```python
#Import necessary libraries
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn.model_selection import train_test_split, GridSearchCV
from sklearn.impute import KNNImputer
from sklearn.preprocessing import StandardScaler, OneHotEncoder
from sklearn.ensemble import RandomForestClassifier, GradientBoostingClassifier
from sklearn.metrics import classification_report, roc_auc_score, roc_curve, confusion_matrix
from imblearn.over_sampling import SMOTE # To handle class imbalance
import warnings

warnings.filterwarnings('ignore') # Ignore warnings for cleaner output
# Set plot style
sns.set(style="whitegrid")
```

```python
# --- Load Data ---
try:
    df = pd.read_csv('ola_driver_scaler.csv')
    print("Dataset loaded successfully.")
    # Drop the 'Unnamed: 0' column if it exists
    if 'Unnamed: 0' in df.columns:
        df = df.drop(columns=['Unnamed: 0'])
        print("Dropped 'Unnamed: 0' column.")
except FileNotFoundError:
    print("Error: ola_driver_scaler.csv not found in the current directory.")
    exit() # Exit if the file is not found
```

```
Dataset loaded successfully.
Dropped 'Unnamed: 0' column.
```

```python
# --- Initial EDA ---
print("\n--- Initial Exploratory Data Analysis ---")

# Display first 5 rows
print("\nFirst 5 rows of the dataset:")
print(df.head())

# Display shape
print(f"\nDataset shape: {df.shape}")

# Display data types and non-null counts
print("\nDataset info:")
df.info()

# Display statistical summary for numerical features
print("\nStatistical summary (numerical features):")
print(df.describe())

# Display statistical summary for object/categorical features (like MMMM-YY, City)
print("\nStatistical summary (categorical features):")
print(df.describe(include=['object']))
```

```
--- Initial Exploratory Data Analysis ---

First 5 rows of the dataset:
     MMM-YY  Driver_ID   Age  Gender City  Education_Level  Income  \
0  01/01/19          1  28.0     0.0  C23                2   57387
1  02/01/19          1  28.0     0.0  C23                2   57387
2  03/01/19          1  28.0     0.0  C23                2   57387
3  11/01/20          2  31.0     0.0   C7                2   67016
4  12/01/20          2  31.0     0.0   C7                2   67016

  Dateofjoining LastWorkingDate  Joining Designation  Grade  \
0      24/12/18             NaN                    1      1
1      24/12/18             NaN                    1      1
2      24/12/18        03/11/19                    1      1
3      11/06/20             NaN                    2      2
4      11/06/20             NaN                    2      2

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

Dataset shape: (19104, 13)
```

```
Dataset info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 19104 entries, 0 to 19103
Data columns (total 13 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   MMM-YY                19104 non-null  object
 1   Driver_ID             19104 non-null  int64
 2   Age                   19043 non-null  float64
 3   Gender                19052 non-null  float64
 4   City                  19104 non-null  object
 5   Education_Level       19104 non-null  int64
 6   Income                19104 non-null  int64
 7   Dateofjoining         19104 non-null  object
 8   LastWorkingDate       1616 non-null   object
 9   Joining Designation   19104 non-null  int64
 10  Grade                 19104 non-null  int64
 11  Total Business Value  19104 non-null  int64
 12  Quarterly Rating      19104 non-null  int64
dtypes: float64(2), int64(7), object(4)
memory usage: 1.9+ MB

Statistical summary (numerical features):
          Driver_ID           Age        Gender  Education_Level  \
count  19104.000000  19043.000000  19052.000000     19104.000000
mean    1415.591133     34.668435      0.418749         1.021671
std      810.705321      6.257912      0.493367         0.800167
min        1.000000     21.000000      0.000000         0.000000
25%      710.000000     30.000000      0.000000         0.000000
50%     1417.000000     34.000000      0.000000         1.000000
75%     2137.000000     39.000000      1.000000         2.000000
```

```python
# --- Data Cleaning & Preprocessing ---
print("\n--- Data Cleaning & Preprocessing ---")

# Convert date columns to datetime objects
print("\nConverting date columns...")
# Corrected column name 'MMM-YY'
try:
    # Try parsing with day first (e.g., 01/01/19)
    df['MMM-YY'] = pd.to_datetime(df['MMM-YY'], format='%d/%m/%y', errors='coerce')
except Exception as e1:
    print(f"Initial date parsing failed: {e1}. Trying alternative formats.")
    try:
        # Fallback format if needed (e.g., Jan-19) - adjust based on actual data if first fails
        df['MMM-YY'] = pd.to_datetime(df['MMM-YY'], format='%b-%y', errors='coerce')
    except Exception as e2:
         print(f"Error converting 'MMM-YY': {e2}. Please check the date format.")
         # Consider exiting or handling this case based on requirements

# Corrected column name 'Dateofjoining' and added dayfirst=False for common formats like MM/DD/YY
df['Dateofjoining'] = pd.to_datetime(df['Dateofjoining'], errors='coerce', dayfirst=False)
df['LastWorkingDate'] = pd.to_datetime(df['LastWorkingDate'], errors='coerce', dayfirst=False)
print("Date columns converted (attempted).")
print("\nData types after date conversion:")
df.info() # Display info again to show converted types

# Check for missing values
print("\nMissing values per column:")
print(df.isnull().sum())
```

```
--- Data Cleaning & Preprocessing ---

Converting date columns...
Date columns converted (attempted).

Data types after date conversion:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 19104 entries, 0 to 19103
Data columns (total 13 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   MMM-YY                19104 non-null  datetime64[ns]
 1   Driver_ID             19104 non-null  int64
 2   Age                   19043 non-null  float64
 3   Gender                19052 non-null  float64
 4   City                  19104 non-null  object
 5   Education_Level       19104 non-null  int64
 6   Income                19104 non-null  int64
 7   Dateofjoining         19104 non-null  datetime64[ns]
 8   LastWorkingDate       1616 non-null   datetime64[ns]
 9   Joining Designation   19104 non-null  int64
 10  Grade                 19104 non-null  int64
 11  Total Business Value  19104 non-null  int64
 12  Quarterly Rating      19104 non-null  int64
```

```
dtypes: datetime64[ns](3), float64(2), int64(7), object(1)
memory usage: 1.9+ MB

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

