# Final Project Submission

Please fill out:

- Student name: Carrie Liu
- Student pace: self paced
- Scheduled project review date/time: December 27, 2021 / 2PM EST
- Instructor name: Claude Fried
- Blog post URL:https://medium.com/@carrielearn/syriatel-churn-analysis-bc2de1574968 (https://medium.com/@carrielearn/syriatel-churn-analysis-bc2de1574968)

# Overview

Our client, SyriaTel, is a telecommunication company and is suffering from a loss of valuable customers to competitors.

Understanding customer churn is essential to evaluating the effectiveness of the company's marketing efforts and the overall satisfaction of the customers. It’s also easier and less expensive to keep existing customers versus to acquire new ones.

Therefore, we are hired to help the management team understand what features are primary determinants of the customer churn. We will further build a classification model to predict whether a customer will ("soon") stop doing business with SyriaTel.

SyriaTel is a Syria based cell phone service company and the dataset we will work on includes 3333 customers of SyriaTel in the U.S., covering 51 states (including D.C.) over a month period.

It is a binary classification problem. Our approach is:

- Perform exploratory data analysis on current data. The raw data is downloaded from Kaggle (https://www.kaggle.com/becksddf/churn-in-telecoms-dataset).
- Build up baseline model: logistic regression
- Apply multiple machine learning algorithms to build classifier: K-Nearest Neighbors, Decision Trees, Random Forest, AdaBoost, Gradient Boost, XGBoost, and Support Vector Machine
- Select the best model for classification

