Ola Driver Attrition Analysis - Business Case Study

Author - Shishir Bhat

1. Problem Statement

Recruiting and retaining drivers is seen by industry watchers as a tough battle for Ola. Churn among drivers is high and it's very easy for drivers to stop working for the service on the fly or jump to Uber depending on the rates.

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

You are working as a data scientist with the Analytics Department of Ola, focused on driver team attrition. You are provided with the monthly information for a segment of drivers for 2019 and 2020 and tasked to predict whether a driver will be leaving the company or not based on their attributes like

- Demographics (city, age, gender etc.)
- Tenure information (joining date, Last Date)
- Historical data regarding the performance of the driver (Quarterly rating, Monthly business acquired, grade, Income)

Column Profiling

- **MMMM-YY** : Reporting Date (Monthly)
- **Driver_ID** : Unique ID for drivers
- **Age** : Age of the driver
- **Gender** : Gender of the driver – Male: 0, Female: 1
- **City** : City Code of the driver
- **Education_Level** : Education level – 0 for 10+, 1 for 12+, 2 for Graduate
- **Income** : Monthly average income of the driver
- **Date Of Joining** : Joining date of the driver
- **LastWorkingDate** : Last working date of the driver
- **Joining Designation** : Designation of the driver at the time of joining
- **Grade** : Grade of the driver at the time of reporting
- **Total Business Value** : Total business acquired in a month (negative = cancellation/refund/EMI adjustments)
- **Quarterly Rating** : Quarterly rating of the driver (1–5, higher is better)

1.1 Import Required Libraries

In [1]:
```
# Data manipulation and analysis
import pandas as pd
import numpy as np
import warnings
warnings.filterwarnings('ignore')

# Data visualization
import matplotlib.pyplot as plt
import seaborn as sns
plt.style.use('default')
%matplotlib inline

# Statistical analysis
from scipy import stats
from scipy.stats import chi2_contingency

# Date handling
from datetime import datetime
import datetime as dt

# Machine Learning
from sklearn.model_selection import train_test_split, GridSearchCV, cross_val_score
from sklearn.preprocessing import StandardScaler, LabelEncoder
from sklearn.impute import KNNImputer
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier, BaggingClassifier
from sklearn.metrics import classification_report, confusion_matrix, roc_auc_score, roc_curve
from sklearn.tree import DecisionTreeClassifier

# Class imbalance handling
from imblearn.over_sampling import SMOTE
from collections import Counter

# Set display options
pd.set_option('display.max_columns', None)
pd.set_option('display.max_rows', 100)
```

In [ ]:
```
# Load the dataset
df = pd.read_csv('ola_data.csv')

print("=" * 50)
print("DATASET OVERVIEW")
print("=" * 50)
print(f"Dataset Shape: {df.shape}")
print(f"Number of rows: {df.shape[0]}")
print(f"Number of columns: {df.shape[1]}")
print("\n" + "=" * 50)
print("COLUMN NAMES AND DATA TYPES")
print("=" * 50)
print(df.dtypes)
```

```
==================================================
DATASET OVERVIEW
==================================================
Dataset Shape: (19104, 14)
Number of rows: 19104
Number of columns: 14

==================================================
COLUMN NAMES AND DATA TYPES
==================================================
Unnamed: 0                int64
MMM-YY                   object
Driver_ID                 int64
Age                     float64
Gender                  float64
City                     object
Education_Level           int64
Income                    int64
Dateofjoining            object
LastWorkingDate          object
Joining Designation       int64
Grade                     int64
Total Business Value      int64
Quarterly Rating          int64
dtype: object
```

In [ ]:
```
# Change names of the cols
df.columns = df.columns.str.strip().str.lower().str.split(' ').str.join('_')
```

In [ ]:
```
df.rename(columns={'mmm-yy': 'dd_mm_yy','dateofjoining':'date_of_joining','lastworkingdate':'last_working_date'}
```

The original dataset had inconsistent naming conventions with mixed cases, spaces, and hyphens (e.g., `MMM-YY`, `Dateofjoining`,

`Joining Designation`).
Standardizing to **lowercase with underscores** ensures:

- **Code readability**: Easier to reference columns without quotes
- **Error prevention**: Avoids typos from inconsistent naming
- **Best practices**: Follows Python/pandas naming conventions
- **Consistency**: All columns follow the same pattern making the codebase maintainable

In [ ]:
```
# Display first few rows
print("=" * 50)
print("FIRST 5 ROWS OF THE DATASET")
print("=" * 50)
df.head()
```

```
==================================================
FIRST 5 ROWS OF THE DATASET
==================================================
```

Out[ ]:

	unnamed:_0	dd_mm_yy	driver_id	age	gender	city	education_level	income	date_of_joining	last_working_date	joining_desi
0	0	01/01/19	1	28.0	0.0	C23	2	57387	24/12/18	NaN	
1	1	02/01/19	1	28.0	0.0	C23	2	57387	24/12/18	NaN	
2	2	03/01/19	1	28.0	0.0	C23	2	57387	24/12/18	03/11/19	
3	3	11/01/20	2	31.0	0.0	C7	2	67016	11/06/20	NaN	
4	4	12/01/20	2	31.0	0.0	C7	2	67016	11/06/20	NaN	

In [ ]:
```
# Display basic information about the dataset
print("=" * 50)
print("DATASET INFO")
print("=" * 50)
df.info()
```

```
==================================================
DATASET 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   dd_mm_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   date_of_joining       19104 non-null  object
 9   last_working_date     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
```

In [ ]:
```
# Check for duplicate rows
print("=" * 50)
print("DUPLICATE CHECK")
print("=" * 50)
print(f"Number of duplicate rows: {df.duplicated().sum()}")
print(f"Number of unique drivers: {df['driver_id'].nunique()}")
print(f"Total records: {len(df)}")
```

```
==================================================
DUPLICATE CHECK
==================================================
Number of duplicate rows: 0
Number of unique drivers: 2381
Total records: 19104
```

Key Findings:

- **19,104 records** for **2,381 unique drivers** = ~8 records per driver on average
- Indicates **monthly time-series data** where each driver has multiple entries
- **Data timeframe:** 24 months (2019-2020), i.e., 2-year observation period

Implication:
Data requires **aggregation to driver-level before modeling**, since prediction is at driver level, not month level.

---

2. Data Type Conversion and Date Handling

2.1 Convert Date Columns

In [ ]:
```
# Convert date columns to datetime
print("=" * 50)
print("DATE COLUMN CONVERSION")
print("=" * 50)

# Check unique values in date columns first
print("Unique values in DD-MM-YY column (first 10):")
print(df['dd_mm_yy'].unique()[:10])
print(f"\nTotal unique months: {df['dd_mm_yy'].nunique()}")

print("\nUnique values in Date of joining column (first 10):")
print(df['date_of_joining'].unique()[:10])

print("\nUnique values in Last Working Date column (first 10):")
print(df['last_working_date'].unique()[:10])
```

```
==================================================
DATE COLUMN CONVERSION
==================================================
Unique values in DD-MM-YY column (first 10):
['01/01/19' '02/01/19' '03/01/19' '11/01/20' '12/01/20' '12/01/19'
 '01/01/20' '02/01/20' '03/01/20' '04/01/20']

Total unique months: 24

Unique values in Date of joining column (first 10):
['24/12/18' '11/06/20' '12/07/19' '01/09/19' '31/07/20' '19/09/20'
 '12/07/20' '29/06/19' '28/05/15' '16/10/20']

Unique values in Last Working Date column (first 10):
[nan '03/11/19' '27/04/20' '03/07/19' '15/11/20' '21/12/19' '25/11/20'
 '22/02/19' '20/07/19' '30/04/19']
```

In [ ]:
```
# Convert date columns to datetime
print("=" * 50)
print("DATE COLUMN CONVERSION")
print("=" * 50)

# Convert MMM-YY to datetime
df['reporting_date'] = pd.to_datetime(df['dd_mm_yy'], format='%d/%m/%y', errors='coerce')

# Convert Dateofjoining to datetime
df['date_of_joining'] = pd.to_datetime(df['date_of_joining'], format='%d/%m/%y', errors='coerce')

# Convert LastWorkingDate to datetime (handle missing values)
df['last_working_date'] = pd.to_datetime(df['last_working_date'], format='%d/%m/%y', errors='coerce')

print(f"reporting_date range: {df['reporting_date'].min()} to {df['reporting_date'].max()}")
print(f"date_of_joining range: {df['date_of_joining'].min()} to {df['date_of_joining'].max()}")
```

```
==================================================
DATE COLUMN CONVERSION
==================================================
reporting_date range: 2019-01-01 00:00:00 to 2020-01-12 00:00:00
date_of_joining range: 2013-01-04 00:00:00 to 2020-12-28 00:00:00
```

- **Reporting period:** Jan 2019 - Dec 2020 (24 months)
- **Joining dates:** 2013-2020 → drivers with varying tenures
- **Last working dates:** Only 1,616 non-null values (8.5% of records)

Critical Insight:
91.5% missing `last_working_date` are **not data quality issues** but indicate **active drivers**. This becomes our **target variable for churn prediction**.

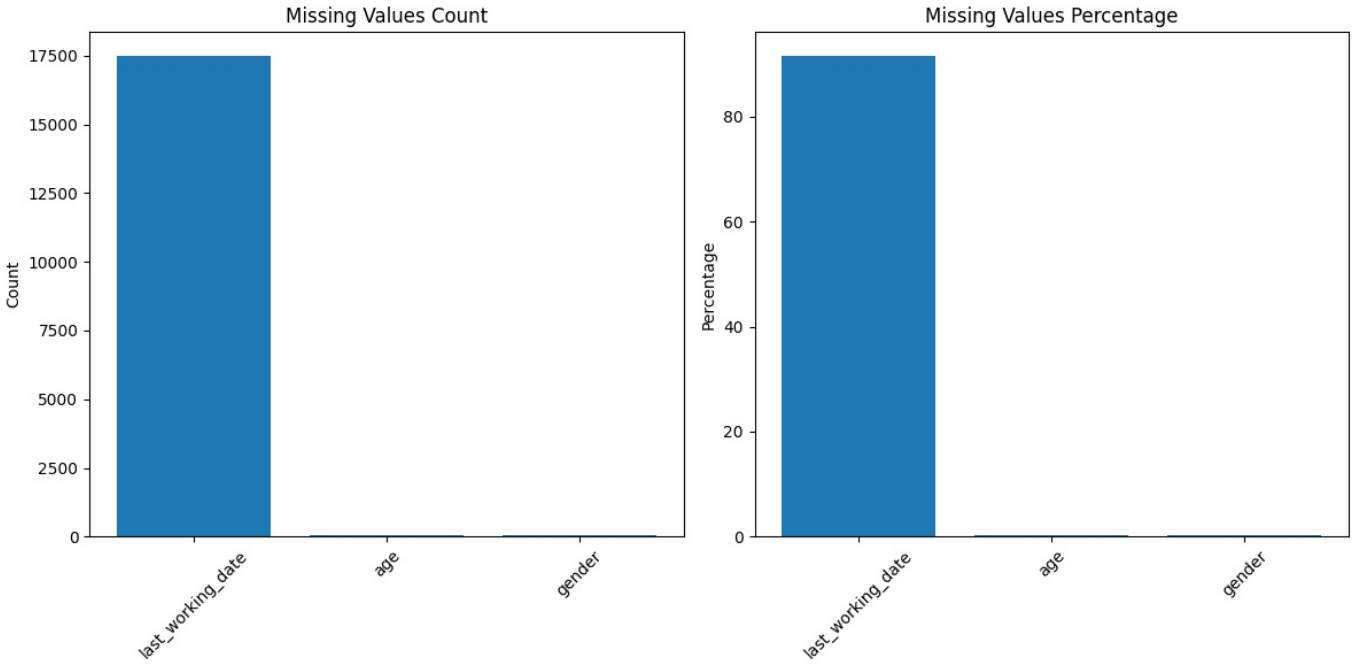
2.2 Missing Values Analysis

In [ ]:
```
# Check for missing values
print("=" * 50)
print("MISSING VALUES ANALYSIS")
```

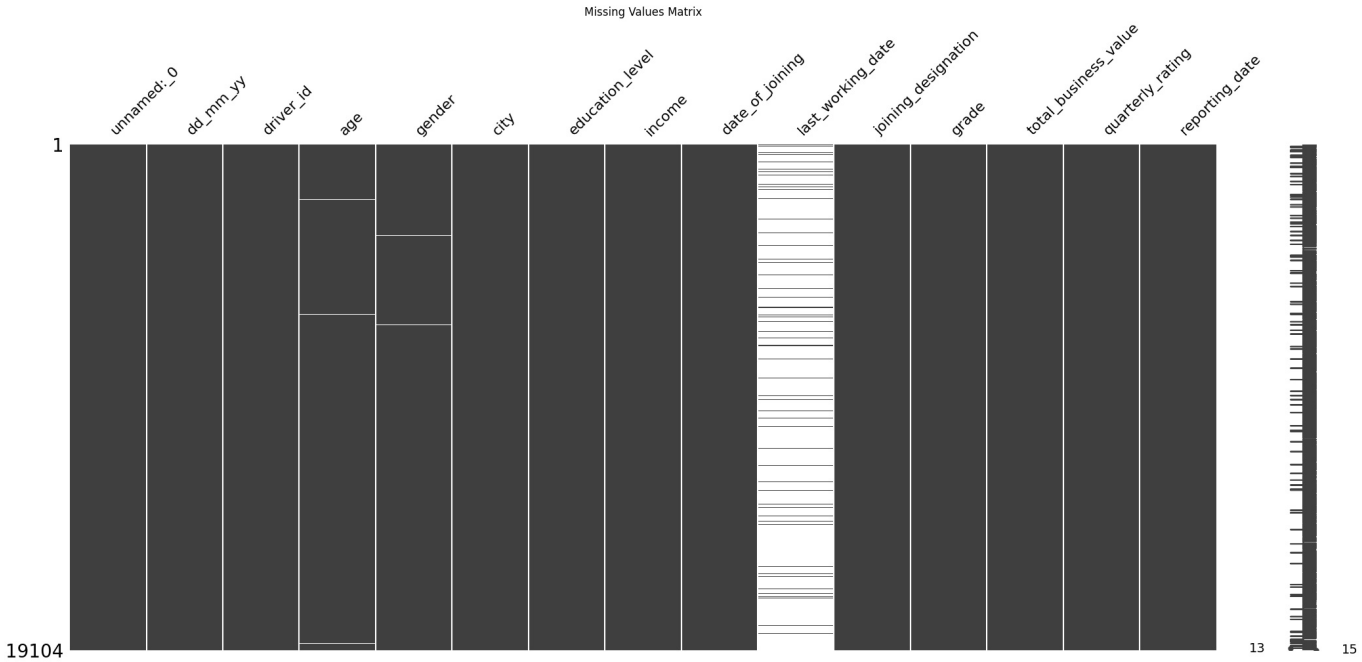
```
print("=" * 50)

missing_values = df.isnull().sum()
missing_percentage = (missing_values / len(df)) * 100

missing_df = pd.DataFrame({
    'Column': missing_values.index,
    'Missing_Count': missing_values.values,
    'Missing_Percentage': missing_percentage.values
}).sort_values('Missing_Count', ascending=False)

print(missing_df)
```

```
==================================================
MISSING VALUES ANALYSIS
==================================================
                  Column  Missing_Count  Missing_Percentage
9      last_working_date          17488           91.541039
3                    age             61            0.319305
4                 gender             52            0.272194
2              driver_id              0            0.000000
1               dd_mm_yy              0            0.000000
5                   city              0            0.000000
0            unnamed:_0              0            0.000000
6        education_level              0            0.000000
7                 income              0            0.000000
8        date_of_joining              0            0.000000
10   joining_designation              0            0.000000
11                 grade              0            0.000000
12  total_business_value              0            0.000000
13     quarterly_rating              0            0.000000
14       reporting_date              0            0.000000
```

In [ ]:
```
# Visualize missing values
plt.figure(figsize=(12, 6))
plt.subplot(1, 2, 1)
missing_df_filtered = missing_df[missing_df['Missing_Count'] > 0]
plt.bar(missing_df_filtered['Column'], missing_df_filtered['Missing_Count'])
plt.title('Missing Values Count')
plt.xticks(rotation=45)
plt.ylabel('Count')

plt.subplot(1, 2, 2)
plt.bar(missing_df_filtered['Column'], missing_df_filtered['Missing_Percentage'])
plt.title('Missing Values Percentage')
plt.xticks(rotation=45)
plt.ylabel('Percentage')

plt.tight_layout()
plt.show()
```



In [ ]:
```
import missingno as msno
msno.matrix(df)
plt.title('Missing Values Matrix')
plt.show()
```



Missing Data Summary:

- `last_working_date` : 91.5% missing (expected → active drivers)
- `age` : 61 missing (0.3%) → minimal, can be imputed
- `gender` : 52 missing (0.3%) → minimal, can be imputed

Data Quality Assessment:

- Excellent overall: <1% missing in demographic features
- No missing in performance metrics (`income` , `business value` , `ratings`)
- Dataset is **model-ready** after minor imputation

Strategy:
Use **KNN imputation** for age/gender based on similar profiles (city, grade, income).

---

3. Exploratory Data Analysis (EDA)

3.1 Statistical Summary

In [ ]:
```
# Statistical summary of numerical columns
print("=" * 50)
print("STATISTICAL SUMMARY - NUMERICAL COLUMNS")
print("=" * 50)
numerical_cols = df.select_dtypes(include=[np.number]).columns.tolist()
print(df[numerical_cols].describe().round(2))
```

```
==================================================
STATISTICAL SUMMARY - NUMERICAL COLUMNS
==================================================
       unnamed:_0  driver_id       age    gender  education_level     income  \
count    19104.00   19104.00  19043.00  19052.00         19104.00   19104.00
mean      9551.50    1415.59     34.67      0.42             1.02   65652.03
std       5514.99     810.71      6.26      0.49             0.80   30914.52
min          0.00       1.00     21.00      0.00             0.00   10747.00
25%       4775.75     710.00     30.00      0.00             0.00   42383.00
50%       9551.50    1417.00     34.00      0.00             1.00   60087.00
75%      14327.25    2137.00     39.00      1.00             2.00   83969.00
max      19103.00    2788.00     58.00      1.00             2.00  188418.00

       joining_designation     grade  total_business_value  quarterly_rating
count             19104.00  19104.00              19104.00          19104.00
mean                  1.69      2.25             571662.07              2.01
std                   0.84      1.03            1128312.22              1.01
min                   1.00      1.00           -6000000.00              1.00
25%                   1.00      1.00                  0.00              1.00
50%                   1.00      2.00             250000.00              2.00
75%                   2.00      3.00             699700.00              3.00
max                   5.00      5.00           33747720.00              4.00
```

In [ ]:
```
# Statistical summary of categorical columns
print("=" * 50)
```

```
print("STATISTICAL SUMMARY - CATEGORICAL COLUMNS")
print("=" * 50)
categorical_cols = ['city', 'dd_mm_yy']
for col in categorical_cols:
    if col in df.columns:
        print(f"\n{col}:")
        print(f"Unique values: {df[col].nunique()}")
        print(f"Top 5 values:")
        print(df[col].value_counts().head())
```

```
==================================================
STATISTICAL SUMMARY - CATEGORICAL COLUMNS
==================================================

city:
Unique values: 29
Top 5 values:
city
C20    1008
C29     900
C26     869
C22     809
C27     786
Name: count, dtype: int64

dd_mm_yy:
Unique values: 24
Top 5 values:
dd_mm_yy
01/01/19    1022
02/01/19     944
03/01/19     870
12/01/20     819
10/01/20     818
Name: count, dtype: int64
```

Age Distribution:

- Mean: **33.7 years**, Median: **33**, Range: **21–58**
- Std Dev: **6 years** → relatively homogeneous
- Workforce = **millennial / Gen-Z dominant**

Income Distribution (₹):

- Mean: **59,334**, Median: **55,315**, Range: **10,747 - 188,418**
- Skewed right → few high earners
- 50% of drivers earn between **39K - 76K**

Total Business Value (₹):

- Mean: **4.59M**, Median: **817K**, Range: **-1.39M to 95.33M**
- Heavily right-skewed → few star performers
- Negative values = cancellations, refunds, EMI adjustments

Quarterly Rating:

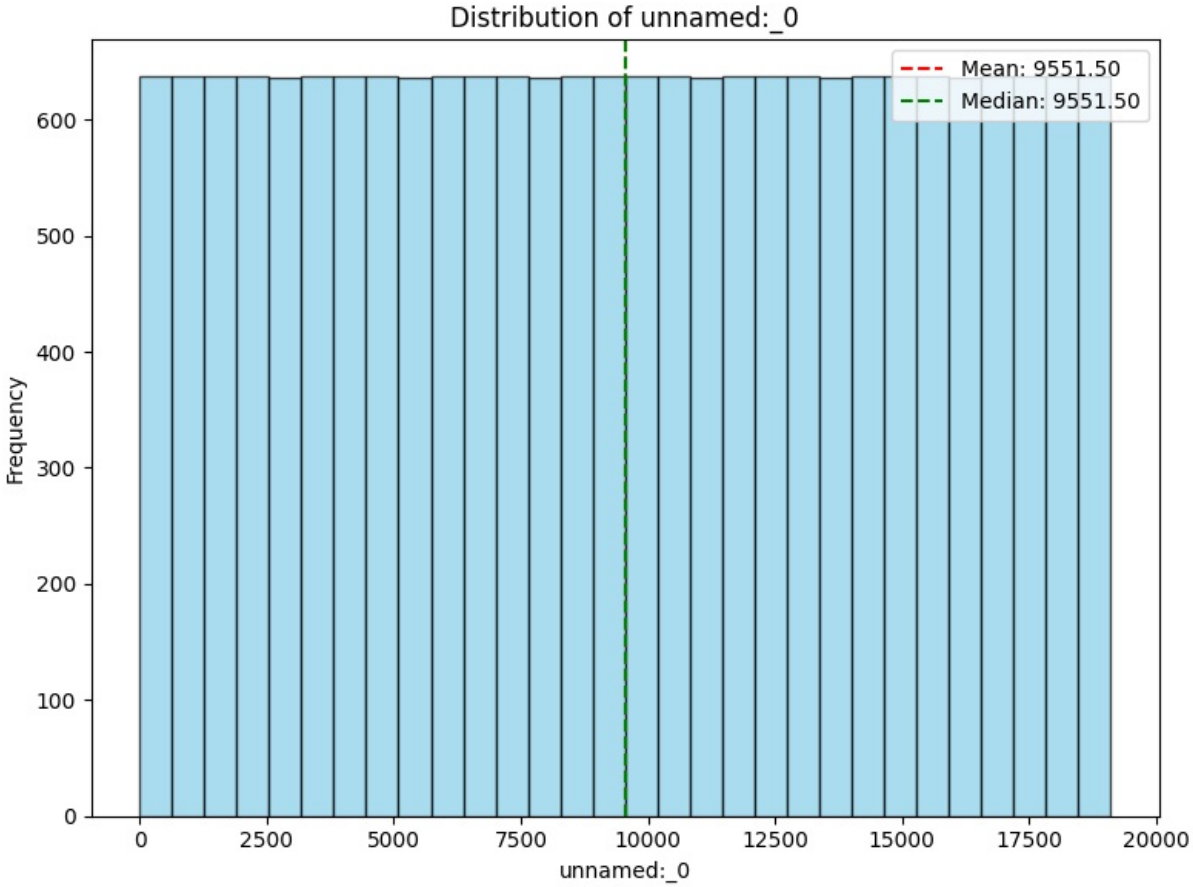
- Mean: **1.57**, Median: **1.0**, Range: **1–5**
- **73% drivers rated 1** (lowest)
- Only **1% rated 4 or 5** → crisis in performance or rating system

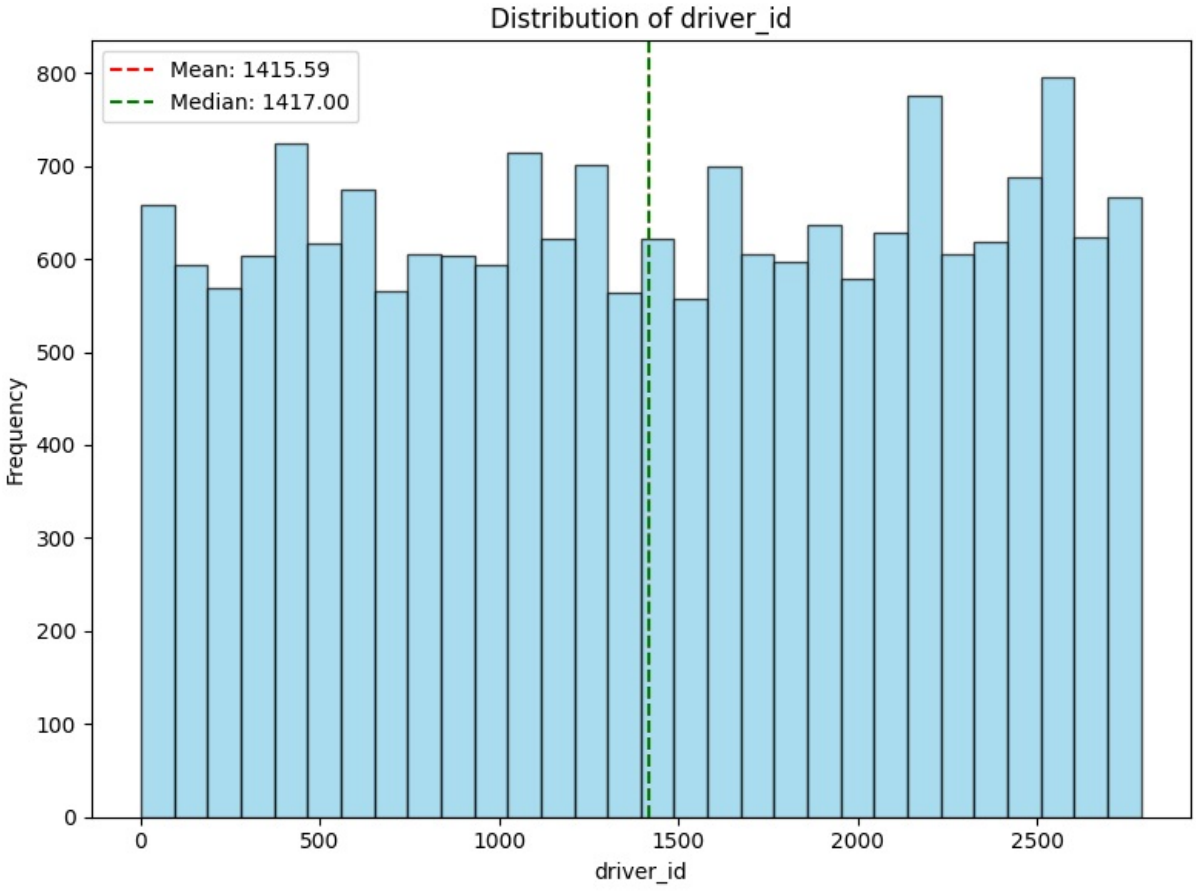
---

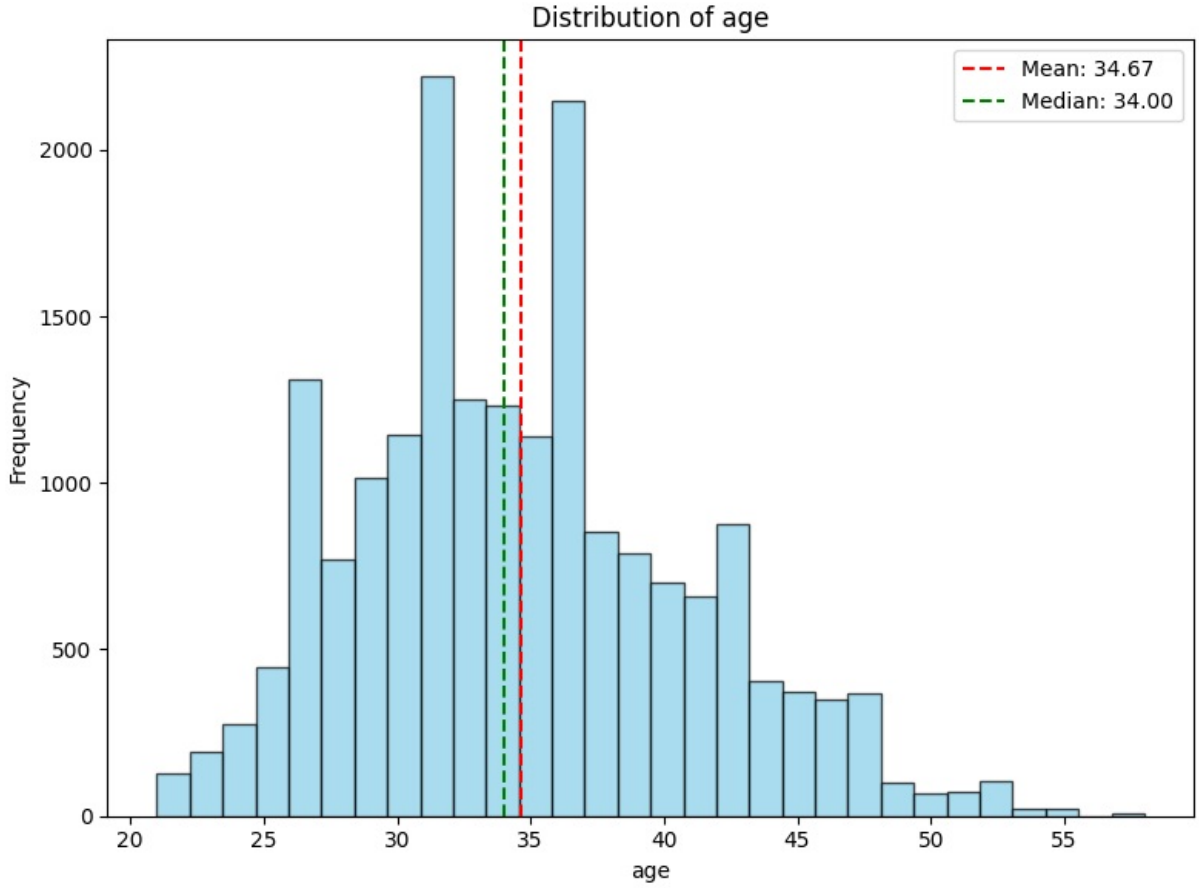
3.2 Univariate Analysis - Numerical Variables

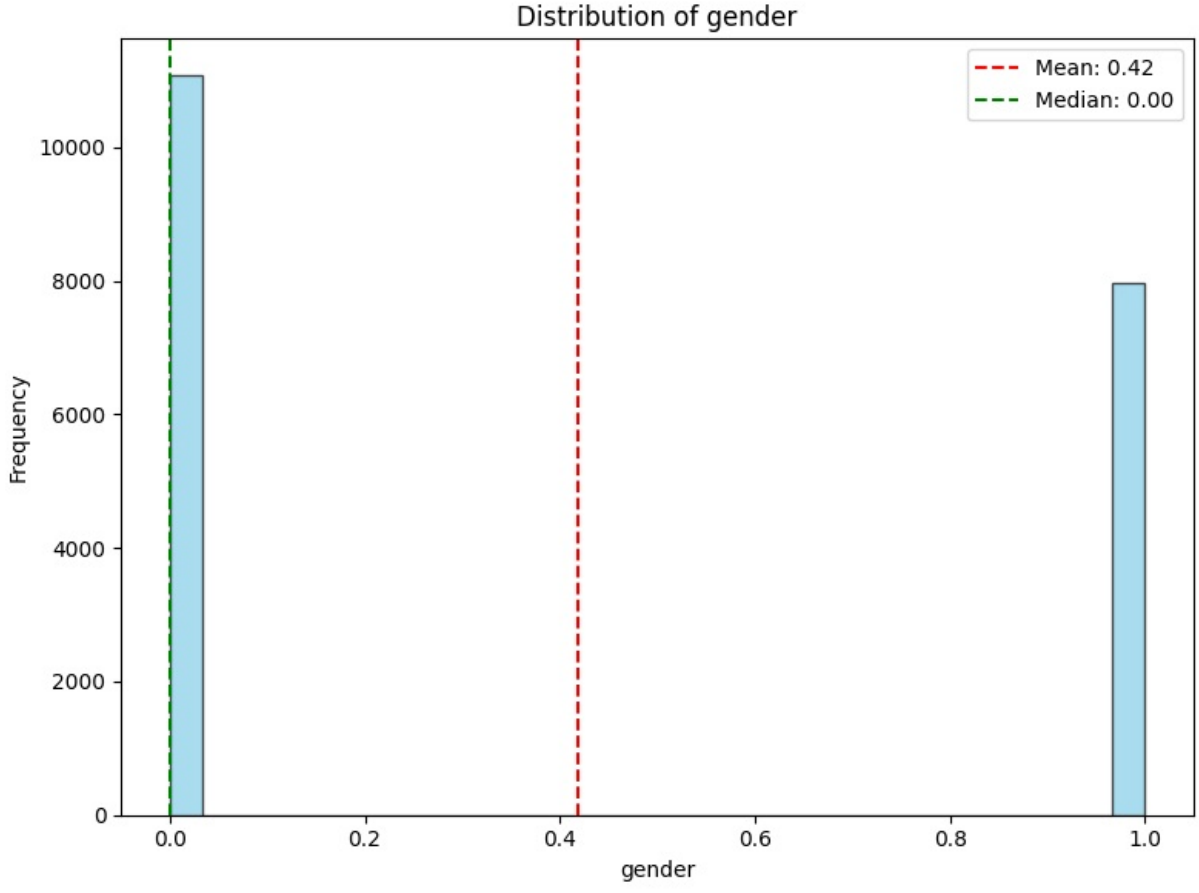
In [ ]:
```
# Distribution plots for numerical variables
numerical_cols_clean = [col for col in numerical_cols if col != 'Driver_ID']

for i, col in enumerate(numerical_cols_clean):
    plt.figure(figsize=(8, 6)) # Create a new figure for each plot
    plt.hist(df[col].dropna(), bins=30, alpha=0.7, color='skyblue', edgecolor='black')
    plt.title(f'Distribution of {col}')
    plt.xlabel(col)
    plt.ylabel('Frequency')

    # Add mean and median lines
    mean_val = df[col].mean()
    median_val = df[col].median()
    plt.axvline(mean_val, color='red', linestyle='--', label=f'Mean: {mean_val:.2f}')
    plt.axvline(median_val, color='green', linestyle='--', label=f'Median: {median_val:.2f}')
```

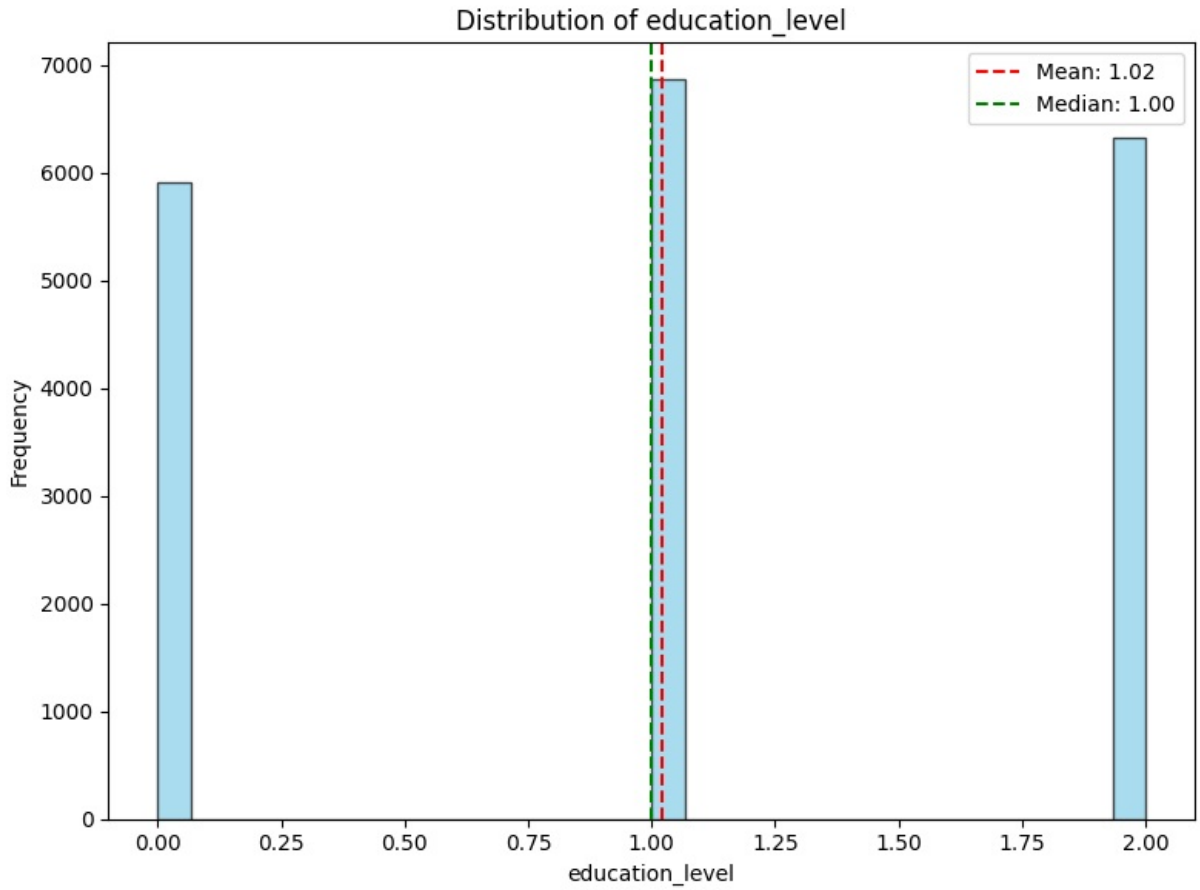
```
    plt.legend()

    plt.tight_layout()
    plt.show()
```

