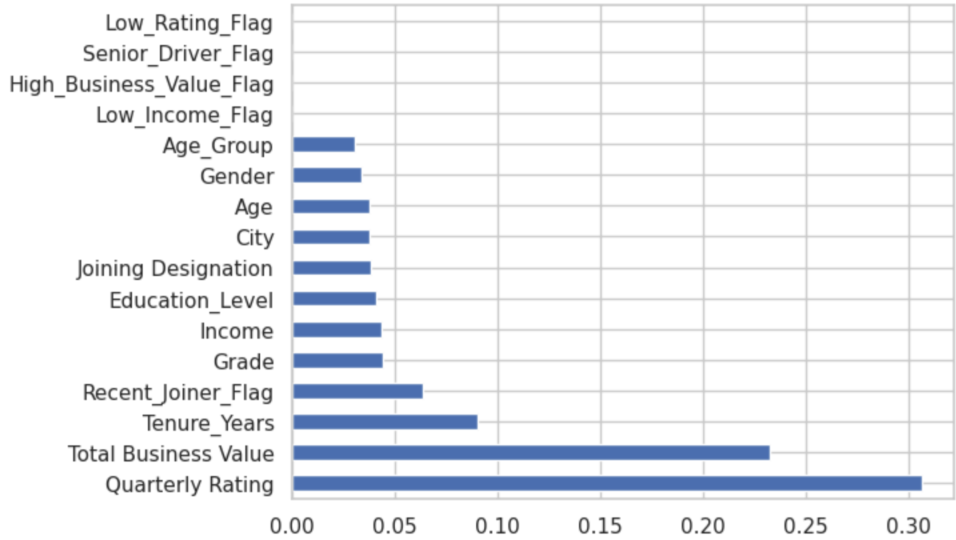
High Recall, Low Precision: Many false alarms → Frustrated loyal drivers

Balanced F1: Best for both

→ Misalignment can lead to driver dissatisfaction or operational inefficiency.

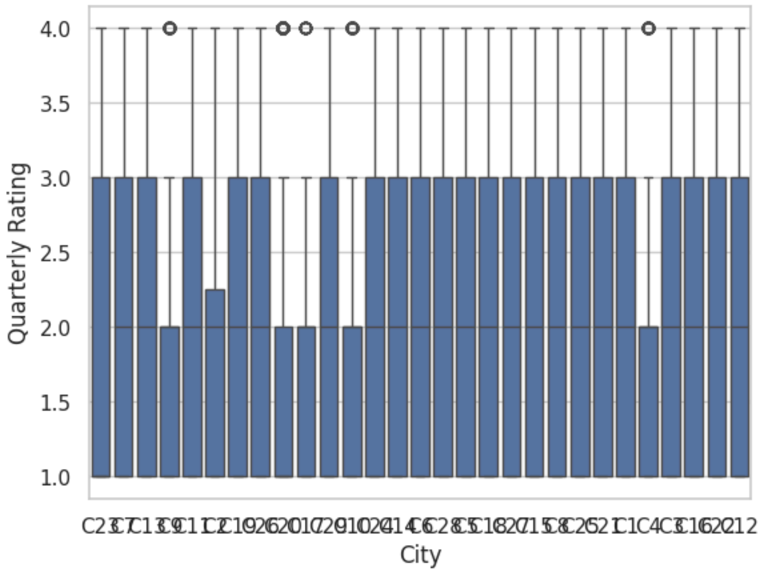
```
# Lesser-discussed features that impact Quarterly Rating

from xgboost import XGBClassifier
model = XGBClassifier()
model.fit(X_train, y_train)

importances = pd.Series(model.feature_importances_, index=X_train.columns)
importances.sort_values(ascending=False).plot(kind='barh')
```

```
<Axes: >
```




```
# Will the driver's performance be affected by the City they operate in?

sns.boxplot(data=df, x='City', y='Quarterly Rating')
```

```
<Axes: xlabel='City', ylabel='Quarterly Rating'>
```