# Exploratory Data Analysis (EDA)

## 1. Import packages

In [1]:
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
%matplotlib inline
import seaborn as sns
import plotly.express as px
```

In [2]:
```
from sklearn.preprocessing import StandardScaler, MinMaxScaler
from sklearn.preprocessing import OneHotEncoder
from sklearn.metrics import make_scorer
from sklearn.metrics import mean_squared_error, log_loss
from sklearn.metrics import accuracy_score, precision_score, recall_score,
from sklearn.metrics import classification_report, confusion_matrix, plot_c
from imblearn.over_sampling import SMOTE
from sklearn.base import clone
```

In [3]:
```
from sklearn.model_selection import train_test_split, cross_val_score, Stra
from sklearn.linear_model import LinearRegression, LogisticRegression
from sklearn.linear_model import Ridge, Lasso, ElasticNet
from sklearn.neighbors import KNeighborsClassifier
from sklearn.tree import DecisionTreeClassifier
from sklearn.ensemble import BaggingClassifier, RandomForestClassifier
from sklearn.ensemble import AdaBoostClassifier, GradientBoostingClassifier
from xgboost import XGBClassifier, plot_importance
from sklearn.svm import SVC
```

In [4]:
```
import warnings
warnings.filterwarnings("ignore")
```

## 2. Load data

In [5]:
```
data = pd.read_csv('data.csv')
```

In [6]:
```
data.head()
```

Out[6]:

| | state | account length | area code | phone number | international plan | voice mail plan | number vmail messages | total day minutes | total day calls | total day charge | ... | tot ev cal |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| **0** | KS | 128 | 415 | 382-4657 | no | yes | 25 | 265.1 | 110 | 45.07 | ... | ( |
| **1** | OH | 107 | 415 | 371-7191 | no | yes | 26 | 161.6 | 123 | 27.47 | ... | 1( |
| **2** | NJ | 137 | 415 | 358-1921 | no | no | 0 | 243.4 | 114 | 41.38 | ... | 1 |
| **3** | OH | 84 | 408 | 375-9999 | yes | no | 0 | 299.4 | 71 | 50.90 | ... | { |
| **4** | OK | 75 | 415 | 330-6626 | yes | no | 0 | 166.7 | 113 | 28.34 | ... | 1: |

5 rows × 21 columns

In [7]:
```
data.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3333 entries, 0 to 3332
Data columns (total 21 columns):
 #   Column                  Non-Null Count  Dtype
---  ------                  --------------  -----
 0   state                   3333 non-null   object
 1   account length          3333 non-null   int64
 2   area code               3333 non-null   int64
 3   phone number            3333 non-null   object
 4   international plan      3333 non-null   object
 5   voice mail plan         3333 non-null   object
 6   number vmail messages   3333 non-null   int64
 7   total day minutes       3333 non-null   float64
 8   total day calls         3333 non-null   int64
 9   total day charge        3333 non-null   float64
 10  total eve minutes       3333 non-null   float64
 11  total eve calls         3333 non-null   int64
 12  total eve charge        3333 non-null   float64
 13  total night minutes     3333 non-null   float64
 14  total night calls       3333 non-null   int64
 15  total night charge      3333 non-null   float64
 16  total intl minutes      3333 non-null   float64
 17  total intl calls        3333 non-null   int64
 18  total intl charge       3333 non-null   float64
 19  customer service calls  3333 non-null   int64
 20  churn                   3333 non-null   bool
dtypes: bool(1), float64(8), int64(8), object(4)
memory usage: 524.2+ KB
```

In [8]:
```
data.columns
```

Out[8]:
```
Index(['state', 'account length', 'area code', 'phone number',
       'international plan', 'voice mail plan', 'number vmail messages',
       'total day minutes', 'total day calls', 'total day charge',
       'total eve minutes', 'total eve calls', 'total eve charge',
       'total night minutes', 'total night calls', 'total night charge',
       'total intl minutes', 'total intl calls', 'total intl charge',
       'customer service calls', 'churn'],
      dtype='object')
```

In [9]:
```
data.describe()
```

Out[9]:

| | account length | area code | number vmail messages | total day minutes | total day calls | total day charge | total eve minutes |
|---|---|---|---|---|---|---|---|
| **count** | 3333.000000 | 3333.000000 | 3333.000000 | 3333.000000 | 3333.000000 | 3333.000000 | 3333.000000 |
| **mean** | 101.064806 | 437.182418 | 8.099010 | 179.775098 | 100.435644 | 30.562307 | 200.980348 |
| **std** | 39.822106 | 42.371290 | 13.688365 | 54.467389 | 20.069084 | 9.259435 | 50.713844 |
| **min** | 1.000000 | 408.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 | 0.000000 |
| **25%** | 74.000000 | 408.000000 | 0.000000 | 143.700000 | 87.000000 | 24.430000 | 166.600000 |
| **50%** | 101.000000 | 415.000000 | 0.000000 | 179.400000 | 101.000000 | 30.500000 | 201.400000 |
| **75%** | 127.000000 | 510.000000 | 20.000000 | 216.400000 | 114.000000 | 36.790000 | 235.300000 |
| **max** | 243.000000 | 510.000000 | 51.000000 | 350.800000 | 165.000000 | 59.640000 | 363.700000 |

## 3. Data cleaning

### Select and rename relevant columns

In [10]:
```
data.columns = data.columns.str.replace(" ", "_")
```

In [11]:
```
data.drop(columns='phone_number', inplace = True)
```

### Check missing values

In [12]:
```
data.isna().sum()
```

Out[12]:
```
state                     0
account_length            0
area_code                 0
international_plan        0
voice_mail_plan           0
number_vmail_messages     0
total_day_minutes         0
total_day_calls           0
total_day_charge          0
total_eve_minutes         0
total_eve_calls           0
total_eve_charge          0
total_night_minutes       0
total_night_calls         0
total_night_charge        0
total_intl_minutes        0
total_intl_calls          0
total_intl_charge         0
customer_service_calls    0
churn                     0
dtype: int64
```

#### Convert columns dtype from object / bool to int

In [13]:
```
data['international_plan'] = data['international_plan'].replace('yes', 1)
data['international_plan'] = data['international_plan'].replace('no', 0)
data['voice_mail_plan'] = data['voice_mail_plan'].replace('yes', 1)
data['voice_mail_plan'] = data['voice_mail_plan'].replace('no', 0)
```

In [14]:
```
data['churn'].replace(False, 0, inplace = True)
data['churn'].replace(True, 1, inplace = True)
```

## 4. Data analysis and visualization

#### Predictor: Churn

The percentage of customers who churned in the sample is 14.5% (i.e. 483 / 3333)

In [15]:
```
data['churn'].value_counts()
```

Out[15]:
```
0.0    2850
1.0     483
Name: churn, dtype: int64
```

In [16]:
```
data['churn'].value_counts(normalize=True)
```

Out[16]:
```
0.0    0.855086
1.0    0.144914
Name: churn, dtype: float64
```

#### Features - State and Areacode

In [17]:
```
#state
len(data['state'].unique())
```

Out[17]:
```
51
```

In [18]:
```
state_series = data.groupby(['state'])['churn'].mean().sort_values(ascendin
state_df = state_series.to_frame().reset_index()
state_df
```

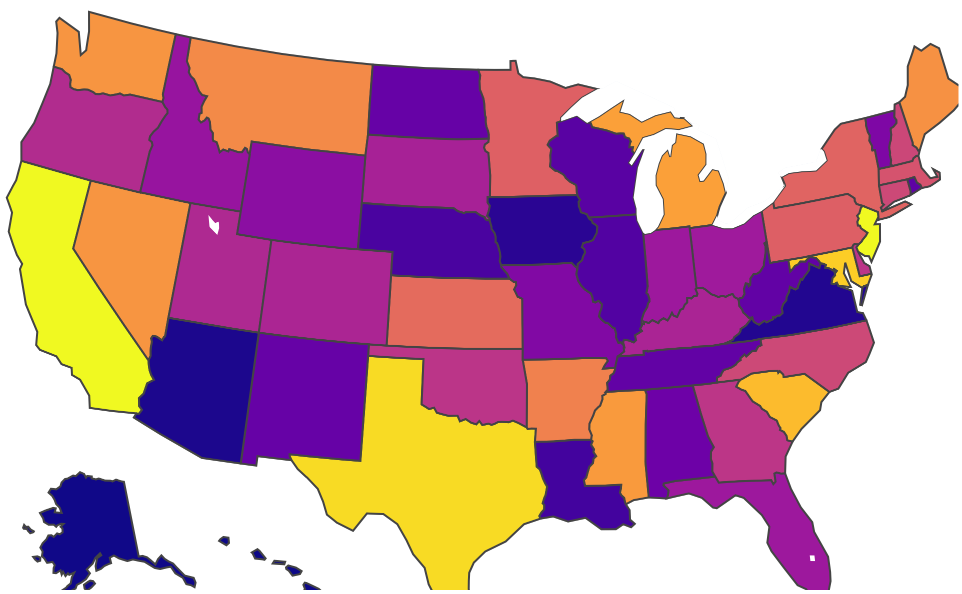
Out[18]:

| | state | churn |
|---|---|---|
| 0 | CA | 0.264706 |
| 1 | NJ | 0.264706 |
| 2 | TX | 0.250000 |
| 3 | MD | 0.242857 |
| 4 | SC | 0.233333 |
| 5 | MI | 0.219178 |
| 6 | MS | 0.215385 |
| 7 | NV | 0.212121 |
| 8 | WA | 0.212121 |
| 9 | ME | 0.209677 |
| 10 | MT | 0.205882 |
| 11 | AR | 0.200000 |
| 12 | KS | 0.185714 |
| 13 | NY | 0.180723 |
| 14 | MN | 0.178571 |
| 15 | PA | 0.177778 |
| 16 | MA | 0.169231 |
| 17 | CT | 0.162162 |
| 18 | NC | 0.161765 |
| 19 | NH | 0.160714 |
| 20 | GA | 0.148148 |
| 21 | DE | 0.147541 |
| 22 | OK | 0.147541 |
| 23 | OR | 0.141026 |
| 24 | UT | 0.138889 |
| 25 | CO | 0.136364 |
| 26 | KY | 0.135593 |
| 27 | SD | 0.133333 |
| 28 | OH | 0.128205 |
| 29 | FL | 0.126984 |
| 30 | IN | 0.126761 |
| 31 | ID | 0.123288 |

| | state | churn |
|---|---|---|
| 32 | WY | 0.116883 |
| 33 | MO | 0.111111 |
| 34 | VT | 0.109589 |
| 35 | AL | 0.100000 |
| 36 | ND | 0.096774 |
| 37 | NM | 0.096774 |
| 38 | WV | 0.094340 |
| 39 | TN | 0.094340 |
| 40 | DC | 0.092593 |
| 41 | RI | 0.092308 |
| 42 | WI | 0.089744 |
| 43 | IL | 0.086207 |
| 44 | NE | 0.081967 |
| 45 | LA | 0.078431 |
| 46 | IA | 0.068182 |
| 47 | VA | 0.064935 |
| 48 | AZ | 0.062500 |
| 49 | AK | 0.057692 |
| 50 | HI | 0.056604 |

In [19]:
```
fig = px.choropleth(state_df,
                    locations='state',
                    color='churn',
                    #color_continuous_scale='spectral_r',
                    #hover_name='state',
                    locationmode='USA-states',
                    labels={'Churn by State'},
                    scope='usa',
                    title='Churn by State')
fig.show()
```

Churn by State



In [20]:
```
import os

