OLA Driver Churn Prediction

Gautam Naik (gautamnaik1994@gmail.com)

Github Link: https://github.com/gautamnaik1994/OLA-Driver-Churn-ML-CaseStudy

render nbviewer Open in Colab

About OLA

Ola Cabs offers mobility solutions by connecting customers to drivers and a wide range of vehicles across bikes, auto-rickshaws, metered taxis, and cabs, enabling convenience and transparency for hundreds of millions of consumers and over 1.5 million driver-partners. The primary concern for Ola is to ensure a quality driving experience for its users and retaining efficient drivers.

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.

Solution

As as a data scientist with the Analytics Department of Ola, focused on driver team attrition, We 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)

We will be bilding a predictive model to determine this. Along with this we are also going to determine which of the driver features is most responsible for driver churn

Predictive Algorithms

- We will be using Random Forest Classifier, Gradient Boosting Classifier, LightGBM and XGBoost

Metrics

- We will be using AUC-ROC and F1 score for selecting the best predictive model.
- We be also using classification report

Dataset

Feature	Description
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 for the driver
LastWorkingDate	Last date of working for 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	The total business value acquired by the driver in a month (negative business indicates cancellation/refund or car EMI adjustments)
Quarterly Rating	Quarterly rating of the driver: 1, 2, 3, 4, 5 (higher is better)

In [ ]:
```
from pyspark.sql import SparkSession
import pyspark.sql.functions as sf
from pyspark.sql.window import Window
import pandas as pd
import numpy as np
import seaborn as sns
import matplotlib.pyplot as plt
from sklearn.impute import KNNImputer
from pyspark.sql.types import IntegerType


import os
from IPython.display import clear_output
import warnings
warnings.filterwarnings("ignore")

sns.set_theme(style="whitegrid")
from IPython.core.interactiveshell import InteractiveShell
InteractiveShell.ast_node_interactivity = "all"
```

In [ ]:
```
os.environ['PYSPARK_PIN_THREAD'] = 'false'

spark = SparkSession.builder \
    .appName("OLA") \
    .config("spark.sql.debug.maxToStringFields", 1000) \
    .config("spark.sql.execution.arrow.pyspark.enabled", "true") \
    .config("spark.sql.shuffle.partitions", 1) \
    .config("spark.network.timeout", "120s") \
    .config("spark.executor.heartbeatInterval", "10s") \
    .config("spark.jars.packages", "com.microsoft.azure:synapseml_2.12:1.0.4,org.mlflow:mlflow-spark_2.13:2.15.1")   \
    .master("local[16]") \
    .getOrCreate();
clear_output()
```

In [ ]:
```
spark.sparkContext.setLogLevel("ERROR")
cores = spark._jsc.sc().getExecutorMemoryStatus().keySet().size()
print("You are working with", cores, "core(s)")
spark
```

```
You are working with 1 core(s)
```

Out[ ]:

SparkSession - in-memory

SparkContext

Spark UI

Version	`v3.5.1`
Master	`local[16]`
AppName	`OLA`

In [ ]:
```
spark_df = spark.read \
    .option("header", "true") \
    .option("inferSchema", "true") \
```

```
    .option("multiLine", "true") \
    .option("escape", "\"") \
    .csv("./ola_driver_scaler.csv")
spark_df.cache();
```

In [ ]:
```
spark_df.printSchema()
```

```
root
 |-- _c0: integer (nullable = true)
 |-- MMM-YY: string (nullable = true)
 |-- Driver_ID: integer (nullable = true)
 |-- Age: integer (nullable = true)
 |-- Gender: integer (nullable = true)
 |-- City: string (nullable = true)
 |-- Education_Level: integer (nullable = true)
 |-- Income: integer (nullable = true)
 |-- Dateofjoining: string (nullable = true)
 |-- LastWorkingDate: string (nullable = true)
 |-- Joining Designation: integer (nullable = true)
 |-- Grade: integer (nullable = true)
 |-- Total Business Value: integer (nullable = true)
 |-- Quarterly Rating: integer (nullable = true)
```

In [ ]:
```
spark_df = spark_df.drop('_c0');
spark_df.show();
```

```
+--------+---------+---+------+----+--------------+------+-------------+---------------+-------------------+-----+--------------------+---------------+
|  MMM-YY|Driver_ID|Age|Gender|City|Education_Level|Income|Dateofjoining|LastWorkingDate|Joining Designation|Grade|Total Business Value|Quarterly Rating|
+--------+---------+---+------+----+--------------+------+-------------+---------------+-------------------+-----+--------------------+---------------+
|01/01/19|        1| 28|     0| C23|             2| 57387|     24/12/18|           NULL|                  1|    1|             2381060|              2|
|02/01/19|        1| 28|     0| C23|             2| 57387|     24/12/18|           NULL|                  1|    1|             -665480|              2|
|03/01/19|        1| 28|     0| C23|             2| 57387|     24/12/18|       03/11/19|                  1|    1|                   0|              2|
|11/01/20|        2| 31|     0|  C7|             2| 67016|     11/06/20|           NULL|                  2|    2|                   0|              1|
|12/01/20|        2| 31|     0|  C7|             2| 67016|     11/06/20|           NULL|                  2|    2|                   0|              1|
|12/01/19|        4| 43|     0| C13|             2| 65603|     12/07/19|           NULL|                  2|    2|                   0|              1|
|01/01/20|        4| 43|     0| C13|             2| 65603|     12/07/19|           NULL|                  2|    2|                   0|              1|
|02/01/20|        4| 43|     0| C13|             2| 65603|     12/07/19|           NULL|                  2|    2|                   0|              1|
|03/01/20|        4| 43|     0| C13|             2| 65603|     12/07/19|           NULL|                  2|    2|              350000|              1|
|04/01/20|        4| 43|     0| C13|             2| 65603|     12/07/19|       27/04/20|                  2|    2|                   0|              1|
|01/01/19|        5| 29|     0|  C9|             0| 46368|     01/09/19|           NULL|                  1|    1|                   0|              1|
|02/01/19|        5| 29|     0|  C9|             0| 46368|     01/09/19|           NULL|                  1|    1|              120360|              1|
|03/01/19|        5| 29|     0|  C9|             0| 46368|     01/09/19|       03/07/19|                  1|    1|                   0|              1|
|08/01/20|        6| 31|     1| C11|             1| 78728|     31/07/20|           NULL|                  3|    3|                   0|              1|
|09/01/20|        6| 31|     1| C11|             1| 78728|     31/07/20|           NULL|                  3|    3|                   0|              1|
|10/01/20|        6| 31|     1| C11|             1| 78728|     31/07/20|           NULL|                  3|    3|                   0|              2|
|11/01/20|        6| 31|     1| C11|             1| 78728|     31/07/20|           NULL|                  3|    3|             1265000|              2|
|12/01/20|        6| 31|     1| C11|             1| 78728|     31/07/20|           NULL|                  3|    3|                   0|              2|
|09/01/20|        8| 34|     0|  C2|             0| 70656|     19/09/20|           NULL|                  3|    3|                   0|              1|
|10/01/20|        8| 34|     0|  C2|             0| 70656|     19/09/20|           NULL|                  3|    3|                   0|              1|
+--------+---------+---+------+----+--------------+------+-------------+---------------+-------------------+-----+--------------------+---------------+
only showing top 20 rows
```

In [ ]:
```
spark_df.filter("LastWorkingDate is not null").show();
```

```
+--------+---------+---+------+----+--------------+------+-------------+---------------+-------------------+-----+--------------------+---------------+
|  MMM-YY|Driver_ID|Age|Gender|City|Education_Level|Income|Dateofjoining|LastWorkingDate|Joining Designation|Grade|Total Business Value|Quarterly Rating|
+--------+---------+---+------+----+--------------+------+-------------+---------------+-------------------+-----+--------------------+---------------+
|03/01/19|        1| 28|     0| C23|             2| 57387|     24/12/18|       03/11/19|                  1|    1|                   0|              2|
|04/01/20|        4| 43|     0| C13|             2| 65603|     12/07/19|       27/04/20|                  2|    2|                   0|              1|
|03/01/19|        5| 29|     0|  C9|             0| 46368|     01/09/19|       03/07/19|                  1|    1|                   0|              1|
|11/01/20|        8| 34|     0|  C2|             0| 70656|     19/09/20|       15/11/20|                  3|    3|                   0|              1|
|12/01/19|       12| 35|     0| C23|             2| 28116|     29/06/19|       21/12/19|                  1|    1|                   0|              1|
|11/01/20|       13| 31|     0| C19|             2|119227|     28/05/15|       25/11/20|                  1|    4|                   0|              1|
|02/01/19|       16| 30|     1| C23|             0| 52963|     30/11/18|       22/02/19|                  2|    2|                   0|              1|
|07/01/19|       17| 43|     0| C20|             2| 51099|     03/06/18|       20/07/19|                  1|    1|                   0|              1|
|05/01/19|       18| 27|     1| C17|             1| 31631|     01/09/19|       30/04/19|                  1|    1|                   0|              1|
|03/01/20|       20| 26|     1| C19|             0| 40342|     25/10/19|       03/01/20|                  3|    3|                   0|              1|
|02/01/20|       21| 34|     1| C29|             1| 22755|     05/12/18|       17/02/20|                  1|    1|              281380|              1|
|04/01/20|       22| 41|     0| C10|             2| 31224|     25/05/18|       26/04/20|                  1|    1|                   0|              1|
|10/01/19|       24| 31|     0| C24|             2| 76308|     25/05/18|       27/10/19|                  1|    2|              343160|              2|
|05/01/19|       29| 30|     0| C13|             2| 30312|     20/05/18|       23/05/19|                  1|    1|                   0|              1|
|12/01/19|       30| 31|     0|  C6|             1| 69457|     10/03/19|       29/11/19|                  3|    3|                   0|              1|
|11/01/20|       34| 28|     1| C14|             1| 65813|     31/07/20|       14/11/20|                  2|    2|                   0|              1|
|05/01/20|       35| 32|     0|  C5|             1| 68070|     03/07/20|       05/05/20|                  3|    3|                   0|              1|
|08/01/20|       36| 41|     1| C18|             1| 32865|     07/01/19|       29/07/20|                  2|    2|                -325140|              2|
|10/01/20|       37| 33|     1| C14|             1| 57375|     16/05/20|       16/10/20|                  2|    2|                   0|              1|
|03/01/20|       38| 22|     0| C27|             0| 13427|     12/06/19|       15/03/20|                  1|    1|                -155610|              1|
+--------+---------+---+------+----+--------------+------+-------------+---------------+-------------------+-----+--------------------+---------------+
only showing top 20 rows
```

In [ ]:
```
spark_df.select([sf.count(sf.when(sf.isnull(c), c)).alias(c) for c in spark_df.columns]).toPandas().T
```

Out[ ]:

	0
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

Observations

- We can see that there are lot of missing values in the dataset for LastWorkingDate
- There are some missing values for Age and Gender

Data Cleaning and Feature Engineering

New features based on dates

In [ ]:
```
spark_df = spark_df.withColumn("reporting_month_year", sf.to_date(spark_df["MMM-YY"], "MM/dd/yy")).drop("MMM-YY")
spark_df = spark_df.withColumn("Dateofjoining", sf.to_date(spark_df["Dateofjoining"], "dd/MM/yy"))
spark_df = spark_df.withColumn("LastWorkingDate", sf.to_date(spark_df["LastWorkingDate"], "dd/MM/yy"))
spark_df = spark_df.withColumn("Quarter", sf.quarter(spark_df["reporting_month_year"]))
spark_df = spark_df.withColumn("ReportingYear", sf.year(spark_df["reporting_month_year"]))
spark_df = spark_df.withColumn("ReportingYear_Quarter", sf.concat(sf.col("ReportingYear"), sf.lit("-"), sf.col("Quarter")))
```

In [ ]:
```
spark_df=spark_df.dropDuplicates()
```

In [ ]:
```
spark_df.filter(sf.col("Gender").isNull()).show()
```

```
+---------+---+------+----+--------------+------+-------------+---------------+-------------------+-----+-------------------+---------------+--------------------+-------+-------------+
--------------------+
|Driver_ID|Age|Gender|City|Education_Level|Income|Dateofjoining|LastWorkingDate|Joining Designation|Grade|Total Business Value|Quarterly Rating|reporting_month_year|Quarter|ReportingYear|
ReportingYear_Quarter|
+---------+---+------+----+--------------+------+-------------+---------------+-------------------+-----+-------------------+---------------+--------------------+-------+-------------+
--------------------+
|       43| 27|  NULL| C15|             0| 12906|   2018-07-13|     2019-02-20|                  1|    1|                  0|              1|          2019-02-01|      1|         2019|
2019-1|
|       49| 21|  NULL| C20|             0| 53039|   2018-05-25|           NULL|                  1|    2|                  0|              1|          2019-02-01|      1|         2019|
2019-1|
|       49| 21|  NULL| C20|             0| 53039|   2018-05-25|           NULL|                  1|    2|             300300|              2|          2019-08-01|      3|         2019|
2019-3|
|       68| 31|  NULL| C29|             0| 79288|   2015-10-18|           NULL|                  1|    3|             544930|              3|          2019-08-01|      3|         2019|
2019-3|
|      116| 21|  NULL| C11|             0| 16477|   2018-04-12|           NULL|                  1|    1|             129590|              1|          2019-02-01|      1|         2019|
2019-1|
|      119| 31|  NULL| C29|             1| 71000|   2019-11-16|           NULL|                  3|    3|                  0|              1|          2019-11-01|      4|         2019|
2019-4|
|      225| 32|  NULL| C14|             0| 44792|   2020-07-13|           NULL|                  3|    3|             337020|              3|          2020-12-01|      4|         2020|
2020-4|
|      296| 31|  NULL| C20|             1| 65094|   2018-10-06|           NULL|                  1|    2|             145670|              2|          2019-08-01|      3|         2019|
2019-3|
|      354| 31|  NULL| C11|             0| 60555|   2018-11-30|           NULL|                  1|    1|                  0|              1|          2019-02-01|      1|         2019|
2019-1|
|      365| 24|  NULL| C22|             0| 44740|   2020-01-02|           NULL|                  2|    2|                  0|              1|          2020-03-01|      1|         2020|
2020-1|
|      407| 40|  NULL| C13|             1| 58207|   2016-05-17|           NULL|                  1|    2|             427570|              4|          2019-06-01|      2|         2019|
2019-2|
|      439| 27|  NULL|  C3|             1| 60246|   2019-11-28|           NULL|                  1|    1|            1561820|              3|          2020-03-01|      1|         2020|
2020-1|
|      446| 31|  NULL| C22|             0| 50832|   2020-01-02|           NULL|                  3|    3|             890060|              2|          2020-07-01|      3|         2020|
2020-3|
|      489| 31|  NULL| C12|             2| 49475|   2019-10-18|           NULL|                  1|    1|             500510|              3|          2020-01-01|      1|         2020|
2020-1|
|      516| 26|  NULL| C29|             0| 41099|   2019-04-07|           NULL|                  1|    1|             422710|              1|          2020-10-01|      4|         2020|
2020-4|
|      541| 27|  NULL|  C1|             2| 71812|   2017-10-02|           NULL|                  1|    2|             200000|              2|          2020-02-01|      1|         2020|
2020-1|
|      611| 32|  NULL| C10|             1| 39216|   2020-01-13|           NULL|                  2|    2|                  0|              1|          2020-02-01|      1|         2020|
2020-1|
|      640| 26|  NULL|  C8|             1|105931|   2019-07-12|           NULL|                  3|    3|                  0|              1|          2019-12-01|      4|         2019|
2019-4|
|      709| 31|  NULL| C13|             1|135436|   2018-04-08|           NULL|                  5|    5|                  0|              1|          2019-04-01|      2|         2019|
2019-2|
|      793| 26|  NULL|  C7|             2| 92670|   2020-10-30|           NULL|                  3|    3|             127800|              1|          2020-12-01|      4|         2020|
2020-4|
+---------+---+------+----+--------------+------+-------------+---------------+-------------------+-----+-------------------+---------------+--------------------+-------+-------------+
--------------------+
only showing top 20 rows
```

In [ ]:
```
spark_df.filter("Driver_ID == 305").sort("reporting_month_year").show()
```

```
+---------+----+------+----+--------------+------+-------------+---------------+-------------------+-----+-------------------+---------------+--------------------+-------+
|Driver_ID| Age|Gender|City|Education_Level|Income|Dateofjoining|LastWorkingDate|Joining Designation|Grade|Total Business Value|Quarterly Rating|reporting_month_year|Quarter|
+---------+----+------+----+--------------+------+-------------+---------------+-------------------+-----+-------------------+---------------+--------------------+-------+
|      305|NULL|     0|  C9|             0| 20176|   2018-08-26|           NULL|                  1|    1|                  0|              1|          2019-01-01|      1|
|      305|  24|     0|  C9|             0| 20176|   2018-08-26|     2019-08-02|                  1|    1|                  0|              1|          2019-02-01|      1|
+---------+----+------+----+--------------+------+-------------+---------------+-------------------+-----+-------------------+---------------+--------------------+-------+
```

Null value treatment

In [ ]:
```
list_of_columns = ['Driver_ID',
 'Age',
 'Gender',
 'Education_Level',
 'Income',
 'Joining Designation',
 'Grade',
 'Total Business Value',
 'Quarterly Rating']

df_numpy = np.array(spark_df.select(list_of_columns).collect())

k_imputer = KNNImputer(n_neighbors=5, weights='distance')
k_imputer.fit(df_numpy)

sc = spark.sparkContext
broadcast_model = sc.broadcast(k_imputer)

column_index_mapping = {col: idx for idx, col in enumerate(list_of_columns)}

def create_knn_imputer_udf(column_name):
    index = column_index_mapping[column_name]

    @sf.udf(IntegerType())
    def knn_impute(*cols):
        row = np.array(cols).reshape(1, -1)
        imputed_row = broadcast_model.value.transform(row)
        return int(imputed_row[0][index])
    return knn_impute
```

Out[ ]:
```
KNNImputer(weights='distance')
```

In [ ]:
```
knn_impute_Age = create_knn_imputer_udf("Age")
knn_impute_Gender = create_knn_imputer_udf("Gender")

spark_df = spark_df.withColumn("Age_Imputed", knn_impute_Age(*list_of_columns))
spark_df = spark_df.withColumn("Gender_Imputed", knn_impute_Gender(*list_of_columns))
```

In [ ]:
```
spark_df = spark_df.withColumn("Age", sf.when(sf.col("Age").isNull(), sf.col("Age_Imputed")).otherwise(sf.col("Age")))
spark_df = spark_df.withColumn("Gender", sf.when(sf.col("Gender").isNull(), sf.col("Gender_Imputed")).otherwise(sf.col("Gender")))
```

In [ ]:
```
spark_df = spark_df.drop("Age_Imputed", "Gender_Imputed")
```

Feature Generation

Since we want to predict whether a driver is going to churn or not, we will create a new column **Churned** as the target column

In [ ]:
```
window_spec = Window.partitionBy("Driver_ID").orderBy("reporting_month_year")

spark_df=spark_df.withColumns({
    # "Age":  sf.coalesce(sf.col("Age"), sf.first("Age", True).over(window_spec)),
    # "Gender":  sf.coalesce(sf.col("Gender"), sf.first("Age", True).over(window_spec)),
    "LastWorkingDate":  sf.coalesce(sf.col("LastWorkingDate"), sf.first("LastWorkingDate", True).over(window_spec)),
    "Churned":  sf.when(sf.col("LastWorkingDate").isNotNull(), 1).otherwise(0),
    "Had_Negative_Business":  sf.when(sf.col("Total Business Value") > 0, 1).otherwise(0),
    "Has_Income_Increased": sf.when(sf.last("Income").over(window_spec) > sf.first("Income").over(window_spec), 1).otherwise(0),
    "Has_Rating_Increased": sf.when(sf.last("Quarterly Rating").over(window_spec) > sf.first("Quarterly Rating").over(window_spec), 1).otherwise(0),
    # "Has_Grade_Increased": sf.when(sf.last("Grade").over(window_spec) > sf.first("Grade").over(window_spec), 1).otherwise(0),
})
```

In [ ]:
```
spark_df.createOrReplaceTempView("ola")
```

In [ ]:
```
spark.sql("""
          with cte as (
            select Driver_ID, reporting_month_year, Income,
              last(Income) over(partition by Driver_ID order by reporting_month_year) - first(Income) over(partition by Driver_ID order by reporting_month_year) as diff
            from ola
          )
          select * from cte where diff > 0
""").show(20)
```

```
+---------+--------------------+------+-----+
|Driver_ID|reporting_month_year|Income| diff|
+---------+--------------------+------+-----+
|       26|          2020-03-01|132577|11048|
|       26|          2020-04-01|132577|11048|
|       26|          2020-05-01|132577|11048|
|       26|          2020-06-01|132577|11048|
|       26|          2020-07-01|132577|11048|
|       26|          2020-08-01|132577|11048|
|       26|          2020-09-01|132577|11048|
|       26|          2020-10-01|132577|11048|
|       26|          2020-11-01|132577|11048|
|       26|          2020-12-01|132577|11048|
|       54|          2020-06-01|127826| 9833|
|       54|          2020-07-01|127826| 9833|
|       54|          2020-08-01|127826| 9833|
|       54|          2020-09-01|127826| 9833|
|       54|          2020-10-01|127826| 9833|
|       54|          2020-11-01|127826| 9833|
|       54|          2020-12-01|127826| 9833|
|       60|          2020-07-01| 89592| 7466|
|       60|          2020-08-01| 89592| 7466|
|       60|          2020-09-01| 89592| 7466|
+---------+--------------------+------+-----+
only showing top 20 rows
```

In [ ]:
```
spark.sql("""
          with cte as (
            select Driver_ID, reporting_month_year, Income,
              last(Grade) over(partition by Driver_ID order by reporting_month_year) - first(Grade) over(partition by Driver_ID order by reporting_month_year) as diff
            from ola
          )
          select distinct Driver_ID from cte where diff > 0
""").show(20)
```

```
+---------+
|Driver_ID|
+---------+
|       26|
|       54|
|       60|
|       98|
|      275|
|      307|
|      320|
|      368|
|      434|
|      537|
|      568|
|      580|
|      582|
|      638|
|      716|
|      789|
|      888|
|     1031|
|     1050|
|     1161|
+---------+
only showing top 20 rows
```

In [ ]:
```
spark.sql("""
    select
          Driver_Id
          , max(reporting_month_year) as max_reporting_month_year
          , max(Dateofjoining) as max_Dateofjoining
          , max(LastWorkingDate) as max_LastWorkingDate
        from ola
        group by Driver_Id
          having max_LastWorkingDate < max_Dateofjoining
""").show()
```

```
+---------+-----------------------+-----------------+-------------------+
|Driver_Id|max_reporting_month_year|max_Dateofjoining|max_LastWorkingDate|
+---------+-----------------------+-----------------+-------------------+
|        5|             2019-03-01|       2019-09-01|         2019-07-03|
|       18|             2019-05-01|       2019-09-01|         2019-04-30|
|       35|             2020-05-01|       2020-07-03|         2020-05-05|
|       59|             2019-03-01|       2019-06-01|         2019-03-28|
|       82|             2020-08-01|       2020-11-04|         2020-07-29|
|      102|             2019-03-01|       2019-05-01|         2019-03-21|
|      108|             2020-06-01|       2020-09-01|         2020-02-06|
|      118|             2020-09-01|       2020-10-03|         2020-09-15|
|      121|             2020-05-01|       2020-07-03|         2020-05-24|
|      139|             2019-12-01|       2019-10-31|         2019-05-12|
|      147|             2019-10-01|       2019-07-20|         2019-01-10|
|      184|             2020-07-01|       2020-12-04|         2020-05-07|
|      208|             2019-05-01|       2019-06-01|         2019-05-24|
|      222|             2019-10-01|       2019-12-09|         2019-08-10|
|      284|             2020-02-01|       2020-03-01|         2020-02-13|
|      286|             2020-11-01|       2020-07-13|         2020-07-11|
|      287|             2020-09-01|       2020-05-22|         2020-04-09|
|      315|             2020-10-01|       2020-12-09|         2020-05-10|
|      332|             2019-03-01|       2019-03-02|         2019-01-03|
|      364|             2020-07-01|       2020-07-03|         2020-06-29|
+---------+-----------------------+-----------------+-------------------+
only showing top 20 rows
```

In [ ]:
```
spark.sql("""
          select
          Driver_ID
          , max(`Quarterly Rating`) as max_qr
          , min(`Quarterly Rating`) as min_qr
          , avg(`Quarterly Rating`) as avg_qr
          , count(*) as count_qr
          from ola
          group by Driver_ID
          order by count_qr desc
           """).show()
```

```
+---------+------+------+------+--------+
|Driver_ID|max_qr|min_qr|avg_qr|count_qr|
+---------+------+------+------+--------+
|       56|     4|     2| 2.875|      24|
|       60|     4|     2|   3.5|      24|
|      486|     3|     1| 1.875|      24|
|       63|     3|     1| 1.875|      24|
|     1472|     3|     1|  1.75|      24|
|       67|     3|     1|   2.5|      24|
|     1664|     2|     2|   2.0|      24|
|       68|     4|     3|  3.25|      24|
|     2168|     3|     2| 2.625|      24|
|       77|     4|     2|   3.0|      24|
|     2545|     3|     1|   2.0|      24|
|       78|     4|     2| 2.625|      24|
|       25|     4|     3|  3.75|      24|
|      112|     4|     2| 2.875|      24|
|      173|     4|     2|  2.75|      24|
|      115|     3|     2| 2.125|      24|
|     1612|     4|     1|  2.25|      24|
|      117|     4|     2|   2.5|      24|
|      199|     4|     2|  3.25|      24|
|      140|     3|     2| 2.625|      24|
+---------+------+------+------+--------+
only showing top 20 rows
```

In [ ]:
```
spark_df.filter("`Total Business Value` = 0").limit(10).toPandas()
```

Out[ ]:

	Driver_ID	Age	Gender	City	Education_Level	Income	Dateofjoining	LastWorkingDate	Joining Designation	Grade	Total Business Value	Quarterly Rating	reporting_month_year	Quarter	Churned	Had_Negative_Business	Has_Incom
0	1	28	0	C23	2	57387	2018-12-24	2019-11-03	1	1	0	2	2019-03-01	1	1	0	
1	2	31	0	C7	2	67016	2020-06-11	None	2	2	0	1	2020-11-01	4	0	0	
2	2	31	0	C7	2	67016	2020-06-11	None	2	2	0	1	2020-12-01	4	0	0	
3	4	43	0	C13	2	65603	2019-07-12	None	2	2	0	1	2019-12-01	4	0	0	
4	4	43	0	C13	2	65603	2019-07-12	None	2	2	0	1	2020-01-01	1	0	0	
5	4	43	0	C13	2	65603	2019-07-12	None	2	2	0	1	2020-02-01	1	0	0	
6	4	43	0	C13	2	65603	2019-07-12	2020-04-27	2	2	0	1	2020-04-01	2	1	0	
7	5	29	0	C9	0	46368	2019-09-01	None	1	1	0	1	2019-01-01	1	0	0	
8	5	29	0	C9	0	46368	2019-09-01	2019-07-03	1	1	0	1	2019-03-01	1	1	0	
9	6	31	1	C11	1	78728	2020-07-31	None	3	3	0	1	2020-08-01	3	0	0	

In [ ]:
```
spark_df.select(sf.max("LastWorkingDate")).show()
```

```
+-------------------+
|max(LastWorkingDate)|
+-------------------+
|         2020-12-28|
+-------------------+
```

In [ ]:
```
spark_df.select(sf.max("reporting_month_year")).show()
```

```
+------------------------+
|max(reporting_month_year)|
+------------------------+
|              2020-12-01|
+------------------------+
```

