ola_ensemble_solution

December 8, 2022

1 Ola case study - Ensemble Learning

1.1 Business Context

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

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

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

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

1.2 Column Profiling:

- MMMM-YY : Reporting Date (Monthly)
- Driver_ID : Unique id for drivers
- Age : Age of the driver
- Gender : Gender of the driver – Male : 0, Female: 1
- City : City Code of the driver
- Education_Level : Education level – 0 for 10+ ,1 for 12+ ,2 for graduate
- Income : Monthly average Income of the driver
- Date Of Joining : Joining date 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)

Concepts Tested:

Ensemble Learning- Bagging Ensemble Learning- Boosting KNN Imputation of Missing Values Working with an imbalanced dataset

1.3 Additional Views:

We will begin by importing the data-set, understanding the data-set and its structure. We will then carry out several feature engineering and data pre-processing activities such as creating derived features (rating_inc, income_inc, tenure_months etc), creating target variable (churn), and aggregating data at drivers level. Post data processing, We will split our data set into training and test sets. For missing values imputation (if necessary) we will use KNN. We will examine outliers and treat them (In general, tree algorithms are relatively robus to outliers, but to avoid decision trees with larger depth over-fitting to outliers, we will address outliers). We will build and evaluate 3 types of models; Decision tree classifier (for benchmarking purpose), bagging model (using random forest classifier), and gradient boosted model (using xgboost). For each model, we will attempt to find optimal parameters using hyper-parameter tuning with k-fold cross validation and build the final models with optimal parameters. We then evaluate each model by reporting test scores, various classification metrics, and plotting ROC/AUC curves. At the end, we provide relevant business recommendations.

2 Solution

[1]:
```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
import seaborn as sns

#set seaborn theme
sns.set_theme(style="whitegrid", palette="pastel")

#read dataset
origdata = pd.read_csv('data/ola_driver_scaler.csv', index_col=0)
origdata.head()
```

[1]:

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

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

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

[2]:
```
origdata.shape
```

[2]:
```
(19104, 13)
```

[3]:
```
origdata.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 19104 entries, 0 to 19103
Data columns (total 13 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   MMM-YY                19104 non-null  object
 1   Driver_ID             19104 non-null  int64
 2   Age                   19043 non-null  float64
 3   Gender                19052 non-null  float64
 4   City                  19104 non-null  object
 5   Education_Level       19104 non-null  int64
 6   Income                19104 non-null  int64
 7   Dateofjoining         19104 non-null  object
 8   LastWorkingDate       1616 non-null   object
 9   Joining Designation   19104 non-null  int64
 10  Grade                 19104 non-null  int64
 11  Total Business Value  19104 non-null  int64
 12  Quarterly Rating      19104 non-null  int64
dtypes: float64(2), int64(7), object(4)
memory usage: 2.0+ MB
```

[4]:
```
origdata.describe()
```

[4]:
```
          Driver_ID           Age        Gender  Education_Level  \
count  19104.000000  19043.000000  19052.000000     19104.000000
mean    1415.591133     34.668435      0.418749         1.021671
std      810.705321      6.257912      0.493367         0.800167
min        1.000000     21.000000      0.000000         0.000000
25%      710.000000     30.000000      0.000000         0.000000
50%     1417.000000     34.000000      0.000000         1.000000
75%     2137.000000     39.000000      1.000000         2.000000
max     2788.000000     58.000000      1.000000         2.000000

              Income  Joining Designation         Grade  Total Business Value  \
```

```
count    19104.000000          19104.000000  19104.000000          1.910400e+04
mean     65652.025126              1.690536      2.252670          5.716621e+05
std      30914.515344              0.836984      1.026512          1.128312e+06
min      10747.000000              1.000000      1.000000         -6.000000e+06
25%      42383.000000              1.000000      1.000000          0.000000e+00
50%      60087.000000              1.000000      2.000000          2.500000e+05
75%      83969.000000              2.000000      3.000000          6.997000e+05
max     188418.000000              5.000000      5.000000          3.374772e+07

       Quarterly Rating
count      19104.000000
mean           2.008899
std            1.009832
min            1.000000
25%            1.000000
50%            2.000000
75%            3.000000
max            4.000000
```

[5]:
```
#check number of unique drivers
unique_driver_cnt = origdata['Driver_ID'].nunique()
unique_driver_cnt
```

[5]:
```
2381
```

[6]:
```
#number of records per driver
origdata.groupby('Driver_ID')['Driver_ID'].count().agg(['max', 'min', 'mean'])
```

[6]:
```
max     24.00000
min      1.00000
mean     8.02352
Name: Driver_ID, dtype: float64
```

[7]:
```
#check number of drives who churned
n_churn = origdata[~origdata['LastWorkingDate'].isna()]['Driver_ID'].nunique()
# n_retain = origdata.shape[0] - n_churn
print(f'Drivers who churned: {n_churn}')
# print(f'churn rate = {n_churn / unique_driver_cnt * 100}')
```

```
Drivers who churned: 1616
```

[8]:
```
#check categorical variables
for col in ['Gender', 'City', 'Education_Level', 'Joining Designation',␣
↪'Grade', 'Quarterly Rating']:
    print(f'unique values for {col}: {origdata[col].nunique()}')
    print(origdata[col].value_counts())
```

```
unique values for Gender: 2
```

```
0.0    11074
1.0     7978
Name: Gender, dtype: int64
unique values for City: 29
C20    1008
C29     900
C26     869
C22     809
C27     786
C15     761
C10     744
C12     727
C8      712
C16     709
C28     683
C1      677
C6      660
C5      656
C14     648
C3      637
C24     614
C7      609
C21     603
C25     584
C19     579
C4      578
C13     569
C18     544
C23     538
C9      520
C2      472
C11     468
C17     440
Name: City, dtype: int64
unique values for Education_Level: 3
1    6864
2    6327
0    5913
Name: Education_Level, dtype: int64
unique values for Joining Designation: 5
1    9831
2    5955
3    2847
4     341
5     130
Name: Joining Designation, dtype: int64
unique values for Grade: 5
2    6627
```

```
1    5202
3    4826
4    2144
5     305
Name: Grade, dtype: int64
unique values for Quarterly Rating: 4
1    7679
2    5553
3    3895
4    1977
Name: Quarterly Rating, dtype: int64
```

[9]:
```
print('check if the following features are Driver level features (i.e. if stay␣
 ↪constant for each driver)')

for col in ['Grade', 'Joining Designation']:
    print(origdata.groupby('Driver_ID')[col].nunique().value_counts())
```

```
check if the following features are Driver level features (i.e. if stay constant
for each driver)
1    2337
2      44
Name: Grade, dtype: int64
1    2381
Name: Joining Designation, dtype: int64
```

[10]:
```
#check relation between Grade and income for a given driver
print(origdata.groupby(['Driver_ID', 'Grade']).ngroups)
print(origdata.groupby(['Driver_ID', 'Income']).ngroups)
print(origdata.groupby(['Driver_ID', 'Grade', 'Income']).ngroups)
```

```
2425
2425
2425
```

Observations:

1. The dataset has 19104 rows and 17 columns.
2. 'MMM-YY', 'Dateofjoining', and 'LastWorkingDate' represent datetime values. We will convert them to datetime columns.
3. 'Total Business Value', 'Age', 'Income' are continuous features. We will convert them to numeric type.
4. 'Gender', 'City', 'Education_Level', 'Joining Designation', 'Grade', 'Quarterly Rating' are categorical features. We will convert them to categorical variables.
5. The dataset has data for 2381 unique drivers, out of which 1616 drivers churned (~68%)
6. We observe that for a given driver, 'Grade' and 'Income' change together.

2.1 Missing value detection

[11]:
```
#check for missing values
missing_cnt = origdata.isna().sum()
missing_cnt.to_frame(name='missing count').assign(missing_percent = np.
 ↪round((missing_cnt/origdata.shape[0])*100,2))
```

[11]:

	missing count	missing_percent
MMM-YY	0	0.00
Driver_ID	0	0.00
Age	61	0.32
Gender	52	0.27
City	0	0.00
Education_Level	0	0.00
Income	0	0.00
Dateofjoining	0	0.00
LastWorkingDate	17488	91.54
Joining Designation	0	0.00
Grade	0	0.00
Total Business Value	0	0.00
Quarterly Rating	0	0.00

Observations:

1. Age, Gender, and LastWorkingDate columns have missing values.
2. Missing values for LastWorkingDate column indicate continuity of employment, whereas, the presence of the values indicate employment termination. We will use this information to create the target column 'Churn'. After data aggregation we will not need this column further.
3. Age and Gender have a few missing values. Since both of these are Driver attributes, and since we plan to aggregate on driver_id, we will check for missing values again after aggregation. In case we still find missing values, we will use knn missing value imputation technique.

2.2 Data pre-processing, feature engineering, and data aggregation

2.2.1 convert datetime variables to datetime64

[12]:
```
# convert MMM-YY, Dateofjoining, and LastWorkingDate to datetime
origdata['MMM-YY'] = origdata['MMM-YY'].astype('datetime64')
origdata['Dateofjoining'] = origdata['Dateofjoining'].astype('datetime64')
origdata['LastWorkingDate'] = origdata['LastWorkingDate'].astype('datetime64')
```

2.2.2 set category dtype for categorical variables

Note: Quarterly Rating is a categorical variable, but we will not convert it to category for now as we will later need to compare its successive values.

[13]:
```

```

```
# set category dtype for categorical variables.
for col in ['Gender', 'City', 'Education_Level', 'Joining Designation',␣
 ↪'Grade']:
    origdata[col] = origdata[col].astype("category")

origdata.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 19104 entries, 0 to 19103
Data columns (total 13 columns):
 #   Column                Non-Null Count  Dtype
---  ------                --------------  -----
 0   MMM-YY                19104 non-null  datetime64[ns]
 1   Driver_ID             19104 non-null  int64
 2   Age                   19043 non-null  float64
 3   Gender                19052 non-null  category
 4   City                  19104 non-null  category
 5   Education_Level       19104 non-null  category
 6   Income                19104 non-null  int64
 7   Dateofjoining         19104 non-null  datetime64[ns]
 8   LastWorkingDate       1616 non-null   datetime64[ns]
 9   Joining Designation   19104 non-null  category
 10  Grade                 19104 non-null  category
 11  Total Business Value  19104 non-null  int64
 12  Quarterly Rating      19104 non-null  int64
dtypes: category(5), datetime64[ns](3), float64(1), int64(4)
memory usage: 1.4 MB
```

2.2.3 create target variable 'Churn'

[14]:
```
#mask to select rows where LastWorkingDate is specified
churn_mask = ~origdata['LastWorkingDate'].isna()

#create target variable
origdata = origdata.assign(Churn = np.zeros((origdata.shape[0]), dtype=int))

#set churn=1 where LastWorkingDate is mentioned
origdata.loc[churn_mask, 'Churn'] = 1

#verify
origdata['Churn'].value_counts()
```

[14]:
```
0    17488
1     1616
Name: Churn, dtype: int64
```

2.2.4 create quarterly rating increase flag 'Rating_Inc'

[15]:
```
#check how many times quarterly rating has changed for drivers
def applyongrp(g):
    curr_rat = g.reset_index()['Quarterly Rating']
    prev_rat = curr_rat[0:1].append(curr_rat[:-1], ignore_index=True)

    delta = curr_rat - prev_rat
    return delta.map(lambda x: np.sign(x))

origdata.sort_values(by='MMM-YY').groupby('Driver_ID').apply(applyongrp).
↪value_counts()
```

[15]:
```
 0    16513
-1     1346
 1     1245
Name: Quarterly Rating, dtype: int64
```

Observation: In most instances(for 16513 records), a driver's quarterly rating does not change between from the previous row (that is between months). In 1345 instances, quarterly rating decreased, while in 1245 cases, it increased. A driver can go through a series of rating changes (both positive and negative) in his/her tenure. Therefore, we will consider the most recent rating change. If that rating change is positive, we will set the new variable to 1, otherwise to 0.

[16]:
```
#For each driver_id, returns 1 if the last rating change was positive, 0␣
↪otherwise
def set_Rating_Inc(curr_rat):
    prev_rat = curr_rat.iloc[0:1].append(curr_rat.iloc[:-1], ignore_index=True)
    prev_rat.index = curr_rat.index #imp to ensure before taking difference␣
↪between curr and prev rate

    delta = (curr_rat - prev_rat).map(lambda x: np.sign(x))

    changes = delta[delta != 0] #get rating changes
    is_last_change_positive = 0 if (len(changes) == 0 or changes.iloc[-1] ==␣
↪-1) else 1

    return is_last_change_positive

#create Rating_Inc column: 1 if the last rating change was positive, 0␣
↪otherwise.
origdata['Rating_Inc'] = origdata.sort_values(by='MMM-YY').
↪groupby('Driver_ID')['Quarterly Rating'].transform(set_Rating_Inc)
```

[17]:
```
origdata.sort_values(by='MMM-YY').groupby('Driver_ID')['Rating_Inc'].max().
↪value_counts()
```

[17]:
```
0    2013
1     368
Name: Rating_Inc, dtype: int64
```

Observation: The last quarterly rating change was positive for 368 drivers. For the remaining 2013 drivers, either the rating did not change, or the last rating change was negative.

2.2.5 create monthly income increase flag 'Income_Inc'

[18]:
```
#check how many times quarterly rating has changed for drivers
def applyongrp(g):
    curr_rat = g.reset_index()['Income']
    prev_rat = curr_rat[0:1].append(curr_rat[:-1], ignore_index=True)

    delta = curr_rat - prev_rat
    return delta.map(lambda x: np.sign(x))

origdata.sort_values(by='MMM-YY').groupby('Driver_ID').apply(applyongrp).
↪value_counts()
```

[18]:
```
 0    19059
 1       44
-1        1
Name: Income, dtype: int64
```

Observation: In most instances (19059), a driver's quarterly income has not changed. There are 44 instances where a driver's monthly income increased, while in 1 case, it decreased. A driver can go hypothetically go through a series of changes in his income (both positive and negative) in his/her tenure. Therefore, we will consider the most recent income change. If that change is positive, we will set the new variable to 1, otherwise to 0.

[19]:
```
#For each driver_id, returns 1 if the last rating change was positive, 0␣
↪otherwise
def set_Income_Inc(curr_inc):
    prev_inc = curr_inc.iloc[0:1].append(curr_inc.iloc[:-1], ignore_index=True)
    prev_inc.index = curr_inc.index #imp to ensure before taking difference␣
↪between curr and prev rate

    delta = (curr_inc - prev_inc).map(lambda x: np.sign(x))

    changes = delta[delta != 0] #get income changes
    is_last_change_positive = 0 if (len(changes) == 0 or changes.iloc[-1] ==␣
↪-1) else 1

    return is_last_change_positive
```

```
#create Income_Inc column: 1 if the last income change was positive, 0␣
 ↪otherwise.
origdata['Income_Inc'] = origdata.sort_values(by='MMM-YY').
 ↪groupby('Driver_ID')['Income'].transform(set_Income_Inc)
```

[20]:
```
origdata.sort_values(by='MMM-YY').groupby('Driver_ID')['Income_Inc'].min().
 ↪value_counts()
```

[20]:
```
0    2338
1      43
Name: Income_Inc, dtype: int64
```

Observation: The last income change was positive for 43 drivers. For the remaining 2338 drivers, either their income remained the same during their (ongoing)tenure, or the last income change was negative.