if not os.path.exists("charts"):
    os.mkdir("charts")
```

In [21]:
```
fig.write_image("charts/churn_by_state.png")
```

In [22]:
```
#account_length
data['account_length'].describe()
```

Out[22]:
```
count    3333.000000
mean      101.064806
std        39.822106
min         1.000000
25%        74.000000
50%       101.000000
75%       127.000000
max       243.000000
Name: account_length, dtype: float64
```

In [23]:
```
#area_code
data['area_code'].value_counts()
```

Out[23]:
```
415    1655
510     840
408     838
Name: area_code, dtype: int64
```

#### Features - International Plan and Voice Mail Plan

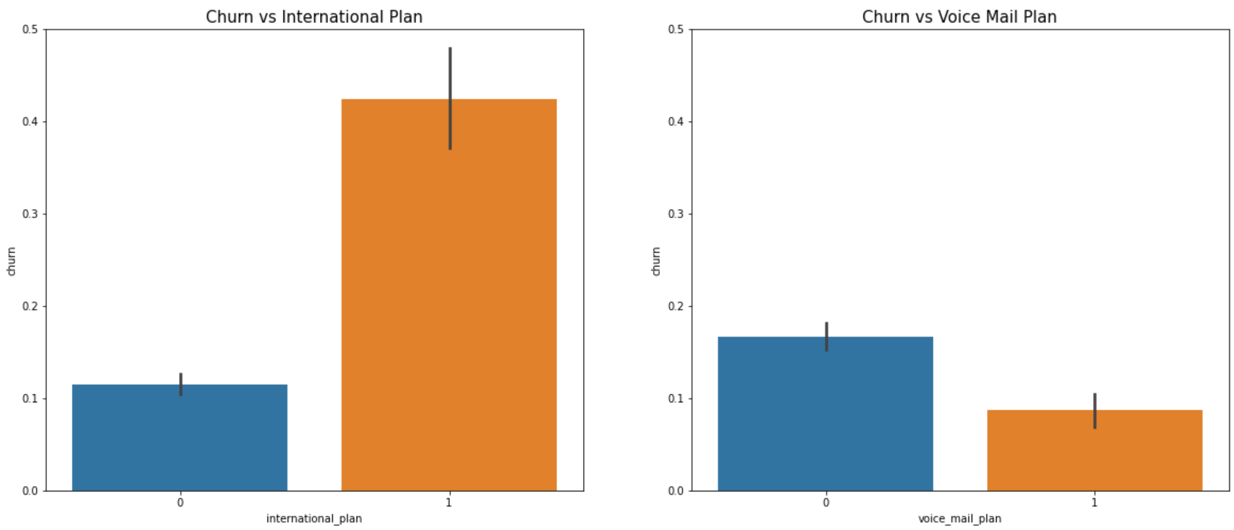
In [24]:
```
#international_plan
data['international_plan'].value_counts()
```

Out[24]:
```
0    3010
1     323
Name: international_plan, dtype: int64
```

In [25]:
```
#voice_mail_plan
data['voice_mail_plan'].value_counts()
```

Out[25]:
```
0    2411
1     922
Name: voice_mail_plan, dtype: int64
```

In [26]:
```
fig = plt.figure(figsize=(20,8))

ax1 = fig.add_subplot(121)
ax1 = sns.barplot(x='international_plan', y='churn', data=data)
ax1.set_title('Churn vs International Plan', fontsize = 15)
ax1.set_ylim((0, 0.50))

ax2 = fig.add_subplot(122)
ax2 = sns.barplot(x='voice_mail_plan', y='churn', data=data)
ax2.set_title('Churn vs Voice Mail Plan', fontsize = 15)
ax2.set_ylim((0, 0.50))