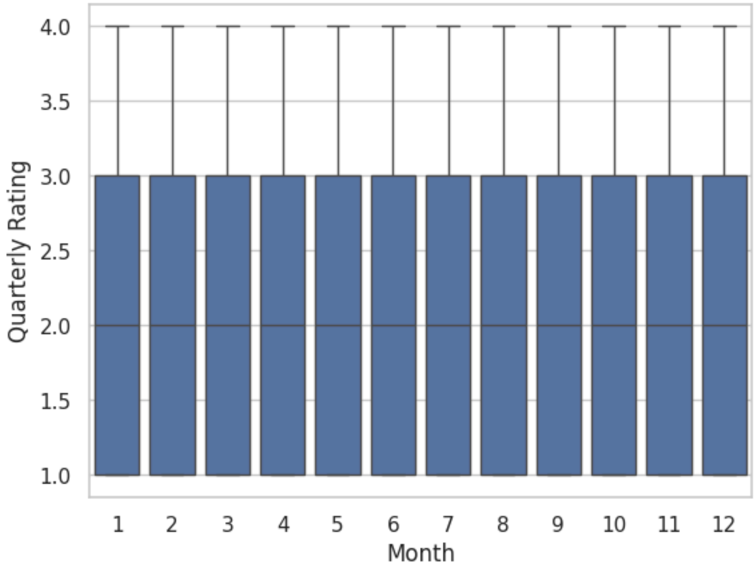

```
from scipy.stats import f_oneway
groups = [group['Quarterly Rating'].values for name, group in df.groupby('City')]
_, p_value = f_oneway(*groups)
print(f"ANOVA p-value: {p_value}")
```

```
ANOVA p-value: 2.812541883697336e-29
```

p < 0.05, then Yes, city affects performance.

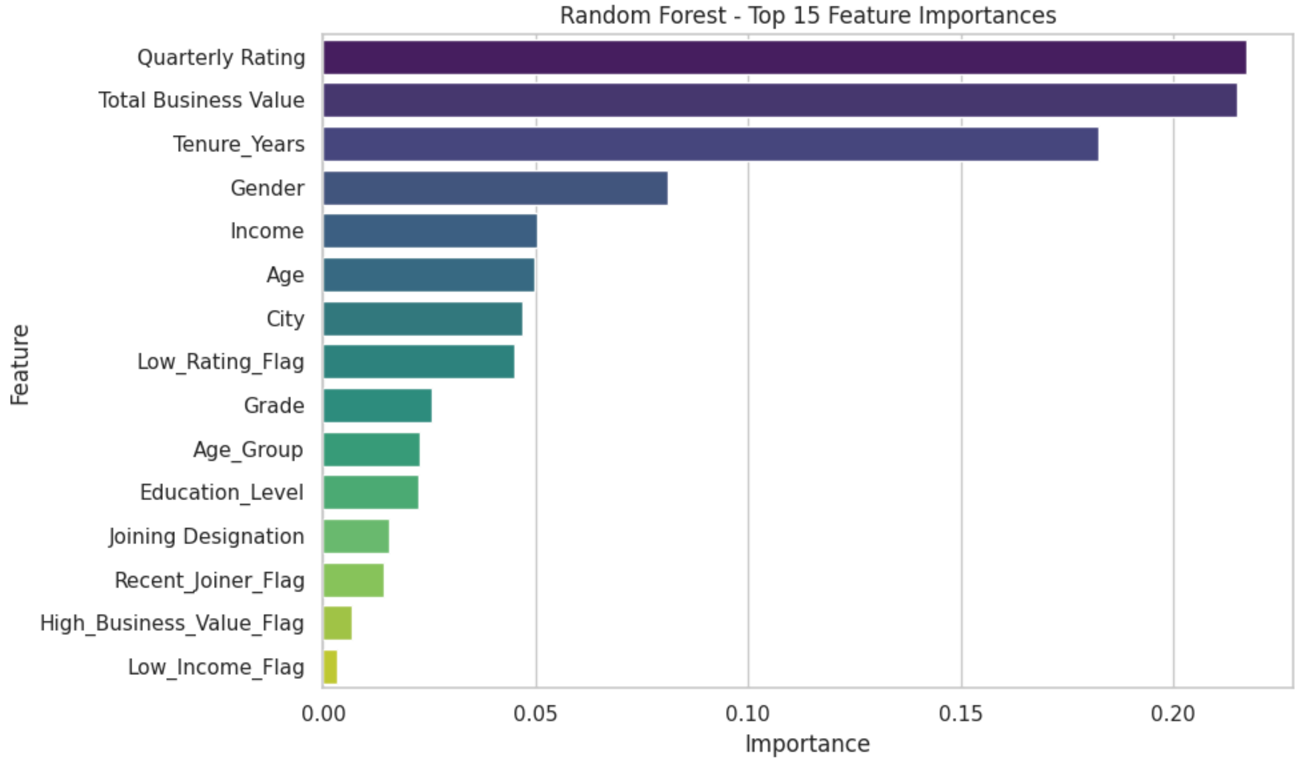
```
# Analyze seasonality in the driver's ratings

df['Month'] = df['MMM-YY'].dt.month
sns.boxplot(data=df, x='Month', y='Quarterly Rating')
```