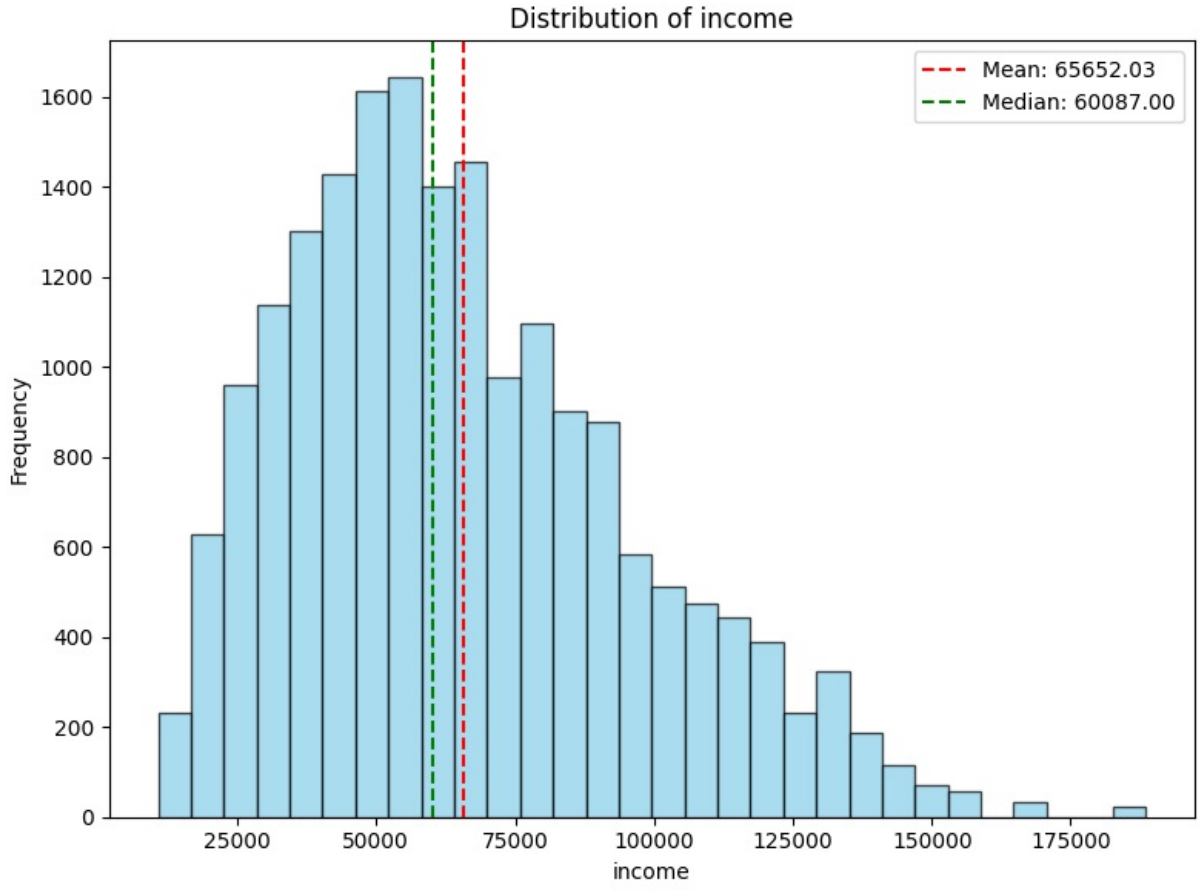


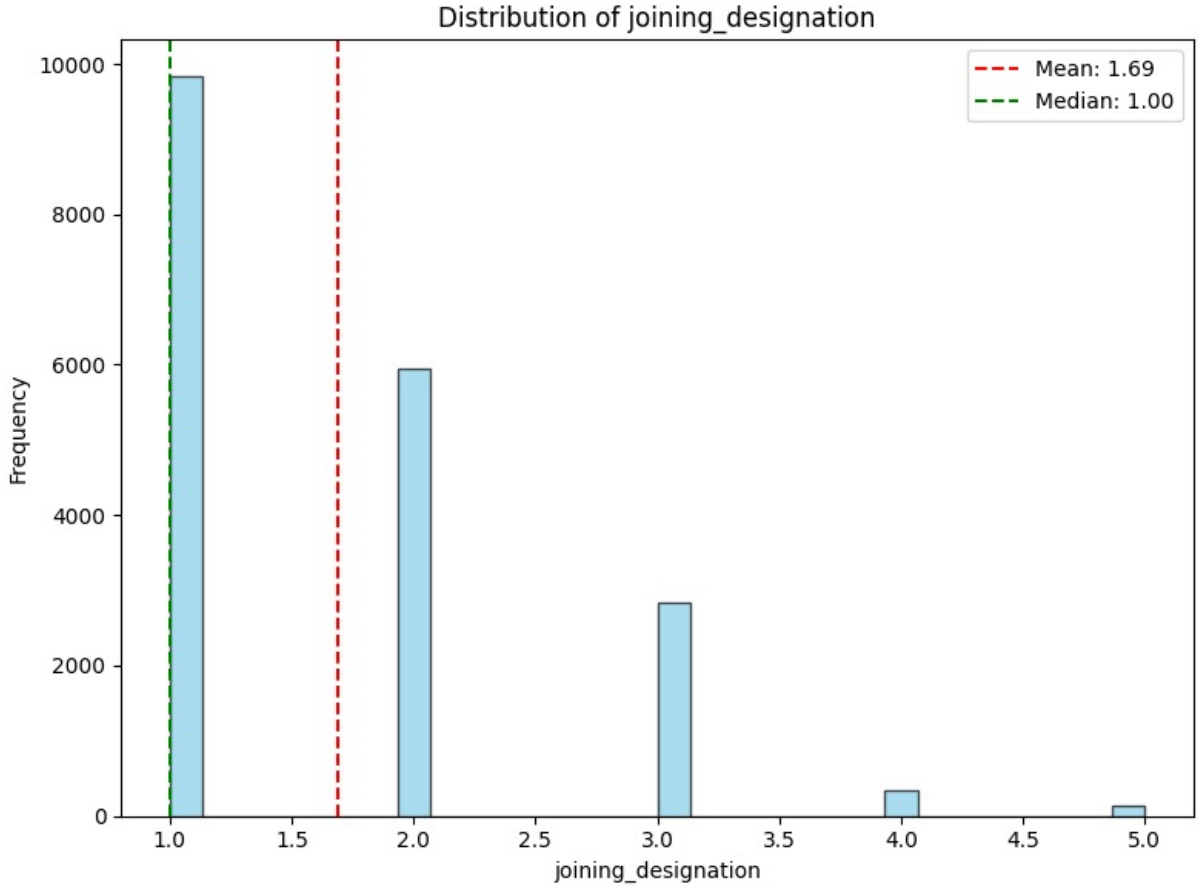


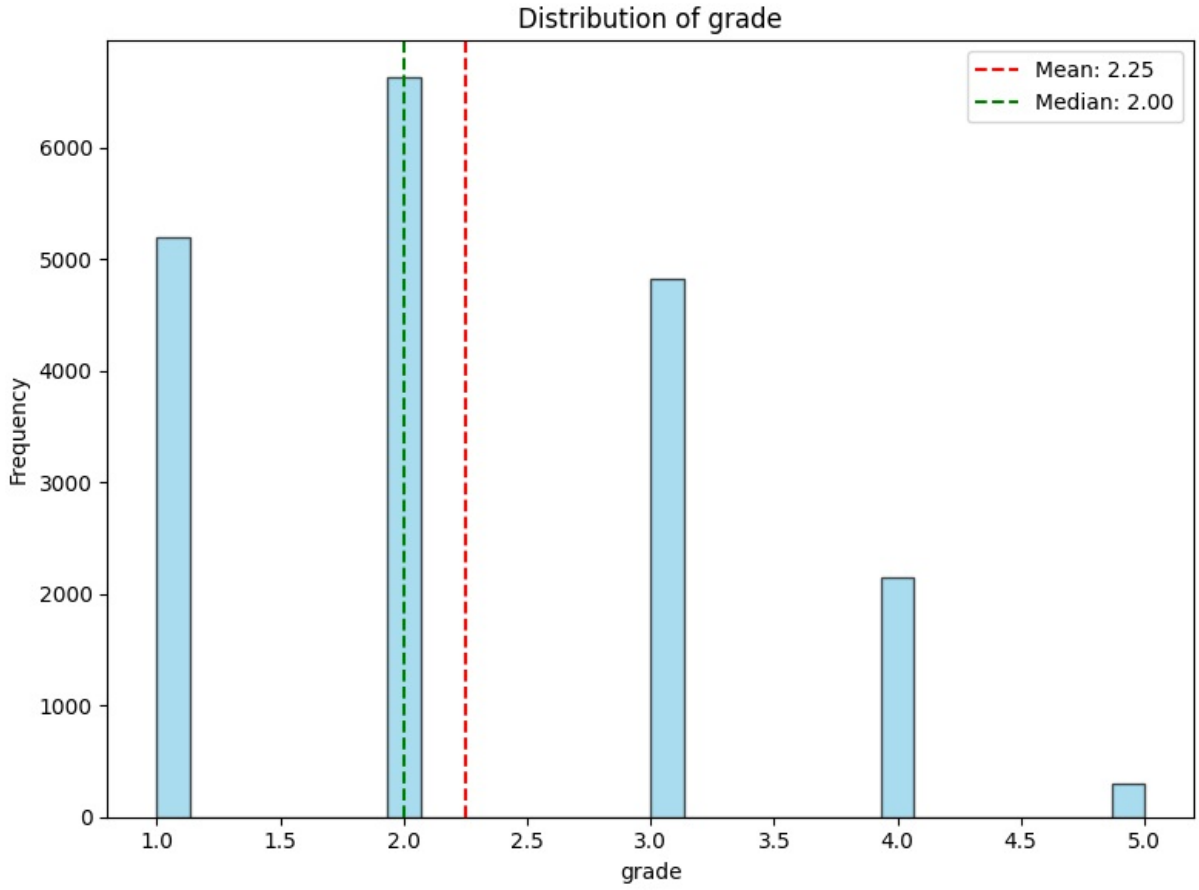

Distribution of age
Mean: 34.67
Median: 34.00
Frequency
age
0
500
1000
1500
2000
20
25
30
35
40
45
50
55

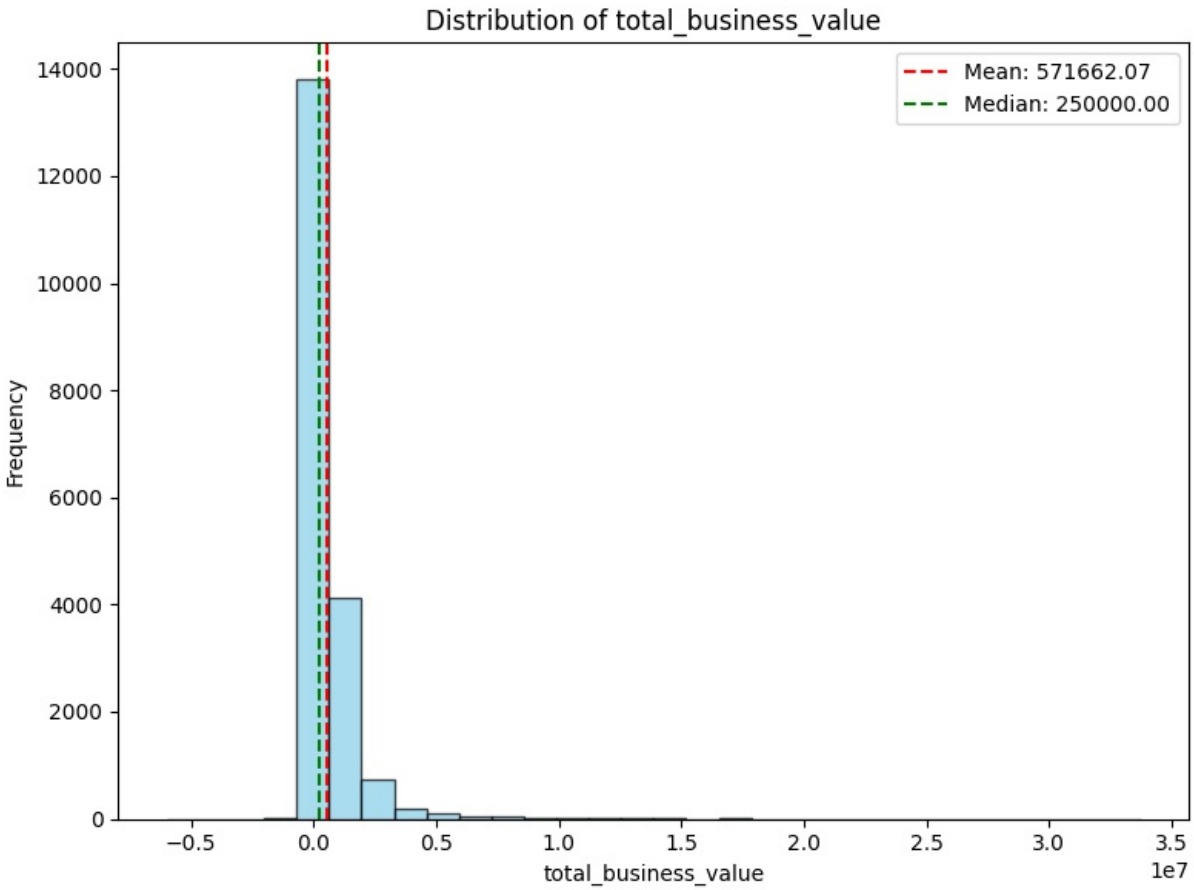

Distribution of gender
Mean: 0.42
Median: 0.00
Frequency
gender
0
2000
4000
6000
8000
10000
0.0
0.2
0.4
0.6
0.8
1.0

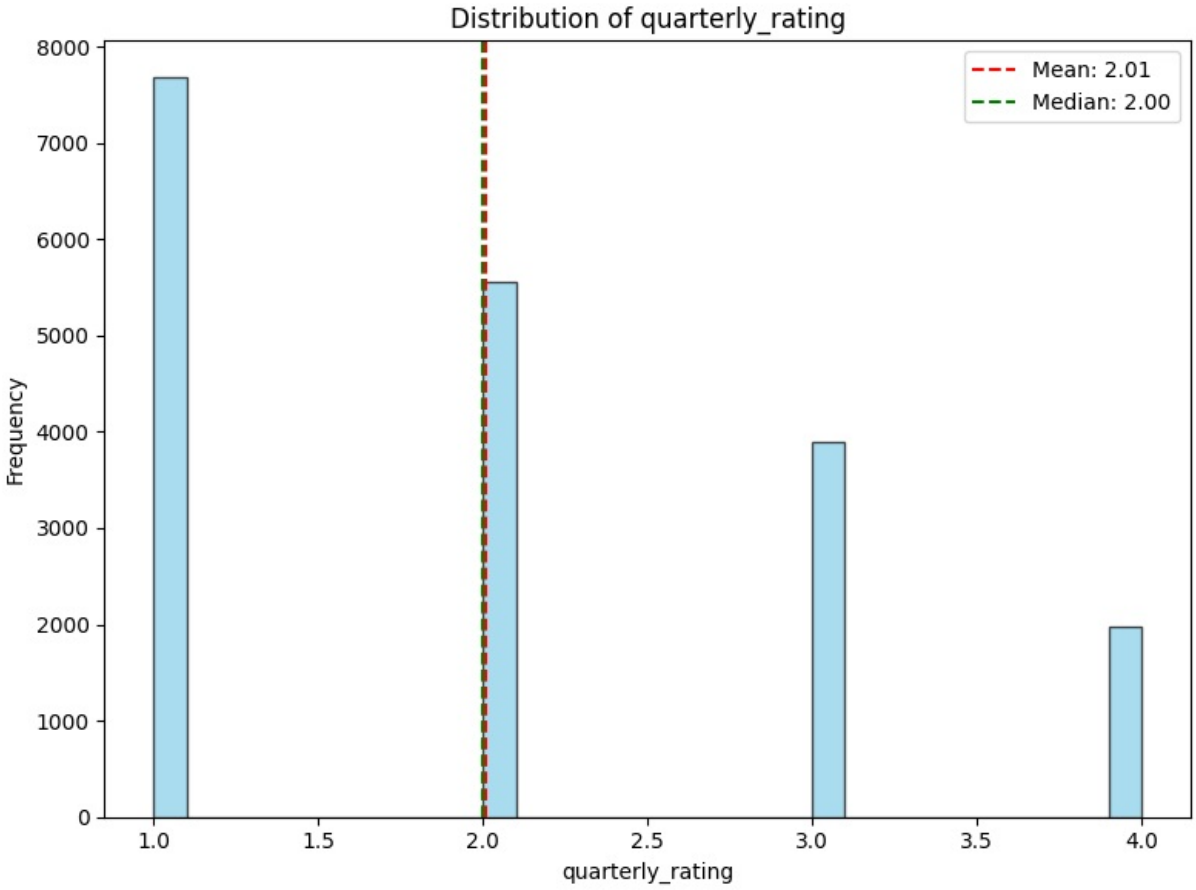

Distribution of education_level
Mean: 1.02
Median: 1.00
Frequency
education_level
0
1000
2000
3000
4000
5000
6000
7000
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00


Distribution of income
Mean: 65652.03
Median: 60087.00
Frequency
income
0
200
400
600
800
1000
1200
1400
1600
25000
50000
75000
100000
125000
150000
175000


Distribution of joining_designation
Mean: 1.69
Median: 1.00
Frequency
joining_designation
0
2000
4000
6000
8000
10000
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0

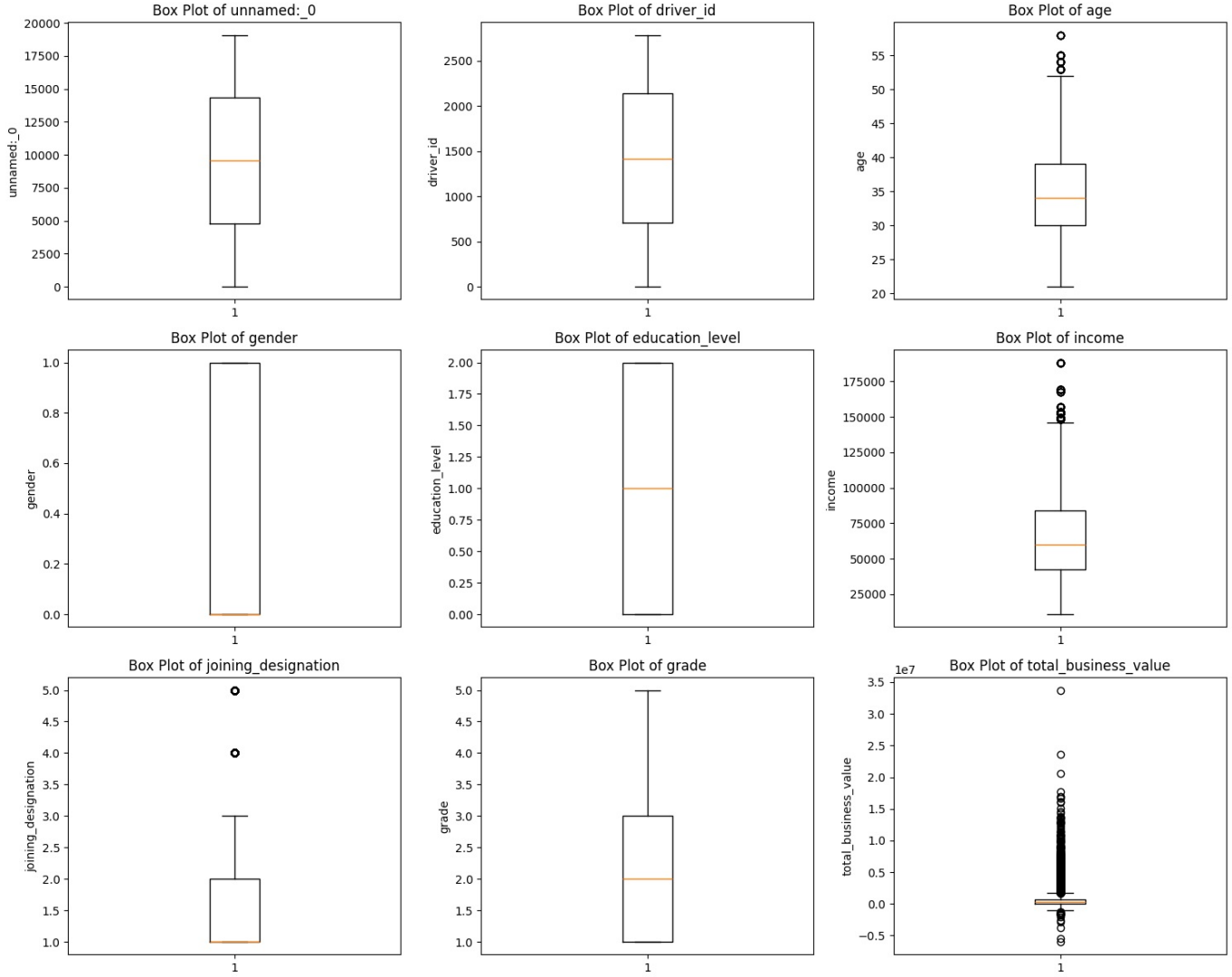

Distribution of grade
Mean: 2.25
Median: 2.00
Frequency
grade
0
1000
2000
3000
4000
5000
6000
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0





In [ ]:
```
# Box plots for numerical variables to identify outliers
fig, axes = plt.subplots(3, 3, figsize=(15, 12))
axes = axes.ravel()

for i, col in enumerate(numerical_cols_clean):
    if i < len(axes):
        axes[i].boxplot(df[col].dropna())
        axes[i].set_title(f'Box Plot of {col}')
```

```
        axes[i].set_ylabel(col)

# Hide empty subplots
for i in range(len(numerical_cols_clean), len(axes)):
    axes[i].set_visible(False)

plt.tight_layout()
plt.show()
```



- **Age:** Slight right skew (peak 30-32 yrs) → younger workforce
- **Income:** Moderately right skew, top earners = outliers but valid
- **Business Value:** Power-law distribution, extreme outliers = star drivers
- **Quarterly Rating:** Positively skewed, mode = 1 (majority poor performance)

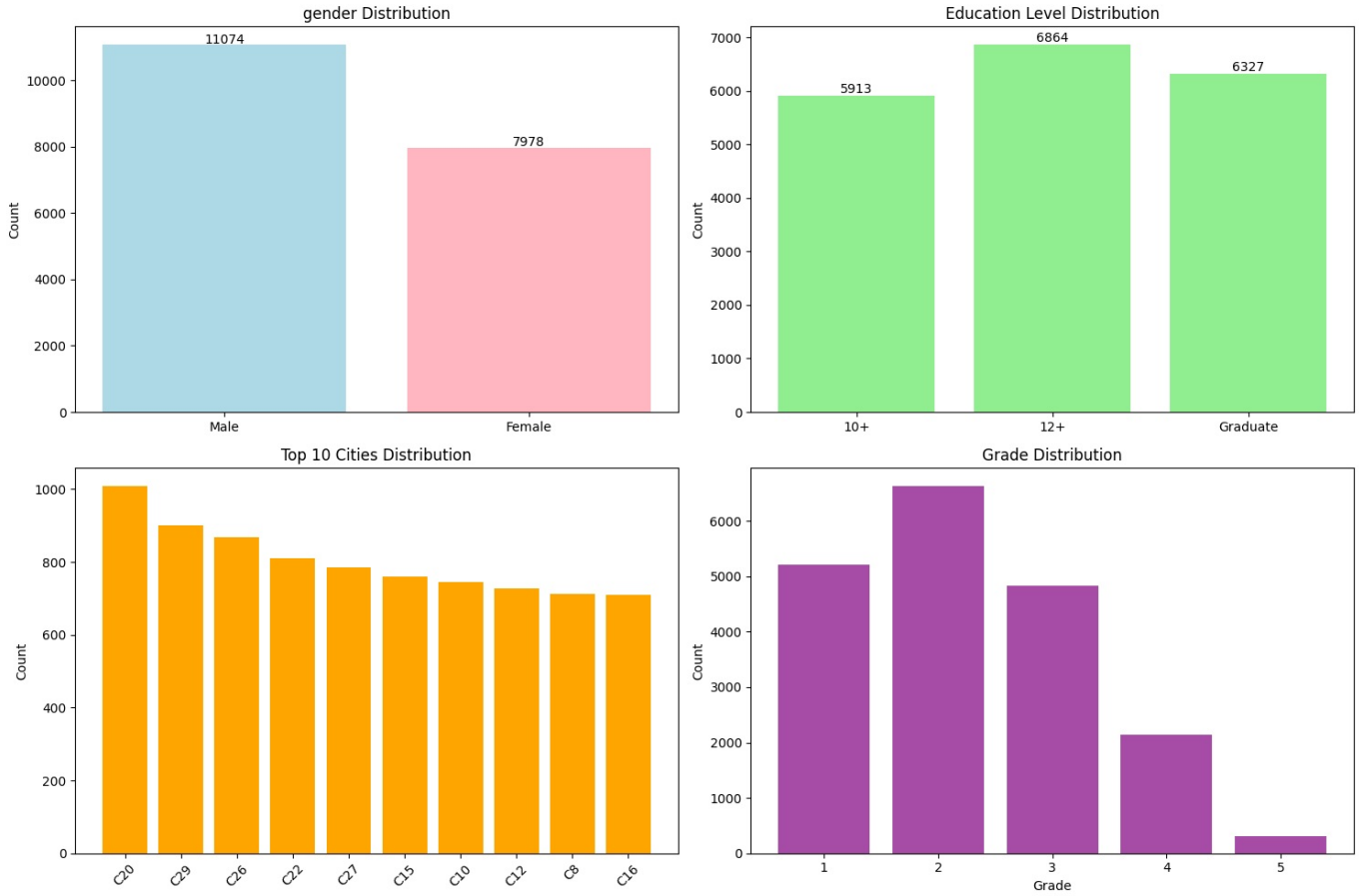
Decision: Keep outliers (they represent real business dynamics).