plt.savefig('charts/churn vs intl plan and voice mail plan.png')

plt.show()
```




*Comments: It seems that customers with international plan and customers without voice mail plan tend to churn. It needs further investigation.*

In [27]:
```
#number_vmail_messages
data['number_vmail_messages'].describe()
```

Out[27]:
```
count    3333.000000
mean        8.099010
std        13.688365
min         0.000000
25%         0.000000
50%         0.000000
75%        20.000000
max        51.000000
Name: number_vmail_messages, dtype: float64
```

In [28]:
```
#number of customers without voice mail
len(data.loc[data['number_vmail_messages'] == 0])
```

Out[28]:
```
2411
```

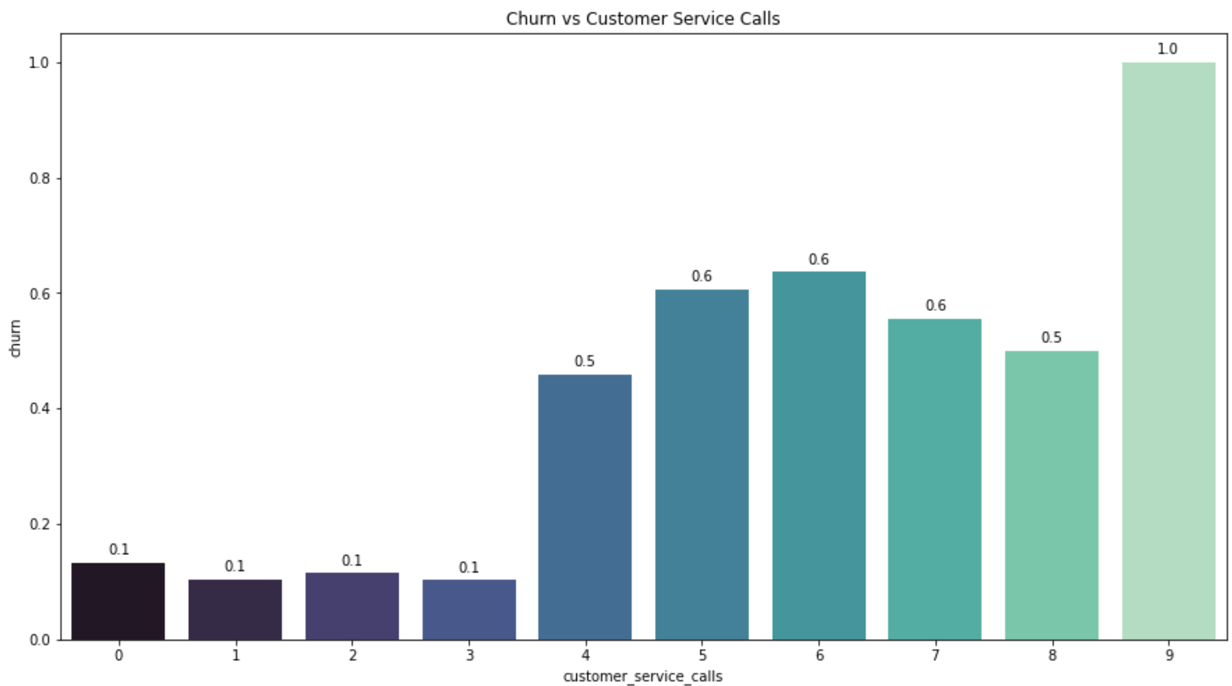
*Comments: The number of customers without voice_mail_plan is the same as the number of* *customers without voice mails.*

#### Features - Customer Service Calls

In [29]:
```
#customer_service_calls
data['customer_service_calls'].value_counts()
```

Out[29]:
```
1    1181
2     759
0     697
3     429
4     166
5      66
6      22
7       9
9       2
8       2
Name: customer_service_calls, dtype: int64
```

In [30]:
```
plt.figure(figsize=(15, 8))
ax = sns.barplot(x='customer_service_calls', y='churn', data=data, palette=
# Add annotations to bars
for p in ax.patches:
    ax.annotate(format(p.get_height(), '.1f'),
                (p.get_x() + p.get_width() / 2., p.get_height()),
                ha = 'center', va = 'center', xytext = (0, 9), textcoords =
plt.title('Churn vs Customer Service Calls')

plt.savefig('charts/churn vs customer service calls.png')