```
# --- KNN Imputation ---
print("\n--- KNN Imputation for Missing Numerical Values ---")

# Identify numerical columns for imputation (excluding Driver_ID and potentially target-related later)
# Based on typical datasets and the info() output, these are likely candidates.
# We'll refine this list based on the actual isnull().sum() output when the script runs.
numerical_cols_for_imputation = ['Age', 'Income', 'Total Business Value', 'Quarterly Rating']

# Filter out columns that might not exist or have no missing values to avoid errors
numerical_cols_to_impute = [col for col in numerical_cols_for_imputation if col in df.columns and df[col].isnull().any()]

if not numerical_cols_to_impute:
    print("No missing values found in the selected numerical columns for imputation.")
else:
    print(f"Performing KNN Imputation on: {numerical_cols_to_impute}")
    # Select only the numerical columns for the imputer
    imputer_data = df[numerical_cols_to_impute]

    # Initialize KNNImputer (using default n_neighbors=5)
    knn_imputer = KNNImputer(n_neighbors=5)

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

    # Convert the imputed data back to a DataFrame with original column names
    imputed_df = pd.DataFrame(imputed_data, columns=numerical_cols_to_impute, index=df.index)

    # Update the original DataFrame with imputed values
    df.update(imputed_df)

    print("KNN Imputation completed.")
    print("\nMissing values after KNN Imputation:")
    print(df[numerical_cols_to_impute].isnull().sum()) # Verify imputation
```

```
--- KNN Imputation for Missing Numerical Values ---
Performing KNN Imputation on: ['Age']
KNN Imputation completed.

Missing values after KNN Imputation:
Age    0
dtype: int64
```

```
# --- Data Aggregation ---
print("\n--- Aggregating Data by Driver_ID ---")

# Sort by Driver_ID and reporting date to ensure 'last' picks the latest record
# Corrected column name 'MMM-YY'
df = df.sort_values(by=['Driver_ID', 'MMM-YY'])

# Define aggregation dictionary
agg_dict = {
    # Static features: take the last known value
    'Age': 'last',
    'Gender': 'last',
    'City': 'last',
    'Education_Level': 'last',
    # Corrected column name 'Dateofjoining'
    'Dateofjoining': 'last',
    'Joining Designation': 'last',
    # Time-varying features:
    'Income': ['mean', 'last'], # Keep mean income and last recorded income
    'Grade': 'last', # Last known grade
    'Total Business Value': ['mean', 'last'], # Keep mean and last business value
```

```
    'Quarterly Rating': ['first', 'last'], # Keep first and last rating for comparison later
    # Date features:
    # Corrected column name 'MMM-YY'
    'MMM-YY': 'max', # Last reporting month
    'LastWorkingDate': 'max' # Last working date (will be NaT if still working)
}

# Perform aggregation
df_agg = df.groupby('Driver_ID').agg(agg_dict)

# Flatten MultiIndex columns (e.g., ('Income', 'mean') becomes 'Income_mean')
df_agg.columns = ['_'.join(col).strip('_') for col in df_agg.columns.values]

# Reset index to bring Driver_ID back as a column
df_agg = df_agg.reset_index()

# Rename columns by removing '_last' suffix for clarity
df_agg.columns = [col.replace('_last', '') if col.endswith('_last') else col for col in df_agg.columns]
# Also remove '_max' suffix from date columns used for tenure/target
df_agg.columns = [col.replace('_max', '') if col.endswith('_max') else col for col in df_agg.columns]
# Rename Quarterly Rating_first if it exists (used for rating_increase)
if 'Quarterly Rating_first' in df_agg.columns:
     df_agg = df_agg.rename(columns={'Quarterly Rating_first': 'Quarterly_Rating_first'})


print("Data aggregated successfully.")
print(f"Aggregated dataset shape: {df_agg.shape}")
print("\nFirst 5 rows of aggregated data:")
print(df_agg.head())
print("\nAggregated data info:")
df_agg.info()
print("\nMissing values in aggregated data:")
print(df_agg.isnull().sum())
```