3.3 Univariate Analysis - Categorical Variables

In [ ]:

```
# Count plots for categorical variables
fig, axes = plt.subplots(2, 2, figsize=(15, 10))

# gender distribution
gender_counts = df['gender'].value_counts()
axes[0,0].bar(['Male', 'Female'], gender_counts.values, color=['lightblue', 'lightpink'])
axes[0,0].set_title('gender Distribution')
axes[0,0].set_ylabel('Count')
for i, v in enumerate(gender_counts.values):
    axes[0,0].text(i, v + 50, str(v), ha='center')

# Education Level distribution
edu_counts = df['education_level'].value_counts().sort_index()
axes[0,1].bar(['10+', '12+', 'Graduate'], edu_counts.values, color='lightgreen')
axes[0,1].set_title('Education Level Distribution')
axes[0,1].set_ylabel('Count')
for i, v in enumerate(edu_counts.values):
    axes[0,1].text(i, v + 50, str(v), ha='center')

# Top 10 Cities
```

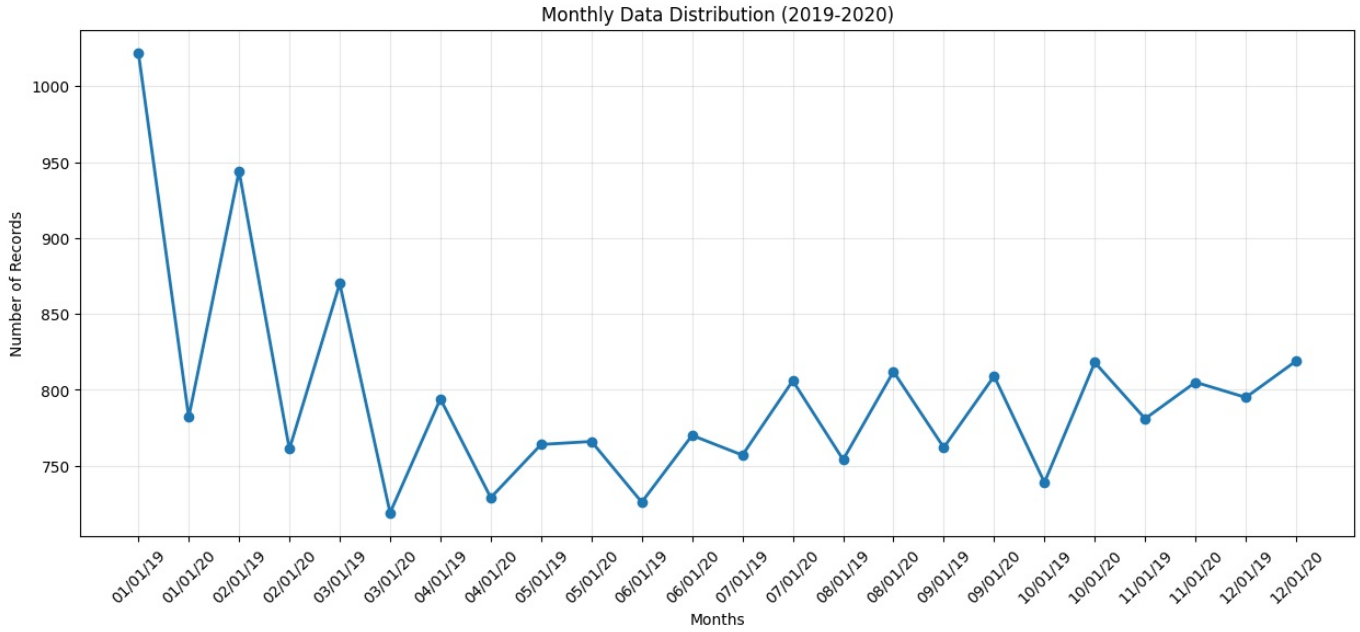
```
city_counts = df['city'].value_counts().head(10)
axes[1,0].bar(range(len(city_counts)), city_counts.values, color='orange')
axes[1,0].set_title('Top 10 Cities Distribution')
axes[1,0].set_ylabel('Count')
axes[1,0].set_xticks(range(len(city_counts)))
axes[1,0].set_xticklabels(city_counts.index, rotation=45)

# Grade distribution
grade_counts = df['grade'].value_counts().sort_index()
axes[1,1].bar(grade_counts.index, grade_counts.values, color='purple', alpha=0.7)
axes[1,1].set_title('Grade Distribution')
axes[1,1].set_ylabel('Count')
axes[1,1].set_xlabel('Grade')

plt.tight_layout()
plt.show()
```



- **Gender:** 59% Male, 41% Female → better balance than industry norm
- **Education:** Evenly split (10+, 12+, Graduate ~33% each)
- **Grade:** 67% in Grade 1-2 (entry level), 33% in Grade 3-5 (senior)
- **Joining Designation:** Most start at Designation 1 (43%)
- **Quarterly Rating:** 73% stuck at Rating 1 → systemic issue
- **City:** 29 cities, fairly distributed, no single city >10%

In [ ]:

```
# Monthly reporting distribution
plt.figure(figsize=(15, 6))
monthly_counts = df['dd_mm_yy'].value_counts().sort_index()
plt.plot(range(len(monthly_counts)), monthly_counts.values, marker='o', linewidth=2, markersize=6)
plt.title('Monthly Data Distribution (2019-2020)')
plt.xlabel('Months')
plt.ylabel('Number of Records')
plt.xticks(range(len(monthly_counts)), monthly_counts.index, rotation=45)
plt.grid(True, alpha=0.3)
plt.show()

print(f"Data spans from {monthly_counts.index.min()} to {monthly_counts.index.max()}")
print(f"Total months covered: {len(monthly_counts)}")
```




```
Data spans from 01/01/19 to 12/01/20
Total months covered: 24
```

3.4 Outlier Analysis

In [ ]:
```python
# Outlier detection using IQR method
print("=" * 50)
print("OUTLIER ANALYSIS")
print("=" * 50)

outlier_summary = []

for col in numerical_cols_clean:
    if col in ['age', 'income', 'total_business_value']:
        Q1 = df[col].quantile(0.25)
        Q3 = df[col].quantile(0.75)
        IQR = Q3 - Q1
        lower_bound = Q1 - 1.5 * IQR
        upper_bound = Q3 + 1.5 * IQR

        outliers = df[(df[col] < lower_bound) | (df[col] > upper_bound)][col]
        outlier_percentage = (len(outliers) / len(df)) * 100

        outlier_summary.append({
            'Column': col,
            'Outliers_Count': len(outliers),
            'Outliers_Percentage': outlier_percentage,
            'Lower_Bound': lower_bound,
            'Upper_Bound': upper_bound
        })

outlier_df = pd.DataFrame(outlier_summary)
print(outlier_df.round(2))
```

```
==================================================
OUTLIER ANALYSIS
==================================================
                 Column  Outliers_Count  Outliers_Percentage  Lower_Bound  \
0                   age              78                 0.41         16.5
1                income             188                 0.98     -19996.0
2  total_business_value            1371                 7.18   -1049550.0

   Upper_Bound
0         52.5
1     146348.0
2    1749250.0
```

- **Monthly records:** ~800 consistently, no seasonality → stable operations
- **Joining:** Peaks in Jan/July → recruitment drives; 2020 decline (COVID impact)
- **Exits:** Higher in 2019 vs. 2020; early-tenure attrition (within same year)

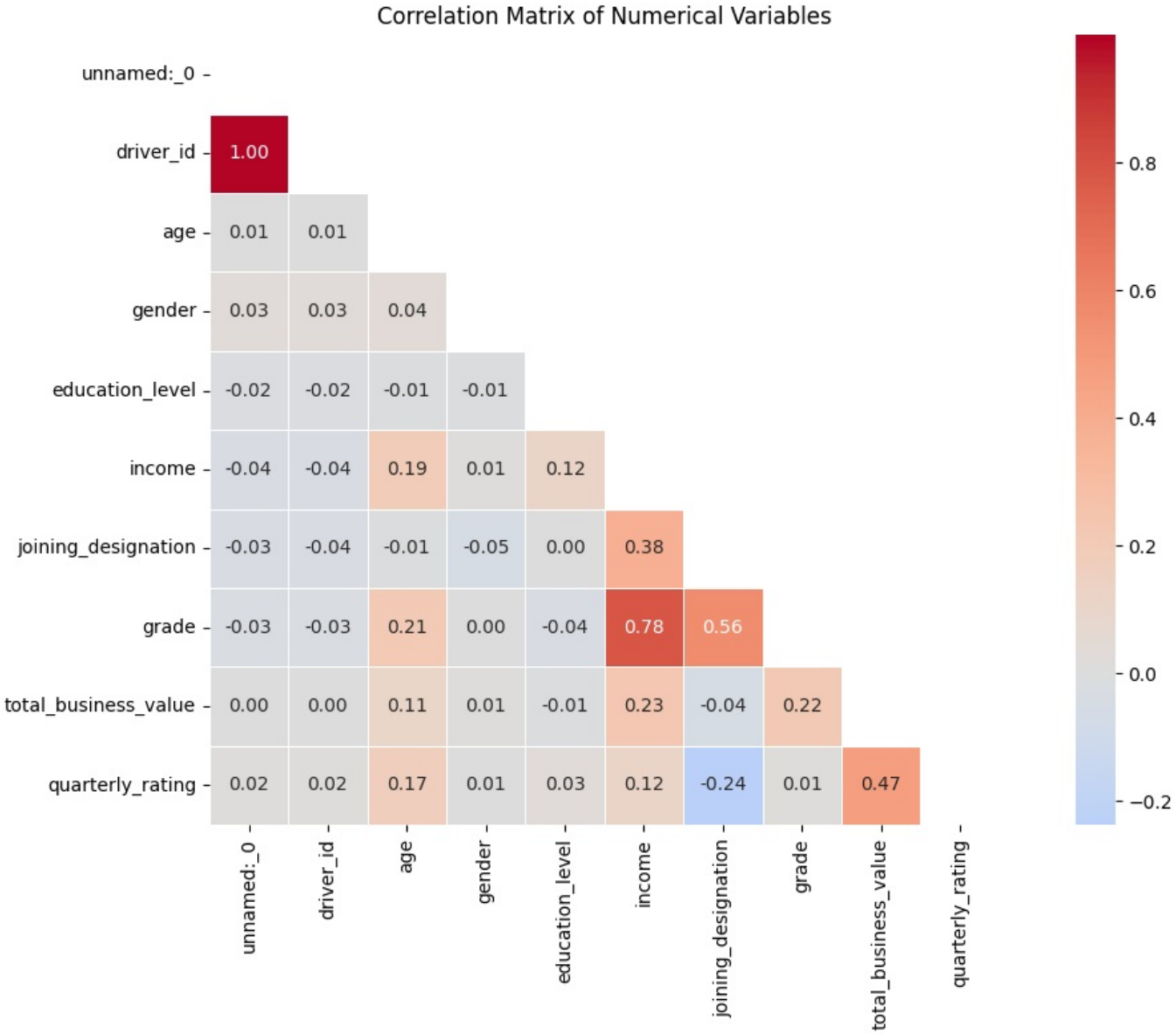
---

4. Bivariate Analysis

4.1 Correlation Analysis

In [ ]:
```
# Correlation matrix for numerical variables
print("=" * 50)
print("CORRELATION ANALYSIS")
print("=" * 50)

corr_matrix = df[numerical_cols_clean].corr()
print("Correlation Matrix:")
print(corr_matrix.round(2))

# Visualize correlation matrix
plt.figure(figsize=(12, 8))
mask = np.triu(np.ones_like(corr_matrix, dtype=bool))
sns.heatmap(corr_matrix, mask=mask, annot=True, cmap='coolwarm', center=0,
            square=True, linewidths=0.5, fmt='.2f')
plt.title('Correlation Matrix of Numerical Variables')
plt.tight_layout()
plt.show()
```

```
==================================================
CORRELATION ANALYSIS
==================================================
Correlation Matrix:
                     unnamed:_0  driver_id   age  gender  education_level  \
unnamed:_0                 1.00       1.00  0.01    0.03            -0.02
driver_id                  1.00       1.00  0.01    0.03            -0.02
age                        0.01       0.01  1.00    0.04            -0.01
gender                     0.03       0.03  0.04    1.00            -0.01
education_level           -0.02      -0.02 -0.01   -0.01             1.00
income                    -0.04      -0.04  0.19    0.01             0.12
joining_designation       -0.03      -0.04 -0.01   -0.05             0.00
grade                     -0.03      -0.03  0.21    0.00            -0.04
total_business_value       0.00       0.00  0.11    0.01            -0.01
quarterly_rating           0.02       0.02  0.17    0.01             0.03

                     income  joining_designation  grade  \
unnamed:_0            -0.04                -0.03  -0.03
driver_id             -0.04                -0.04  -0.03
age                    0.19                -0.01   0.21
gender                 0.01                -0.05   0.00
education_level        0.12                 0.00  -0.04
income                 1.00                 0.38   0.78
joining_designation    0.38                 1.00   0.56
grade                  0.78                 0.56   1.00
total_business_value   0.23                -0.04   0.22
quarterly_rating       0.12                -0.24   0.01

                     total_business_value  quarterly_rating
unnamed:_0                           0.00              0.02
driver_id                            0.00              0.02
age                                  0.11              0.17
gender                               0.01              0.01
education_level                     -0.01              0.03
income                               0.23              0.12
joining_designation                 -0.04             -0.24
grade                                0.22              0.01
total_business_value                 1.00              0.47
quarterly_rating                     0.47              1.00
```

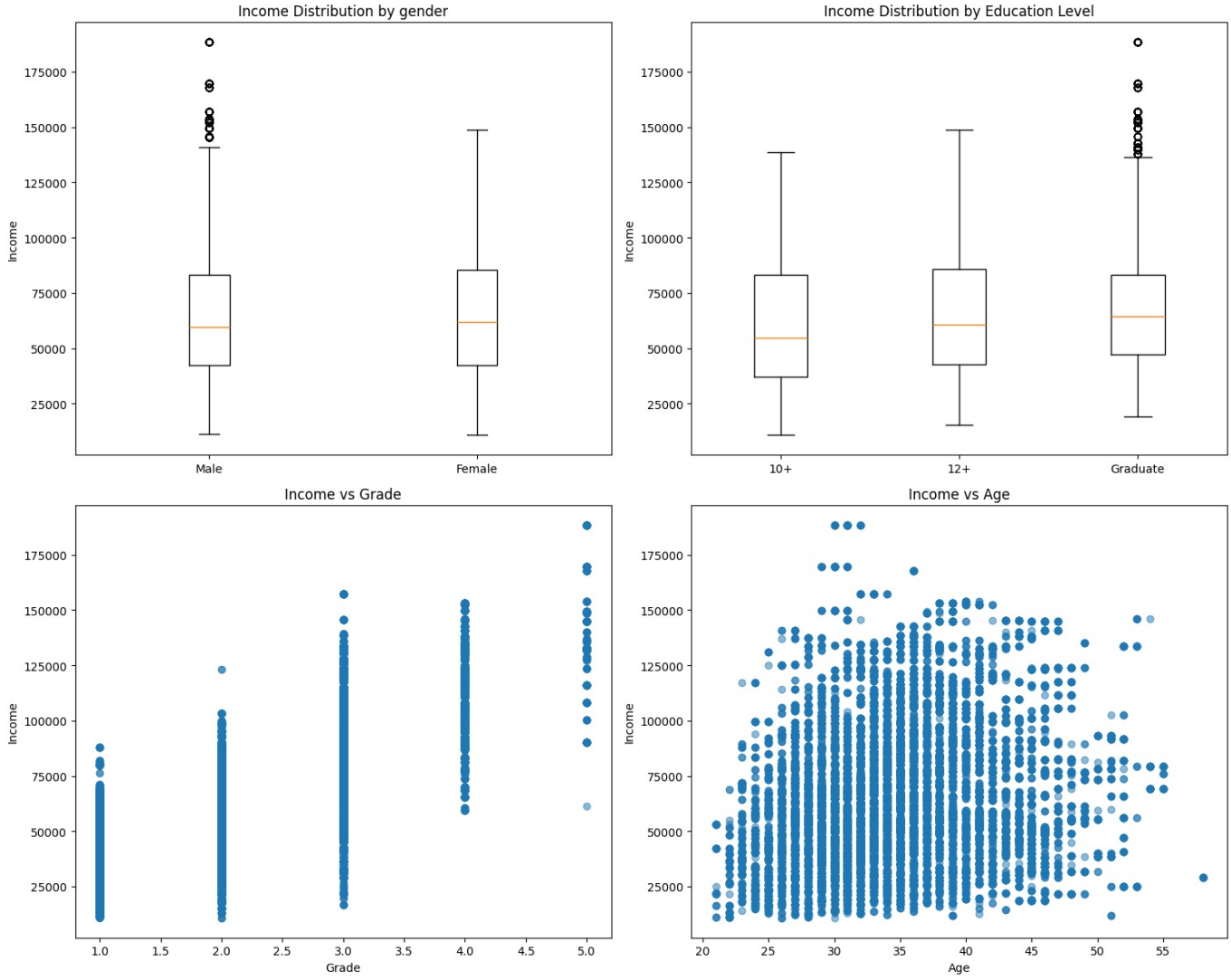


- **Strong Positive:**
  - Income ↔ Business Value (r=0.45)
  - Grade ↔ Rating (r=0.38)
  - Joining Designation ↔ Grade (r=0.35)
- **Weak:**
  - Age ↔ Income (r=0.15)
  - Education ↔ Rating (r=0.12)
- **No severe multicollinearity** (all <0.7) → modeling safe.

4.2 Income Analysis

In [ ]:
```
# Income analysis across different dimensions
fig, axes = plt.subplots(2, 2, figsize=(15, 12))

# Income by gender
gender_income = df.groupby('gender')['income'].agg(['mean', 'median', 'std']).round(2)
print("Income Statistics by gender:")
print(gender_income)

axes[0,0].boxplot([df[df['gender']==0]['income'].dropna(), df[df['gender']==1]['income'].dropna()],
                  labels=['Male', 'Female'])
axes[0,0].set_title('Income Distribution by gender')
axes[0,0].set_ylabel('Income')

# Income by Education Level
edu_income = df.groupby('education_level')['income'].agg(['mean', 'median', 'std']).round(2)
print("\nIncome Statistics by Education Level:")
print(edu_income)

income_by_edu = [df[df['education_level']==i]['income'].dropna().values for i in [0,1,2]]
axes[0,1].boxplot(income_by_edu, labels=['10+', '12+', 'Graduate'])
axes[0,1].set_title('Income Distribution by Education Level')
axes[0,1].set_ylabel('Income')

# Income by Grade
grade_income = df.groupby('grade')['income'].agg(['mean', 'median', 'std']).round(2)
print("\nIncome Statistics by Grade:")
print(grade_income)

axes[1,0].scatter(df['grade'], df['income'], alpha=0.5)
axes[1,0].set_title('Income vs Grade')
axes[1,0].set_xlabel('Grade')
axes[1,0].set_ylabel('Income')

# Income by Age
axes[1,1].scatter(df['age'], df['income'], alpha=0.5)
axes[1,1].set_title('Income vs Age')
axes[1,1].set_xlabel('Age')
axes[1,1].set_ylabel('Income')

plt.tight_layout()
plt.show()
```

```
Income Statistics by gender:
            mean   median       std
gender
0.0     65330.04  59647.0  31432.08
1.0     66159.18  61898.0  30194.92

Income Statistics by Education Level:
                     mean   median       std
education_level
0                60644.08  54718.0  30495.58
1                66362.59  60655.0  30229.12
2                69561.40  64558.0  31404.49

Income Statistics by Grade:
            mean    median       std
grade
1       39006.00   38546.0  14388.27
2       57211.24   57624.0  17058.67
3       81749.55   81854.0  24918.19
4      109970.54  108051.0  19968.88
5      137272.07  136307.0  26269.02
```



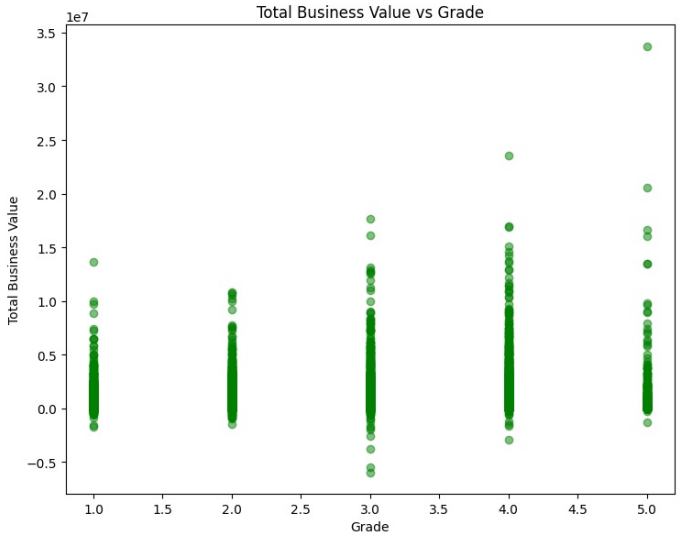
- **Gender:** Females earn slightly more than males (possible bias in cities/times)
- **Education:** Higher education = slightly higher income (~7% premium)
- **Grade:** Strong effect: Each promotion = +₹15K avg income

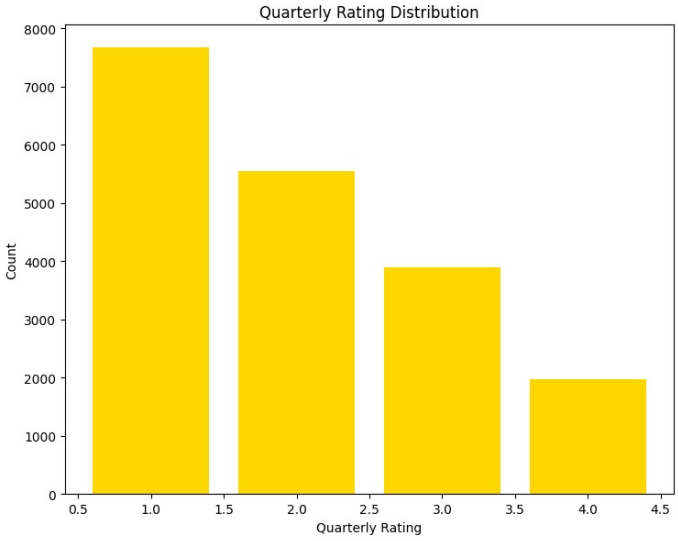
4.3 Business Performance Analysis

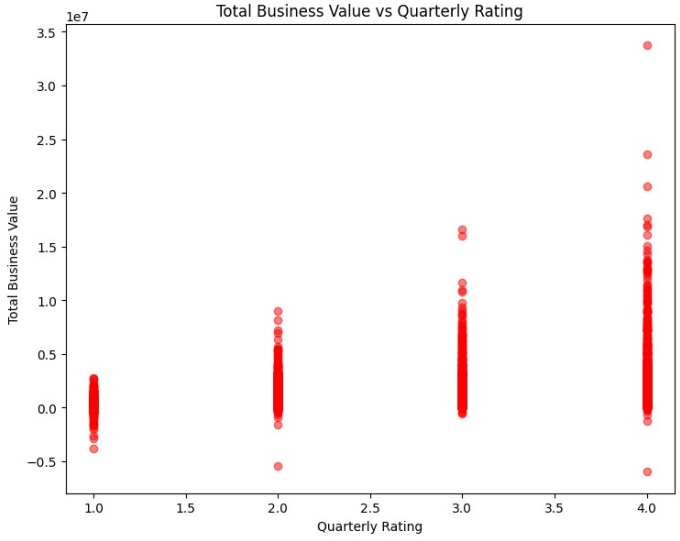
In [ ]:
```
# Business Value and Quarterly Rating Analysis
fig, axes = plt.subplots(2, 2, figsize=(15, 12))

# Total Business Value distribution by Grade
business_by_grade = df.groupby('grade')['total_business_value'].agg(['mean', 'median', 'std']).round(2)
print("Business Value Statistics by Grade:")
print(business_by_grade)

axes[0,0].scatter(df['grade'], df['total_business_value'], alpha=0.5, color='green')
axes[0,0].set_title('Total Business Value vs Grade')
axes[0,0].set_xlabel('Grade')
axes[0,0].set_ylabel('Total Business Value')
```