plt.show()
```




*Comments: The customers with more customer service calls are more likely to churn.*

### Features - Charges

In [31]:
```
#total_domestic_charge = total_day_charge + total_eve_charge + total_night_
data['total_domestic_charge'] = data.loc[:,['total_day_charge', 'total_eve_
                                        'total_night_charge']].sum(axis=1)
```

In [32]:
```
#monthly_charge = total_domestic_charge + total_intl_charge
data['monthly_charge'] = data['total_domestic_charge'] + data['total_intl_c
```

In [33]:
```
#total_charge = montly_charge * account_length
data['total_charge'] = data['monthly_charge'] * data['account_length']
```

In [34]:
```
charge_cols = ['total_day_charge','total_eve_charge','total_night_charge',
               'total_domestic_charge', 'total_intl_charge', 'monthly_charg
data[charge_cols].describe()
```

Out[34]:

| | total_day_charge | total_eve_charge | total_night_charge | total_domestic_charge | total_intl_charg |
|---|---|---|---|---|---|
| **count** | 3333.000000 | 3333.000000 | 3333.000000 | 3333.000000 | 3333.00000 |
| **mean** | 30.562307 | 17.083540 | 9.039325 | 56.685173 | 2.76458 |
| **std** | 9.259435 | 4.310668 | 2.275873 | 10.487816 | 0.75377 |
| **min** | 0.000000 | 0.000000 | 1.040000 | 19.980000 | 0.00000 |
| **25%** | 24.430000 | 14.160000 | 7.520000 | 49.590000 | 2.30000 |
| **50%** | 30.500000 | 17.120000 | 9.050000 | 56.630000 | 2.78000 |
| **75%** | 36.790000 | 20.000000 | 10.590000 | 63.650000 | 3.27000 |
| **max** | 59.640000 | 30.910000 | 17.770000 | 92.560000 | 5.40000 |

In [35]:
```
avg_day_charge = data['total_day_charge'] / data['total_day_minutes']
avg_day_charge.mean()
```

Out[35]: 0.1700032343415996

In [36]:
```
avg_eve_charge = data['total_eve_charge'] / data['total_eve_minutes']
avg_eve_charge.mean()
```

Out[36]: 0.08500117298813872

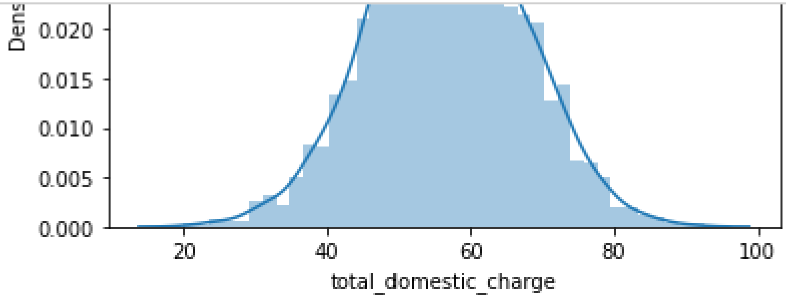
In [37]:
```
avg_night_charge = data['total_night_charge'] / data['total_night_minutes']
avg_night_charge.mean()
```

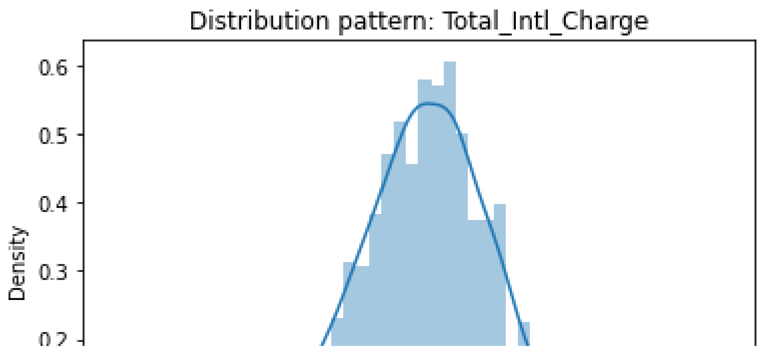
Out[37]: 0.045000345702212126

In [38]:
```
avg_intl_charge = data['total_intl_charge'] / data['total_intl_minutes']
avg_intl_charge.mean()
```

Out[38]: 0.27005654558216224

In [39]:
```
for col in charge_cols:
    sns.distplot(data[col])
    plt.title(f'Distribution pattern: {col.title()}')
    plt.show()
```



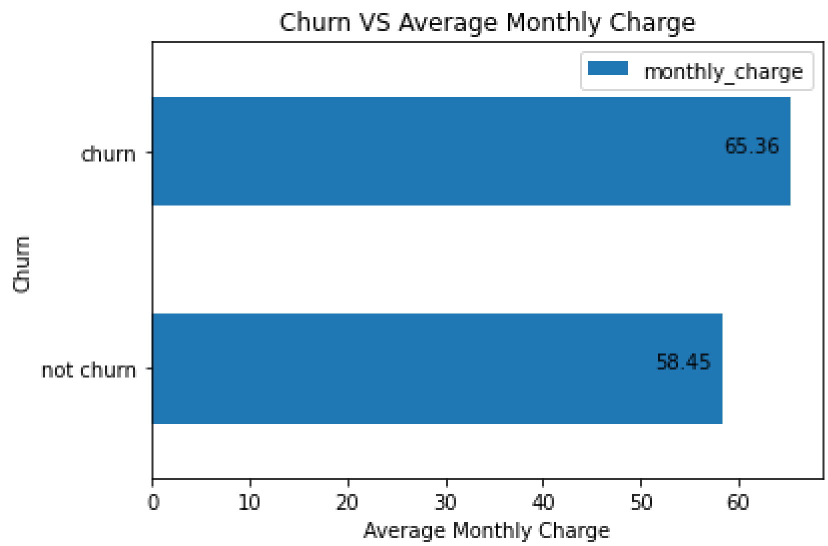




In [40]:
```
#Monthly Charge
ds_mc = data.groupby(['churn'])['monthly_charge'].mean()
ds_mc.rename({0.0: 'not churn', 1.0: 'churn'}, inplace=True)
df_mc = pd.DataFrame(ds_mc)
df_mc
```

Out[40]:

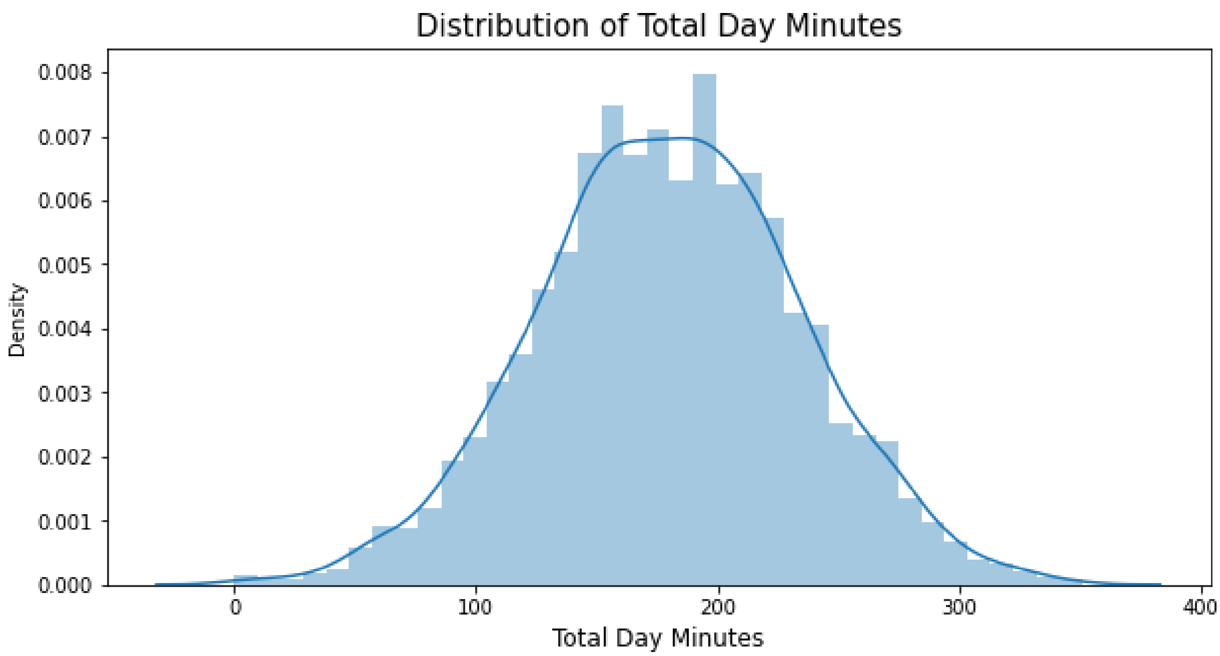
| | monthly_charge |
|---|---|
| **churn** | |
| **not churn** | 58.448807 |
| **churn** | 65.355963 |

In [41]:
```
df_mc.plot.barh()
x = df_mc['monthly_charge'].round(2)
y = df_mc.index
plt.title('Churn VS Average Monthly Charge')
plt.xlabel('Average Monthly Charge')
plt.ylabel('Churn')

