Problem Statement

Overview

Telecom industry is highly competitive market where customers are able to choose from multiple service provider and actively switch from one operator to another which results in average 15-25% annual churn rate. Given the fact that it costs 5-10 times more to acquire a new customer than to retain an existing one, **customer retention** especially the **high profitable customers** has now become very important. To reduce the customer churn, telecom companies need to predict which customers are at high risk of churn.

There are two main models of payment in the telecom industry -

1. Prepaid: Customers pay/recharge with a certain amount in advance and then use the services.
2. postpaid: Customers pay a monthly/annual bill after using the services.

Churn can essentially be of 2 types:

1. Revenue-based churn: Customers who have not utilised any revenue-generating facilities such as mobile internet, outgoing calls, SMS etc. over a given period of time
2. Usage-based churn: Customers who have not done any usage, either incoming or outgoing - in terms of calls, internet etc. over a period of time.

This project is based on the Indian and Southeast Asian market where the focus should be on **Prepaid customers**, churn type should be **Usage-based churn**.

Business Objective

The dataset contains customer-level information for a span of four consecutive months - June, July, August and September. The months are encoded as 6, 7, 8 and 9, respectively. The business objective is to predict the churn in the last (i.e. the ninth) month using the data (features) from the first three months.

There are three phases of customer **churn lifecycle**:

1. The ‘good’ phase: In this phase, the customer is happy with the service and behaves as usual.
2. The ‘action’ phase: The customer experience starts to sore in this phase, for e.g. he/she gets a compelling offer from a competitor, faces unjust charges, becomes unhappy with service quality etc. In this phase, the customer usually shows different behaviour than the ‘good’ months. Also, it is crucial to identify high-churn-risk customers in this phase, since some corrective actions can be taken at this point (such as matching the competitor’s offer/improving the service quality etc.)
3. The ‘churn’ phase: In this phase, the customer is said to have churned. The churn should be based on this phase.

Prerequisite

Following 2 libraries are the extra Libraries used in this project. They must be installed to proceed. Below is the installation guide to install these libraries.

1. imblearn

Installation guide: https://imbalanced-learn.readthedocs.io/en/stable/install.html (https://imbalanced-learn.readthedocs.io/en/stable/install.html)

2. xgboost

Installation guide: https://xgboost.readthedocs.io/en/latest/build.html (https://xgboost.readthedocs.io/en/latest/build.html)

Note

The **Hyperparameter tuning** is a resource intensive operation which can take significant amount of time to execute depending on the system performance. The execution time depends on the system configuration like RAM. CPU. OS. etc.

In [1]:

```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
from sklearn import linear_model
from sklearn.model_selection import GridSearchCV
from sklearn.metrics import r2_score
import gc

import warnings
warnings.filterwarnings('ignore')
%matplotlib inline
# setting display format so that large values are shown properly
pd.set_option('display.float_format', lambda x: '%.4f' % x)
pd.set_option('display.max_rows', None)
pd.set_option('display.max_columns', None)

sns.set_style(style='dark')
sns.set_context("notebook")
```

In [2]:

```
telecom_data = pd.read_csv('telecom_churn_data.csv')
```

In [3]:

```
telecom_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 99999 entries, 0 to 99998
Columns: 226 entries, mobile_number to sep_vbc_3g
dtypes: float64(179), int64(35), object(12)
memory usage: 172.4+ MB
```

In [4]:

```
def get_columns_with_nan_percentage(df):
    nan_cols = [{
        "column":
        c,
        "percentage":
        round(100 * (df[c].isnull().sum() / len(df[c].index)), 2),
        "type":
        df[c].dtype
    } for c in df.columns
                if round(100 *
                         (df[c].isnull().sum() / len(df[c].index)), 2) > 0]

    if len(nan_cols)>0:
        return pd.DataFrame.from_records(nan_cols).sort_values(by=['percentage'
],
                                                               ascending=False)
    else:
        return pd.DataFrame.from_records(nan_cols)


def convert_to_category(columns):
    for column in columns:
        telecom_data[column] = telecom_data[column].astype('object')


def get_int_float_columns_with_Zero_percentage(df):
    nan_cols = [{
        "column":
        c,
        "percentage":
        round(100 * ((df[c] == 0).sum() / len(df[c].index)), 2),
        "type":
        df[c].dtype
    } for c in df.columns
                if round(100 *
                         ((df[c] == 0).sum().sum() / len(df[c].index)), 2) > 0]
    return pd.DataFrame.from_records(nan_cols)


def get_columns_with_similar_values(df, threshold):
    columns_to_delete = []
    for c in df.columns:
        if (any(y >= threshold for y in df[c].value_counts(
                dropna=False, normalize=True).tolist())):
            columns_to_delete.append(c)
    return columns_to_delete
```

High value customers

In order to identify the high value customer, we need to find the customers who spent the most. For this we will use

    total_rech_data_6, av_rech_amt_data_6, total_rech_data_7 , av_rech_amt_data_7, total_rech_amt_6 and total_rech_amt_7

In [5]:

```
columns_high_value_calculation= ['total_rech_data_6', 'av_rech_amt_data_6', 'total_rech_data_7' , 'av_rech_amt_data_7', 'total_rech_amt_6', 'total_rech_amt_7']

telecom_data[columns_high_value_calculation].describe()
```

Out[5]:

	total_rech_data_6	av_rech_amt_data_6	total_rech_data_7	av_rech_amt_data_7	total_rec
count	25153.0000	25153.0000	25571.0000	25571.0000	99
mean	2.4638	192.6010	2.6664	200.9813	
std	2.7891	192.6463	3.0316	196.7912	
min	1.0000	1.0000	1.0000	0.5000	
25%	1.0000	82.0000	1.0000	92.0000	
50%	1.0000	154.0000	1.0000	154.0000	
75%	3.0000	252.0000	3.0000	252.0000	
max	61.0000	7546.0000	54.0000	4365.0000	35

In [6]:

```
nan_df = get_columns_with_nan_percentage(telecom_data[columns_high_value_calculation])
nan_df
```

Out[6]:

	column	percentage	type
0	total_rech_data_6	74.8500	float64
1	av_rech_amt_data_6	74.8500	float64
2	total_rech_data_7	74.4300	float64
3	av_rech_amt_data_7	74.4300	float64

If we see the minimum for each type of column, we can see it is 1. So, we can put 0 in case of NAN for the customers

In [7]:

```
telecom_data[columns_high_value_calculation] = telecom_data[columns_high_value_calculation].fillna(0)
```

In [8]:

```
# Similarly, we can also put 0 for month 8
telecom_data[['total_rech_data_8', 'av_rech_amt_data_8','total_rech_amt_8']] = t
elecom_data[['total_rech_data_8', 'av_rech_amt_data_8','total_rech_amt_8']].fill
na(0)
```

In [9]:

```
def get_average_recharge(row):
    amount = 0.0
    amount += row['total_rech_data_6'] * row['av_rech_amt_data_6']
    amount += row['total_rech_data_7'] * row['av_rech_amt_data_7']
    amount += row['total_rech_amt_6']
    amount += row['total_rech_amt_7']

    return amount / 2.0


telecom_data['average_recharge_amount'] = telecom_data.apply(
    get_average_recharge, axis=1)
```

In [10]:

```
percentile_70 = telecom_data['average_recharge_amount'].quantile(.7)
percentile_70
```

Out[10]:

```
478.0
```

In [11]:

```
# As per the problem statement, we need to consider customers as high value if t
hey have more than 70 percentile expense

def check_high_value_customer(row):
    return 1 if row['average_recharge_amount'] > percentile_70 else 0


telecom_data['high_value_customer'] = telecom_data.apply(
    check_high_value_customer, axis=1)
```

In [12]:

```
telecom_data['high_value_customer'].value_counts()
```

Out[12]:

```
0    70046
1    29953
Name: high_value_customer, dtype: int64
```

We can see that there are **29953** high value customer. From now onwards, we will only consider these customers for further analysis.

In [13]:

```
# Getting all the high value customers
telecom_data = telecom_data[telecom_data['high_value_customer'] == 1]
gc.collect()
```

Out[13]:

```
11
```

In [14]:

```
telecom_data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Int64Index: 29953 entries, 0 to 99997
Columns: 228 entries, mobile_number to high_value_customer
dtypes: float64(180), int64(36), object(12)
memory usage: 52.3+ MB
```

Churned customers

In [15]:

```
def check_churn(row):
     return 1 if (row['total_ic_mou_9'] == 0 and row['total_og_mou_9'] == 0 and
row['vol_2g_mb_9'] == 0 and row['vol_3g_mb_9'] == 0)  else 0

telecom_data['churn'] = telecom_data.apply(check_churn, axis=1)
telecom_data['churn'] = telecom_data['churn'].astype('category')
```

In [16]:

```
telecom_data['churn'].value_counts()
```

Out[16]:

```
0    27520
1     2433
Name: churn, dtype: int64
```

Now we will delete columns for **September**

In [17]:

```
def get_columns_by_pattern(df,func):
    return [c for c in telecom_data.columns if func(c)]

telecom_data = telecom_data.drop('sep_vbc_3g',axis=1)
```

In [18]:

```
columns_to_drop = get_columns_by_pattern(telecom_data,
                                       lambda x: x.endswith("_9"))
telecom_data = telecom_data.drop(columns_to_drop, axis=1)
```

In [19]:

```
# We can delete the mobile number column as it will not help in analysis
telecom_data = telecom_data.drop('mobile_number',axis=1)
```

In [20]:

```
nan_df = get_columns_with_nan_percentage(telecom_data)
nan_df
```

Out[20]:

	column	percentage	type
118	fb_user_8	46.8000	float64
112	arpu_2g_8	46.8000	float64
100	max_rech_data_8	46.8000	float64
103	count_rech_2g_8	46.8000	float64
106	count_rech_3g_8	46.8000	float64
109	arpu_3g_8	46.8000	float64
97	date_of_last_rech_data_8	46.8000	object
115	night_pck_user_8	46.8000	float64
116	fb_user_6	44.1100	float64
110	arpu_2g_6	44.1100	float64
101	count_rech_2g_6	44.1100	float64
98	max_rech_data_6	44.1100	float64
107	arpu_3g_6	44.1100	float64
113	night_pck_user_6	44.1100	float64
95	date_of_last_rech_data_6	44.1100	object
104	count_rech_3g_6	44.1100	float64
105	count_rech_3g_7	43.1200	float64
99	max_rech_data_7	43.1200	float64
117	fb_user_7	43.1200	float64
102	count_rech_2g_7	43.1200	float64
96	date_of_last_rech_data_7	43.1200	object
108	arpu_3g_7	43.1200	float64
114	night_pck_user_7	43.1200	float64
111	arpu_2g_7	43.1200	float64
79	std_ic_t2o_mou_8	3.9100	float64
46	std_og_mou_8	3.9100	float64
49	isd_og_mou_8	3.9100	float64
82	std_ic_mou_8	3.9100	float64
43	std_og_t2c_mou_8	3.9100	float64
55	og_others_8	3.9100	float64
76	std_ic_t2f_mou_8	3.9100	float64
52	spl_og_mou_8	3.9100	float64
73	std_ic_t2m_mou_8	3.9100	float64
40	std_og_t2f_mou_8	3.9100	float64
70	std_ic_t2t_mou_8	3.9100	float64
58	loc_ic_t2t_mou_8	3.9100	float64
67	loc_ic_mou_8	3.9100	float64

	column	percentage	type
61	loc_ic_t2m_mou_8	3.9100	float64
85	spl_ic_mou_8	3.9100	float64
31	loc_og_mou_8	3.9100	float64
88	isd_ic_mou_8	3.9100	float64
64	loc_ic_t2f_mou_8	3.9100	float64
7	onnet_mou_8	3.9100	float64
10	offnet_mou_8	3.9100	float64
13	roam_ic_mou_8	3.9100	float64
16	roam_og_mou_8	3.9100	float64
19	loc_og_t2t_mou_8	3.9100	float64
22	loc_og_t2m_mou_8	3.9100	float64
25	loc_og_t2f_mou_8	3.9100	float64
28	loc_og_t2c_mou_8	3.9100	float64
34	std_og_t2t_mou_8	3.9100	float64
91	ic_others_8	3.9100	float64
37	std_og_t2m_mou_8	3.9100	float64
94	date_of_last_rech_8	1.9400	object
77	std_ic_t2o_mou_6	1.8100	float64
86	isd_ic_mou_6	1.8100	float64
74	std_ic_t2f_mou_6	1.8100	float64
80	std_ic_mou_6	1.8100	float64
71	std_ic_t2m_mou_6	1.8100	float64
83	spl_ic_mou_6	1.8100	float64
68	std_ic_t2t_mou_6	1.8100	float64
89	ic_others_6	1.8100	float64
65	loc_ic_mou_6	1.8100	float64
59	loc_ic_t2m_mou_6	1.8100	float64
20	loc_og_t2m_mou_6	1.8100	float64
26	loc_og_t2c_mou_6	1.8100	float64
44	std_og_mou_6	1.8100	float64
23	loc_og_t2f_mou_6	1.8100	float64
47	isd_og_mou_6	1.8100	float64
35	std_og_t2m_mou_6	1.8100	float64
29	loc_og_mou_6	1.8100	float64
50	spl_og_mou_6	1.8100	float64
17	loc_og_t2t_mou_6	1.8100	float64
38	std_og_t2f_mou_6	1.8100	float64
53	og_others_6	1.8100	float64
14	roam_og_mou_6	1.8100	float64

	column	percentage	type
56	loc_ic_t2t_mou_6	1.8100	float64
11	roam_ic_mou_6	1.8100	float64
62	loc_ic_t2f_mou_6	1.8100	float64
8	offnet_mou_6	1.8100	float64
32	std_og_t2t_mou_6	1.8100	float64
5	onnet_mou_6	1.8100	float64
41	std_og_t2c_mou_6	1.8100	float64
30	loc_og_mou_7	1.7900	float64
27	loc_og_t2c_mou_7	1.7900	float64
63	loc_ic_t2f_mou_7	1.7900	float64
24	loc_og_t2f_mou_7	1.7900	float64
21	loc_og_t2m_mou_7	1.7900	float64
18	loc_og_t2t_mou_7	1.7900	float64
15	roam_og_mou_7	1.7900	float64
12	roam_ic_mou_7	1.7900	float64
9	offnet_mou_7	1.7900	float64
6	onnet_mou_7	1.7900	float64
33	std_og_t2t_mou_7	1.7900	float64
36	std_og_t2m_mou_7	1.7900	float64
75	std_ic_t2f_mou_7	1.7900	float64
60	loc_ic_t2m_mou_7	1.7900	float64
69	std_ic_t2t_mou_7	1.7900	float64
57	loc_ic_t2t_mou_7	1.7900	float64
72	std_ic_t2m_mou_7	1.7900	float64
54	og_others_7	1.7900	float64
51	spl_og_mou_7	1.7900	float64
78	std_ic_t2o_mou_7	1.7900	float64
48	isd_og_mou_7	1.7900	float64
81	std_ic_mou_7	1.7900	float64
45	std_og_mou_7	1.7900	float64
84	spl_ic_mou_7	1.7900	float64
42	std_og_t2c_mou_7	1.7900	float64
87	isd_ic_mou_7	1.7900	float64
39	std_og_t2f_mou_7	1.7900	float64
90	ic_others_7	1.7900	float64
66	loc_ic_mou_7	1.7900	float64
2	loc_ic_t2o_mou	0.7400	float64
0	loc_og_t2o_mou	0.7400	float64
1	std_og_t2o_mou	0.7400	float64

	column	percentage	type
4	last_date_of_month_8	0.5500	object
93	date_of_last_rech_7	0.3300	object
92	date_of_last_rech_6	0.2400	object
3	last_date_of_month_7	0.0900	object

Among the columns, following are categorical columns

```
fb_user_6,fb_user_7,fb_user_8,night_pck_user_6,night_pck_user_7,night_pck_user_8
```

In [21]:

```
categorical_columns = ['fb_user_6','fb_user_7','fb_user_8','night_pck_user_6','night_pck_user_7','night_pck_user_8']
```

In [22]:

```
# We will put -1 as a new category for the null in the above mentioned columns.

telecom_data[categorical_columns] = telecom_data[categorical_columns].fillna(-1)
telecom_data[categorical_columns] = telecom_data[categorical_columns].astype('int').astype('category')
```

In [23]:

```
telecom_data.describe()
```

Out[23]:

	circle_id	loc_og_t2o_mou	std_og_t2o_mou	loc_ic_t2o_mou	arpu_6	arpu_7
count	29953.0000	29730.0000	29730.0000	29730.0000	29953.0000	29953.0000
mean	109.0000	0.0000	0.0000	0.0000	558.8201	561.1605
std	0.0000	0.0000	0.0000	0.0000	460.8682	480.0285
min	109.0000	0.0000	0.0000	0.0000	-2258.7090	-2014.0450
25%	109.0000	0.0000	0.0000	0.0000	310.1420	310.0710
50%	109.0000	0.0000	0.0000	0.0000	482.3540	481.4960
75%	109.0000	0.0000	0.0000	0.0000	700.2400	698.8290
max	109.0000	0.0000	0.0000	0.0000	27731.0880	35145.8340

In [24]:

```
nan_df = get_columns_with_nan_percentage(telecom_data)
nan_df
```

Out[24]:

	column	percentage	type
112	arpu_2g_8	46.8000	float64
106	count_rech_3g_8	46.8000	float64
97	date_of_last_rech_data_8	46.8000	object
103	count_rech_2g_8	46.8000	float64
100	max_rech_data_8	46.8000	float64
109	arpu_3g_8	46.8000	float64
101	count_rech_2g_6	44.1100	float64
104	count_rech_3g_6	44.1100	float64
98	max_rech_data_6	44.1100	float64
107	arpu_3g_6	44.1100	float64
110	arpu_2g_6	44.1100	float64
95	date_of_last_rech_data_6	44.1100	object
102	count_rech_2g_7	43.1200	float64
99	max_rech_data_7	43.1200	float64
105	count_rech_3g_7	43.1200	float64
111	arpu_2g_7	43.1200	float64
96	date_of_last_rech_data_7	43.1200	object
108	arpu_3g_7	43.1200	float64
79	std_ic_t2o_mou_8	3.9100	float64
40	std_og_t2f_mou_8	3.9100	float64
46	std_og_mou_8	3.9100	float64
76	std_ic_t2f_mou_8	3.9100	float64
43	std_og_t2c_mou_8	3.9100	float64
37	std_og_t2m_mou_8	3.9100	float64
73	std_ic_t2m_mou_8	3.9100	float64
67	loc_ic_mou_8	3.9100	float64
70	std_ic_t2t_mou_8	3.9100	float64
49	isd_og_mou_8	3.9100	float64
34	std_og_t2t_mou_8	3.9100	float64
52	spl_og_mou_8	3.9100	float64
64	loc_ic_t2f_mou_8	3.9100	float64
55	og_others_8	3.9100	float64
61	loc_ic_t2m_mou_8	3.9100	float64
82	std_ic_mou_8	3.9100	float64
58	loc_ic_t2t_mou_8	3.9100	float64
19	loc_og_t2t_mou_8	3.9100	float64
7	onnet_mou_8	3.9100	float64

	column	percentage	type
31	loc_og_mou_8	3.9100	float64
88	isd_ic_mou_8	3.9100	float64
10	offnet_mou_8	3.9100	float64
13	roam_ic_mou_8	3.9100	float64
28	loc_og_t2c_mou_8	3.9100	float64
91	ic_others_8	3.9100	float64
85	spl_ic_mou_8	3.9100	float64
25	loc_og_t2f_mou_8	3.9100	float64
22	loc_og_t2m_mou_8	3.9100	float64
16	roam_og_mou_8	3.9100	float64
94	date_of_last_rech_8	1.9400	object
83	spl_ic_mou_6	1.8100	float64
62	loc_ic_t2f_mou_6	1.8100	float64
65	loc_ic_mou_6	1.8100	float64
71	std_ic_t2m_mou_6	1.8100	float64
68	std_ic_t2t_mou_6	1.8100	float64
74	std_ic_t2f_mou_6	1.8100	float64
59	loc_ic_t2m_mou_6	1.8100	float64
77	std_ic_t2o_mou_6	1.8100	float64
80	std_ic_mou_6	1.8100	float64
86	isd_ic_mou_6	1.8100	float64
89	ic_others_6	1.8100	float64
56	loc_ic_t2t_mou_6	1.8100	float64
32	std_og_t2t_mou_6	1.8100	float64
14	roam_og_mou_6	1.8100	float64
35	std_og_t2m_mou_6	1.8100	float64
26	loc_og_t2c_mou_6	1.8100	float64
38	std_og_t2f_mou_6	1.8100	float64
23	loc_og_t2f_mou_6	1.8100	float64
41	std_og_t2c_mou_6	1.8100	float64
20	loc_og_t2m_mou_6	1.8100	float64
44	std_og_mou_6	1.8100	float64
47	isd_og_mou_6	1.8100	float64
17	loc_og_t2t_mou_6	1.8100	float64
50	spl_og_mou_6	1.8100	float64
11	roam_ic_mou_6	1.8100	float64
53	og_others_6	1.8100	float64
8	offnet_mou_6	1.8100	float64
5	onnet_mou_6	1.8100	float64

	column	percentage	type
29	loc_og_mou_6	1.8100	float64
12	roam_ic_mou_7	1.7900	float64
15	roam_og_mou_7	1.7900	float64
30	loc_og_mou_7	1.7900	float64
18	loc_og_t2t_mou_7	1.7900	float64
9	offnet_mou_7	1.7900	float64
21	loc_og_t2m_mou_7	1.7900	float64
24	loc_og_t2f_mou_7	1.7900	float64
6	onnet_mou_7	1.7900	float64
27	loc_og_t2c_mou_7	1.7900	float64
90	ic_others_7	1.7900	float64
60	loc_ic_t2m_mou_7	1.7900	float64
87	isd_ic_mou_7	1.7900	float64
75	std_ic_t2f_mou_7	1.7900	float64
63	loc_ic_t2f_mou_7	1.7900	float64
54	og_others_7	1.7900	float64
66	loc_ic_mou_7	1.7900	float64
51	spl_og_mou_7	1.7900	float64
69	std_ic_t2t_mou_7	1.7900	float64
48	isd_og_mou_7	1.7900	float64
72	std_ic_t2m_mou_7	1.7900	float64
57	loc_ic_t2t_mou_7	1.7900	float64
45	std_og_mou_7	1.7900	float64
42	std_og_t2c_mou_7	1.7900	float64
78	std_ic_t2o_mou_7	1.7900	float64
39	std_og_t2f_mou_7	1.7900	float64
81	std_ic_mou_7	1.7900	float64
36	std_og_t2m_mou_7	1.7900	float64
84	spl_ic_mou_7	1.7900	float64
33	std_og_t2t_mou_7	1.7900	float64
1	std_og_t2o_mou	0.7400	float64
2	loc_ic_t2o_mou	0.7400	float64
0	loc_og_t2o_mou	0.7400	float64
4	last_date_of_month_8	0.5500	object
93	date_of_last_rech_7	0.3300	object
92	date_of_last_rech_6	0.2400	object
3	last_date_of_month_7	0.0900	object