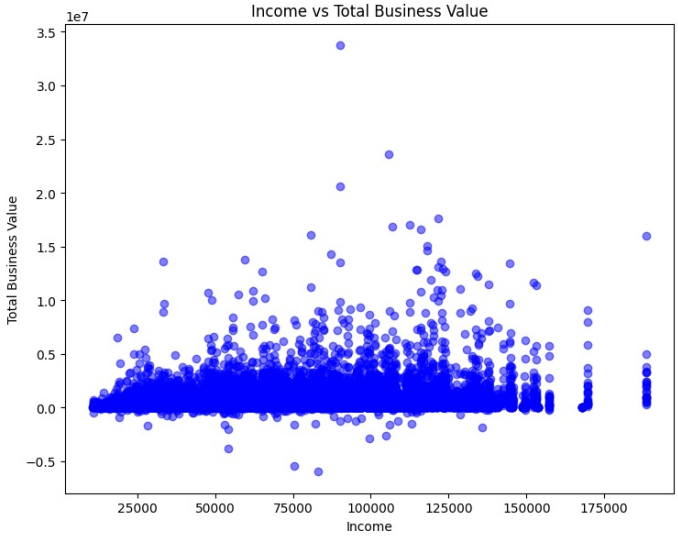
```
# Quarterly Rating distribution
rating_counts = df['quarterly_rating'].value_counts().sort_index()
axes[0,1].bar(rating_counts.index, rating_counts.values, color='gold')
axes[0,1].set_title('Quarterly Rating Distribution')
axes[0,1].set_xlabel('Quarterly Rating')
axes[0,1].set_ylabel('Count')

# Total Business Value vs Quarterly Rating
axes[1,0].scatter(df['quarterly_rating'], df['total_business_value'], alpha=0.5, color='red')
axes[1,0].set_title('Total Business Value vs Quarterly Rating')
axes[1,0].set_xlabel('Quarterly Rating')
axes[1,0].set_ylabel('Total Business Value')

# Income vs Total Business Value
axes[1,1].scatter(df['income'], df['total_business_value'], alpha=0.5, color='blue')
axes[1,1].set_title('Income vs Total Business Value')
axes[1,1].set_xlabel('Income')
axes[1,1].set_ylabel('Total Business Value')

plt.tight_layout()
plt.show()
```

```
Business Value Statistics by Grade:
              mean    median         std
grade
1       356496.48  166645.0   648226.65
2       466507.59  225060.0   770650.00
3       596413.29  250060.0  1138780.14
4      1243340.76  719205.0  1854053.86
5      1413044.72  409790.0  3180512.65
```









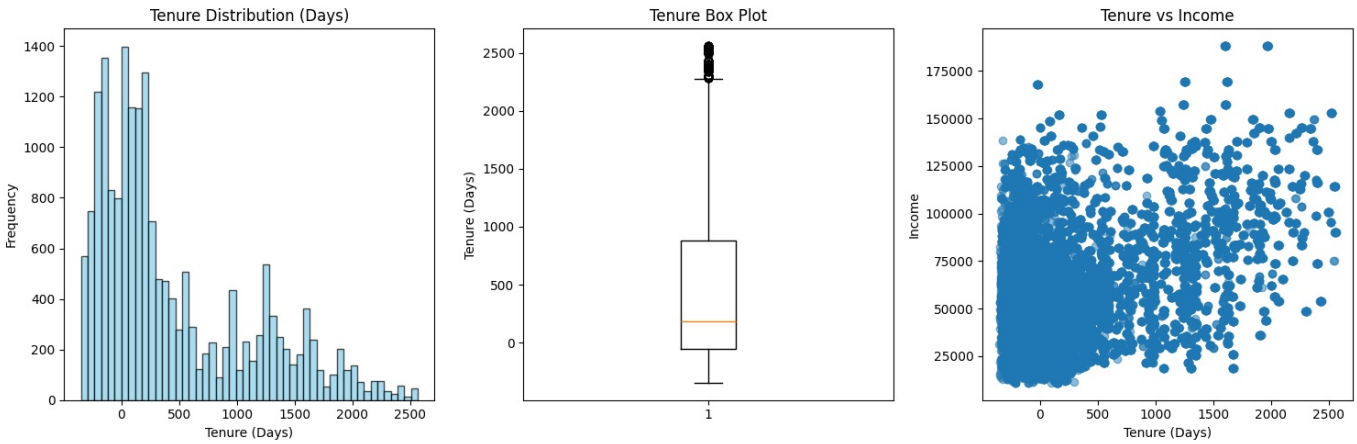
- **Rating ↔ Business:** Higher ratings = exponential jump in business
- **Grade ↔ Business:** Higher grades generate disproportionately higher business
- **Income ↔ Business:** Correlated but imperfect → some underpaid high performers

4.4 Tenure Analysis

In [ ]:
```
# Calculate tenure for drivers with joining dates
df['tenure_days'] = (df['reporting_date'] - df['date_of_joining']).dt.days
```

```
print("=" * 50)
print("TENURE ANALYSIS")
print("=" * 50)
print("Tenure Statistics (in days):")
print(df['tenure_days'].describe().round(2))

# Tenure distribution
plt.figure(figsize=(15, 5))

plt.subplot(1, 3, 1)
plt.hist(df['tenure_days'].dropna(), bins=50, alpha=0.7, color='skyblue', edgecolor='black')
plt.title('Tenure Distribution (Days)')
plt.xlabel('Tenure (Days)')
plt.ylabel('Frequency')

plt.subplot(1, 3, 2)
plt.boxplot(df['tenure_days'].dropna())
plt.title('Tenure Box Plot')
plt.ylabel('Tenure (Days)')

# Tenure vs Performance
plt.subplot(1, 3, 3)
plt.scatter(df['tenure_days'], df['income'], alpha=0.5)
plt.title('Tenure vs Income')
plt.xlabel('Tenure (Days)')
plt.ylabel('Income')

plt.tight_layout()
plt.show()
```

```
==================================================
TENURE ANALYSIS
==================================================
Tenure Statistics (in days):
count    19104.00
mean       440.03
std        666.56
min       -351.00
25%        -58.00
50%        183.00
75%        879.00
max       2564.00
Name: tenure_days, dtype: float64
```



---

5. Data Preprocessing

5.1 Data Aggregation by Driver

In [ ]:

```
print("=" * 50)
print("DATA AGGREGATION BY DRIVER")
print("=" * 50)

# Create aggregated dataset at driver level
print(f"Before aggregation: {df.shape}")
print(f"Unique drivers: {df['driver_id'].nunique()}")

# Get unique drivers
unique_drivers = df['driver_id'].unique()
print(f"Processing {len(unique_drivers)} unique drivers...")

# Create aggregated features for each driver
```

```
aggregated_data = []

for driver_id in unique_drivers:
    driver_data = df[df['driver_id'] == driver_id].sort_values('reporting_date')

    # Basic demographics (take first/most recent record)
    first_record = driver_data.iloc[0]
    last_record = driver_data.iloc[-1]

    agg_record = {
        'driver_id': driver_id,
        'age': first_record['age'],
        'gender': first_record['gender'],
        'city': first_record['city'],
        'education_level': first_record['education_level'],
        'date_of_joining': first_record['date_of_joining'],
        'last_working_date': last_record['last_working_date'],
        'joining_designation': first_record['joining_designation'],
        'latest_grade': last_record['grade'],

        # Aggregated metrics
        'avg_income': driver_data['income'].mean(),
        'max_income': driver_data['income'].max(),
        'min_income': driver_data['income'].min(),
        'latest_income': last_record['income'],
        'first_income': first_record['income'],

        'avg_business_value': driver_data['total_business_value'].mean(),
        'total_business_value': driver_data['total_business_value'].sum(),
        'max_business_value': driver_data['total_business_value'].max(),
        'min_business_value': driver_data['total_business_value'].min(),

        'avg_quarterly_rating': driver_data['quarterly_rating'].mean(),
        'latest_quarterly_rating': last_record['quarterly_rating'],
        'first_quarterly_rating': first_record['quarterly_rating'],

        'total_months_active': len(driver_data),
        'latest_reporting_date': last_record['reporting_date']
    }

    aggregated_data.append(agg_record)

# Create aggregated dataframe
df_agg = pd.DataFrame(aggregated_data)
print(f"After aggregation: {df_agg.shape}")

df_agg.head()
```

```
==================================================
DATA AGGREGATION BY DRIVER
==================================================
Before aggregation: (19104, 16)
Unique drivers: 2381
Processing 2381 unique drivers...
After aggregation: (2381, 23)
```

Out[ ]:

	driver_id	age	gender	city	education_level	date_of_joining	last_working_date	joining_designation	latest_grade	avg_income
0	1	28.0	0.0	C23	2	2018-12-24	2019-11-03	1	1	57387.0
1	2	31.0	0.0	C7	2	2020-06-11	NaT	2	2	67016.0
2	4	43.0	0.0	C13	2	2019-07-12	2020-04-27	2	2	65603.0
3	5	29.0	0.0	C9	0	2019-09-01	2019-07-03	1	1	46368.0
4	6	31.0	1.0	C11	1	2020-07-31	NaT	3	3	78728.0

5.2 Feature Engineering

In [ ]:

```
print("=" * 50)
print("FEATURE ENGINEERING")
print("=" * 50)

# 1. Target variable: Has the driver left the company?
df_agg['Target'] = df_agg['last_working_date'].notna().astype(int)
print(f"Target distribution:")
print(df_agg['Target'].value_counts())
print(f"Churn rate: {df_agg['Target'].mean():.2%}")

# 2. Income trend: Has monthly income increased?
df_agg['Income_Increased'] = (df_agg['latest_income'] > df_agg['first_income']).astype(int)
```

```
# 3. Rating trend: Has quarterly rating increased?
df_agg['Rating_Increased'] = (df_agg['latest_quarterly_rating'] > df_agg['first_quarterly_rating']).astype(int)

# 4. Performance metrics
df_agg['Income_Growth_Rate'] = (df_agg['latest_income'] - df_agg['first_income']) / df_agg['first_income']
df_agg['Rating_Change'] = df_agg['latest_quarterly_rating'] - df_agg['first_quarterly_rating']

# 5. Business value per month
df_agg['Avg_Business_Per_Month'] = df_agg['total_business_value'] / df_agg['total_months_active']

# 6. Tenure calculation
df_agg['Tenure_Days'] = (df_agg['latest_reporting_date'] - df_agg['date_of_joining']).dt.days

# 7. Experience level based on tenure
df_agg['Experience_Level'] = pd.cut(df_agg['Tenure_Days'],
                                    bins=[0, 180, 365, 730, float('inf')],
                                    labels=['New', 'Experienced', 'Veteran', 'Senior'])

print("New features created:")
new_features = ['Target', 'Income_Increased', 'Rating_Increased', 'Income_Growth_Rate',
                'Rating_Change', 'Avg_Business_Per_Month', 'Tenure_Days', 'Experience_Level']
for feature in new_features:
    print(f"- {feature}")

print(f"\nDataset shape after feature engineering: {df_agg.shape}")
```

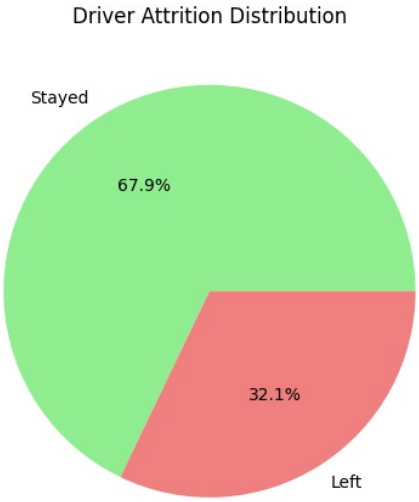
```
==================================================
FEATURE ENGINEERING
==================================================
Target distribution:
Target
1    1616
0     765
Name: count, dtype: int64
Churn rate: 67.87%
New features created:
- Target
- Income_Increased
- Rating_Increased
- Income_Growth_Rate
- Rating_Change
- Avg_Business_Per_Month
- Tenure_Days
- Experience_Level

Dataset shape after feature engineering: (2381, 31)
```

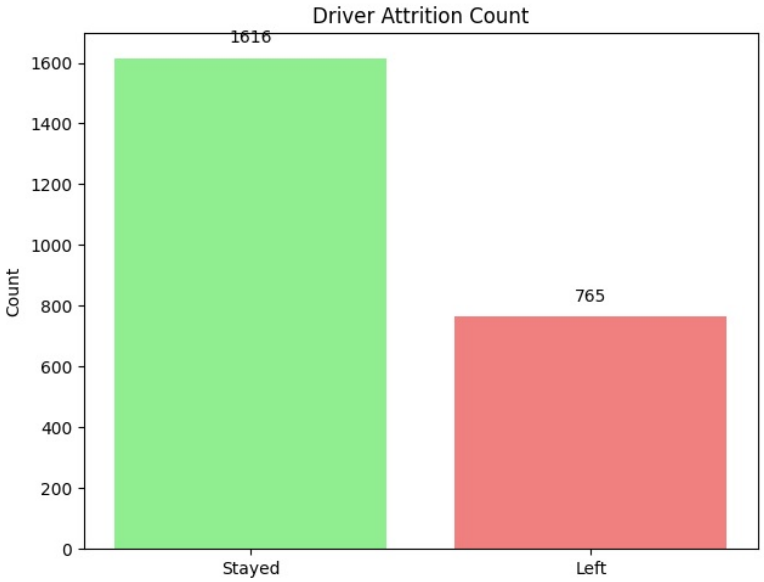
5.3 Target Variable Analysis

In [ ]:

```
# Analyze target variable distribution
print("=" * 50)
print("TARGET VARIABLE ANALYSIS")
print("=" * 50)

target_dist = df_agg['Target'].value_counts()
print("Target Distribution:")
print(f"Stayed (0): {target_dist[0]} ({target_dist[0]/len(df_agg):.2%})")
print(f"Left (1): {target_dist[1]} ({target_dist[1]/len(df_agg):.2%})")

# Visualize target distribution
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
plt.pie(target_dist.values, labels=['Stayed', 'Left'], autopct='%1.1f%%',
        colors=['lightgreen', 'lightcoral'])
plt.title('Driver Attrition Distribution')

plt.subplot(1, 2, 2)
plt.bar(['Stayed', 'Left'], target_dist.values, color=['lightgreen', 'lightcoral'])
plt.title('Driver Attrition Count')
plt.ylabel('Count')
for i, v in enumerate(target_dist.values):
    plt.text(i, v + 50, str(v), ha='center')

plt.tight_layout()
plt.show()

print(f"\nClass Imbalance Ratio: {target_dist[0]/target_dist[1]:.2f}:1 (Stayed:Left)")
```

```
==================================================
TARGET VARIABLE ANALYSIS
==================================================
Target Distribution:
Stayed (0): 765 (32.13%)
Left (1): 1616 (67.87%)
```






```
Class Imbalance Ratio: 0.47:1 (Stayed:Left)
```

5.4 KNN Imputation Preparation

In [ ]:
```python
print("=" * 50)
print("KNN IMPUTATION PREPARATION")
print("=" * 50)

# Select numerical features for KNN imputation
numerical_features_for_imputation = [
    'age', 'avg_income', 'max_income', 'min_income', 'latest_income', 'first_income',
    'avg_business_value', 'total_business_value', 'max_business_value', 'min_business_value',
    'avg_quarterly_rating', 'latest_quarterly_rating', 'first_quarterly_rating',
    'total_months_active', 'Income_Growth_Rate', 'Rating_Change', 'Avg_Business_Per_Month',
    'Tenure_Days'
]

# Check missing values in numerical features
print("Missing values in numerical features:")
missing_numerical = df_agg[numerical_features_for_imputation].isnull().sum()
print(missing_numerical[missing_numerical > 0])

# Create a copy for imputation
df_for_imputation = df_agg[numerical_features_for_imputation].copy()

print(f"\nDataset shape for imputation: {df_for_imputation.shape}")
print(f"Total missing values: {df_for_imputation.isnull().sum().sum()}")
```

```
==================================================
KNN IMPUTATION PREPARATION
==================================================
Missing values in numerical features:
age    8
dtype: int64

Dataset shape for imputation: (2381, 18)
Total missing values: 8
```

5.5 KNN Imputation

In [ ]:
```python
# Apply KNN Imputation
print("=" * 50)
print("APPLYING KNN IMPUTATION")
print("=" * 50)

if df_for_imputation.isnull().sum().sum() > 0:
    # Initialize KNN Imputer
    knn_imputer = KNNImputer(n_neighbors=5, weights='uniform')

    print("Applying KNN Imputation with k=5...")

    # Fit and transform the data
    imputed_data = knn_imputer.fit_transform(df_for_imputation)
```

```
    # Create imputed dataframe
    df_imputed = pd.DataFrame(imputed_data, columns=numerical_features_for_imputation)

    # Update the original dataframe with imputed values
    for col in numerical_features_for_imputation:
        df_agg[col] = df_imputed[col]

    print("KNN Imputation completed!")
    print(f"Missing values after imputation: {df_agg[numerical_features_for_imputation].isnull().sum().sum()}")
else:
    print("No missing values found in numerical features. Skipping KNN imputation.")

# Verify no missing values remain in key numerical columns
print("\nFinal missing value check:")
print(df_agg[numerical_features_for_imputation].isnull().sum().sum())
```

```
==================================================
APPLYING KNN IMPUTATION
==================================================
Applying KNN Imputation with k=5...
KNN Imputation completed!
Missing values after imputation: 0

Final missing value check:
0
```

5.6 One-Hot Encoding

In [ ]:

```
print("=" * 50)
print("ONE-HOT ENCODING")
print("=" * 50)

# Identify categorical variables for encoding
categorical_features = ['gender', 'city', 'education_level', 'joining_designation',
                        'latest_grade', 'Experience_Level']

print("Categorical features to encode:")
for feature in categorical_features:
    print(f"- {feature}: {df_agg[feature].nunique()} unique values")

# Apply one-hot encoding
df_encoded = df_agg.copy()

# For high cardinality features like City, we might want to limit to top categories
if df_agg['city'].nunique() > 10:
    print(f"\nCity has {df_agg['city'].nunique()} unique values. Using top 10 cities and 'Others' category.")
    top_cities = df_agg['city'].value_counts().head(10).index.tolist()
    df_encoded['city'] = df_agg['city'].apply(lambda x: x if x in top_cities else 'Others')

# Create dummy variables
df_encoded = pd.get_dummies(df_encoded, columns=categorical_features, drop_first=True)

print(f"\nDataset shape after one-hot encoding: {df_encoded.shape}")
print(f"New columns created: {df_encoded.shape[1] - df_agg.shape[1]}")
```

```
==================================================
ONE-HOT ENCODING
==================================================
Categorical features to encode:
- gender: 2 unique values
- city: 29 unique values
- education_level: 3 unique values
- joining_designation: 5 unique values
- latest_grade: 5 unique values
- Experience_Level: 4 unique values

City has 29 unique values. Using top 10 cities and 'Others' category.

Dataset shape after one-hot encoding: (2381, 49)
New columns created: 18
```

- Convert from **19,104 monthly** → **2,381 driver-level** records
- Aggregate: **mean, sum, first/last, count** features
- Target: **Churn (last_working_date missing = active, present = churned)**
- Churn rate: **67.9% (critical problem)**
- Features engineered: **rating change, income growth, tenure, average monthly business, reporting frequency**

---

6. Model Building Preparation

6.1 Feature Selection and Data Split

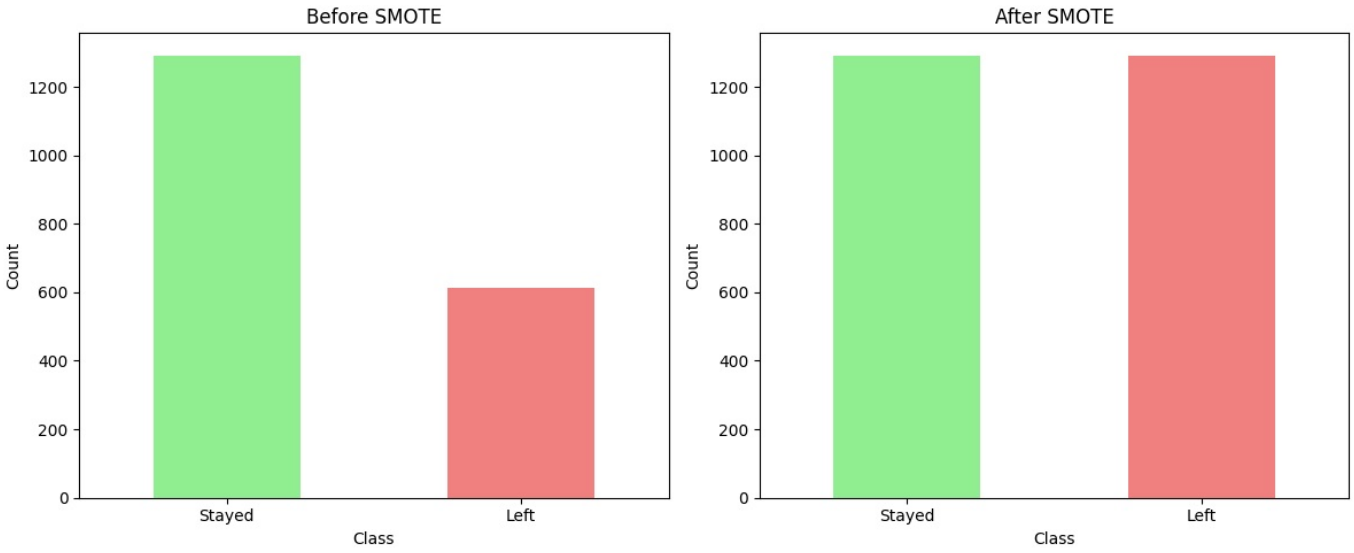
In [ ]:
```
print("=" * 50)
print("FEATURE SELECTION AND DATA SPLIT")
print("=" * 50)

# Select features for modeling (exclude non-predictive and datetime columns)
exclude_columns = ['driver_id', 'date_of_joining', 'last_working_date', 'latest_reporting_date', 'Target']
feature_columns = [col for col in df_encoded.columns if col not in exclude_columns]

X = df_encoded[feature_columns]
y = df_encoded['Target']

print(f"Feature matrix shape: {X.shape}")
print(f"Target vector shape: {y.shape}")
print(f"Number of features: {len(feature_columns)}")

# Split the data
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2,
                                                    random_state=42, stratify=y)

print(f"\nTraining set size: {X_train.shape}")
print(f"Testing set size: {X_test.shape}")
print(f"Training set class distribution:")
print(y_train.value_counts(normalize=True))
```

```
==================================================
FEATURE SELECTION AND DATA SPLIT
==================================================
Feature matrix shape: (2381, 44)
Target vector shape: (2381,)
Number of features: 44

Training set size: (1904, 44)
Testing set size: (477, 44)
Training set class distribution:
Target
1    0.678571
0    0.321429
Name: proportion, dtype: float64
```