for index, value in enumerate(x):
    plt.text(value-7, index, str(value))

plt.savefig('charts/churn vs average monthly charge.png')

plt.show()
```



In [42]:

```
#Total Day Minutes
fig, ax = plt.subplots(figsize=(10,5))
plt.title('Distribution of Total Day Minutes', fontsize = 15)
sns.distplot(data['total_day_minutes'], ax = ax)
ax.tick_params(axis = 'both', labelsize = 10)
plt.xlabel('Total Day Minutes', fontsize = 12)
plt.savefig('charts/Distribution of Total Day Minutes.png')
plt.show()
```



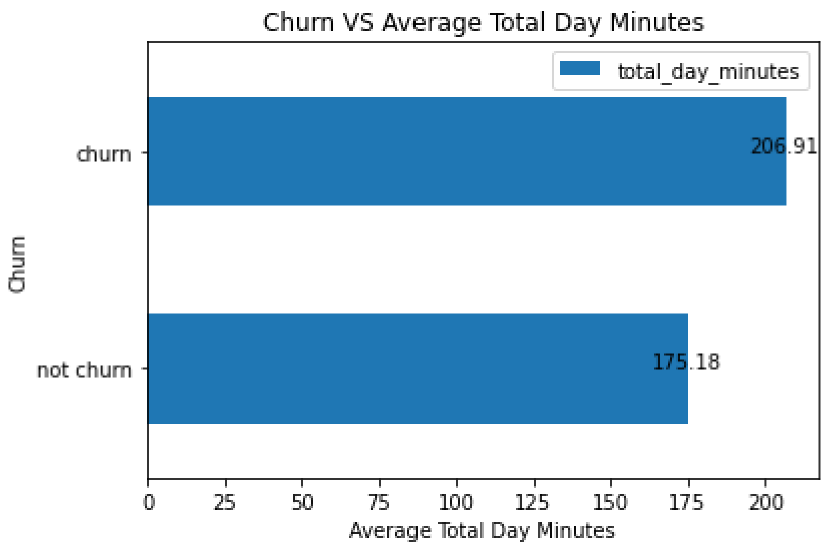

In [43]:

```
ds_dm = data.groupby(['churn'])['total_day_minutes'].mean()
ds_dm.rename({0.0: 'not churn', 1.0: 'churn'}, inplace=True)
df_dm = pd.DataFrame(ds_dm)
df_dm
```

Out[43]:

| | total_day_minutes |
|---|---|
| **churn** | |
| **not churn** | 175.175754 |
| **churn** | 206.914079 |

In [44]:
```
df_dm.plot.barh()
x = df_dm['total_day_minutes'].round(2)
y = df_dm.index
plt.title('Churn VS Average Total Day Minutes')
plt.xlabel('Average Total Day Minutes')
plt.ylabel('Churn')

for index, value in enumerate(x):
    plt.text(value-12, index, str(value))

plt.savefig('charts/churn vs average total day minutes.png')