We can see many columns which has count in them and have NAN. We can put 0 as the default value for such columns

In [25]:

```
counts_column = get_columns_by_pattern(telecom_data,
                                       lambda x: x.startswith('count_'))
telecom_data[counts_column] = telecom_data[counts_column].fillna(0)
```

In [26]:

```
def segregate_columns(df, n=10):
    segregation = [{
        "col": c,
        "col_type": 'continuous' if df[c].nunique(dropna=False) > n else 'catego
rical',
        "unique_count": df[c].nunique(dropna=False),
         "na_percentage": round(100 * (df[c].isnull().sum() / len(df[c].index)),
2),
    } for c in df.columns]
    return pd.DataFrame.from_records(segregation).sort_values(by=['unique_count'
],
                                                           ascending=True).reset
_index()
```

In [27]:

```
nan_df_numerical = nan_df[nan_df['type'] == 'float64']['column']
nan_df_numerical
```

Out[27]:

```
112           arpu_2g_8
106      count_rech_3g_8
103      count_rech_2g_8
100      max_rech_data_8
109           arpu_3g_8
101      count_rech_2g_6
104      count_rech_3g_6
98       max_rech_data_6
107           arpu_3g_6
110           arpu_2g_6
102      count_rech_2g_7
99       max_rech_data_7
105      count_rech_3g_7
111           arpu_2g_7
108           arpu_3g_7
79      std_ic_t2o_mou_8
40      std_og_t2f_mou_8
46          std_og_mou_8
76      std_ic_t2f_mou_8
43      std_og_t2c_mou_8
37      std_og_t2m_mou_8
73      std_ic_t2m_mou_8
67          loc_ic_mou_8
70      std_ic_t2t_mou_8
49          isd_og_mou_8
34      std_og_t2t_mou_8
52          spl_og_mou_8
64      loc_ic_t2f_mou_8
55           og_others_8
61      loc_ic_t2m_mou_8
82          std_ic_mou_8
58      loc_ic_t2t_mou_8
19      loc_og_t2t_mou_8
7            onnet_mou_8
31          loc_og_mou_8
88          isd_ic_mou_8
10          offnet_mou_8
13         roam_ic_mou_8
28      loc_og_t2c_mou_8
91           ic_others_8
85          spl_ic_mou_8
25      loc_og_t2f_mou_8
22      loc_og_t2m_mou_8
16         roam_og_mou_8
83          spl_ic_mou_6
62      loc_ic_t2f_mou_6
65          loc_ic_mou_6
71      std_ic_t2m_mou_6
68      std_ic_t2t_mou_6
74      std_ic_t2f_mou_6
59      loc_ic_t2m_mou_6
77      std_ic_t2o_mou_6
80          std_ic_mou_6
86          isd_ic_mou_6
89           ic_others_6
56      loc_ic_t2t_mou_6
32      std_og_t2t_mou_6
14         roam_og_mou_6
35      std_og_t2m_mou_6
```

```
26      loc_og_t2c_mou_6
38      std_og_t2f_mou_6
23      loc_og_t2f_mou_6
41      std_og_t2c_mou_6
20      loc_og_t2m_mou_6
44          std_og_mou_6
47          isd_og_mou_6
17      loc_og_t2t_mou_6
50          spl_og_mou_6
11         roam_ic_mou_6
53           og_others_6
8           offnet_mou_6
5            onnet_mou_6
29          loc_og_mou_6
12         roam_ic_mou_7
15         roam_og_mou_7
30          loc_og_mou_7
18      loc_og_t2t_mou_7
9           offnet_mou_7
21      loc_og_t2m_mou_7
24      loc_og_t2f_mou_7
6            onnet_mou_7
27      loc_og_t2c_mou_7
90           ic_others_7
60      loc_ic_t2m_mou_7
87          isd_ic_mou_7
75      std_ic_t2f_mou_7
63      loc_ic_t2f_mou_7
54           og_others_7
66          loc_ic_mou_7
51          spl_og_mou_7
69      std_ic_t2t_mou_7
48          isd_og_mou_7
72      std_ic_t2m_mou_7
57      loc_ic_t2t_mou_7
45          std_og_mou_7
42      std_og_t2c_mou_7
78      std_ic_t2o_mou_7
39      std_og_t2f_mou_7
81          std_ic_mou_7
36      std_og_t2m_mou_7
84          spl_ic_mou_7
33      std_og_t2t_mou_7
1         std_og_t2o_mou
2         loc_ic_t2o_mou
0         loc_og_t2o_mou
Name: column, dtype: object
```

We can see that there are many columns which are float and have NAN. We can fill them with 0

In [28]:

```
telecom_data[nan_df_numerical] = telecom_data[nan_df_numerical].fillna(0)
```

In [29]:

```
nan_df = get_columns_with_nan_percentage(telecom_data)
nan_df
```

Out[29]:

	column	percentage	type
7	date_of_last_rech_data_8	46.8000	object
5	date_of_last_rech_data_6	44.1100	object
6	date_of_last_rech_data_7	43.1200	object
4	date_of_last_rech_8	1.9400	object
1	last_date_of_month_8	0.5500	object
3	date_of_last_rech_7	0.3300	object
2	date_of_last_rech_6	0.2400	object
0	last_date_of_month_7	0.0900	object

We will be deleting the date columns as there is no significance use of these columns and we have multiple other columns which have correlation with them like number of recharge etc.

In [30]:

```
date_columns_to_drop = get_columns_by_pattern(telecom_data,lambda x: 'date' in x
)
telecom_data = telecom_data.drop(date_columns_to_drop, axis=1)
```

In [31]:

```
nan_df = get_columns_with_nan_percentage(telecom_data)
nan_df
```

Out[31]:

—

We can see now there are no columns with NAN

Deleting column with no variance

In [32]:

```
columns_with_more_than_100_percent_same_value = get_columns_with_similar_values(
telecom_data,1)
columns_with_more_than_100_percent_same_value
```

Out[32]:

```
['circle_id',
 'loc_og_t2o_mou',
 'std_og_t2o_mou',
 'loc_ic_t2o_mou',
 'std_og_t2c_mou_6',
 'std_og_t2c_mou_7',
 'std_og_t2c_mou_8',
 'std_ic_t2o_mou_6',
 'std_ic_t2o_mou_7',
 'std_ic_t2o_mou_8',
 'high_value_customer']
```

In [33]:

```
telecom_data = telecom_data.drop(columns_with_more_than_100_percent_same_value,
                                 axis=1)
```

In [34]:

```
telecom_data.shape
```

Out[34]:

```
(29953, 153)
```

New derived features

In [35]:

```
# Creating new columns for the month wise amount spent
def month_wise_amount_spent(row, month):
    return row['total_rech_amt_' + month] + (row['total_rech_data_' + month] *
                                              row['av_rech_amt_data_' + month])
```

In [36]:

```
telecom_data['total_amount_spent_6'] = telecom_data.apply(
    month_wise_amount_spent, args=('6'), axis=1)
telecom_data['total_amount_spent_7'] = telecom_data.apply(
    month_wise_amount_spent, args=('7'), axis=1)
telecom_data['total_amount_spent_8'] = telecom_data.apply(
    month_wise_amount_spent, args=('8'), axis=1)
```

Now we can delete the columns which we used to calculate the total monthly amount.

In [37]:

```
telecom_data = telecom_data.drop([
    'total_rech_amt_6', 'total_rech_amt_7', 'total_rech_amt_8',
    'total_rech_data_6', 'total_rech_data_7', 'total_rech_data_8',
    'av_rech_amt_data_6', 'av_rech_amt_data_7', 'av_rech_amt_data_8',
    'total_rech_num_6', 'total_rech_num_7', 'total_rech_num_8',
    'max_rech_amt_6', 'max_rech_amt_7', 'max_rech_amt_8', 'max_rech_data_6',
    'max_rech_data_7', 'max_rech_data_8'
],
                                 axis=1)
```

As we have added average amount for month of June and July, we can drop the total amount spent column.

In [38]:

```
telecom_data = telecom_data.drop(
    ['total_amount_spent_6', 'total_amount_spent_7'], axis=1)
```

In [39]:

```
telecom_data.shape
```

Out[39]:

```
(29953, 136)
```

In [40]:

```
def merge_column_by_month(df, pattern, month, final_column_name):
    value = 0
    for p in pattern:
        value += value + df[p + month]
    df[final_column_name + month] = value
```

In [41]:

```
# We can derive new column total_data_mb_* for each month by combining 2g and 3g
data
for m in ['6', '7', '8']:
    merge_column_by_month(telecom_data, ['vol_2g_mb_', 'vol_3g_mb_'], m,
                          'total_data_mb_')
```

In [42]:

```
# We can derive new column total_recharge_count_* for each month by combining 2g
and 3g data recharge count
for m in ['6', '7', '8']:
    merge_column_by_month(telecom_data, ['count_rech_3g_', 'count_rech_3g_'], m,
                          'total_recharge_count_')
```

In [43]:

```
# We can derive new column total_arpu_data_* for each month by combining 2g and
 3g arpu
for m in ['6', '7', '8']:
    merge_column_by_month(telecom_data, ['arpu_3g_', 'arpu_2g_'], m,
                          'total_arpu_data_')
```

In [44]:

```
# We can derive new column total_sachet_data_* for each month by combining 2g an
d 3g sachet
for m in ['6', '7', '8']:
    merge_column_by_month(telecom_data, ['sachet_3g_', 'sachet_2g_'], m,
                          'total_sachet_data_')
```

As we have created new combined columns, we can drop the individual columns

In [45]:

```
telecom_data = telecom_data.drop([
    'vol_2g_mb_6', 'vol_2g_mb_7', 'vol_2g_mb_8', 'vol_3g_mb_6', 'vol_3g_mb_7',
    'vol_3g_mb_8', 'count_rech_3g_6', 'count_rech_3g_7', 'count_rech_3g_8',
    'count_rech_2g_6', 'count_rech_2g_7', 'count_rech_2g_8', 'arpu_3g_6',
    'arpu_3g_7', 'arpu_3g_8', 'arpu_2g_6', 'arpu_2g_7', 'arpu_2g_8',
    'sachet_3g_6', 'sachet_3g_7', 'sachet_3g_8', 'sachet_2g_6', 'sachet_2g_7',
    'sachet_2g_8'
],
                                 axis=1)
```

In [46]:

```
gc.collect()
telecom_data.shape
```

Out[46]:

```
(29953, 124)
```

Analysis of the data

Reference for the following methods: [https://towardsdatascience.com/a-starter-pack-to-exploratory-data-analysis-with-python-pandas-seaborn-and-scikit-learn-a77889485baf#89dd (https://towardsdatascience.com/a-starter-pack-to-exploratory-data-analysis-with-python-pandas-seaborn-and-scikit-learn-a77889485baf#89dd)](https://towardsdatascience.com/a-starter-pack-to-exploratory-data-analysis-with-python-pandas-seaborn-and-scikit-learn-a77889485baf#89dd) and the previous assignments.

In [47]:

```
default_figsize = (15, 5)
default_xtick_angle = 50
```

In [48]:

```
def categorical_summarized(dataframe,
                           x=None,
                           y=None,
                           hue=None,
                           palette='Set1',
                           verbose=True,
                           figsize=default_figsize,
                           title="",
                           xlabel=None,
                           ylabel=None,
                           rotate_labels=False):
    '''
    Helper function that gives a quick summary of a given column of categorical
 data
    Arguments
    =========
    dataframe: pandas dataframe
    x: str. horizontal axis to plot the labels of categorical data, y would be t
he count
    y: str. vertical axis to plot the labels of categorical data, x would be the
count
    hue: str. if you want to compare it another variable (usually the target var
iable)
    palette: array-like. Colour of the plot
    Returns
    =======
    Quick Stats of the data and also the count plot
    '''
    if x == None:
        column_interested = y
    else:
        column_interested = x
    series = dataframe[column_interested]

    if verbose:
        print(series.describe())
        print('mode: ', series.mode())
        print('=' * 80)
        print(series.value_counts())

    sns.set(rc={'figure.figsize': figsize})
    sorted_df = dataframe.sort_values(column_interested)
    ax = sns.countplot(x=x, y=y, hue=hue, data=sorted_df)

    plt.title(title)
    if not xlabel:
        xlabel = column_interested
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    total = len(dataframe[column_interested])
    if rotate_labels:
        plt.setp(ax.get_xticklabels(),
                 rotation=30,
                 horizontalalignment='right')
    for p in ax.patches:
        percentage = '{:.1f}%'.format(100 * p.get_height() / total)
        x = p.get_x() + p.get_width() + 0.02
        y = p.get_y() + p.get_height() / 2
        ax.annotate(percentage, (x, y))
```

```
    plt.tight_layout()
    plt.style.use('fivethirtyeight')
    plt.xticks(rotation=default_xtick_angle)
    plt.show()
```

In [49]:

```
def quantitative_summarized(dataframe,
                            x=None,
                            y=None,
                            hue=None,
                            palette='Set1',
                            ax=None,
                            verbose=True,
                            swarm=False,
                            figsize=default_figsize):
    '''
    Helper function that gives a quick summary of quantattive data
    Arguments
    =========
    dataframe: pandas dataframe
    x: str. horizontal axis to plot the labels of categorical data (usually the
 target variable)
    y: str. vertical axis to plot the quantitative data
    hue: str. if you want to compare it another categorical variable (usually th
e target variable if x is another variable)
    palette: array-like. Colour of the plot
    swarm: if swarm is set to True, a swarm plot would be overlayed
    Returns
    =======
    Quick Stats of the data and also the box plot of the distribution
    '''
    series = dataframe[y]
    print(series.describe())
    if verbose:
        print('mode: ', series.mode())
        print('=' * 80)
        print(series.value_counts())
    sns.set(rc={'figure.figsize': figsize})

    sns.boxplot(x=x, y=y, hue=hue, data=dataframe, palette=palette, ax=ax)

    if swarm:
        sns.swarmplot(x=x,
                      y=y,
                      hue=hue,
                      data=dataframe,
                      palette=palette,
                      ax=ax)
    plt.tight_layout()
    plt.style.use('fivethirtyeight')
    plt.xticks(rotation=default_xtick_angle)
    plt.show()
```

In [50]:

```
def plot_column(df,
                col,
                chart_type='Hist',
                dtype=int,
                bins=25,
                figsize=default_figsize):
    temp_df = df[col]
    sns.set(rc={'figure.figsize': figsize})
    if chart_type == 'Hist':
        ax = sns.countplot(temp_df)
    elif chart_type == 'Dens':
        ax = sns.distplot(temp_df)
    xmin, xmax = ax.get_xlim()
    ax.set_xticks(np.round(np.linspace(xmin, xmax, bins), 2))
    plt.tight_layout()
    plt.locator_params(axis='y', nbins=6)
    plt.xticks(rotation=default_xtick_angle)
    plt.style.use('fivethirtyeight')
    plt.show()
```

In [51]:

```
def univariate_analysis(col,
                        chart_type='Dens',
                        df=telecom_data,
                        is_categorical=False,
                        title="",
                        xlabel=None,
                        ylabel=None,
                        rotate_labels=False,
                        bins=25):
    if is_categorical:
        categorical_summarized(df,
                               x=col,
                               title=title,
                               xlabel=xlabel,
                               ylabel=ylabel,
                               rotate_labels=rotate_labels,
                               verbose=False)
    else:
        quantitative_summarized(df, y=col, verbose=False)
        plot_column(df, col, chart_type=chart_type, bins=bins)
```

In [52]:

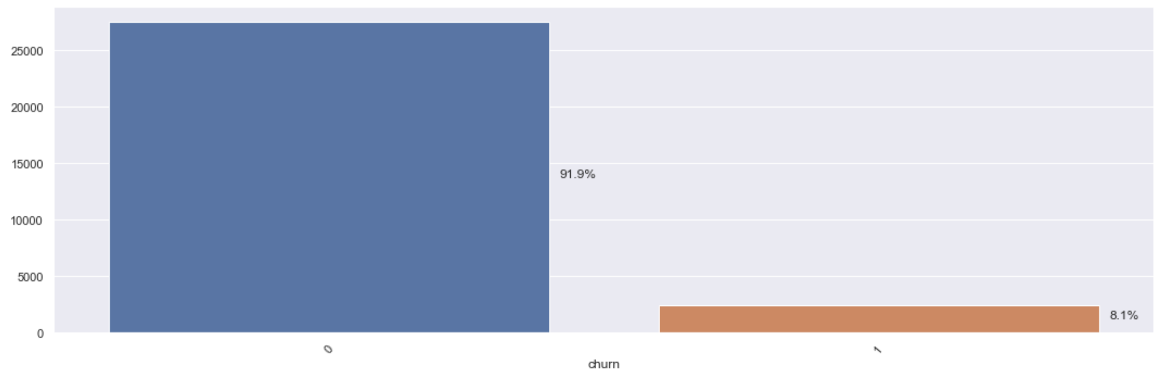
```
c_palette = ['tab:green', 'tab:red']
```

In [53]:

```
def bivariate_analysis(x,
                        hue,
                        df=telecom_data,
                        is_categorical=False,
                        title="",
                        xlabel=None,
                        ylabel=None,
                        rotate_labels=False,
                        bins=25):
    colors_list = ['green', 'red']
    temp = telecom_data[[x, hue]]
    temp = pd.crosstab(temp[x], temp[hue], margins=False)

    # Change this line to plot percentages instead of absolute values
    ax = (temp.div(temp.sum(1), axis=0)).plot(kind='bar',
                                              figsize=(15, 4),
                                              width=0.8,
                                              color=colors_list,
                                              edgecolor=None)
    plt.legend(labels=['Not churned', 'Churned'], fontsize=14)
    plt.title(title, fontsize=16)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.xticks(fontsize=14)
    for spine in plt.gca().spines.values():
        spine.set_visible(False)
    plt.yticks([])

    # Add this loop to add the annotations
    for p in ax.patches:
        width, height = p.get_width(), p.get_height()
        x, y = p.get_xy()
        ax.annotate('{:.0%}'.format(height), (x, y + height + 0.01))
```

In [54]:

```
univariate_analysis('churn', is_categorical=True)
```

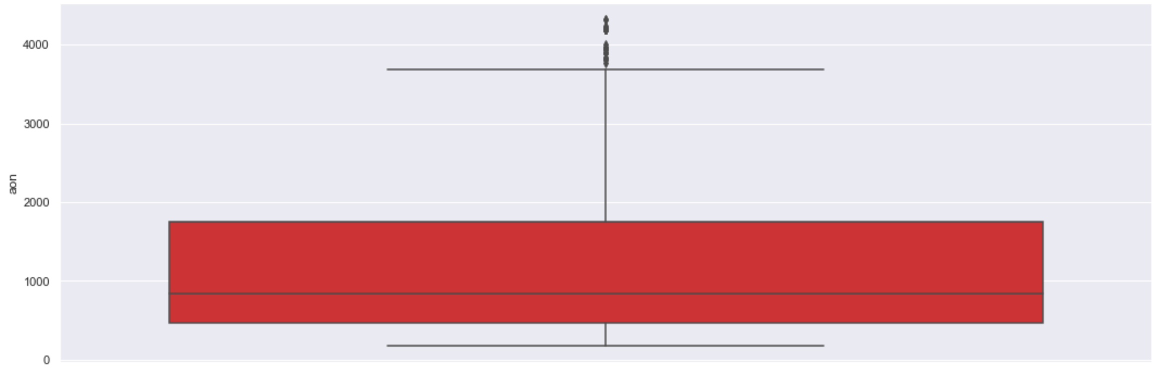


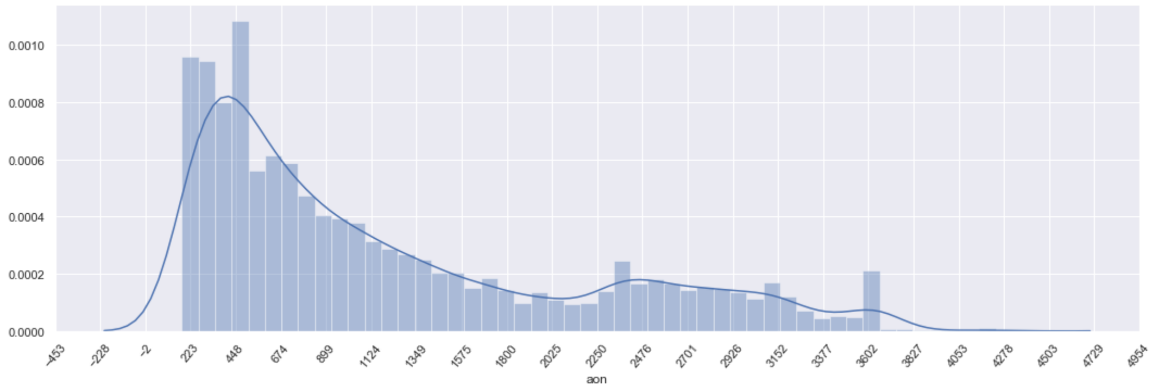
We can see that this data is highly imbalanced as the positive class (churn=1) is very less in number compared to negative class (churn=0). We will use class imbalance techniques like SMOTE to balance the data once we start with the model creation.