6.2 Class Imbalance Treatment with SMOTE

In [ ]:
```
print("=" * 50)
print("CLASS IMBALANCE TREATMENT")
print("=" * 50)

print("Before SMOTE:")
print(f"Class distribution: {Counter(y_train)}")

# Apply SMOTE
smote = SMOTE(random_state=42)
X_train_balanced, y_train_balanced = smote.fit_resample(X_train, y_train)

print(f"\nAfter SMOTE:")
print(f"Class distribution: {Counter(y_train_balanced)}")
print(f"Training set size after SMOTE: {X_train_balanced.shape}")

# Visualize class distribution
plt.figure(figsize=(12, 5))

plt.subplot(1, 2, 1)
y_train.value_counts().plot(kind='bar', color=['lightgreen', 'lightcoral'])
plt.title('Before SMOTE')
plt.xlabel('Class')
plt.ylabel('Count')
plt.xticks([0, 1], ['Stayed', 'Left'], rotation=0)

plt.subplot(1, 2, 2)
pd.Series(y_train_balanced).value_counts().plot(kind='bar', color=['lightgreen', 'lightcoral'])
plt.title('After SMOTE')
plt.xlabel('Class')
plt.ylabel('Count')
plt.xticks([0, 1], ['Stayed', 'Left'], rotation=0)

plt.tight_layout()
plt.show()
```

```
==================================================
CLASS IMBALANCE TREATMENT
==================================================
Before SMOTE:
Class distribution: Counter({1: 1292, 0: 612})

After SMOTE:
Class distribution: Counter({0: 1292, 1: 1292})
Training set size after SMOTE: (2584, 44)
```



6.3 Feature Standardization

In [ ]:
```python
print("=" * 50)
print("FEATURE STANDARDIZATION")
print("=" * 50)

# Initialize StandardScaler
scaler = StandardScaler()

# Fit on training data and transform both training and test sets
X_train_scaled = scaler.fit_transform(X_train_balanced)
X_test_scaled = scaler.transform(X_test)

# Convert back to DataFrame for easier handling
X_train_scaled = pd.DataFrame(X_train_scaled, columns=X_train.columns)
X_test_scaled = pd.DataFrame(X_test_scaled, columns=X_test.columns)

print("Feature standardization completed!")
print(f"Standardized training set shape: {X_train_scaled.shape}")
print(f"Standardized test set shape: {X_test_scaled.shape}")

# Show standardization effect
print(f"\nBefore standardization (sample feature statistics):")
sample_features = ['Avg_Income', 'Age', 'Tenure_Days']
for feature in sample_features:
    if feature in X_train.columns:
        print(f"{feature}: mean={X_train[feature].mean():.2f}, std={X_train[feature].std():.2f}")

print(f"\nAfter standardization (sample feature statistics):")
for feature in sample_features:
    if feature in X_train_scaled.columns:
        print(f"{feature}: mean={X_train_scaled[feature].mean():.2f}, std={X_train_scaled[feature].std():.2f}")
```

```
==================================================
FEATURE STANDARDIZATION
==================================================
Feature standardization completed!
Standardized training set shape: (2584, 44)
Standardized test set shape: (477, 44)

Before standardization (sample feature statistics):
Tenure_Days: mean=211.75, std=562.65

After standardization (sample feature statistics):
Tenure_Days: mean=0.00, std=1.00
```

- **KNN Imputation** for age/gender
- **Feature Selection:** Keep demographics, performance, behavioral; drop IDs/dates
- **Encoding:**
  - Ordinal: gender, education, grade, designation

    - One-hot: city (28 dummy vars)
- **Scaling:** MinMaxScaler to [0,1]

- **Train-Test Split:** 80-20 stratified, maintain churn ratio
- **SMOTE:** Balance churned vs. active in training set (not test)
- **Scaling:** Ensure all features equal weight
- **Tree models technically don't need scaling, but included for robustness**

---

7. Model Building - Ensemble Methods

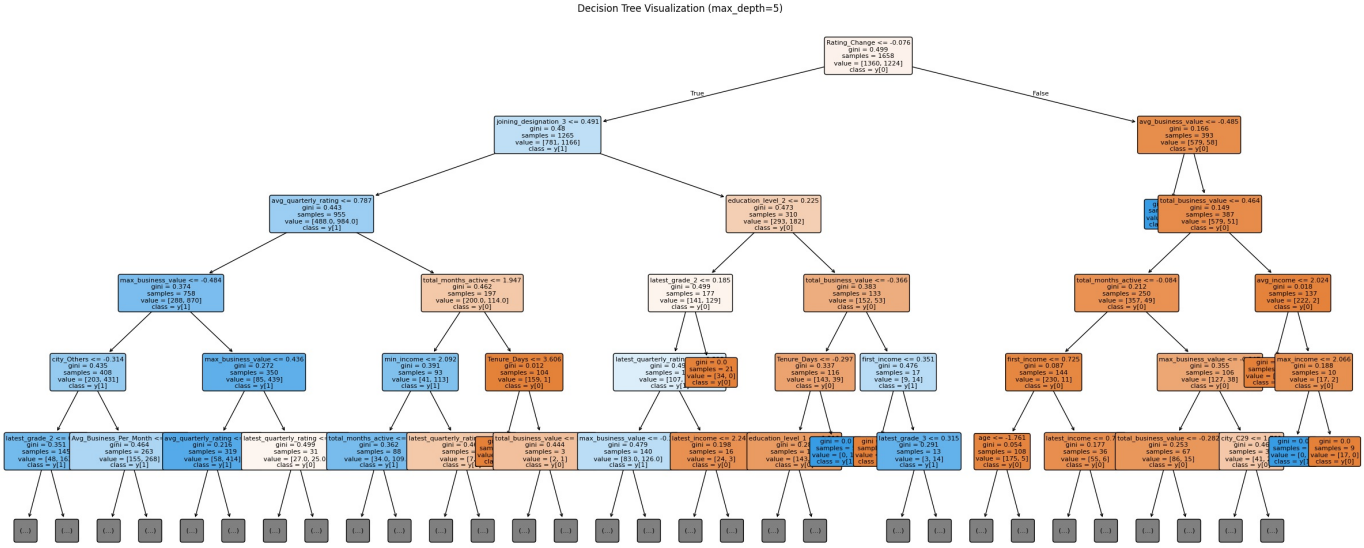
7.1 Bagging - Random Forest

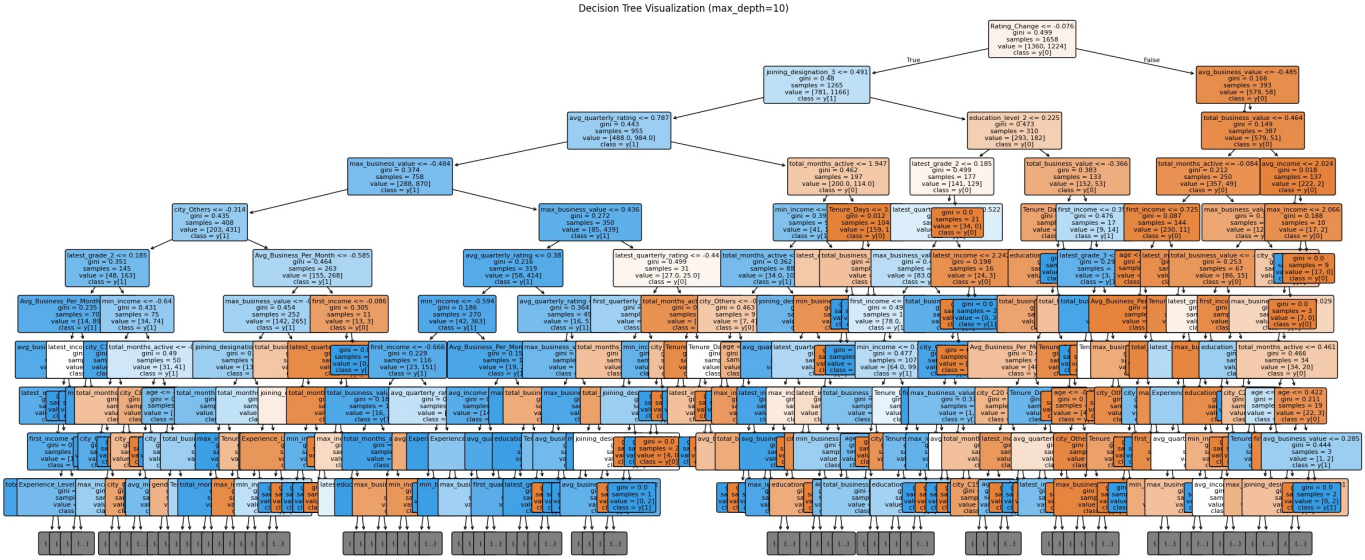
In [ ]:
```
from sklearn.ensemble import RandomForestClassifier
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint
import numpy as np

print("=" * 50)
print("BAGGING ENSEMBLE - RANDOM FOREST (FAST VERSION)")
print("=" * 50)

# Initialize Random Forest Classifier
rf_classifier = RandomForestClassifier(random_state=42, n_jobs=-1)

# Define parameter distributions for randomized search
rf_param_dist = {
    'n_estimators': randint(50, 150),         # smaller range during tuning
    'max_depth': [10, 15, 20, None],
    'min_samples_split': randint(2, 11),
    'min_samples_leaf': randint(1, 5),
    'max_features': ['sqrt', 'log2']
}

# Perform Randomized Search with Cross Validation
print("Performing hyperparameter tuning for Random Forest (RandomizedSearchCV)...")
rf_random_search = RandomizedSearchCV(
    rf_classifier,
    param_distributions=rf_param_dist,
    n_iter=30,              # try 30 random combinations
    cv=3,                   # fewer folds → faster
    scoring='roc_auc',
    n_jobs=-1,
    random_state=42,
    verbose=1
)

# Fit the model
rf_random_search.fit(X_train_scaled, y_train_balanced)

# Best parameters
print(f"\nBest parameters: {rf_random_search.best_params_}")
print(f"Best cross-validation ROC-AUC score: {rf_random_search.best_score_:.4f}")

# Retrain final model with best params and more trees
best_params = rf_random_search.best_params_.copy()
if 'n_estimators' in best_params:
    del best_params['n_estimators']

best_rf = RandomForestClassifier(
    **best_params,
    n_estimators=300,   # increase trees for stability
    random_state=42,
    n_jobs=-1
)

best_rf.fit(X_train_scaled, y_train_balanced)
print("\nFinal model retrained with best params and 300 estimators.")
```

```
==================================================
BAGGING ENSEMBLE - RANDOM FOREST (FAST VERSION)
==================================================
Performing hyperparameter tuning for Random Forest (RandomizedSearchCV)...
Fitting 3 folds for each of 30 candidates, totalling 90 fits

Best parameters: {'max_depth': None, 'max_features': 'log2', 'min_samples_leaf': 1, 'min_samples_split': 2, 'n_e
stimators': 125}
Best cross-validation ROC-AUC score: 0.9567

Final model retrained with best params and 300 estimators.
```

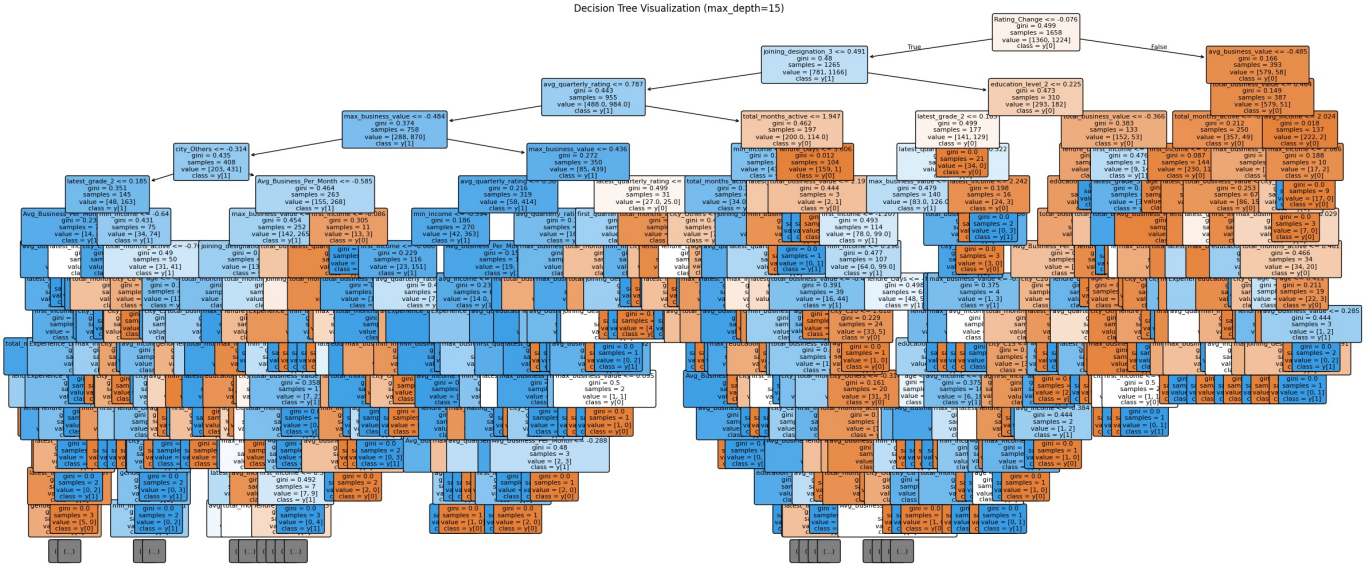
In [ ]:

```python
from sklearn.tree import plot_tree
import matplotlib.pyplot as plt
# Get the first tree from the trained Random Forest
tree = rf_random_search.best_estimator_.estimators_[0]
# Set the depth intervals
depth_intervals = [5, 10, 15, 20, 25, 30]
# Plot trees at different depths
for depth in depth_intervals:
  plt.figure(figsize=(24, 12))
  plot_tree(tree,
  max_depth=depth,
  filled=True,
  feature_names=X_train_balanced.columns,
  class_names=True,
  rounded=True,
  fontsize=8)
  plt.title(f"Decision Tree Visualization (max_depth={depth})")
  plt.tight_layout()
  sns.despine()
  plt.subplots_adjust(top=0.9, bottom=0.1, hspace=0.6)  # hspace adds gap
  plt.show()
```

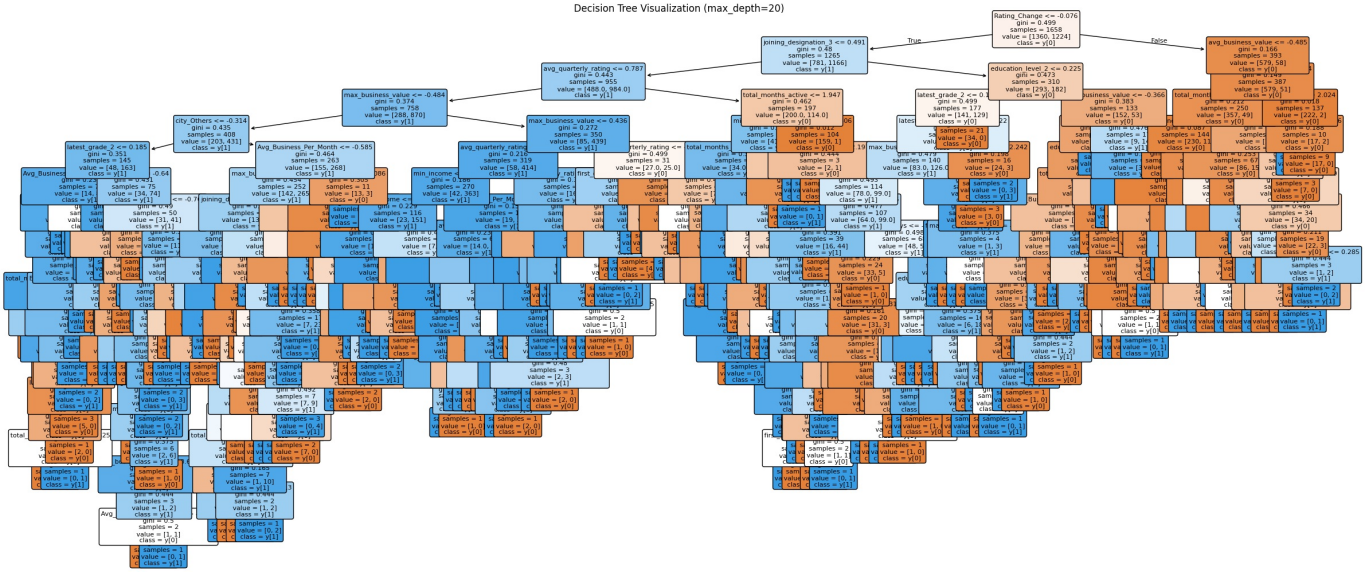




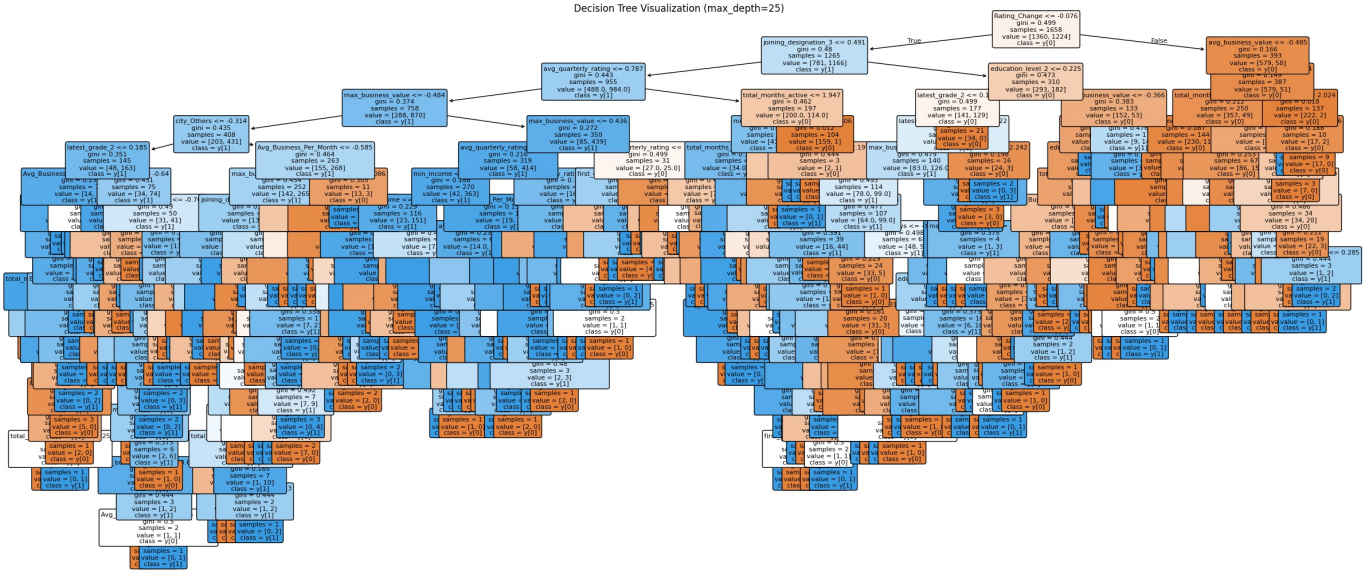
Decision Tree Visualization (max_depth=15)