Data Aggregation based on Driver ID

In [ ]:
```
agg_map=[
    sf.first("Dateofjoining").alias("Date_Of_Joining"),
    sf.sum("Total Business Value").alias("Total_Business_Value"),
    sf.sum("Had_Negative_Business").alias("Total_Had_Negative_Business"),
    sf.max("Has_Income_Increased").alias("Has_Income_Increased"),
    sf.max("Has_Rating_Increased").alias("Has_Rating_Increased"),
    # sf.max("Has_Grade_Increased").alias("Has_Grade_Increased"),
    sf.avg("Total Business Value").cast("int").alias("Avg_Business_Value"),
    sf.max("reporting_month_year").alias("Last_Reporting_Month"),
    sf.max("Age").alias("Age"),
    sf.mode("Gender").alias("Gender"),
    sf.last("Income").alias("Income"),
    sf.sum("Income").alias("Total_Income"),
    sf.first("Education_Level").alias("Education_Level"),
    sf.last("City").alias("City"),
    sf.first("Joining Designation").alias("Joining_Designation"),
    sf.last("Grade").alias("Grade"),
    sf.last("Quarterly Rating").alias("Quarterly_Rating"),
    # sf.min("Quarterly Rating").alias("Min_Quarterly_Rating"),
    # sf.max("Quarterly Rating").alias("Max_Quarterly_Rating"),
    sf.max("LastWorkingDate").alias("Last_Working_Date"),
    sf.max("churned").alias("Churned"),
]


merged_df = spark_df.sort("reporting_month_year").groupBy("Driver_ID").agg(*agg_map)
```

Generating new features

In [ ]:
```
default_date = "2020-12-31"

merged_df = merged_df.withColumn(
    "Tenure",
      sf.abs(
        sf.datediff(
            sf.when(sf.col("Last_Working_Date").isNull(), sf.lit(default_date)).otherwise(sf.col("Last_Working_Date")),
            sf.col("Date_Of_Joining")
        )
    )
)


merged_df=merged_df.withColumn("Date_Of_Joining_month", sf.month(merged_df["Date_Of_Joining"]))
merged_df=merged_df.withColumn("Date_Of_Joining_year", sf.year(merged_df["Date_Of_Joining"]))
merged_df=merged_df.withColumn("Is_Valuable_Driver", sf.when(merged_df["Total_Business_Value"] > merged_df["Total_Income"], 1).otherwise(0))

merged_df.limit(5).toPandas().T
```

Out[ ]:

	0	1	2	3	4
Driver_ID	1	2	4	5	6
Date_Of_Joining	2018-12-24	2020-06-11	2019-07-12	2019-09-01	2020-07-31
Total_Business_Value	1715580	0	350000	120360	1265000
Total_Had_Negative_Business	1	0	1	1	1
Has_Income_Increased	0	0	0	0	0
Has_Rating_Increased	0	0	0	0	1
Avg_Business_Value	571860	0	70000	40120	253000
Last_Reporting_Month	2019-03-01	2020-12-01	2020-04-01	2019-03-01	2020-12-01
Age	28	31	43	29	31
Gender	0	0	0	0	1
Income	57387	67016	65603	46368	78728
Total_Income	172161	134032	328015	139104	393640
Education_Level	2	2	2	0	1
City	C23	C7	C13	C9	C11
Joining_Designation	1	2	2	1	3
Grade	1	2	2	1	3
Quarterly_Rating	2	1	1	1	2
Last_Working_Date	2019-11-03	None	2020-04-27	2019-07-03	None
Churned	1	0	1	1	0
Tenure	314	203	290	60	153
Date_Of_Joining_month	12	6	7	9	7
Date_Of_Joining_year	2018	2020	2019	2019	2020
Is_Valuable_Driver	1	0	1	0	1

In [ ]:
```
spark_df.write.parquet("ola_driver_cleaned.parquet", mode='overwrite')
merged_df.write.parquet("ola_driver_merged.parquet", mode='overwrite')
```

EDA

In [ ]:
```
ola_df = spark.read.parquet("ola_driver_cleaned.parquet")
merged_df = spark.read.parquet("ola_driver_merged.parquet")
ola_df.createOrReplaceTempView("ola_driver")
merged_df.createOrReplaceTempView("ola_driver_merged")
```

In [ ]:
```
ola_pdf=ola_df.toPandas()
merged_df=merged_df.toPandas()
```

In [ ]:
```
merged_df["Last_Working_Date"] = pd.to_datetime(merged_df["Last_Working_Date"], errors="coerce")
```

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

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2381 entries, 0 to 2380
Data columns (total 23 columns):
 #   Column                       Non-Null Count  Dtype         
---  ------                       --------------  -----         
 0   Driver_ID                    2381 non-null   int32         
 1   Date_Of_Joining              2381 non-null   object        
 2   Total_Business_Value         2381 non-null   int64         
 3   Total_Had_Negative_Business  2381 non-null   int64         
 4   Has_Income_Increased         2381 non-null   int32         
 5   Has_Rating_Increased         2381 non-null   int32         
 6   Avg_Business_Value           2381 non-null   int32         
 7   Last_Reporting_Month         2381 non-null   object        
 8   Age                          2381 non-null   int32         
 9   Gender                       2381 non-null   int32         
 10  Income                       2381 non-null   int32         
 11  Total_Income                 2381 non-null   int64         
 12  Education_Level              2381 non-null   int32         
 13  City                         2381 non-null   object        
 14  Joining_Designation          2381 non-null   int32         
 15  Grade                        2381 non-null   int32         
 16  Quarterly_Rating             2381 non-null   int32         
 17  Last_Working_Date            1616 non-null   datetime64[ns]
 18  Churned                      2381 non-null   int32         
 19  Tenure                       2381 non-null   int32         
 20  Date_Of_Joining_month        2381 non-null   int32         
 21  Date_Of_Joining_year         2381 non-null   int32         
 22  Is_Valuable_Driver           2381 non-null   int32         
dtypes: datetime64[ns](1), int32(16), int64(3), object(3)
memory usage: 279.2+ KB
```

In [ ]:
```
merged_df.describe().T
```

Out[ ]:

	count	mean	min	25%	50%	75%	max	std
Driver_ID	2381.0	1397.559009	1.0	695.0	1400.0	2100.0	2788.0	806.161628
Total_Business_Value	2381.0	4586741.822764	-1385530.0	0.0	817680.0	4173650.0	95331060.0	9127115.313446
Total_Had_Negative_Business	2381.0	5.231415	0.0	0.0	2.0	7.0	24.0	6.956693
Has_Income_Increased	2381.0	0.01848	0.0	0.0	0.0	0.0	1.0	0.134706
Has_Rating_Increased	2381.0	0.293574	0.0	0.0	0.0	1.0	1.0	0.455495
Avg_Business_Value	2381.0	312085.152037	-197932.0	0.0	150624.0	429498.0	3972127.0	449570.40148
Age	2381.0	33.740865	21.0	29.0	33.0	37.0	58.0	5.947781
Gender	2381.0	0.409912	0.0	0.0	0.0	1.0	1.0	0.49192
Income	2381.0	59317.239815	10747.0	39104.0	55315.0	75986.0	188418.0	28362.787012
Total_Income	2381.0	526760.305754	10883.0	139895.0	292980.0	651456.0	4522032.0	623163.278373
Education_Level	2381.0	1.00756	0.0	0.0	1.0	2.0	2.0	0.81629
Joining_Designation	2381.0	1.820244	1.0	1.0	2.0	2.0	5.0	0.841433
Grade	2381.0	2.094498	1.0	1.0	2.0	3.0	5.0	0.939278
Quarterly_Rating	2381.0	1.449391	1.0	1.0	1.0	2.0	4.0	0.82175
Last_Working_Date	1616	2019-12-26 23:22:34.455445760	2018-12-31 00:00:00	2019-06-10 00:00:00	2019-12-20 12:00:00	2020-07-14 00:00:00	2020-12-28 00:00:00	NaN
Churned	2381.0	0.678706	0.0	0.0	1.0	1.0	1.0	0.467071
Tenure	2381.0	462.698026	0.0	116.0	241.0	508.0	2918.0	566.519585
Date_Of_Joining_month	2381.0	6.958001	1.0	5.0	7.0	10.0	12.0	3.221762
Date_Of_Joining_year	2381.0	2018.536329	2013.0	2018.0	2019.0	2020.0	2020.0	1.609597
Is_Valuable_Driver	2381.0	0.640907	0.0	0.0	1.0	1.0	1.0	0.479835

In [ ]:
```
merged_df[merged_df["Tenure"]==0].T
```

Out[ ]:

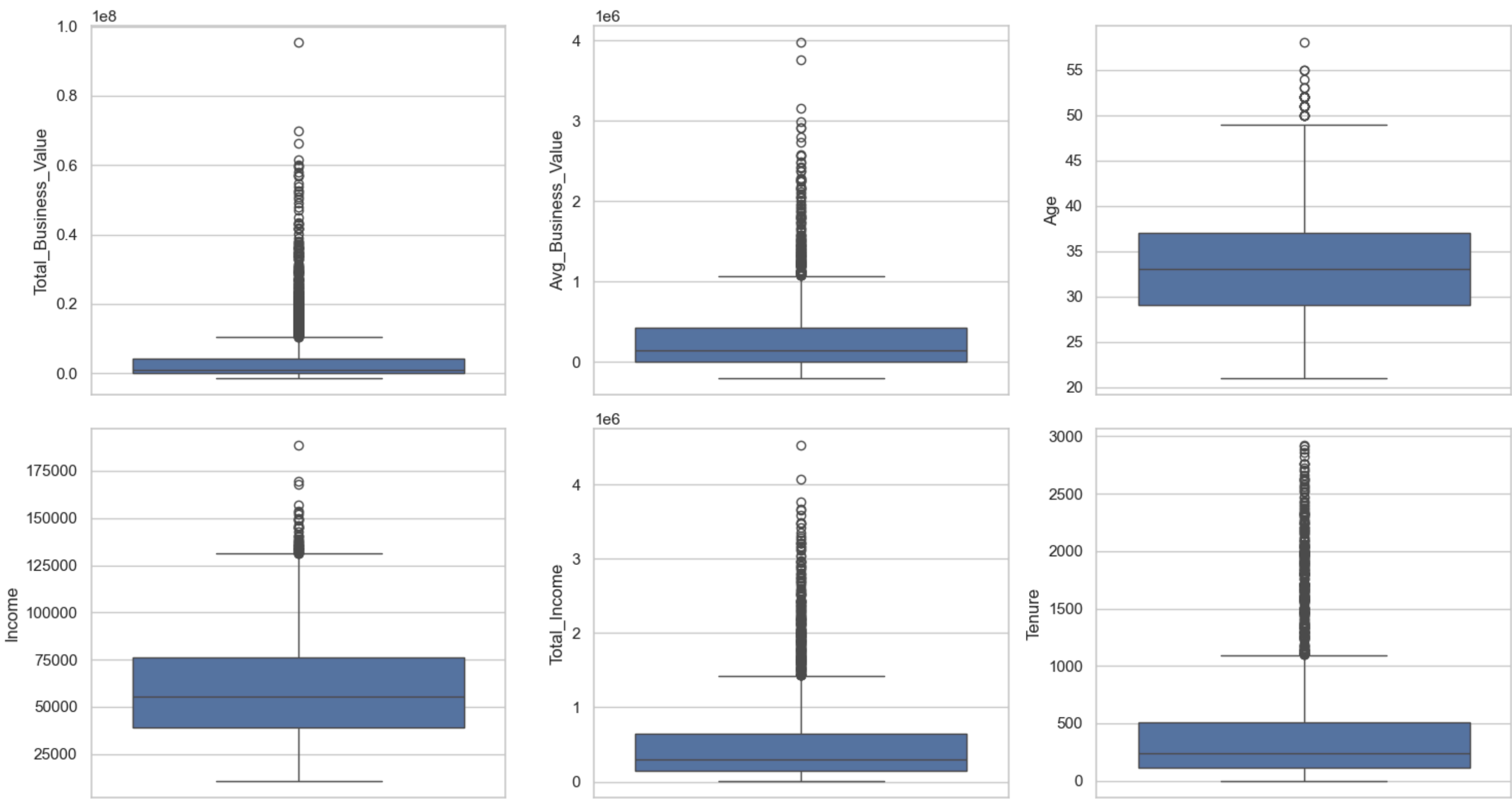
	220	1026	1344	2041
Driver_ID	264	1207	1581	2397
Date_Of_Joining	2020-12-18	2020-12-04	2019-06-30	2020-05-15
Total_Business_Value	0	0	0	0
Total_Had_Negative_Business	0	0	0	0
Has_Income_Increased	0	0	0	0
Has_Rating_Increased	0	0	0	0
Avg_Business_Value	0	0	0	0
Last_Reporting_Month	2020-12-01	2020-04-01	2019-07-01	2020-05-01
Age	25	28	29	38
Gender	0	0	0	1
Income	49439	56498	25873	47818
Total_Income	49439	56498	25873	47818
Education_Level	2	0	0	0
City	C11	C24	C15	C8
Joining_Designation	1	2	1	2
Grade	1	2	1	2
Quarterly_Rating	1	1	1	1
Last_Working_Date	2020-12-18 00:00:00	2020-12-04 00:00:00	2019-06-30 00:00:00	2020-05-15 00:00:00
Churned	1	1	1	1
Tenure	0	0	0	0
Date_Of_Joining_month	12	12	6	5
Date_Of_Joining_year	2020	2020	2019	2020
Is_Valuable_Driver	0	0	0	0

Observations

- We can see that there are some drivers who left on same day

Outlier Check

In [ ]:
```
numerical_cols = ['Total_Business_Value', 'Avg_Business_Value', 'Age','Income', 'Total_Income', 'Tenure']

fig, ax = plt.subplots(2, 3, figsize=(15, 8))
for i, col in enumerate(numerical_cols):
    sns.boxplot(merged_df[col], ax=ax[i//3, i%3])

plt.tight_layout();
```



Observations

- From above plot we can see there are some outliers in total business value and average business value

Univariate Analysis

In [ ]:
```
merged_df["Churned"].value_counts()
merged_df["Gender"].value_counts()
merged_df["Education_Level"].value_counts()
merged_df["Joining_Designation"].value_counts()
merged_df["Has_Income_Increased"].value_counts()
merged_df["Has_Rating_Increased"].value_counts()
merged_df["Is_Valuable_Driver"].value_counts()

merged_df.groupby(["Gender", "Churned"]).size()
```

Out[ ]:
```
Churned
1    1616
0     765
Name: count, dtype: int64
```

Out[ ]:
```
Gender
0    1405
1     976
Name: count, dtype: int64
```

Out[ ]:
```
Education_Level
2    802
1    795
0    784
Name: count, dtype: int64
```

Out[ ]:
```
Joining_Designation
1    1026
2     815
3     493
4      36
5      11
Name: count, dtype: int64
```

Out[ ]:
```
Has_Income_Increased
0    2337
1      44
Name: count, dtype: int64
```

Out[ ]:
```
Has_Rating_Increased
0    1682
1     699
Name: count, dtype: int64
```

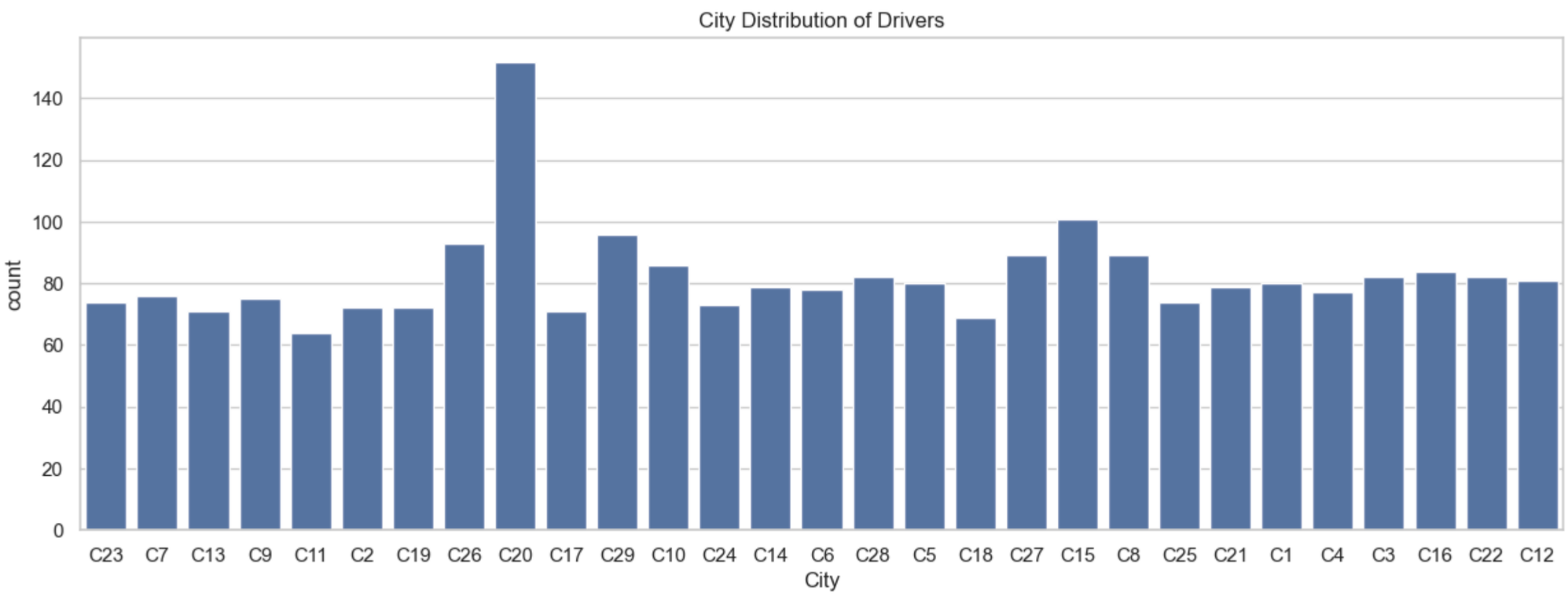
Out[ ]:
```
Is_Valuable_Driver
1    1526
0     855
Name: count, dtype: int64
```

Out[ ]:
```
Gender  Churned
0       0          456
        1          949
1       0          309
        1          667
dtype: int64
```

Observations

- In the dataset there are 1616 churned drivers
- There are 1400 male and 976 female drivers
- Almost all educations levels of drivers aew same
- Most drivers joined at 1 designation
- We can see that only 44 drivers have increased their income
- Out of all the 2381 drivers only 699 drivers has increased their rating
- Out of all the drivers 44 drivers have increased their grade.
- 667 female drivers have churned

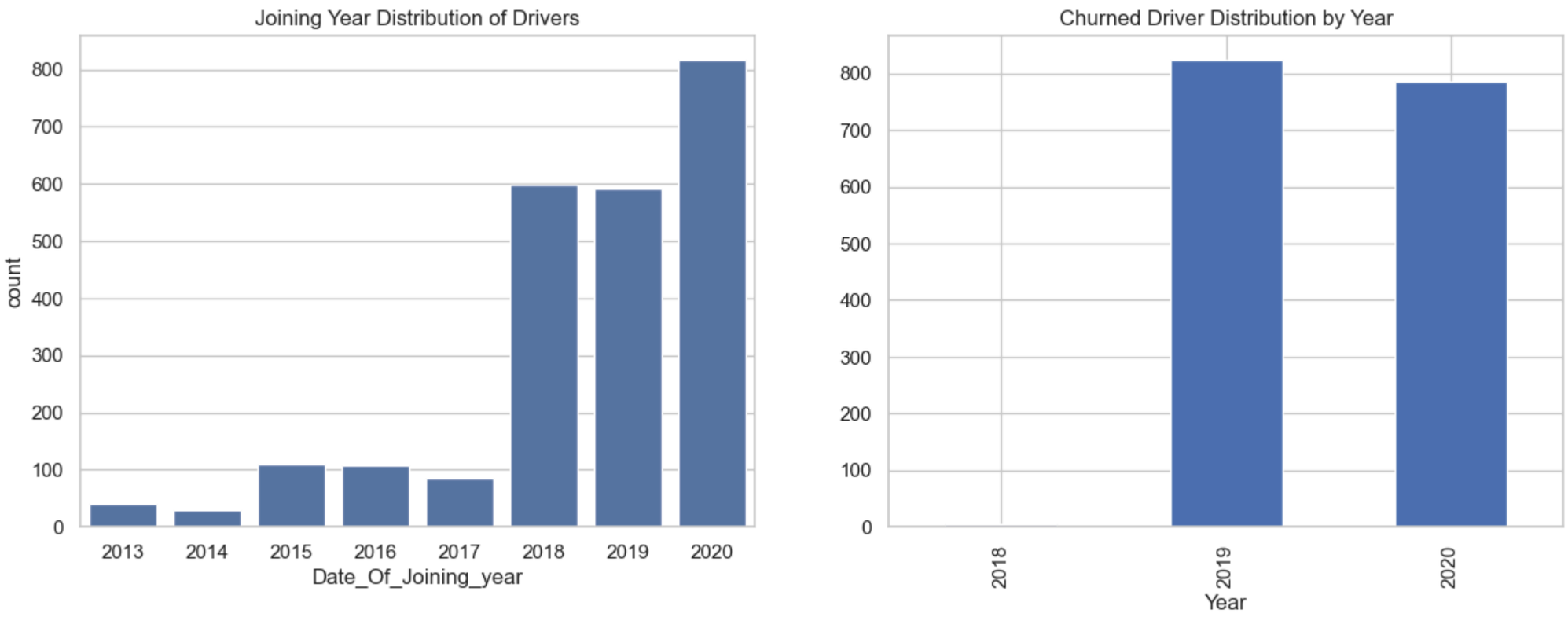
In [ ]:
```
plt.figure(figsize=(15, 5))
sns.countplot(data=merged_df, x="City");
plt.title("City Distribution of Drivers");
```



Observations

- We can see that C20 has the highest number of drivers

In [ ]:
```
fig , ax = plt.subplots(1, 2, figsize=(15, 5))
sns.countplot(x="Date_Of_Joining_year", data=merged_df, ax=ax[0]);
merged_df[merged_df["Churned"]==1]["Last_Working_Date"].dt.year.value_counts().sort_index().plot(kind="bar", ax=ax[1]);
ax[0].set_title("Joining Year Distribution of Drivers");
ax[1].set_title("Churned Driver Distribution by Year");
ax[1].set_xlabel("Year");
```

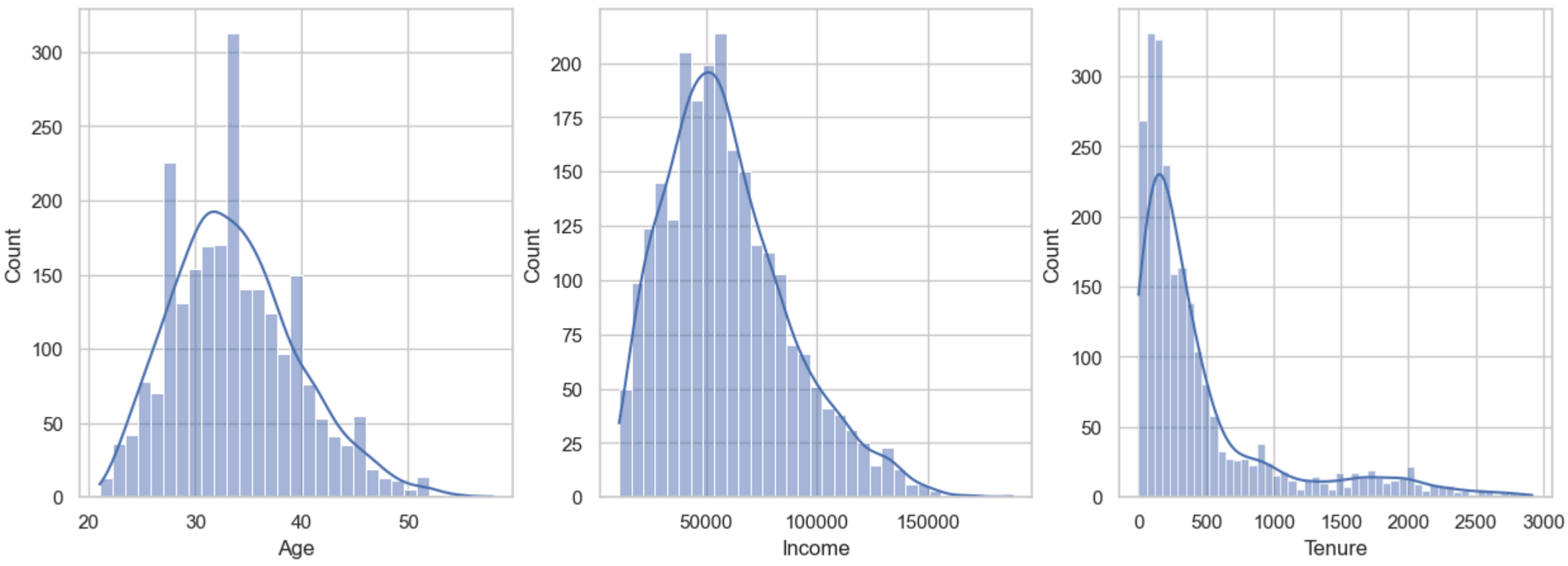


Observations

- We can see that there was large number of drivers who joined during 2018, and large exit of drivers during 2019

Normality and Skewness Check

In [ ]:
```
fig, ax = plt.subplots(1, 3, figsize=(15, 5))