<Axes: xlabel='Month', ylabel='Quarterly Rating'>

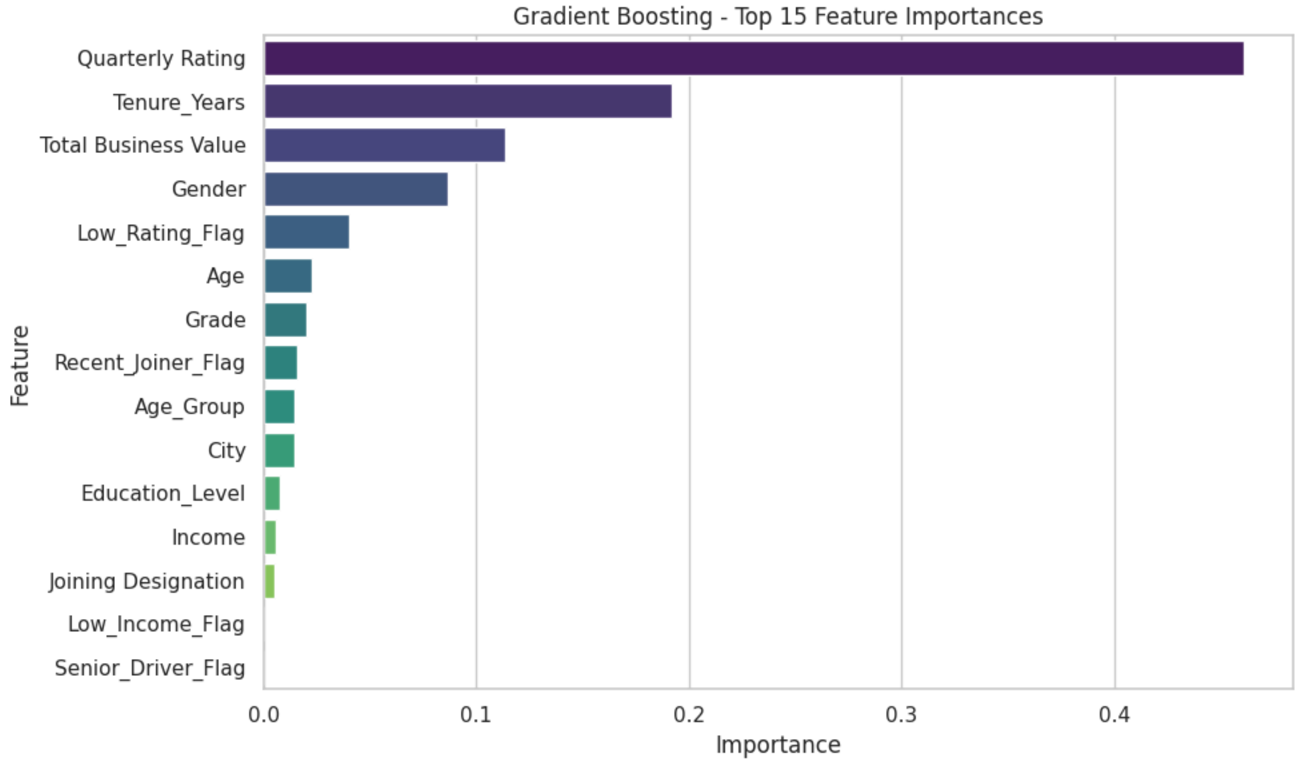



```
# features importatnce
import matplotlib.pyplot as plt
import seaborn as sns
import pandas as pd

# Assuming X_train or X_resampled is your training data
feature_names = X_resampled.columns  # or X_train.columns if used instead

# Create a function to plot feature importance
def plot_feature_importance(model, model_name):
    if hasattr(model, 'feature_importances_'):
        importances = model.feature_importances_
    else:
        raise ValueError(f"{model_name} does not support feature_importances_")

    # Create DataFrame for visualization
    feat_df = pd.DataFrame({
        'Feature': feature_names,
        'Importance': importances
    }).sort_values(by='Importance', ascending=False)

    plt.figure(figsize=(10, 6))
    sns.barplot(x='Importance', y='Feature', data=feat_df.head(15), palette='viridis')
    plt.title(f'{model_name} - Top 15 Feature Importances')
    plt.tight_layout()
    plt.show()

# Plotting for each final tuned model
plot_feature_importance(rf_final, "Random Forest")
plot_feature_importance(gb_final, "Gradient Boosting")
plot_feature_importance(xgb_final, "XGBoost")
plot_feature_importance(lgb_final, "LightGBM")
```