In [55]:

```
univariate_analysis('aon')
```

```
count    29953.0000
mean      1209.2806
std        957.4494
min        180.0000
25%        460.0000
50%        846.0000
75%       1756.0000
max       4321.0000
Name: aon, dtype: float64
```





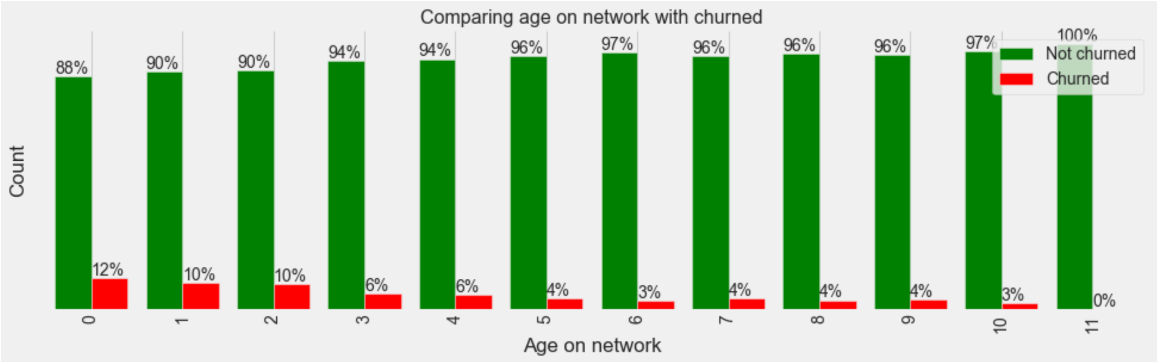
We can see that the all the customers are with the company for than a half year. We can create a column which will have the age on the network in years.

In [56]:

```
telecom_data['aon_year'] = telecom_data['aon'].apply(lambda x: x//365)
telecom_data['aon_year'] = telecom_data['aon_year'].astype('category')
telecom_data = telecom_data.drop('aon',axis=1)
```

In [57]:

```
bivariate_analysis('aon_year',
                   'churn',
                   title="Comparing age on network with churned",
                   xlabel='Age on network',
                   ylabel='Count')
```



We can see as the age on network increases the churn rate decreases. Most churn happens in the starting 2 years.

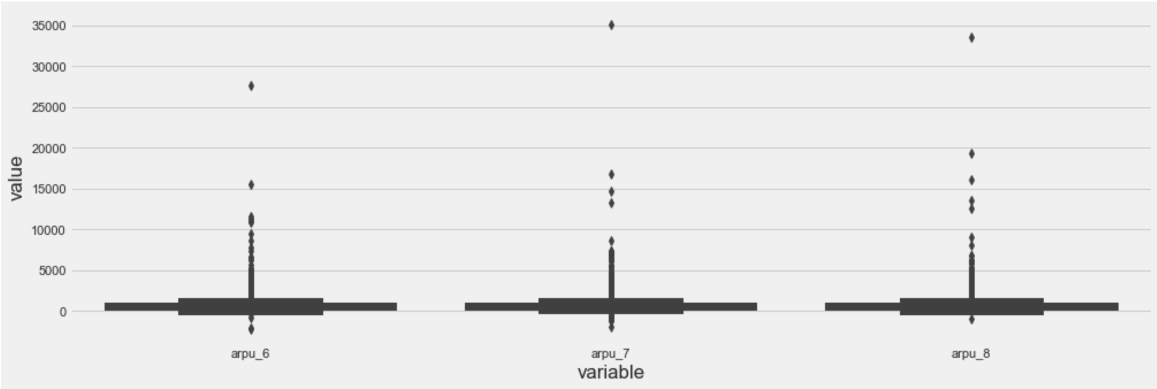
Now, we will various features averaged for each (Jun, July and Aug)

In [58]:

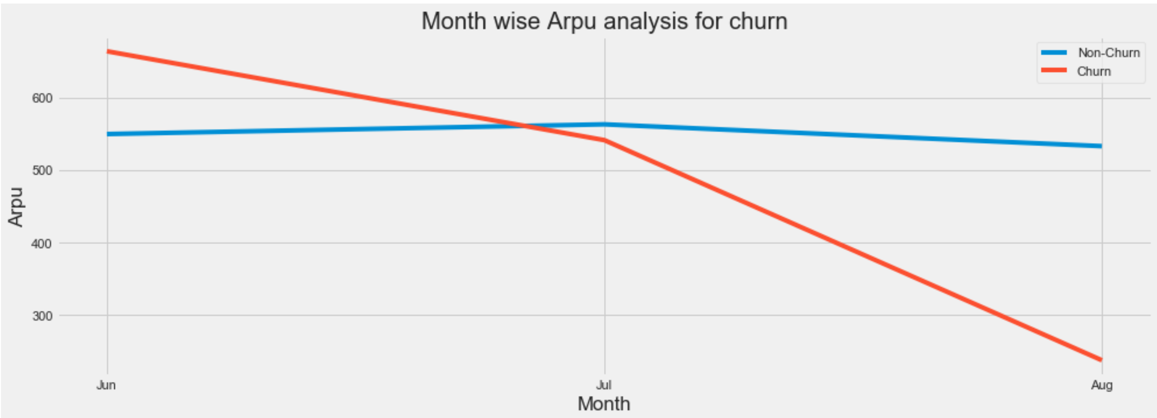
```
def month_wise_analysis(columns, title, xlabel, ylabel, df=telecom_data):
    plot1 = plt.figure(1)
    sns.boxplot(x="variable", y="value", data=pd.melt(df[columns]))
    plt.show()
    means = df.groupby('churn')[columns].mean()
    means.rename(columns={means.columns[0]: "Jun", means.columns[1]: "Jul", mean
s.columns[2]: "Aug"}, inplace=True)
    print(means)
    plot2 = plt.figure(1)
    plt.plot(means.T)
    plt.title(title)
    plt.xlabel(xlabel)
    plt.ylabel(ylabel)
    plt.legend(['Non-Churn', 'Churn'])
    plt.show()
```

In [59]:

```
month_wise_analysis(['arpu_6', 'arpu_7', 'arpu_8'],
                    'Month wise Arpu analysis for churn', 'Month', 'Arpu')
```



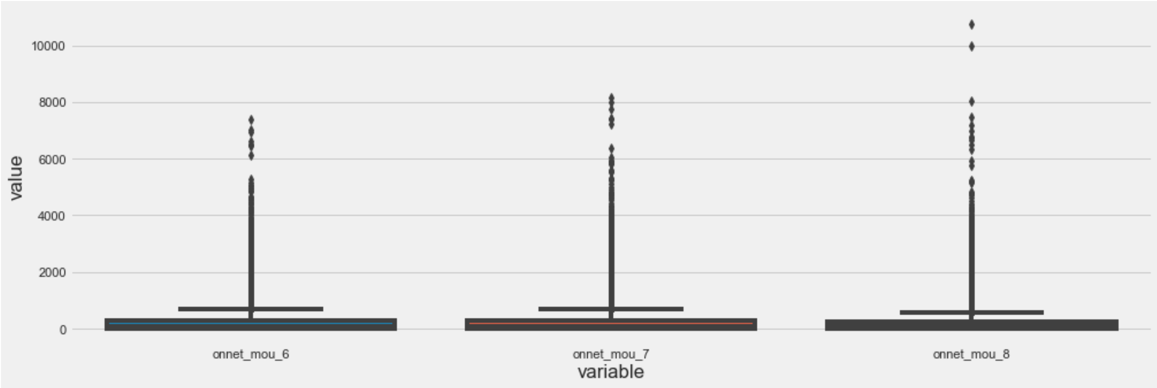

```
           Jun       Jul       Aug
churn
0      549.5470  562.9300  532.8697
1      663.7094  541.1461  237.6555
```



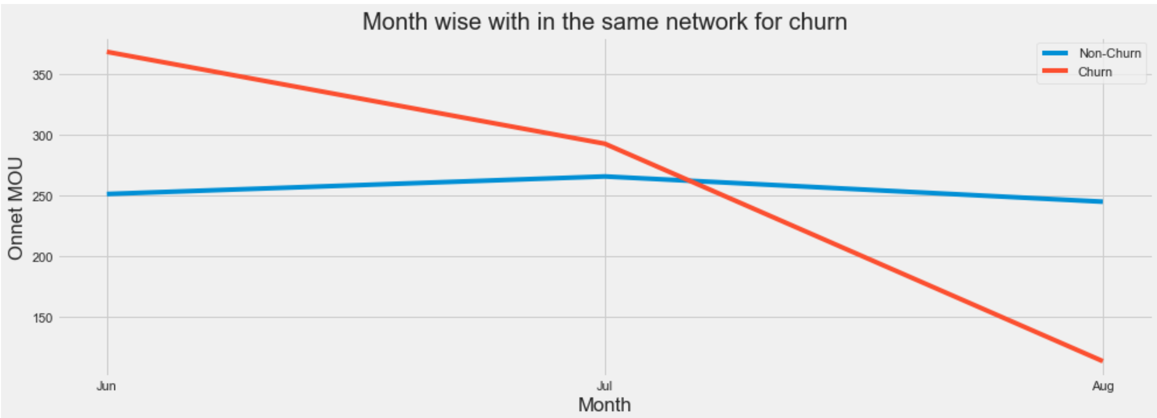
We can see that the average revenue decreases significantly for the churned customer from Jun to Aug. In case of non-churned customers it is almost constant. We can see there are outliers. We will treat them in outliers treatment section.

In [60]:

```
month_wise_analysis(['onnet_mou_6', 'onnet_mou_7', 'onnet_mou_8'],
                    'Month wise with in the same network for churn', 'Month', 'O
nnet MOU')
```



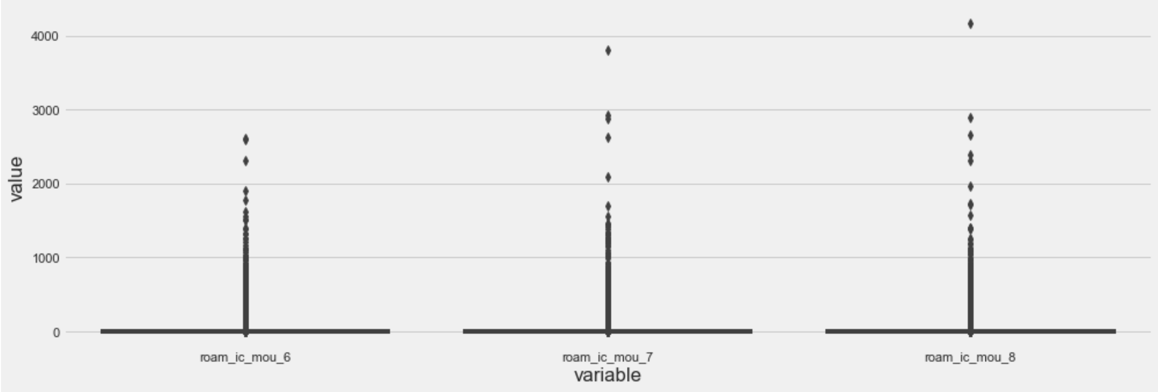

```
            Jun       Jul       Aug
churn
0      251.3741  265.8597  245.0309
1      368.6594  292.8466  113.4780
```



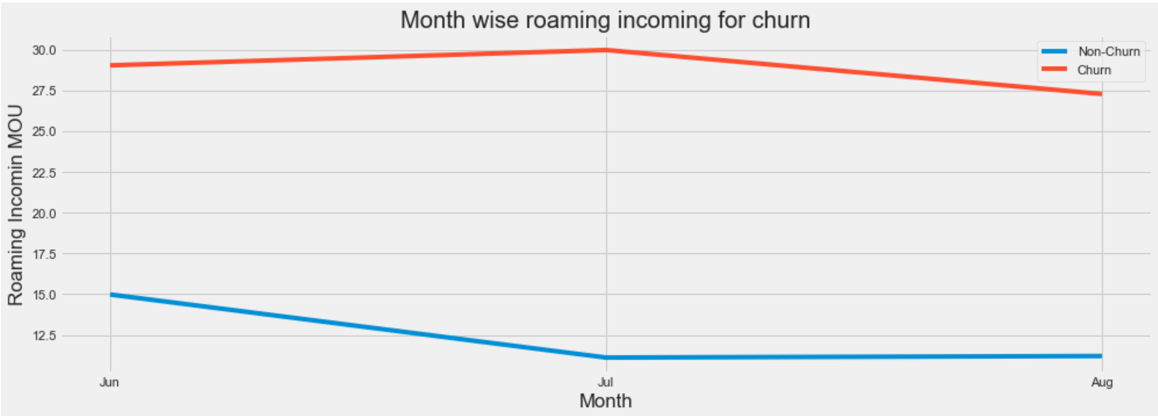
We can see that the average revenue decreases significantly for the churned customer from Jun to Aug. In case of non-churned customers it is almost constant. We can see there are outliers. We will treat them in outliers treatment section.

In [61]:

```
month_wise_analysis(['roam_ic_mou_6', 'roam_ic_mou_7', 'roam_ic_mou_8'],
                    'Month wise roaming incoming for churn', 'Month', 'Roaming I
ncomin MOU')
```



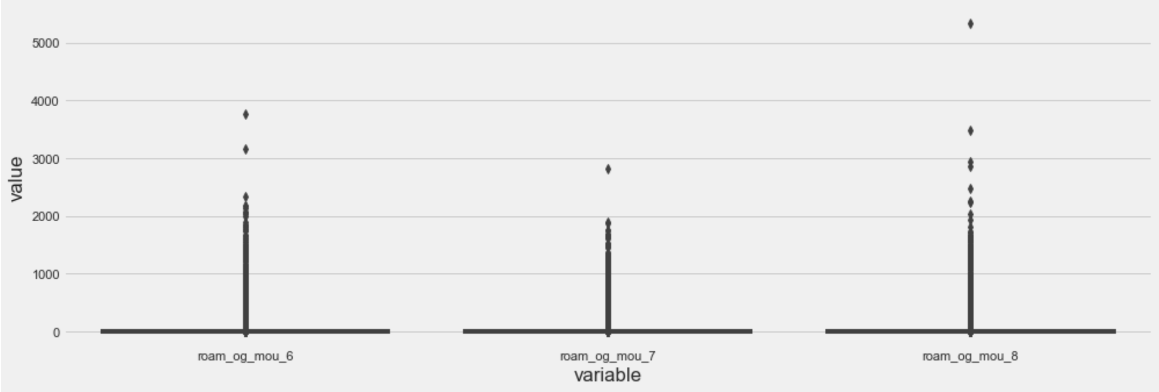

```
           Jun      Jul      Aug
churn
0      14.9823 11.1138 11.2065
1      29.0377 29.9788 27.2821
```



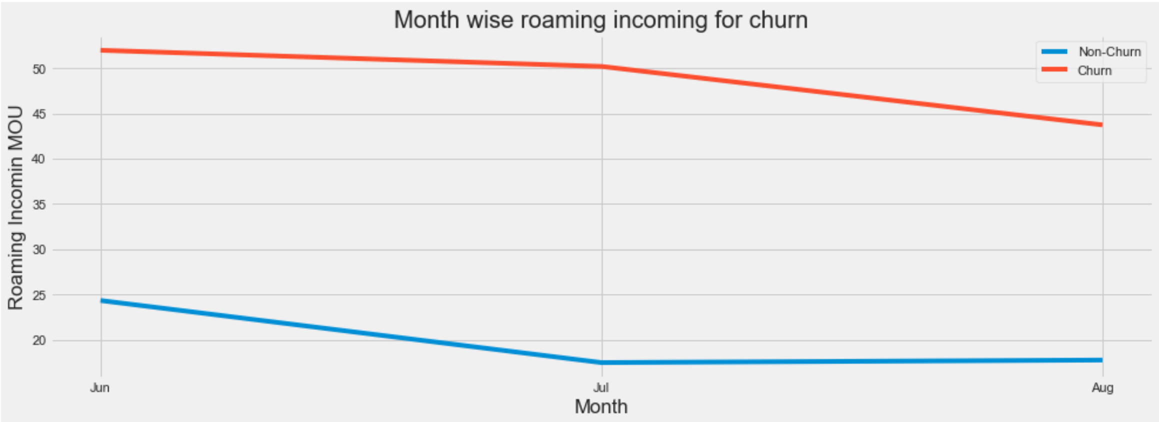
We can see that the customers who churned had high roaming incoming usage. Hence, a better pack or deal on incoming roaming can be given to stop the churn.

In [62]:

```
month_wise_analysis(['roam_og_mou_6', 'roam_og_mou_7', 'roam_og_mou_8'],
                    'Month wise roaming incoming for churn', 'Month', 'Roaming I
ncomin MOU')
```



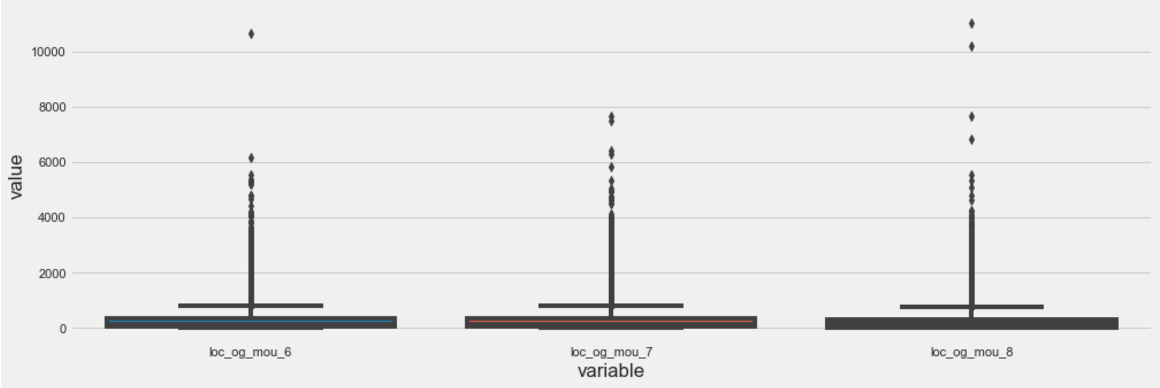

```
         Jun     Jul     Aug
churn
0      24.3533 17.5008 17.7816
1      51.9642 50.1792 43.7300
```



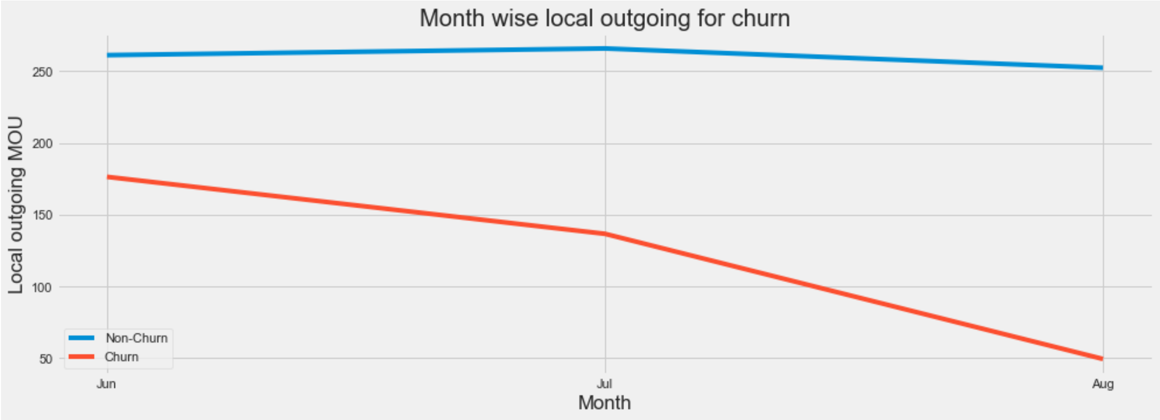
We can see that the customers who churned had high roaming outgoing usage. Hence, a better pack or deal on outgoing roaming can be given to stop the churn.