```
--- Aggregating Data by Driver_ID ---
Data aggregated successfully.
Aggregated dataset shape: (2381, 16)

First 5 rows of aggregated data:
   Driver_ID   Age  Gender City  Education_Level Dateofjoining  \
0          1  28.0     0.0  C23                2    2018-12-24
1          2  31.0     0.0   C7                2    2020-11-06
2          4  43.0     0.0  C13                2    2019-12-07
3          5  29.0     0.0   C9                0    2019-01-09
4          6  31.0     1.0  C11                1    2020-07-31

   Joining Designation  Income_mean  Income  Grade  Total Business Value_mean  \
0                    1      57387.0   57387      1                   571860.0
1                    2      67016.0   67016      2                        0.0
2                    2      65603.0   65603      2                    70000.0
3                    1      46368.0   46368      1                    40120.0
4                    3      78728.0   78728      3                   253000.0

   Total Business Value  Quarterly_Rating_first  Quarterly Rating     MMM-YY  \
0                     0                       2                 2 2019-01-03
1                     0                       1                 1 2020-01-12
2                     0                       1                 1 2020-01-04
3                     0                       1                 1 2019-01-03
4                     0                       1                 2 2020-01-12

  LastWorkingDate
0      2019-03-11
1             NaT
2      2020-04-27
3      2019-03-07
4             NaT

Aggregated data info:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2381 entries, 0 to 2380
Data columns (total 16 columns):
 #   Column                     Non-Null Count  Dtype
---  ------                     --------------  -----
 0   Driver_ID                  2381 non-null   int64
 1   Age                        2381 non-null   float64
 2   Gender                     2381 non-null   float64
 3   City                       2381 non-null   object
 4   Education_Level            2381 non-null   int64
 5   Dateofjoining              2381 non-null   datetime64[ns]
 6   Joining Designation        2381 non-null   int64
 7   Income_mean                2381 non-null   float64
 8   Income                     2381 non-null   int64
 9   Grade                      2381 non-null   int64
 10  Total Business Value_mean  2381 non-null   float64
 11  Total Business Value       2381 non-null   int64
```

```
 12  Quarterly_Rating_first     2381 non-null   int64
 13  Quarterly Rating           2381 non-null   int64
 14  MMM-YY                     2381 non-null   datetime64[ns]
 15  LastWorkingDate            1616 non-null   datetime64[ns]
dtypes: datetime64[ns](3), float64(4), int64(8), object(1)
memory usage: 297.8+ KB
```

```
# --- Feature Engineering ---
print("\n--- Feature Engineering ---")

# 1. Target Variable: 1 if driver left, 0 otherwise
# Uses the renamed 'LastWorkingDate' column
df_agg['target'] = df_agg['LastWorkingDate'].notna().astype(int)
print(f"\nTarget variable 'target' created. Distribution:\n{df_agg['target'].value_counts(normalize=True)}")

# 2. Quarterly Rating Increase: 1 if last rating > first rating
# Ensure both columns exist before creating the feature (using renamed columns)
if 'Quarterly_Rating_first' in df_agg.columns and 'Quarterly_Rating' in df_agg.columns:
    df_agg['rating_increase'] = (df_agg['Quarterly_Rating'] > df_agg['Quarterly_Rating_first']).astype(int)
    print("\nFeature 'rating_increase' created.")
    # Drop the original first rating column as it's now captured in rating_increase
    df_agg = df_agg.drop(columns=['Quarterly_Rating_first'])
else:
    print("\nWarning: 'Quarterly_Rating_first' or 'Quarterly_Rating' not found after aggregation/renaming. Skipping 'rating_increase' fe


# 3. Monthly Income Increase (Proxy): 1 if last income > mean income
# Ensure both columns exist
if 'Income_mean' in df_agg.columns and 'Income_last' in df_agg.columns:
    df_agg['income_increase_over_mean'] = (df_agg['Income_last'] > df_agg['Income_mean']).astype(int)
    print("Feature 'income_increase_over_mean' created.")
    # Decide whether to keep Income_mean and Income_last or just one. Let's keep both for now.
else:
     print("\nWarning: 'Income_mean' or 'Income_last' not found. Skipping 'income_increase_over_mean' feature.")
```

```
--- Feature Engineering ---

Target variable 'target' created. Distribution:
target
1    0.678706
0    0.321294
Name: proportion, dtype: float64

Warning: 'Quarterly_Rating_first' or 'Quarterly_Rating' not found after aggregation/renaming. Skipping 'rating_increase' feature.