2.2.6 create tenure (employment length) variable 'Tenure_Months'

For the rows where 'LastWorkingDate' is available, we can compute tenure as the different between their 'LastWorkingDate' and 'Dateofjoining'. For the rows where 'LastWorkingDate' is not available, we can compute tenure as the difference between 'MMM-YY' and 'Dateofjoining'. Later, while aggregating data, we will select the highest tenure value per driver. Thus this variable will either indicate completed tenure length (if the driver churned), or ongoing tenure length (for non-churning driver)

[21]:
```
#create Tenure_Months column
origdata = origdata.assign(Tenure_Months = np.zeros(origdata.shape[0],␣
 ↪dtype=int))

def month_diff(a, b):
    return int(np.round(((a - b) / np.timedelta64(1, 'M')),0))

def compute_tenure(row):
    if(not pd.isna(row['LastWorkingDate'])):
        row['Tenure_Months'] = month_diff(row['LastWorkingDate'],␣
 ↪row['Dateofjoining'])
    else:
        # when a driver joins in a specific month, in that month, dateofjoining␣
 ↪> MMM-YY. For each month after that MMM-YY > dateofjoining.
        # so we just take absolute difference.
        row['Tenure_Months'] = abs(month_diff(row['MMM-YY'],␣
 ↪row['Dateofjoining']))
    return row

origdata = origdata.transform(compute_tenure, axis=1)
```

2.3 Data Aggregation

We will now aggregate data at driver's level. We first sort the dataframe in descending order by 'MMM-YY' column, and then group by 'Driver_ID' We will apply the following aggregation rules for different columns.

1. 'Driver_ID', 'Age', 'Gender', 'City', 'Education_Level', 'Dateofjoining', 'Joining Designation', 'Rating_Inc', 'Income_Inc', and 'Churn' are group level columns. We can take the first value from each group.
2. 'Income_Inc', 'Rating_Inc': Also group level columns. We can take first values.
3. 'MMM-YY', 'Dateofjoining', 'LastWorkingDate' - we can remove theses columns as we have created derived columns such as 'Tenure_Months' and 'Churn'. We do need specific dates in our models.
4. 'Driver_ID' can be removed as well after aggregation along the similar reasoning.
5. 'Tenure_Months' - we will take maximum value.
6. 'Total Business Value' - we will take average value.
7. 'Grade', 'Quarterly Rating', 'Income': we take the first (most recent) values for these columns.

[22]:
```
df = origdata.sort_values(['MMM-YY'], ascending=[True]).groupby('Driver_ID').
↪agg({
    'Age' : 'first',
    'Gender' : 'first',
    'City' : 'first',
    'Education_Level' : 'first',
    'Rating_Inc' : 'first',
    'Income_Inc' : 'first',
    'Joining Designation' : 'first',

    'Tenure_Months' : 'max',
    'Churn' : 'max',
    'Total Business Value' : 'mean',

    'Grade' : 'first',
    'Quarterly Rating' : 'first',
    'Income' : 'first'
}).rename(columns={
    'Total Business Value': 'Monthly Business Value'
}).reset_index()

df = df.drop('Driver_ID', axis=1)
df_bk = df.copy(deep=True)
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 2381 entries, 0 to 2380
Data columns (total 13 columns):
```

```
 #   Column                 Non-Null Count  Dtype
---  ------                 --------------  -----
 0   Age                    2381 non-null   float64
 1   Gender                 2381 non-null   float64
 2   City                   2381 non-null   object
 3   Education_Level        2381 non-null   int64
 4   Rating_Inc             2381 non-null   int64
 5   Income_Inc             2381 non-null   int64
 6   Joining Designation    2381 non-null   int64
 7   Tenure_Months          2381 non-null   int64
 8   Churn                  2381 non-null   int64
 9   Monthly Business Value 2381 non-null   float64
 10  Grade                  2381 non-null   int64
 11  Quarterly Rating       2381 non-null   int64
 12  Income                 2381 non-null   int64
dtypes: float64(3), int64(9), object(1)
memory usage: 241.9+ KB
```

[23]:
```
df.describe().T
```

[23]:

	count	mean	std	min	\
Age	2381.0	33.090718	5.840686	21.000000	
Gender	2381.0	0.410332	0.491997	0.000000	
Education_Level	2381.0	1.007560	0.816290	0.000000	
Rating_Inc	2381.0	0.154557	0.361558	0.000000	
Income_Inc	2381.0	0.018060	0.133195	0.000000	
Joining Designation	2381.0	1.820244	0.841433	1.000000	
Tenure_Months	2381.0	13.979840	18.511898	0.000000	
Churn	2381.0	0.678706	0.467071	0.000000	
Monthly Business Value	2381.0	312085.359327	449570.506711	-197932.857143	
Grade	2381.0	2.078538	0.931321	1.000000	
Quarterly Rating	2381.0	1.486350	0.834348	1.000000	
Income	2381.0	59209.060899	28275.899087	10747.000000	

	25%	50%	75%	max
Age	29.0	33.000000	37.00	58.0
Gender	0.0	0.000000	1.00	1.0
Education_Level	0.0	1.000000	2.00	2.0
Rating_Inc	0.0	0.000000	0.00	1.0
Income_Inc	0.0	0.000000	0.00	1.0
Joining Designation	1.0	2.000000	2.00	5.0
Tenure_Months	3.0	6.000000	15.00	92.0
Churn	0.0	1.000000	1.00	1.0
Monthly Business Value	0.0	150624.444444	429498.75	3972127.5
Grade	1.0	2.000000	3.00	5.0
Quarterly Rating	1.0	1.000000	2.00	4.0
Income	39104.0	55276.000000	75765.00	188418.0

Observations:

1. We note that after aggregating data, we no longer have any missing values in the dataset. However, Monthly Business Value has a lot of zero values. This may indicate missing data. We will explore this further in the next section.

2. For the rest of the case_study, we will be working on the aggregated data-set

2.4 Univariate and bi-variate analysis

2.4.1 utility functions

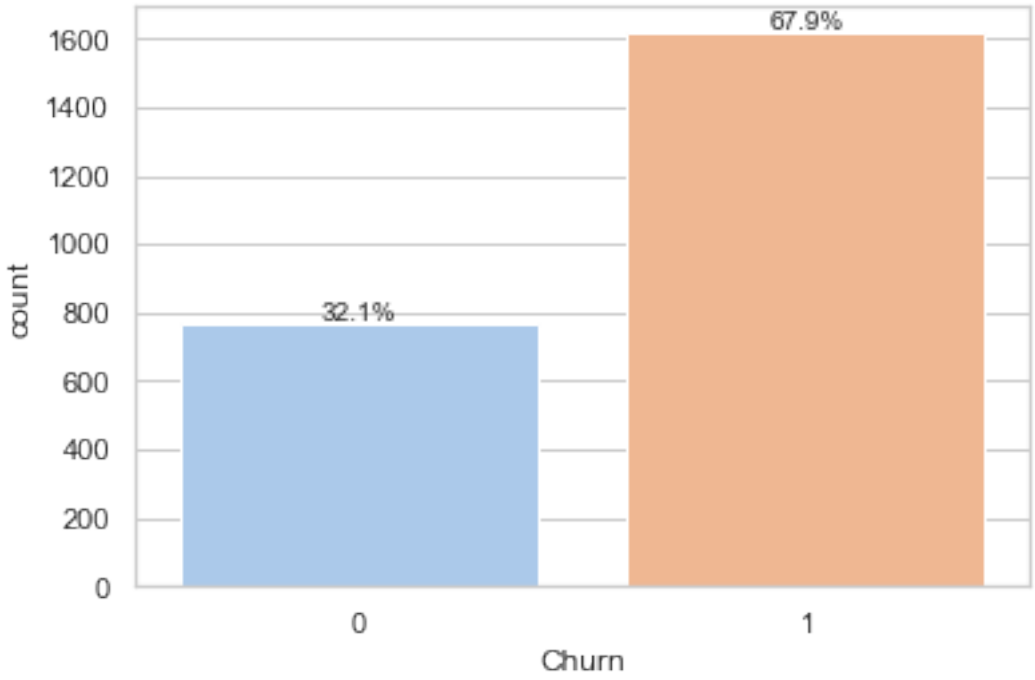
[160]:
```
#Common utility functions

# plots univariate plot(count) for feature x.
# Optionally also plots bivariate plot (count or scatter) between x and y.
def show_cat_plots(x, y=None, axs=None, title=None):
    total = float(x.shape[0])
    bivar=(y is not None)

    #initialize
    if(axs is None):
        size = None
        if(bivar):
            size = (12, 4) if (size is None) else size
            gridrows = 1
            gridcols = 2
        else:
            size = (6, 4) if (size is None) else size
            gridrows = 1
            gridcols = 1

        fig, ax = plt.subplots(gridrows, gridcols, figsize=size)

        axs = (ax[0], ax[1]) if (bivar) else (ax, None)

    #plot univariate
    ax_curr = axs[0]
    sns.countplot(x=x, ax=ax_curr)
    showpercent(ax_curr, total)

    #plot bivariate
    if(bivar):
        ax_curr = axs[1]
        sns.countplot(x=x, hue=y, ax=ax_curr);
        showpercent_with_hue(ax_curr, hue_levels = len(y.unique()),␣
↪x_levels=len(x.unique()))

    #set title if available
```

```
    if(title is not None):
        fig.suptitle(title)

    plt.show()


def show_num_plots(x, y=None, axs=None, title=None):
    total = float(x.shape[0])
    bivar=(y is not None)

    #initialize
    if(axs is None):
        size = None
        if(bivar):
            size = (12, 8) if (size is None) else size
            gridrows = 2
            gridcols = 2
        else:
            size = (12, 4) if (size is None) else size
            gridrows = 1
            gridcols = 2

        fig, ax = plt.subplots(gridrows, gridcols, figsize=size)
        axs = (ax[0][0], ax[0][1], ax[1][0], ax[1][1]) if (bivar) else (ax,␣
↪None)

    #plot univariate
    sns.histplot(x=x, kde=True, ax=axs[0])
    sns.boxplot(x=x, ax=axs[1])

    #plot bivariate
    if(bivar):
        sns.histplot(x=x, hue=y.astype('category'), kde=True, ax=axs[2])
        sns.boxplot(x=x, y=y.astype('category'), ax=axs[3])

    if(title is not None):
        fig.suptitle(title)

    plt.show()

def showpercent(ax, total):
    for p in ax.patches:
            percent = '{:.1f}%'.format(100 * p.get_height()/total)
            xpos = p.get_x() + p.get_width()/2
            ypos = p.get_height()
            ax.annotate(percent, (xpos, ypos),ha='center', va='bottom')
```

```
def showpercent_with_hue(ax, hue_levels, x_levels):

    heights_arr = np.array([p.get_height() for p in ax.patches])
    heights = np.array([p.get_height() for p in ax.patches]).
↪reshape((hue_levels, int(len(heights_arr)/hue_levels)))
    percents = np.round((heights * 100) / np.sum(heights, axis=0),1)
    perclist = percents.flatten(order='C') #flatten in column-major (F-style)␣
↪order

    for i in range(len(ax.patches)):
        p = ax.patches[i]
        percent = f'{perclist[i]}%'
        xpos = p.get_x() + p.get_width()/2
        ypos = p.get_height()
        ax.annotate(percent, (xpos, ypos),ha='center', va='bottom')
```

2.4.2 target variable 'Churn'

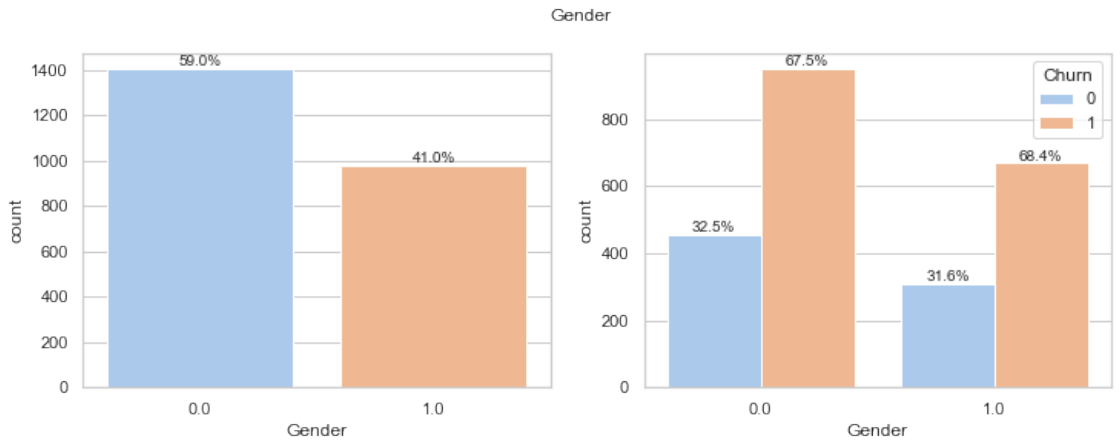
[25]:
```
#show plots for target variable 'Churn'
y = df['Churn']
show_cat_plots(y)
```

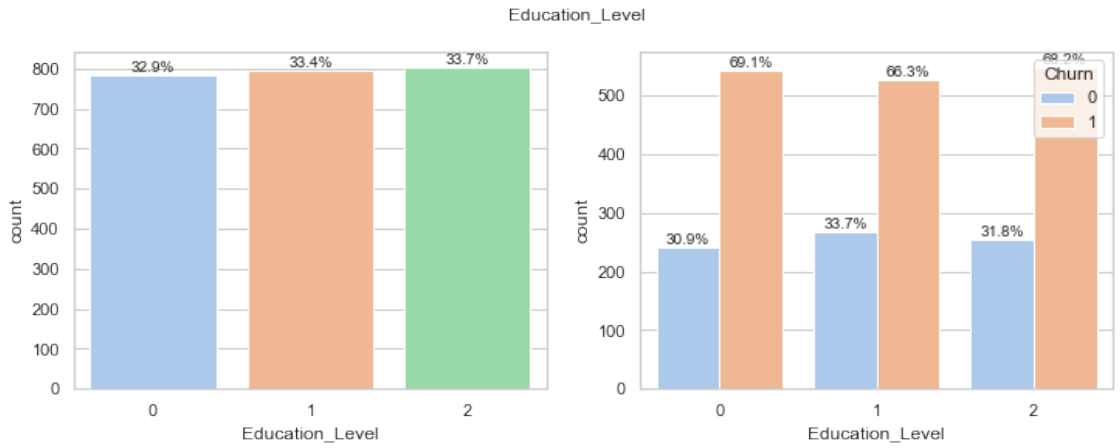


Observations: About 68% of the drivers have churned.