In [63]:

```
month_wise_analysis(['loc_og_mou_6', 'loc_og_mou_7', 'loc_og_mou_8'],
                    'Month wise local outgoing for churn', 'Month', 'Local outgo
ing MOU')
```



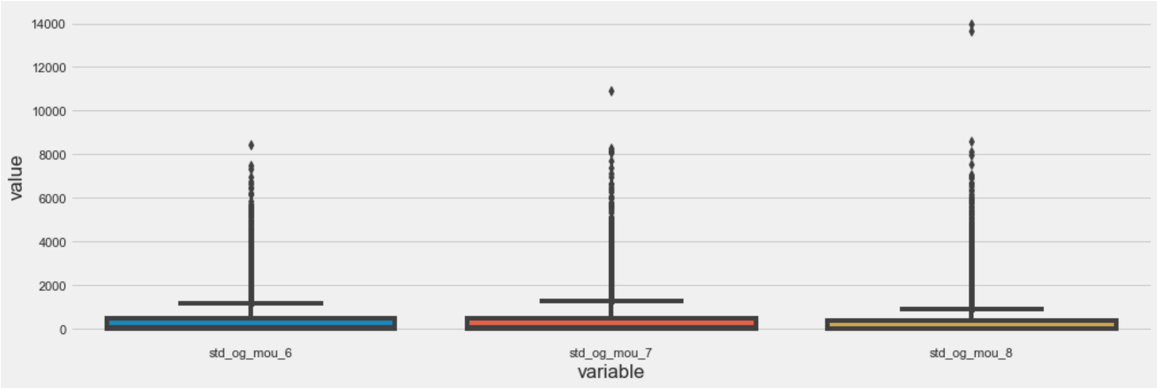

```
          Jun       Jul       Aug
churn
0      261.1244  265.7696  252.3358
1      176.3724  136.6957   49.5382
```



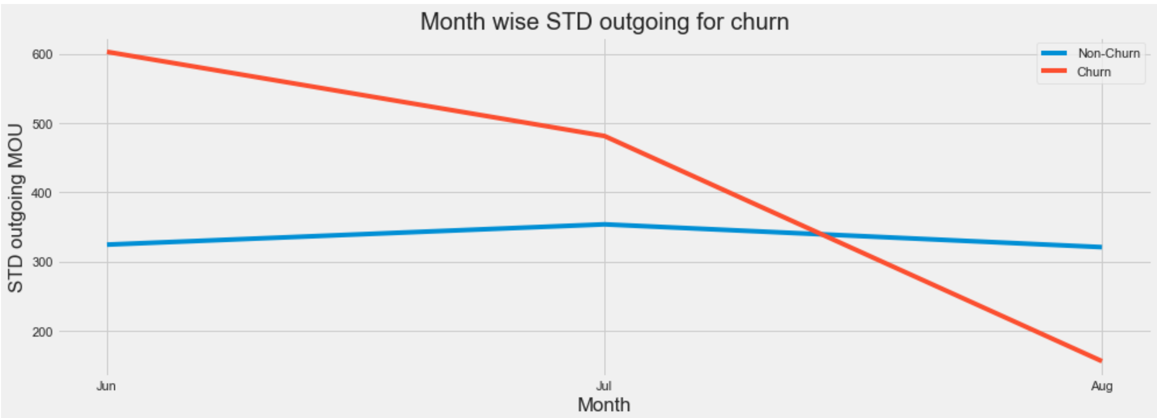
We can see the local outgoing usage for the churn customer is decreasing as the time increases. The company can provide pack etc to encourage more outgoing calls.

In [64]:

```
month_wise_analysis(['std_og_mou_6', 'std_og_mou_7', 'std_og_mou_8'],
                    'Month wise STD outgoing for churn', 'Month', 'STD outgoing
 MOU')
```



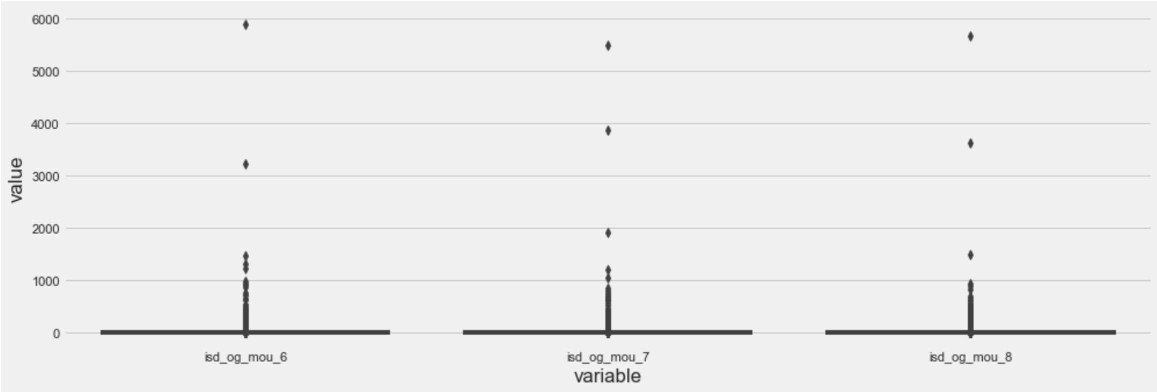

```
          Jun      Jul      Aug
churn
0      324.5269 353.8073 320.8792
1      603.0072 481.4109 156.1696
```



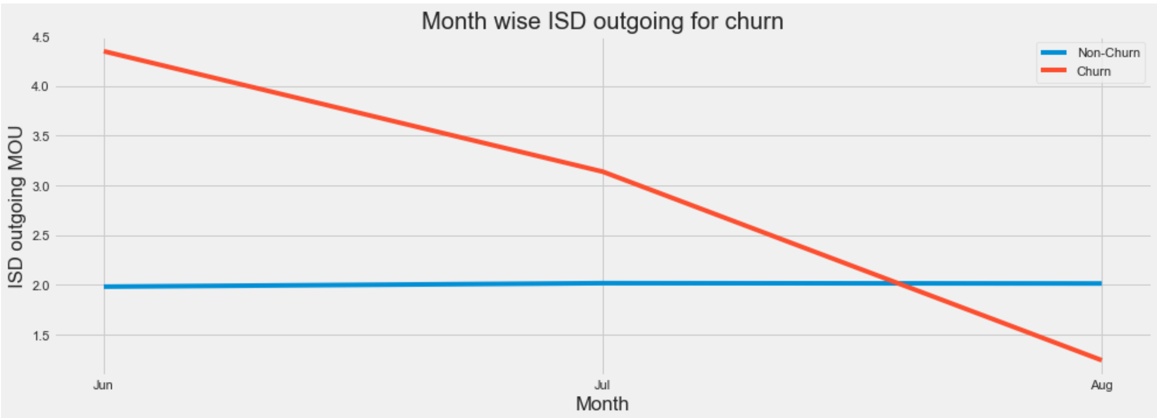
We can see the std outgoing usage for the churn customer is decreasing as the time increases. The company can provide pack etc to encourage more std outgoing calls.

In [65]:

```
month_wise_analysis(['isd_og_mou_6', 'isd_og_mou_7', 'isd_og_mou_8'],
                    'Month wise ISD outgoing for churn', 'Month', 'ISD outgoing
 MOU')
```



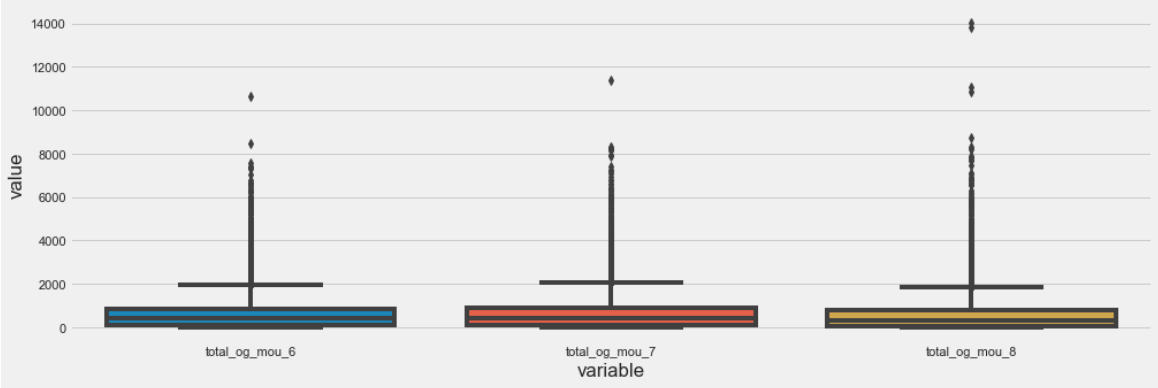

```
         Jun     Jul     Aug
churn
0       1.9827  2.0180  2.0148
1       4.3501  3.1390  1.2425
```



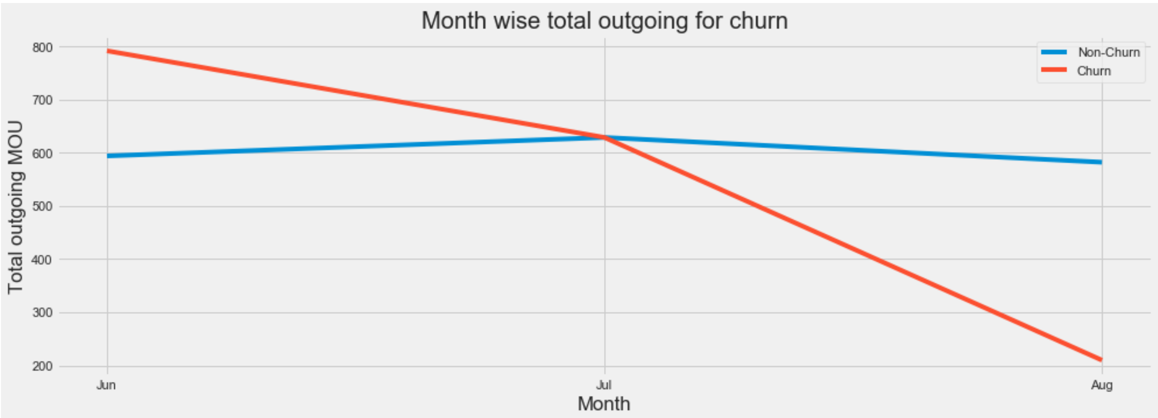
We can see the ISD outgoing usage for the churn customer is decreasing as the time increases. The company can provide pack etc to encourage more ISD outgoing calls.

In [66]:

```
month_wise_analysis(['total_og_mou_6', 'total_og_mou_7', 'total_og_mou_8'],
                    'Month wise total outgoing for churn', 'Month', 'Total outgo
ing MOU')
```



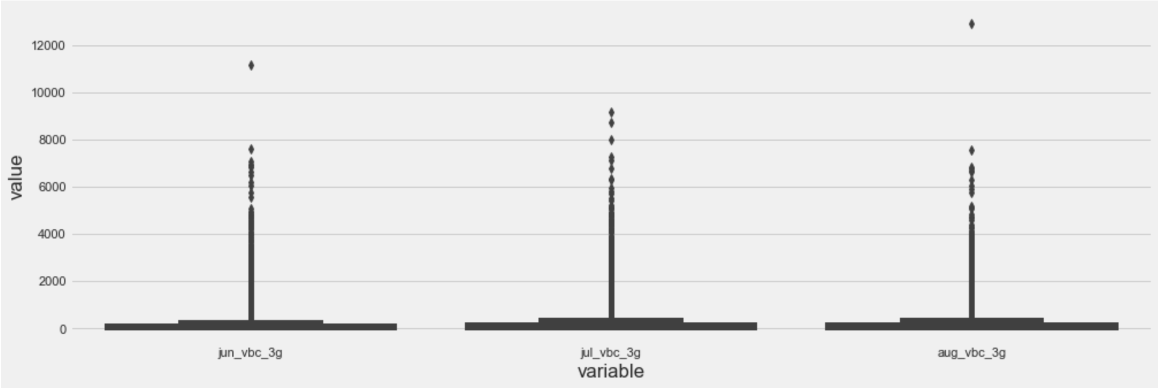

```
          Jun      Jul      Aug
churn
0      593.9961 628.7205 582.1774
1      791.7370 628.7653 209.7945
```



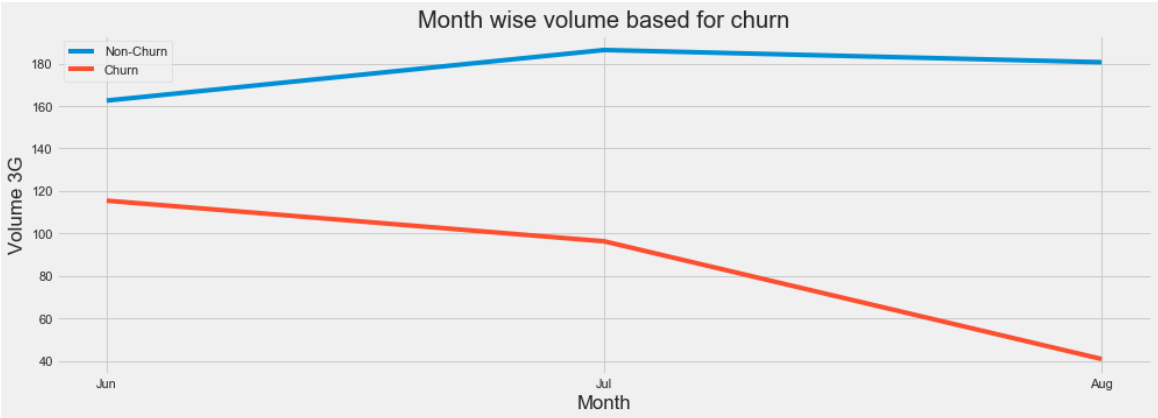
We can see that the total outgoing mou decreases significantly for the churned customer from Jun to Aug. In case of non-churned customers it is almost constant. We can see there are outliers. We will treat them in outliers treatment section.

In [67]:

```
month_wise_analysis(['jun_vbc_3g', 'jul_vbc_3g', 'aug_vbc_3g'],
                    'Month wise volume based for churn', 'Month', 'Volume 3G')
```




```
          Jun      Jul      Aug
churn
0      162.5573 186.3705 180.6226
1      115.4618  96.3407  40.9409
```

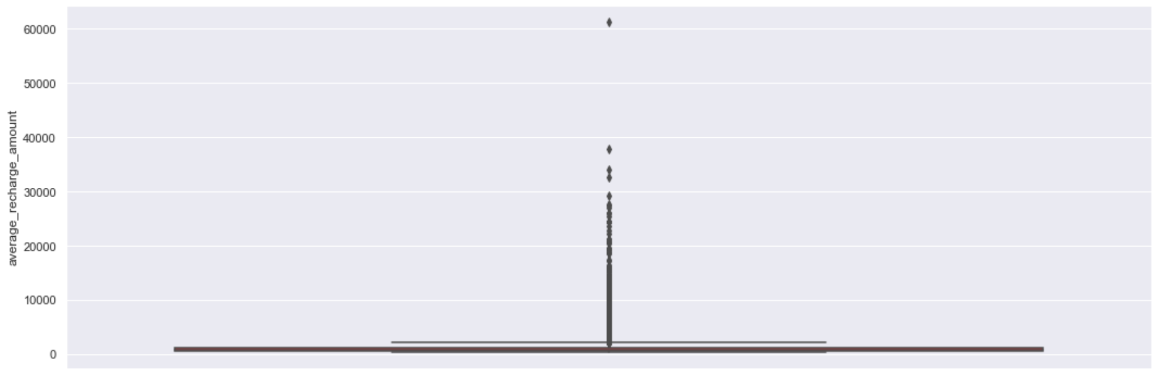


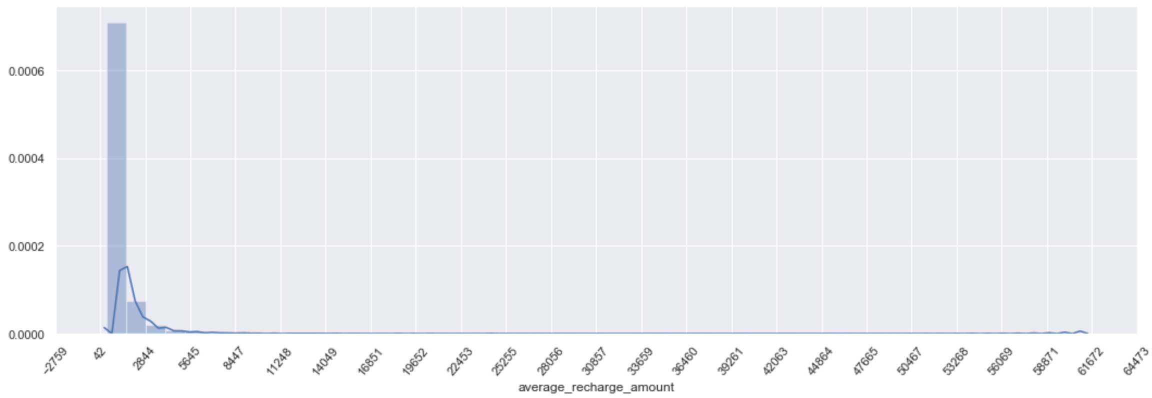
We can see that the 3g volume mou decreases significantly for the churned customer from Jun to Aug. In case of non-churned customers it is almost constant. We can see there are outliers. We will treat them in outliers treatment section.

In [68]:

```
univariate_analysis('average_recharge_amount')
```

```
count    29953.0000
mean      1153.7017
std       1359.5336
min        478.5000
25%        604.0000
50%        800.5000
75%       1209.0000
max      61236.0000
Name: average_recharge_amount, dtype: float64
```

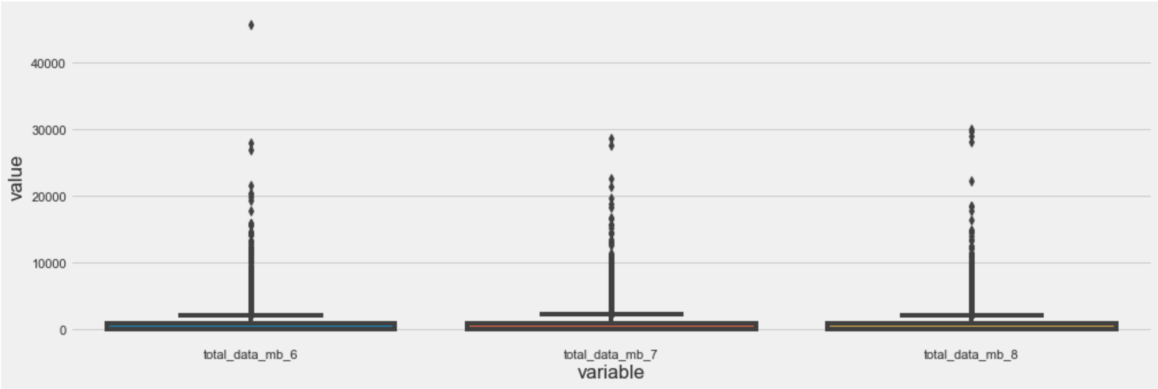




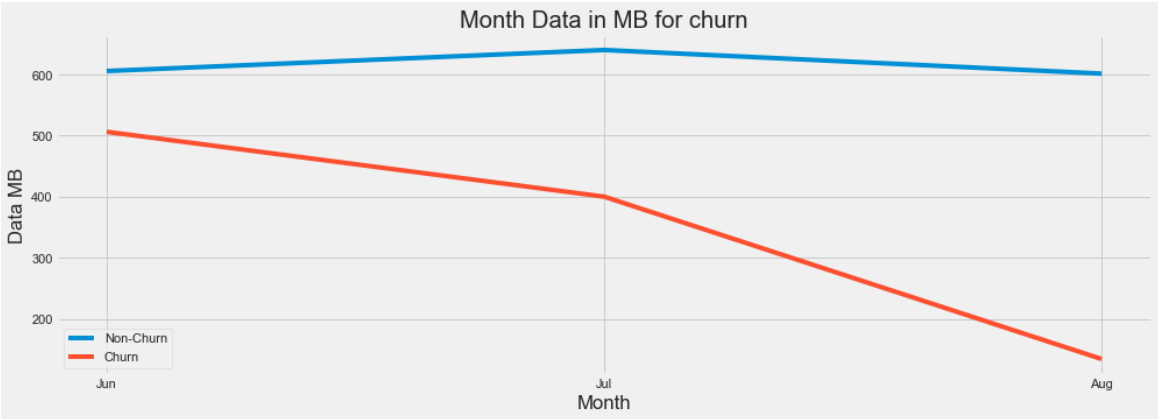
We can see that there are outliers here which makes sense as there will be some customers who pay huge money for recharge.