sns.histplot(merged_df["Age"], kde=True, ax=ax[0])
sns.histplot(merged_df["Income"], kde=True, ax=ax[1])
sns.histplot(merged_df["Tenure"], kde=True, ax=ax[2]);
```

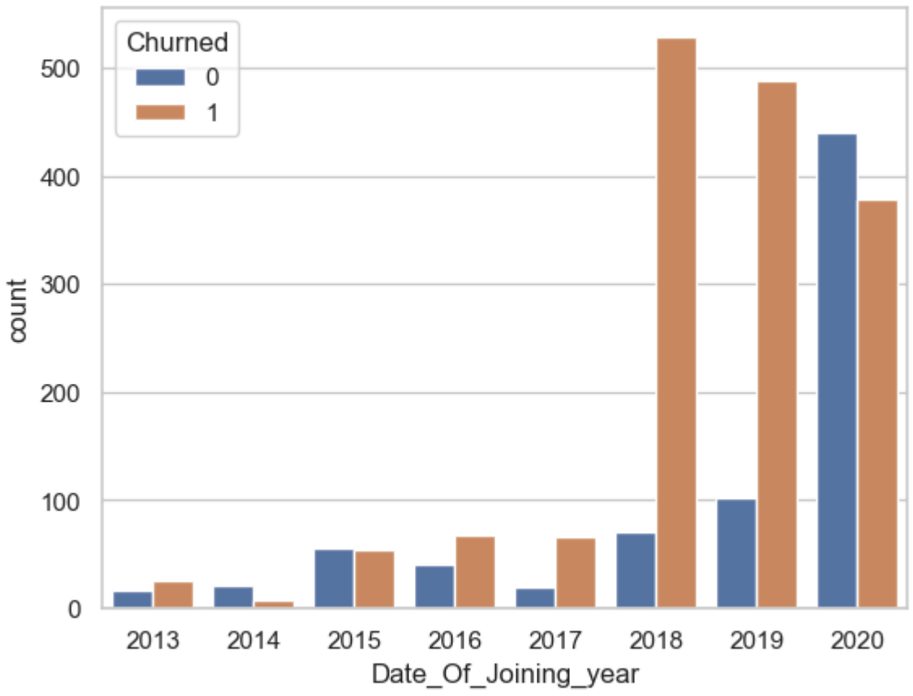


Observations

- Above plots shows the distribution of Age, Income and Tenure.

Bivariate Analysis

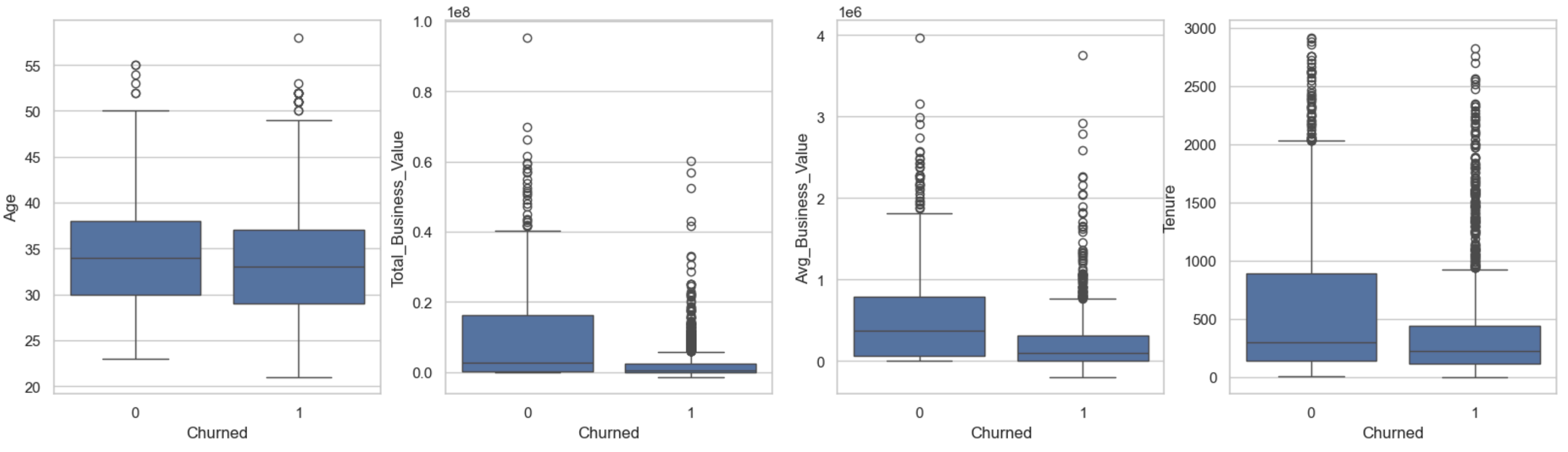
In [ ]:
```
sns.countplot(x="Date_Of_Joining_year", data=merged_df, hue="Churned");
```



Observations

- From plot we can see the churned and hired driver count during each year
- We can see that there was big churn for drivers who joined in 2018, 2019 and 2020

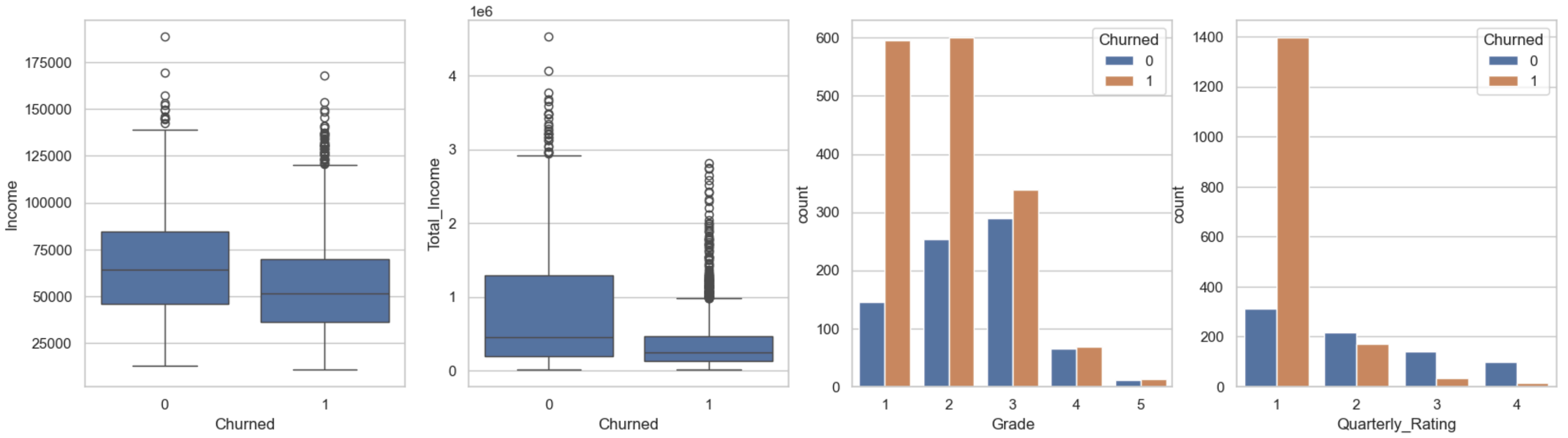
In [ ]:
```
fig, ax = plt.subplots(1, 4, figsize=(20, 5))

sns.boxplot(data=merged_df, x="Churned", y="Age", ax=ax[0]);
sns.boxplot(data=merged_df, x="Churned", y="Total_Business_Value", ax=ax[1]);
sns.boxplot(data=merged_df, x="Churned", y="Avg_Business_Value", ax=ax[2]);
sns.boxplot(data=merged_df, x="Churned", y="Tenure", ax=ax[3]);
# plt.tight_layout();
```



Observations

- Above plots show how different features affect the target variable
- We can see the Tenure, average business value and total business value is less for churned drivers.

In [ ]:
```
fig, ax = plt.subplots(1, 4, figsize=(20, 5))
sns.boxplot(data=merged_df, x="Churned", y="Income", ax=ax[0]);
sns.boxplot(data=merged_df, x="Churned", y="Total_Income", ax=ax[1]);
sns.countplot(data=merged_df, x="Grade", hue="Churned", ax=ax[2]);
sns.countplot(data=merged_df, x="Quarterly_Rating", hue="Churned", ax=ax[3]);
```

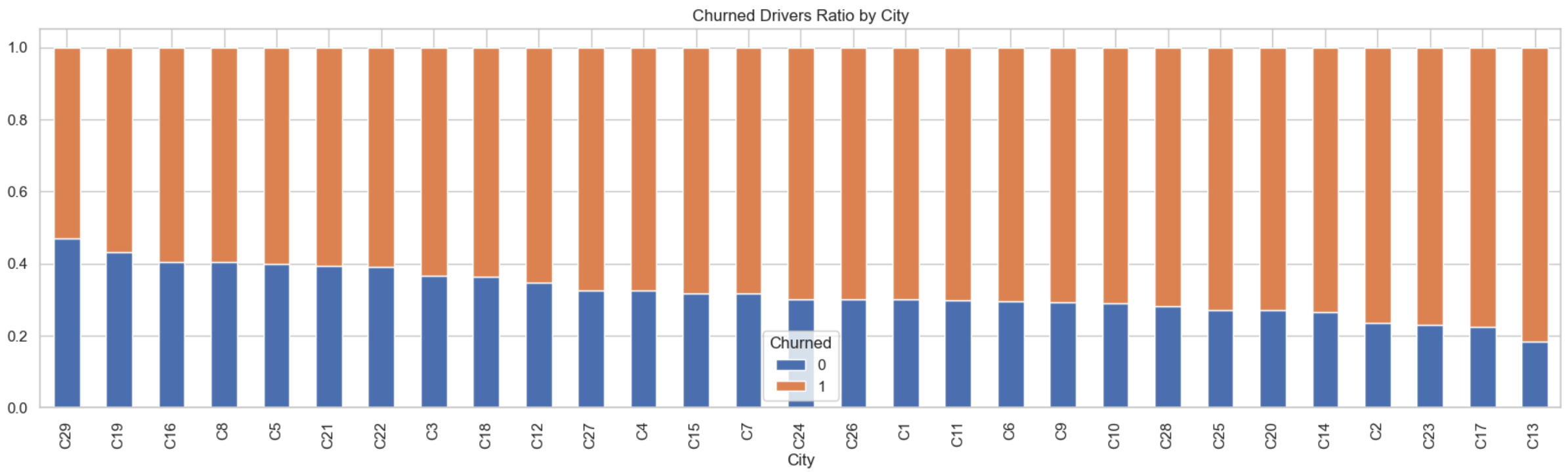


Observations

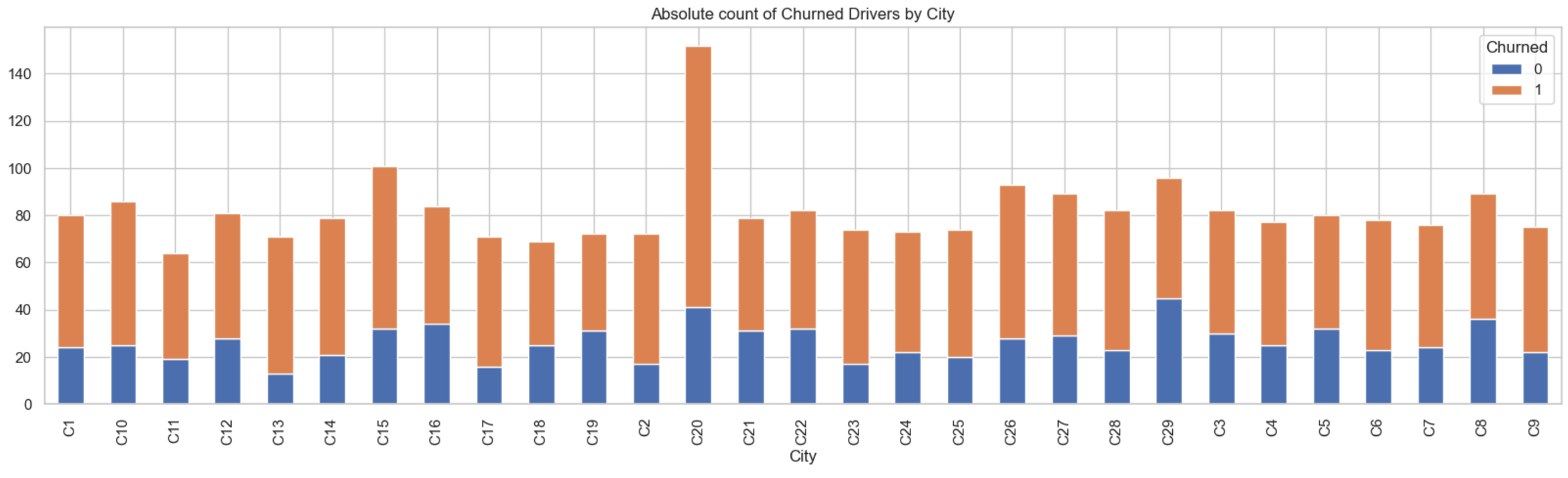
- We can see that Income is less for churned drivers as compared to current drivers.
- Low Income may be responsible for driver churn.
- There are very few Grade 4 and 5 employees as compared to Grade 1, 2 and 3.
- Majority of the drivers churning belong to Grade 1 and 2.
- Majority of the drivers who Churned had 1 rating.

Ratio of Churned drivers by City

In [ ]:
```
pd.crosstab(merged_df["City"], merged_df["Churned"],  normalize="index").sort_values(1).plot(kind="bar", stacked=True, figsize=(20, 5));
plt.title("Churned Drivers Ratio by City");
```



In [ ]:
```
pd.crosstab(merged_df["City"], merged_df["Churned"]).plot(kind="bar", stacked=True, figsize=(20, 5));
plt.title("Absolute count of Churned Drivers by City");
```

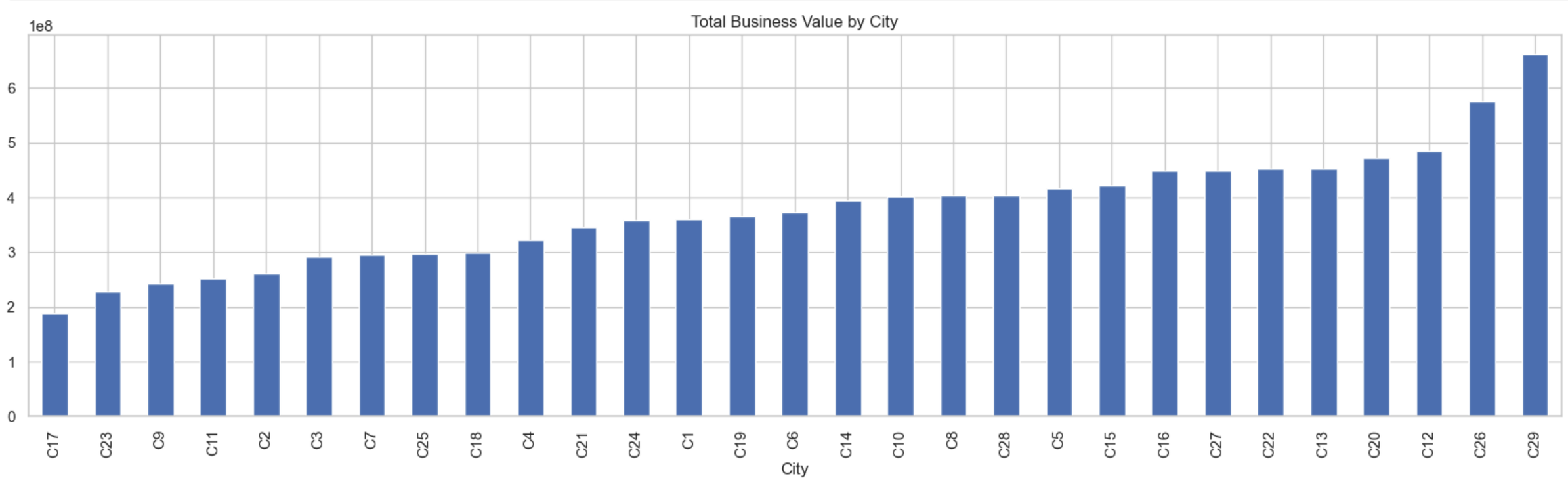


Observations

- Above plot shows the ratio of churned drivers in each city
- C13 have highest ratio of Churned drivers
- C29 have the lowest ratio of Churned drivers
- C20 had highest number of Churned drivers

Revenue Distribution by City

In [ ]:
```
merged_df.groupby("City")["Total_Business_Value"].sum().sort_values().plot(kind="bar", figsize=(20, 5));
plt.title("Total Business Value by City");
```

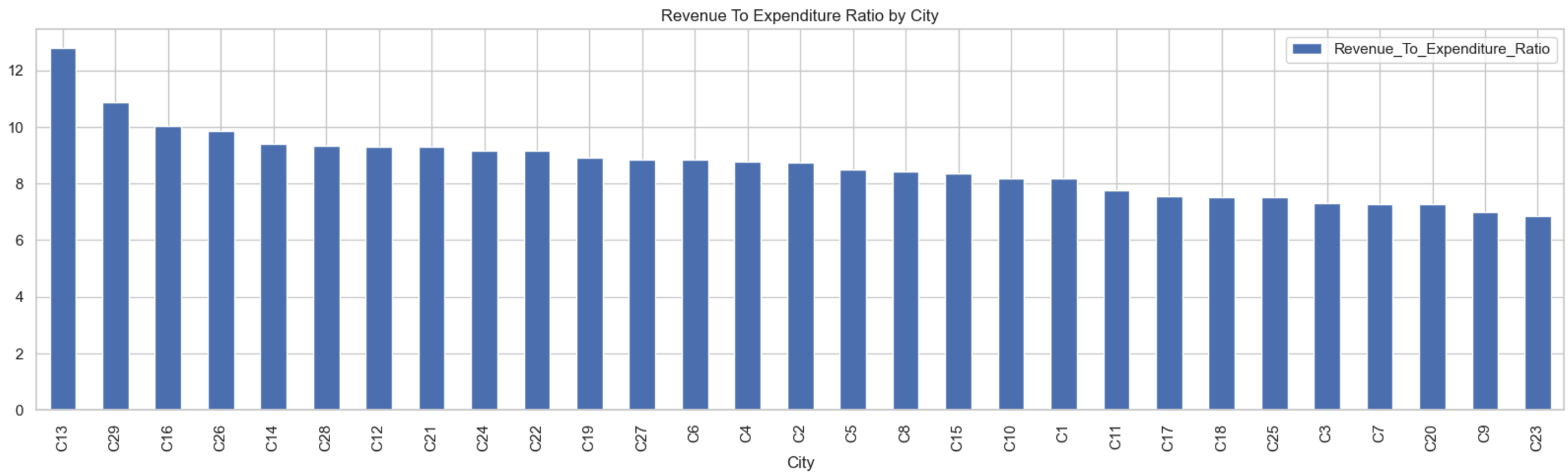


Observations

- From above plot we can see that C29 makes the highest revenue

Revenue_To_Expenditure_Ratio by City

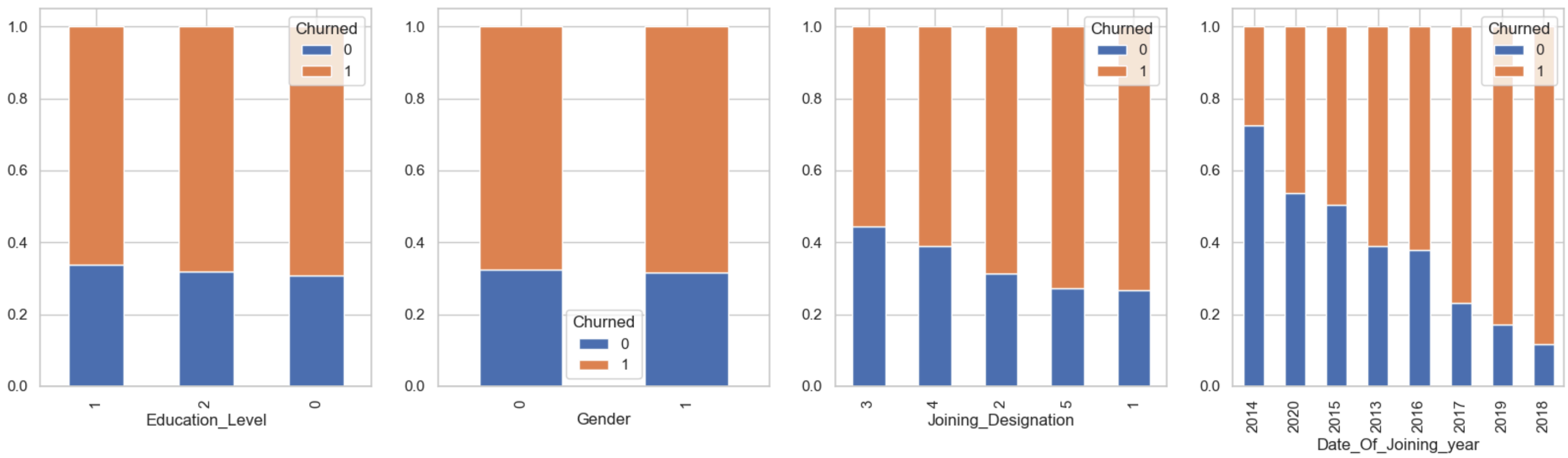
In [ ]:
```
spark.sql("""
        select
          City
          , round((sum(Total_Business_Value) / sum(Total_Income)),2) as Revenue_To_Expenditure_Ratio
          from ola_driver_merged
          group by City
          order by Revenue_To_Expenditure_Ratio desc
 """).toPandas().plot(kind="bar", x="City", y="Revenue_To_Expenditure_Ratio", figsize=(20, 5))
plt.title("Revenue To Expenditure Ratio by City")
```



Observations

- Above plot shows that C13 city has the best revenue to expense ratio

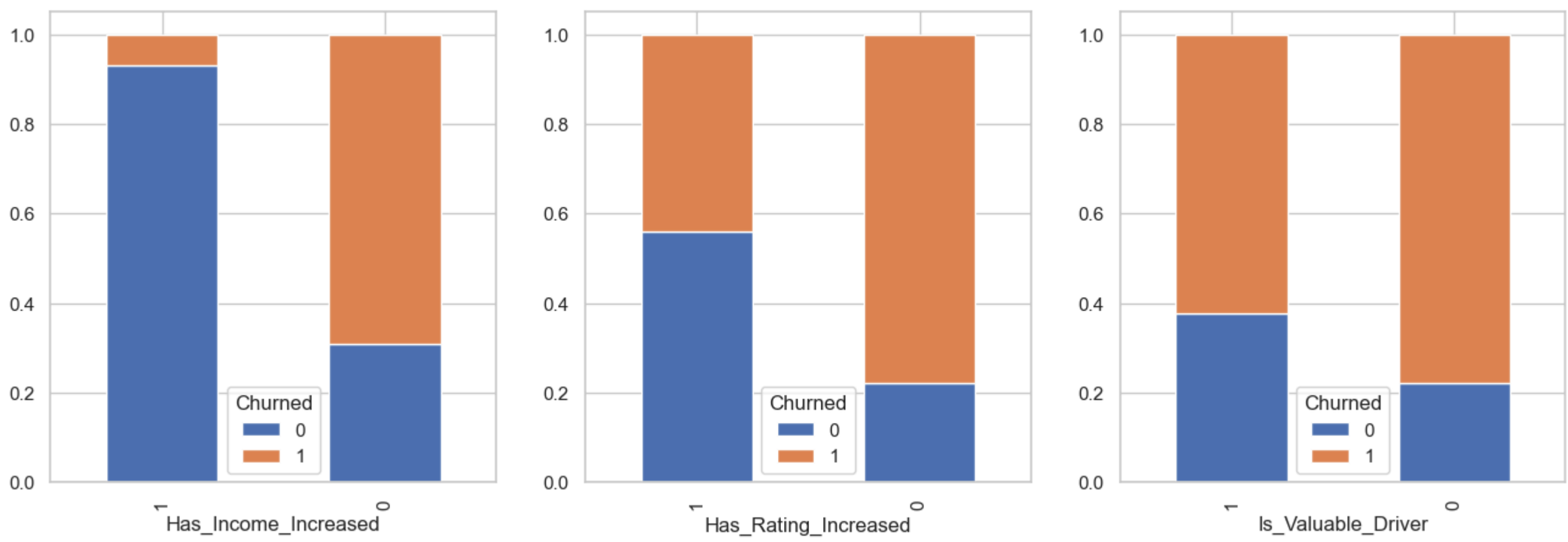
In [ ]:
```
fig, ax = plt.subplots(1, 4, figsize=(20, 5))

pd.crosstab(merged_df["Education_Level"], merged_df["Churned"],  normalize="index").sort_values(1).plot(kind="bar", stacked=True, ax=ax[0]);
pd.crosstab(merged_df["Gender"], merged_df["Churned"],  normalize="index").sort_values(1).plot(kind="bar", stacked=True, ax=ax[1]);
pd.crosstab(merged_df["Joining_Designation"], merged_df["Churned"],  normalize="index").sort_values(1).plot(kind="bar", stacked=True, ax=ax[2]);
pd.crosstab(merged_df["Date_Of_Joining_year"], merged_df["Churned"],  normalize="index").sort_values(1).plot(kind="bar", stacked=True, ax=ax[3]);
```



Observations

- We can see that Gender and Education level donot have significant effect on Churn
- From above plot we can see that Joining designation and Joining year have significant effect on churn

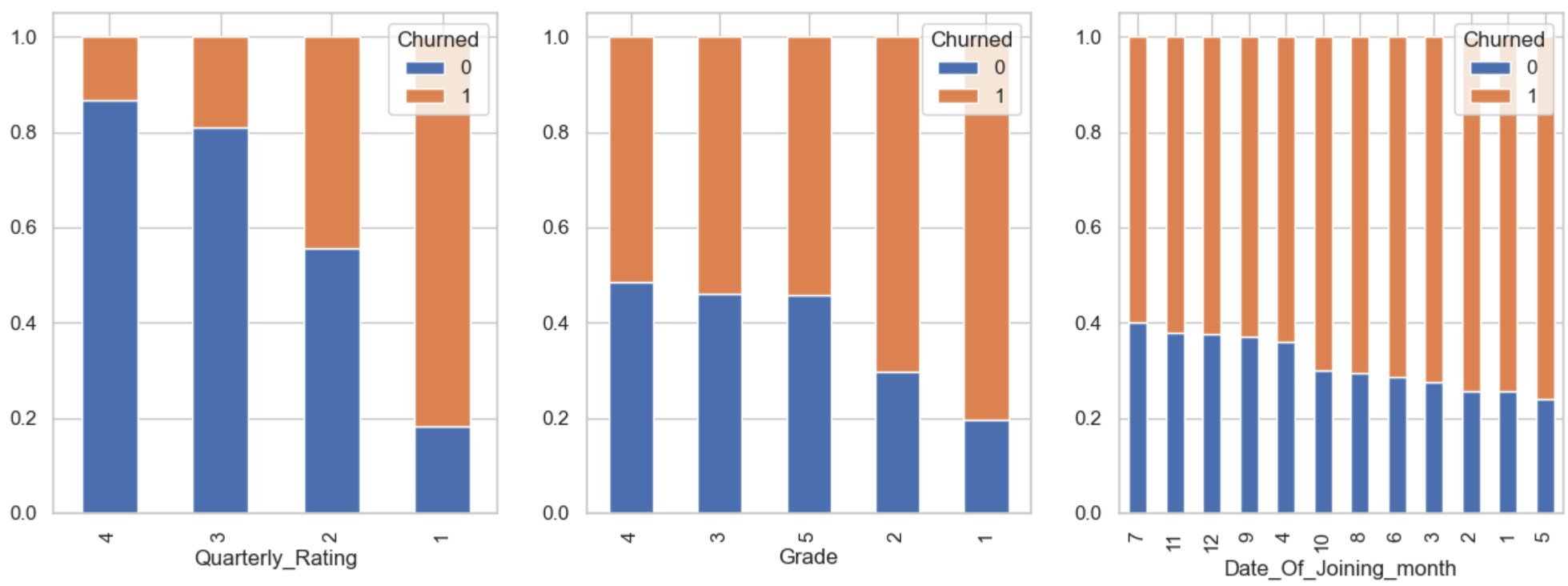
In [ ]:
```
fig, ax = plt.subplots(1, 3, figsize=(16, 5))

pd.crosstab(merged_df["Has_Income_Increased"], merged_df["Churned"],  normalize="index").sort_values(1).plot(kind="bar", stacked=True, ax=ax[0]);
pd.crosstab(merged_df["Has_Rating_Increased"], merged_df["Churned"],  normalize="index").sort_values(1).plot(kind="bar", stacked=True, ax=ax[1]);
pd.crosstab(merged_df["Is_Valuable_Driver"], merged_df["Churned"],  normalize="index").sort_values(1).plot(kind="bar", stacked=True, ax=ax[2]);
```



Observations

- From above plot we can see that their is higher ratio of churned drivers when salary was not increased
- From above plot we can see that their is higher ratio of churned drivers when ratings were not increased

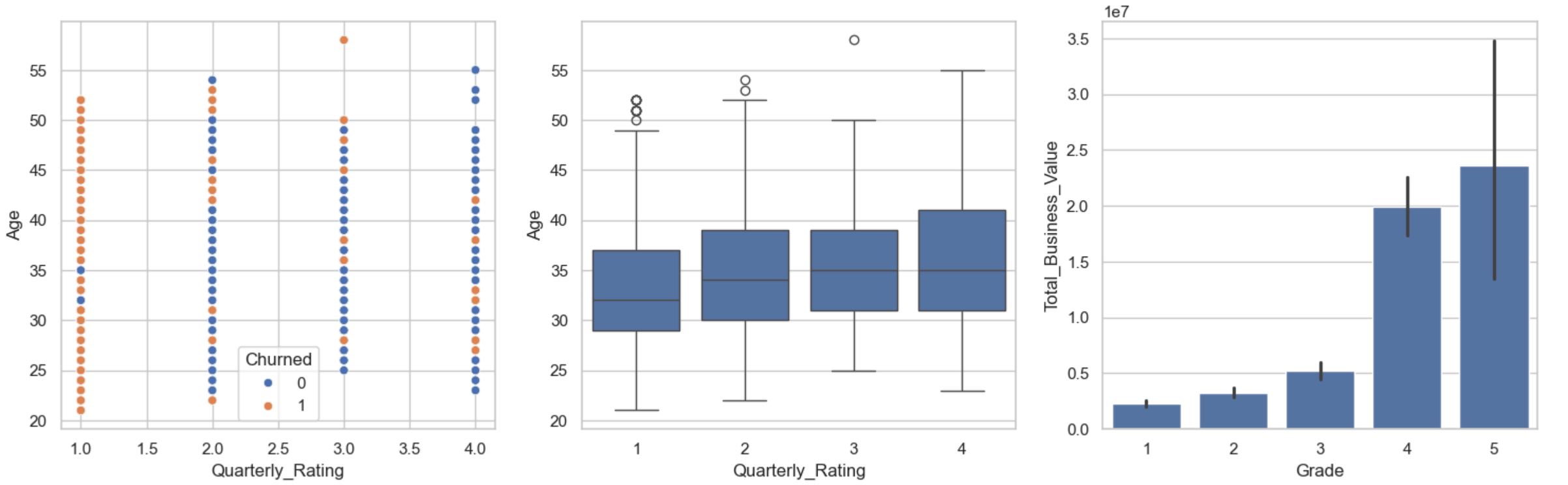
In [ ]:
```
fig, ax = plt.subplots(1, 3, figsize=(15, 5))