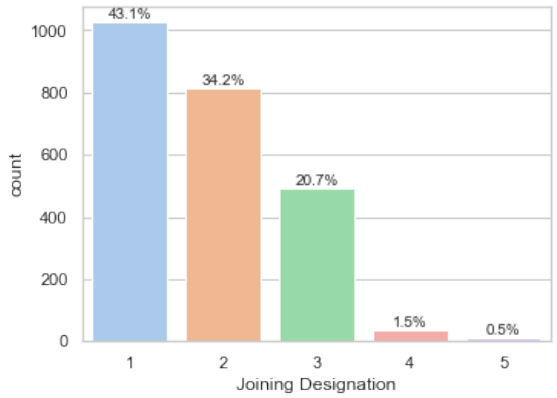
2.4.3 indepedent categorical variables

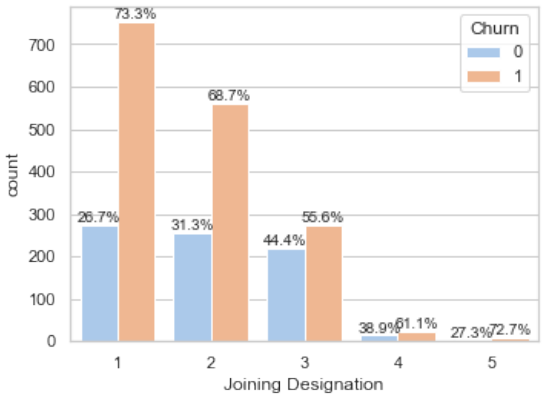
[26]:
```
#plots for categorical variables except city
for col in ['Gender', 'Education_Level', 'Joining Designation', 'Grade',␣
 ↪'Quarterly Rating', 'Rating_Inc', 'Income_Inc']:
    show_cat_plots(df[col], y, title=col)

#plots for city
fig, ax = plt.subplots(2, 1, figsize=(14, 10))
show_cat_plots(df['City'], df['Churn'], axs = ax)
fig.suptitle('City')
plt.show()
```



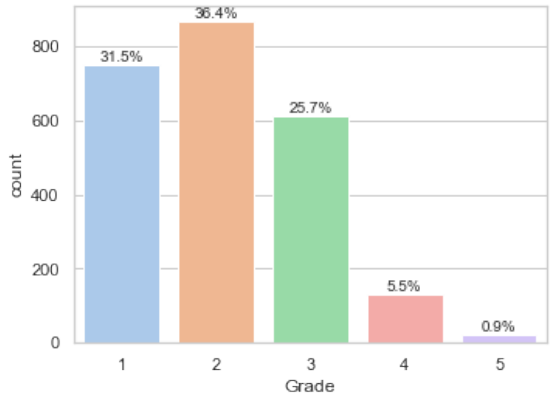


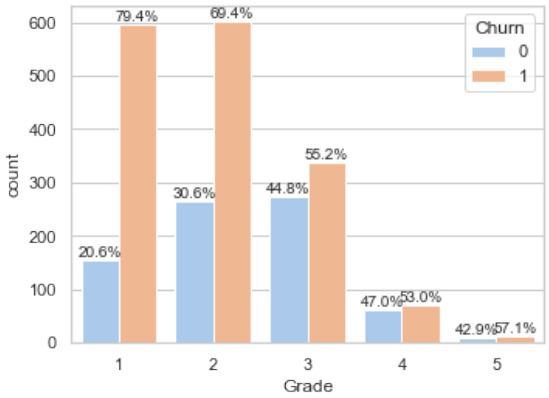
Joining Designation



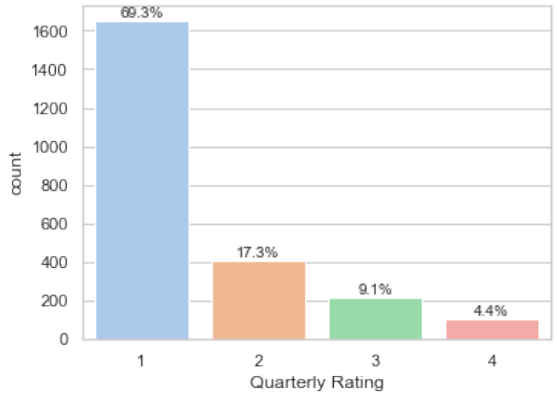


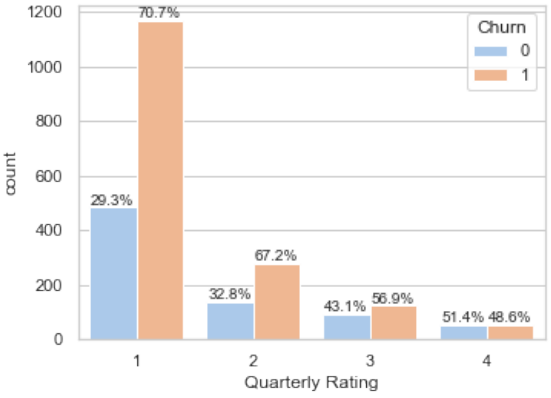
Grade



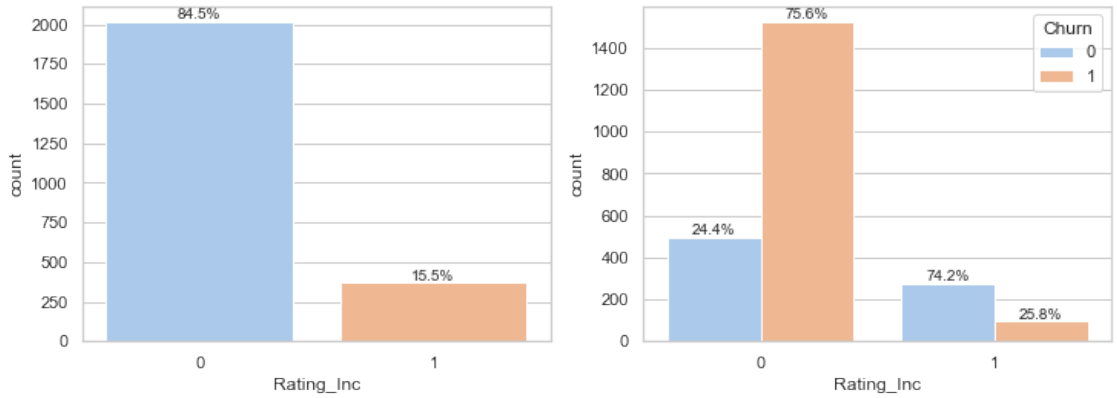


Quarterly Rating

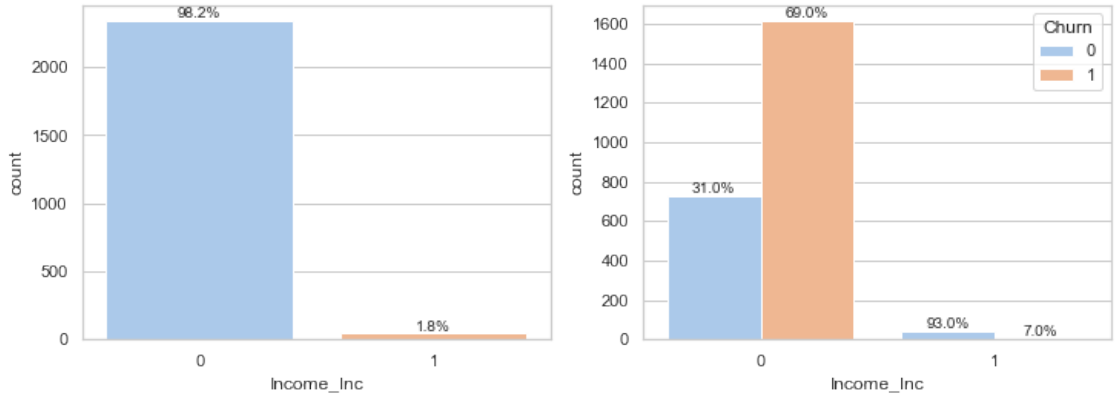


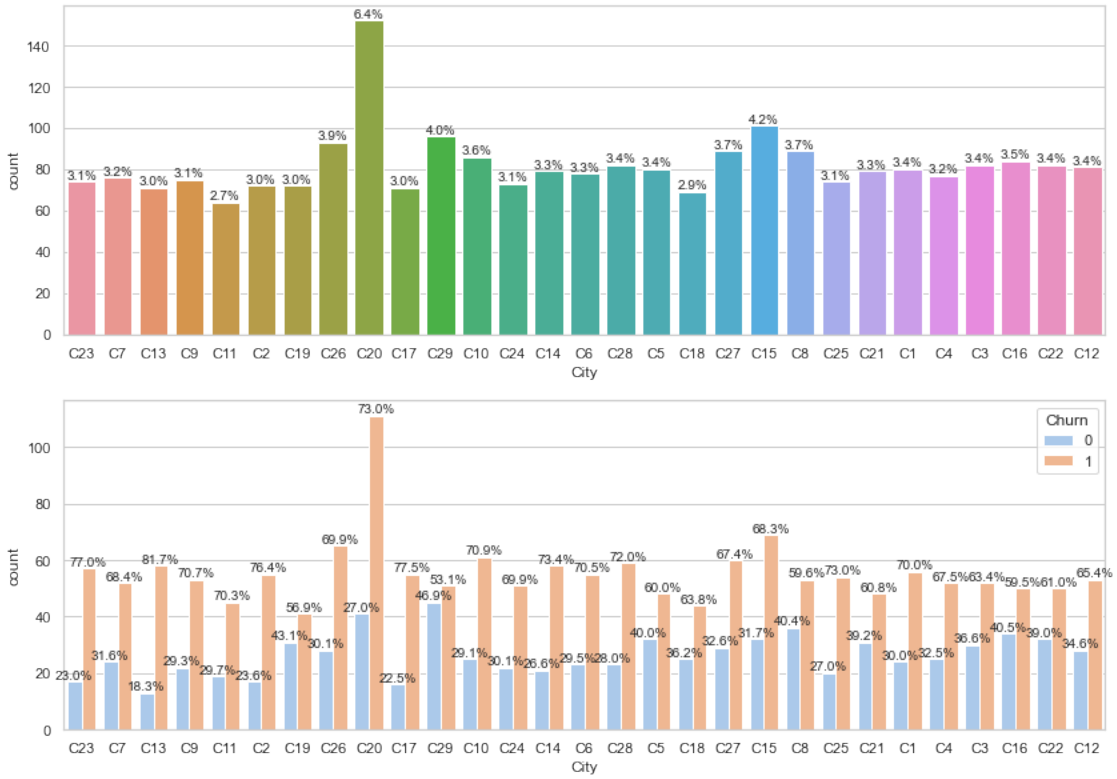


Rating_Inc



Income_Inc





[27]:
```
df.groupby('City').apply(lambda x: (x[x['Churn'] == 1].shape[0] / x.
↪shape[0])*100).sort_values()
```

[27]:
```
City
C29     53.125000
C19     56.944444
C16     59.523810
C8      59.550562
C5      60.000000
C21     60.759494
C22     60.975610
C3      63.414634
C18     63.768116
C12     65.432099
C27     67.415730
C4      67.532468
C15     68.316832
C7      68.421053
C24     69.863014
C26     69.892473
C1      70.000000
C11     70.312500
```

```
C6     70.512821
C9     70.666667
C10    70.930233
C28    71.951220
C25    72.972973
C20    73.026316
C14    73.417722
C2     76.388889
C23    77.027027
C17    77.464789
C13    81.690141
dtype: float64
```

Observations:

1. In the given data-set, around 68% of the total drivers have churned.
2. 59% of drivers are gender 0. 41% are gender 1. Churn rate for both the genders are close (67.5% and 68.4%), thus Gender doesn't seem like a significant factor in predicting churn rate.
3. The distribution of drivers with respect to education level is almost uniform. The churn rates for education level 0, level 1, and level 2 are 69.1%, 66.3%, 68.2%. Thus education level doesn't seem like a very important factor in predicting churn rate.
4. The proportion of drivers having joining designation values as 1, 2, 3, 4, and 5 are respectively 43%, 34%, 21%, 1.5%, and 0.5%. Thus around 98% of the drivers have joining designation 1,2 or 3. For these top 3 joining designations (1, 2, and 3), churn rates are 73.3%, 68.7%, and 56.6%. **Thus joining designation appears to be an important factor in predicting churn.**
5. The proportion of drivers with Grades 1, 2, 3, 4, and 5 are respectively 31.5%, 36.4%, 25.7%, 5,5%, and 1%. The churn rates for these grades are 79.4%, 69.4%, 55%, 53%, and 57%. Thus churn rate is very high for grade 1, around overall average for grade 2, and lower than avergae for the ramining grades. **Thus grade appears to impact churn. In general, churn rate decreases with higher grades, except from grade 4 to 5, where it increases somewhat.**
6. The proportion of drivers with Quarterly rating 1, 2, 3, and 4 are 70%, 17.3%, 9.1%, and 4.4% respectively. Their churn rates are 70.7%, 67%, 57%, and 48.6%. Thus churn rate for rating 1 is very high. For rating 2 is similar to the overall average churn rate. For rating 3 and 4 it reduces. **Thus Quarterly rating does appear to impact churn. In general, drivers with higher quarterly rating tend to churn less.**
7. For 15.5% drivers, the last quarterly rating change was positive. For the remaining drivers, either their rating did not change throughout their tenure, or it decreased. Among the drivers with positive rating change, we see that churn rate drops to 25.8%. Whereas among drivers with zero or negative rating change, churn rate increases to 75.6%. **Thus rating_inc derived feature is significant in predicting churn outcome.**
8. For around 98.2% of drivers, their income did not change during their tenure (one one driver's

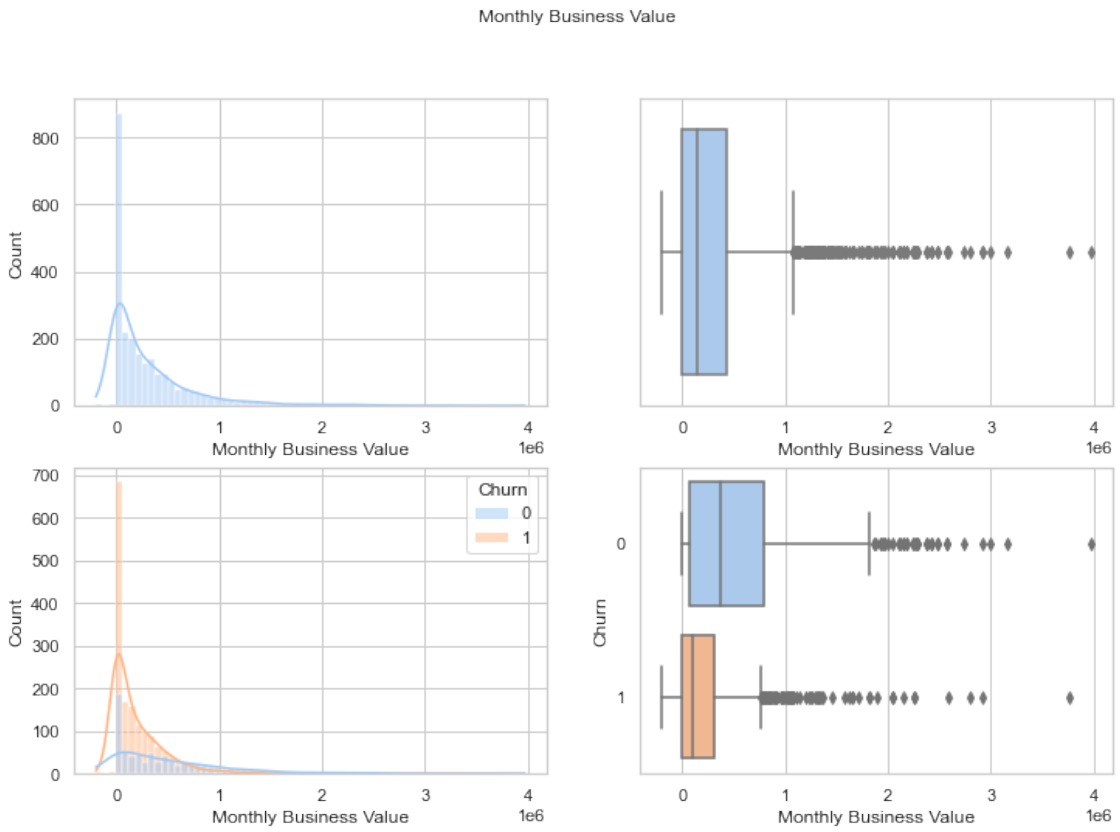
income decreased). For about 1.8% of the drivers, their income increased in their tenure. Among these 1.8% drivers, churn rate is much lower at 7%. **So, drivers whose income increased, are much more likely to stay. Having said that, we have very few data points to generalize this.**

9. There are total 29 cities, each one of them contributing between 2.7% to 4.2% (with one exception of city C20 contributing 6.4%) to the overall driver population. Top 3 cities with highest churn rates are C13 (81%), C17(77%), and C23(77%). Top 3 cities with lowest churn rates are C29(53%), C19(56%), and C16(59%). **Thus city seems like an important factor in predicting churn.**

2.4.4 Numerical indepedent variables

Monthly Business Value

[28]:
```
col = 'Monthly Business Value'
show_num_plots(df[col], y , title=col)
print(f'{col}: overall mean - {df[col].mean()}, overall median - {df[col].
 ↪median()}')
print('\n',np.round(df.groupby('Churn')[col].describe(), 1))
```