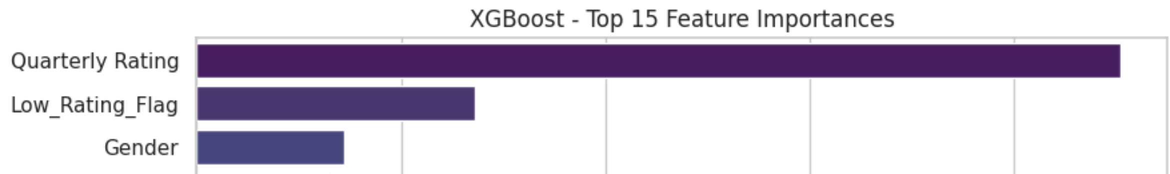
```
/tmp/ipython-input-2311097265.py:23: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `y` variable to `hue` and set `le

  sns.barplot(x='Importance', y='Feature', data=feat_df.head(15), palette='viridis')
```

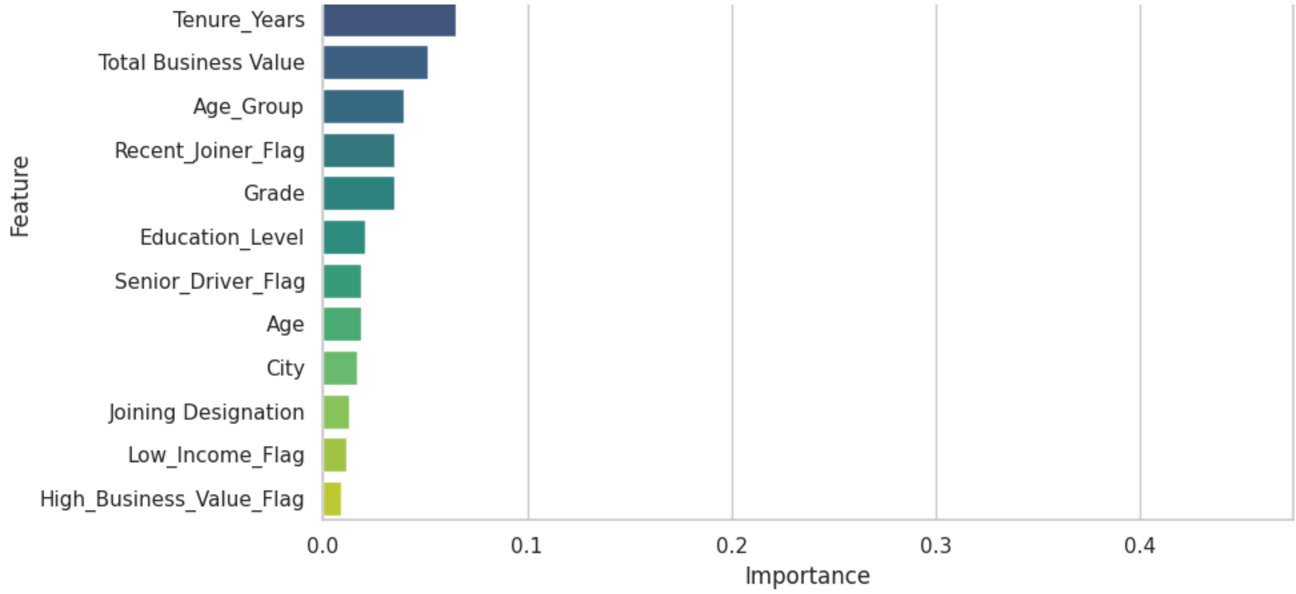



```
/tmp/ipython-input-2311097265.py:23: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `y` variable to `hue` and set `le

  sns.barplot(x='Importance', y='Feature', data=feat_df.head(15), palette='viridis')
```