pd.crosstab(merged_df["Quarterly_Rating"], merged_df["Churned"],  normalize="index").sort_values(1).plot(kind="bar", stacked=True, ax=ax[0]);
pd.crosstab(merged_df["Grade"], merged_df["Churned"],  normalize="index").sort_values(1).plot(kind="bar", stacked=True, ax=ax[1]);
pd.crosstab(merged_df["Date_Of_Joining_month"], merged_df["Churned"],  normalize="index").sort_values(1).plot(kind="bar", stacked=True, ax=ax[2]);
```



Observations

- From above plot we can see that none of the drivers got Quarterly rating as 5
- We can also see that Churn increases as driver rating decreases

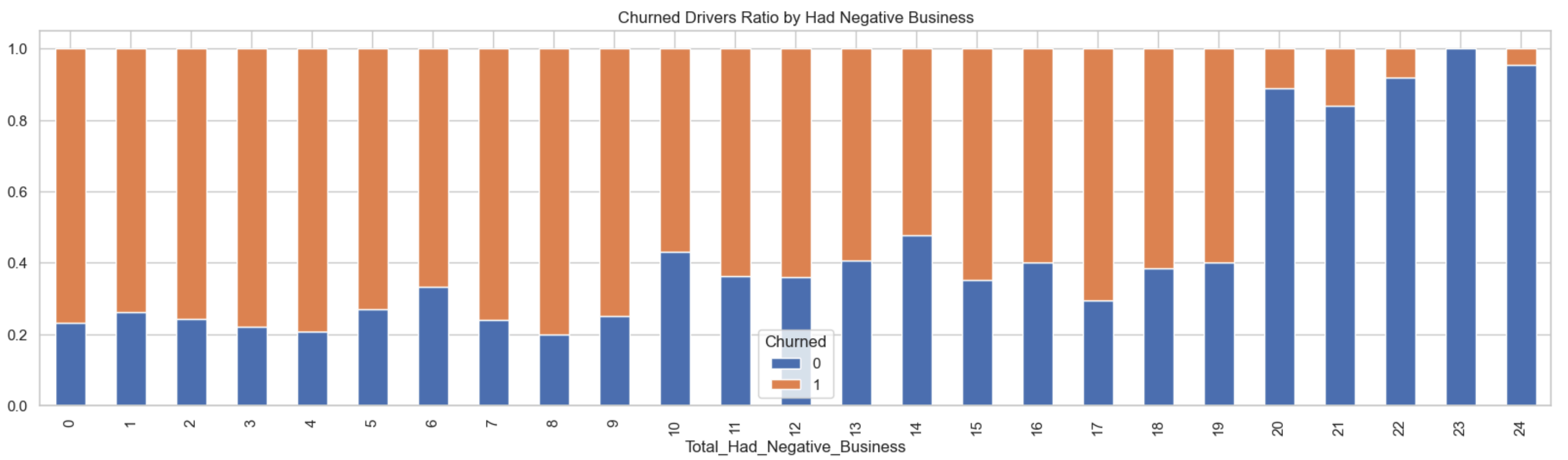
In [ ]:
```
fig, ax = plt.subplots(1, 3, figsize=(18, 5))
sns.scatterplot(data=merged_df, y="Age", x="Quarterly_Rating", hue="Churned", ax=ax[0]);
sns.boxplot(data=merged_df, y="Age", x="Quarterly_Rating", ax=ax[1]);
sns.barplot(data=merged_df, y="Total_Business_Value", x="Grade",  ax=ax[2]);
```



Observations

- From above plot we can see that there is some correlation between Age and Quaterly Rating.
- As Age increases, Quarterly ratings move towards higher side.
- From above plot we can see that driver with higher grade drivers have have business value

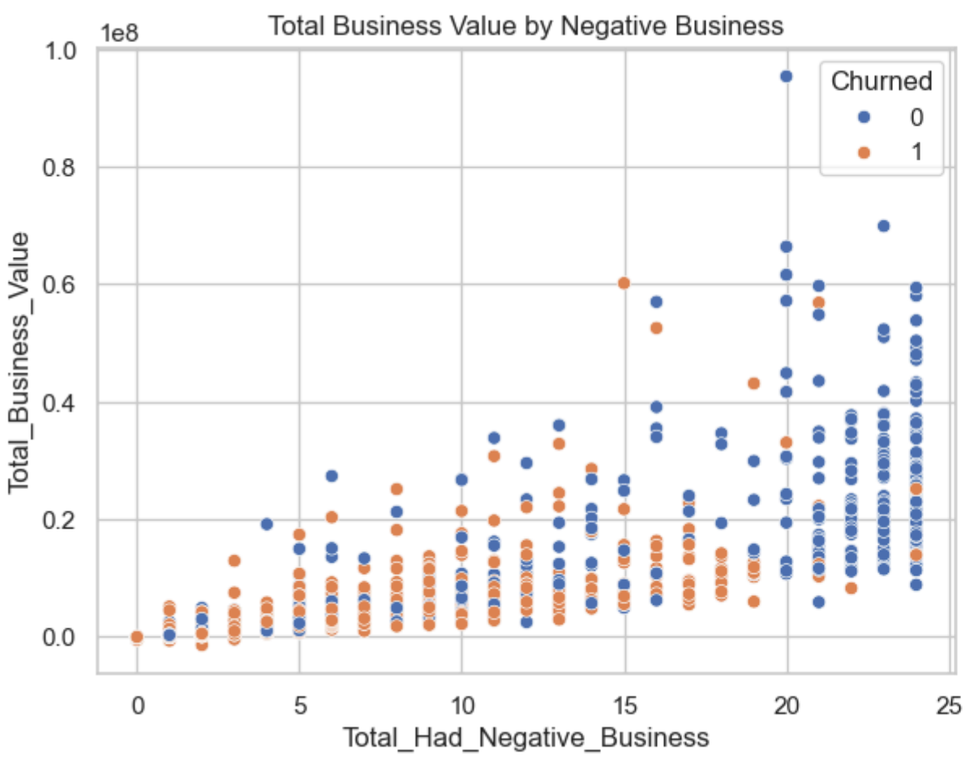
In [ ]:
```
pd.crosstab(merged_df["Total_Had_Negative_Business"], merged_df["Churned"],  normalize="index").plot(kind="bar", stacked=True, figsize=(20, 5));
plt.title("Churned Drivers Ratio by Had Negative Business");
```

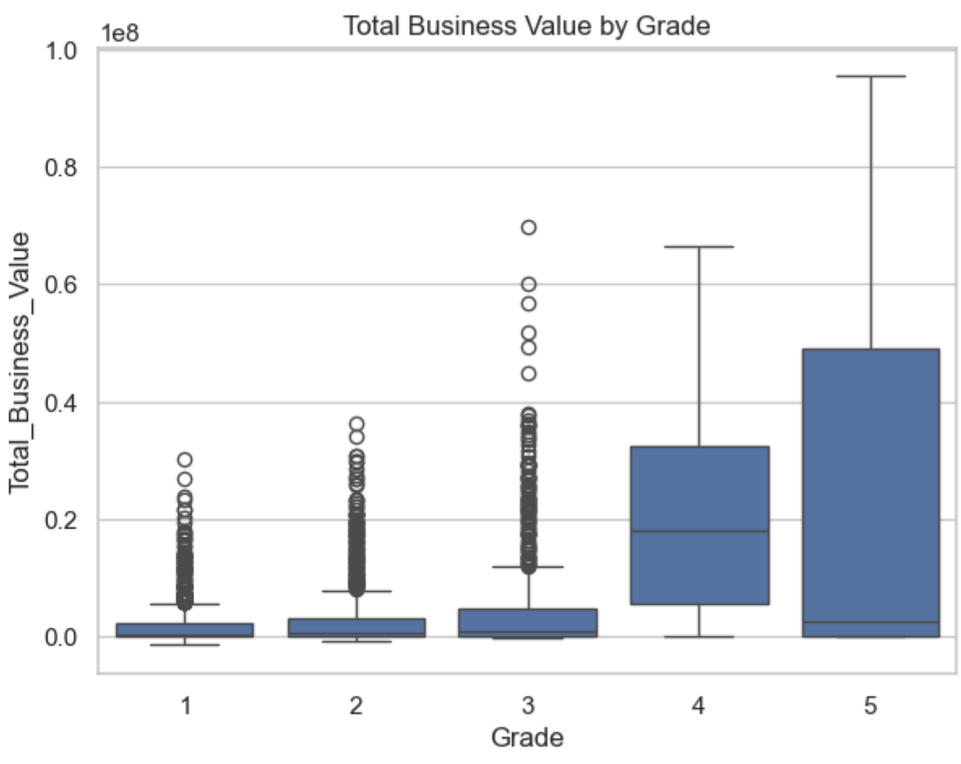


Observations

- We can see that Drivers who have negative business months are less likely to churn

In [ ]:
```
fig, ax = plt.subplots(1, 2, figsize=(15, 5))
sns.scatterplot(data=merged_df, x="Total_Had_Negative_Business", y="Total_Business_Value", hue="Churned", ax=ax[0]);
sns.boxplot(data=merged_df, x="Grade", y="Total_Business_Value", ax=ax[1]);
ax[0].set_title("Total Business Value by Negative Business");
ax[1].set_title("Total Business Value by Grade");
```





Observations

- from above plot we can see that drivers who had higher months negative business didnt churn
- From above plot we can see that Higher grade drivers have larger business value

Change in ratings for different cities

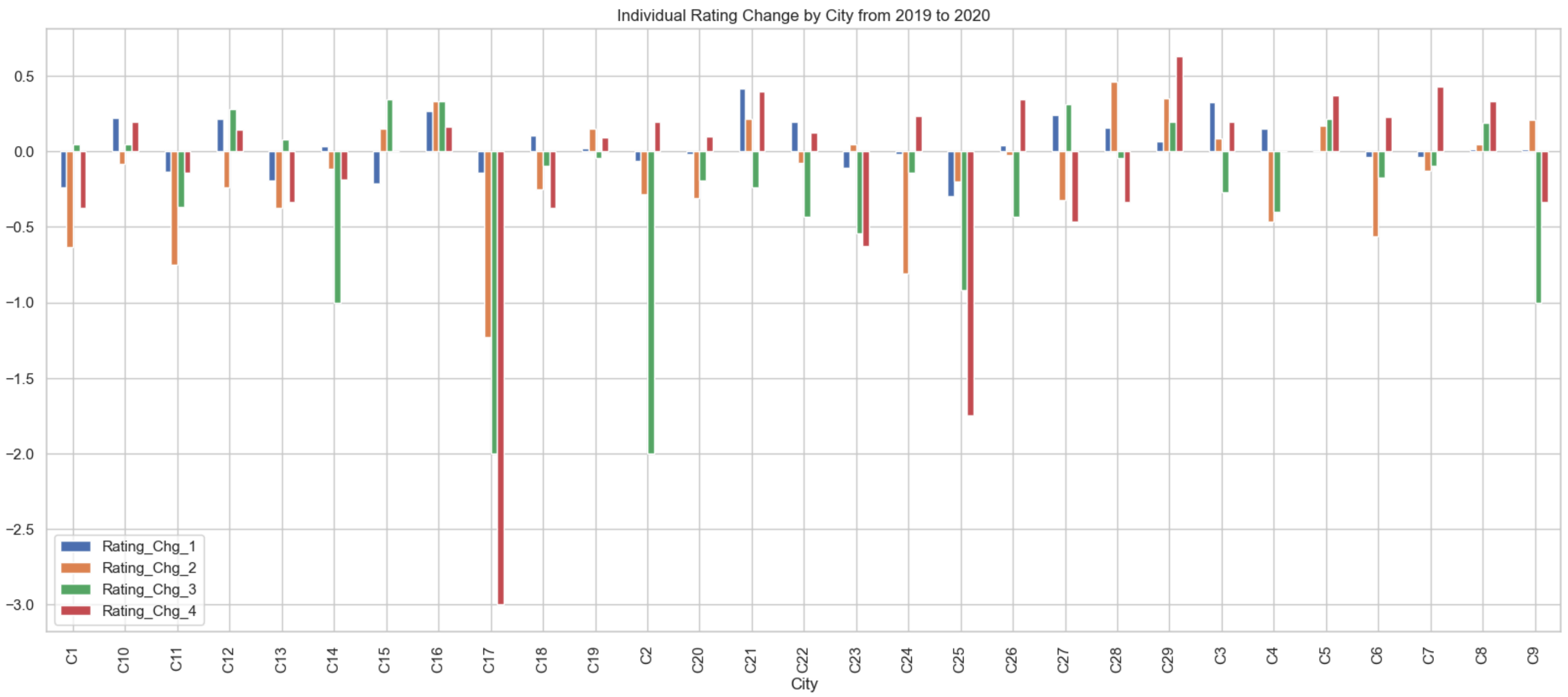
In [ ]:
```
rating_df = spark.sql("""
            with Quarterly_Agg as (
                select
                    Driver_ID,
                    City,
                    ReportingYear_Quarter,
                    case when max(`Quarterly Rating`) = 1 then 1 else 0 end as rating_1_flag,
                    case when max(`Quarterly Rating`) = 2 then 1 else 0 end as rating_2_flag,
                    case when max(`Quarterly Rating`) = 3 then 1 else 0 end as rating_3_flag,
                    case when max(`Quarterly Rating`) = 4 then 1 else 0 end as rating_4_flag,
                    ReportingYear
                from
                    ola_driver
                group by
                    ReportingYear_Quarter,
                    City,
                    Driver_ID,
                    ReportingYear
            ),
            City_Agg as (
                select
                    City,
                    ReportingYear,
                    sum(rating_1_flag) as Total_Rating_1,
                    sum(rating_2_flag) as Total_Rating_2,
                    sum(rating_3_flag) as Total_Rating_3,
                    sum(rating_4_flag) as Total_Rating_4
                from
                    Quarterly_Agg
                group by
                    City,
                    ReportingYear
            )
            select
                City,
                round((Total_Rating_1 - lag(Total_Rating_1) over(partition by City order by ReportingYear))/ Total_Rating_1,3) as Rating_Chg_1,
                round((Total_Rating_2 - lag(Total_Rating_2) over(partition by City order by ReportingYear))/ Total_Rating_2,3) as Rating_Chg_2,
                round((Total_Rating_3 - lag(Total_Rating_3) over(partition by City order by ReportingYear))/ Total_Rating_3,3) as Rating_Chg_3,
                round((Total_Rating_4 - lag(Total_Rating_4) over(partition by City order by ReportingYear))/ Total_Rating_4,3) as Rating_Chg_4
            from
                City_Agg
            order by
                City,
                ReportingYear
          """).toPandas().dropna()
rating_df.head()
```

Out[ ]:

	City	Rating_Chg_1	Rating_Chg_2	Rating_Chg_3	Rating_Chg_4
1	C1	-0.235	-0.633	0.045	-0.375
3	C10	0.222	-0.085	0.050	0.200
5	C11	-0.132	-0.750	-0.364	-0.143
7	C12	0.214	-0.237	0.280	0.143
9	C13	-0.192	-0.375	0.080	-0.333

In [ ]:
```
plt.figure(figsize=(20, 8))
rating_df.plot(kind="bar", x="City", y=["Rating_Chg_1", "Rating_Chg_2", "Rating_Chg_3", "Rating_Chg_4"], figsize=(20, 8))
plt.title("Individual Rating Change by City from 2019 to 2020");
```

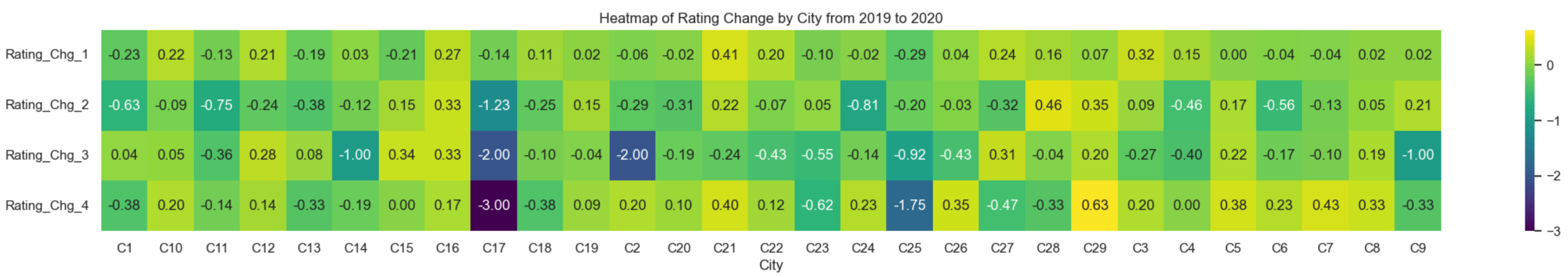
```
<Figure size 2000x800 with 0 Axes>
```



Observations

- From above plots we can see that C29 had the highest positive change in 4 star rating
- C17 had the biggest fall of all type of rating
- C2, C14, C9 had big fall of 3 star rating

In [ ]:
```
plt.figure(figsize=(25, 3))
sns.heatmap(rating_df.set_index("City").T, cmap="viridis", annot=True, fmt=".2f");
plt.title("Heatmap of Rating Change by City from 2019 to 2020");
```



Observations

- From above plots we can see that C29 had the highest change in 4 star rating
- C17 had the biggest fall of all type of rating

Effect on business value when ratings decrease

In [ ]:
```
business_decrease_df= spark.sql("""
        with Lower_Rating_Drivers as (
            select
                distinct Driver_ID
            from
                (select
                    Driver_ID,
                    first(`Quarterly Rating`) over(partition by Driver_ID order by reporting_month_year) as first_rating,
                    last(`Quarterly Rating`) over(partition by Driver_ID order by reporting_month_year) as last_rating
                from
                    ola_driver
                )
            where
                first_rating > last_rating
        ),
        Quarterly_Business_Value as (
            select
                Driver_ID,
                ReportingYear_Quarter,
                sum(`Total Business Value`) over( partition by Driver_ID, ReportingYear_Quarter) as Total_Business_Value
            from
                ola_driver
                join Lower_Rating_Drivers using(Driver_ID)
            --where `Total Business Value` > 0
        )
        select
            Driver_ID,
            first(Total_Business_Value) as First_Business_Value,
            last(Total_Business_Value) as Last_Business_Value,
            (Last_Business_Value - First_Business_Value)/ First_Business_Value as Business_Value_Change,
            case when Last_Business_Value < First_Business_Value then 1 else 0 end as Business_Decrease
        from
            Quarterly_Business_Value
        group by
            Driver_ID
""").toPandas()
business_decrease_df.head()
```

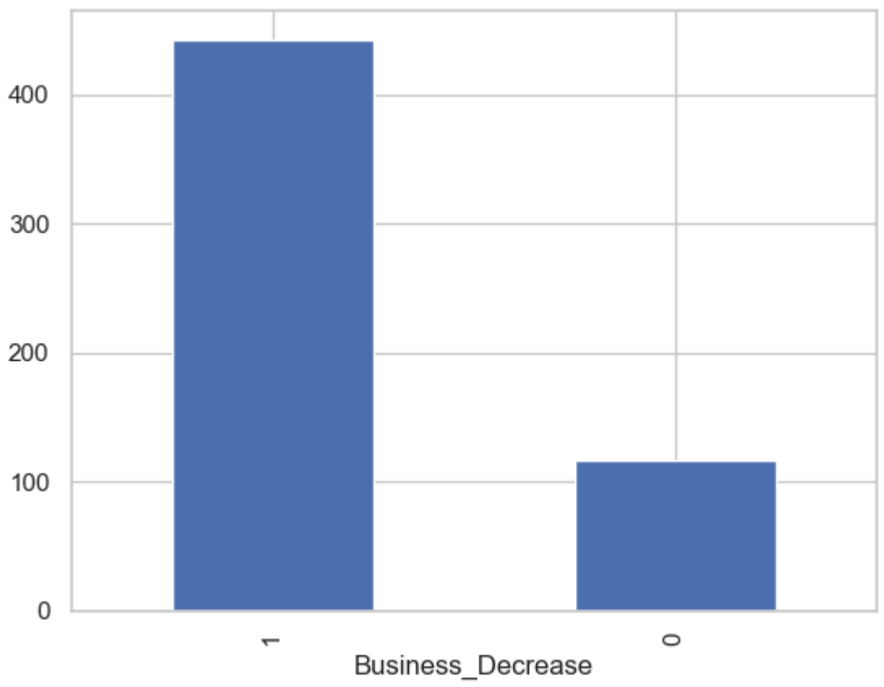
Out[ ]:

	Driver_ID	First_Business_Value	Last_Business_Value	Business_Value_Change	Business_Decrease
0	12	2607180	0	-1.000000	1
1	17	774990	0	-1.000000	1
2	21	4795670	431550	-0.910013	1
3	22	1440730	0	-1.000000	1
4	26	18541380	3776080	-0.796343	1

In [ ]:
```
business_decrease_df["Business_Decrease"].value_counts()
```

Out[ ]:
```
Business_Decrease
1    540
0     19
Name: count, dtype: int64
```

In [ ]:
```
business_decrease_df["Business_Decrease"].value_counts().plot(kind="bar");
```



Observations

- From above plot we can see that out of 559 drivers whose rating decreased, 540 drivers business decreases significantly.
- This shows that Driver rating has significant impact on business

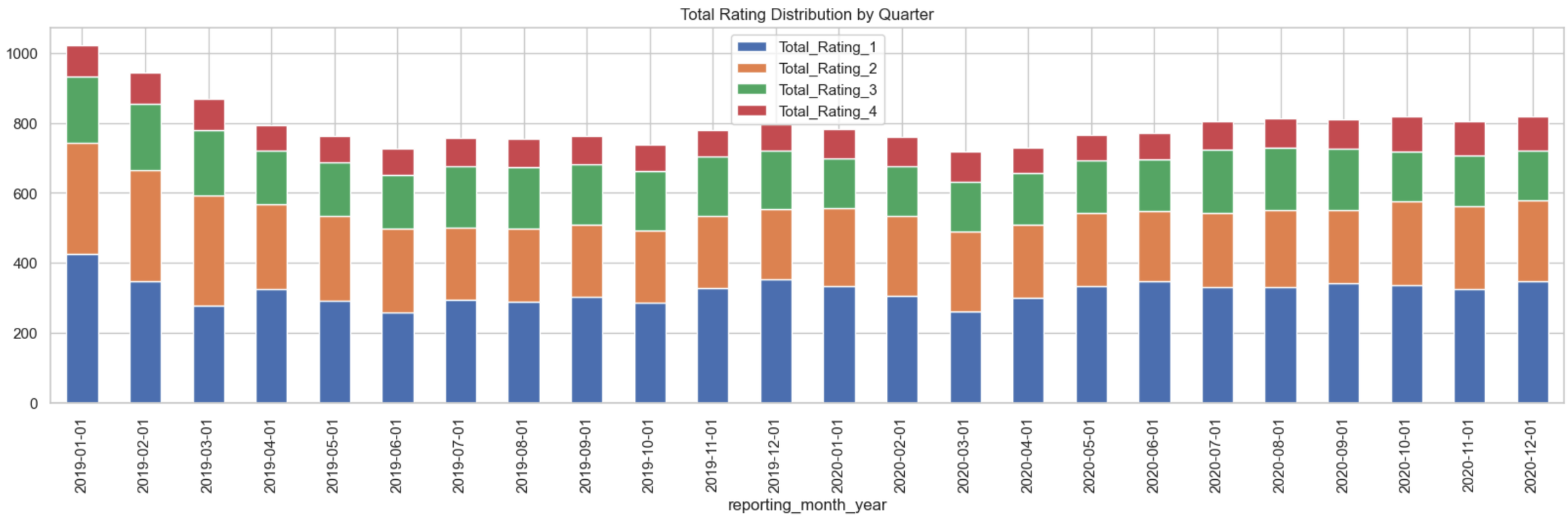
Effect of rating based on month of year

In [ ]:
```
rating_df = spark.sql("""
            with Quarterly_Agg as (
            select
                Driver_ID,
                ReportingYear_Quarter,
                case when max(`Quarterly Rating`) = 1 then 1 else 0 end as rating_1_flag,
                case when max(`Quarterly Rating`) = 2 then 1 else 0 end as rating_2_flag,
                case when max(`Quarterly Rating`) = 3 then 1 else 0 end as rating_3_flag,
                case when max(`Quarterly Rating`) = 4 then 1 else 0 end as rating_4_flag,
                reporting_month_year
            from
                ola_driver
            group by
                ReportingYear_Quarter,
                Driver_ID,
                reporting_month_year
            )
            select
                reporting_month_year,
                sum(rating_1_flag) as Total_Rating_1,
                sum(rating_2_flag) as Total_Rating_2,
                sum(rating_3_flag) as Total_Rating_3,
                sum(rating_4_flag) as Total_Rating_4
            from
                Quarterly_Agg
            group by
                reporting_month_year
            order by
                reporting_month_year
          """).toPandas()
rating_df.head()
```

Out[ ]:

	reporting_month_year	Total_Rating_1	Total_Rating_2	Total_Rating_3	Total_Rating_4
0	2019-01-01	426	318	188	90
1	2019-02-01	348	318	188	90
2	2019-03-01	279	315	187	89
3	2019-04-01	325	242	154	73
4	2019-05-01	292	243	154	75

In [ ]:
```
rating_df.plot(kind="bar", x="reporting_month_year", y=["Total_Rating_1", "Total_Rating_2", "Total_Rating_3", "Total_Rating_4"], figsize=(20, 5), stacked=True);
plt.title("Total Rating Distribution by Month");
```



Observations

- From above plot we can see that demand increases during November to January, but falls slightly during other months.
- This is because of the holiday season. Since the numbers of rides increases, the corresponding ratings also increase

Effect of Ratings based on City

In [ ]:
```
rating_df = spark.sql("""
            with Quarterly_Agg as (
             select
                Driver_ID,
                City,
```

```
                ReportingYear_Quarter,
                case when max(`Quarterly Rating`) = 1 then 1 else 0 end as rating_1_flag,
                case when max(`Quarterly Rating`) = 2 then 1 else 0 end as rating_2_flag,
                case when max(`Quarterly Rating`) = 3 then 1 else 0 end as rating_3_flag,
                case when max(`Quarterly Rating`) = 4 then 1 else 0 end as rating_4_flag
            from
                ola_driver
            group by
                ReportingYear_Quarter,
                City,
                Driver_ID
            )
            select
                City,
                sum(rating_1_flag) as Total_Rating_1,
                sum(rating_2_flag) as Total_Rating_2,
                sum(rating_3_flag) as Total_Rating_3,
                sum(rating_4_flag) as Total_Rating_4
            from
                Quarterly_Agg
            group by
                City
          """).toPandas()
rating_df.head()
```

Out[ ]:

	City	Total_Rating_1	Total_Rating_2	Total_Rating_3	Total_Rating_4
0	C23	120	41	28	21
1	C7	110	66	44	11
2	C13	114	38	48	21
3	C9	113	59	24	14
4	C11	81	55	26	15

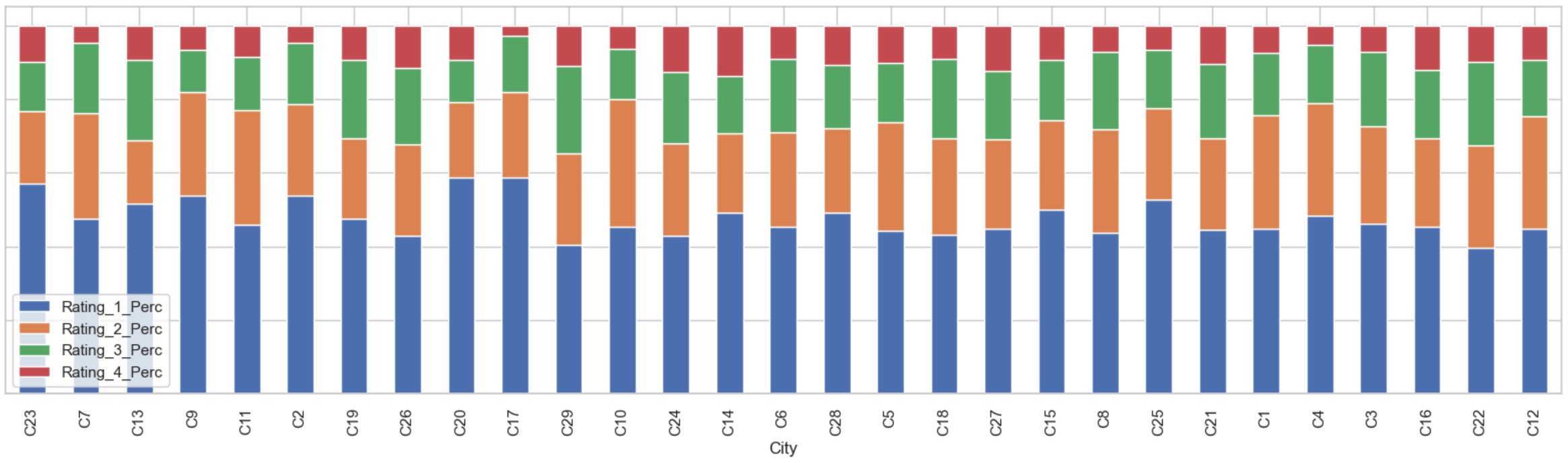
In [ ]:
```
rating_df['Total'] = rating_df[['Total_Rating_1', 'Total_Rating_2', 'Total_Rating_3', 'Total_Rating_4']].sum(axis=1)

rating_df['Rating_1_Perc'] = (rating_df['Total_Rating_1'] / rating_df['Total']) * 100
rating_df['Rating_2_Perc'] = (rating_df['Total_Rating_2'] / rating_df['Total']) * 100
rating_df['Rating_3_Perc'] = (rating_df['Total_Rating_3'] / rating_df['Total']) * 100
rating_df['Rating_4_Perc'] = (rating_df['Total_Rating_4'] / rating_df['Total']) * 100

rating_df = rating_df.drop(columns=['Total'])
rating_df.head()
```

Out[ ]:

	City	Total_Rating_1	Total_Rating_2	Total_Rating_3	Total_Rating_4	Rating_1_Perc	Rating_2_Perc	Rating_3_Perc	Rating_4_Perc
0	C23	120	41	28	21	57.142857	19.523810	13.333333	10.000000
1	C7	110	66	44	11	47.619048	28.571429	19.047619	4.761905
2	C13	114	38	48	21	51.583710	17.194570	21.719457	9.502262
3	C9	113	59	24	14	53.809524	28.095238	11.428571	6.666667
4	C11	81	55	26	15	45.762712	31.073446	14.689266	8.474576

In [ ]:
```
plt.figure(figsize=(20, 5))
rating_df.reset_index().plot(kind="bar", x="City", y=["Rating_1_Perc", "Rating_2_Perc", "Rating_3_Perc", "Rating_4_Perc"], figsize=(20, 5), stacked=True);
```