Warning: 'Income_mean' or 'Income_last' not found. Skipping 'income_increase_over_mean' feature.
```

```
# 4. Tenure: Calculate tenure in days
# Ensure required date columns exist (using renamed columns)
if 'Dateofjoining' in df_agg.columns and 'LastWorkingDate' in df_agg.columns and 'MMM-YY' in df_agg.columns:
    # For drivers who left
    left_mask = df_agg['target'] == 1
    df_agg.loc[left_mask, 'tenure_days'] = (df_agg['LastWorkingDate'] - df_agg['Dateofjoining']).dt.days

    # For drivers still working (use last reporting date)
    working_mask = df_agg['target'] == 0
    df_agg.loc[working_mask, 'tenure_days'] = (df_agg['MMM-YY'] - df_agg['Dateofjoining']).dt.days

    # Handle potential negative tenure if dates are inconsistent (e.g., joining date after last working date)
    df_agg['tenure_days'] = df_agg['tenure_days'].apply(lambda x: max(x, 0) if pd.notna(x) else 0)
    # Handle potential negative tenure if dates are inconsistent
    df_agg['tenure_days'] = df_agg['tenure_days'].apply(lambda x: max(x, 0) if pd.notna(x) else 0)
    # Fill any remaining NaNs in tenure_days (e.g., if Dateofjoining was NaT) with 0
    df_agg['tenure_days'] = df_agg['tenure_days'].fillna(0)
    print("Feature 'tenure_days' created.")

    # Drop original date columns used for tenure calculation (using renamed columns)
    df_agg = df_agg.drop(columns=['Dateofjoining', 'LastWorkingDate', 'MMM-YY'])
else:
    print("\nWarning: Required date columns ('Dateofjoining', 'LastWorkingDate', 'MMM-YY') for tenure calculation not found after aggre


print("\nData after Feature Engineering:")
print(df_agg.head())
print("\nInfo after Feature Engineering:")
df_agg.info()
```

```
Feature 'tenure_days' created.

Data after Feature Engineering:
   Driver_ID   Age  Gender City  Education_Level  Joining Designation  \
0          1  28.0     0.0  C23                2                    1
1          2  31.0     0.0   C7                2                    2
```

```
2         4  43.0     0.0  C13                2                    2
3         5  29.0     0.0   C9                0                    1
4         6  31.0     1.0  C11                1                    3

   Income_mean  Income  Grade  Total Business Value_mean  \
0      57387.0   57387      1                   571860.0
1      67016.0   67016      2                        0.0
2      65603.0   65603      2                    70000.0
3      46368.0   46368      1                    40120.0
4      78728.0   78728      3                   253000.0

   Total Business Value  Quarterly_Rating_first  Quarterly Rating  target  \
0                     0                       2                 2       1
1                     0                       1                 1       0
2                     0                       1                 1       1
3                     0                       1                 1       1
4                     0                       1                 2       0

   tenure_days
0         77.0
1          0.0
2        142.0
3         57.0
4          0.0

Info after Feature Engineering:
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2381 entries, 0 to 2380
Data columns (total 15 columns):
 #   Column                     Non-Null Count  Dtype
---  ------                     --------------  -----
 0   Driver_ID                  2381 non-null   int64
 1   Age                        2381 non-null   float64
 2   Gender                     2381 non-null   float64
 3   City                       2381 non-null   object
 4   Education_Level            2381 non-null   int64
 5   Joining Designation        2381 non-null   int64
 6   Income_mean                2381 non-null   float64
 7   Income                     2381 non-null   int64
 8   Grade                      2381 non-null   int64
 9   Total Business Value_mean  2381 non-null   float64
 10  Total Business Value       2381 non-null   int64
 11  Quarterly_Rating_first     2381 non-null   int64
 12  Quarterly Rating           2381 non-null   int64
 13  target                     2381 non-null   int64
 14  tenure_days                2381 non-null   float64
dtypes: float64(5), int64(9), object(1)
memory usage: 279.2+ KB
```

```
# --- Further EDA on Aggregated Data ---
print("\n--- Further EDA on Aggregated Data ---")

# Statistical summary of the final aggregated dataset
print("\nStatistical summary of aggregated data:")
# Include 'all' to get summary for both numerical and categorical (if any remain as object)
print(df_agg.describe(include='all'))

# Correlation Analysis
print("\nCorrelation matrix:")
# Select only numerical columns for correlation calculation
numerical_cols = df_agg.select_dtypes(include=np.number).columns
# Exclude Driver_ID from correlation matrix if it's numerical
if 'Driver_ID' in numerical_cols:
    numerical_cols = numerical_cols.drop('Driver_ID')

correlation_matrix = df_agg[numerical_cols].corr()
print(correlation_matrix)

# Visualize correlation matrix and save to file
plt.figure(figsize=(15, 12)) # Increased size for more features
sns.heatmap(correlation_matrix, annot=False, cmap='coolwarm') # Annot=False if too cluttered
plt.title('Correlation Matrix of Numerical Features')
plt.tight_layout()
plt.savefig('correlation_heatmap.png')
print("\nCorrelation heatmap saved to correlation_heatmap.png")
plt.close() # Close the plot to free memory

print("\nTarget variable distribution:")
print(df_agg['target'].value_counts())
print(df_agg['target'].value_counts(normalize=True))
```