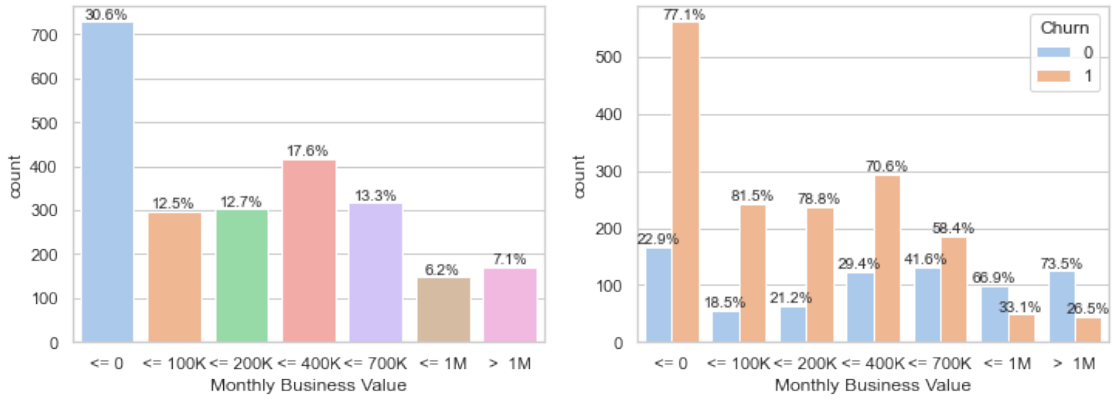

```
Monthly Business Value: overall mean - 312085.35932712955, overall median -
150624.44444444444
```

```
          count      mean       std        min      25%       50%       75%  \
Churn
0         765.0  527430.9  583827.0        0.0  66408.0  366410.0  786370.4
1        1616.0  210142.7  322862.8 -197932.9      0.0  100922.0  305637.3

              max
Churn
0       3972127.5
1       3759614.4
```

[130]:
```
df['Monthly Business Value'].describe()
```

[130]:
```
count    2.381000e+03
mean     3.120854e+05
std      4.495705e+05
min     -1.979329e+05
25%      0.000000e+00
50%      1.506244e+05
75%      4.294988e+05
max      3.972128e+06
Name: Monthly Business Value, dtype: float64
```

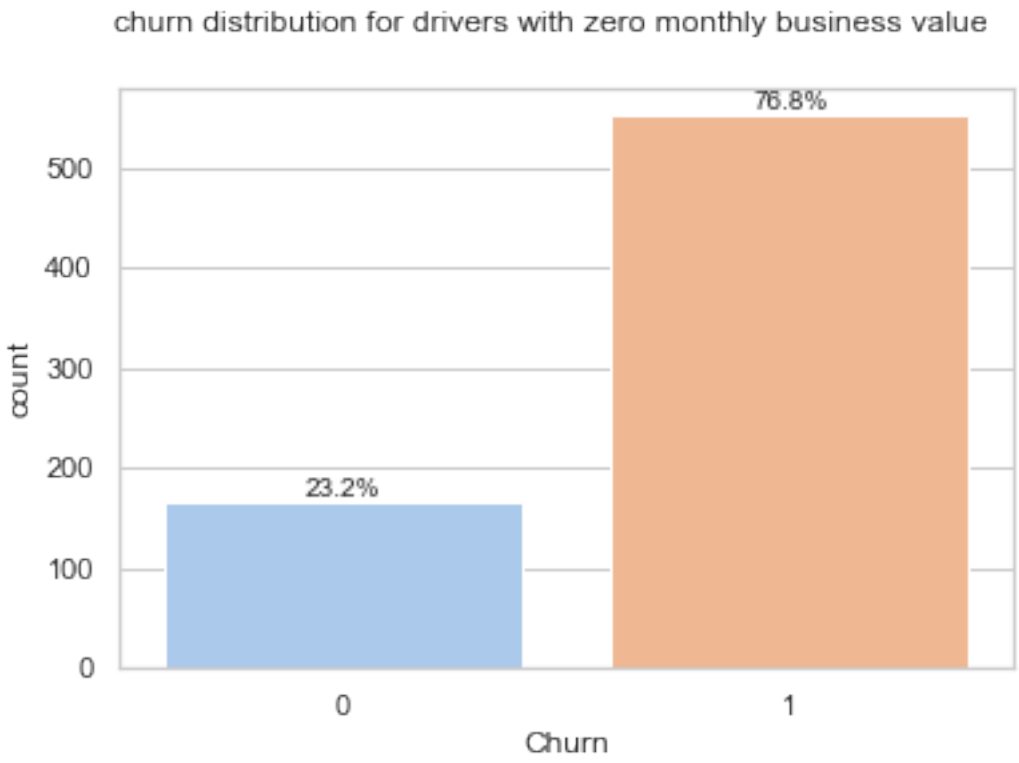
[161]:
```
mbv_binned = pd.cut(df['Monthly Business Value'], bins=[-200000 , 0, 100000,␣
↪200000, 400000, 700000, 1000000, 10000000], labels=['<= 0', '<= 100K', '<=␣
↪200K', '<= 400K', '<= 700K', '<= 1M', '>  1M'])
show_cat_plots(mbv_binned, y)
```

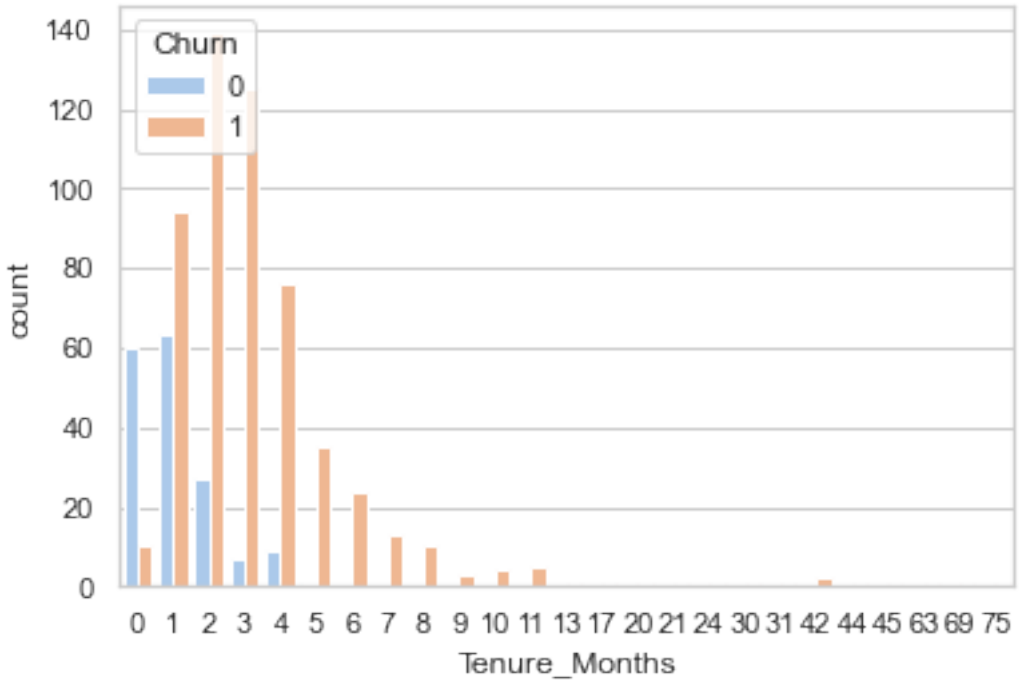


[29]:
```
#check number of records with zero monthly business value

df_zero_mbv = df[df['Monthly Business Value'] == 0]
print(f'Number of drivers with zero monthly business value: {df_zero_mbv.
↪shape[0]}')
```

```
show_cat_plots(df_zero_mbv['Churn'], title='churn distribution for drivers with␣
 ↪zero monthly business value')
plt.show()

sns.countplot(data=df_zero_mbv, x='Tenure_Months', hue='Churn')
plt.show()
```

```
Number of drivers with zero monthly business value: 719
```





Observations:

1. The distribution of monthly business value has a high concentration around zero and overall it's a positively skewed distribution with several outliers towards higher positive side.
2. The overall (mean, median) values for monthly business value is (312085, 150624). For drivers who churned, both mean and median values drop to (210142, 100922). For drivers who didn't churn, mean and median monthly business values increase to (527430, 366410).
3. After binning Monthly business value, we observe that Churn rates are very high for drivers whose monthly business value is <200K, higher than overall average for <400K. For drivers having monthly business value between >700K and 1M, churn rate reduces, and for drivers with MBV > 1M, churn rates are low. **Thus, drivers with lower monthly business value tend to churn more.** Thus monthly business value is a significant factor in predicting churn.

Imp Note:

Out of 2381 drivers, 719 drivers have monthly business value equal to zero (around 30%). Our intuition tells us that active drivers will rarely have monthly business value exactly equal to zero. This can indicate two things. 1. This is a potential case of missing/noisy data. 2. The drivers with zero monthly business value are inactive.

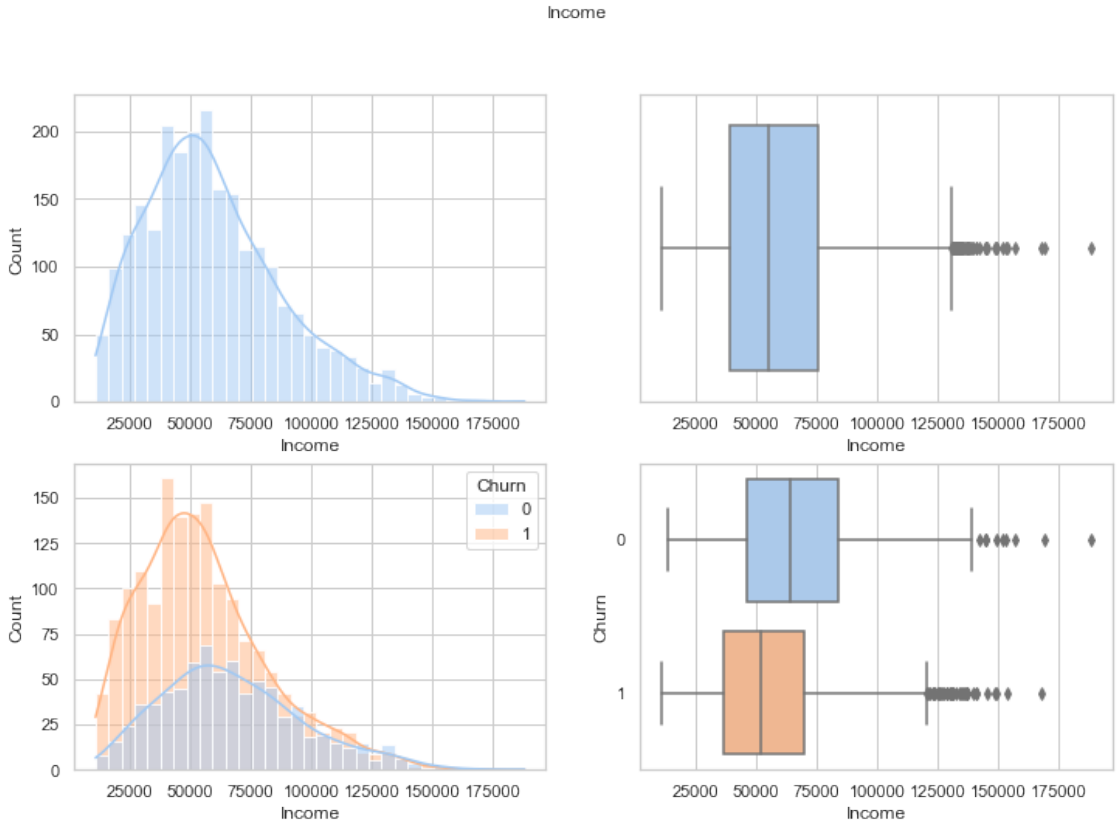
We notice that around 77% drivers with zero monthly business value churned. This is higher than the overall churn rate. When We further check the distribution of Tenure_Months for these drivers, we notice that for drivers with zero tenure_months, churn rate is quite low. This most likely represents those drivers who have just joined, but whose earnings have not yet been recorded by the system. For drivers with tenure_months > 1, however, we see high churn rate. As discussed above, we can address these set of drivers in two ways.

1. Consider this the case of missing data. Use imputation technique (such as KNeighborregressi
2. Consider these set of drivers as inactive; In that case, zero monthly business value is a v

In the absence of any further clarity, we take the second approach in this case study. We will not attempt to update records with zero monthly business value.

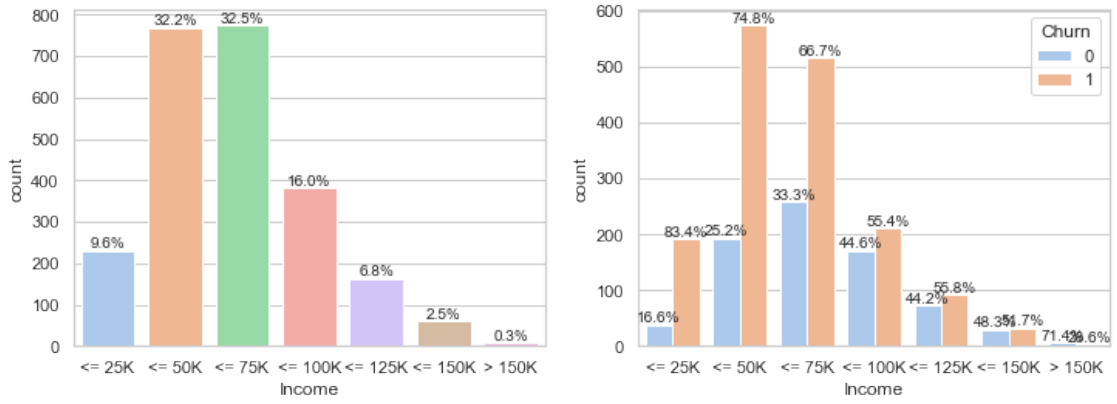
Income

[30]:
```
col = 'Income'
show_num_plots(df[col], y , title=col)
print(f'{col}: overall mean - {df[col].mean()}, overall median - {df[col].
 ↪median()}')
print('\n',np.round(df.groupby('Churn')[col].describe(), 1))
```




```
Income: overall mean - 59209.06089878202, overall median - 55276.0

         count     mean      std      min      25%      50%      75%       max
Churn
0        765.0  67303.5  29392.0  12938.0  46131.0  63632.0  83969.0  188418.0
1       1616.0  55377.2  26904.4  10747.0  36117.5  51630.0  69816.8  167758.0
```

[162]:
```
income_binned = pd.cut(df['Income'], bins=[0, 25000, 50000, 75000, 100000,␣
↪125000, 150000, 300000], labels=['<= 25K', '<= 50K', '<= 75K', '<= 100K',␣
↪'<= 125K', '<= 150K', '> 150K'])
show_cat_plots(income_binned, y)
```