```
<Figure size 2000x500 with 0 Axes>
```

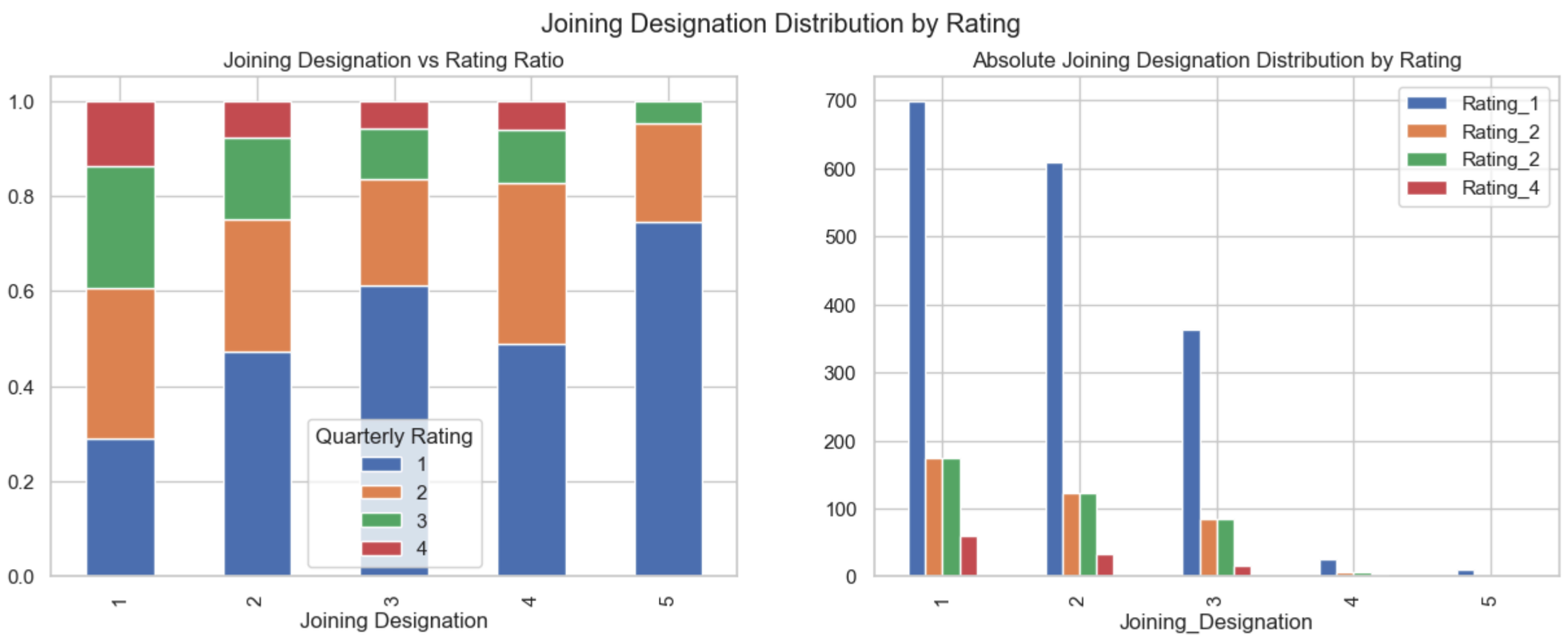


Observations

- from above plot we can see that C17 has lowest 4 star rating and highest 1 star.
- C14,C16 ans C24 have good pct of 4 star rating among other cities

Effect of Joining Designation to Quarterly Rating

In [ ]:
```
fig, ax = plt.subplots(1, 2, figsize=(15, 5))
pd.crosstab(ola_pdf["Joining Designation"], ola_pdf["Quarterly Rating"], normalize="index").plot(kind="bar", stacked=True, ax=ax[0]);

spark.sql("""
    select
        Joining_Designation,
        sum(case when Quarterly_Rating = 1 then 1 else 0 end) as Rating_1,
        sum(case when Quarterly_Rating = 2 then 1 else 0 end) as Rating_2,
        sum(case when Quarterly_Rating = 3 then 1 else 0 end) as Rating_3,
        sum(case when Quarterly_Rating = 4 then 1 else 0 end) as Rating_4
    from
          ola_driver_merged
    group by
          Joining_Designation
""").toPandas().plot(kind="bar", x="Joining_Designation", y=["Rating_1","Rating_2","Rating_2","Rating_4" ], ax=ax[1]);
ax[0].set_title("Joining Designation vs Rating Ratio");
ax[1].set_title("Absolute Joining Designation Distribution by Rating");
plt.suptitle("Joining Designation Distribution by Rating");
```



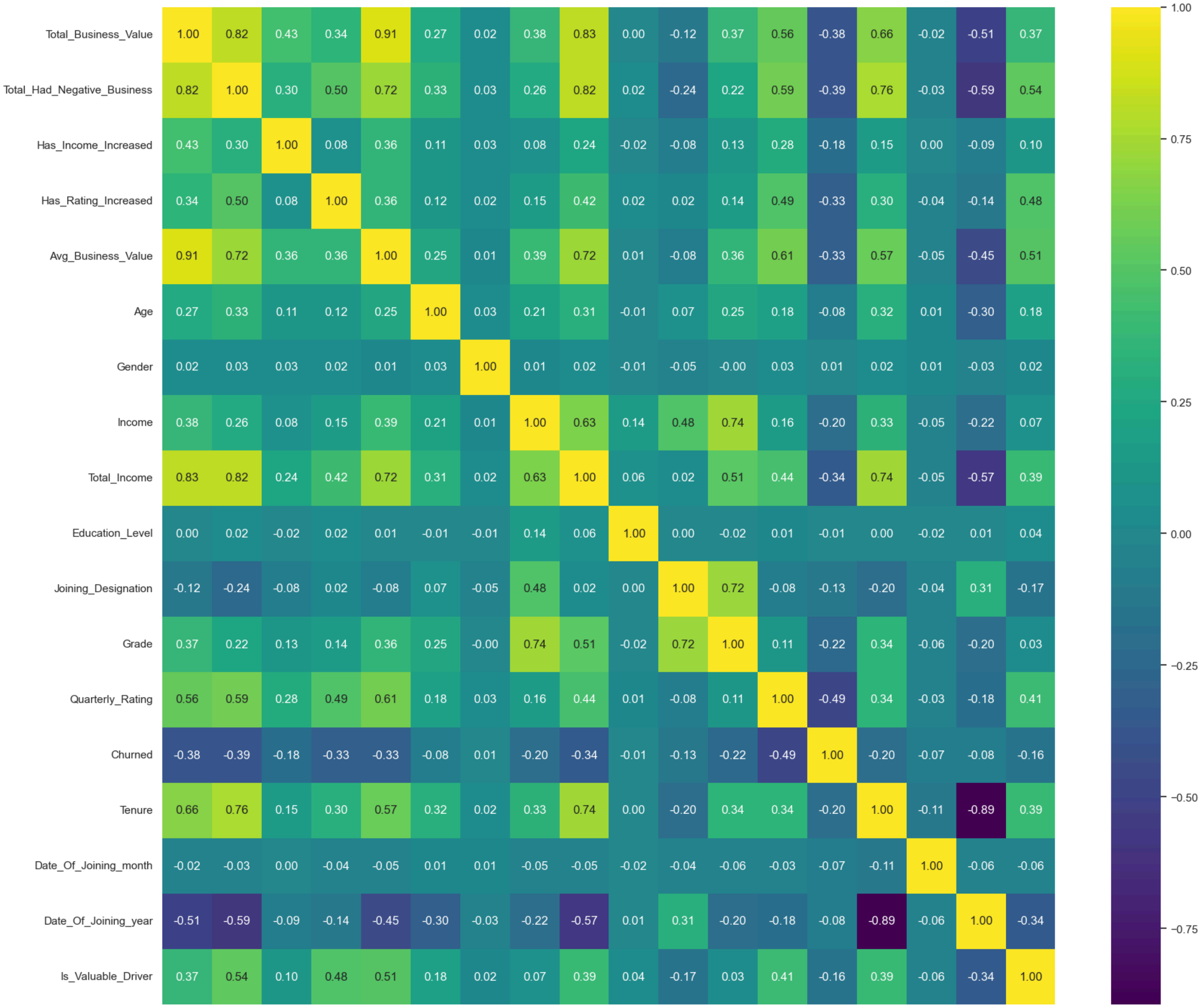
Observations

- From above plot we can see that drivers joining at higher designation have higher ratio of lower rating.
- But their absolute rating count is lesser as compared to lower designation.

Correlation Plot

In [ ]:
```
merged_df_corr = merged_df.select_dtypes(include=[np.number]).drop("Driver_ID", axis=1).corr()
```

In [ ]:
```
plt.figure(figsize=(20, 18))
sns.heatmap(merged_df_corr, annot=True, cmap="viridis", fmt=".2f");
```



Observations

- We can see that Tenure is correlated with Joining Year
- Average business value is correlated with total business value

Model Building

In [ ]:
```
from pyspark.ml.feature import StringIndexer, VectorAssembler, OneHotEncoder, StandardScaler
from pyspark.ml.classification import RandomForestClassifier, GBTClassifier, RandomForestClassificationModel, GBTClassificationModel
from pyspark.ml.evaluation import MulticlassClassificationEvaluator
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder, CrossValidatorModel, TrainValidationSplit, TrainValidationSplitModel
from pyspark.ml import Model
from pyspark.sql import DataFrame
from typing import Callable, Tuple
import mlflow
import mlflow.spark
from sklearn.metrics import precision_recall_curve, classification_report, confusion_matrix, roc_curve, roc_auc_score, auc, f1_score, accuracy_score, precision_score, recall_score
from xgboost.spark import SparkXGBClassifier, SparkXGBClassifierModel

from synapse.ml.lightgbm import LightGBMClassifier, LightGBMClassificationModel
```

In [ ]:
```
mlflow.set_tracking_uri("sqlite:///mlflow.db")
mlflow.set_experiment("ola_driver_churn_v3")

mlflow.pyspark.ml.autolog(
   log_models=False,
   log_input_examples=False,
   exclusive=False,
   silent=True,
   disable=True
);
```

Helper functions

In [ ]:
```
palette = sns.color_palette("deep", 8)  # You can choose other palettes like "deep", "muted", "bright", etc.

def print_metrics(y_true, y_pred, y_prob):
    print(classification_report(y_true, y_pred))
    print("Accuracy: ", accuracy_score(y_true, y_pred))
    print("F1 Score: ", f1_score(y_true, y_pred))
    print("AUC: ", roc_auc_score(y_true, y_prob))

def plot_confusion_matrix(y_true, y_pred, normalize='true'):
    cm = confusion_matrix(y_true, y_pred, normalize=normalize, labels=[0, 1])
    plt.figure(figsize=(6, 5))
    sns.heatmap(cm, annot=True, cmap="viridis", xticklabels=[0, 1], yticklabels=[0, 1]);
    plt.title('Confusion Matrix')
    plt.xlabel('Predicted')
    plt.ylabel('Actual');


def plot_roc_curve(y_true, y_prob, label="Model", color='orange', show_random=True):
    fpr, tpr, threshold = roc_curve(y_true, y_prob, pos_label=1)

    if show_random:
        random_probs = [0 for i in range(len(y_true))]
        p_fpr, p_tpr, _ = roc_curve(y_true, random_probs, pos_label=1)
        plt.plot(p_fpr, p_tpr, linestyle='--', color='blue', label='Random Model')
    plt.plot(fpr, tpr, linestyle='--', color=color, label=label)
    plt.title('ROC curve')
    plt.xlabel('False Positive Rate')
    plt.ylabel('True Positive Rate')
    plt.legend()


def plot_precision_recall_curve(y_true, y_prob, label="Model", color='orange'):
    precision, recall, thr = precision_recall_curve(y_true,y_prob)
    plt.plot(recall, precision, color=color, label=label)

    start, end = plt.xlim()
    plt.xticks(np.round(np.arange(start, end, 0.1), 2))

    plt.xlabel('Recall Value')
    plt.ylabel('Precision Value')
    plt.title("PR Curve")
    plt.legend()

def plot_precision_recall_threshold_curve(y_true, y_prob):
    precision, recall, thr = precision_recall_curve(y_true,y_prob)
    threshold_boundary = thr.shape[0]

    plt.plot(thr, precision[0:threshold_boundary], linestyle='--', label='precision')
    plt.plot(thr, recall[0:threshold_boundary], label='recall')

    start, end = plt.xlim()
    plt.xticks(np.round(np.arange(start, end, 0.1), 2))

    plt.xlabel('Threshold Value')
    plt.ylabel('Precision Recall Value')
    plt.legend()
    plt.title("PR - Threshold Curve")
    plt.show();

def get_feature_importance(
    model: Model,
    spark_df: DataFrame,
    model_type: str = "pyspark"
) -> Tuple[Callable[[], None], pd.DataFrame]:
    """
    Generates a feature importance DataFrame and a lazy plot function for a given model and DataFrame.

    Parameters:
    model : pyspark.ml.Model
        The trained model from which to extract feature importances.
    spark_df : pyspark.sql.DataFrame
        The DataFrame containing the features used in the model. The DataFrame
        should have a schema with metadata that includes feature names and indices.
    model_type : str
        The type of the model ("pyspark", "lightgbm", or "xgboost").

    Returns:
    tuple
        A tuple containing:
        - plot (function): A function that, when called, generates and displays the feature importance plot.
        - pandas_df (pd.DataFrame): A Pandas DataFrame containing the feature names and their corresponding importance scores, sorted in descending order of importance.
    """

    feature_names = sorted(
        spark_df.schema["features"].metadata["ml_attr"]["attrs"]["binary"] +
        spark_df.schema["features"].metadata["ml_attr"]["attrs"]["numeric"],
        key=lambda x: x["idx"]
    )

    if model_type == "pyspark":
        importances = model.featureImportances
    elif model_type == "lightgbm":
        importances = model.getFeatureImportances()
    elif model_type == "xgboost":
        feature_imp_obj = [{"idx": int(k[1:]), "Importance": v} for k, v in model.get_feature_importances().items()]
        pandas_df = pd.DataFrame(feature_names).merge(
```

```
            pd.DataFrame(feature_imp_obj), on="idx", how="left"
        ).rename(columns={'name': "Feature"}).fillna(0).sort_values("Importance", ascending=False).drop("idx", axis=1)

        def plot() -> None:
            pandas_df.plot(kind="bar", x="Feature", y="Importance", figsize=(20, 5))

        return plot, pandas_df
    else:
        raise ValueError(f"Unsupported model type: {model_type}")

    feature_importance_mapping = {
        x["name"]: y for x, y in zip(feature_names, importances)
    }
    pandas_df = pd.DataFrame(
        feature_importance_mapping.items(),
        columns=["Feature", "Importance"]
    ).sort_values("Importance", ascending=False)

    def plot() -> None:
        pandas_df.plot(kind="bar", x="Feature", y="Importance", figsize=(20, 5))

    return plot, pandas_df


def spark_prediction_to_numpy(predictions: DataFrame):
    """
    Converts Spark DataFrame predictions to NumPy arrays.

    Parameters:
    predictions : pyspark.sql.DataFrame
        The Spark DataFrame containing the columns 'prediction', 'probability', and 'Churned'.

    Returns:
    tuple
        A tuple containing:
        - y_true (np.ndarray): The true labels.
        - y_pred (np.ndarray): The predicted labels.
        - y_prob (np.ndarray): The predicted probabilities.
    """
    data = predictions.select("prediction", "probability", "Churned").collect()

    y_true = np.array([row['Churned'] for row in data])
    y_pred = np.array([row['prediction'] for row in data])
    y_prob = np.array([row['probability'][1] for row in data])

    return y_true, y_pred, y_prob



def get_all_model_params(model: CrossValidatorModel, metric="f1"):
    params = [{p.name: v for p, v in m.items()} for m in model.getEstimatorParamMaps()]

    _ = pd.DataFrame.from_dict([
        {model.getEvaluator().getMetricName(): _metric, **ps}
        for ps, _metric in zip(params, model.avgMetrics)
    ])
    return _.sort_values(metric, ascending=False)

def get_all_model_params_tvs(model: TrainValidationSplitModel, metric="f1"):
    params = [{p.name: v for p, v in m.items()} for m in model.getEstimatorParamMaps()]

    _ = pd.DataFrame.from_dict([
        {model.getEvaluator().getMetricName(): _metric, **ps}
        for ps, _metric in zip(params, model.validationMetrics)
    ])
    return _.sort_values(metric, ascending=False)



def fit_model(model, paramGrid, data, use_train_validation_split=False):
    """
    Cross validates a model using the given data and parameter grid.

    Parameters:
    model : pyspark.ml.Model
        The model to be cross-validated.
    paramGrid : pyspark.ml.tuning.ParamGridBuilder
        The parameter grid to be used for cross-validation.
    data : pyspark.sql.DataFrame
        The DataFrame containing the features and labels.
    use_train_validation_split : bool
        Whether to use TrainValidationSplit or CrossValidator.

    Returns:
    tuple
        A tuple containing:
        - cvModel (pyspark.ml.Model): The cross-validated model.
        - evaluator (pyspark.ml.evaluation.Evaluator): The evaluator used for cross-validation.
    """
    evaluator = MulticlassClassificationEvaluator(labelCol="Churned", predictionCol="prediction", metricName="f1", weightCol='classWeightCol')

    if use_train_validation_split:
        _model = TrainValidationSplit(estimator=model,
                              estimatorParamMaps=paramGrid,
                              evaluator=evaluator,
                              parallelism=16,
                              # collectSubModels=True,
                              trainRatio = 0.8,
                              seed=42)
    else:
        _model = CrossValidator(estimator=model,
                              estimatorParamMaps=paramGrid,
                              evaluator=evaluator,
                              numFolds=3,
                              parallelism=16,
                              # collectSubModels=True,
                              seed=42)
    cvModel = _model.fit(data)
    return cvModel, evaluator

# def get_best_threshold(y_true, y_prob):
#     fpr, tpr, thresholds = roc_curve(y_true, y_prob, pos_label=1)
#     gmeans = np.sqrt(tpr * (1-fpr))
#     ix = np.argmax(gmeans)
#     return thresholds[ix]

def get_best_threshold(y_true, y_prob):
    thresholds = np.arange(0.0, 1.0, 0.01)

    precisions = []
    recalls = []
    f1s = []

    for threshold in thresholds:
        y_pred = (y_prob >= threshold).astype(int)
        precisions.append(precision_score(y_true, y_pred))
        recalls.append(recall_score(y_true, y_pred))
        f1s.append(f1_score(y_true, y_pred))

    best_threshold = thresholds[np.argmax(f1s)]

    print(f'Best threshold: {best_threshold}')
    print(f'Precision at best threshold: {precisions[np.argmax(f1s)]}')
```

```
    print(f'Recall at best threshold: {recalls[np.argmax(f1s)]}')
    print(f'F1 score at best threshold: {f1s[np.argmax(f1s)]}')
    return best_threshold
```

Preprocessing

In [ ]:
```
model_df = spark.read.parquet("ola_driver_merged.parquet")
model_df.cache();
```

In [ ]:
```
model_df.limit(3).toPandas().T
```

Out[ ]:

	0	1	2
Driver_ID	1	2	4
Date_Of_Joining	2018-12-24	2020-06-11	2019-07-12
Total_Business_Value	1715580	0	350000
Total_Had_Negative_Business	1	0	1
Has_Income_Increased	0	0	0
Has_Rating_Increased	0	0	0
Avg_Business_Value	571860	0	70000
Last_Reporting_Month	2019-03-01	2020-12-01	2020-04-01
Age	28	31	43
Gender	0	0	0
Income	57387	67016	65603
Total_Income	172161	134032	328015
Education_Level	2	2	2
City	C23	C7	C13
Joining_Designation	1	2	2
Grade	1	2	2
Quarterly_Rating	2	1	1
Last_Working_Date	2019-11-03	None	2020-04-27
Churned	1	0	1
Tenure	314	203	290
Date_Of_Joining_month	12	6	7
Date_Of_Joining_year	2018	2020	2019
Is_Valuable_Driver	1	0	1

Categorical and OneHot Encoding

In [ ]:
```
categorical_columns = ["City"]
indexers = [StringIndexer(inputCol=column, outputCol=column + "Index")
            for column in categorical_columns]

for indexer in indexers:
    model_df = indexer.fit(model_df).transform(model_df)
    encoder = OneHotEncoder(inputCol=indexer.getOutputCol(),
                            outputCol=indexer.getInputCol() + "_Vec")
    model_df = encoder.fit(model_df).transform(model_df)

feature_columns = [
    'Total_Business_Value',
    'Total_Had_Negative_Business',
    'Has_Income_Increased',
    'Has_Rating_Increased',
    'Avg_Business_Value',
    'Age',
    'Gender',
    'Income',
    'Total_Income',
    'Education_Level',
    'City_Vec',
    'Joining_Designation',
    'Grade',
    'Quarterly_Rating',
    'Tenure',
    'Date_Of_Joining_month',
    'Date_Of_Joining_year',
    'Is_Valuable_Driver'
]

to_drop = ["Driver_ID", "Date_Of_Joining",
           "Last_Working_Date", "Last_Reporting_Month", "CityIndex"]
model_df = model_df.drop(*to_drop)
```

Vectorization

In [ ]:
```
assembler = VectorAssembler(inputCols=feature_columns, outputCol="features")
model_df = assembler.transform(model_df)
class_weights = model_df.groupBy("Churned").count().collect()
total_count = model_df.count()
weights = {row['Churned']: total_count / row['count'] for row in class_weights}
model_df = model_df.withColumn("classWeightCol", sf.when(model_df["Churned"] == 1, weights[1]).otherwise(weights[0]))
```

Train Test Validation Split

In [ ]:
```
(trainingData, testData) = model_df.randomSplit([0.8, 0.2], seed=42)

sc = StandardScaler(inputCol="features", outputCol="sc_features")
sc_model = sc.fit(trainingData)
trainingData = sc_model.transform(trainingData)
testData = sc_model.transform(testData)

trainingData_v2, validationData = trainingData.randomSplit([0.8, 0.2], seed=42)
```

In [ ]:
```
trainingData.write.parquet("ola_driver_training.parquet", mode='overwrite')
testData.write.parquet("ola_driver_test.parquet", mode='overwrite')
validationData.write.parquet("ola_driver_validation.parquet", mode='overwrite')
trainingData_v2.write.parquet("ola_driver_training_v2.parquet", mode='overwrite')
```

In [ ]:
```
trainingData = spark.read.parquet("ola_driver_training.parquet")
testData = spark.read.parquet("ola_driver_test.parquet")
validationData = spark.read.parquet("ola_driver_validation.parquet")
trainingData_v2 = spark.read.parquet("ola_driver_training_v2.parquet")
trainingData.cache()
testData.cache()
validationData.cache()
trainingData_v2.cache();
```

Random Forest

Hyperparamenter Tuning using Grid Search and Cross Validation

In [ ]:
```
rf = RandomForestClassifier(labelCol="Churned", featuresCol="features", weightCol="classWeightCol", seed=42)
paramGrid = ParamGridBuilder().addGrid(rf.numTrees, [200, 250, 300, 350]).addGrid(rf.maxDepth, [5, 10, 15, 20]).build()

with mlflow.start_run(run_name="Random Forest" ):
    rf_cv_model, rf_evaluator = fit_model(rf, paramGrid, trainingData)
    mlflow.log_params({
        "best_numTrees": rf_cv_model.bestModel.getNumTrees,
        "best_maxDepth": rf_cv_model.bestModel.getMaxDepth()
    })
    test_predictions = rf_cv_model.bestModel.transform(testData)

    mlflow.log_metrics({
        "best_f1": rf_evaluator.evaluate(test_predictions),
        "best_accuracy": rf_evaluator.setMetricName("accuracy").evaluate(test_predictions),
        "best_precision": rf_evaluator.setMetricName("weightedPrecision").evaluate(test_predictions),
        "best_recall": rf_evaluator.setMetricName("weightedRecall").evaluate(test_predictions)
    })
```

Get the best model

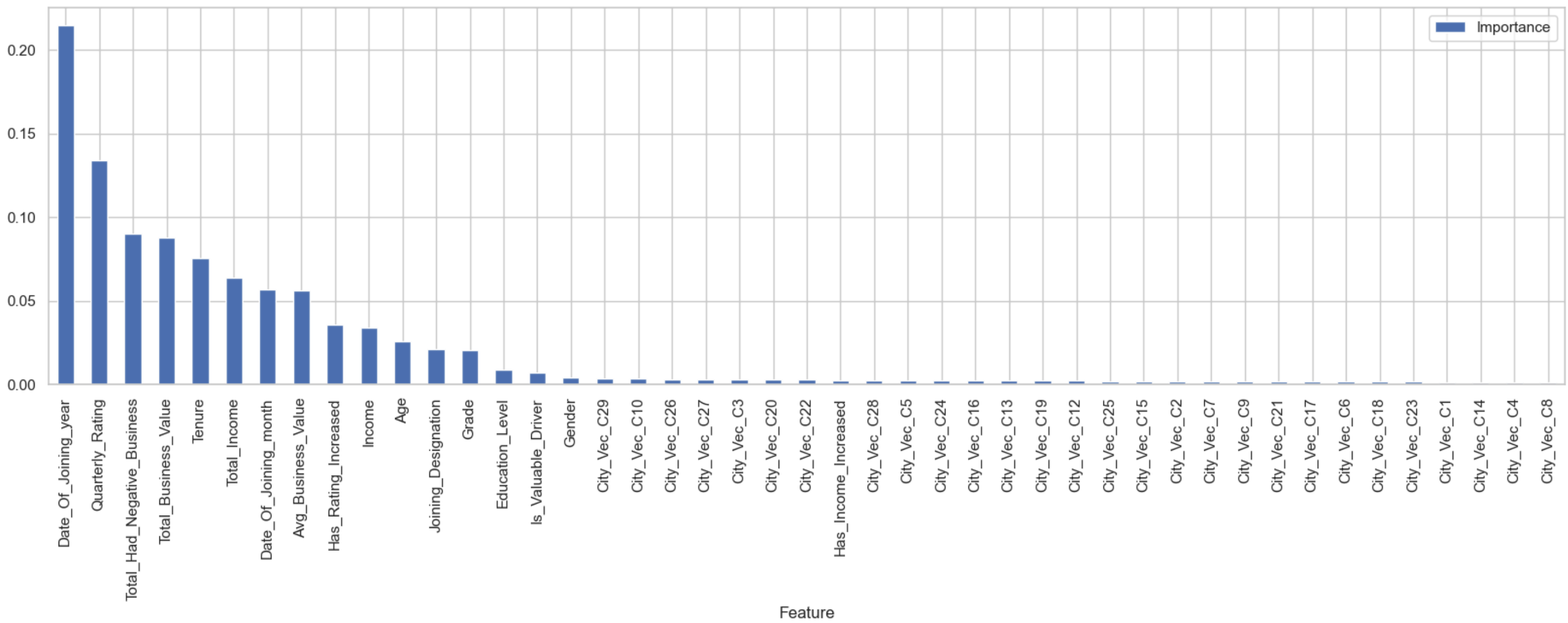
In [ ]:
```
best_rf = rf_cv_model.bestModel
print("NumTrees: ", best_rf.getNumTrees, "MaxDepth: ", best_rf.getMaxDepth())
best_rf.write().overwrite().save("./model/rf_model")
```