In [69]:

```
month_wise_analysis(['total_data_mb_6', 'total_data_mb_7', 'total_data_mb_8'],
                    'Month Data in MB for churn', 'Month', 'Data MB')
```



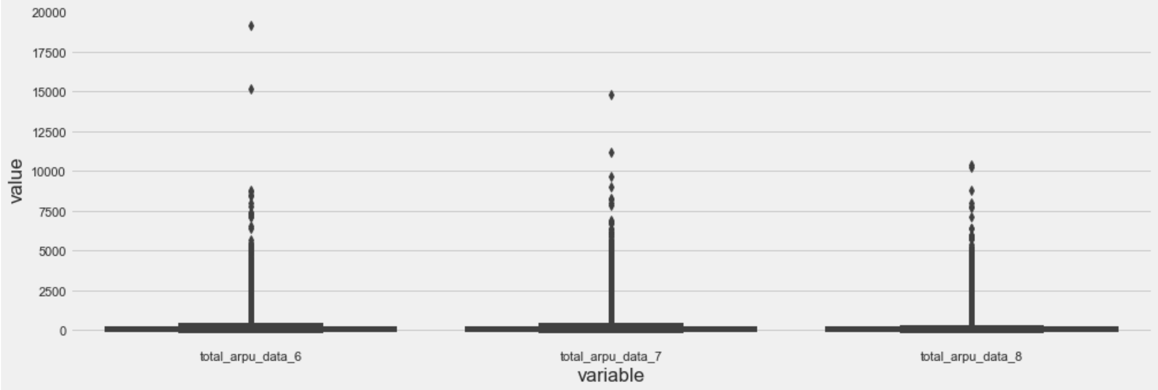

```
          Jun       Jul       Aug
churn
0      605.7925  640.2835  601.5430
1      506.2850  400.0708  134.1448
```



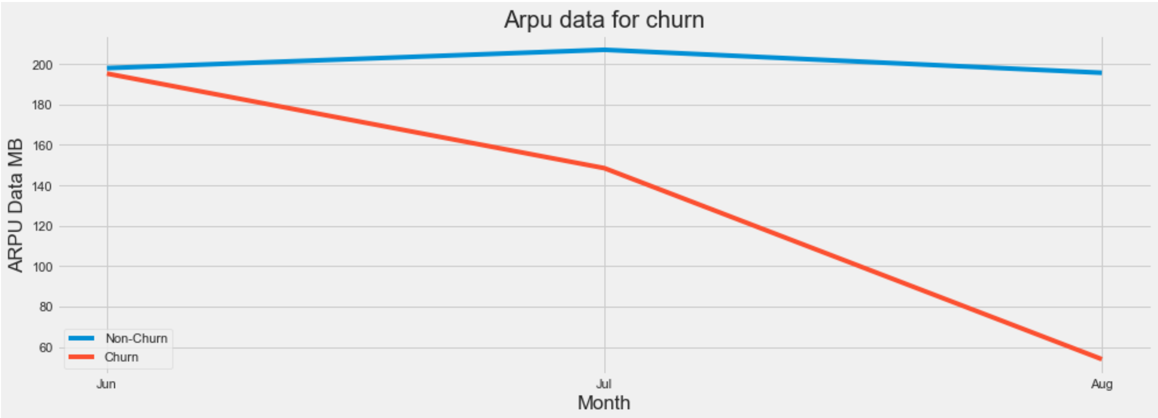
We can see that the total outgoing mou decreases significantly for the churned customer from Jun to Aug. In case of non-churned customers it is almost constant. We can see there are outliers. We will treat them in outliers treatment section.

In [70]:

```
month_wise_analysis(['total_arpu_data_6', 'total_arpu_data_7', 'total_arpu_data_
8'],
                    'Arpu data for churn', 'Month', 'ARPU Data MB')
```




```
          Jun      Jul      Aug
churn
0      197.9403 206.9873 195.6237
1      195.2407 148.4653  53.9988
```



We can see that the data arpu decreases significantly for the churned customer from Jun to Aug. In case of non-churned customers it is almost constant. We can see there are outliers. We will treat them in outliers treatment section.

Outliers treatment

We saw that there are many features which have outliers. However, we can remove all of them as it may impact the model accuracy. We will remove the rows which have more than 99 percentile for following features:

1. arpu
2. average_recharge_amount
3. total_data_mb
4. total_og_mou
5. total_arpu_data

In [71]:

```
outlier_features = ['arpu_6','arpu_7','average_recharge_amount','total_data_mb_
6',
                    'total_data_mb_7','total_og_mou_6','total_og_mou_7','total_a
rpu_data_6',
                    'total_arpu_data_7']

for column in outlier_features:
    upper = telecom_data[column].quantile(.99)
    telecom_data = telecom_data[telecom_data[column] < upper]
```

In [72]:

```
telecom_data.shape
```

Out[72]:

```
(27359, 124)
```

In [73]:

```
numerical_columns = telecom_data.select_dtypes(
    include=['int64', 'float64']).columns

columns_to_encode = telecom_data.select_dtypes(
    include=['category', 'object']).columns.tolist()
columns_to_encode.remove('churn')
```

In [74]:

```
# Creating the dummy variables and deleting the unknown class (-1) which we subs
tituted for `NAN`
for c in columns_to_encode:
    dummy_pd = pd.get_dummies(telecom_data[c],
                              prefix=c)
    if((c + "-1") in dummy_pd.columns):
        dummy_pd = dummy_pd.drop((c+"-1"), axis=1)
    else:
        dummy_pd = dummy_pd.drop(dummy_pd.columns[0], axis=1)

    telecom_data = pd.concat([telecom_data, dummy_pd], axis=1)
```

In [75]:

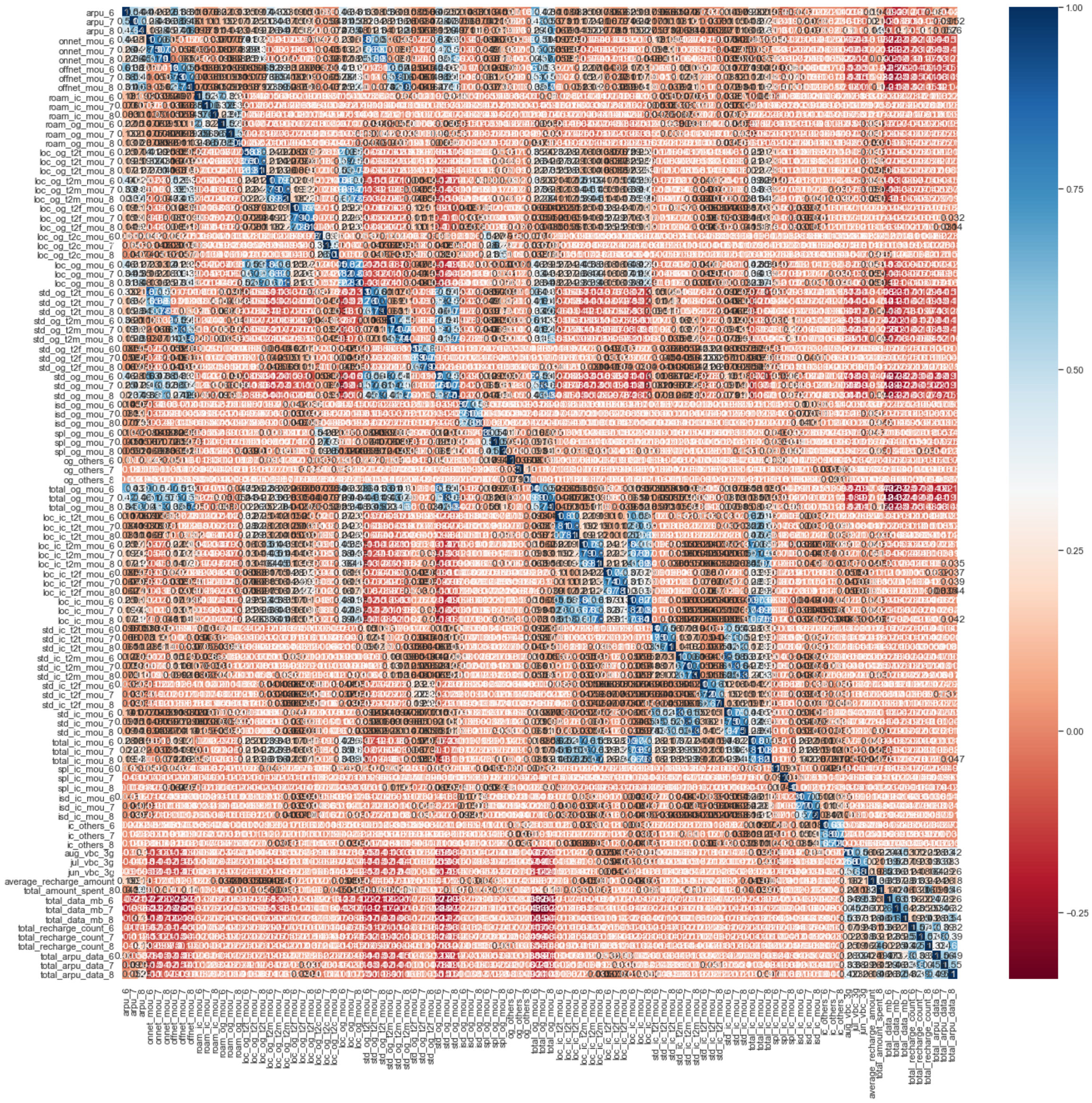
```
# Deleting the original columns
telecom_data = telecom_data.drop(columns_to_encode, axis=1)
```

In [76]:

```
def heat_map(data):
    corr = data.corr()
    sns.set(rc={'figure.figsize': (20, 20)})
    plt.tight_layout()
    ax = sns.heatmap(corr, xticklabels=corr.columns, yticklabels=corr.columns, c
map='RdBu',annot=True)
    bottom, top = ax.get_ylim()
    ax.set_ylim(bottom + 0.5, top - 0.5)
```

In [77]:

```
heat_map(telecom_data[numerical_columns])
```



In [78]:

```
# Reference https://chrisalbon.com/machine_learning/feature_selection/drop_highl
y_correlated_features/

# List of correlated columns

corr_matrix = telecom_data[numerical_columns].corr().abs()

# Select upper triangle of correlation matrix
upper = corr_matrix.where(np.triu(np.ones(corr_matrix.shape), k=1).astype(np.boo
l))

# Find index of feature columns with correlation greater than 0.80
to_drop = [column for column in upper.columns if any(upper[column] > 0.80)]
print('found ',len(to_drop),' highly correlated features.')
print(to_drop)
```

```
found  26  highly correlated features.
['loc_og_t2t_mou_7', 'loc_og_t2t_mou_8', 'loc_og_t2m_mou_8', 'loc_og
_mou_6', 'loc_og_mou_7', 'loc_og_mou_8', 'std_og_t2t_mou_6', 'std_og
_t2t_mou_7', 'std_og_t2t_mou_8', 'std_og_t2m_mou_7', 'std_og_t2m_mou
_8', 'total_og_mou_6', 'total_og_mou_7', 'total_og_mou_8', 'loc_ic_t
2t_mou_7', 'loc_ic_t2t_mou_8', 'loc_ic_t2m_mou_8', 'loc_ic_mou_6',
'loc_ic_mou_7', 'loc_ic_mou_8', 'std_ic_mou_6', 'std_ic_mou_7', 'std
_ic_mou_8', 'total_ic_mou_6', 'total_ic_mou_7', 'total_ic_mou_8']
```

We found 26 features which are around 80% correlated. We will not be deleting them as we will use PCA for dimension reduction.

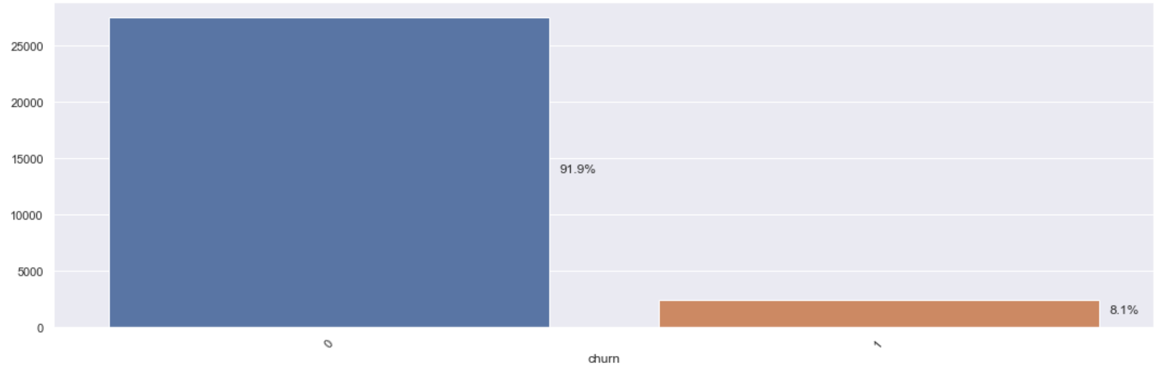
In [79]:

```
## Saving the cleaned data
from pathlib import Path
def save_clean_data():
    clean_file = Path("clean_data.csv")
    if clean_file.is_file():
        telecom_data = pd.read_csv('clean_data.csv')
    else:
        telecom_data.to_csv('clean_data.csv',header=True)
```

Class imbalance

In [80]:

```
univariate_analysis('churn', is_categorical=True)
```



We can see that churn class is highly imbalanced. We will `SMOTE` to over sample the less frequent class.

In [81]:

```
gc.collect()
random_seed = 101
```

In [82]:

```
y = telecom_data.pop('churn')
X = telecom_data
```

In [83]:

```
from sklearn.model_selection import train_test_split

X_train, X_test, y_train, y_test = train_test_split(X,
                                                    y,
                                                    train_size=0.7,
                                                    test_size=0.3,
                                                    random_state=random_seed)
```

In [84]:

```
# We will scale the data as PCA is senstive to the scale.
from sklearn.preprocessing import StandardScaler
scaler = StandardScaler()

X_train = scaler.fit_transform(X_train)
X_test = scaler.transform(X_test)
```

In [85]:

```
from imblearn.over_sampling import SMOTE
smote = SMOTE(random_state=random_seed)

X_train_resampled, y_train_resampled = smote.fit_sample(X_train, y_train)

from collections import Counter
print("Before SMOTE:", Counter(y_train))
print("After SMOTE:", Counter(y_train_resampled))
```

```
Before SMOTE: Counter({0: 17676, 1: 1475})
After SMOTE: Counter({0: 17676, 1: 17676})
```

PCA

In [86]:

```
from sklearn.decomposition import PCA
```

In [87]:

```
pca = PCA(random_state=random_seed)
```

In [88]:

```
pca.fit(X_train_resampled)
```

Out[88]:

```
PCA(random_state=101)
```

In [89]:

```
pca.explained_variance_ratio_
```

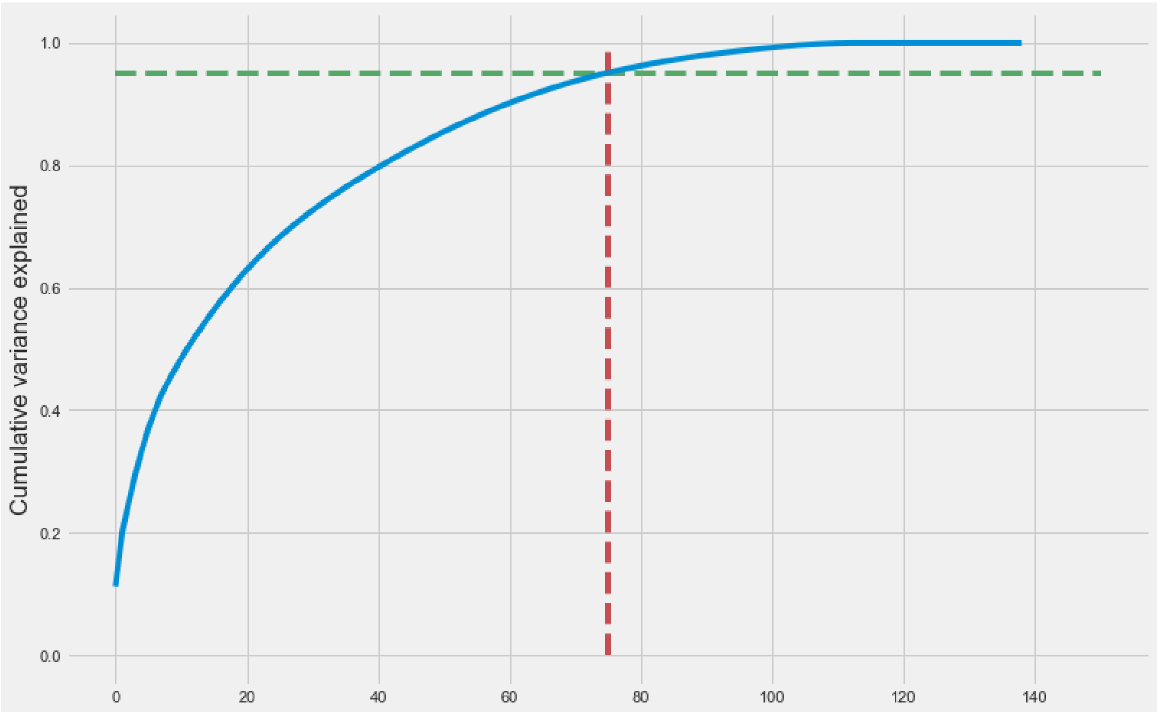
Out[89]:

```
array([1.11910502e-01, 8.74800371e-02, 4.88635501e-02, 4.63694782e-0
2,
       3.96015692e-02, 3.58777514e-02, 2.87417954e-02, 2.70271915e-0
2,
       2.07202659e-02, 1.94569168e-02, 1.84594739e-02, 1.74101414e-0
2,
       1.62928275e-02, 1.55942474e-02, 1.55418306e-02, 1.48801729e-0
2,
       1.42206397e-02, 1.31889784e-02, 1.31050408e-02, 1.28863971e-0
2,
       1.19499667e-02, 1.13089562e-02, 1.10938199e-02, 1.06891214e-0
2,
       1.03716058e-02, 9.74637682e-03, 9.37239913e-03, 8.97508338e-0
3,
       8.67442360e-03, 8.41381745e-03, 8.22867812e-03, 7.89034758e-0
3,
       7.60853741e-03, 7.46040907e-03, 7.31440549e-03, 7.12919900e-0
3,
       6.92812780e-03, 6.62033690e-03, 6.52535650e-03, 6.41449670e-0
3,
       6.38121834e-03, 6.34677515e-03, 6.22914779e-03, 6.03223759e-0
3,
       5.91694324e-03, 5.88645033e-03, 5.74182498e-03, 5.64272437e-0
3,
       5.54707186e-03, 5.48461710e-03, 5.35732837e-03, 5.22124892e-0
3,
       5.05857318e-03, 4.91217596e-03, 4.84341808e-03, 4.72689760e-0
3,
       4.67485882e-03, 4.54808588e-03, 4.44771041e-03, 4.35000859e-0
3,
       4.13806060e-03, 3.96332281e-03, 3.92872293e-03, 3.89982340e-0
3,
       3.74208191e-03, 3.65675938e-03, 3.51590363e-03, 3.42995166e-0
3,
       3.25087963e-03, 3.20415069e-03, 3.08718675e-03, 2.96369644e-0
3,
       2.91818874e-03, 2.85643842e-03, 2.62435275e-03, 2.45331324e-0
3,
       2.41155398e-03, 2.32465403e-03, 2.20557260e-03, 2.19290351e-0
3,
       2.14607297e-03, 2.12783010e-03, 1.94545162e-03, 1.90440515e-0
3,
       1.88644651e-03, 1.74960180e-03, 1.71633177e-03, 1.68253636e-0
3,
       1.61702582e-03, 1.56542035e-03, 1.45848865e-03, 1.38832983e-0
3,
       1.36790022e-03, 1.31862022e-03, 1.28321093e-03, 1.24665935e-0
3,
       1.22162596e-03, 1.20991066e-03, 1.09599176e-03, 1.08600508e-0
3,
       1.04987485e-03, 9.53451912e-04, 9.32206775e-04, 8.79688134e-0
4,
       8.47949783e-04, 7.29731284e-04, 6.80791461e-04, 5.89562995e-0
4,
       5.40045759e-04, 4.61139775e-04, 4.05729275e-04, 3.20514652e-0
4,
       8.25534308e-05, 2.95179031e-05, 1.69667620e-05, 8.14566683e-0
7,
       3.02219333e-07, 1.86876427e-07, 8.49970959e-12, 4.04983195e-1
```

```
2,
        3.42233610e-12, 3.24787934e-12, 2.00447145e-12, 1.67271766e-1
2,
        1.63068113e-12, 1.18561754e-12, 1.07523982e-12, 1.04163716e-1
2,
        7.89468893e-13, 7.53113224e-13, 6.87758957e-13, 4.79765155e-1
3,
        3.98507833e-13, 2.94673244e-13, 2.22758622e-13, 1.73372757e-1
3,
        1.69534732e-29, 2.20704551e-32, 5.93184322e-34])
```

In [90]:

```
# Making a scree plot for the explained variance
var_cumu = np.cumsum(pca.explained_variance_ratio_)
fig = plt.figure(figsize=[12,8])
plt.vlines(x=75, ymax=1, ymin=0, colors="r", linestyles="--")
plt.hlines(y=0.95, xmax=150, xmin=0, colors="g", linestyles="--")
plt.plot(var_cumu)
plt.ylabel("Cumulative variance explained")
plt.show()
```



From the graph we can see that 95% variance of the data. From now onwards, we will consider 75 components for the analysis

In [91]:

```
from sklearn.decomposition import IncrementalPCA
pca_final = IncrementalPCA(n_components=75)
X_train_pca = pca_final.fit_transform(X_train_resampled)
X_train_pca.shape
```

Out[91]:

```
(35352, 75)
```

In [92]:

```
# Getting the minimum and maximum value from the correlation matrix.
# Reference https://stackoverflow.com/questions/29394377/minimum-of-numpy-array-ignoring-diagonal

corr_matrix = np.corrcoef(X_train_pca.transpose())
mask = np.ones(corr_matrix.shape, dtype=bool)
np.fill_diagonal(mask, 0)
max_value = corr_matrix[mask].max()
min_value = corr_matrix[mask].min()

print('Max: ', max_value, ' , Min: ', min_value )
```

```
Max:  0.018890464513361  , Min:  -0.025974033119940152
```

We can say that after PCA, there is no multi collinearity in the data set.