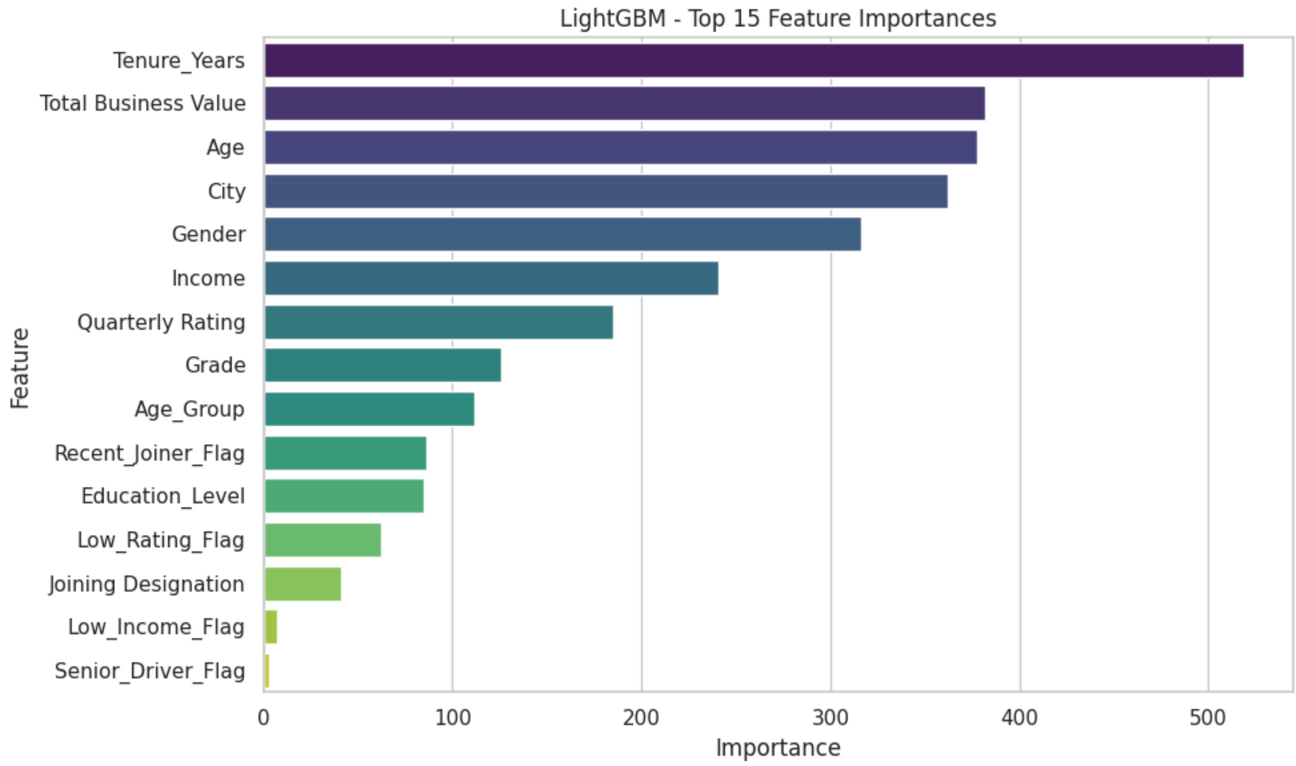



```
/tmp/ipython-input-2311097265.py:23: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `y` variable to `hue` and set `le

  sns.barplot(x='Importance', y='Feature', data=feat_df.head(15), palette='viridis')
```






```
/tmp/ipython-input-2311097265.py:23: FutureWarning:

Passing `palette` without assigning `hue` is deprecated and will be removed in v0.14.0. Assign the `y` variable to `hue` and set `le

  sns.barplot(x='Importance', y='Feature', data=feat_df.head(15), palette='viridis')
```