```
NumTrees:  250 MaxDepth:  10
```

In [ ]:
```
best_rf = RandomForestClassificationModel.load("./model/rf_model")
print("NumTrees: ", best_rf.getNumTrees, "MaxDepth: ", best_rf.getMaxDepth());
```

Feature Importance

In [ ]:
```
plot_func, _= get_feature_importance(best_rf, model_df, model_type="pyspark")
plot_func();
```



Observations

- We can see that Joining year is the most important feature followed by Quaterly rating

Metrics

In [ ]:
```
test_predictions = best_rf.transform(testData)
```

In [ ]:
```
y_true, y_pred, y_prob = spark_prediction_to_numpy(test_predictions);
```

Classification Report

In [ ]:
```
print_metrics(y_true, y_pred, y_prob)
```

```
              precision    recall  f1-score   support

           0       0.81      0.82      0.82       154
           1       0.90      0.89      0.89       263

    accuracy                           0.86       417
   macro avg       0.85      0.86      0.85       417
weighted avg       0.86      0.86      0.86       417

Accuracy:  0.8633093525179856
F1 Score:  0.8910133843212237
AUC:  0.9364969631129327
```

In [ ]:
```
best_threshold= get_best_threshold(y_true, y_prob)
```

```
Best threshold: 0.4
Precision at best threshold: 0.872791519434629
Recall at best threshold: 0.9391634980988594
F1 score at best threshold: 0.9047619047619048
```

In [ ]:
```
y_pred = (y_prob >= best_threshold).astype(int)
```

Best F1 score adjusted Classification Report

In [ ]:
```
print_metrics(y_true, y_pred, y_prob)
```

```
              precision    recall  f1-score   support

           0       0.88      0.77      0.82       154
           1       0.87      0.94      0.90       263

    accuracy                           0.88       417
   macro avg       0.88      0.85      0.86       417
weighted avg       0.88      0.88      0.87       417

Accuracy:  0.8752997601918465
F1 Score:  0.9047619047619048
AUC:  0.9364969631129327
```

Observations

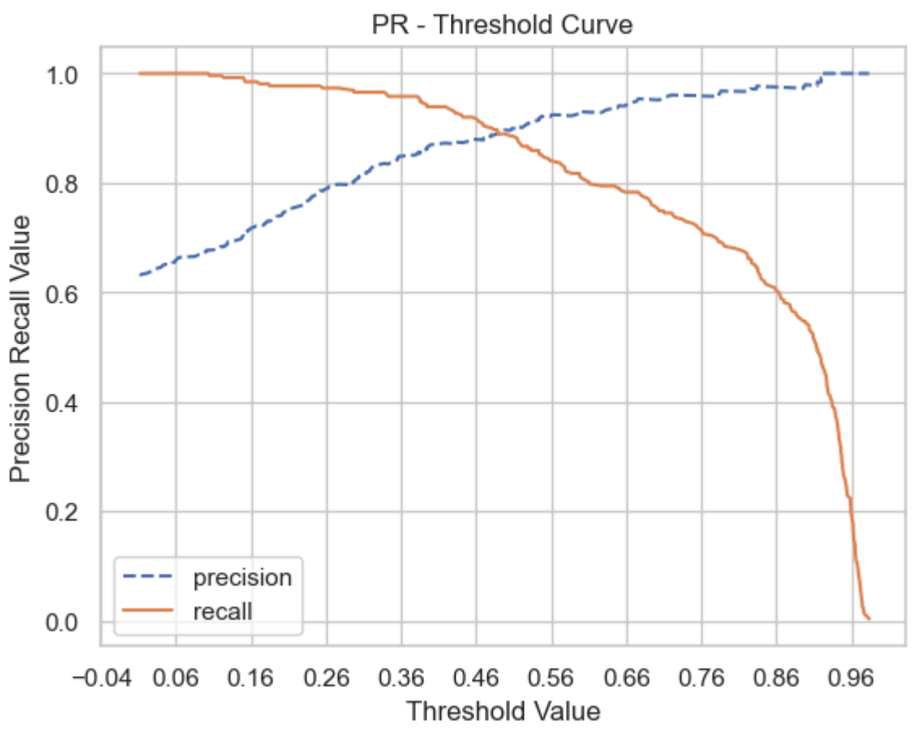
- With these new adjustment the, the f1 score increased by 1%, recall by 5% but precision is decreased by 3%

Insights

- A Recall of 94% means that out of all churned drivers, the model was correct 94% of times
- A precision of 87% means that out of all churned drivers predicted, 87% were actually churned.

PR vs Threshold Curve

In [ ]:
```
plot_precision_recall_threshold_curve(y_true, y_prob)
```

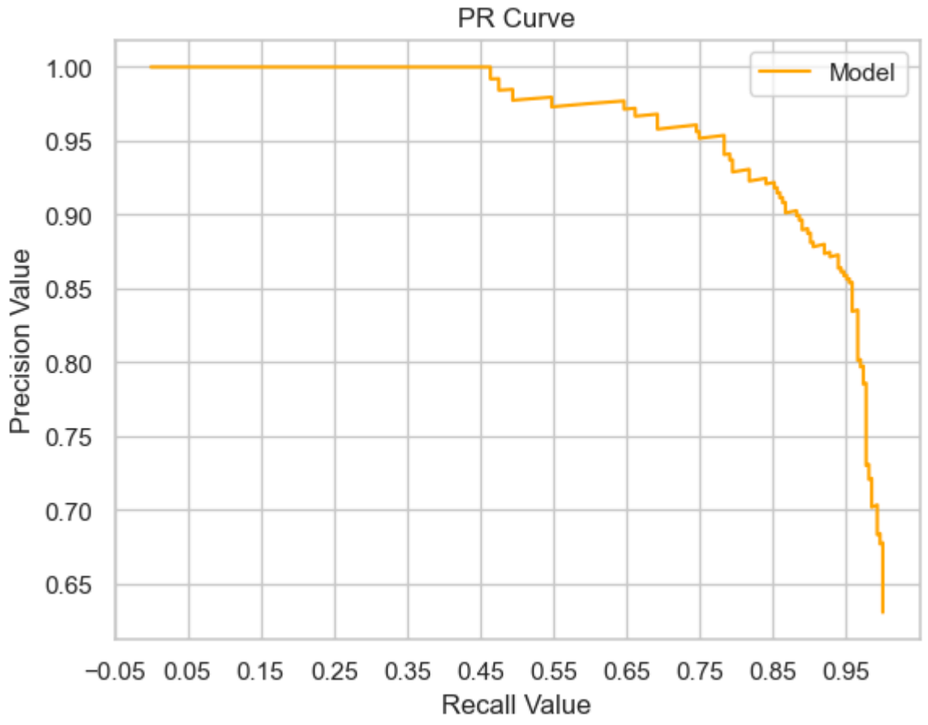


Observations

- From above plot we can decide on optimal value of threshold as that can maximize precision or recall as per business needs

Precision Recall Curve

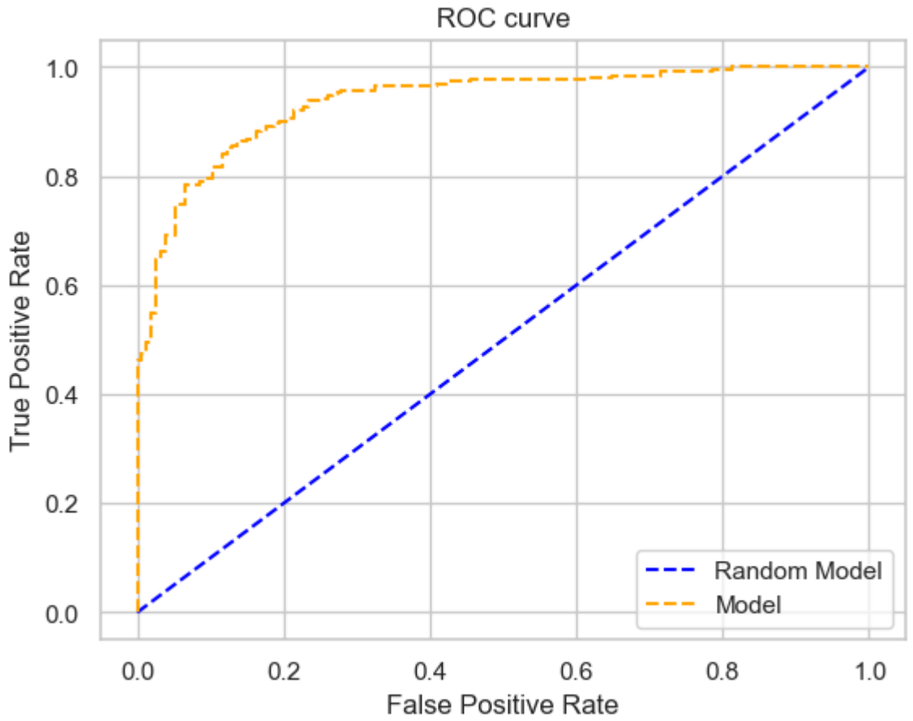
In [ ]:
```
plot_precision_recall_curve(y_true, y_prob)
```



Observations

- Above plot illustrates the tradeoff between precision and recall value of the model

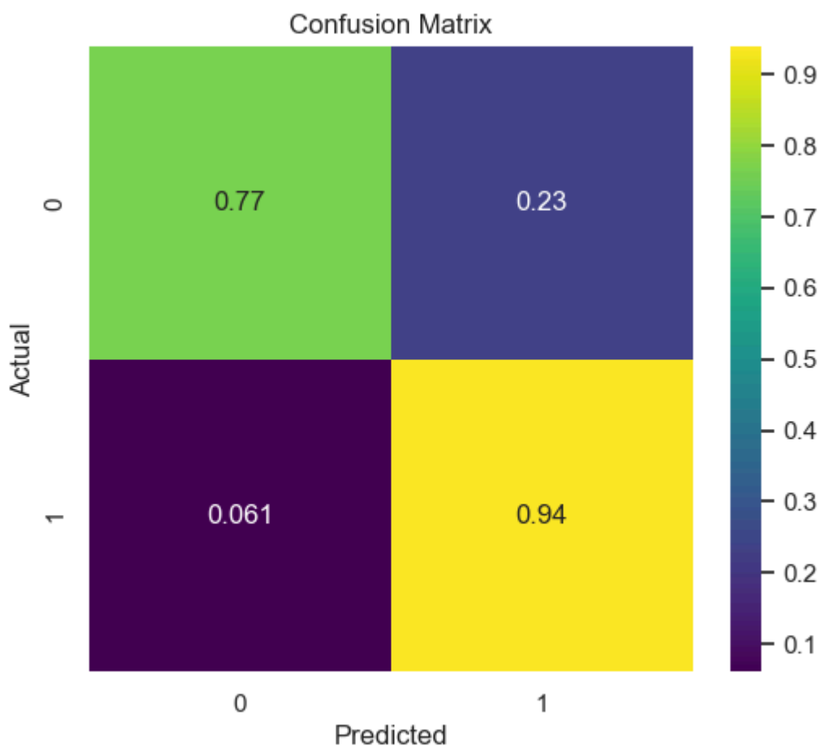
In [ ]:
```
plot_roc_curve(y_true, y_prob)
```



Observations

- Above plot is used to judge the performance of the model and compare it with plots of other models

In [ ]:
```
plot_confusion_matrix(y_true, y_pred)
```



Observations

- 77% of the actual negative cases (class 0, i.e., drivers who did not churn) were correctly predicted by the model as not having churned.
- 23% of the actual negative cases (class 0) were incorrectly predicted by the model as having churned (false alarms).
- 6.1% of the actual positive cases (class 1, i.e., drivers who did churn) were incorrectly predicted by the model as not having churned.
- 94% of the actual positive cases (class 1) were correctly predicted by the model as having churned.

Summary of Model

- The model has high recall for the positive class (churn), correctly identifying 94% of actual churn cases.
- The model's precision could be impacted by the 23% false positive rate, meaning that when the model predicts churn, there is a non-negligible chance that the prediction could be wrong.

Insights

Overall, the model seems to be more effective at identifying churn cases (high recall) than avoiding false alarms (moderate precision).

Gradient Boosting Classifier

Hyperparameter tuning using Grid Search without Cross Validation

In [ ]:
```
paramGrid = ParamGridBuilder()

maxDepth_values = np.random.choice(range(3, 10), size=4, replace=False)
# maxBins_values = np.random.choice(range(32, 257), size=5, replace=False)
stepSize_values =[0.1, 0.2]
maxIter_values = [30, 50, 100, 150]
# subsamplingRate_values = [0.8,1.0]

paramGrid = paramGrid.addGrid(gbt.maxDepth, maxDepth_values) \
                     .addGrid(gbt.maxIter, maxIter_values) \
                     .addGrid(gbt.stepSize, stepSize_values) \
                     .build()
```

In [ ]:
```
%%time
with mlflow.start_run(run_name="Gradient Boosting" ):
    gbt_cv_model, gbt_evaluator = fit_model(gbt, paramGrid, trainingData, use_train_validation_split=True)

    mlflow.log_params({
        "best_maxDepth": gbt_cv_model.bestModel.getMaxDepth(),
        "best_stepSize": gbt_cv_model.bestModel.getStepSize(),
        "best_maxIter": gbt_cv_model.bestModel.getMaxIter(),
    })
    test_predictions = gbt_cv_model.bestModel.transform(testData)

    mlflow.log_metrics({
        "best_f1": gbt_evaluator.evaluate(test_predictions),
        "best_accuracy": gbt_evaluator.setMetricName("accuracy").evaluate(test_predictions),
        "best_precision": gbt_evaluator.setMetricName("weightedPrecision").evaluate(test_predictions),
        "best_recall": gbt_evaluator.setMetricName("weightedRecall").evaluate(test_predictions)
    })
```

```
CPU times: user 1.9 s, sys: 937 ms, total: 2.84 s
Wall time: 9min 56s
```

In [ ]:
```
gbt_best = gbt_cv_model.bestModel
print("MaxDepth:", gbt_best.getMaxDepth(), ", MaxIter:", gbt_best.getMaxIter(), ", StepSize:", gbt_best.getStepSize())
```

```
MaxDepth: 4 , MaxIter: 30 , StepSize: 0.2
```

In [ ]:
```
gbt_best.write().overwrite().save("./model/gbt_model")
```

In [ ]:
```
gbt_best = GBTClassificationModel.load("./model/gbt_model")
print("MaxDepth:", gbt_best.getMaxDepth(), ", MaxIter:", gbt_best.getMaxIter(), ", StepSize:", gbt_best.getStepSize())
```

```
MaxDepth: 6 , MaxIter: 100 , StepSize: 0.1
```

Metrics

In [ ]:
```
y_true, y_pred, y_prob = spark_prediction_to_numpy(gbt_best.transform(testData))
```

Classification Report

In [ ]:
```
print_metrics(y_true, y_pred, y_prob)
```

```
              precision    recall  f1-score   support

           0       0.80      0.89      0.84       154
           1       0.93      0.87      0.90       263

    accuracy                           0.88       417
   macro avg       0.87      0.88      0.87       417
weighted avg       0.88      0.88      0.88       417

Accuracy:  0.8776978417266187
F1 Score:  0.899803536345776
AUC:  0.9558169966915213
```

In [ ]:
```
best_threshold= get_best_threshold(y_true, y_prob)
```

```
Best threshold: 0.26
Precision at best threshold: 0.9074074074074074
Recall at best threshold: 0.9315589353612167
F1 score at best threshold: 0.9193245778611632
```

In [ ]:
```
y_pred = (y_prob >= best_threshold).astype(int)
```

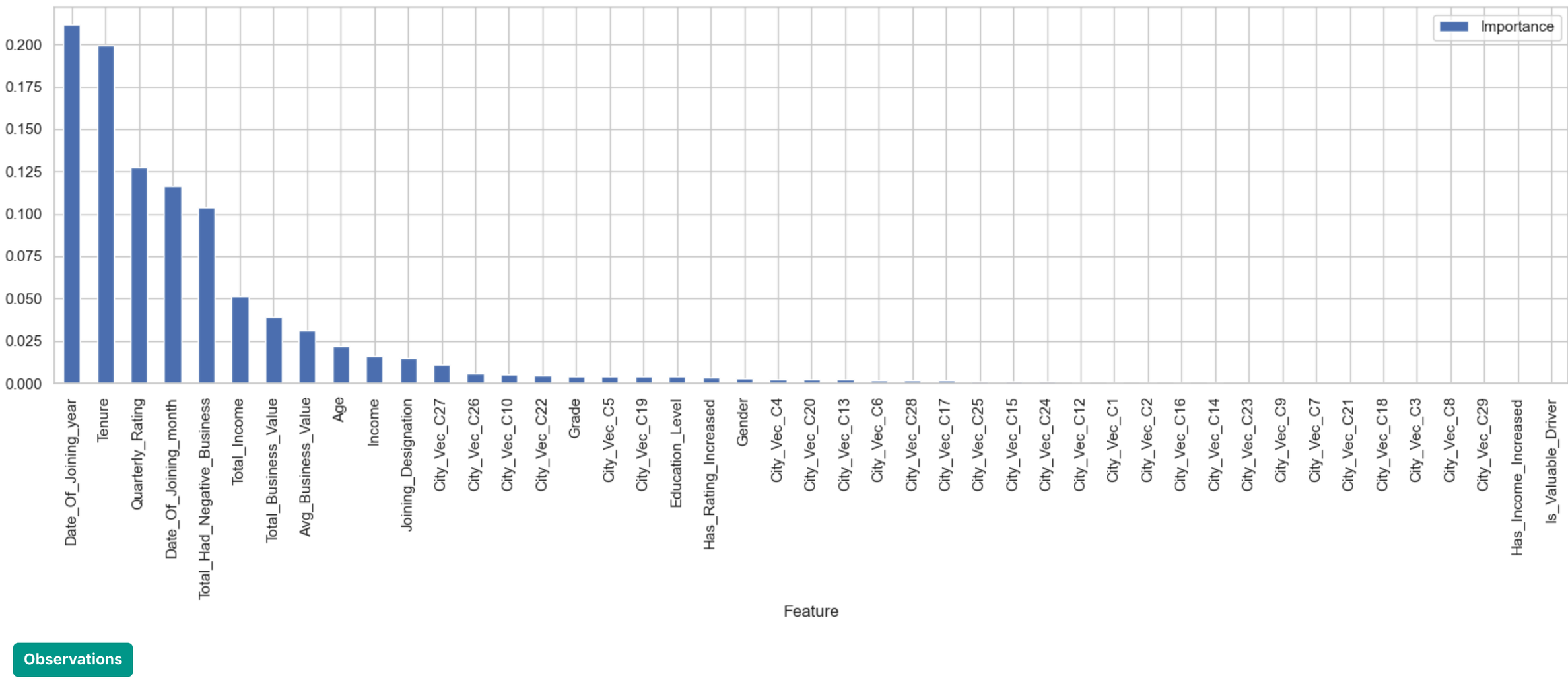
Best F1 score adjusted Classification Report

In [ ]:
```
print_metrics(y_true, y_pred, y_prob)
```

```
              precision    recall  f1-score   support

           0       0.88      0.84      0.86       154
           1       0.91      0.93      0.92       263

    accuracy                           0.90       417
   macro avg       0.89      0.88      0.89       417
weighted avg       0.90      0.90      0.90       417

Accuracy:  0.8968824940047961
F1 Score:  0.9193245778611632
AUC:  0.9558169966915213
```

Feature Importance

In [ ]:
```
plotfn, _=get_feature_importance(gbt_best, model_df, model_type="pyspark")
plotfn()
```

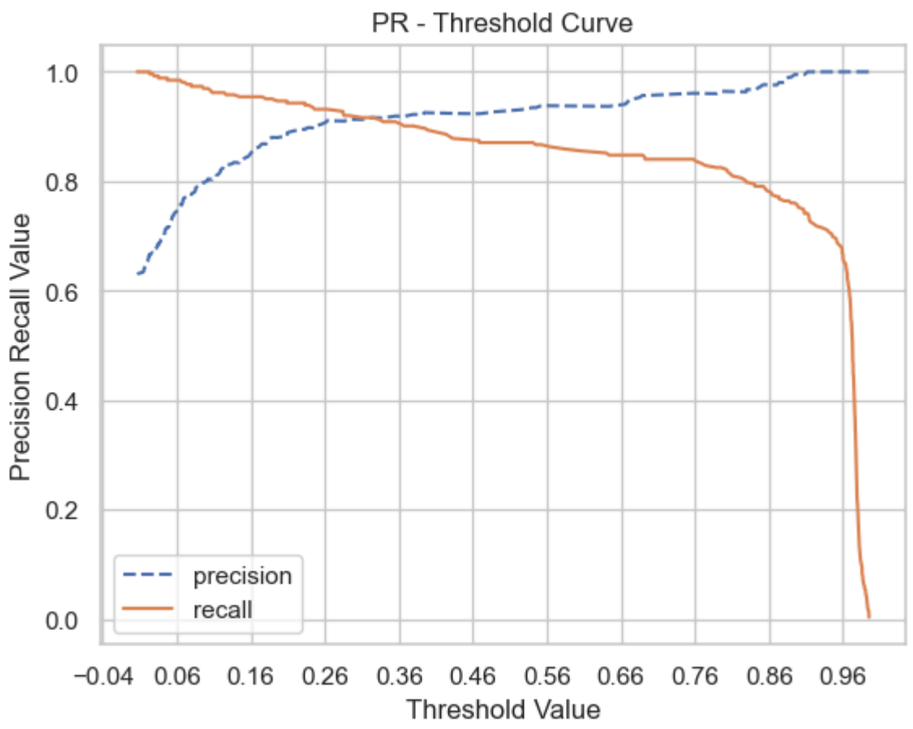


Observations

- We can see that Joining year is the biggest contributor when deciding on driver churn

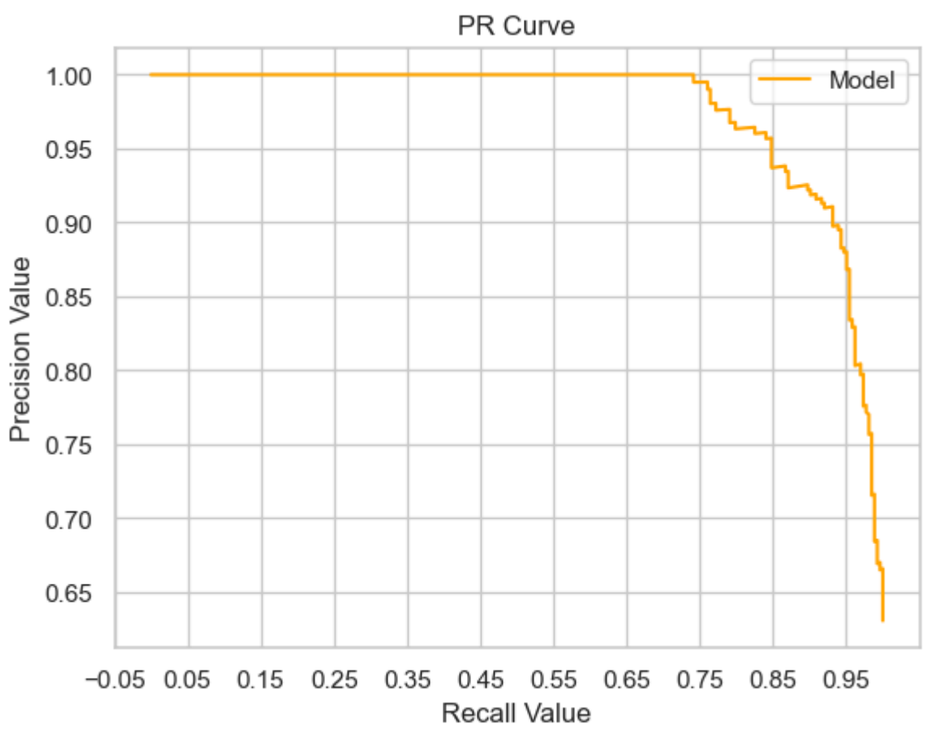
Precision Recall vs Threshold Curve

In [ ]:
```
plot_precision_recall_threshold_curve(y_true, y_prob)
```



Precision Recall Curve

In [ ]:
```
plot_precision_recall_curve(y_true, y_prob)
```

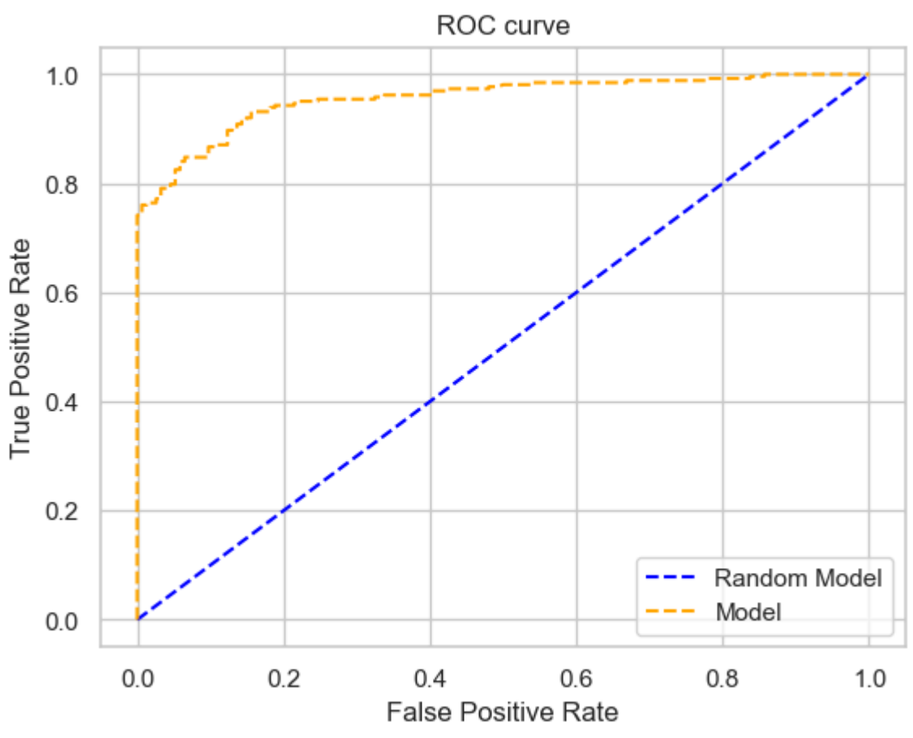


Observations

- From above plot we can decide on optimal value of threshold as that can maximize precision or recall as per business needs

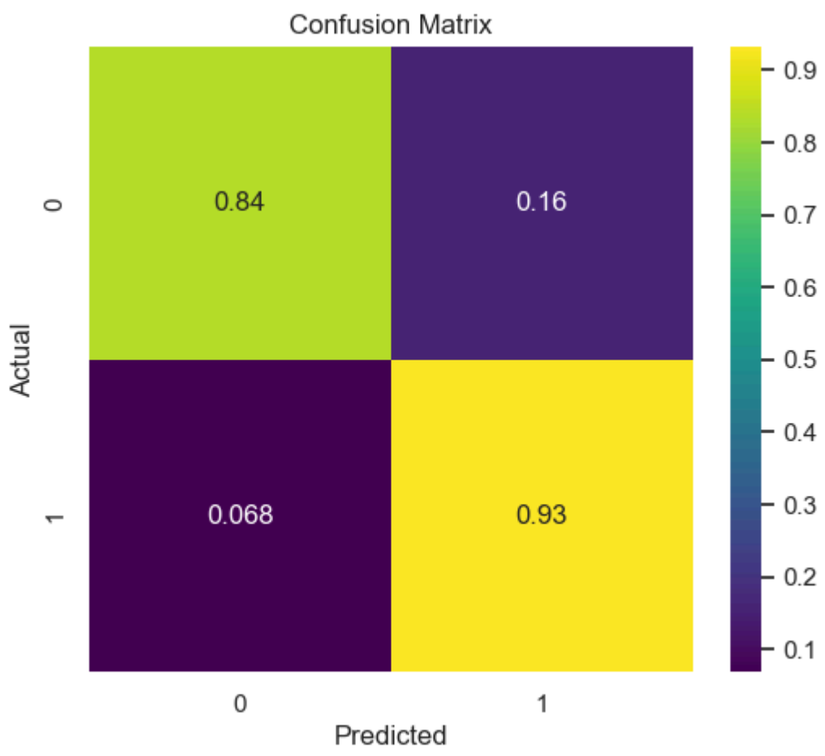
ROC-AUC Curve

In [ ]:
```
plot_roc_curve(y_true, y_prob)
```



Confusion Matrix

In [ ]:
```
plot_confusion_matrix(y_true, y_pred)
```



Observations

- 84% of the actual negative cases (class 0, i.e., drivers who did not churn) were correctly predicted by the model as not having churned.
- 16% of the actual negative cases (class 0) were incorrectly predicted by the model as having churned.
- 6.8% of the actual positive cases (class 1, i.e., drivers who did churn) were incorrectly predicted by the model as not having churned.
- 93% of the actual positive cases (class 1) were correctly predicted by the model as having churned.

Summary

- The model has a high recall for the positive class (churn), correctly identifying 93% of actual churn cases.
- It also has a relatively high precision as indicated by a lower false positive rate (16%).
- True Negative Rate (specificity) is also good, with 84% of non-churn cases correctly identified.

Insights

Overall, the model shows strong performance with a good balance between precision and recall, effectively identifying most churn cases while keeping the false positives reasonably low. This model seems well-tuned, especially for applications where catching potential churners is critical but where it's also important to minimize false positives.

XGBoost

In [ ]:
```
xgb = SparkXGBClassifier(label_col="Churned", features_col="features", weight_col="classWeightCol", seed=42,  verbose=False)
# xgb_feat = SparkXGBClassifier(label_col="Churned", features_col="features", weight_col="classWeightCol", seed=42, max_depth=5, n_estimators=150, learning_rate=0.1, subsample=0.8)
```

Hyperparameter tuning using Cross Validation and GridSearch

In [ ]:
```
paramGrid = ParamGridBuilder()
```