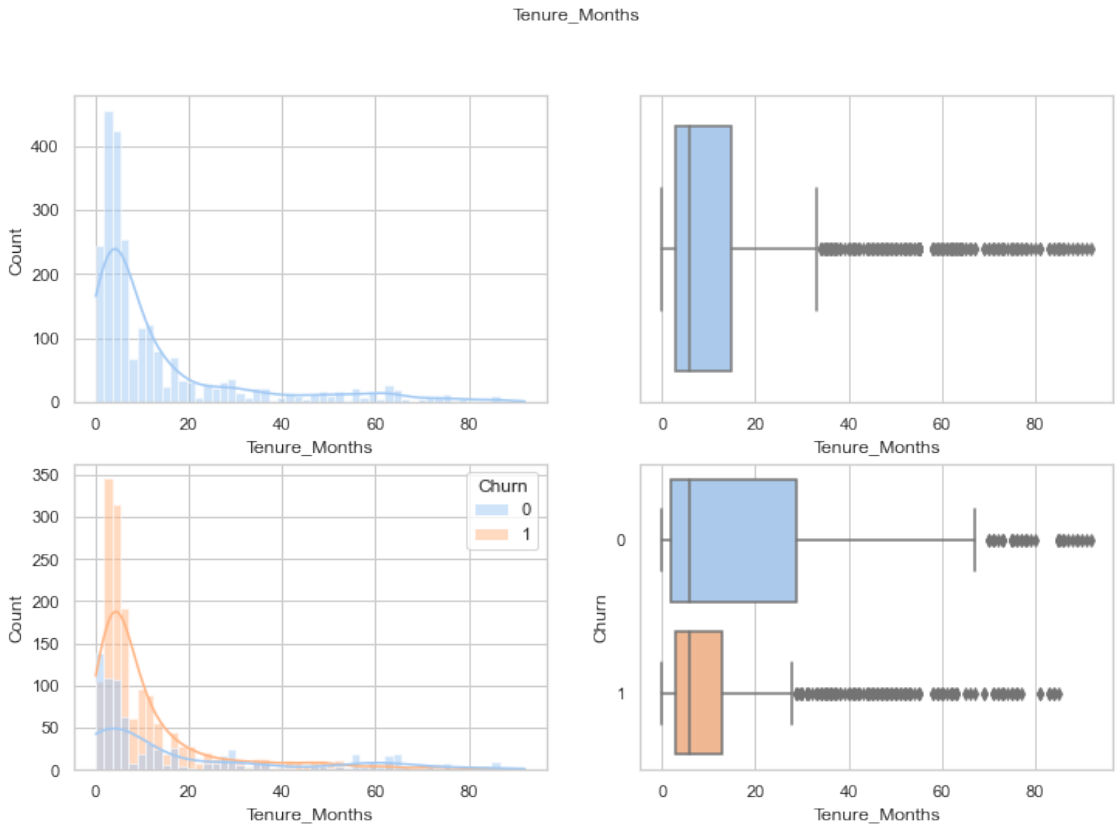


Observations:

1. Income has a positively skewed distribution with several outliers on the right tail of the distribution. However, unlike monthly business value, we do not see any noisy data.
2. The overall (mean, median) values for income is (59209, 55276). For drivers who churned, the (mean, median) values drop to (55377, 51630). For drivers who didn't churn, mean and median values increase to (67303, 63632)
3. Drivers with income <25K and <50K have high churn rates (83% and 74.8% respectively). For drivers with income between <50K and <75K, the churn rate is average. For drivers having income > 75K, churn rate reduces and income increases. **Thus drivers with lower income tend to churn more.**

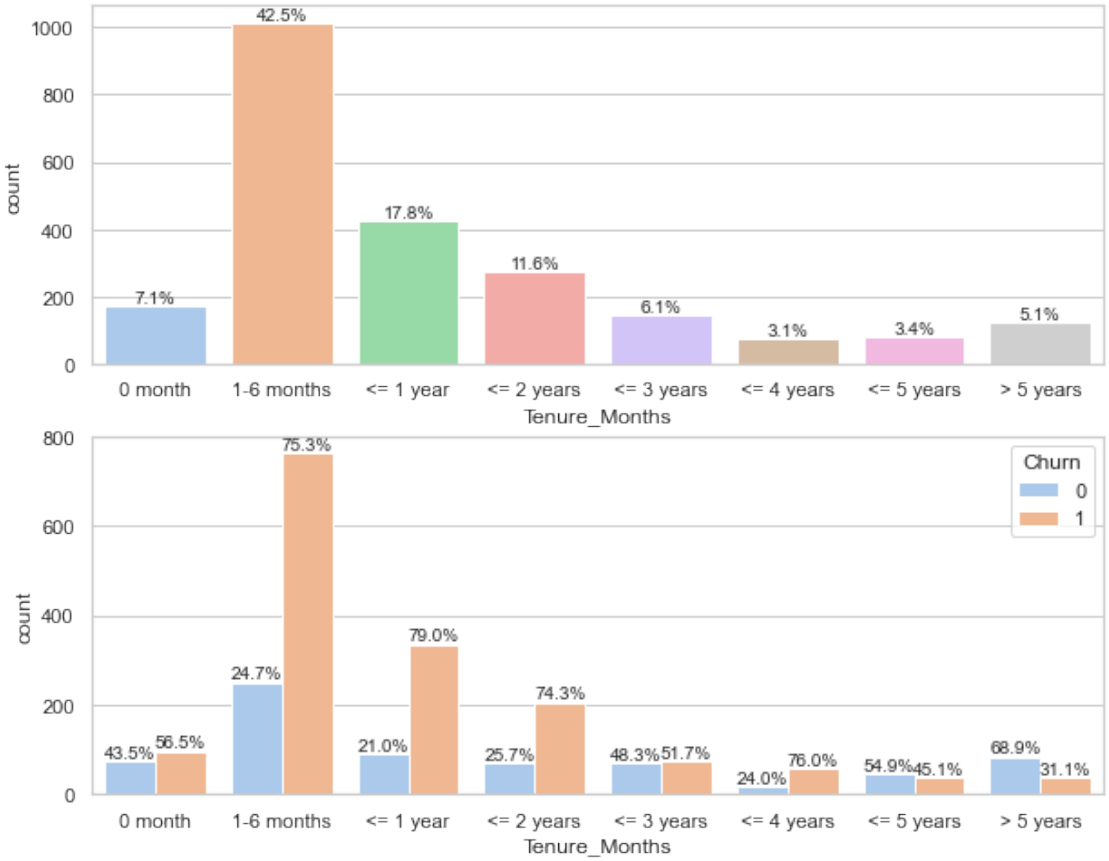
Tenure_months

[31]:
```
col = 'Tenure_Months'
show_num_plots(df[col], y , title=col)
print(f'{col}: overall mean - {df[col].mean()}, overall median - {df[col].
↪median()}')
print('\n',np.round(df.groupby('Churn')[col].describe(), 1))
```




```
Tenure_Months: overall mean - 13.979840403191936, overall median - 6.0

         count  mean   std  min  25%  50%   75%   max
Churn
0        765.0  18.7  23.7  0.0  2.0  6.0  29.0  92.0
1       1616.0  11.8  15.0  0.0  3.0  6.0  13.0  85.0
```

[167]:
```
tm_binned = pd.cut(df['Tenure_Months'], bins=[0, 1, 6, 12, 24, 36, 48, 60,␣
 ↪100], labels=['0 month', '1-6 months', '<= 1 year', '<= 2 years', '<= 3␣
 ↪years', '<= 4 years', '<= 5 years', '> 5 years'])
fig, ax = plt.subplots(2, 1, figsize=(10, 8))
show_cat_plots(tm_binned, y, axs=ax)
```

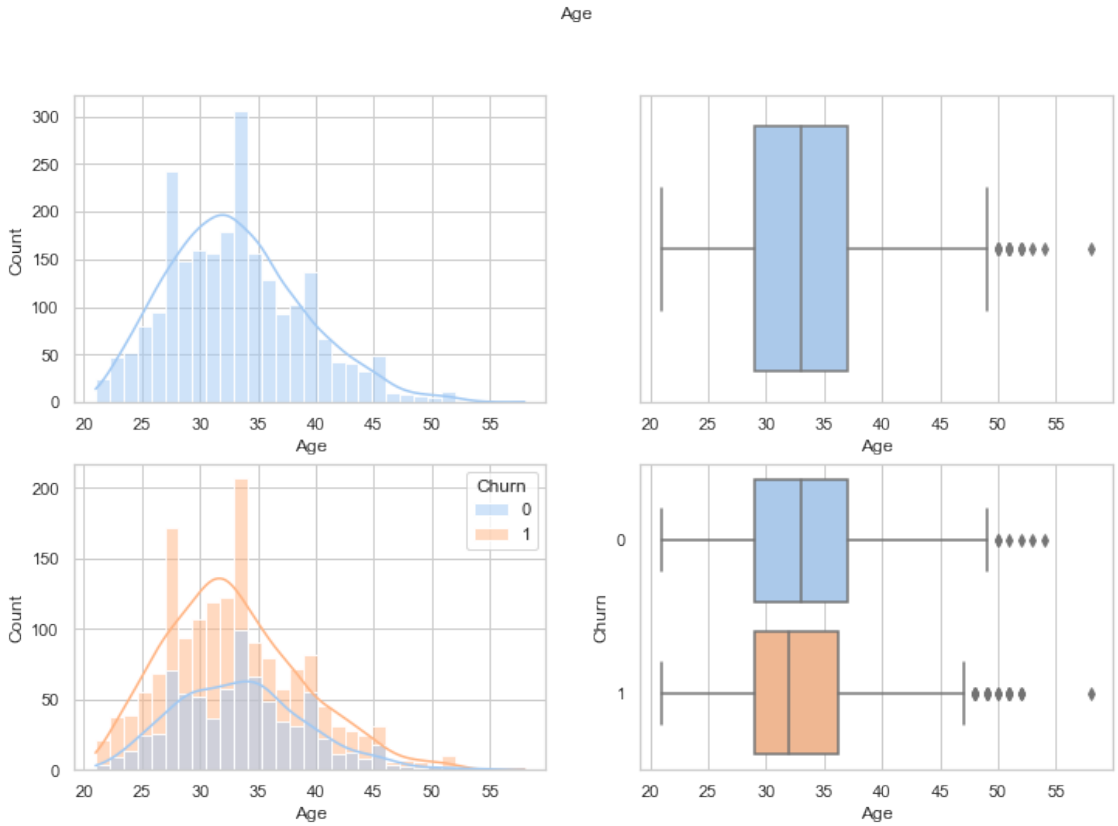


Observations:

1. Tenure_Months is a positively skewed distribution with several outliers on right tail.
2. The overall average tenure length is 14 months. Drivers who churned have the average tenure length of 11.7 months. Whereas those who did not churn have the average tenure lengthof 18.6 months. However, when we compare the median tenure length, they are identical for drivers who churned and those who did not (6 months).
3. Churn rates are especially higher for drivers with tenure less than 2 year and between 3 to 4 years. Churn rates are reduces after 4 years and lowest once a driver crosses 5 years tenure.

Age

[32]:
```
col = 'Age'
show_num_plots(df[col], y , title=col)
print(f'{col}: overall mean - {df[col].mean()}, overall median - {df[col].
 ↪median()}')
print('\n',np.round(df.groupby('Churn')[col].describe(), 1))
```




```
Age: overall mean - 33.090718185636284, overall median - 33.0
```

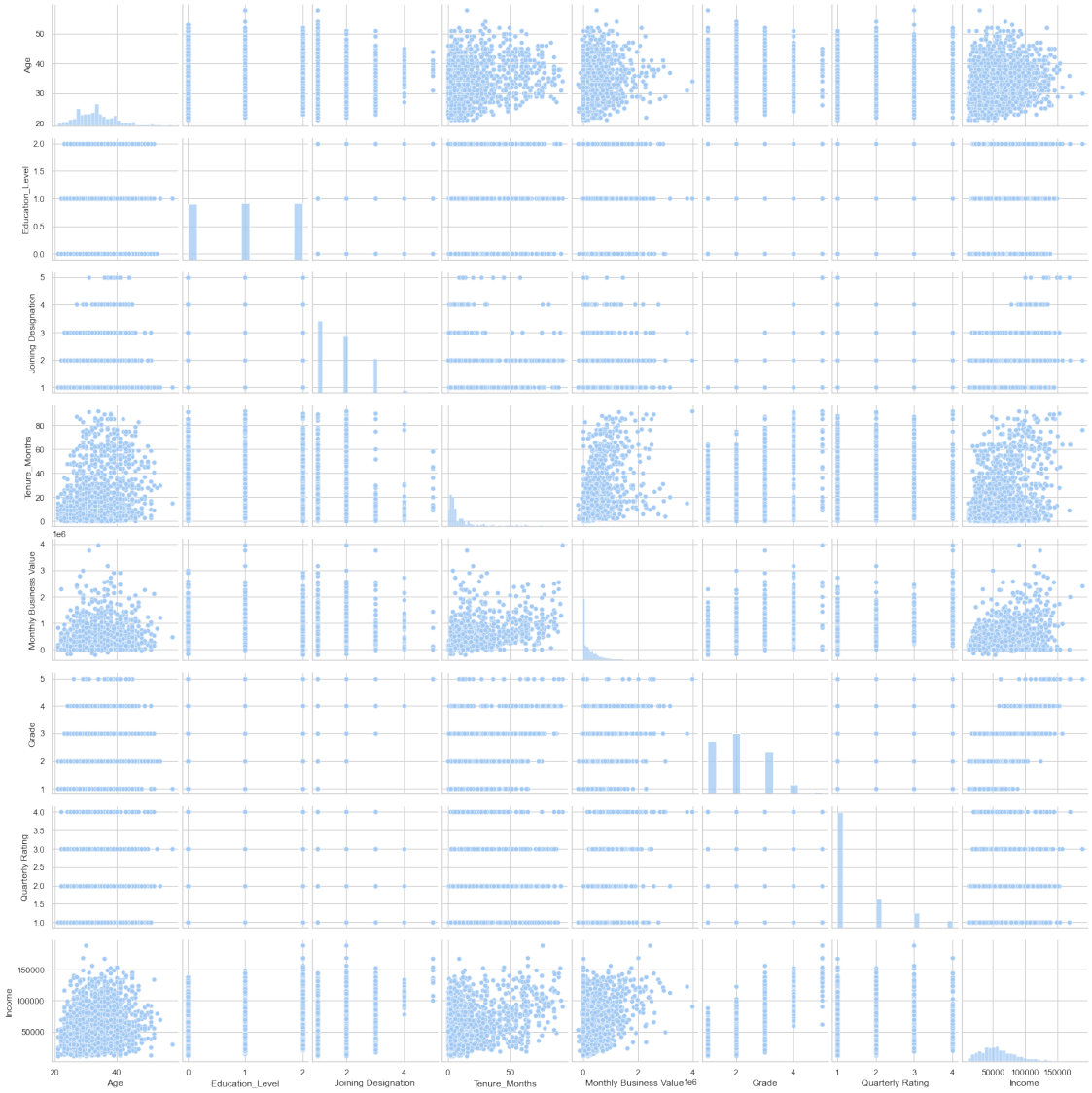
Churn	count	mean	std	min	25%	50%	75%	max
0	765.0	33.5	5.6	21.0	29.0	33.0	37.0	54.0
1	1616.0	32.9	5.9	21.0	29.0	32.0	36.2	58.0

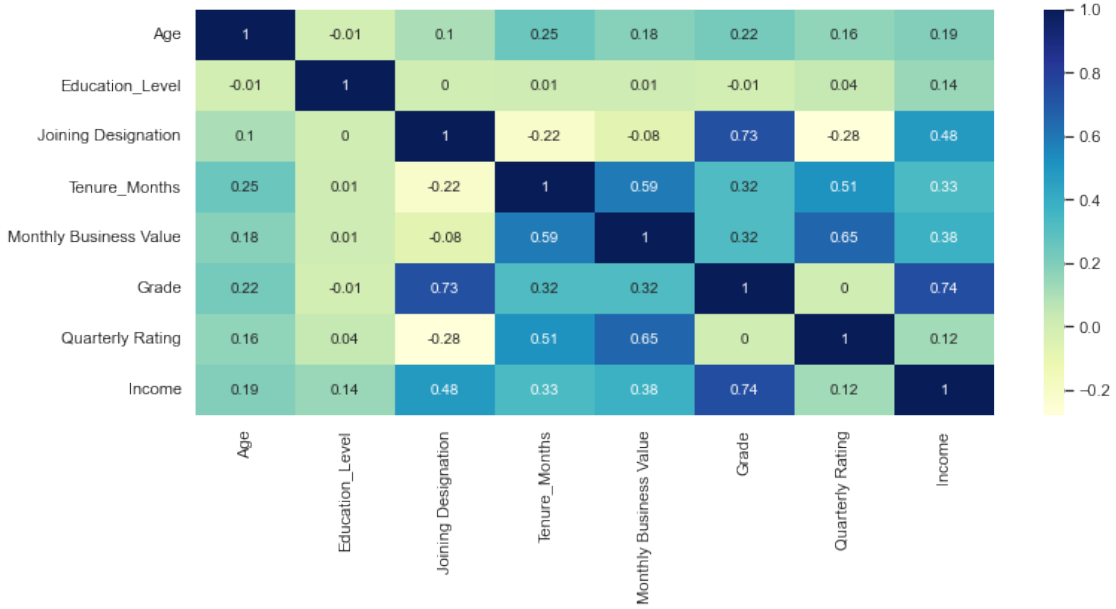
Observations:

Overall mean age value for drivers is 33. Similarly, mean age values for drivers who churned and did not churn are also around 33. **Age is not an important factor in predicting churn.**

2.4.5 Correlation and pairplots

[33]:
```
cols = df.columns.drop(['Gender', 'Rating_Inc', 'Income_Inc', 'Churn'])
sns.pairplot(df[cols])

corrmat = np.round(df[cols].corr(), 2)
plt.figure(figsize = (12,5))
sns.heatmap(corrmat, cmap="YlGnBu", annot=True)
plt.show()
```


Age
Education_Level
Joining Designation
Tenure_Months
Monthly Business Value
Grade
Quarterly Rating
Income
1e6



Observations:

1. We see high positive correlation between {Joining Designation, Grade} variables (0.73), {Grad, income} (0.74), and {Monthly business value, Quarterly rating} (0.65)
2. We also see moderately high correlation between {Tenure months, monthly business value}(0.59), {tenure months, quarterly rating} (0.51), and {joining designation, income} (0.48)
3. We also see weak positive correlation between {Monthly business value, grade} (0.32), {monthly business value, income} (0.38), {tenure_months, income} (033), {tenure_months, grade} (0.32).
4. We see weak negative correlation between joining designation and quarterly rating (-0.28)

2.5 Additional data pre-processing

1. As observed before, our aggregated data-set does not have any missing values, so we will not need missing value imputations.
2. Tree based algorithms are usually robust to outliers, however, they may impact decision trees when the depth is higher. WE will address this further in outliers detection and treatment section once we split data into training and test sets.
3. Tree based algorithms can handle non-scaled data well, therefore, we will not standardize/normalize our data.
4. Tree based algorithms are not affected by multicollinearity among independent features, therefore, we will not address it further.
5. We observe that target variable churn is somewhat imbalanced (68% 1 and 32% 0). We have two alternatives. Using oversampling techniques such as SMOTE to generate additional non-

churn samples, or using class weights while building tree based models. We will prefer the option of class weights in this case_study.