```
--- Further EDA on Aggregated Data ---

Statistical summary of aggregated data:
         Driver_ID          Age       Gender  City  Education_Level  \
```

```
count   2381.000000  2381.000000  2381.000000  2381       2381.00000
unique          NaN          NaN          NaN    29              NaN
top             NaN          NaN          NaN   C20              NaN
freq            NaN          NaN          NaN   152              NaN
mean    1397.559009    33.686551     0.410332   NaN          1.00756
std      806.161628     5.968622     0.491997   NaN          0.81629
min        1.000000    21.000000     0.000000   NaN          0.00000
25%      695.000000    29.000000     0.000000   NaN          0.00000
50%     1400.000000    33.000000     0.000000   NaN          1.00000
75%     2100.000000    37.000000     1.000000   NaN          2.00000
max     2788.000000    58.000000     1.000000   NaN          2.00000

        Joining Designation    Income_mean         Income        Grade  \
count           2381.000000    2381.000000    2381.000000  2381.000000
unique                  NaN            NaN            NaN          NaN
top                     NaN            NaN            NaN          NaN
freq                    NaN            NaN            NaN          NaN
mean               1.820244   59232.460484   59334.157077     2.096598
std                0.841433   28298.214012   28383.666384     0.941522
min                1.000000   10747.000000   10747.000000     1.000000
25%                1.000000   39104.000000   39104.000000     1.000000
50%                2.000000   55285.000000   55315.000000     2.000000
75%                2.000000   75835.000000   75986.000000     3.000000
max                5.000000  188418.000000  188418.000000     5.000000

        Total Business Value_mean  Total Business Value  \
count                2.381000e+03          2.381000e+03
unique                        NaN                   NaN
top                           NaN                   NaN
freq                          NaN                   NaN
mean                 3.120854e+05          2.667694e+05
std                  4.495705e+05          1.134681e+06
min                 -1.979329e+05         -9.900000e+05
25%                  0.000000e+00          0.000000e+00
50%                  1.506244e+05          0.000000e+00
75%                  4.294988e+05          1.969200e+05
max                  3.972128e+06          3.374772e+07

        Quarterly_Rating_first  Quarterly Rating       target  tenure_days
count              2381.000000       2381.000000  2381.000000  2381.000000
unique                     NaN               NaN          NaN          NaN
top                        NaN               NaN          NaN          NaN
freq                       NaN               NaN          NaN          NaN
mean                  1.486350          1.427971     0.678706   363.968501
std                   0.834348          0.809839     0.467071   521.767726
min                   1.000000          1.000000     0.000000     0.000000
25%                   1.000000          1.000000     0.000000    52.000000
50%                   1.000000          1.000000     1.000000   147.000000
75%                   2.000000          2.000000     1.000000   419.000000
max                   4.000000          4.000000     1.000000  2582.000000

Correlation matrix:
                              Age    Gender  Education_Level  \
```

```python
# --- Encoding Categorical Variables ---
print("\n--- Encoding Categorical Variables ---")

# Identify categorical columns (object or category dtype)
categorical_cols = df_agg.select_dtypes(include=['object', 'category']).columns

# Exclude Driver_ID if it was read as object, though it should be numerical
if 'Driver_ID' in categorical_cols:
    categorical_cols = categorical_cols.drop('Driver_ID')

if len(categorical_cols) > 0:
    print(f"Applying One-Hot Encoding to: {list(categorical_cols)}")
    # Apply one-hot encoding
    df_encoded = pd.get_dummies(df_agg, columns=categorical_cols, drop_first=True) # drop_first=True to avoid multicollinearity

    print("Categorical variables encoded.")
    print(f"Shape after encoding: {df_encoded.shape}")
    print("\nColumns after encoding:")
    print(df_encoded.columns)

    # Update df_agg to the encoded version
    df_agg = df_encoded

    # Ensure dummy columns are integer type
    dummy_cols = [col for col in df_agg.columns if col.startswith(tuple(categorical_cols))]
    for col in dummy_cols:
        if df_agg[col].dtype not in [np.int64, np.int32, np.uint8, np.float64, np.float32]:
            df_agg[col] = df_agg[col].astype(int)
    print("Ensured dummy columns are integer type.")

else:
    print("No categorical columns found to encode.")
```

```
# Datetime columns should have been dropped during tenure calculation now
# Ensure City_last (original categorical column before dummifying) is dropped if it still exists
if 'City' in df_agg.columns:
     df_agg = df_agg.drop(columns=['City'], errors='ignore')
     print("Dropped original 'City' column.")
```

```
--- Encoding Categorical Variables ---
Applying One-Hot Encoding to: ['City']
Categorical variables encoded.
Shape after encoding: (2381, 42)