plt.show()
```



In [45]:
```
#account length
ds_al = data.groupby(['churn'])['account_length'].mean()
ds_al.rename({0.0: 'not churn', 1.0: 'churn'}, inplace=True)
df_al = pd.DataFrame(ds_al)
df_al
```

Out[45]:

| | account_length |
|---|---|
| **churn** | |
| **not churn** | 100.793684 |
| **churn** | 102.664596 |

# Build Classification Models

## 1. Set target variable, features and train/test split

### One-Hot Encoding on State

In [46]:
```
df = pd.get_dummies(data, drop_first = True)
```

In [47]:
```
df.head()
```

Out[47]:

| | account_length | area_code | international_plan | voice_mail_plan | number_vmail_messages | total_day |
|---|---|---|---|---|---|---|
| **0** | 128 | 415 | 0 | 1 | 25 | |
| **1** | 107 | 415 | 0 | 1 | 26 | |
| **2** | 137 | 415 | 0 | 0 | 0 | |
| **3** | 84 | 408 | 1 | 0 | 0 | |
| **4** | 75 | 415 | 1 | 0 | 0 | |

5 rows × 72 columns

In [48]:
```
df.columns
```

Out[48]:
```
Index(['account_length', 'area_code', 'international_plan', 'voice_mail_p
lan',
       'number_vmail_messages', 'total_day_minutes', 'total_day_calls',
       'total_day_charge', 'total_eve_minutes', 'total_eve_calls',
       'total_eve_charge', 'total_night_minutes', 'total_night_calls',
       'total_night_charge', 'total_intl_minutes', 'total_intl_calls',
       'total_intl_charge', 'customer_service_calls', 'churn',
       'total_domestic_charge', 'monthly_charge', 'total_charge', 'state_
AL',
       'state_AR', 'state_AZ', 'state_CA', 'state_CO', 'state_CT', 'state
_DC',
       'state_DE', 'state_FL', 'state_GA', 'state_HI', 'state_IA', 'state
_ID',
       'state_IL', 'state_IN', 'state_KS', 'state_KY', 'state_LA', 'state
_MA',
       'state_MD', 'state_ME', 'state_MI', 'state_MN', 'state_MO', 'state
_MS',
       'state_MT', 'state_NC', 'state_ND', 'state_NE', 'state_NH', 'state
_NJ',
       'state_NM', 'state_NV', 'state_NY', 'state_OH', 'state_OK', 'state
_OR',
       'state_PA', 'state_RI', 'state_SC', 'state_SD', 'state_TN', 'state
_TX',
       'state_UT', 'state_VA', 'state_VT', 'state_WA', 'state_WI', 'state
_WV',
       'state_WY'],
      dtype='object')
```

In [49]:
```
X = df.drop(['churn'], axis = 1)
y = df['churn']

X_train, X_test, y_train, y_test = train_test_split(X, y, random_state=42,
```

In [50]:
```
print('Training Set: ')
print(y_train.value_counts())
print('Normalized:')
print(y_train.value_counts(normalize=True))
print('\n')
# Test set
print('Test Set')
print(y_test.value_counts())
print('Normalized:')
print(y_test.value_counts(normalize=True))
```

```
Training Set:
0.0    2137
1.0     362
Name: churn, dtype: int64
Normalized:
0.0    0.855142
1.0    0.144858
Name: churn, dtype: float64


Test Set
0.0    713
1.0    121
Name: churn, dtype: int64
Normalized:
0.0    0.854916
1.0    0.145084
Name: churn, dtype: float64
```

## 2. Baseline model - Logistic Regression

In [51]:
```
#Instantiate a LogistcRegression with random_state=42
draft_model = LogisticRegression(random_state=42)
```

In [52]:
```
#Use cross_val_score with scoring='neg_log_loss' to evaluate the model on X
draft_neg_log_loss_cv = cross_val_score(draft_model, X_train, y_train,
                                         scoring='neg_log_loss')
draft_log_loss = -(draft_neg_log_loss_cv.mean())
draft_log_loss
```

Out[52]:
```
0.3879326574407237
```

In [53]:
```
#If we had a model that just chose 0 (the majority class) every time, the l
log_loss(y_train, np.zeros(len(y_train)))
```

Out[53]:
```
5.003216108426439
```

*Comments: Loss is a metric where lower is better, so our baseline model is clearly an improvement over just guessing the majority class every time, even though it is difficult to interpret log loss.*

In [54]:
```
#Write a custom cross validation function with StratifiedKFlod
#Essentially StratifiedKFold is just providing the information you need to
#inside of X_train. Then there is other logic within cross_val_score to fit

def custom_cross_val_score(estimator, X, y):
    # Create a list to hold the scores from each fold
    kfold_train_scores = np.ndarray(5)
    kfold_val_scores = np.ndarray(5)

    neg_log_loss = make_scorer(log_loss, greater_is_better=False, needs_pro

    # Instantiate a splitter object and loop over its result
    kfold = StratifiedKFold(n_splits=5)
    for fold, (train_index, val_index) in enumerate(kfold.split(X, y)):
        # Extract train and validation subsets using the provided indices
        X_t, X_val = X.iloc[train_index], X.iloc[val_index]
        y_t, y_val = y.iloc[train_index], y.iloc[val_index]

        # Instantiate StandardScaler
        scaler = StandardScaler()
        # Fit and transform X_t
        X_t_scaled = scaler.fit_transform(X_t)
        # Transform X_val
        X_val_scaled = scaler.transform(X_val)

        # Instantiate SMOTE with random_state=42 and sampling_strategy=0.28
        sm = SMOTE(random_state=42, sampling_strategy=0.28)
        # Fit and transform X_t_scaled and y_t using sm
        X_t_oversampled, y_t_oversampled = sm.fit_resample(X_t_scaled, y_t)

        # Clone the provided model and fit it on the train subset
        temp_model = clone(estimator)
        temp_model.fit(X_t_oversampled, y_t_oversampled)