2.5.1 categorical variable encoding

In general, tree based algorithms are capable of handling categorical features. However, sklearn implementation does not support categorical variables at the moment. So we will encode the categorical variables before creating models.

1. For dichotomous/ordinal variables such as Gender, Grade, Quarterly_rating, Education_Level, Rating_Inc, Income_Inc, Joining Designation, we will simply convert them to numeric values.
2. For City, which is a nominam variable, we will use one-hot encoding.

[34]:
```
#convert dichotomous/ordinal variables to numeric
df['Gender'] = df['Gender'].astype('int')
df['Rating_Inc'] = df['Rating_Inc'].astype('int')
df['Income_Inc'] = df['Income_Inc'].astype('int')
df['Education_Level'] = df['Education_Level'].astype('int')
df['Joining Designation'] = df['Joining Designation'].astype('int')
df['Grade'] = df['Grade'].astype('int')
df['Quarterly Rating'] = df['Quarterly Rating'].astype('int')
df['Churn'] = df['Churn'].astype('int')
```

[35]:
```
#onehot encoding for city
from sklearn.preprocessing import OneHotEncoder

enc = OneHotEncoder().fit(df[['City']])
cols = [enc.categories_]
df_city_enc = pd.DataFrame(enc.transform(df[['City']]).toarray(),␣
 ↪columns=['City_' + cat for cat in enc.categories_[0]])
df = df.join(df_city_enc)
df.drop('City', axis=1, inplace=True)
```

2.6 Train-test split

[80]:
```
from sklearn.model_selection import train_test_split

dfX = df[df.columns.drop('Churn')]
dfy = df['Churn']

X_train, X_test, y_train, y_test = train_test_split(dfX, dfy, train_size=0.8␣
 ↪,random_state=100)

X_train = X_train.reset_index(drop=True)
X_test = X_test.reset_index(drop=True)
y_train = y_train.reset_index(drop=True)
```

```
y_test = y_test.reset_index(drop=True)

df_train = pd.merge(X_train, y_train, left_index=True, right_index=True)

#create backup
X_train_bk = X_train.copy(deep=True)
X_test_bk = X_test.copy(deep=True)
y_train_bk = y_train.copy(deep=True)
y_test_bk = y_test.copy(deep=True)
df_train_bk = df_train.copy(deep=True)
```

[81]:
```
X_train.shape
```

[81]: (1904, 40)

[82]:
```
X_test.shape
```

[82]: (477, 40)

[83]:
```
#restore from backup
X_train = X_train_bk.copy(deep=True)
X_test = X_test_bk.copy(deep=True)
y_train = y_train_bk.copy(deep=True)
y_test = y_test_bk.copy(deep=True)
```

2.6.1 [Optional] Addressing rows with zero monthly business value

If we know that records with zero monthly business values are noisy records, we can treat them by updating their value by looking at the neighboring records. To prevent data leakage, We remediate this after splitting data into training and test datasets. Post split, We can use kneighborregressor to fit and predict monthly business values wherever it's set to zero (in training data) before building and validating models. Finally, We can predict monthly business values for test data before evaluating test score. Please see next section for more details.

treat_zero_monthly_business_value flag controls this code and is set to False.

[84]:
```
from sklearn.neighbors import KNeighborsRegressor

#treat zero monthly business values
def treat_monthly_business_values(df):
    zero_mbv_mask = (df['Monthly Business Value'] == 0)
    if(np.any(df[zero_mbv_mask])):
        mbv_imputed = neigh.predict(df[zero_mbv_mask][df.columns.drop('Monthly␣
↪Business Value')])
        df.loc[zero_mbv_mask,'Monthly Business Value'] = mbv_imputed


treat_zero_monthly_business_value = False
```

```
if(treat_zero_monthly_business_value):

    #select records with non zero monthly business value
    df_mbv = X_train[X_train['Monthly Business Value'] != 0]
    X_mbv = df_mbv[df_mbv.columns.drop('Monthly Business Value')] #all other␣
 ↪indepedent  variables
    y_mbv = df_mbv['Monthly Business Value'] #monthly business value

    #create KNeighborsRegressor model and fit on training data
    neigh = KNeighborsRegressor(n_neighbors=20)
    neigh.fit(X_mbv, y_mbv)

    treat_monthly_business_values(X_train)

    sns.histplot(X_train['Monthly Business Value'])
    plt.show()
```

2.7 Outliers detection and treatment

Tree based algorithms are generally robust to outliers. However, they can impact a decision tree when the depth is high (may potentially cause overfitting). In this section, we detect outliers and remove them if needed before building our models.

[87]:
```
#outlier detection using local outlier factor
from sklearn.neighbors import LocalOutlierFactor
from collections import Counter

lof_clf = LocalOutlierFactor(n_neighbors=10)
res = lof_clf.fit_predict(X_train)
res = [(lof == 1) for lof in res]

print('Inlier counts:')
Counter(res)
```

```
Inlier counts:
```

[87]:
```
Counter({True: 1869, False: 35})
```

[88]:
```
#remove outliers
X_train = X_train[res]
y_train = y_train[res]

print(X_train.shape, y_train.shape)
```

```
(1869, 40) (1869,)
```

2.8 Model building and evaluation

2.8.1 [optional] prepare test dataset

[89]:
```
#update monthly business values if they are set to zero (using␣
 ↪knearibor regression on training dataset)
if(treat_zero_monthly_business_value):
    treat_monthly_business_values(X_test)
    sns.histplot(X_test['Monthly Business Value'])
```

2.8.2 common utility functions

[44]:
```
from sklearn.model_selection import KFold, cross_validate, StratifiedKFold
from sklearn import metrics
from sklearn.metrics import classification_report
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier
from sklearn.experimental import enable_halving_search_cv
from sklearn.model_selection import RandomizedSearchCV, GridSearchCV,␣
 ↪HalvingGridSearchCV
from xgboost import XGBClassifier
from datetime import datetime
import datetime as dt

#function to plot feature importance graph
def plot_feat_imp(clf, df, figsize=(15,5)):
    importances = clf.feature_importances_
    indices = np.argsort(importances)[::-1]
    names = [df.columns[i] for i in indices] # rearranged columns

    plt.figure(figsize=figsize)
    plt.title("Feature Importance")
    plt.bar(range(df.shape[1]), importances[indices])
    plt.xticks(range(df.shape[1]), names, rotation=90)
    plt.show()

#function to plot ROC curve
def draw_roc( actual, probs ):
    fpr, tpr, thresholds = metrics.roc_curve( actual, probs,
                                              drop_intermediate = False )
    auc_score = metrics.roc_auc_score( actual, probs )
    plt.figure(figsize=(6, 4))
    plt.plot( fpr, tpr, label='ROC curve (area = %0.2f)' % auc_score )
    plt.plot([0, 1], [0, 1], 'k--')
    plt.xlim([0.0, 1.0])
    plt.ylim([0.0, 1.05])
    plt.xlabel('False Positive Rate or [1 - True Negative Rate]')
```

```
    plt.ylabel('True Positive Rate')
    plt.title('Receiver operating characteristic example')
    plt.legend(loc="lower right")
    plt.show()

    return fpr, tpr, thresholds

#helper function to hypertune parameters and retrieve results in dataframe
def hyper_tune(tuning_function, X_train, y_train):
    ts = datetime.now()
    tuning_function.fit(X_train, y_train)
    te = datetime.now()

    res = pd.DataFrame(tuning_function.cv_results_['params']).assign(
    validation_score = tuning_function.cv_results_['mean_test_score'])

    if('mean_train_score' in tuning_function.cv_results_):
        res = res.assign(train_score = tuning_function.
 ↪cv_results_['mean_train_score'])

    res = res.sort_values(by='validation_score', ascending=False)

    return (tuning_function, res, (te - ts))
```

2.9 1. Decision Tree classifier (For benchmarking)

[110]:
```
# Build unconstrained (potentially overfitted tree)
tree_clf = DecisionTreeClassifier(random_state=7, class_weight = None)
tree_clf.fit(X_train, y_train)

print('training score: ', tree_clf.score(X_train, y_train))
print('test score:', tree_clf.score(X_test, y_test))
```

```
training score:  1.0
test score: 0.7547169811320755
```

[91]:
```
#hyper parameter tuning

params = {
    'max_depth': [2,3,4,5,6,7,8],
    'class_weight':['balanced', None]
}

dt_tuning_function = GridSearchCV(estimator =␣
 ↪DecisionTreeClassifier(random_state=7),
                                  param_grid = params,
                                  scoring = 'accuracy',
```

```
                                cv = 5,
                                n_jobs=-1,
                                return_train_score=True
                                )

dt_tuning_function, dt_tuning_res, dt_time_taken = ␣
 ↪hyper_tune(dt_tuning_function, X_train, y_train)
print(f'time taken for hyper parameter tuning: {dt_time_taken}')
dt_tuning_res.head(20)
```

```
time taken for hyper parameter tuning: 0:00:00.667214
```

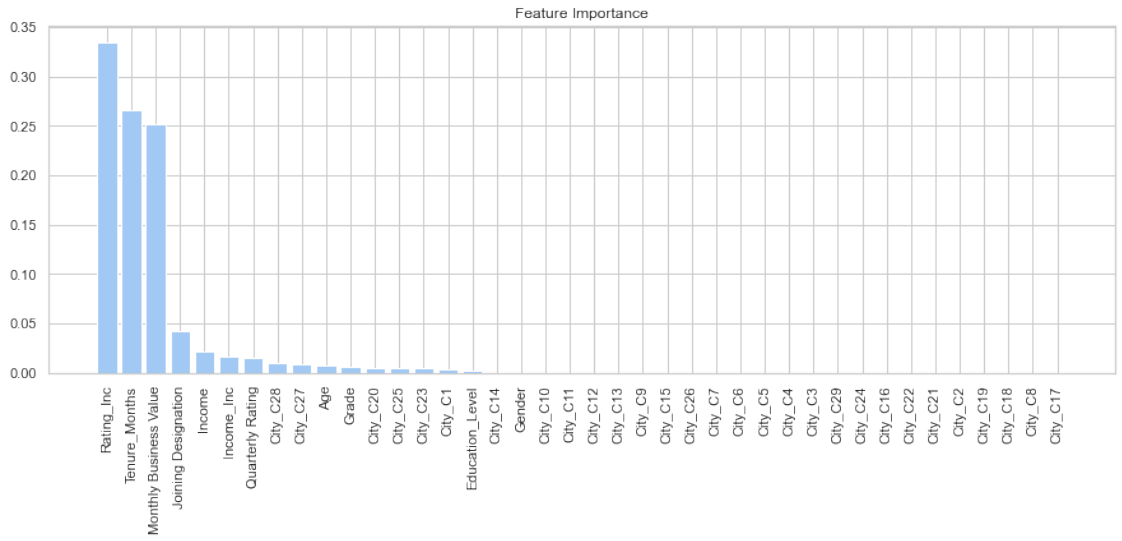
[91]:

	class_weight	max_depth	validation_score	train_score
10	None	5	0.790275	0.834003
9	None	4	0.787597	0.816748
11	None	6	0.786518	0.853265
8	None	3	0.784945	0.800564
13	None	8	0.780643	0.888978
12	None	7	0.777428	0.869316
1	balanced	3	0.770488	0.778090
4	balanced	6	0.765664	0.826779
3	balanced	5	0.763503	0.806448
5	balanced	7	0.759765	0.850056
6	balanced	8	0.759222	0.870788
2	balanced	4	0.756541	0.778096
0	balanced	2	0.754950	0.776486
7	None	2	0.754950	0.776753

[117]:
```
#build decision tree classifier and measure score on test data
tree_clf = DecisionTreeClassifier(random_state=7, max_depth = 5, class_weight = ␣
 ↪None)
tree_clf.fit(X_train, y_train)

print('training score: ', tree_clf.score(X_train, y_train))
print('test score:', tree_clf.score(X_test, y_test))
```

```
training score:  0.8293204922418406
test score: 0.7924528301886793
```

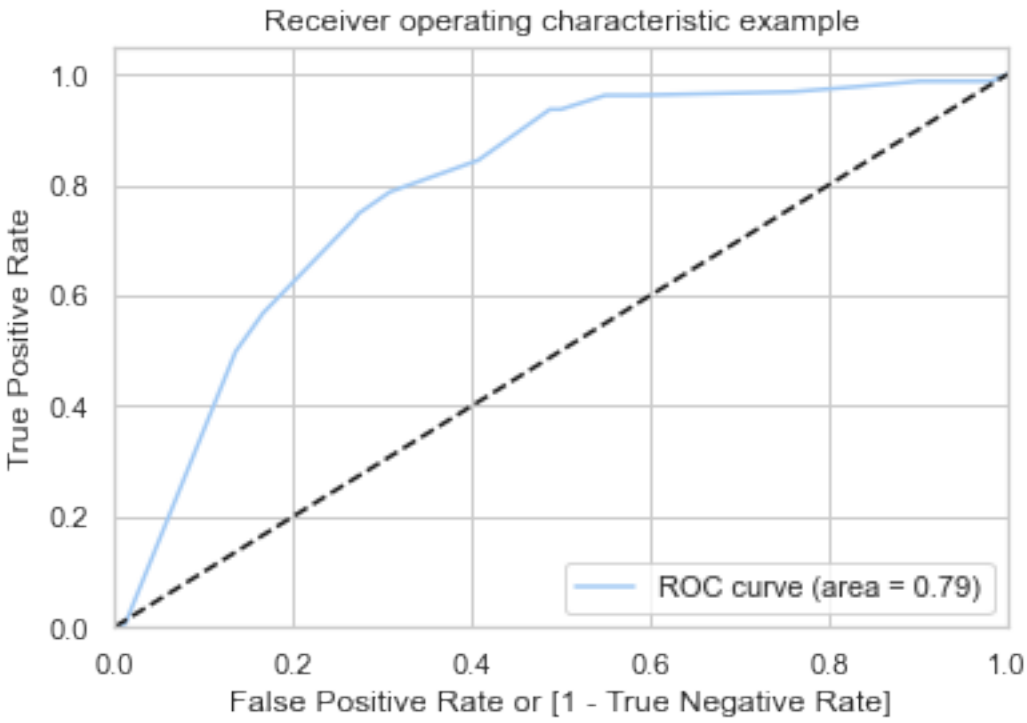
[119]:
```
#plot feature importance
plot_feat_imp(tree_clf, X_train)
```



[120]:
```
#classification reports
y_pred = tree_clf.predict(X_test)
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.81      0.51      0.63       162
           1       0.79      0.94      0.86       315

    accuracy                           0.79       477
   macro avg       0.80      0.72      0.74       477
weighted avg       0.79      0.79      0.78       477
```

[121]:
```
#plot ROC curve
y_pred_prob = tree_clf.predict_proba(X_test)
fpr, tpr, thresholds = draw_roc(y_test, y_pred_prob[:,1])
```



Observations:

1. The unconstrained decision tree overfitted and produced validation score of 75.47%. After hypertuning max_depth parameter, we got the best validation score for 79% for max_depth=5. The final model produced test score of 79.24%
2. feature importance: Top 3 features (in the order of importance) are: Rating_inc, tenure_months, Monthly business value.
3. precision score: 0.79, recall: 0.94, f1-score: 0.86, ROC area = 0.79.
4. We will use this result as benchmark for comparing performance of bagging and boosting models discussed in the next sections.