Columns after encoding:
Index(['Driver_ID', 'Age', 'Gender', 'Education_Level', 'Joining Designation',
       'Income_mean', 'Income', 'Grade', 'Total Business Value_mean',
       'Total Business Value', 'Quarterly_Rating_first', 'Quarterly Rating',
       'target', 'tenure_days', 'City_C10', 'City_C11', 'City_C12', 'City_C13',
       'City_C14', 'City_C15', 'City_C16', 'City_C17', 'City_C18', 'City_C19',
       'City_C2', 'City_C20', 'City_C21', 'City_C22', 'City_C23', 'City_C24',
       'City_C25', 'City_C26', 'City_C27', 'City_C28', 'City_C29', 'City_C3',
       'City_C4', 'City_C5', 'City_C6', 'City_C7', 'City_C8', 'City_C9'],
      dtype='object')
Ensured dummy columns are integer type.
```

```
# Final check for any remaining NaN values before splitting
print("\nChecking for NaN values before splitting:")
nan_check = df_agg.isnull().sum()
print(nan_check[nan_check > 0])

# If there are NaNs in numerical columns, fill with median
numerical_cols_final = df_agg.select_dtypes(include=np.number).columns
if 'Driver_ID' in numerical_cols_final:
    numerical_cols_final = numerical_cols_final.drop(['Driver_ID', 'target'], errors='ignore') # Exclude ID and target
else:
     numerical_cols_final = numerical_cols_final.drop(['target'], errors='ignore')

for col in numerical_cols_final:
    if df_agg[col].isnull().any():
        median_val = df_agg[col].median()
        df_agg[col] = df_agg[col].fillna(median_val)
        print(f"Filled NaN in numerical column {col} with median value {median_val}")
```

```
Checking for NaN values before splitting:
Series([], dtype: int64)
```

```
# --- Data Splitting ---
print("\n--- Splitting Data into Training and Testing Sets ---")

# Define features (X) and target (y)
# Ensure Driver_ID exists before trying to drop it
columns_to_drop_for_X = ['target']
if 'Driver_ID' in df_agg.columns:
    columns_to_drop_for_X.append('Driver_ID')

X = df_agg.drop(columns=columns_to_drop_for_X)
y = df_agg['target']


# Ensure all feature columns are numeric before proceeding
non_numeric_cols = X.select_dtypes(exclude=np.number).columns
if len(non_numeric_cols) > 0:
    print(f"Error: Non-numeric columns found in features: {list(non_numeric_cols)}")
    print("Please ensure all categorical features are encoded.")
    exit()

# Split data into training and testing sets (e.g., 80% train, 20% test)
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42, stratify=y) # Stratify by y for imbalance

print(f"Training set shape: X_train={X_train.shape}, y_train={y_train.shape}")
print(f"Testing set shape: X_test={X_test.shape}, y_test={y_test.shape}")
print(f"Training target distribution:\n{y_train.value_counts(normalize=True)}")
print(f"Testing target distribution:\n{y_test.value_counts(normalize=True)}")
```

```
--- Splitting Data into Training and Testing Sets ---
Training set shape: X_train=(1904, 40), y_train=(1904,)
Testing set shape: X_test=(477, 40), y_test=(477,)
Training target distribution:
target
1    0.678571
0    0.321429
Name: proportion, dtype: float64
Testing target distribution:
```

```
target
1    0.679245
0    0.320755
Name: proportion, dtype: float64
```

```
# --- Class Imbalance Treatment (SMOTE) ---
print("\n--- Handling Class Imbalance using SMOTE (on training data) ---")
smote = SMOTE(random_state=42)
X_train_resampled, y_train_resampled = smote.fit_resample(X_train, y_train)

print(f"Shape after SMOTE: X_train_resampled={X_train_resampled.shape}, y_train_resampled={y_train_resampled.shape}")
print(f"Training target distribution after SMOTE:\n{y_train_resampled.value_counts(normalize=True)}")
```

```
--- Handling Class Imbalance using SMOTE (on training data) ---
Shape after SMOTE: X_train_resampled=(2584, 40), y_train_resampled=(2584,)
Training target distribution after SMOTE:
target
0    0.5
1    0.5
Name: proportion, dtype: float64
```

```
# --- Standardization ---
print("\n--- Standardizing Numerical Features ---")

# Identify numerical columns to scale (should be all columns in X now)
# We fit the scaler ONLY on the training data (resampled)
scaler = StandardScaler()
X_train_scaled = scaler.fit_transform(X_train_resampled)
X_test_scaled = scaler.transform(X_test) # Use the same scaler fitted on training data

# Convert scaled arrays back to DataFrames (optional, but can be helpful)
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("Standardization complete.")
print("\nScaled Training Data Head:")
print(X_train_scaled.head())
```

```
--- Standardizing Numerical Features ---
Standardization complete.