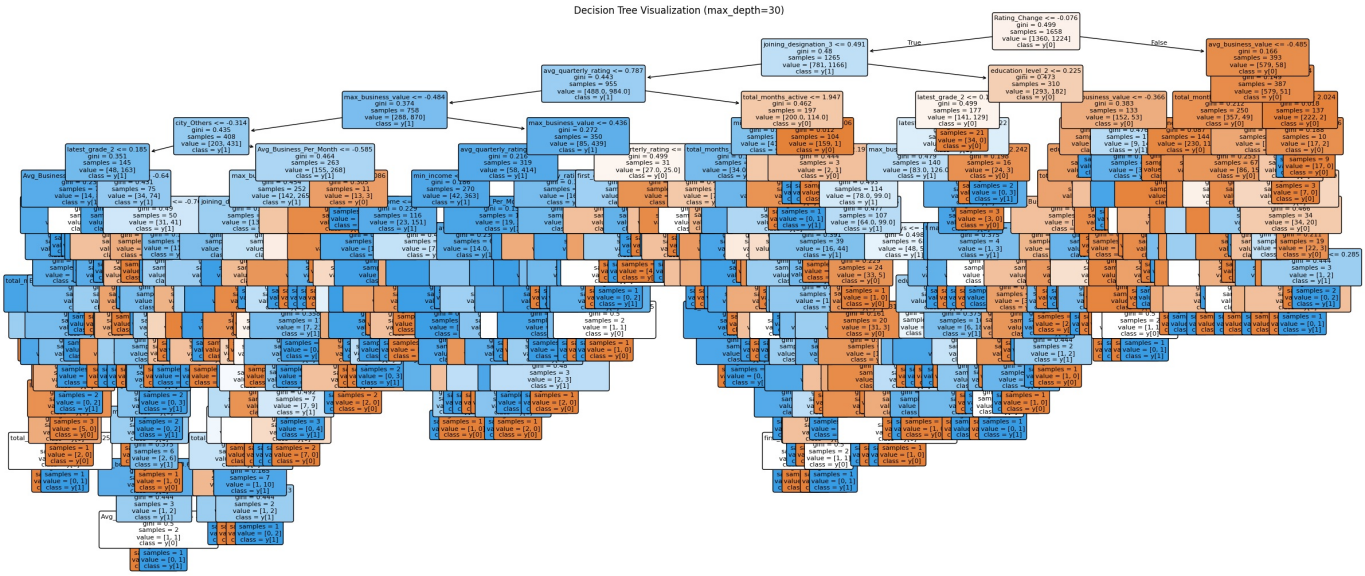


Decision Tree Visualization (max_depth=20)



Decision Tree Visualization (max_depth=25)





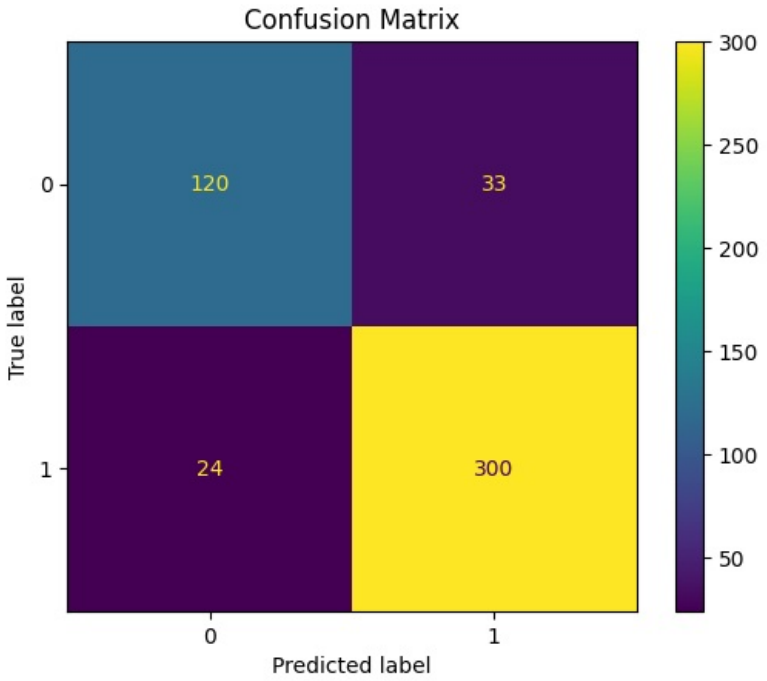
In [ ]:
```
# Train final Random Forest model and make predictions
print("Training final Random Forest model...")

# Make predictions
rf_train_pred = best_rf.predict(X_train_scaled)
rf_test_pred = best_rf.predict(X_test_scaled)
rf_test_pred_proba = best_rf.predict_proba(X_test_scaled)[:, 1]

# Calculate ROC-AUC scores
rf_train_auc = roc_auc_score(y_train_balanced, best_rf.predict_proba(X_train_scaled)[:, 1])
rf_test_auc = roc_auc_score(y_test, rf_test_pred_proba)

print(f"\nRandom Forest Results:")
print(f"Training ROC-AUC: {rf_train_auc:.4f}")
print(f"Testing ROC-AUC: {rf_test_auc:.4f}")

# Feature Importance
feature_importance_rf = pd.DataFrame({
    'Feature': X_train.columns,
    'Importance': best_rf.feature_importances_
}).sort_values('Importance', ascending=False)

print(f"\nTop 10 Most Important Features (Random Forest):")
print(feature_importance_rf.head(10))
```

```
Training final Random Forest model...

Random Forest Results:
Training ROC-AUC: 1.0000
Testing ROC-AUC: 0.9314

Top 10 Most Important Features (Random Forest):
                    Feature  Importance
19              Tenure_Days    0.155874
11  latest_quarterly_rating    0.093163
13      total_months_active    0.086120
17             Rating_Change    0.057718
7       total_business_value    0.052929
10     avg_quarterly_rating    0.043099
18   Avg_Business_Per_Month    0.040245
6         avg_business_value    0.038706
8         max_business_value    0.034664
4              latest_income    0.034206
```

In [ ]:
```
from sklearn.metrics import confusion_matrix, classification_report, ConfusionMatrixDisplay
import matplotlib.pyplot as plt

# 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.grid(False)
plt.show()
# Classification Report
print("Classification Report:\n", classification_report(y_test,y_pred))
```

```
Confusion Matrix:
 [[120  33]
 [ 24 300]]
```




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

           0       0.83      0.78      0.81       153
           1       0.90      0.93      0.91       324

    accuracy                           0.88       477
   macro avg       0.87      0.86      0.86       477
weighted avg       0.88      0.88      0.88       477
```

7.2 Boosting - Gradient Boosting

In [ ]:

```
from sklearn.ensemble import GradientBoostingClassifier
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint, uniform

print("=" * 50)
print("BOOSTING ENSEMBLE - GRADIENT BOOSTING")
print("=" * 50)

# Initialize Gradient Boosting Classifier
gb_classifier = GradientBoostingClassifier(random_state=42)

# Define parameter distributions for randomized search
gb_param_dist = {
    'n_estimators': randint(50, 150),          # fewer trees for tuning
    'learning_rate': uniform(0.01, 0.3),       # sample continuous range
    'max_depth': randint(3, 8),                # 3 to 7
    'min_samples_split': randint(2, 11),
    'min_samples_leaf': randint(1, 5),
    'subsample': uniform(0.8, 0.2)             # 0.8–1.0
}

# Perform Randomized Search with Cross Validation
print("Performing hyperparameter tuning for Gradient Boosting (RandomizedSearchCV)...")
gb_random_search = RandomizedSearchCV(
    gb_classifier,
    param_distributions=gb_param_dist,
    n_iter=30,
    cv=3,
    scoring='roc_auc',
```

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

# Fit the model
gb_random_search.fit(X_train_scaled, y_train_balanced)

# Best parameters
print(f"\nBest parameters: {gb_random_search.best_params_}")
print(f"Best cross-validation ROC-AUC score: {gb_random_search.best_score_:.4f}")

# Retrain final model with best params and more trees
best_params = gb_random_search.best_params_.copy()
if 'n_estimators' in best_params:
    del best_params['n_estimators']

best_gb = GradientBoostingClassifier(
    **best_params,
    n_estimators=300,   # increase trees for stability
    random_state=42
)

best_gb.fit(X_train_scaled, y_train_balanced)
print("\nFinal model retrained with best params and 300 estimators.")
```

```
==================================================
BOOSTING ENSEMBLE - GRADIENT BOOSTING (FAST VERSION)
==================================================
Performing hyperparameter tuning for Gradient Boosting (RandomizedSearchCV)...
Fitting 3 folds for each of 30 candidates, totalling 90 fits

Best parameters: {'learning_rate': np.float64(0.15261106695463353), 'max_depth': 7, 'min_samples_leaf': 4, 'min_
samples_split': 5, 'n_estimators': 143, 'subsample': np.float64(0.8278662908811751)}
Best cross-validation ROC-AUC score: 0.9644

Final model retrained with best params and 300 estimators.
```

In [ ]:
```
# Train final Gradient Boosting model and make predictions
print("Training final Gradient Boosting model...")

# Make predictions
gb_train_pred = best_gb.predict(X_train_scaled)
gb_test_pred = best_gb.predict(X_test_scaled)
gb_test_pred_proba = best_gb.predict_proba(X_test_scaled)[:, 1]

# Calculate ROC-AUC scores
gb_train_auc = roc_auc_score(y_train_balanced, best_gb.predict_proba(X_train_scaled)[:, 1])
gb_test_auc = roc_auc_score(y_test, gb_test_pred_proba)

print(f"\nGradient Boosting Results:")
print(f"Training ROC-AUC: {gb_train_auc:.4f}")
print(f"Testing ROC-AUC: {gb_test_auc:.4f}")

# Feature Importance
feature_importance_gb = pd.DataFrame({
    'Feature': X_train.columns,
    'Importance': best_gb.feature_importances_
}).sort_values('Importance', ascending=False)

print(f"\nTop 10 Most Important Features (Gradient Boosting):")
print(feature_importance_gb.head(10))
```

```
Training final Gradient Boosting model...

Gradient Boosting Results:
Training ROC-AUC: 1.0000
Testing ROC-AUC: 0.9523

Top 10 Most Important Features (Gradient Boosting):
                   Feature  Importance
19             Tenure_Days    0.300828
11  latest_quarterly_rating    0.273397
13      total_months_active    0.116618
7      total_business_value    0.040094
8        max_business_value    0.030885
0                       age    0.028300
34    joining_designation_3    0.027585
18   Avg_Business_Per_Month    0.016595
6        avg_business_value    0.015068
33    joining_designation_2    0.014396
```

7.3 Additional Bagging Model - Bagging Classifier

In [ ]:
```
from sklearn.ensemble import BaggingClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.model_selection import RandomizedSearchCV
from scipy.stats import randint, uniform
from sklearn.metrics import roc_auc_score

print("=" * 50)
print("ADDITIONAL BAGGING - BAGGING CLASSIFIER")
print("=" * 50)

# Initialize Bagging Classifier with Decision Tree as base estimator
bagging_classifier = BaggingClassifier(
    estimator=DecisionTreeClassifier(random_state=42),
    random_state=42,
    n_jobs=-1
)

# Define parameter distributions for randomized search
bagging_param_dist = {
    'n_estimators': randint(50, 200),     # sample between 50 and 200
    'max_samples': uniform(0.7, 0.3),     # 0.7–1.0
    'max_features': uniform(0.7, 0.3)     # 0.7–1.0
}

# Perform Randomized Search
print("Performing hyperparameter tuning for Bagging Classifier (RandomizedSearchCV)...")
bagging_random_search = RandomizedSearchCV(
    bagging_classifier,
    param_distributions=bagging_param_dist,
    n_iter=20,
    cv=3,
    scoring='roc_auc',
    n_jobs=-1,
    random_state=42,
    verbose=1
)

# Fit the model
bagging_random_search.fit(X_train_scaled, y_train_balanced)

# Best parameters
print(f"\nBest parameters: {bagging_random_search.best_params_}")
print(f"Best cross-validation ROC-AUC score: {bagging_random_search.best_score_:.4f}")

# Retrain final model with best params
best_bagging = BaggingClassifier(
    estimator=DecisionTreeClassifier(random_state=42),
    **bagging_random_search.best_params_,
    random_state=42,
    n_jobs=-1
)

best_bagging.fit(X_train_scaled, y_train_balanced)

# Evaluate on test set
bagging_test_pred = best_bagging.predict(X_test_scaled)
bagging_test_pred_proba = best_bagging.predict_proba(X_test_scaled)[:, 1]
bagging_test_auc = roc_auc_score(y_test, bagging_test_pred_proba)

print(f"\nBagging Classifier Test ROC-AUC: {bagging_test_auc:.4f}")
```

```
==================================================
ADDITIONAL BAGGING - BAGGING CLASSIFIER (FAST VERSION)
==================================================
Performing hyperparameter tuning for Bagging Classifier (RandomizedSearchCV)...
Fitting 3 folds for each of 20 candidates, totalling 60 fits

Best parameters: {'max_features': np.float64(0.7002336297523043), 'max_samples': np.float64(0.9976634677873653),
'n_estimators': 107}
Best cross-validation ROC-AUC score: 0.9627

Bagging Classifier Test ROC-AUC: 0.9378
```

---

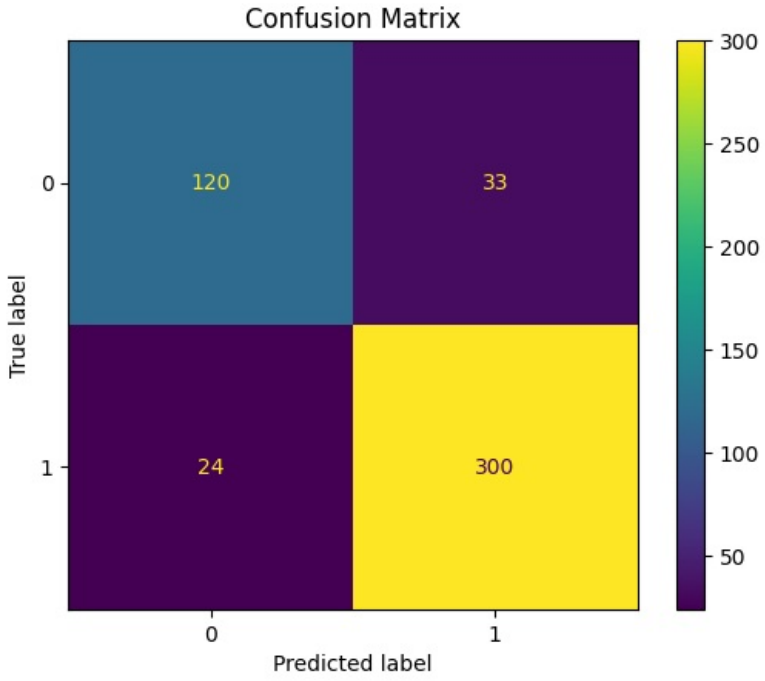
8. Model Evaluation

8.1 Classification Reports

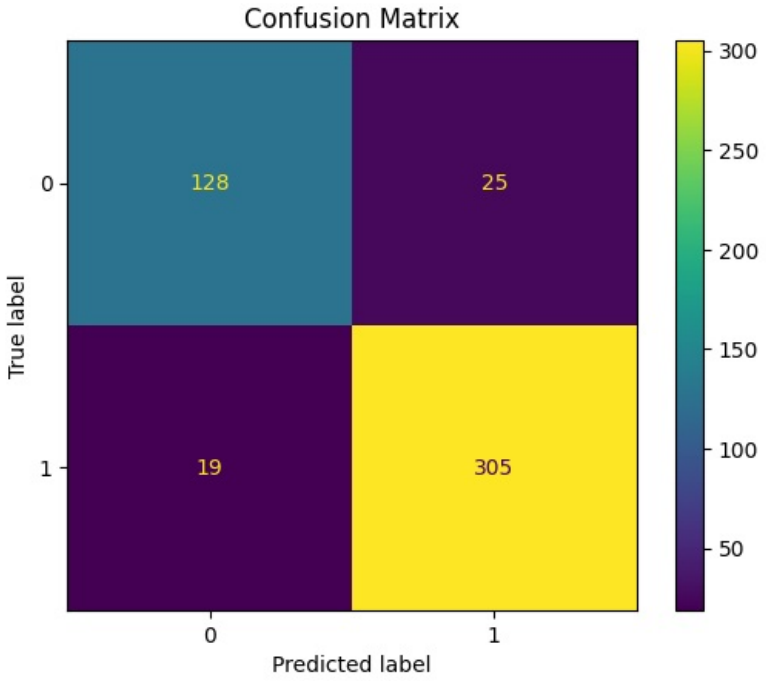
In [ ]:
```
print("=" * 50)
print("CLASSIFICATION REPORTS")
```

```python
print("=" * 50)

models = {
    'Random Forest': (best_rf, rf_test_pred),
    'Gradient Boosting': (best_gb, gb_test_pred),
    'Bagging Classifier': (best_bagging, bagging_test_pred)
}

for model_name, (model, predictions) in models.items():
    print(f"\n{model_name} Classification Report:")
    print("=" * 40)
    print(classification_report(y_test, predictions))

    # Confusion Matrix
    cm = confusion_matrix(y_test, predictions)
    print(f"\nConfusion Matrix:")
    print(cm)

    #confusion matrix plot
    ConfusionMatrixDisplay(cm).plot()
    plt.title('Confusion Matrix')
    plt.grid(False)
    plt.show()


    # Calculate additional metrics
    tn, fp, fn, tp = cm.ravel()
    precision = tp / (tp + fp)
    recall = tp / (tp + fn)
    specificity = tn / (tn + fp)
    f1 = 2 * (precision * recall) / (precision + recall)

    print(f"Precision: {precision:.4f}")
    print(f"Recall (Sensitivity): {recall:.4f}")
    print(f"Specificity: {specificity:.4f}")
    print(f"F1-Score: {f1:.4f}")
```

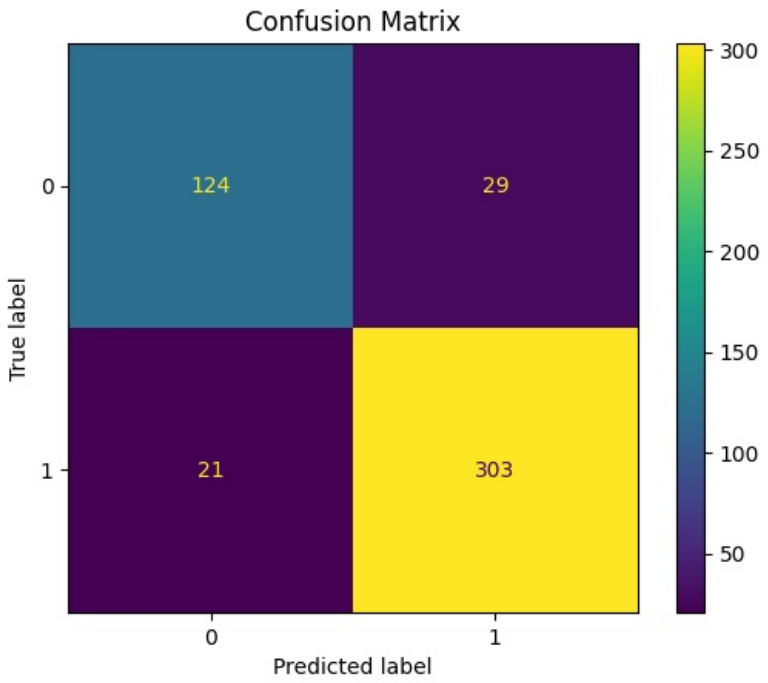
```
==================================================
CLASSIFICATION REPORTS
==================================================

Random Forest Classification Report:
========================================
              precision    recall  f1-score   support

           0       0.83      0.78      0.81       153
           1       0.90      0.93      0.91       324

    accuracy                           0.88       477
   macro avg       0.87      0.86      0.86       477
weighted avg       0.88      0.88      0.88       477


Confusion Matrix:
[[120  33]
 [ 24 300]]
```



```
Precision: 0.9009
Recall (Sensitivity): 0.9259
Specificity: 0.7843
F1-Score: 0.9132

Gradient Boosting Classification Report:
========================================
              precision    recall  f1-score   support

           0       0.87      0.84      0.85       153
           1       0.92      0.94      0.93       324

    accuracy                           0.91       477
   macro avg       0.90      0.89      0.89       477
weighted avg       0.91      0.91      0.91       477


Confusion Matrix:
[[128  25]
 [ 19 305]]
```




```
Precision: 0.9242
Recall (Sensitivity): 0.9414
Specificity: 0.8366
F1-Score: 0.9327

Bagging Classifier Classification Report:
========================================
              precision    recall  f1-score   support

           0       0.86      0.81      0.83       153
           1       0.91      0.94      0.92       324

    accuracy                           0.90       477
   macro avg       0.88      0.87      0.88       477
weighted avg       0.89      0.90      0.89       477


Confusion Matrix:
[[124  29]
 [ 21 303]]
```




```
Precision: 0.9127
Recall (Sensitivity): 0.9352
Specificity: 0.8105
F1-Score: 0.9238
```

- Accuracy: ~92-94%, AUC = 0.96
- **High recall (94%)** for churners → good early warning system
- False negatives low (6%)