Applying the transformation on the test set

In [93]:

```
X_test_pca = pca_final.transform(X_test)
X_test_pca.shape
```

Out[93]:

```
(8208, 75)
```

Model creation

With PCA

We will be creating following model:

1. DummyClassifier (Base Model)
2. Logistic Regression
3. Decision Tree
4. Random Forest
5. Boosting models

In [94]:

```
from sklearn.dummy import DummyClassifier
from sklearn.linear_model import LogisticRegression
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import RandomForestClassifier, AdaBoostClassifier, Gradien
tBoostingClassifier
from sklearn.metrics import accuracy_score
from sklearn.metrics import precision_score
from sklearn.metrics import recall_score
from sklearn.metrics import f1_score
from sklearn.metrics import roc_auc_score
from sklearn.metrics import roc_curve
from sklearn.model_selection import GridSearchCV
from xgboost import XGBClassifier
from time import time
```

In [95]:

```
def draw_roc( actual, probs ):
    fpr, tpr, thresholds = roc_curve( actual, probs,
                                      drop_intermediate = False )
    auc_score = roc_auc_score( actual, probs )
    plt.figure(figsize=(6, 6))
    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
```

In [96]:

```
def train_model(classifier_name, classifier, X_train_model, y_train_model,
                X_test_model, y_test_model):
    start = time()
    classifier.fit(X_train_model, y_train_model)
    df_train = predict_and_get_metrics('train', classifier, X_train_model,
                                       y_train_model)
    train_time = time() - start
    start = time()
    df_test = predict_and_get_metrics('test', classifier, X_test_model,
                                      y_test_model)
    df = pd.concat([df_train, df_test], axis=1)
    df.insert(0, "name", [classifier_name], True)
    score_time = time()-start
    print("ModelName: {:<15} | time (training/test) = {:,.3f}s/{:,.3f}s".format(
classifier_name, train_time, score_time))
    return df
```

In [97]:

```
def predict_and_get_metrics(score_type, classifier, X_model, y_model):
    y_pred = classifier.predict(X_model)
    y_pred_prob = classifier.predict_proba(X_model)[:, 1]
    if(score_type=='test'):
        draw_roc(y_model, y_pred_prob)
    accuracy = accuracy_score(y_model, y_pred)
    precision = precision_score(y_model, y_pred)
    recall = recall_score(y_model, y_pred)
    f1 = f1_score(y_model, y_pred)
    auc = roc_auc_score(y_model, y_pred_prob)

    metrics_dict = {}
    metrics_dict[score_type + '_accuracy'] = accuracy
    metrics_dict[score_type + '_precision'] = precision
    metrics_dict[score_type + '_recall'] = recall
    metrics_dict[score_type + '_f1'] = f1
    metrics_dict[score_type + '_auc'] = auc
    records = []
    records.append(metrics_dict)
    return pd.DataFrame.from_records(records)
```

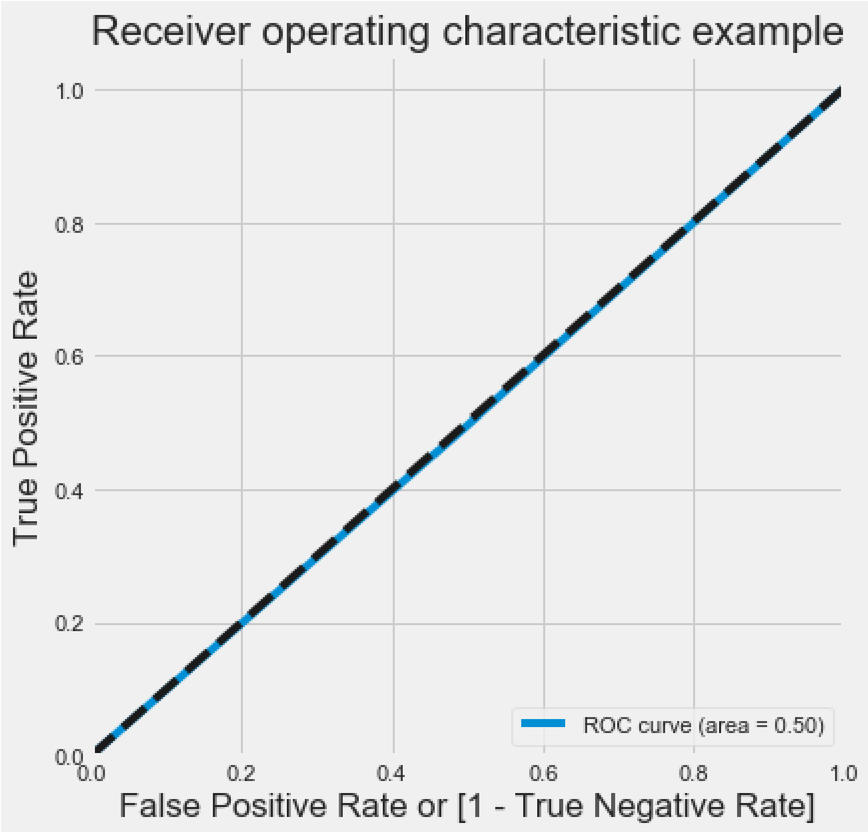
In [98]:

```
def hyperparameter_tuning(classifier,
                          params_grid,
                          n_folds=5,
                          scoring='recall',
                          X_train=X_train_pca):
    grid_search = GridSearchCV(estimator=classifier,
                               param_grid=params_grid,
                               cv=n_folds,
                               verbose=1,
                               n_jobs=-1,
                               scoring=scoring)
    grid_search.fit(X_train, y_train_resampled)
    print(grid_search.best_estimator_)
```

Baseline Model

In [99]:

```
base_line_model = DummyClassifier(random_state=random_seed)

model_metrics = train_model('base_line', base_line_model, X_train_pca, y_train_r
esampled,
                X_test_pca, y_test)
model_metrics
```




```
ModelName: base_line          | time (training/test) = 0.051s/0.217s
```

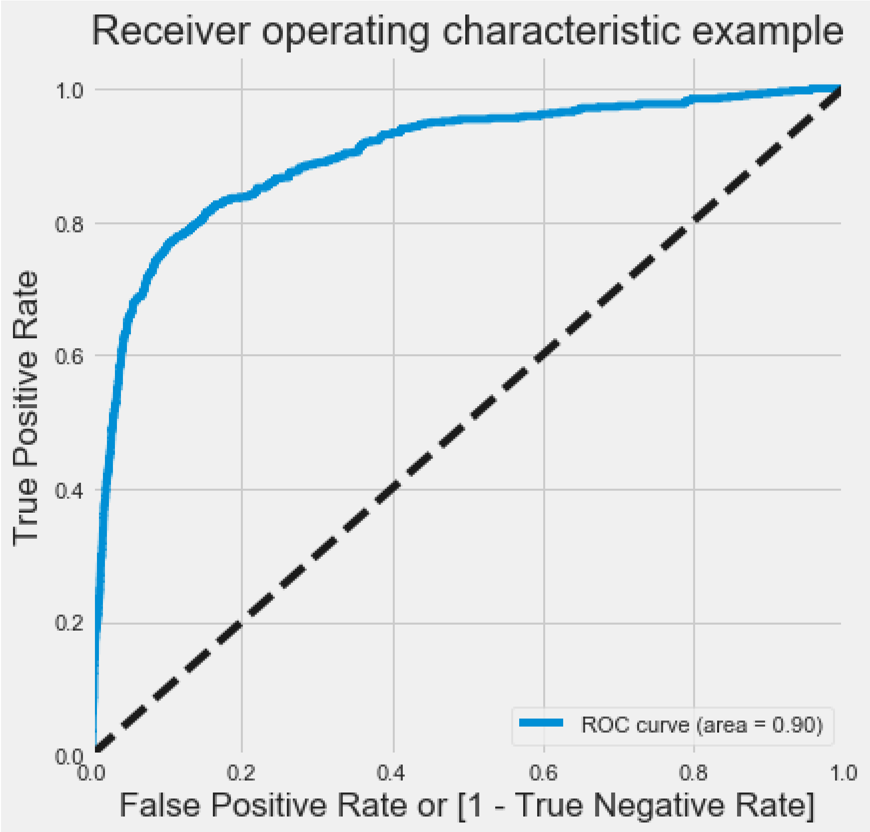
Out[99]:

	name	train_accuracy	train_precision	train_recall	train_f1	train_auc	test_accuracy	tes
0	base_line	0.5006	0.5006	0.5004	0.5005	0.5006	0.5011	

Logistic Regression

In [101]:

```
logistic_model = LogisticRegression(random_state=random_seed)

df = train_model('logistic_regression', logistic_model, X_train_pca, y_train_res
ampled,
            X_test_pca, y_test)
model_metrics = pd.concat([model_metrics,df],axis=0)
df
```




```
ModelName: logistic_regression | time (training/test) = 0.244s/0.223
s
```

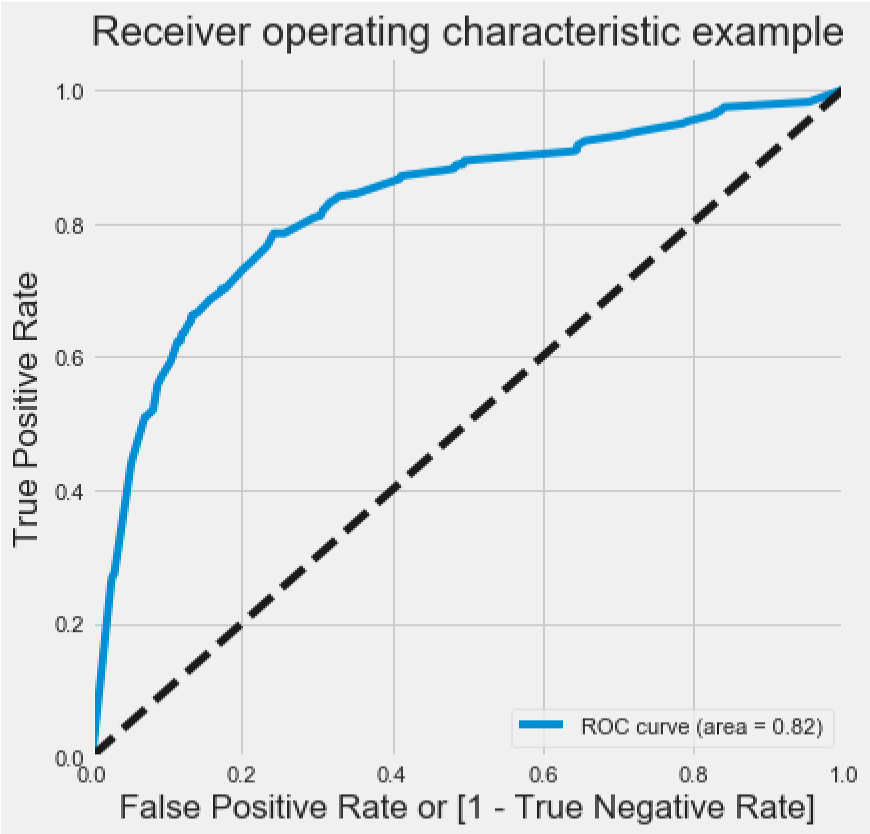
Out[101]:

	name	train_accuracy	train_precision	train_recall	train_f1	train_auc	test_accui
0	logistic_regression	0.8433	0.8312	0.8616	0.8461	0.9144	0.8

Decision Tree

In [102]:

```
decision_tree_model = DecisionTreeClassifier(random_state=random_seed,
                                             max_depth=7,
                                             min_samples_leaf=1,
                                             min_samples_split=2)

df = train_model('decision_tree_default', decision_tree_model, X_train_pca,
                 y_train_resampled, X_test_pca, y_test)
model_metrics = pd.concat([model_metrics, df], axis=0)
df
```




```
ModelName: decision_tree_default | time (training/test) = 2.229s/0.2
15s
```

Out[102]:

	name	train_accuracy	train_precision	train_recall	train_f1	train_auc	test_ac
0	decision_tree_default	0.8457	0.8509	0.8383	0.8445	0.9117	

Hyperparameter tuning

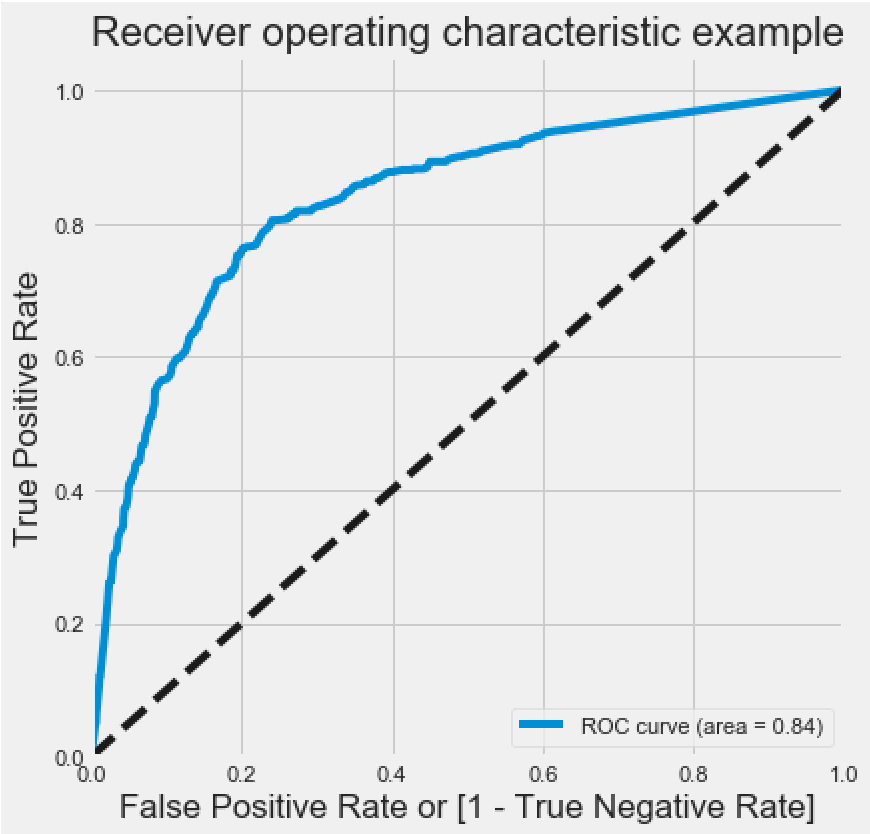
In [103]:

```
param_grid = {
    'max_depth': range(5, 15, 5),
    'min_samples_leaf': range(50, 150, 50),
    'min_samples_split': range(50, 150, 50),
    'criterion': ["entropy", "gini"]
}
hyperparameter_tuning(DecisionTreeClassifier(), param_grid)
```

```
Fitting 5 folds for each of 16 candidates, totalling 80 fits


[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent
workers.
[Parallel(n_jobs=-1)]: Done  26 tasks      | elapsed:   12.3s
[Parallel(n_jobs=-1)]: Done  80 out of  80 | elapsed:   22.9s finish
ed

DecisionTreeClassifier(criterion='entropy', max_depth=10, min_sample
s_leaf=50,
                       min_samples_split=50)
```

In [104]:

```
decision_tree_model = DecisionTreeClassifier(random_state=random_seed,
                                             criterion='entropy',
                                             max_depth=10,
                                             min_samples_leaf=50,
                                             min_samples_split=50)
df = train_model('decision_tree_tuned', decision_tree_model, X_train_pca,
                 y_train_resampled, X_test_pca, y_test)
model_metrics = pd.concat([model_metrics, df], axis=0)
df
```




```
ModelName: decision_tree_tuned | time (training/test) = 2.732s/0.195
s
```

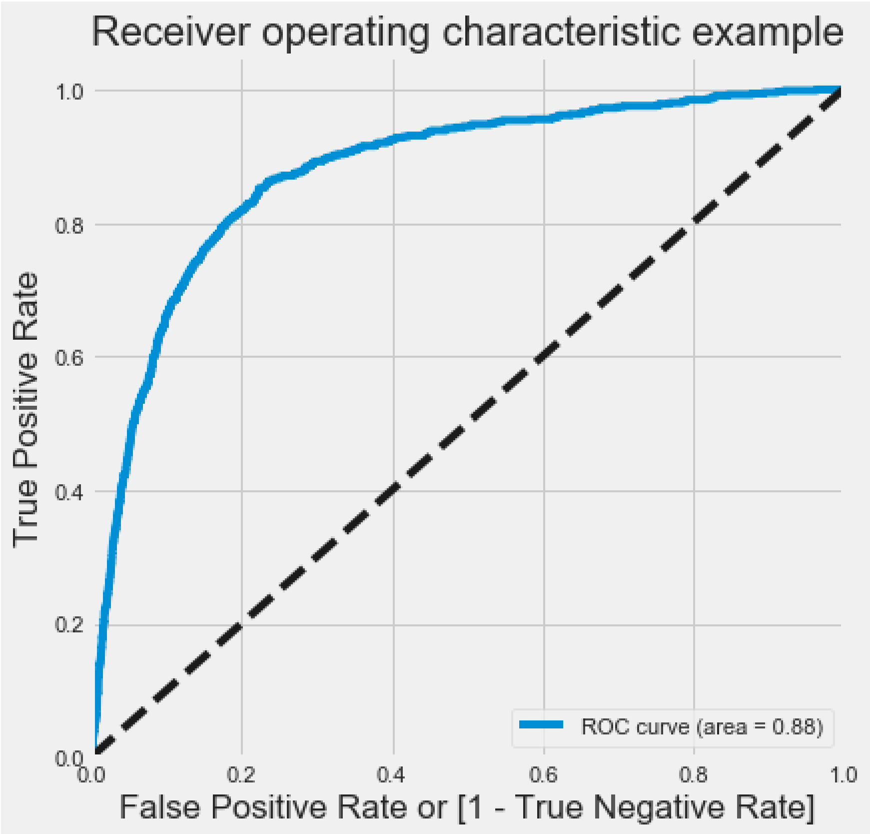
Out[104]:

	name	train_accuracy	train_precision	train_recall	train_f1	train_auc	test_acc
0	decision_tree_tuned	0.8700	0.8649	0.8770	0.8709	0.9463	0