Scaled Training Data Head:
        Age    Gender  Education_Level  Joining Designation  Income_mean  \
0  0.021597  1.255608        -1.147954             1.489580     1.598601
1  1.426191 -0.873650        -1.147954            -0.976195    -0.319129
2 -1.207422 -0.873650         0.116998             0.256693     0.553477
3  0.197172 -0.873650         0.116998            -0.976195     0.345110
4  0.372746 -0.873650        -1.147954             0.256693     1.790012

     Income     Grade  Total Business Value_mean  Total Business Value  \
0  1.588499  1.006710                  -0.720311             -0.270449
1 -0.322971 -1.192224                  -0.720311             -0.270449
2  0.546786 -0.092757                  -0.720311             -0.270449
3  0.339099  1.006710                   1.748135              0.823505
4  1.779285  1.006710                  -0.720311             -0.270449

   Quarterly_Rating_first  ...  City_C27  City_C28  City_C29   City_C3  \
0               -0.564950  ... -0.166865   -0.1693 -0.181035 -0.164399
1               -0.564950  ... -0.166865   -0.1693 -0.181035 -0.164399
2               -0.564950  ... -0.166865   -0.1693 -0.181035 -0.164399
3                0.710012  ... -0.166865   -0.1693 -0.181035 -0.164399
4               -0.564950  ... -0.166865   -0.1693 -0.181035 -0.164399

    City_C4   City_C5   City_C6   City_C7   City_C8   City_C9
0 -0.155491 -0.158083 -0.147471  6.541912 -0.168087 -0.159364
1 -0.155491 -0.158083 -0.147471 -0.152861 -0.168087  6.274950
2 -0.155491 -0.158083 -0.147471 -0.152861 -0.168087 -0.159364
3 -0.155491 -0.158083 -0.147471 -0.152861 -0.168087 -0.159364
4 -0.155491 -0.158083 -0.147471 -0.152861 -0.168087 -0.159364

[5 rows x 40 columns]
```

```
# --- Model Building ---
print("\n--- Model Building ---")

# --- Model 1: Random Forest (Bagging) ---
print("\nTraining Random Forest Classifier...")
rf_clf = RandomForestClassifier(random_state=42, n_estimators=100, class_weight='balanced') # Using default parameters + balanced weigh

# Optional: Hyperparameter Tuning with GridSearchCV (can be time-consuming)
# param_grid_rf = {
#     'n_estimators': [100, 200],
```

```
#     'max_depth': [None, 10, 20],
#     'min_samples_split': [2, 5]
# }
# grid_search_rf = GridSearchCV(RandomForestClassifier(random_state=42, class_weight='balanced'), param_grid_rf, cv=3, scoring='roc_auc
# grid_search_rf.fit(X_train_scaled, y_train_resampled)
# rf_clf = grid_search_rf.best_estimator_
# print(f"Best RF Params: {grid_search_rf.best_params_}")

rf_clf.fit(X_train_scaled, y_train_resampled)
print("Random Forest training complete.")
```

```
--- Model Building ---

Training Random Forest Classifier...
Random Forest training complete.
```

```
# --- Model 2: Gradient Boosting (Boosting) ---
print("\nTraining Gradient Boosting Classifier...")
gb_clf = GradientBoostingClassifier(random_state=42, n_estimators=100) # Using default parameters

# Optional: Hyperparameter Tuning with GridSearchCV
# param_grid_gb = {
#     'n_estimators': [100, 200],
#     'learning_rate': [0.1, 0.05],
#     'max_depth': [3, 5]
# }
# grid_search_gb = GridSearchCV(GradientBoostingClassifier(random_state=42), param_grid_gb, cv=3, scoring='roc_auc', n_jobs=-1)
# grid_search_gb.fit(X_train_scaled, y_train_resampled)
# gb_clf = grid_search_gb.best_estimator_
# print(f"Best GB Params: {grid_search_gb.best_params_}")

gb_clf.fit(X_train_scaled, y_train_resampled)
print("Gradient Boosting training complete.")
```

```
Training Gradient Boosting Classifier...
Gradient Boosting training complete.
```

```
# --- Results Evaluation ---
print("\n--- Results Evaluation ---")

# Predictions on the test set
y_pred_rf = rf_clf.predict(X_test_scaled)
y_prob_rf = rf_clf.predict_proba(X_test_scaled)[:, 1] # Probabilities for ROC AUC

y_pred_gb = gb_clf.predict(X_test_scaled)
y_prob_gb = gb_clf.predict_proba(X_test_scaled)[:, 1] # Probabilities for ROC AUC
```

```
--- Results Evaluation ---
```

```
# --- Evaluation Metrics ---

# Random Forest
print("\n--- Random Forest Evaluation ---")
print("Classification Report:")
print(classification_report(y_test, y_pred_rf))
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred_rf))
roc_auc_rf = roc_auc_score(y_test, y_prob_rf)
print(f"ROC AUC Score: {roc_auc_rf:.4f}")

# Gradient Boosting
print("\n--- Gradient Boosting Evaluation ---")
print("Classification Report:")
print(classification_report(y_test, y_pred_gb))
print("Confusion Matrix:")
print(confusion_matrix(y_test, y_pred_gb))
roc_auc_gb = roc_auc_score(y_test, y_prob_gb)
print(f"ROC AUC Score: {roc_auc_gb:.4f}")

# --- ROC Curve Data (for potential plotting) ---
fpr_rf, tpr_rf, _ = roc_curve(y_test, y_prob_rf)
fpr_gb, tpr_gb, _ = roc_curve(y_test, y_prob_gb)