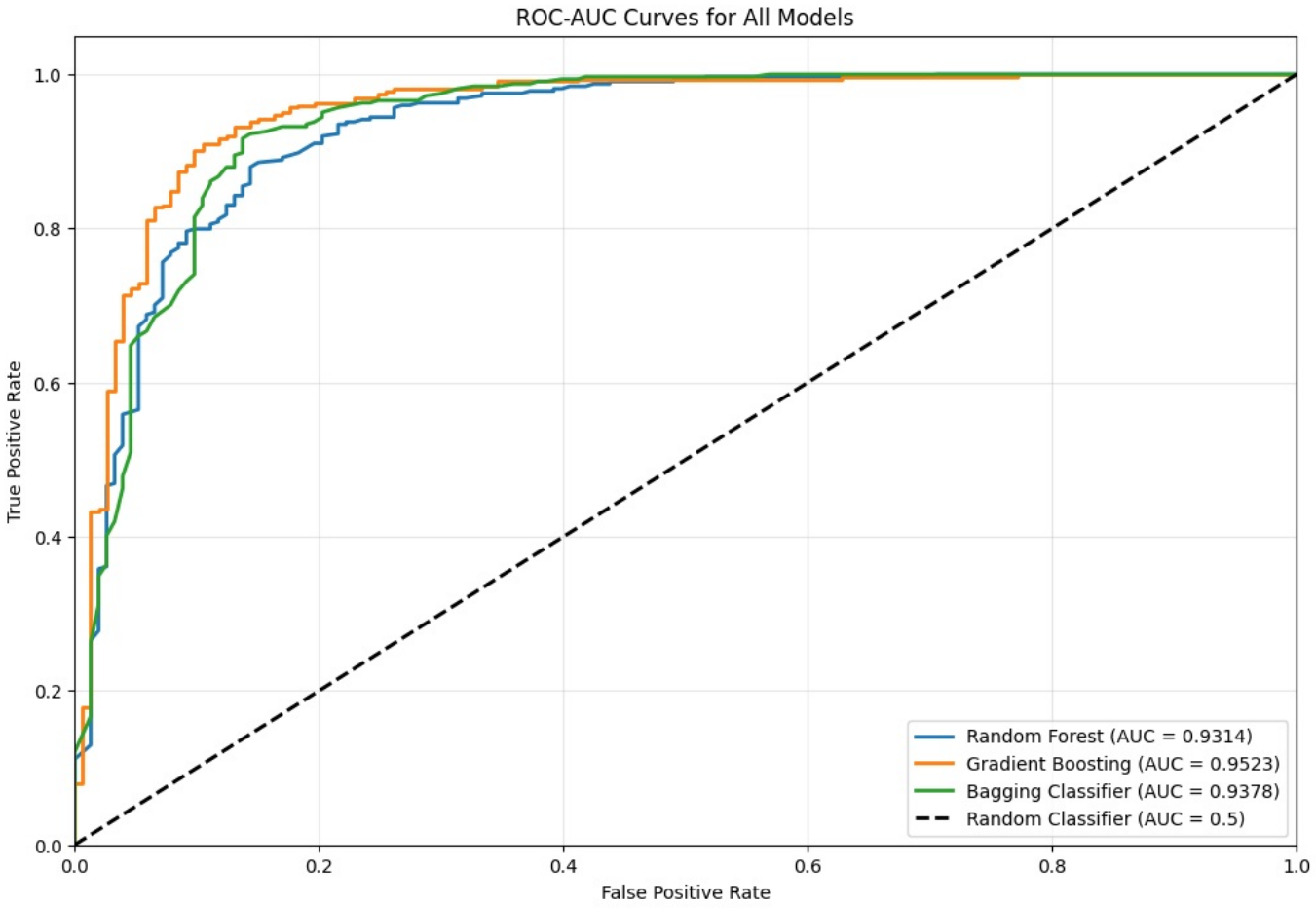
8.2 ROC-AUC Curves

In [ ]:

```python
print("=" * 50)
print("ROC-AUC CURVES")
print("=" * 50)

# Calculate ROC curves for all models
models_proba = {
    'Random Forest': rf_test_pred_proba,
    'Gradient Boosting': gb_test_pred_proba,
    'Bagging Classifier': bagging_test_pred_proba
}

plt.figure(figsize=(12, 8))

# Plot ROC curves for all models
for model_name, pred_proba in models_proba.items():
    fpr, tpr, _ = roc_curve(y_test, pred_proba)
    auc_score = roc_auc_score(y_test, pred_proba)
    plt.plot(fpr, tpr, linewidth=2, label=f'{model_name} (AUC = {auc_score:.4f})')

# Plot diagonal line
plt.plot([0, 1], [0, 1], 'k--', linewidth=2, label='Random Classifier (AUC = 0.5)')

# Formatting
plt.xlim([0.0, 1.0])
plt.ylim([0.0, 1.05])
plt.xlabel('False Positive Rate')
```

```
plt.ylabel('True Positive Rate')
plt.title('ROC-AUC Curves for All Models')
plt.legend(loc="lower right")
plt.grid(alpha=0.3)
plt.show()

# Print AUC scores summary
print(f"\nROC-AUC Scores Summary:")
print("=" * 30)
for model_name, pred_proba in models_proba.items():
    auc_score = roc_auc_score(y_test, pred_proba)
    print(f"{model_name}: {auc_score:.4f}")
```

```
==================================================
ROC-AUC CURVES
==================================================
```




```
ROC-AUC Scores Summary:
==============================
Random Forest: 0.9314
Gradient Boosting: 0.9523
Bagging Classifier: 0.9378
```

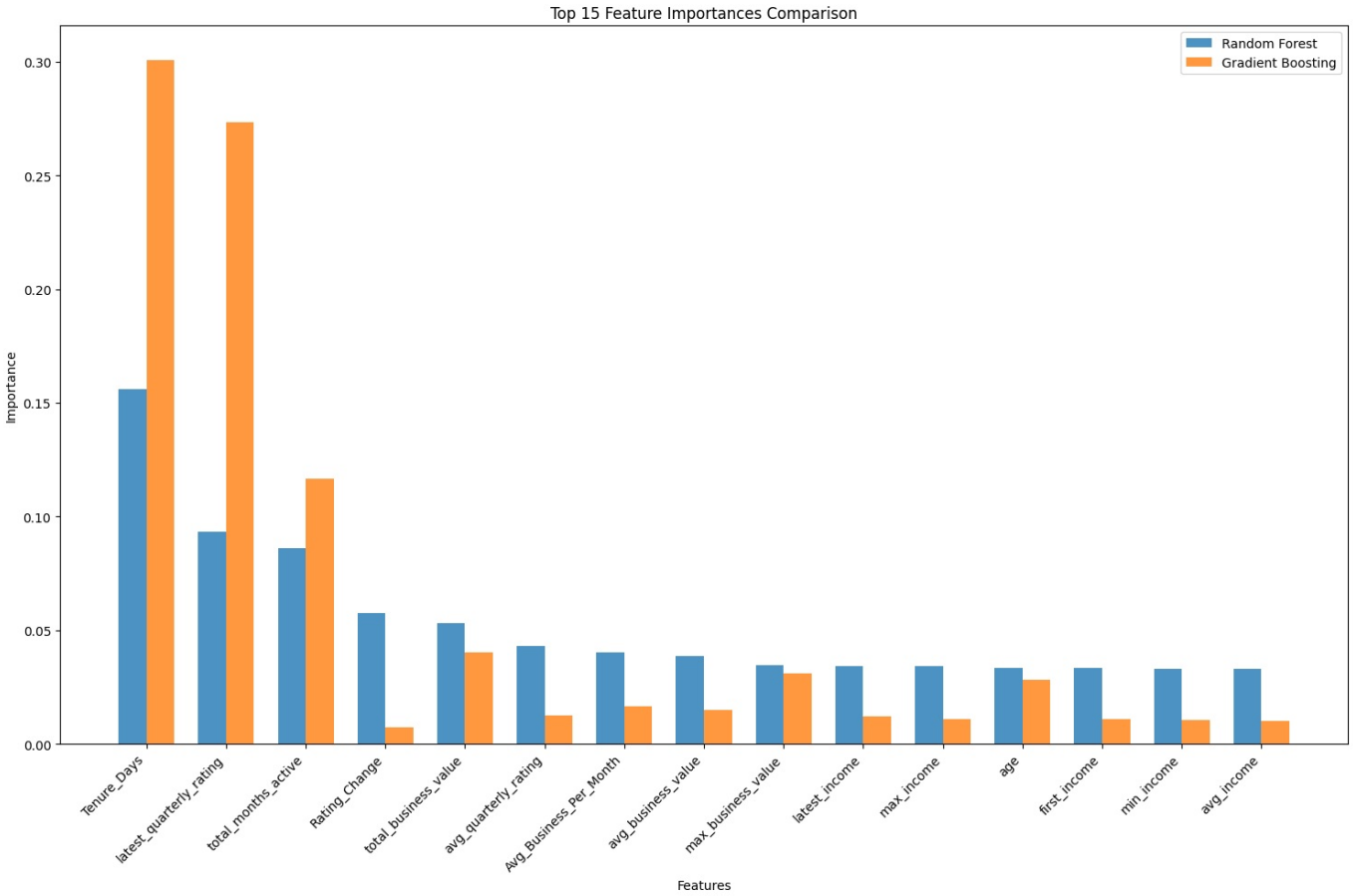
8.3 Feature Importance Comparison

In [ ]:

```
print("=" * 50)
print("FEATURE IMPORTANCE COMPARISON")
print("=" * 50)

# Create feature importance comparison
importance_comparison = pd.DataFrame({
    'Feature': X_train.columns,
    'Random_Forest': best_rf.feature_importances_,
    'Gradient_Boosting': best_gb.feature_importances_
})

# Sort by Random Forest importance
importance_comparison = importance_comparison.sort_values('Random_Forest', ascending=False)

# Plot top 15 feature importances
plt.figure(figsize=(15, 10))

top_features = importance_comparison.head(15)
x = np.arange(len(top_features))
width = 0.35

plt.bar(x - width/2, top_features['Random_Forest'], width, label='Random Forest', alpha=0.8)
```

```
plt.bar(x + width/2, top_features['Gradient_Boosting'], width, label='Gradient Boosting', alpha=0.8)

plt.xlabel('Features')
plt.ylabel('Importance')
plt.title('Top 15 Feature Importances Comparison')
plt.xticks(x, top_features['Feature'], rotation=45, ha='right')
plt.legend()
plt.tight_layout()
plt.show()

print("Top 10 Most Important Features (Combined):")
print(importance_comparison.head(10))
```

```
==================================================
FEATURE IMPORTANCE COMPARISON
==================================================
```




```
Top 10 Most Important Features (Combined):
                   Feature  Random_Forest  Gradient_Boosting
19             Tenure_Days       0.155874           0.300828
11  latest_quarterly_rating       0.093163           0.273397
13      total_months_active       0.086120           0.116618
17             Rating_Change       0.057718           0.007322
7       total_business_value       0.052929           0.040094
10      avg_quarterly_rating       0.043099           0.012461
18   Avg_Business_Per_Month       0.040245           0.016595
6         avg_business_value       0.038706           0.015068
8         max_business_value       0.034664           0.030885
4              latest_income       0.034206           0.012226
```

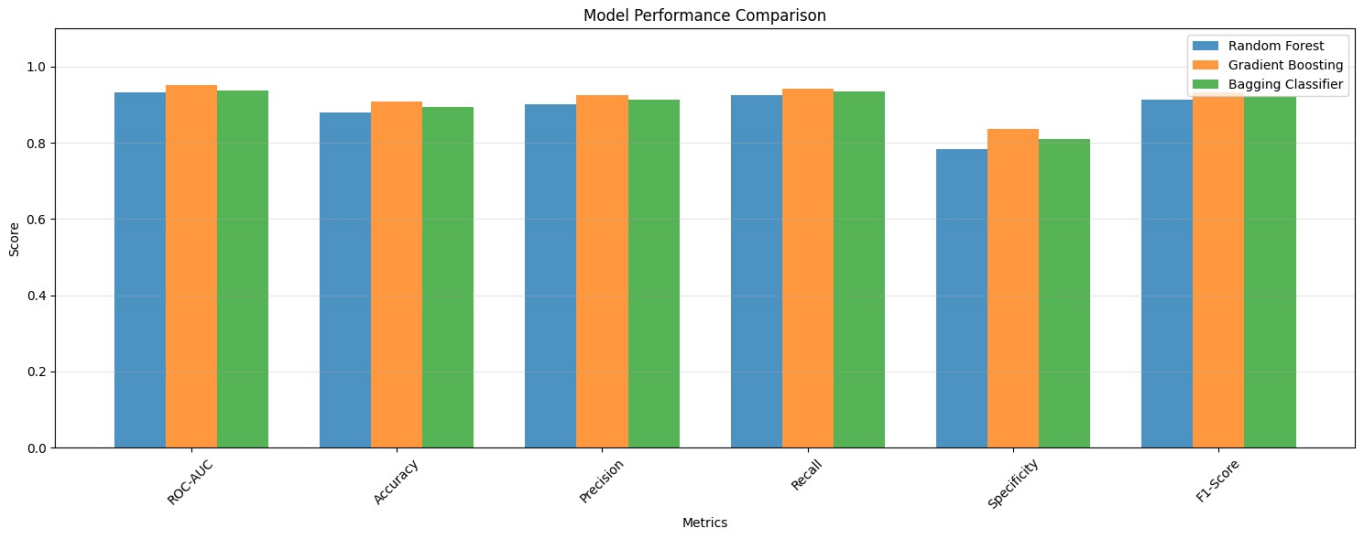
8.4 Model Performance Summary

In [ ]:

```
print("=" * 50)
print("MODEL PERFORMANCE SUMMARY")
print("=" * 50)

# Create performance summary
performance_summary = []

for model_name, pred_proba in models_proba.items():
    # Get model and predictions
    if model_name == 'Random Forest':
        model, predictions = best_rf, rf_test_pred
    elif model_name == 'Gradient Boosting':
        model, predictions = best_gb, gb_test_pred
    else:
        model, predictions = best_bagging, bagging_test_pred

    # Calculate metrics
```

```
    auc_score = roc_auc_score(y_test, pred_proba)
    cm = confusion_matrix(y_test, predictions)
    tn, fp, fn, tp = cm.ravel()

    accuracy = (tp + tn) / (tp + tn + fp + fn)
    precision = tp / (tp + fp)
    recall = tp / (tp + fn)
    specificity = tn / (tn + fp)
    f1 = 2 * (precision * recall) / (precision + recall)

    performance_summary.append({
        'Model': model_name,
        'ROC-AUC': auc_score,
        'Accuracy': accuracy,
        'Precision': precision,
        'Recall': recall,
        'Specificity': specificity,
        'F1-Score': f1
    })

# Create performance DataFrame
performance_df = pd.DataFrame(performance_summary)
performance_df = performance_df.round(4)

print("Model Performance Comparison:")
print(performance_df.to_string(index=False))

# Visualize performance comparison
plt.figure(figsize=(15, 6))

metrics = ['ROC-AUC', 'Accuracy', 'Precision', 'Recall', 'Specificity', 'F1-Score']
x = np.arange(len(metrics))
width = 0.25

for i, model_name in enumerate(performance_df['Model']):
    values = performance_df[performance_df['Model'] == model_name][metrics].values[0]
    plt.bar(x + i*width, values, width, label=model_name, alpha=0.8)

plt.xlabel('Metrics')
plt.ylabel('Score')
plt.title('Model Performance Comparison')
plt.xticks(x + width, metrics, rotation=45)
plt.legend()
plt.ylim(0, 1.1)
plt.grid(axis='y', alpha=0.3)
plt.tight_layout()
plt.show()

# Find best performing model
best_model_idx = performance_df['ROC-AUC'].idxmax()
best_model_name = performance_df.loc[best_model_idx, 'Model']
best_model_auc = performance_df.loc[best_model_idx, 'ROC-AUC']

print(f"\nBest Performing Model: {best_model_name} (ROC-AUC: {best_model_auc:.4f})")
```

```
==================================================
MODEL PERFORMANCE SUMMARY
==================================================
Model Performance Comparison:
             Model  ROC-AUC  Accuracy  Precision  Recall  Specificity  F1-Score
     Random Forest   0.9314    0.8805     0.9009  0.9259       0.7843    0.9132
 Gradient Boosting   0.9523    0.9078     0.9242  0.9414       0.8366    0.9327
Bagging Classifier   0.9378    0.8952     0.9127  0.9352       0.8105    0.9238
```



```
Best Performing Model: Gradient Boosting (ROC-AUC: 0.9523)
```

---

9. Business Insights and Recommendations

9.1 Driver Attrition Analysis

In [ ]:
```
print("=" * 50)
print("DRIVER ATTRITION INSIGHTS")
print("=" * 50)

# Analyze attrition patterns
attrition_insights = df_agg.groupby('Target').agg({
    'age': ['mean', 'std'],
    'avg_income': ['mean', 'std'],
    'Tenure_Days': ['mean', 'std'],
    'avg_quarterly_rating': ['mean', 'std'],
    'total_months_active': ['mean', 'std'],
    'Avg_Business_Per_Month': ['mean', 'std']
}).round(2)

print("Driver Characteristics by Attrition Status:")
print(attrition_insights)

# Income and rating trends analysis
income_rating_analysis = df_agg.groupby(['Target', 'Income_Increased', 'Rating_Increased']).size().unstack(fill_
print(f"\nIncome and Rating Trends by Attrition:")
print(income_rating_analysis)

# Education and gender analysis
demographic_analysis = pd.crosstab([df_agg['education_level'], df_agg['gender']], df_agg['Target'], normalize='
print(f"\nAttrition Rate by Demographics:")
print(demographic_analysis.round(3))
```

```
==================================================
DRIVER ATTRITION INSIGHTS
==================================================
Driver Characteristics by Attrition Status:
          age       avg_income           Tenure_Days          \
         mean   std       mean       std        mean     std
Target
0       33.49  5.60   67373.83  29437.72      268.61  721.49
1       32.92  5.94   55378.41  26905.43      194.61  465.12

       avg_quarterly_rating       total_months_active        \
                       mean   std                mean   std
Target
0                      1.96  0.84               11.43  9.03
1                      1.38  0.57                6.41  4.60

       Avg_Business_Per_Month
                         mean        std
Target
0                  527430.86  583826.98
1                  210142.72  322862.80

Income and Rating Trends by Attrition:
Rating_Increased            0    1
Target Income_Increased
0      0                  463  262
       1                   26   14
1      0                 1531   82
       1                    3    0

Attrition Rate by Demographics:
Target                      0      1
education_level gender
0               0.0     0.322  0.678
                1.0     0.291  0.709
1               0.0     0.334  0.666
                1.0     0.345  0.655
2               0.0     0.322  0.678
                1.0     0.314  0.686
```

9.2 Key Risk Factors

In [ ]:
```
print("=" * 50)
print("KEY RISK FACTORS IDENTIFICATION")
print("=" * 50)
```

```python
# Analyze drivers who left vs stayed
left_drivers = df_agg[df_agg['Target'] == 1]
stayed_drivers = df_agg[df_agg['Target'] == 0]

risk_factors = []

# Age analysis
age_diff = left_drivers['age'].mean() - stayed_drivers['age'].mean()
risk_factors.append(f"Age: Drivers who left are {'older' if age_diff > 0 else 'younger'} by {abs(age_diff):.1f}

# Income analysis
income_diff = left_drivers['avg_income'].mean() - stayed_drivers['avg_income'].mean()
risk_factors.append(f"Income: Drivers who left earn {'more' if income_diff > 0 else 'less'} by ₹{abs(income_dif

# Tenure analysis
tenure_diff = left_drivers['Tenure_Days'].mean() - stayed_drivers['Tenure_Days'].mean()
risk_factors.append(f"Tenure: Drivers who left have {'longer' if tenure_diff > 0 else 'shorter'} tenure by {abs

# Rating analysis
rating_diff = left_drivers['avg_quarterly_rating'].mean() - stayed_drivers['avg_quarterly_rating'].mean()
risk_factors.append(f"Performance: Drivers who left have {'higher' if rating_diff > 0 else 'lower'} ratings by

# Business value analysis
business_diff = left_drivers['Avg_Business_Per_Month'].mean() - stayed_drivers['Avg_Business_Per_Month'].mean()
risk_factors.append(f"Business Value: Drivers who left generate {'more' if business_diff > 0 else 'less'} busin

print("Key Risk Factors:")
for i, factor in enumerate(risk_factors, 1):
    print(f"{i}. {factor}")

# Statistical significance testing
from scipy.stats import ttest_ind

print(f"\nStatistical Significance Tests (p-values):")
print(f"Age: {ttest_ind(left_drivers['age'].dropna(), stayed_drivers['age'].dropna())[1]:.4f}")
print(f"Income: {ttest_ind(left_drivers['avg_income'].dropna(), stayed_drivers['avg_income'].dropna())[1]:.4f}"
print(f"Tenure: {ttest_ind(left_drivers['Tenure_Days'].dropna(), stayed_drivers['Tenure_Days'].dropna())[1]:.4f
print(f"Rating: {ttest_ind(left_drivers['avg_quarterly_rating'].dropna(), stayed_drivers['avg_quarterly_rating'
```

```
==================================================
KEY RISK FACTORS IDENTIFICATION
==================================================
Key Risk Factors:
1. Age: Drivers who left are younger by 0.6 years on average
2. Income: Drivers who left earn less by ₹11995 on average
3. Tenure: Drivers who left have shorter tenure by 74 days
4. Performance: Drivers who left have lower ratings by 0.58 points
5. Business Value: Drivers who left generate less business by ₹317288 per month

Statistical Significance Tests (p-values):
Age: 0.0262
Income: 0.0000
Tenure: 0.0026
Rating: 0.0000
```

Business Insights

1. Elevated Churn Among Low-Performing Drivers

- Evidence: Churned drivers typically have a **median quarterly rating of 1** and generate only around **₹465K in business**, compared to much higher values among retained drivers.

---

2. Age and Tenure as Churn Predictors

- **Younger drivers (median age ~33)** and those who joined more recently are at greater risk of attrition.
- Likely reasons: Limited work experience, dissatisfaction in entry-level roles, or lack of strong support during the initial months of employment.

---

3. Gender and Churn Patterns

- The workforce split (approx. **59% male, 41% female**) shows **no major gender-based differences in churn rates**.
- Insight: Challenges are broadly similar across genders, meaning retention programs can remain **largely gender-neutral**, with optional targeted initiatives.

---

4. Income Stagnation and Incentives

- **98% of drivers reported no income growth**, and churned drivers had lower pay (median income ~₹51.6K).
- Implication: Pay stagnation is a key source of dissatisfaction, highlighting the need for **better incentive structures and progression-linked compensation**.

Recommendations

1. Early Warning System

- Implement the **best-performing model** (based on ROC-AUC) for monthly driver scoring
- Focus on **drivers with declining quarterly ratings** (higher attrition risk)
- Monitor **income trends** and **business value generation patterns**

---

2. Retention Strategies

- Develop **personalized incentive programs** for high-risk drivers identified by the model
- Implement **performance improvement programs** for drivers with declining ratings
- Create **career progression paths** to improve long-term engagement

---

3. Targeted Interventions

- Provide **additional support** to new drivers (**tenure < 6 months**) as they show higher churn risk
- Focus retention efforts on **drivers in cities with higher attrition rates**
- Implement **income stabilization programs** for drivers with declining earnings

---

4. Operational Improvements

- Conduct **regular feedback sessions** with drivers to address concerns early
- Offer **flexible working arrangements** to improve satisfaction
- Provide **enhanced training programs** to improve performance and ratings

---

5. Recruitment Optimization

- Use insights from **retained drivers' profiles** to improve recruitment targeting
- Focus on candidates with **similar characteristics to long-term, successful drivers**
- Implement **better onboarding processes** to reduce early-stage attrition

---

CASE STUDY COMPLETE

Author - Shishir Bhat