Random Forest

In [105]:

```
random_forest_model = RandomForestClassifier(random_state=random_seed,
                                             n_estimators=100,
                                             max_depth=7,
                                             min_samples_leaf=1,
                                             min_samples_split=2)

df = train_model('random_forest_default', random_forest_model, X_train_pca,
                 y_train_resampled, X_test_pca, y_test)
model_metrics = pd.concat([model_metrics, df], axis=0)
df
```




```
ModelName: random_forest_default | time (training/test) = 13.139s/0.
384s
```

Out[105]:

	name	train_accuracy	train_precision	train_recall	train_f1	train_auc	test_a
0	random_forest_default	0.8721	0.8798	0.8620	0.8708	0.9438	

Hyperparameter tuning

This hyper parameter tuning for Random Forest takes around **4** minutes on a machine with 32 GB ram and 12 processor

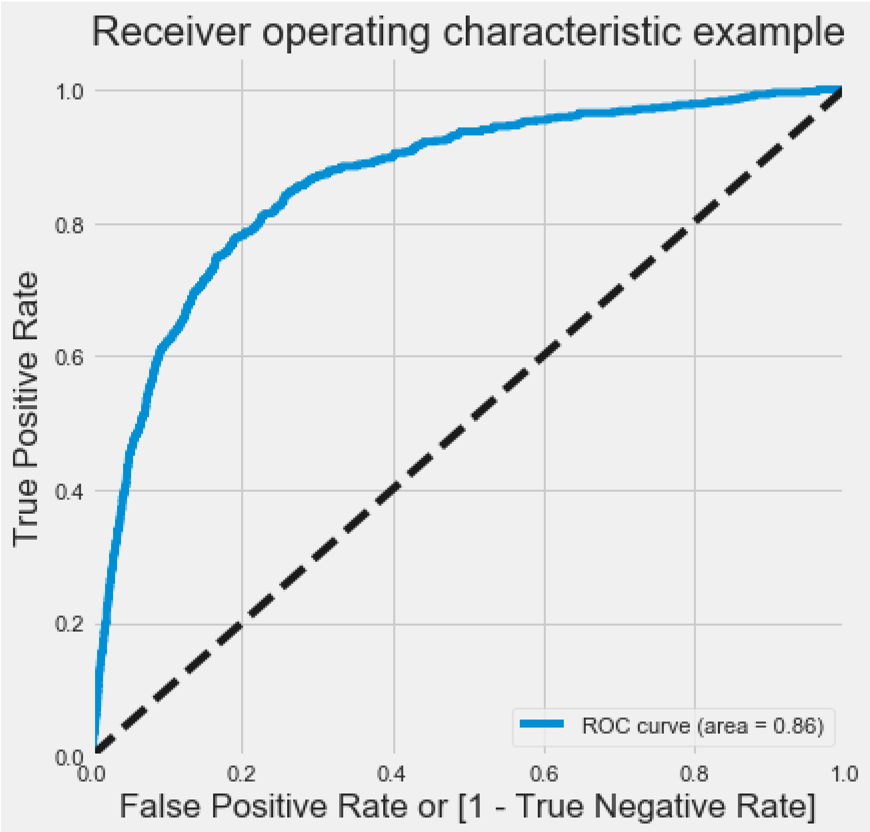
In [106]:

```
param_grid = {
    'max_depth': range(4, 8, 10),
    'min_samples_leaf': range(50, 150, 50),
    'min_samples_split': range(50, 150, 50),
    'n_estimators': [100, 150, 200],
    'max_features': [5, 10]
}
hyperparameter_tuning(RandomForestClassifier(), param_grid)
```

```
Fitting 5 folds for each of 24 candidates, totalling 120 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent
workers.
[Parallel(n_jobs=-1)]: Done  26 tasks      | elapsed:   35.8s
[Parallel(n_jobs=-1)]: Done 120 out of 120 | elapsed:  3.4min finish
ed

RandomForestClassifier(max_depth=4, max_features=5, min_samples_leaf
=50,
                       min_samples_split=100, n_estimators=200)
```

In [107]:

```
random_forest_model = RandomForestClassifier(random_state=random_seed,
                                             n_estimators=200,
                                             max_depth=4,
                                             min_samples_leaf=100,
                                             min_samples_split=100)

df = train_model('random_forest_tuned', random_forest_model, X_train_pca,
                 y_train_resampled, X_test_pca, y_test)
model_metrics = pd.concat([model_metrics, df], axis=0)
df
```




```
ModelName: random_forest_tuned | time (training/test) = 15.770s/0.35
6s
```

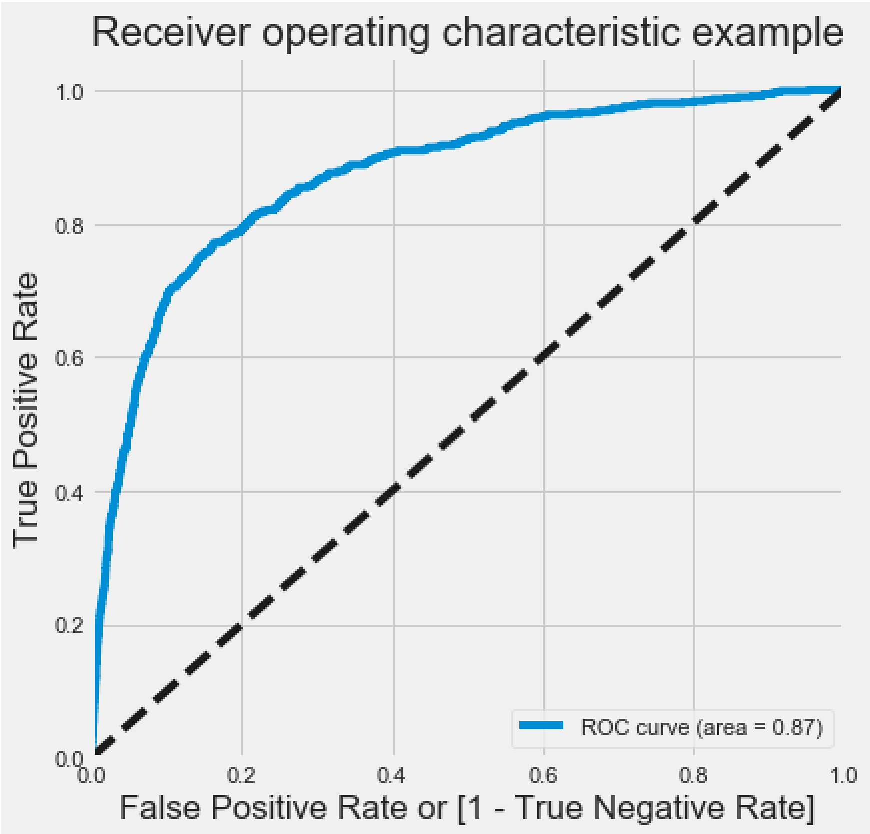
Out[107]:

	name	train_accuracy	train_precision	train_recall	train_f1	train_auc	test_ac
0	random_forest_tuned	0.8260	0.8405	0.8045	0.8221	0.9016	

AdaBoost

In [108]:

```
weak_learner = DecisionTreeClassifier(max_depth=2, random_state=random_seed)

adaboost = AdaBoostClassifier(base_estimator=weak_learner,
                              n_estimators=200,
                              learning_rate=1.5,
                              algorithm="SAMME")

df = train_model('adaboost_default', adaboost, X_train_pca,
                 y_train_resampled, X_test_pca, y_test)
model_metrics = pd.concat([model_metrics, df], axis=0)
df
```




```
ModelName: adaboost_default | time (training/test) = 133.890s/1.623s
```

Out[108]:

	name	train_accuracy	train_precision	train_recall	train_f1	train_auc	test_accura
0	adaboost_default	0.8910	0.8929	0.8885	0.8907	0.9606	0.85

Hyperparameter tuning

This hyper parameter tuning for AdaBoost Classifier takes around **15** minutes on a machine with 32 GB ram and 12 processor

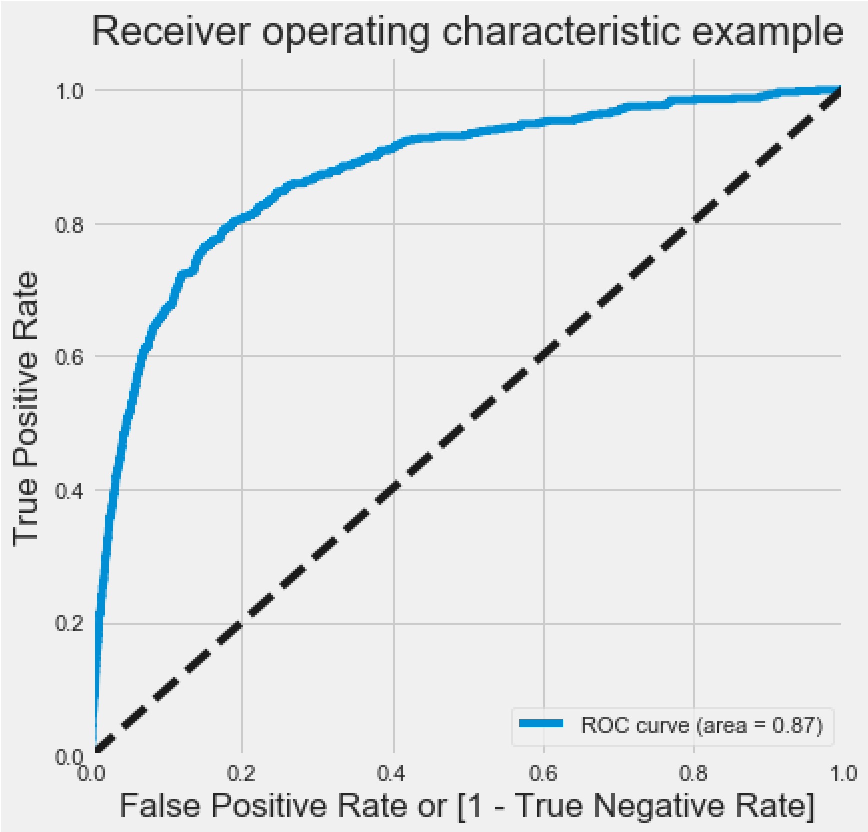
In [109]:

```
param_grid = {
    "base_estimator__max_depth": [2, 4],
    "n_estimators": [200, 300],
    "learning_rate": [.3, .9, .3]
}

tree = DecisionTreeClassifier(random_state=random_seed)

ABC = AdaBoostClassifier(base_estimator=tree, algorithm="SAMME")

hyperparameter_tuning(ABC, param_grid, 3, 'roc_auc')
```

```
Fitting 3 folds for each of 12 candidates, totalling 36 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent
workers.
[Parallel(n_jobs=-1)]: Done  36 out of  36 | elapsed: 15.3min finish
ed

AdaBoostClassifier(algorithm='SAMME',
                   base_estimator=DecisionTreeClassifier(max_depth=
4,
                                                         random_stat
e=101),
                   learning_rate=0.9, n_estimators=300)
```

In [110]:

```
weak_learner = DecisionTreeClassifier(max_depth=4, random_state=random_seed)

adaboost = AdaBoostClassifier(base_estimator=weak_learner,
                              n_estimators=300,
                              learning_rate=0.9,
                              algorithm="SAMME")

df = train_model('adaboost_tuned', adaboost, X_train_pca, y_train_resampled,
                 X_test_pca, y_test)
model_metrics = pd.concat([model_metrics, df], axis=0)
df
```




```
ModelName: adaboost_tuned   | time (training/test) = 310.034s/0.918s
```

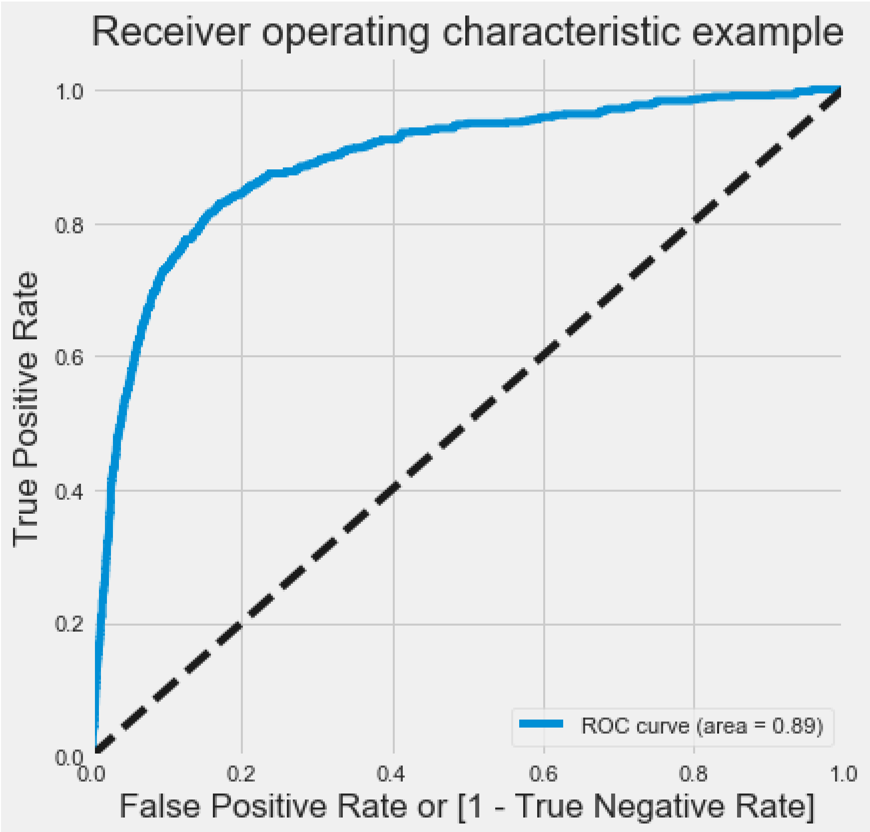
Out[110]:

	name	train_accuracy	train_precision	train_recall	train_f1	train_auc	test_accurac
0	adaboost_tuned	0.9797	0.9683	0.9919	0.9799	0.9988	0.895

Gradient Boosting Classifier

In [111]:

```
GBC = GradientBoostingClassifier(max_depth=2, n_estimators=200, random_state=ran
dom_seed)
df = train_model('gradient_boost_default', GBC, X_train_pca, y_train_resampled,
                 X_test_pca, y_test)
model_metrics = pd.concat([model_metrics, df], axis=0)
df
```




```
ModelName: gradient_boost_default | time (training/test) = 96.156s/
0.221s
```

Out[111]:

	name	train_accuracy	train_precision	train_recall	train_f1	train_auc	test_a
0	gradient_boost_default	0.8669	0.8650	0.8697	0.8673	0.9371	

Hyperparameter tuning

This hyper parameter tuning for Gradient Boosting Classifier takes around **4** minutes on a machine with 32 GB ram and 12 processor

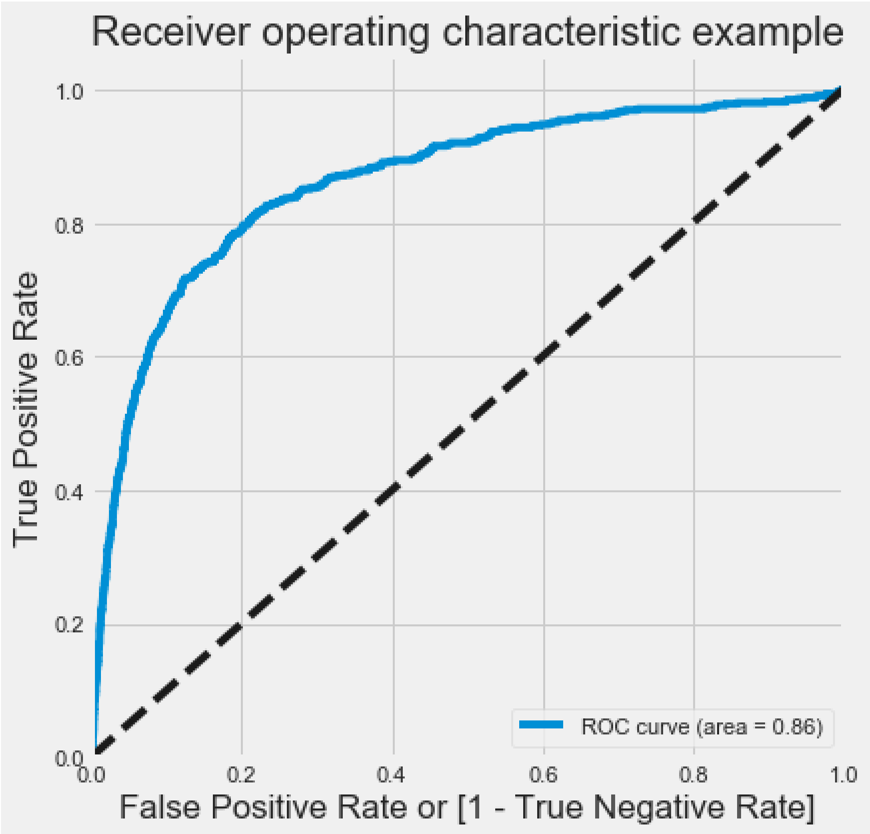
In [112]:

```
param_grid = {"learning_rate": [0.2, 0.6, 0.9], "subsample": [0.3, 0.6, 0.9]}
GBC = GradientBoostingClassifier(max_depth=2,
                                 n_estimators=200,
                                 random_state=random_seed)
hyperparameter_tuning(GBC, param_grid, n_folds=3)
```

```
Fitting 3 folds for each of 9 candidates, totalling 27 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent
workers.
[Parallel(n_jobs=-1)]: Done  27 out of  27 | elapsed:  3.2min finish
ed


GradientBoostingClassifier(learning_rate=0.9, max_depth=2, n_estimat
ors=200,
                           random_state=101, subsample=0.9)
```

In [113]:

```
GBC = GradientBoostingClassifier(max_depth=2,
                                  n_estimators=200,
                                  learning_rate=0.9,
                                  subsample=0.9,
                                  random_state=random_seed)
df = train_model('gradient_boost_tuned', GBC, X_train_pca, y_train_resampled,
                 X_test_pca, y_test)
model_metrics = pd.concat([model_metrics, df], axis=0)
df
```




```
ModelName: gradient_boost_tuned | time (training/test) = 84.870s/0.2
17s
```

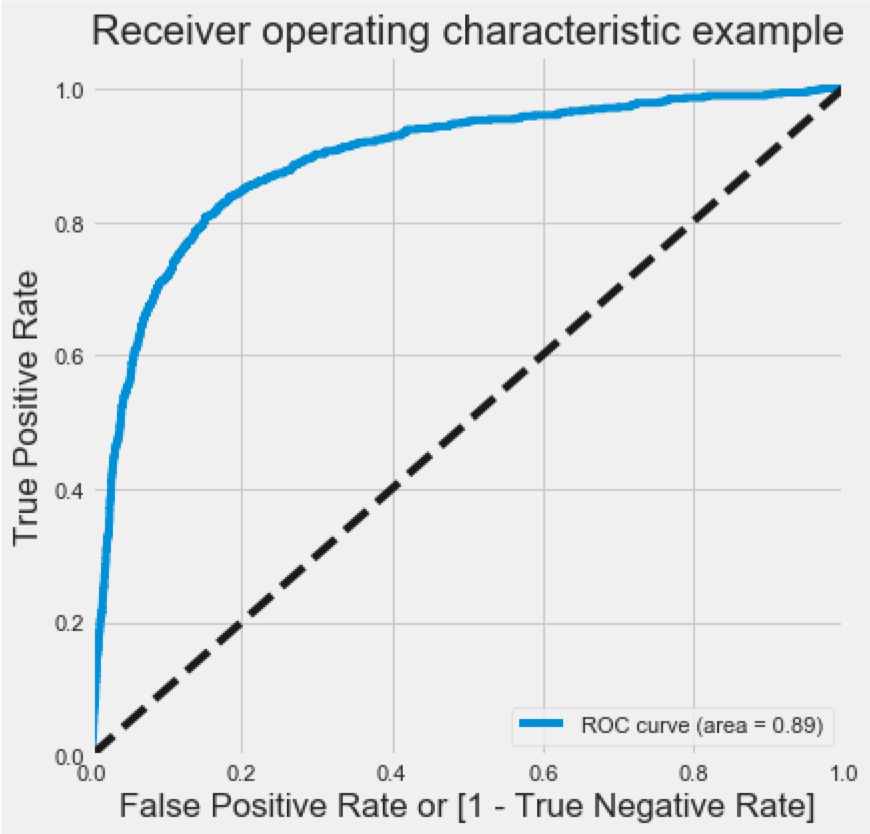
Out[113]:

	name	train_accuracy	train_precision	train_recall	train_f1	train_auc	test_ac
0	gradient_boost_tuned	0.9314	0.9185	0.9469	0.9325	0.9774	