2.10 2. Bagging (Random Forest classifier)

[109]:
```
#hyper parameter tuning 1
params = [
    {
        'n_estimators' : [200, 400, 500],
        'max_depth' : [5,8,10,12],
        'bootstrap' : [True],
        'max_features' : [None,8,9,10,15], #col sampling
        'max_samples' : [0.2, 0.4, 0.6, 0.8], #row sampling
        'class_weight': ['balanced', None]
```

```
    }
]

rf_tuning_fn = GridSearchCV(
    estimator = RandomForestClassifier(),
    param_grid = params,
    scoring = 'accuracy',
    cv = 3,
    n_jobs=-1,
    return_train_score=True
)

rf_tuning_fn, rf_tuning_res, rf_time_taken = hyper_tune(rf_tuning_fn, X_train,␣
 ↪y_train)
print(f'time taken for hyper parameter tuning: {rf_time_taken}')
```

```
time taken for hyper parameter tuning: 0:10:38.742645
```

[111]:

```
rf_tuning_res.head(20)
```

[111]:

	bootstrap	class_weight	max_depth	max_features	max_samples
463	True	None	12	10.0	0.6
473	True	None	12	15.0	0.4
412	True	None	10	15.0	0.4
419	True	None	10	15.0	0.8
415	True	None	10	15.0	0.6
467	True	None	12	10.0	0.8
472	True	None	12	15.0	0.4
454	True	None	12	9.0	0.8
475	True	None	12	15.0	0.6
390	True	None	10	9.0	0.6
246	True	None	5	NaN	0.6
393	True	None	10	9.0	0.8
417	True	None	10	15.0	0.8
459	True	None	12	10.0	0.4
407	True	None	10	10.0	0.8
464	True	None	12	10.0	0.6
461	True	None	12	10.0	0.4
416	True	None	10	15.0	0.6
443	True	None	12	8.0	0.8
359	True	None	8	15.0	0.8

	n_estimators	validation_score	train_score
463	400	0.807919	0.930444
473	500	0.806314	0.910915
412	400	0.805778	0.895934
419	500	0.805778	0.931782

415	400	0.805243	0.917068
467	500	0.804708	0.944088
472	400	0.804708	0.909310
454	400	0.804708	0.940342
475	400	0.804173	0.941680
390	200	0.804173	0.901819
246	200	0.804173	0.846709
393	200	0.804173	0.909845
417	200	0.804173	0.932852
459	200	0.804173	0.907170
407	500	0.803638	0.914928
464	500	0.803638	0.932049
461	500	0.803638	0.903157
416	500	0.803638	0.914393
443	500	0.803103	0.931782
359	500	0.803103	0.892723

Observations:

We hyper-tuned n_estimators, max_depth, bootstrap, max_features, max_samples, and class_weight parameters using gridsearchcv. The best validation score was 80.79% for parameters {bootstrap:True, class_weight:None, max_depth:12, ,max_features: 10, max_samples: 0.6, n_estimators:400}. For this parameter, training score is 93%. We can see a rather large difference between the training and test score indicating potential overfitting. We will try with a second set of parameters, as shown below.

[107]:
```
# Try with smaller max_samples, smaller max_depth, but larger number of␣
 ↪n_estimators. Also with both balanced and balanced_subsample.
params = [
    {
        'n_estimators' : [400, 500, 600],
        'max_depth' : [5,8,10],
        'bootstrap' : [True],
        'max_features' : [None,8,9,10,15], #col sampling
        'max_samples' : [0.4, 0.6], #row sampling
        'class_weight': ['balanced', 'balanced_subsample'],
        'min_samples_split': [4,6,8]
    }
]

rf_tuning_fn2 = GridSearchCV(
    estimator = RandomForestClassifier(),
    param_grid = params,
    scoring = 'accuracy',
    cv = 3,
    n_jobs=-1,
    return_train_score=True
)
```

```
rf_tuning_fn2, rf_tuning_res2, rf_time_taken2 = hyper_tune(rf_tuning_fn2,␣
↪X_train, y_train)
```

[108]:
```
print('Hypertuning time: ', rf_time_taken2)
rf_tuning_res2.head(20)
```

```
Hypertuning time:  0:16:30.024902
```

[108]:

	bootstrap	class_weight	max_depth	max_features	max_samples	min_samples_split	n_estimators	validation_score	train_score
487	True	balanced_subsample	10	9.0	0.4	4	500	0.800963	0.889513
468	True	balanced_subsample	10	8.0	0.4	4	400	0.797753	0.891118
522	True	balanced_subsample	10	15.0	0.4	4	400	0.797218	0.899679
504	True	balanced_subsample	10	10.0	0.4	4	400	0.796683	0.892723
237	True	balanced	10	10.0	0.4	6	400	0.796683	0.883093
218	True	balanced	10	9.0	0.4	4	600	0.796683	0.894596
495	True	balanced_subsample	10	9.0	0.6	4	400	0.796683	0.914125
478	True	balanced_subsample	10	8.0	0.6	4	500	0.796148	0.909042
203	True	balanced	10	8.0	0.4	6	600	0.796148	0.881487
488	True	balanced_subsample	10	9.0	0.4	4	600	0.796148	0.892723
471	True	balanced_subsample	10	8.0	0.4	6	400	0.796148	0.882825
475	True	balanced_subsample	10	8.0	0.4	8	500	0.796148	0.871589
486	True	balanced_subsample	10	9.0	0.4	4	400	0.795613	0.894864
206	True	balanced	10	8.0	0.4	8	600	0.795613	0.876672
489	True	balanced_subsample	10	9.0	0.4	6	400	0.795613	0.882825
224	True	balanced	10	9.0	0.4	4	500	0.795613	0.889513
494	True	balanced_subsample	10	9.0	0.4	4	500	0.795613	0.889513
507	True	balanced_subsample	10	10.0	0.4	4	500	0.795613	0.889513
224	True	balanced	10	9.0	0.4	4	500	0.795613	0.889513

472	6	500	0.795613	0.878277
238	6	500	0.795613	0.888175
494	8	600	0.795613	0.877207
507	6	400	0.795078	0.885233
224	8	600	0.795078	0.877742

Observations:

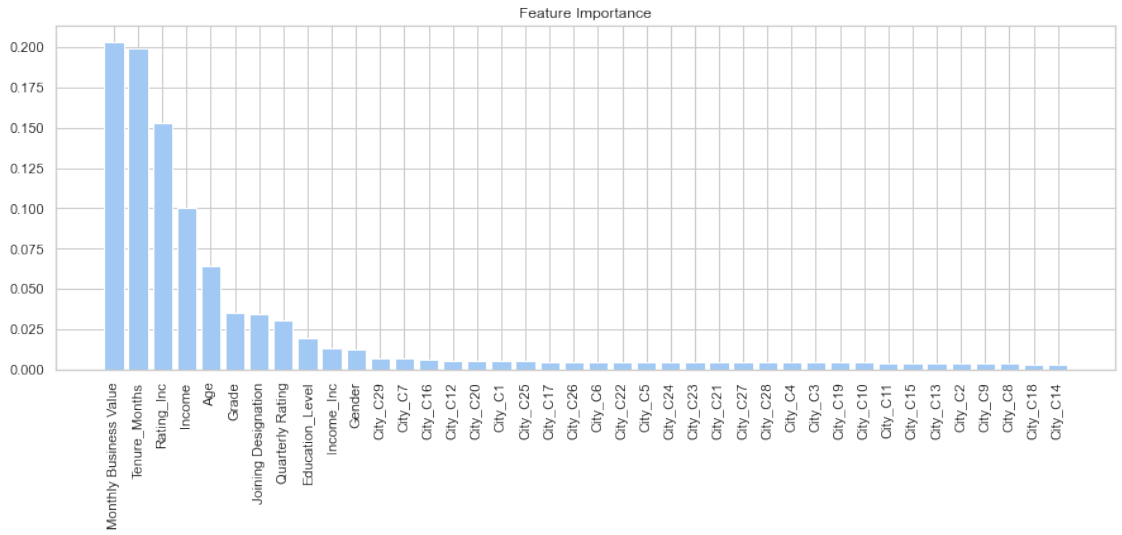
After further hypertuning, the best validation score we could get is 80.09% which is lower than the validation score we obtained with the previous hypertuning attempt. We will therefore build the final bagging model with the parameters obtained from the first hypertuning try.

parameters: **{bootstrap:True, class_weight:None, max_depth:10, ,max_features: 12, max_samples: 0.6, n_estimators:400}**

[112]:
```
#Build final randomforest model with best parameters and measure score on test␣
↪data

rf_clf = RandomForestClassifier(random_state=7, bootstrap=True, max_depth = 10,␣
↪max_features=12, max_samples=0.6, n_estimators=400, class_weight = None)
rf_clf.fit(X_train, y_train)

print('training score: ', rf_clf.score(X_train, y_train))
print('test score:', rf_clf.score(X_test, y_test))
```

```
training score:  0.8908507223113965
test score: 0.7945492662473794
```

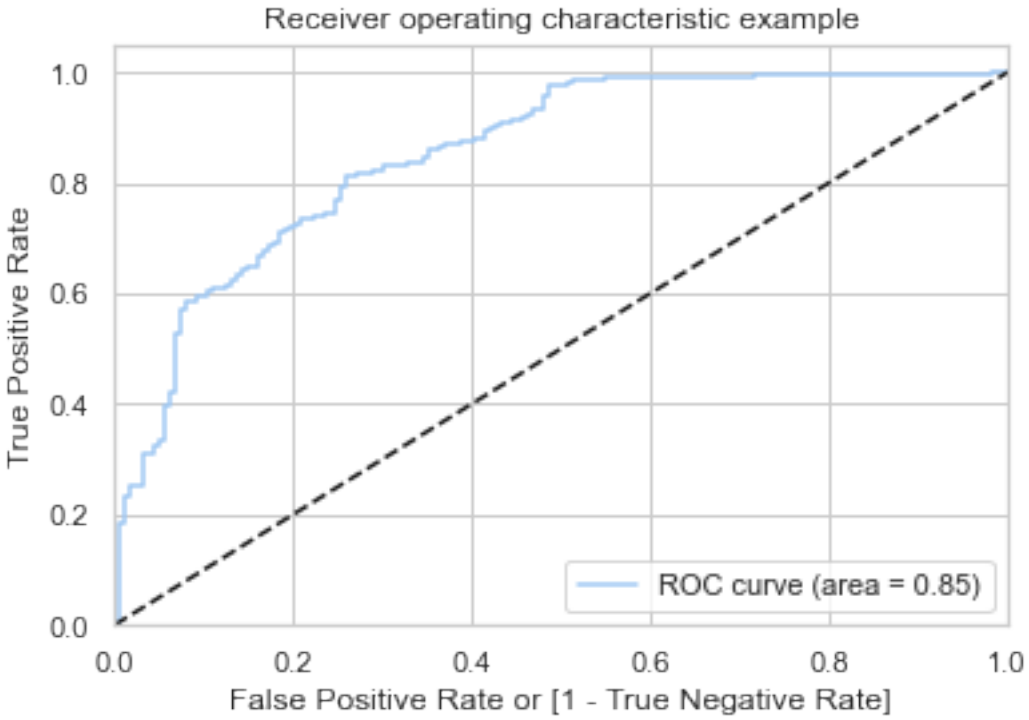
[113]:
```
#plot feature importance
plot_feat_imp(rf_clf, X_train)
```



[114]:
```
y_pred = rf_clf.predict(X_test)
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.80      0.53      0.64       162
           1       0.79      0.93      0.86       315

    accuracy                           0.79       477
   macro avg       0.80      0.73      0.75       477
weighted avg       0.79      0.79      0.78       477
```

[115]:
```
#plot ROC curve
y_pred_prob = rf_clf.predict_proba(X_test)
fpr, tpr, thresholds = draw_roc(y_test, y_pred_prob[:,1])
```



Observations:

1. We built the final bagging model with this parameter set: {bootstrap:True, class_weight:None, max_depth:10, ,max_features: 12, max_samples: 0.6, n_estimators:400}. This model produced test score of 79.45% which is similar score compared to decision tree classifier.
2. feature importance: Top 3 features (in the order of importance) are: Monthly business value,

tenure_months, Rating_inc.

3. precision score: 0.79, recall: 0.93, f1-score: 0.86 (same as decision tree classifier)
4. ROC area: 0.85 (improvement over decision tree classifier)

2.11 3. Boosting (XGBoost)

In this section, we will use xgboost library to build gradient boosting model. As before, we will try to find optimal parameters through hyper parameter tuning.

[94]:
```
#hypertuning 1
params = [
    {
        'n_estimators' : [50, 100, 150, 200],
        'learning_rate': [0.1, 0.5, 0.8],
        'subsample': [0.4, 0.6, 0.8, 1.0],
        'colsample_bytree': [0.6, 0.8, 1.0],
        'max_depth': [3, 4, 5]
    }
]

xgb = XGBClassifier()
skf = StratifiedKFold(n_splits=3, shuffle = True, random_state = 100)

xg_tuning_fn = GridSearchCV(
    estimator = xgb,
    param_grid = params,
    scoring = 'accuracy',
    cv = skf.split(X_train,y_train),
    n_jobs=-1,
    return_train_score=True
)

xg_tuning_fn, xg_res, xg_time_taken = hyper_tune(xg_tuning_fn, X_train, y_train)
print('Hyperparameter tuning time: ', xg_time_taken)
```

```
Hyperparameter tuning time:  0:02:17.687095
```

[95]:
```
xg_res.head(20)
```

[95]:

	colsample_bytree	learning_rate	max_depth	n_estimators	subsample	\
7	0.6	0.1	3	100	1.0	
183	0.8	0.1	5	100	1.0	
322	1.0	0.1	5	50	0.8	
294	1.0	0.1	3	100	0.8	
298	1.0	0.1	3	150	0.8	
11	0.6	0.1	3	150	1.0	
292	1.0	0.1	3	100	0.4	

167	0.8	0.1	4	100	1.0
155	0.8	0.1	3	150	1.0
10	0.6	0.1	3	150	0.8
147	0.8	0.1	3	50	1.0
6	0.6	0.1	3	100	0.8
15	0.6	0.1	3	200	1.0
161	0.8	0.1	4	50	0.6
154	0.8	0.1	3	150	0.8
305	1.0	0.1	4	50	0.6
310	1.0	0.1	4	100	0.8
21	0.6	0.1	4	100	0.6
3	0.6	0.1	3	50	1.0
307	1.0	0.1	4	50	1.0

	validation_score	train_score
7	0.820760	0.865169
183	0.816479	0.920546
322	0.816479	0.897271
294	0.815944	0.872659
298	0.815944	0.887908
11	0.815944	0.876940
292	0.815409	0.868914
167	0.815409	0.892991
155	0.815409	0.882825
10	0.815409	0.882825
147	0.814874	0.840021
6	0.814874	0.866506
15	0.814874	0.891386
161	0.814339	0.866506
154	0.814339	0.887105
305	0.814339	0.866239
310	0.814339	0.902889
21	0.814339	0.889781
3	0.814339	0.844837
307	0.813804	0.865436

Observations:

We hyper-tuned n_estimators, max_depth, learning_rate, colsample_bytree, subsample parameters using gridsearchcv. The best validation score is 82.04% for parameters {colsample_bytree:0.6, subsample:1.0, learning_rate:0.1, max_depth:3, n_estimators:100}. For this parameter, training score is 86.5%. We will try further finetune learning_rate, sab_sample, and colsample_bytree.