Start coding or generate with AI.

**Trade-Off Analysis** a. Recruiting More Educated Drivers Pros:

May lead to better professionalism, communication, and adherence to protocols.

Educated drivers may handle tech platforms more efficiently (app usage, navigation).

Cons:

Higher salaries or incentives may be required.

Limited pool in certain cities, especially Tier 2/3 areas.

May not guarantee better customer experience or loyalty without soft-skill training.

Conclusion: The cost of recruiting highly educated drivers may not justify the ROI unless it's paired with performance-linked incentives and training.

b. Investing in Driver Training vs. Customer Satisfaction Benefits of Training:

Can standardize driver behavior, improve ride quality, and reduce complaints.

May enhance Quarterly Ratings, which models show correlate positively with Total Business Value.

Trade-off:

Training costs can be significant (venue, content, lost driving hours).

Not all trained drivers may apply the learning.

Conclusion: A targeted training program (e.g., for low-performing drivers or in specific cities) offers better ROI than blanket training across all drivers.

**Recommendations** a. Specific Strategies from Model Insights Targeted Training Programs

Focus on drivers with <3 Quarterly Rating and low business value.

Prioritize cities with lower average ratings but high ride volumes (indicating high-impact areas).

Improved Recruitment Processes

Use model insights (e.g., Age 25–35, Grade A, high prior rating) to filter candidates.

Recruit from cities with strong driver performance for pilot expansion.

Incentive Schemes

Reward top 10% drivers per city using metrics like Quarterly Rating + Total Business Value.

Introduce early access to rides, bonuses, or priority support for consistently top-rated drivers.

b. Evidence from Analysis City-Based Growth: If City X saw the most improvement in ratings over the year, direct more investments there for recruitment/training.

Demographic Targeting:

E.g., Age group 30–40 might show the highest total business value per ride.

These insights can guide ad campaigns or partnerships (e.g., with driving schools).

**Feedback Loop** a. Periodic Review Process Set up a quarterly model validation process:

Re-evaluate feature importance.

Retrain models using updated data (driver ratings, churn, new ride data).

Use dashboards (e.g., in Power BI or Tableau) for live KPI monitoring.

b. Collecting Ongoing Feedback Driver Feedback Channels:

Anonymous quarterly surveys on job satisfaction, customer interaction issues, app usability.

Customer Feedback:

Post-ride surveys with open text and rating fields.

Use NLP techniques to detect themes (e.g., "late pickup," "rude driver").

Model Adjustment:

Include new features (e.g., real-time location issues, ride cancellations) based on feedback.

**Final Note:** Balancing business cost and customer satisfaction requires a feedback-driven loop — one that learns from both quantitative model results and qualitative human insights.

Would you like this compiled into a formal presentation/report format or summarized in bullet points for slides?

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.

Start coding or generate with AI.