# Plot ROC Curve and save to file
plt.figure(figsize=(8, 6))
plt.plot(fpr_rf, tpr_rf, label=f'Random Forest (AUC = {roc_auc_rf:.4f})')
plt.plot(fpr_gb, tpr_gb, label=f'Gradient Boosting (AUC = {roc_auc_gb:.4f})')
```

```
plt.plot([0, 1], [0, 1], 'k--', label='Chance') # Diagonal line
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('ROC Curve - Ola Driver Attrition')
plt.legend()
plt.grid(True)
plt.savefig('roc_curve.png')
print("\nROC curve saved to roc_curve.png")
plt.close()
```

```
--- Random Forest Evaluation ---
Classification Report:
              precision    recall  f1-score   support

           0       0.92      0.93      0.92       153
           1       0.97      0.96      0.96       324

    accuracy                           0.95       477
   macro avg       0.94      0.94      0.94       477
weighted avg       0.95      0.95      0.95       477

Confusion Matrix:
[[142  11]
 [ 13 311]]
ROC AUC Score: 0.9770

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

           0       0.92      0.93      0.93       153
           1       0.97      0.96      0.96       324

    accuracy                           0.95       477
   macro avg       0.94      0.95      0.94       477
weighted avg       0.95      0.95      0.95       477

Confusion Matrix:
[[143  10]
 [ 13 311]]
ROC AUC Score: 0.9811

ROC curve saved to roc_curve.png
```

```
# --- Feature Importance (Example for Random Forest) ---
print("\n--- Feature Importance (Random Forest) ---")
try:
    feature_importances = pd.DataFrame({
        'feature': X_train.columns,
        'importance': rf_clf.feature_importances_
    }).sort_values('importance', ascending=False)
    print("Top 10 Features (Random Forest):")
    print(feature_importances.head(10))
except AttributeError:
    print("Could not retrieve feature importances for Random Forest.")

# --- Feature Importance (Example for Gradient Boosting) ---
print("\n--- Feature Importance (Gradient Boosting) ---")
try:
    feature_importances_gb = pd.DataFrame({
        'feature': X_train.columns,
        'importance': gb_clf.feature_importances_
    }).sort_values('importance', ascending=False)
    print("Top 10 Features (Gradient Boosting):")
    print(feature_importances_gb.head(10))
except AttributeError:
    print("Could not retrieve feature importances for Gradient Boosting.")
```

```
--- Feature Importance (Random Forest) ---
Top 10 Features (Random Forest):
                     feature  importance
11               tenure_days    0.403484
8       Total Business Value    0.190034
10          Quarterly Rating    0.088285
7    Total Business Value_mean    0.084040
4                Income_mean    0.037928
5                     Income    0.037823
0                        Age    0.032485
1                     Gender    0.029033
9      Quarterly_Rating_first    0.013934
2            Education_Level    0.012983

--- Feature Importance (Gradient Boosting) ---
Top 10 Features (Gradient Boosting):
```

```
                 feature  importance
11           tenure_days    0.461960
8   Total Business Value    0.434459
7   Total Business Value_mean    0.065188
10      Quarterly Rating    0.009455
0                    Age    0.006926
1                 Gender    0.005044
29              City_C26    0.002224
4            Income_mean    0.002208
6                  Grade    0.002126
3    Joining Designation    0.001819
```

```
# --- Actionable Insights & Recommendations ---
print("\n--- Actionable Insights & Recommendations ---")
print("Based on the final model results and feature importances:")
print("1. Dominant Predictors: Driver tenure ('tenure_days') and the most recent month's 'Total Business Value' are overwhelmingly the r
print("2. High Predictive Power: Both Random Forest and Gradient Boosting models achieved excellent performance (AUC ~0.98), indicating
print("3. Retention Strategy - Tenure Milestones: Implement targeted engagement strategies based on tenure. Drivers might be more prone
print("4. Retention Strategy - Business Value Monitoring: Closely monitor drivers with low or declining 'Total Business Value'. Investig
print("5. Secondary Factors: While less dominant, factors like 'Quarterly Rating', 'Income', and 'Age' still play a role. Continue monit
print("6. Model Utility: The high accuracy suggests these models can be effectively deployed to proactively identify at-risk drivers, al


print("\n--- Analysis Complete ---")
```

```
--- Actionable Insights & Recommendations ---
Based on the final model results and feature importances:
1. Dominant Predictors: Driver tenure ('tenure_days') and the most recent month's 'Total Business Value' are overwhelmingly the most
2. High Predictive Power: Both Random Forest and Gradient Boosting models achieved excellent performance (AUC ~0.98), indicating a s
3. Retention Strategy - Tenure Milestones: Implement targeted engagement strategies based on tenure. Drivers might be more prone to
4. Retention Strategy - Business Value Monitoring: Closely monitor drivers with low or declining 'Total Business Value'. Investigate
5. Secondary Factors: While less dominant, factors like 'Quarterly Rating', 'Income', and 'Age' still play a role. Continue monitori
6. Model Utility: The high accuracy suggests these models can be effectively deployed to proactively identify at-risk drivers, allow

--- Analysis Complete ---
```