```
maxDepth_values = [3, 5, 7, 10]
# maxBins_values = np.random.choice(range(32, 257), size=5, replace=False)
eta =[0.1, 0.2, 0.3]
num_boosting_rounds_values = [ 150, 200, 250]
subsample_values = [0.8, 1.0]

paramGrid = paramGrid.addGrid(xgb.max_depth, maxDepth_values) \
                     .addGrid(xgb.learning_rate, eta) \
                     .addGrid(xgb.n_estimators, num_boosting_rounds_values) \
                     .addGrid(xgb.subsample, subsample_values) \
                     .build()
```

In [ ]:
```
with mlflow.start_run(run_name="XGBoost" ):
    xgb_cv_model, xgb_evaluator= fit_model(xgb, paramGrid, trainingData);

    params = {param.name: xgb_cv_model.bestModel.getOrDefault(param)
              for param in xgb_cv_model.bestModel.extractParamMap()}

    mlflow.log_params({
        "best_maxDepth": params["max_depth"],
        "best_learningRate": params["learning_rate"],
        "best_nEstimators": params["n_estimators"],
        "best_subsample": params["subsample"]
    })
    test_predictions = xgb_cv_model.bestModel.transform(testData)

    mlflow.log_metrics({
        "best_f1": xgb_evaluator.evaluate(test_predictions),
        "best_accuracy": xgb_evaluator.setMetricName("accuracy").evaluate(test_predictions),
        "best_precision": xgb_evaluator.setMetricName("weightedPrecision").evaluate(test_predictions),
        "best_recall": xgb_evaluator.setMetricName("weightedRecall").evaluate(test_predictions)
    })
clear_output()
```

In [ ]:
```
xgb_best = xgb_cv_model.bestModel
xgb_best.write().overwrite().save("./model/xgb_model")
```

In [ ]:
```
xgb_best.explainParam("max_depth"), xgb_best.explainParam("learning_rate"), xgb_best.explainParam("n_estimators"), xgb_best.explainParam("subsample")
```

Out[ ]:
```
('max_depth: Refer to XGBoost doc of xgboost.sklearn.XGBClassifier for this param max_depth (default: None, current: 5)',
 'learning_rate: Refer to XGBoost doc of xgboost.sklearn.XGBClassifier for this param learning_rate (default: None, current: 0.1)',
 'n_estimators: Refer to XGBoost doc of xgboost.sklearn.XGBClassifier for this param n_estimators (default: 100, current: 200)',
 'subsample: Refer to XGBoost doc of xgboost.sklearn.XGBClassifier for this param subsample (default: None, current: 0.8)')
```

In [ ]:
```
xgb_best = SparkXGBClassifierModel.load("./model/xgb_model")
xgb_best.explainParam("max_depth"), xgb_best.explainParam("learning_rate"), xgb_best.explainParam("n_estimators"), xgb_best.explainParam("subsample")
```

Out[ ]:
```
('max_depth: Refer to XGBoost doc of xgboost.sklearn.XGBClassifier for this param max_depth (default: None, current: 5)',
 'learning_rate: Refer to XGBoost doc of xgboost.sklearn.XGBClassifier for this param learning_rate (default: None, current: 0.1)',
 'n_estimators: Refer to XGBoost doc of xgboost.sklearn.XGBClassifier for this param n_estimators (default: 100, current: 200)',
 'subsample: Refer to XGBoost doc of xgboost.sklearn.XGBClassifier for this param subsample (default: None, current: 0.8)')
```

Metric

In [ ]:
```
test_predictions = xgb_best.transform(testData)
```

In [ ]:
```
y_true, y_pred, y_prob = spark_prediction_to_numpy(test_predictions)
```

```
2024-08-21 12:47:33,187 INFO XGBoost-PySpark: predict_udf Do the inference on the CPUs
```

In [ ]:
```
best_threshold= get_best_threshold(y_true, y_prob)
```

```
Best threshold: 0.52
Precision at best threshold: 0.9601593625498008
Recall at best threshold: 0.9163498098859315
F1 score at best threshold: 0.9377431906614786
```

In [ ]:
```
y_pred = (y_prob >= best_threshold).astype(int)
```

Best F1 score adjusted Classification Report

In [ ]:
```
print_metrics(y_true, y_pred, y_prob)
```

```
              precision    recall  f1-score   support

           0       0.87      0.94      0.90       154
           1       0.96      0.92      0.94       263

    accuracy                           0.92       417
   macro avg       0.91      0.93      0.92       417
weighted avg       0.93      0.92      0.92       417

Accuracy:  0.9232613908872902
F1 Score:  0.9377431906614786
AUC:  0.9682731716952249
```

Feature Importance

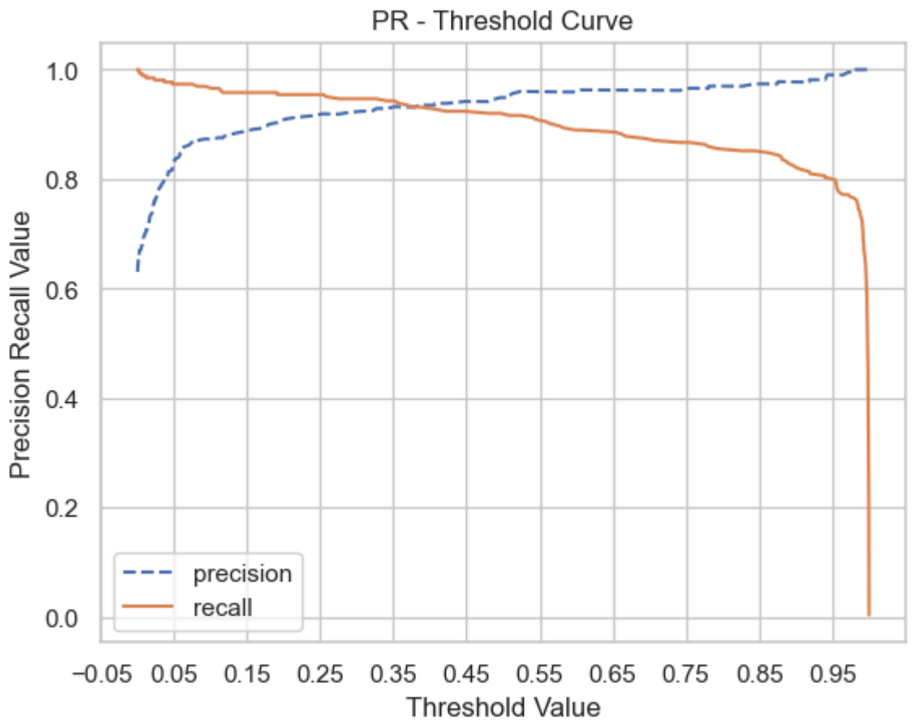
In [ ]:
```
plotfn, _=get_feature_importance(xgb_best, model_df, model_type="xgboost")
plotfn()
```


Observations

- From above plot we can see that Tenure is the most important feature

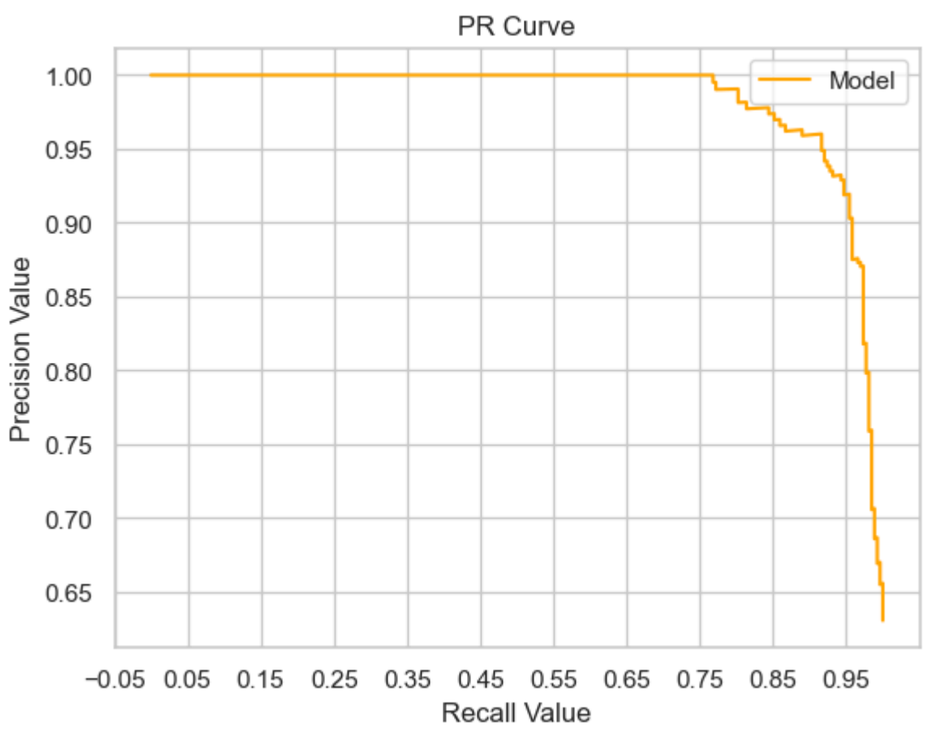
Precision Recall vs Threshold Curve

In [ ]:
```
plot_precision_recall_threshold_curve(y_true, y_prob)
```



Precision Recall Curve

In [ ]:
```
plot_precision_recall_curve(y_true, y_prob)
```

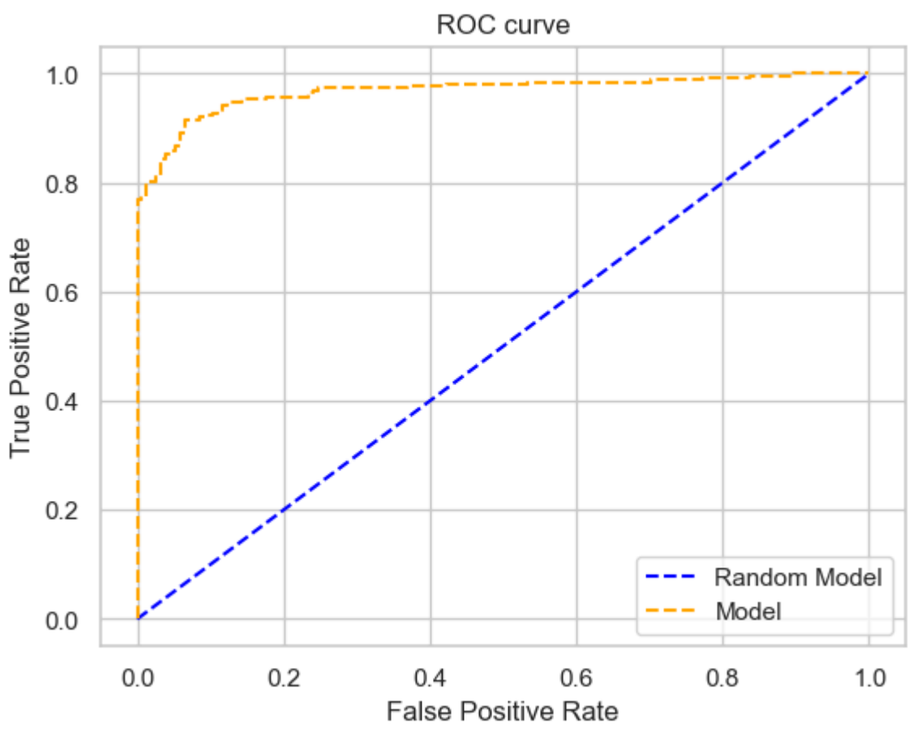


Observations

- From above plot we can decide on optimal value of threshold as that can maximize precision or recall as per business needs

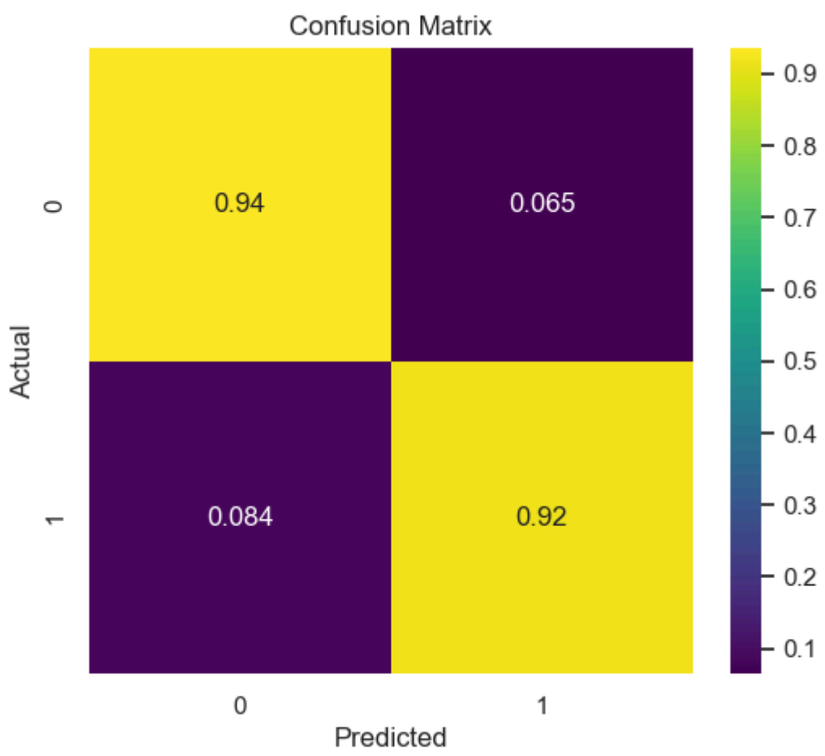
AUC-ROC Curve

In [ ]:
```
plot_roc_curve(y_true, y_prob)
```



Confusion Matrix

In [ ]:
```
plot_confusion_matrix(y_true, y_pred)
```



Observations

- 94% of the actual negative cases (class 0, i.e., drivers who did not churn) were correctly predicted by the model as not having churned.
- 6.5% of the actual negative cases (class 0) were incorrectly predicted by the model as having churned.
- 8.4% of the actual positive cases (class 1, i.e., drivers who did churn) were incorrectly predicted by the model as not having churned.
- 92% of the actual positive cases (class 1) were correctly predicted by the model as having churned.

Summary

- The model has a high true negative rate (specificity), correctly identifying 94% of non-churn cases, which indicates it is very good at avoiding false alarms.
- It also has a high true positive rate (recall) of 92%, correctly identifying most of the churn cases.
- False positive rate is relatively low at 6.5%, meaning the model is effective in minimizing incorrect predictions of churn where there is none.
- False negative rate is 8.4%, so the model misses some churn cases, but overall, the recall is strong.

Insights

Overall, this confusion matrix indicates a model that performs well, with strong precision and recall, particularly favoring specificity, making it a good candidate for situations where it is important to correctly identify non-churners while still catching most churners.

Light GBM

In [ ]:
```python
def train_tree(learningRate, numIterations, baggingFraction, tuning=False):
    with mlflow.start_run(nested=True, run_name="LightGBM_default"):

        lgbm = LightGBMClassifier(labelCol="Churned",
                                  featuresCol="features",
                                  weightCol="classWeightCol",
                                  featuresShapCol="shap_values",
                                  seed=42,
                                  learningRate=learningRate,
                                  numIterations=numIterations,
                                  #   maxDepth=maxDepth,
                                  baggingFraction=baggingFraction
                                  )

        if tuning:
            _train = trainingData_v2
            _validation = validationData
        else:
            _train = trainingData
            _validation = testData

        model = lgbm.fit(_train)

        evaluator = MulticlassClassificationEvaluator(
            labelCol="Churned", predictionCol="prediction", metricName="f1", weightCol='classWeightCol')

        predictions = model.transform(_validation)
        f1 = evaluator.evaluate(predictions)
        mlflow.log_params({
            "learningRate": learningRate,
            "numIterations": numIterations,
            # "maxDepth": maxDepth,
            "baggingFraction": baggingFraction
        })
        mlflow.log_metric("f1", f1)
    return model, f1
```

Hyperparameter tuning using hyperopt

In [ ]:
```python
from hyperopt import fmin, tpe, hp, Trials, STATUS_OK
```

```python
def train_with_hyperopt(params):
    """
    :param params: hyperparameters as a dict. Its structure is consistent with how search space is defined. See below.
    :return: dict with fields 'loss' (scalar loss) and 'status' (success/failure status of run)
    """

    learningRate = params["learningRate"]
    numIterations = params["numIterations"]
    # maxDepth=params["maxDepth"]
    baggingFraction = params["baggingFraction"]

    model, val_metric = train_tree(
        learningRate=learningRate, numIterations=numIterations, baggingFraction=baggingFraction, tuning=True
    )
    loss = -val_metric

    return {"loss": loss, "status": STATUS_OK}


space = {
    "learningRate": hp.uniform("learningRate", 0.1, 0.3),
    "numIterations": hp.uniformint("numIterations", 50, 200),
    # "maxDepth": hp.uniformint("maxDepth", 3, 10),
    "baggingFraction": hp.uniform("baggingFraction", 0.6, 1.0),
}

algo = tpe.suggest

with mlflow.start_run(run_name="Hyperopt_LGBM"):
    best_params = fmin(fn=train_with_hyperopt, space=space,
                       algo=algo, max_evals=30)
```

```
clear_output()
```

In [ ]:
```
best_params
```

Out[ ]:
```
{'baggingFraction': 0.7612915269858092,
 'learningRate': 0.19183342083552588,
 'numIterations': 114.0}
```

Hyperparamenter tuning using Optuna

In [ ]:
```
import optuna
import joblib
from joblibspark import register_spark

register_spark()
study = optuna.create_study(direction="maximize")
```

```
[I 2024-08-21 13:55:48,698] A new study created in memory with name: no-name-3778dcd1-688b-4bc9-ba73-73c1bd7cd750
```

In [ ]:
```
def objective(trial):

    learningRate = trial.suggest_float("learningRate", 0.1, 0.3)
    numIterations = trial.suggest_int("numIterations", 50, 200)
    baggingFraction = trial.suggest_float("baggingFraction", 0.6, 1.0)

    model, val_metric = train_tree(
        learningRate=learningRate, numIterations=numIterations, baggingFraction=baggingFraction, tuning=True
    )
    return val_metric
```

In [ ]:
```
with joblib.parallel_backend("spark", n_jobs=-1):
    study.optimize(objective, n_trials=50);
clear_output()
```

In [ ]:
```
trial = study.best_trial

print(f"Best trial f1 score: {trial.value}")
print("Best trial params: ")
for key, value in trial.params.items():
    print(f"    {key}: {value}")
```

```
Best trial f1 score: 0.9414124608301843
Best trial params:
    learningRate: 0.17115815041559626
    numIterations: 164
    baggingFraction: 0.6810831568144191
```

In [ ]:
```
final_model, metric = train_tree(**best_params, tuning=False)
```

```
[LightGBM] [Info] Saving data reference to binary buffer
[Stage 229:>                                                        (0 + 1) / 1]
[LightGBM] [Info] Loaded reference dataset: 45 features, 1964 num_data
```

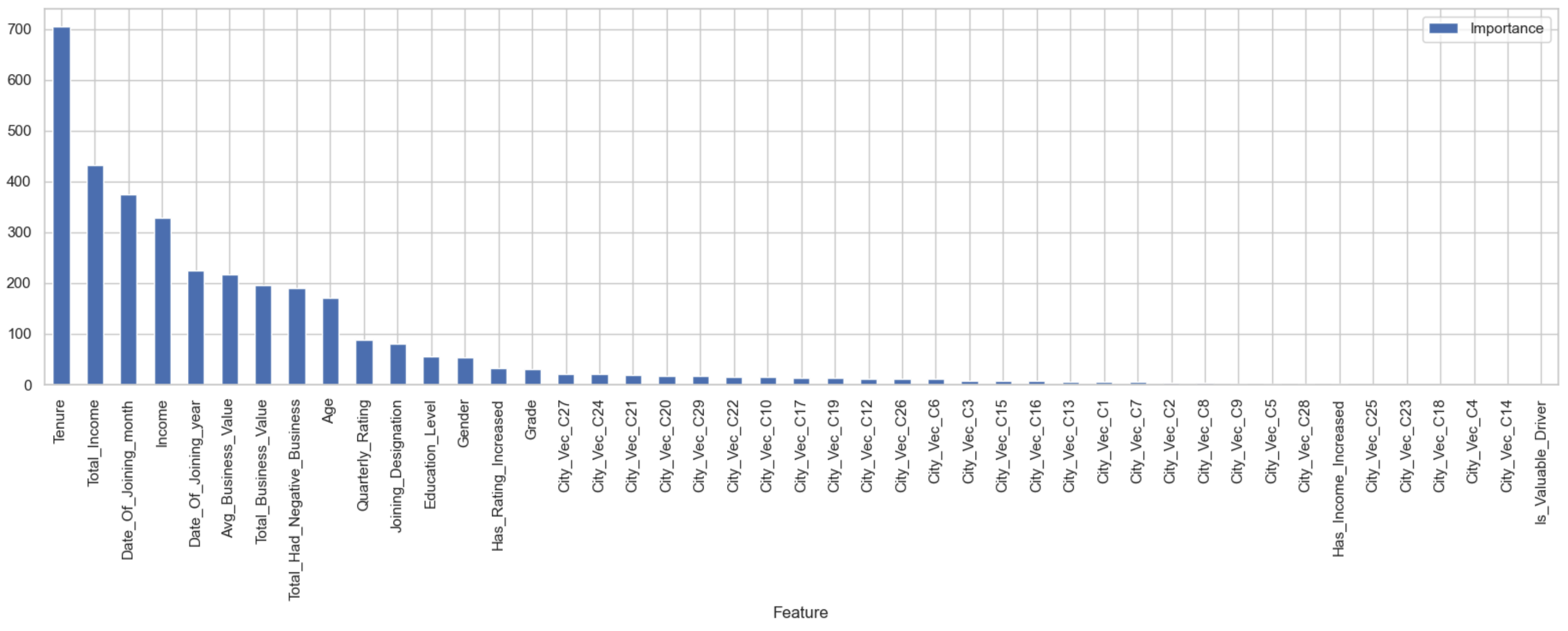
In [ ]:
```
metric
```

Out[ ]:
```
0.9285341045543781
```

In [ ]:
```
final_model.write().overwrite().save("./model/lgbm_model")
```

Feature Importance

In [ ]:
```
plot_func, _=get_feature_importance(final_model, model_df, model_type="lightgbm")
plot_func();
```



Observations

- We can see that Tenure is the most important feature

Metrics

In [ ]:
```
test_predictions = final_model.transform(testData)
```

In [ ]:
```
y_true, y_pred, y_prob = spark_prediction_to_numpy(test_predictions)
```

Classification Report

In [ ]:
```
print_metrics(y_true, y_pred, y_prob)
```

```
              precision    recall  f1-score   support

           0       0.86      0.94      0.90       154
           1       0.96      0.91      0.94       263

    accuracy                           0.92       417
   macro avg       0.91      0.93      0.92       417
weighted avg       0.93      0.92      0.92       417

Accuracy:  0.9232613908872902
F1 Score:  0.9375
AUC:  0.9692854673843266
```

In [ ]:
```
best_threshold = get_best_threshold(y_true=y_true, y_prob=y_prob)
```

```
Best threshold: 0.25
Precision at best threshold: 0.9461538461538461
Recall at best threshold: 0.935361216730038
F1 score at best threshold: 0.9407265774378585
```

In [ ]:
```
y_pred = np.where(y_prob > best_threshold, 1, 0)
```

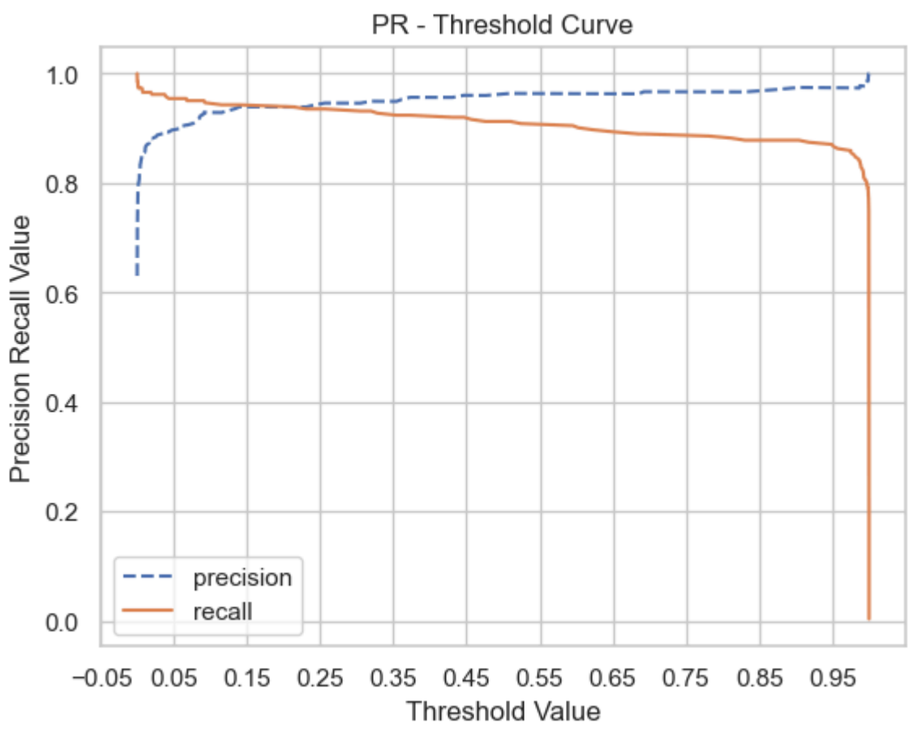
Best F1 score adjusted Classification Report

In [ ]:
```
print_metrics(y_true, y_pred, y_prob)
```