XGBoost

In [114]:

```
xgb = XGBClassifier(max_depth=2,
                    n_estimators=200,
                    nthread=-1,
                    random_state=random_seed)
df = train_model('xgboost_default', xgb, X_train_pca, y_train_resampled,
                 X_test_pca, y_test)
model_metrics = pd.concat([model_metrics, df], axis=0)
df
```




```
ModelName: xgboost_default | time (training/test) = 5.361s/0.249s
```

Out[114]:

	name	train_accuracy	train_precision	train_recall	train_f1	train_auc	test_accurac
0	xgboost_default	0.8656	0.8647	0.8668	0.8657	0.9368	0.846

Hyperparameter tuning

This hyper parameter tuning for XGBoostClassifier takes around **12** minutes on a machine with 32 GB ram and 12 processor

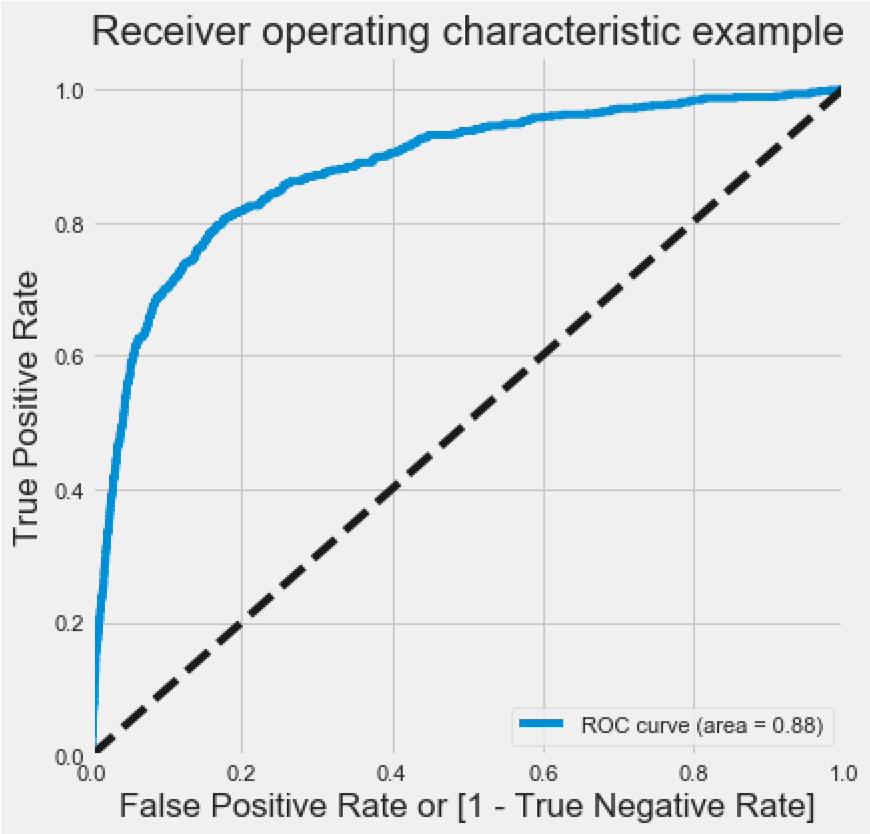
In [115]:

```
param_grid = {
    "learning_rate": [0.1, 0.2, 0.3],
    "subsample": [0.3, 0.6, 0.9],
    'n_estimators': [200, 400, 600]
}
xgb = XGBClassifier(max_depth=2,
                    n_estimators=200,
                    nthread=-1,
                    random_state=random_seed)
hyperparameter_tuning(xgb, param_grid, n_folds=3)
```

```
Fitting 3 folds for each of 27 candidates, totalling 81 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent
workers.
[Parallel(n_jobs=-1)]: Done  26 tasks      | elapsed:  2.7min
[Parallel(n_jobs=-1)]: Done  81 out of  81 | elapsed:  8.2min finish
ed

XGBClassifier(learning_rate=0.3, max_depth=2, n_estimators=600, nthr
ead=-1,
              random_state=101, subsample=0.6)
```

In [116]:

```
xgb = XGBClassifier(max_depth=2,
                    n_estimators=600,
                    nthread=-1,
                    learning_rate=0.3,
                    subsample=0.9)

df = train_model('xgboost_tuned', xgb, X_train_pca, y_train_resampled,
                 X_test_pca, y_test)
model_metrics = pd.concat([model_metrics, df], axis=0)
df
```




```
ModelName: xgboost_tuned    | time (training/test) = 18.266s/0.313s
```

Out[116]:

	name	train_accuracy	train_precision	train_recall	train_f1	train_auc	test_accuracy
0	xgboost_tuned	0.9450	0.9282	0.9646	0.9461	0.9846	0.8745

Model evaluation

In [117]:

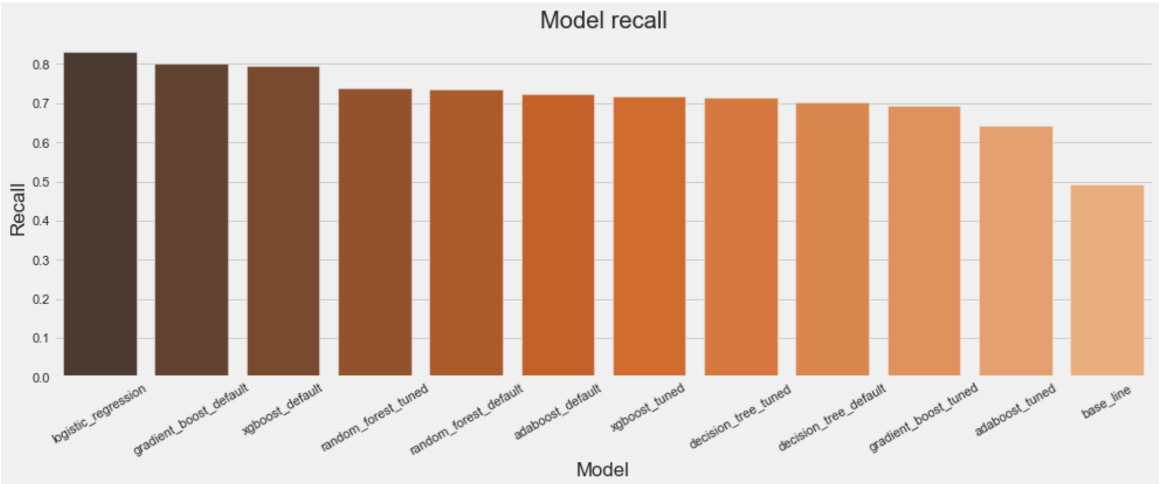
```
model_metrics
```

Out[117]:

	name	train_accuracy	train_precision	train_recall	train_f1	train_auc	test_a
0	base_line	0.5006	0.5006	0.5004	0.5005	0.5006	
0	logistic_regression	0.8433	0.8312	0.8616	0.8461	0.9144	
0	decision_tree_default	0.8457	0.8509	0.8383	0.8445	0.9117	
0	decision_tree_tuned	0.8700	0.8649	0.8770	0.8709	0.9463	
0	random_forest_default	0.8721	0.8798	0.8620	0.8708	0.9438	
0	random_forest_tuned	0.8260	0.8405	0.8045	0.8221	0.9016	
0	adaboost_default	0.8910	0.8929	0.8885	0.8907	0.9606	
0	adaboost_tuned	0.9797	0.9683	0.9919	0.9799	0.9988	
0	gradient_boost_default	0.8669	0.8650	0.8697	0.8673	0.9371	
0	gradient_boost_tuned	0.9314	0.9185	0.9469	0.9325	0.9774	
0	xgboost_default	0.8656	0.8647	0.8668	0.8657	0.9368	
0	xgboost_tuned	0.9450	0.9282	0.9646	0.9461	0.9846	

In [118]:

```
model_metrics = model_metrics.sort_values(by='test_recall', ascending=False)
sns.barplot(model_metrics['name'],
            model_metrics['test_recall'],
            palette='Oranges_d')
plt.xticks(rotation=30, horizontalalignment="center")
plt.title("Model recall")
plt.xlabel("Model")
plt.ylabel("Recall")
plt.show()
```



In this problem, data points corresponding to the target class is very less in number. Thus max `recall` will be the metrics which we will be targeting for. **We want to reduce *Type 2* error which *False Negative*.**

We created baseline model which always selected a single class. This model resulted in a recall value of around **.50**.

As we can see from the above chart and the table that the ***Logistic Regression*** has the best *recall* value of ***0.83***. Default ***Gradient Boost Classifier*** has the next best *recall* value of ***0.79***. Other boosting classifiers worked well on the training data however, performed poorly on the test data.

Thus we will be selecting ***Logistic Regression*** as our final classifier

Important features for churn

As suggested in the problem statement, we can either use *Logistic Regression* or *Random Forest* to identify the most important features. We will be using ***Random Forest*** as we need not handle multi collinearity of the features.

In [119]:

```
def plot_important_features(classifer):
    importance = dict(zip(X.columns, classifer.feature_importances_))
    importance_list = []
    importance_list.append(importance)
    df = pd.DataFrame.from_dict(importance_list).T.reset_index()
    df.columns = ['feature', 'importance']
    df = df.sort_values(by='importance', ascending=False)
    df = df.head(35)
    sns.set(rc={'figure.figsize': (20, 10)})
    sns.barplot(df['feature'],
            df['importance'],
            palette='Oranges_d')
    plt.xticks(rotation=90, horizontalalignment="center")
    plt.title("Feature Importance")
    plt.xlabel("Feature")
    plt.ylabel("Importance")
    plt.show()
    return df
```

The following hyper parameter tuning will take around *30 minutes* on a machine with 32GB ram and 12 processors.

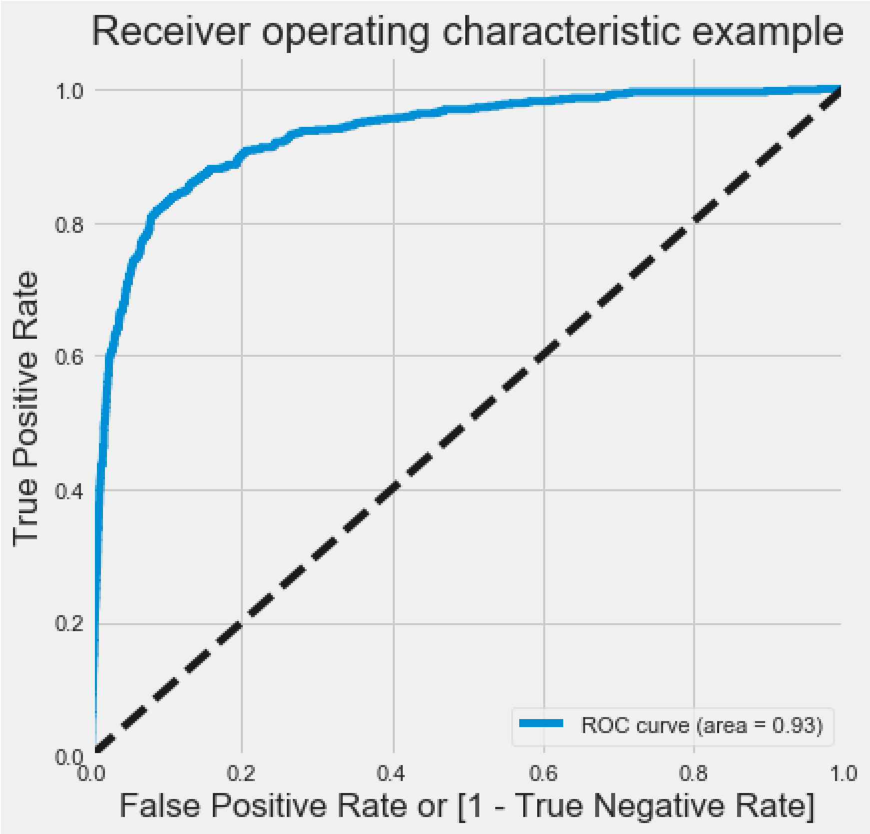
In [120]:

```
param_grid = {
    'max_depth': range(4, 12, 4),
    'min_samples_leaf': range(50, 200, 50),
    'min_samples_split': range(50, 200, 50),
    'n_estimators': range(100, 500, 100),
    'max_features': range(5, 20, 5)
}
hyperparameter_tuning(RandomForestClassifier(random_state=random_seed),
                      param_grid,
                      n_folds=3,
                      X_train=X_train_resampled)
```

```
Fitting 3 folds for each of 216 candidates, totalling 648 fits

[Parallel(n_jobs=-1)]: Using backend LokyBackend with 12 concurrent
workers.
[Parallel(n_jobs=-1)]: Done  26 tasks      | elapsed:   25.1s
[Parallel(n_jobs=-1)]: Done 176 tasks      | elapsed:  3.1min
[Parallel(n_jobs=-1)]: Done 426 tasks      | elapsed:  9.5min
[Parallel(n_jobs=-1)]: Done 648 out of 648 | elapsed: 19.2min finish
ed

RandomForestClassifier(max_depth=8, max_features=15, min_samples_lea
f=50,
                       min_samples_split=50, n_estimators=400,
                       random_state=101)
```

In [121]:

```
random_forest_model = RandomForestClassifier(random_state=random_seed,
                                             n_estimators=400,
                                             max_depth=8,
                                             max_features=15,
                                             min_samples_leaf=50,
                                             min_samples_split=50)

df = train_model('random_forest', random_forest_model, X_train_resampled,
                 y_train_resampled, X_test, y_test)
df
```



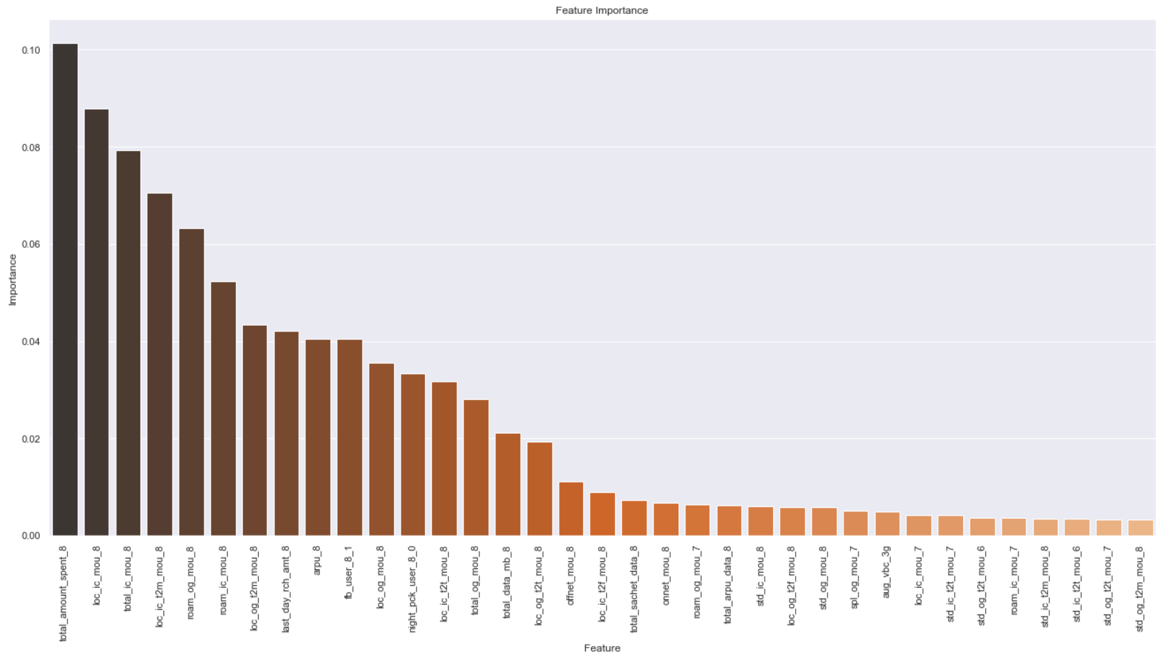

```
ModelName: random_forest    | time (training/test) = 44.049s/0.643s
```

Out[121]:

	name	train_accuracy	train_precision	train_recall	train_f1	train_auc	test_accuracy
0	random_forest	0.9162	0.9225	0.9087	0.9156	0.9702	0.9090

In [122]:

```
features = plot_important_features(random_forest_model)
```



Top 25 important features

In [127]:

```
features.head(25).reset_index(drop=True)
```

Out[127]:

	feature	importance
0	total_amount_spent_8	0.1013
1	loc_ic_mou_8	0.0879
2	total_ic_mou_8	0.0793
3	loc_ic_t2m_mou_8	0.0705
4	roam_og_mou_8	0.0632
5	roam_ic_mou_8	0.0524
6	loc_og_t2m_mou_8	0.0435
7	last_day_rch_amt_8	0.0421
8	arpu_8	0.0405
9	fb_user_8_1	0.0405
10	loc_og_mou_8	0.0356
11	night_pck_user_8_0	0.0334
12	loc_ic_t2t_mou_8	0.0318
13	total_og_mou_8	0.0280
14	total_data_mb_8	0.0211
15	loc_og_t2t_mou_8	0.0193
16	offnet_mou_8	0.0111
17	loc_ic_t2f_mou_8	0.0089
18	total_sachet_data_8	0.0073
19	onnet_mou_8	0.0067
20	roam_og_mou_7	0.0064
21	total_arpu_data_8	0.0061
22	std_ic_mou_8	0.0060
23	loc_og_t2f_mou_8	0.0059
24	std_og_mou_8	0.0058

Top 25 features with description

Feature	Description
total_amount_spent_8	Total amount spent in August
loc_ic_mou_8	Local incoming usage in August
total_ic_mou_8	Total incoming usage in August
loc_ic_t2m_mou_8	Local incoming usage other mobile network in August
roam_og_mou_8	Outgoing roaming usage in August
roam_ic_mou_8	Incoming roaming usage in August
loc_og_t2m_mou_8	Local outgoing usage to other mobile network in August
last_day_rch_amt_8	Last recharge amount in August
arpu_8	Average revenue in August
fb_user_8_1	Mobile pack recharge for social surfing in August
loc_og_mou_8	Local outgoing usage in August
night_pck_user_8_0	Night pack recharge in August
loc_ic_t2t_mou_8	Local incoming usage within same operator in August
total_og_mou_8	Total outgoing monthly usage in August
total_data_mb_8	Total data usage in August
loc_og_t2t_mou_8	Local outgoing usage within same operator in August
offnet_mou_8	Outside network monthly usage in August
loc_ic_t2f_mou_8	Local incoming usage for operator to fixed line in August
total_sachet_data_8	Number of data validity pack in August
onnet_mou_8	Inside operator network monthly usage in August
roam_og_mou_7	Roaming outgoing monthly usage in July
total_arpu_data_8	Total data average revenue in August
std_ic_mou_8	STD incoming monthly usage in August
loc_og_t2f_mou_8	Local outgoing operator to fixed line monthly usage in August
std_og_mou_8	STD outgoing monthly usage in August

We can see from the above table that ***24 out of top 25*** features are from August, which as per the problem statement is the action month. Reduction is the total amount spent is the strong predictor for a customer to churn. This followed by a ***drop in the monthly usage*** of almost all the services like ***incoming, outgoing calls, roaming etc*** strongly suggest in the customer behavior to churn.

Recommended strategies to manage customer churn

As can be seen from the above mentioned top churn predictors: the most important one is to monitor the drop in overall usage of the service like

1. Drop in incoming and outgoing calls
2. Drop in roaming incoming and outgoing calls
3. Drop in data usages
4. Drop in recharges frequency or amount

Suggested strategies:

1. Revisit the calling packs/plans. Suggest users plans which suits to their needs. For example: If a customer who does too many STD calls, should be suggested packs/plan which reduces his/her cost. These plans can be bundled with other plans like data.
2. The customers can be suggested to use bundle packs which reduces overall cost for calling. These can be weekly or monthly packs.
3. Monitor the competitors calls plans/packs to provide better deals to the existing customers so that they do not churn to the competitors
4. Data usages is a strong indicator for the customer churn. The company can run campaigns/deals/bundles/packs which reduces the data cost like monthly/quarterly/half-yearly/yearly plans/packs which provide bulk data in advance. These long term plans tie the customers thus reducing the churn.
5. Combining different data packs into simpler packs like combining social packs with data etc.
6. Revisiting roaming plans/packs: roaming charges play a major role in customer churn. The company can come up with plans like free incoming in nights etc to lower customers roaming expenses thus reducing customer churn.
7. The company can come up with plans which are tailored for the customer usages. Like if a customer usage local calls, he can be suggested plans which reduce this cost.
8. The company can also come up with plans which reduce cost for with in network calls. This will also drive customer to create his/her circle of same network thus increasing customer base.

In [ ]:

```
```