[96]:
```
#hypertuning 2: finer learning rate, sab_Sample, and colsample_bytree
params = [
    {
        'n_estimators' : [50, 100, 150, 200],
        'learning_rate': [0.05, 0.1, 0.15],
```

```
        'subsample': [0.8, 0.9, 1.0],
        'colsample_bytree': [0.8, 0.9, 1.0],
        'max_depth': [3, 4]
    }
]

xgb = XGBClassifier()
skf = StratifiedKFold(n_splits=3, shuffle = True, random_state = 100)

xg_tuning_fn2 = GridSearchCV(
    estimator = xgb,
    param_grid = params,
    scoring = 'accuracy',
    cv = skf.split(X_train,y_train),
    n_jobs=-1,
    return_train_score=True
)

xg_tuning_fn2, xg_res2, xg_time_taken2 = hyper_tune(xg_tuning_fn2, X_train,␣
 ↪y_train)
print('Hyperparameter tuning time: ', xg_time_taken2)
```

```
Hyperparameter tuning time:  0:01:08.723184
```

[97]:
```
xg_res2.head(20)
```

[97]:

	colsample_bytree	learning_rate	max_depth	n_estimators	subsample	\
193	1.0	0.15	3	50	0.9	
99	0.9	0.10	3	100	0.8	
62	0.8	0.15	4	50	1.0	
181	1.0	0.10	4	50	0.9	
169	1.0	0.10	3	50	0.9	
154	1.0	0.05	3	200	0.9	
194	1.0	0.15	3	50	1.0	
80	0.9	0.05	3	150	1.0	
81	0.9	0.05	3	200	0.8	
88	0.9	0.05	4	100	0.9	
172	1.0	0.10	3	100	0.9	
197	1.0	0.15	3	100	1.0	
15	0.8	0.05	4	100	0.8	
171	1.0	0.10	3	100	0.8	
152	1.0	0.05	3	150	1.0	
174	1.0	0.10	3	150	0.8	
32	0.8	0.10	3	150	1.0	
122	0.9	0.15	3	50	1.0	
50	0.8	0.15	3	50	1.0	
41	0.8	0.10	4	100	1.0	

	validation_score	train_score
193	0.819690	0.857945
99	0.819155	0.872124
62	0.818085	0.879080
181	0.817549	0.870519
169	0.817549	0.845639
154	0.817014	0.871322
194	0.816479	0.858480
80	0.816479	0.855270
81	0.815944	0.872124
88	0.815944	0.868379
172	0.815944	0.869449
197	0.815944	0.883360
15	0.815944	0.869984
171	0.815944	0.872659
152	0.815944	0.855538
174	0.815944	0.887908
32	0.815409	0.882825
122	0.815409	0.857143
50	0.815409	0.857945
41	0.815409	0.892991

Observations:

We see similar optimal scores as seen in the first tuning attempt. We will now try tuning regularization parameters lambda and alpha. This time, we will use randomizedsearchcv to reduce tuning time.

[98]:
```
#hyperparameter tuning 3: with L1,L2 regularization and randomizedsearchcv
params = [
    {
        'n_estimators' : [50, 100, 150, 200],
        'learning_rate': [0.05, 0.1, 0.15],
        'subsample': [0.8, 0.9, 1.0],
        'colsample_bytree': [0.8, 0.9, 1.0],
        'max_depth': [2, 3, 4, 5],
        'lambda': [1.4, 1.2, 1, 0.8, 0.6], #default is 1
        'alpha': [0, 0.2, 0.4, 0.6, 0.8, 1] #default is 0
    }
]

xgb = XGBClassifier()
skf = StratifiedKFold(n_splits=3, shuffle = True, random_state = 100)

#xg_tuning_fn = GridSearchCV(xgb, param_grid=params, scoring='accuracy',␣
 ↪n_jobs=4, cv=skf.split(X_train,y_train), random_state=100,␣
 ↪return_train_score=True)
```

```
xg_tuning_fn3 = RandomizedSearchCV(
    xgb,
    param_distributions=params,
    scoring='accuracy',
    n_jobs=4,
    n_iter=100,
    cv=skf.split(X_train,y_train),
    verbose=3,
    return_train_score=True,
    random_state=100)

xg_tuning_fn3, xg_res3, xg_time_taken3 = hyper_tune(xg_tuning_fn3, X_train,␣
 ↪y_train)
print('Hyperparameter tuning time: ', xg_time_taken3)
```

```
Fitting 3 folds for each of 100 candidates, totalling 300 fits
Hyperparameter tuning time:  0:00:32.834791
```

[99]:
```
xg_res3.head(20)
```

[99]:

	subsample	n_estimators	max_depth	learning_rate	lambda	\
35	0.9	150	2	0.10	0.6	
53	1.0	50	3	0.15	1.0	
70	1.0	150	3	0.05	0.8	
57	1.0	50	5	0.10	1.4	
62	0.8	150	3	0.05	0.6	
61	0.9	100	5	0.05	0.6	
99	0.9	200	2	0.10	1.2	
28	0.9	50	5	0.15	1.0	
80	1.0	150	3	0.10	1.4	
32	0.9	150	5	0.05	1.4	
5	1.0	50	3	0.10	1.0	
18	0.9	100	3	0.10	1.0	
67	1.0	50	4	0.05	1.2	
23	0.8	100	4	0.05	1.4	
69	0.8	200	3	0.05	1.4	
55	1.0	200	5	0.05	1.4	
49	0.9	50	5	0.10	1.0	
44	0.9	50	4	0.05	1.2	
37	1.0	100	3	0.05	0.8	
56	0.9	50	3	0.10	1.0	

	colsample_bytree	alpha	validation_score	train_score
35	1.0	0.2	0.820760	0.852060
53	1.0	0.2	0.819155	0.855003
70	1.0	0.4	0.817549	0.855805

57	0.9	0.0	0.817549	0.890583
62	0.8	0.6	0.817014	0.858213
61	0.8	0.6	0.817014	0.889513
99	0.8	1.0	0.816479	0.855805
28	0.8	0.6	0.815944	0.908507
80	0.9	0.8	0.815409	0.876940
32	0.8	1.0	0.815409	0.895666
5	0.8	0.0	0.814874	0.840021
18	0.8	1.0	0.814874	0.863831
67	0.8	0.8	0.814339	0.844569
23	1.0	0.4	0.814339	0.867041
69	0.9	0.6	0.814339	0.864901
55	0.9	1.0	0.813804	0.912253
49	0.9	0.2	0.813804	0.892723
44	0.9	0.0	0.813804	0.847780
37	0.9	1.0	0.813804	0.841091
56	1.0	0.2	0.813804	0.844837

Observations:

After trying various combinations, we get the best validation score of 82.15% for the parameters: **{colsample_bytree:0.6, learning_rate:0.1, max_depth:3, n_estimator:100, subsample:1.0}**. We will now build the final boosting model using these parameters.

[122]:
```
# Build final boosting model

xgb = XGBClassifier(n_estimators=100, learning_rate=0.1, subsample=1.0,␣
 ↪colsample_bytree=0.6, max_depth=3)
xgb.fit(X_train, y_train)
```

[122]:
```
XGBClassifier(base_score=0.5, booster='gbtree', callbacks=None,
              colsample_bylevel=1, colsample_bynode=1, colsample_bytree=0.6,
              early_stopping_rounds=None, enable_categorical=False,
              eval_metric=None, feature_types=None, gamma=0, gpu_id=-1,
              grow_policy='depthwise', importance_type=None,
              interaction_constraints='', learning_rate=0.1, max_bin=256,
              max_cat_threshold=64, max_cat_to_onehot=4, max_delta_step=0,
              max_depth=3, max_leaves=0, min_child_weight=1, missing=nan,
              monotone_constraints='()', n_estimators=100, n_jobs=0,
              num_parallel_tree=1, predictor='auto', random_state=0, …)
```

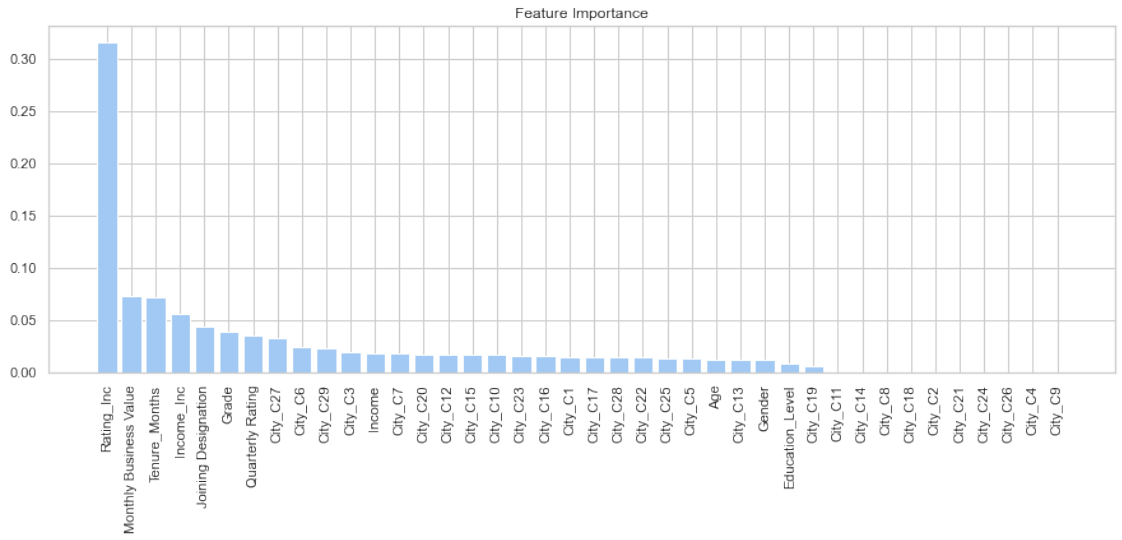
[123]:
```
print('Train score: ', xgb.score(X_train, y_train))
print('Test score: ', xgb.score(X_test, y_test))
```

```
Train score:  0.8533975387907973
Test score:  0.8155136268343816
```

Observations: - The train and test scores for the final models are 85.33% and 81.55% respectively. Compared to the random forest model, not just the validation/test scores have improved, but the

difference between the training and validation/test scores has reduced too. This indicates that boosting model has lower variance than the bagging model.

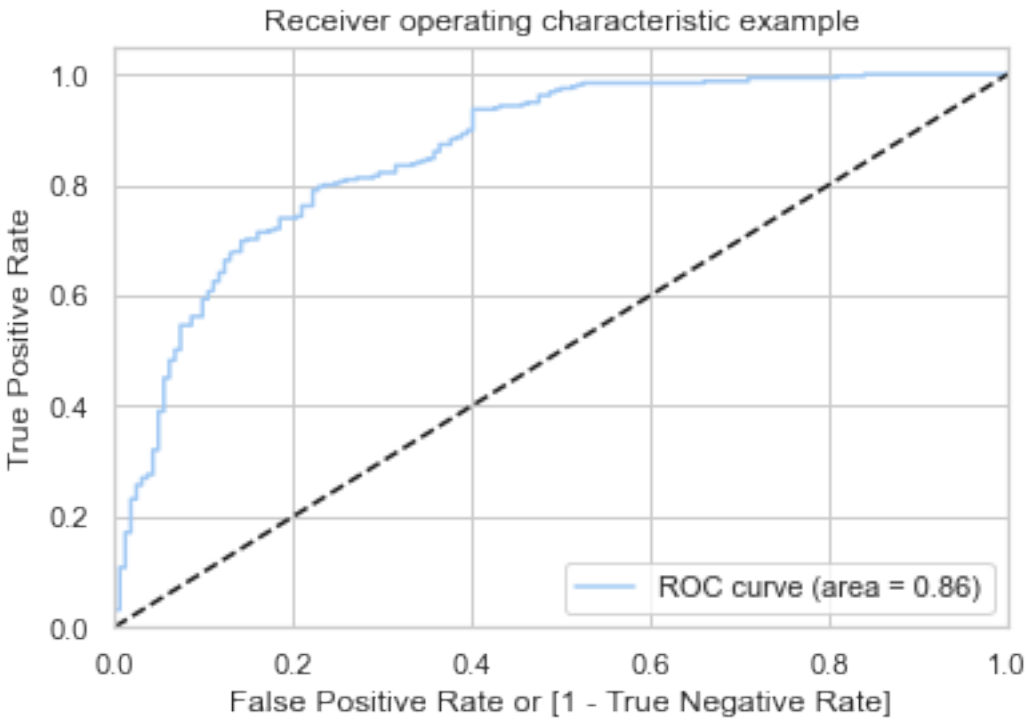
[124]:
```
#plot feature importance
plot_feat_imp(xgb, X_train)
```



[125]:
```
#classification report
y_pred = xgb.predict(X_test)
print(classification_report(y_test, y_pred))
```

```
              precision    recall  f1-score   support

           0       0.84      0.57      0.68       162
           1       0.81      0.94      0.87       315

    accuracy                           0.82       477
   macro avg       0.82      0.76      0.77       477
weighted avg       0.82      0.82      0.80       477
```

[126]:
```
#plot ROC curve
y_pred_prob = xgb.predict_proba(X_test)
fpr, tpr, thresholds = draw_roc(y_test, y_pred_prob[:,1])
```



Observations:

1. We built the final boosting model with this parameter set: {colsample_bytree:1.0, learning_rate:0.1, max_depth:4, n_estimator:50, subsample:1.0}. This model produced test score of 81.76% and training score of 86.48%. We see improvement in the test score compared to boosting/decision tree classifiers.
2. feature importance: Top 3 features (in the order of importance) are: Rating_inc, Monthly business value, tenure_months.
3. precision score: 0.82 (improvement), recall: 0.92 (same as bagging), f1-score: 0.87 (slight improvement)
4. ROC area: 0.86 (same as bagging model)

2.12 Insights

Please refer to the individual sections for more detailed insights.

2.13 Business Recommendations

1. The drivers with the following profiles are more likely to churn.
   - Drivers with monthly business value (MBV) less than 200K are most likly to churn. Drivers with MBA between 200K to 400K are moderately likely to churn. Drivers with MBV between 400K and 700K are lower likely to churn. Those with MBV >700K are least likely to churn.

- Drivers with income < 50K are most likely to churn. Drivers with income between 50K and 75K are also moderately likely to churn. Drivers with income 75K to 150K are lesser likely to churn, and drivers with income > 150K are least likely to churn.
- Grade 1 drivers are very highly likely to churn.
- Drivers with Tenure length of <2 years and between 3-4 years are more likely to churn. Drivers with tenure length > 5 years are least likely to churn.
- Drivers whose rating or income has increased in their tenure are less likely to churn.
- Drivers from city C13, C17, C23, and C2 highly likely to churn.

For drivers with the above profiles, bussiness can plan several preventive actions such as offering better incentives, training/counseling/professional sessions, and engaging with them to understand common patterns for their poor business performance/grades.

2. Drivers' Gender, Education Level, and Age are not significant factor causing churn. Thus business can consider encouraging diversity in drivers on-boarding without risking retention rates.