```
              precision    recall  f1-score   support

           0       0.89      0.91      0.90       154
           1       0.95      0.94      0.94       263

    accuracy                           0.93       417
   macro avg       0.92      0.92      0.92       417
weighted avg       0.93      0.93      0.93       417

Accuracy:  0.9256594724220624
F1 Score:  0.9407265774378585
AUC:  0.9692854673843266
```

Precision Recall vs Threshold Curve

In [ ]:
```
plot_precision_recall_threshold_curve(y_true, y_prob)
```

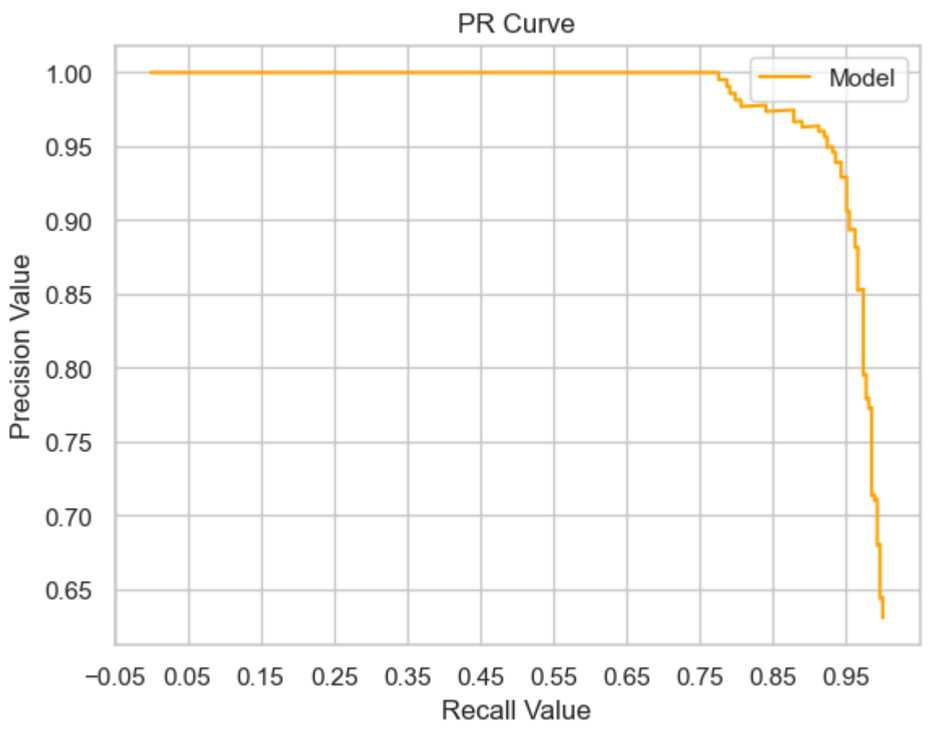


Observations

- From above plot we can decide on optimal value of threshold as that can maximize precision or recall as per business needs

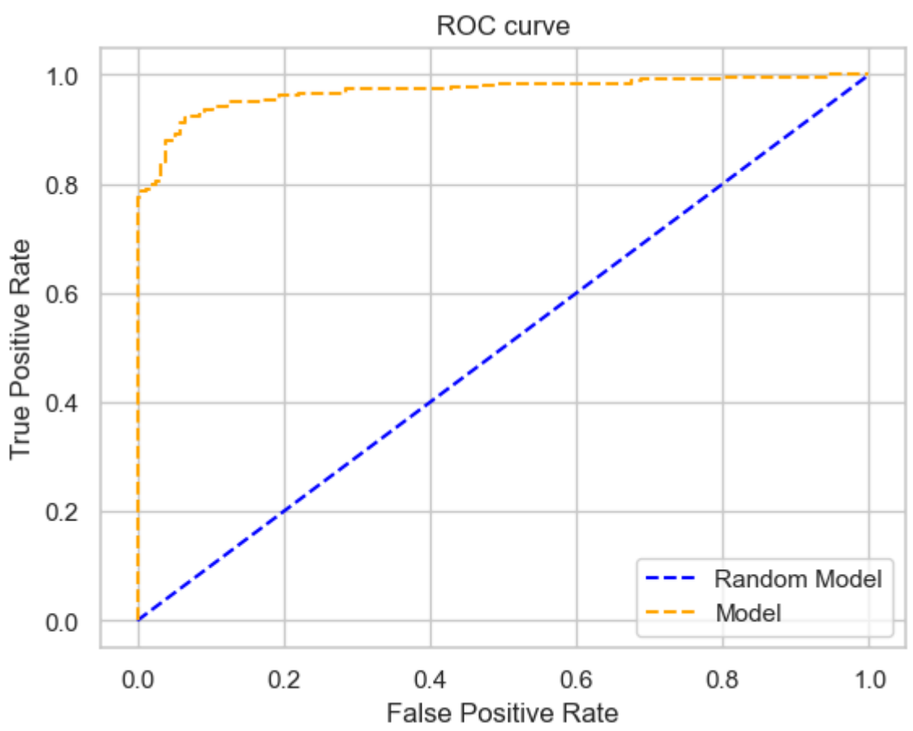
Precision Recall Curve

In [ ]:
```
plot_precision_recall_curve(y_true, y_prob)
```



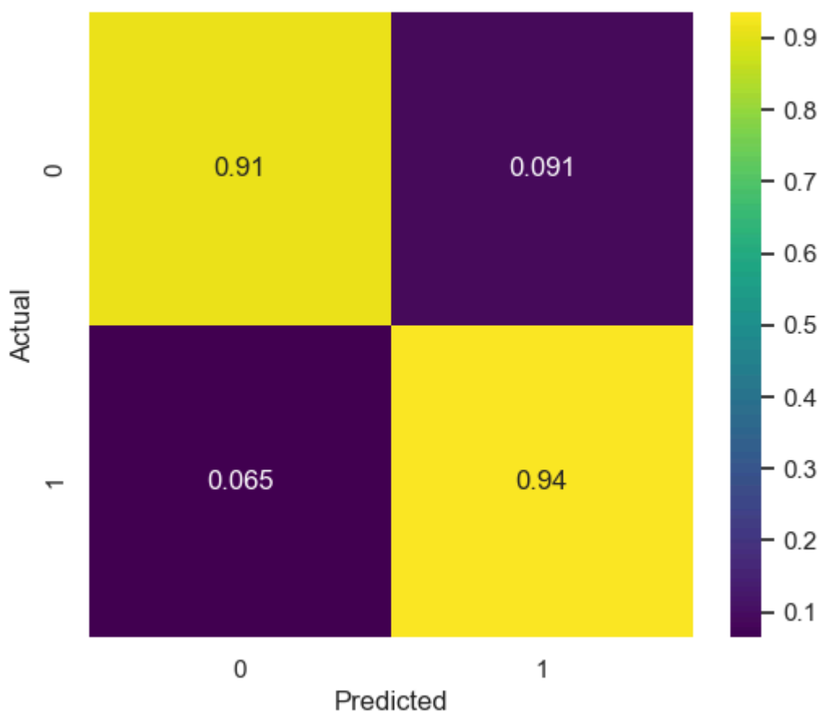
ROC-AUC Curve

In [ ]:
```
plot_roc_curve(y_true, y_prob)
```



Confusion Matrix

In [ ]:
```
plot_confusion_matrix(y_true, y_pred)
```



Observations

- The model correctly predicted 91% of the drivers who did not churn (class 0).
- The model incorrectly predicted that 9.1% of the drivers would churn, but they did not.
- The model incorrectly predicted that 6.5% of the drivers would not churn, but they actually did.
- The model correctly predicted 94% of the drivers who churned (class 1).

Summary

- The model has a high true negative rate (specificity), correctly identifying 91% of non-churn cases, which indicates it is very good at avoiding false alarms.
- It also has a high true positive rate (recall) of 94%, correctly identifying most of the churn cases.
- False positive rate is relatively low at 9.1%, meaning the model is effective in minimizing incorrect predictions of churn where there is none.
- False negative rate is 6.5%, so the model misses some churn cases, but overall, the recall is strong.

Insights

Overall, this confusion matrix indicates a model that performs well, with strong precision and recall, particularly favoring recall, making it a good candidate for situations where it is important to correctly identify churners while still maintaining good specificity.

Model Comparison

In [ ]:
```
best_rf = RandomForestClassificationModel.load("./model/rf_model")
best_gbt = GBTClassificationModel.load("./model/gbt_model")
best_xgb = SparkXGBClassifierModel.load("./model/xgb_model")
best_lgbm = LightGBMClassificationModel.load("./model/lgbm_model")
```

In [ ]:
```
test_predictions_rf = best_rf.transform(testData)
test_predictions_gbt = best_gbt.transform(testData)
test_predictions_xgb = best_xgb.transform(testData)
test_predictions_lgbm = best_lgbm.transform(testData)
```

In [ ]:
```
y_true_rf, y_pred_rf, y_prob_rf = spark_prediction_to_numpy(test_predictions_rf)
y_true_gbt, y_pred_gbt, y_prob_gbt = spark_prediction_to_numpy(test_predictions_gbt)
y_true_xgb, y_pred_xgb, y_prob_xgb = spark_prediction_to_numpy(test_predictions_xgb)
y_true_lgbm, y_pred_lgbm, y_prob_lgbm = spark_prediction_to_numpy(test_predictions_lgbm)
```

```
2024-08-23 16:46:59,296 INFO XGBoost-PySpark: predict_udf Do the inference on the CPUs
```

In [ ]:
```
best_threshold_rf = get_best_threshold(y_true_rf, y_prob_rf)
best_threshold_gbt = get_best_threshold(y_true_gbt, y_prob_gbt)
best_threshold_xgb = get_best_threshold(y_true_xgb, y_prob_xgb)
best_threshold_lgbm = get_best_threshold(y_true_lgbm, y_prob_lgbm)

y_pred_rf = (y_prob_rf >= best_threshold_rf).astype(int)
y_pred_gbt = (y_prob_gbt >= best_threshold_gbt).astype(int)
y_pred_xgb = (y_prob_xgb >= best_threshold_xgb).astype(int)
y_pred_lgbm = (y_prob_lgbm >= best_threshold_lgbm).astype(int)
```

```
Best threshold: 0.4
Precision at best threshold: 0.872791519434629
Recall at best threshold: 0.9391634980988594
F1 score at best threshold: 0.9047619047619048
Best threshold: 0.26
Precision at best threshold: 0.9074074074074074
Recall at best threshold: 0.9315589353612167
F1 score at best threshold: 0.9193245778611632
Best threshold: 0.52
Precision at best threshold: 0.9601593625498008
Recall at best threshold: 0.9163498098859315
F1 score at best threshold: 0.9377431906614786
Best threshold: 0.25
Precision at best threshold: 0.9461538461538461
Recall at best threshold: 0.935361216730038
F1 score at best threshold: 0.9407265774378585
```

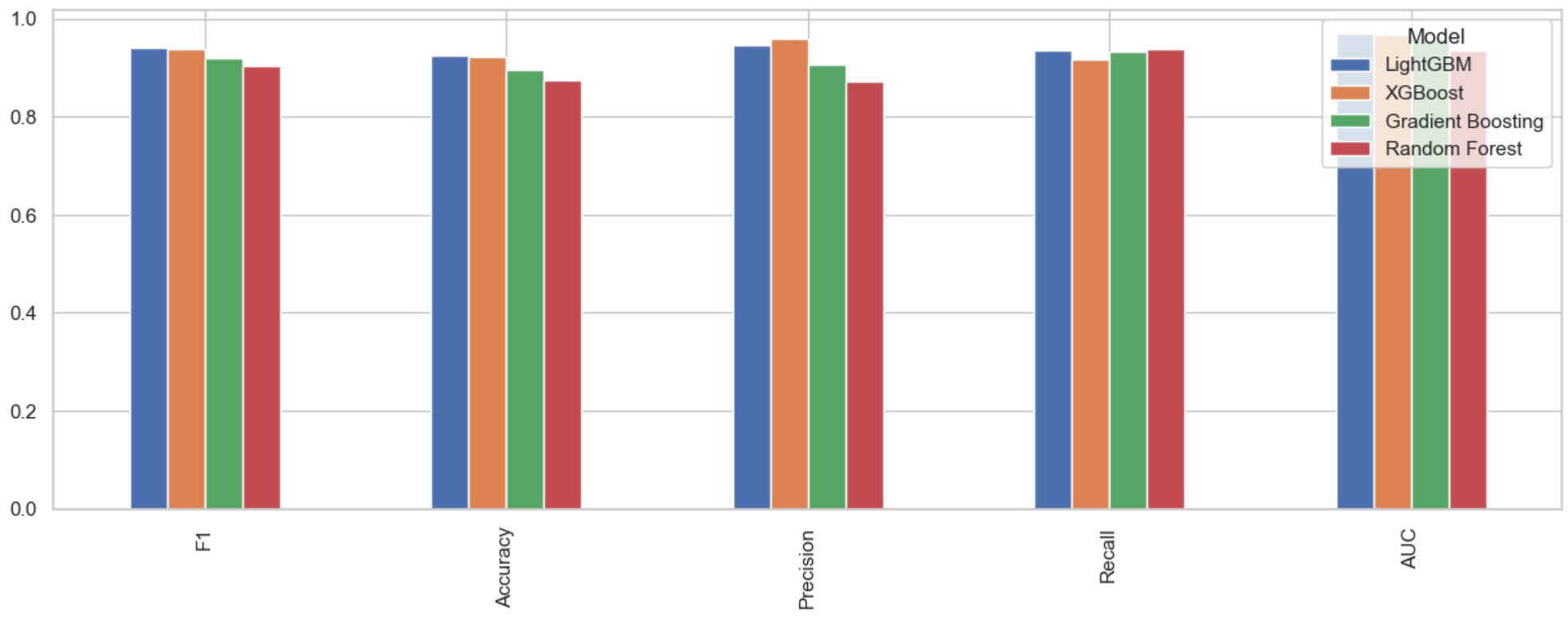
In [ ]:
```
columns = ["Model", "F1", "Accuracy", "Precision", "Recall", "AUC"]

results = [
    ("Random Forest", f1_score(y_true_rf, y_pred_rf), accuracy_score(y_true_rf, y_pred_rf), precision_score(y_true_rf, y_pred_rf), recall_score(y_true_rf, y_pred_rf), roc_auc_score(y_true
    ("Gradient Boosting", f1_score(y_true_gbt, y_pred_gbt), accuracy_score(y_true_gbt, y_pred_gbt), precision_score(y_true_gbt, y_pred_gbt), recall_score(y_true_gbt, y_pred_gbt), roc_auc_
    ("XGBoost", f1_score(y_true_xgb, y_pred_xgb), accuracy_score(y_true_xgb, y_pred_xgb), precision_score(y_true_xgb, y_pred_xgb), recall_score(y_true_xgb, y_pred_xgb), roc_auc_score(y_tr
    ("LightGBM", f1_score(y_true_lgbm, y_pred_lgbm), accuracy_score(y_true_lgbm, y_pred_lgbm), precision_score(y_true_lgbm, y_pred_lgbm), recall_score(y_true_lgbm, y_pred_lgbm), roc_auc_s
]
```

In [ ]:
```
results_df = pd.DataFrame(results, columns=columns).sort_values("F1", ascending=False)
results_df
```

Out[ ]:

	Model	F1	Accuracy	Precision	Recall	AUC
3	LightGBM	0.940727	0.925659	0.946154	0.935361	0.969285
2	XGBoost	0.937743	0.923261	0.960159	0.916350	0.968273
1	Gradient Boosting	0.919325	0.896882	0.907407	0.931559	0.955817
0	Random Forest	0.904762	0.875300	0.872792	0.939163	0.936497

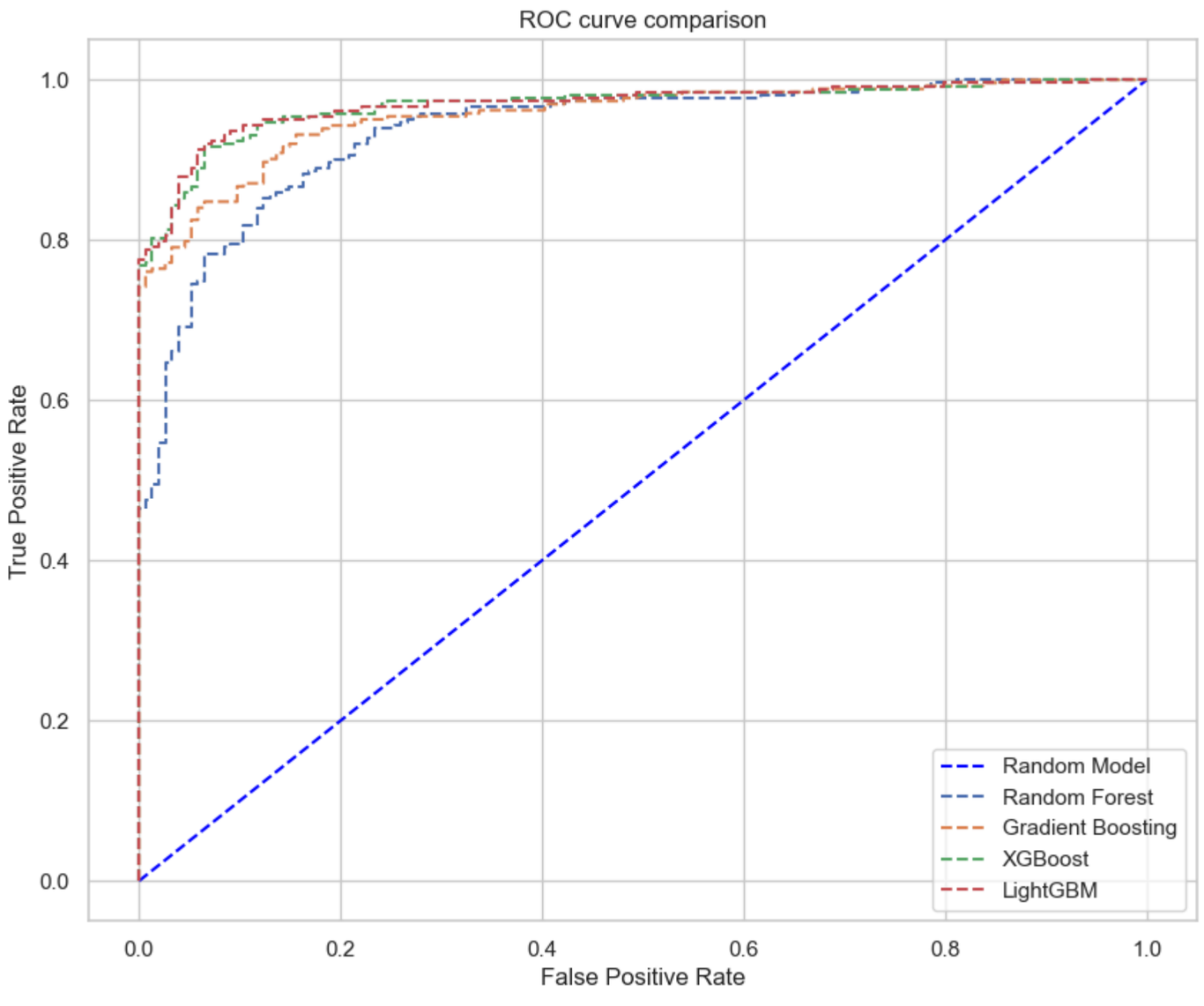
In [ ]:
```
results_df.set_index("Model").T.plot(kind="bar", figsize=(15, 5));
```



Observations

- We can see that LightGBM has highest F1 score

In [ ]:
```
plt.figure(figsize=(10, 8))

plot_roc_curve(y_true_rf, y_prob_rf, label='Random Forest', color=palette[0], show_random=True)
plot_roc_curve(y_true_gbt, y_prob_gbt, label='Gradient Boosting', color=palette[1], show_random=False)
plot_roc_curve(y_true_xgb, y_prob_xgb, label='XGBoost', color=palette[2], show_random=False)
plot_roc_curve(y_true_lgbm, y_prob_lgbm, label='LightGBM', color=palette[3], show_random=False)

# Show the plot
plt.title('ROC curve comparison')
plt.show();
```

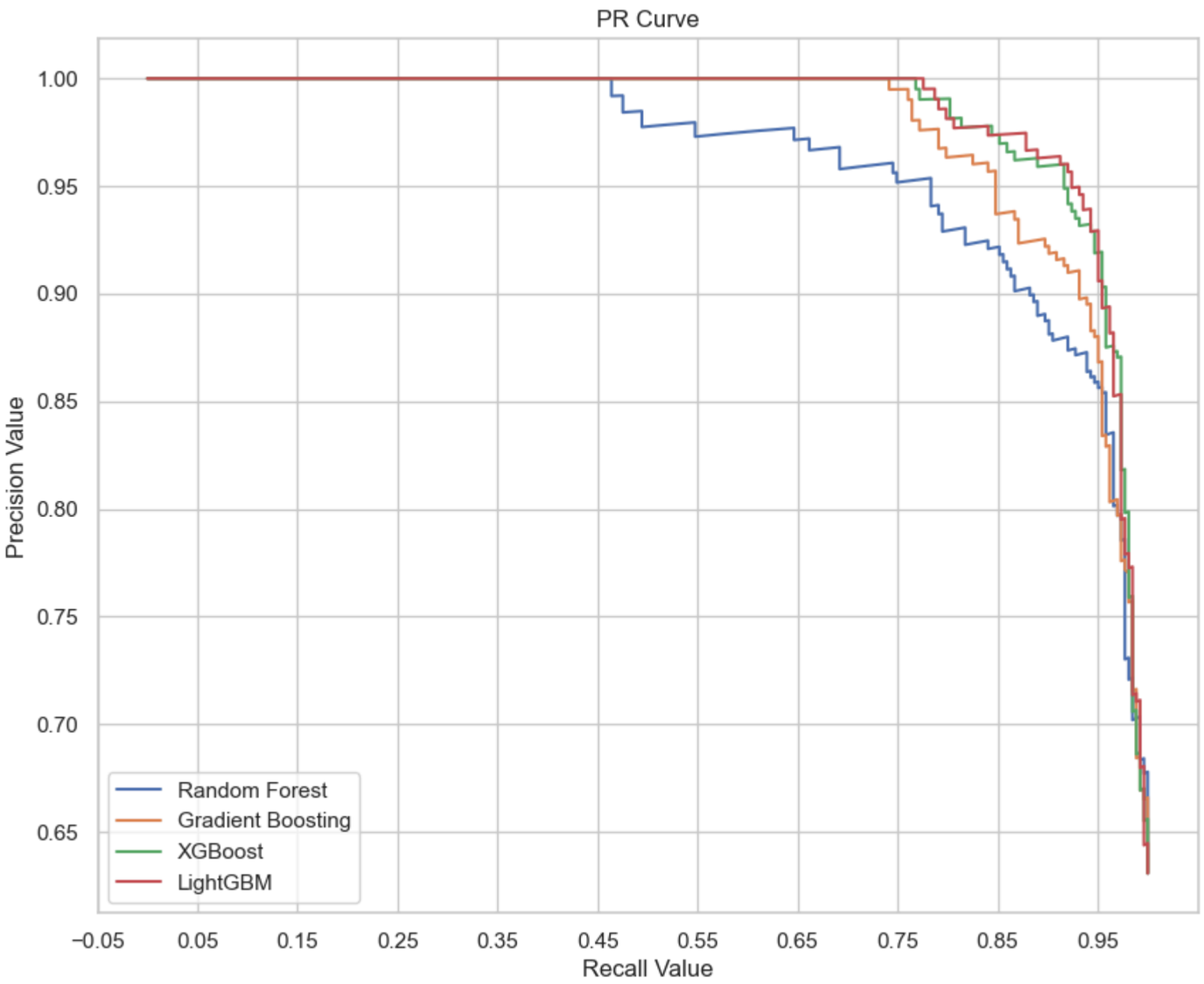


Observations

- From above plot we can see that Light Gradient Boosting and XGBoost algorithms has highest area under the curve

In [ ]:
```
plt.figure(figsize=(10, 8))

plot_precision_recall_curve(y_true_rf, y_prob_rf, label='Random Forest', color=palette[0])
plot_precision_recall_curve(y_true_gbt, y_prob_gbt, label='Gradient Boosting', color=palette[1])
plot_precision_recall_curve(y_true_xgb, y_prob_xgb, label='XGBoost', color=palette[2])
plot_precision_recall_curve(y_true_lgbm, y_prob_lgbm, label='LightGBM', color=palette[3])
```

Out[ ]:
```
<Figure size 1000x800 with 0 Axes>
```



Observations

- From above plot we can say that LightGBM has highest area under the curve

Insights and Recommendations

Driver Related

General Observation

- In the dataset there are 1616 churned drivers out of which 667 are females.
- Only 699 drivers increased their rating
- Only 44 drivers were given increment
- There was large number of drivers who joined during 2018, and large exit of drivers during 2019
- Income of Churned drivers is less than current drivers
- Drivers with 1 star rating are more likely to churn as compared to higher rating
- Driver having Grade 1, 2 and 3 are more likely to churn
- C13 have highest ratio of Churned drivers
- C29 have the lowest ratio of Churned drivers and makes the highest revenue.
- C13 city has the best revenue to expense ratio
- Gender and Education level do not have significant effect on Churn
- Joining designation and Joining year have significant effect on churn
- Drivers whose salary and rating increased are less likely to churn
- Higher grade drivers have high business value
- We can see that Drivers who have negative business months are less likely to churn

Driver Rating

- Majority of driver ratings are 1 star. But there was no single 5 star rating.
- As Age increases, Quarterly ratings move towards higher side.

Change in ratings for different cities

- C29 had the highest positive change in 4 star rating from 2019 to 2020
- C17 had the biggest fall of all type of rating from 2019 to 2020
- C2, C14, C9 had big fall of 3 star rating from 2019 to 2020

Effect on business value when ratings decrease

- From above plot we can see that out of 559 drivers whose rating decreased, 540 drivers business decreases significantly.
- This shows that Driver rating has significant impact on business

Effect of rating based on month of year

- we can see that demand increases during November to January, but falls slightly during other months.
- This is because of the holiday season. Since the numbers of rides increases, the corresponding ratings also increase.
- There is not much seasonality

Effect of Ratings based on City

- There is not much effect for drivers working in different cities, but there are some important points related to some cities.
- C17 has lowest 4 star rating and highest 1 star.
- C14,C16 ans C24 have good pct of 4 star rating among other cities.

Other features affecting Quarterly Rating

- We can see that drivers joining at higher designation have higher ratio of lower rating.
- But their absolute rating count is lesser as compared to lower designation.

Recommendations

- Drivers should be given raises more often.
- Expectation from the job has to be asked from the drivers who joined recently as they tend to churn the most.
- Feedback must be taken from employee with consistent low rating.
- Drivers can be assigned different city to check if their ratings can be increased.
- Ratings can be changed from Quarterly to monthly to better reflect progress

Predictive Model Related

- From all the above model feature importances Tenure, Quarterly Rating and Income are the biggest contributor for generating the predictions

Choosing the right model

- From above analysis of models, we can conclude that **LightGBM** has better stats wrt to other models
- Following are the model stats
  - F1 score: 94%
  - Accuracy: 92.5%
  - Precision: 94.6%
  - Recall: 93.5%
  - AUC: 96.9%

Precision Recall Gap

Recall means out of all the drivers, how correctly the model identifies churning and Precision means from all the drivers identified to be churned, how many actually churned.
Assume that the company has decided to give a raise for those drivers which are predicted to churn

Case 1: Business want my model to detect even a small possibility of Churn.

- In this case I will decrease my threshold value. This will lead increase in recall, but will decrease precision.
- This means more drivers will be predicted to churn. ie More false positives will occur and less False negative.
- This will lead to company spending more money in retaining drivers.
- This model is important when retaining driver is more important than cost saving

Case 2: Business wants to make absolutely sure that the predicted driver will churn.

- In this case I will increase the threshold value. This will lead to increase in precision but decrease in recall
- This means less number of drivers will be predicted to churn. ie Less false positives and more false negative
- There is a possibility that the model will miss the driver that would actually churn. In this case the company will lose a experienced driver.
- This model is important when cost saving has higher priority than retaining driver

Recommendation

- The company should focus more on Recall as this will help them retain more drivers.
- This might lead to higher cost but in the longer run it will be beneficial for the company