        # Evaluate the provided model on the train and validation subsets
        neg_log_loss_score_train = neg_log_loss(temp_model, X_t_oversampled
        neg_log_loss_score_val = neg_log_loss(temp_model, X_val_scaled, y_v
        kfold_train_scores[fold] = neg_log_loss_score_train
        kfold_val_scores[fold] = neg_log_loss_score_val

    return kfold_train_scores, kfold_val_scores
```

In [55]:
```
model_with_preprocessing = LogisticRegression(random_state=42, class_weight
preprocessed_train_scores, preprocessed_neg_log_loss_cv = \
custom_cross_val_score(model_with_preprocessing, X_train, y_train)
```

In [56]:
```
preprocessed_train_scores
```

Out[56]:
```
array([-0.45007005, -0.4395607 , -0.44892305, -0.44707121, -0.42890291])
```

In [57]:
```
preprocessed_neg_log_loss_cv
```

Out[57]:
```
array([-0.34819567, -0.40122779, -0.35430651, -0.35480173, -0.40464214])
```

In [58]:
```
custom_cross_val_score(model_with_preprocessing, X_train, y_train)
```

Out[58]:
```
(array([-0.45007005, -0.4395607 , -0.44892305, -0.44707121, -0.4289029
1]),
 array([-0.34819567, -0.40122779, -0.35430651, -0.35480173, -0.4046421
4]))
```

In [59]:
```
preprocessed_log_loss = - (preprocessed_neg_log_loss_cv.mean())
preprocessed_log_loss
```

Out[59]:
```
0.37263476677167784
```

In [60]:
```
print(f'Log loss of Draft Model is', round(-draft_neg_log_loss_cv.mean(),4)
print(f'Log loss of Preprocessed Model is', round(-preprocessed_neg_log_los
```

```
Log loss of Draft Model is 0.3879
Log loss of Preprocessed Model is 0.3726
```

*Comments: Looks like our preprocessing with StandardScaler and SMOTE has provided some improvement over the very first draft model!*

In [61]:
```
print("Train:      ", -preprocessed_train_scores)
print("Validation:", -preprocessed_neg_log_loss_cv)
```

```
Train:      [0.45007005 0.4395607  0.44892305 0.44707121 0.42890291]
Validation: [0.34819567 0.40122779 0.35430651 0.35480173 0.40464214]
```

*Comments: While SMOTE makes it somewhat challenging to compare these numbers directly, it does not appear that we are overfitting. Overfitting would mean getting significantly better scores on the training data than the validation data.*

In [62]:
```
model_with_preprocessing.get_params()
```

Out[62]:
```
{'C': 1.0,
 'class_weight': {1: 0.28},
 'dual': False,
 'fit_intercept': True,
 'intercept_scaling': 1,
 'l1_ratio': None,
 'max_iter': 100,
 'multi_class': 'auto',
 'n_jobs': None,
 'penalty': 'l2',
 'random_state': 42,
 'solver': 'lbfgs',
 'tol': 0.0001,
 'verbose': 0,
 'warm_start': False}
```

### Reduce regularization

In [63]:
```
# instantiate a LogisticRegression model with lower regularization
model_less_regularization = LogisticRegression(
    random_state=42,
    class_weight={1: 0.28},
    C=1e5
)
```

In [64]:
```
# Check variable type
assert type(model_less_regularization) == LogisticRegression

# Check params
assert model_less_regularization.get_params()["random_state"] == 42
assert model_less_regularization.get_params()["class_weight"] == {1: 0.28}
assert model_less_regularization.get_params()["C"] != 1.0
```

In [65]:
```
less_regularization_train_scores, less_regularization_val_scores = custom_c
    model_less_regularization,
    X_train,
    y_train
)

print("Previous Model")
print("Train average:     ", -preprocessed_train_scores.mean())
print("Validation average:", -preprocessed_neg_log_loss_cv.mean())
print("Current Model")
print("Train average:     ", -less_regularization_train_scores.mean())
print("Validation average:", -less_regularization_val_scores.mean())
```

```
Previous Model
Train average:      0.4429055851819033
Validation average: 0.37263476677167784
Current Model
Train average:      0.44178653201297
Validation average: 0.38898240401938755
```

### Alternative Solver

In [66]:
```
print("solver:", model_less_regularization.get_params()["solver"])
print("penalty:", model_less_regularization.get_params()["penalty"])
```

```
solver: lbfgs
penalty: l2
```

In [67]:
```
# the only models that support L1 or elastic net penalties are liblinear ar
# liblinear is going to be quite slow with the size of our dataset, so we w

model_alternative_solver = LogisticRegression(
    random_state=42,
    class_weight={1: 0.28},
    C=1e5,
    solver="saga",
    penalty="elasticnet",
    l1_ratio=0.5
)

alternative_solver_train_scores, alternative_solver_val_scores = custom_crc
    model_alternative_solver,
    X_train,
    y_train
)

print("Previous Model (Less Regularization)")
print("Train average:      ", -less_regularization_train_scores.mean())
print("Validation average:", -less_regularization_val_scores.mean())
print("Current Model")
print("Train average:      ", -alternative_solver_train_scores.mean())
print("Validation average:", -alternative_solver_val_scores.mean())
```

```
Previous Model (Less Regularization)
Train average:      0.44178653201297
Validation average: 0.38898240401938755
Current Model
Train average:      0.4422776314871618
Validation average: 0.3756181789946792
```

#### Adjusting Gradient Descent Parameters

In [68]:
```
model_more_iterations = LogisticRegression(
    random_state=42,
    class_weight={1: 0.28},
    C=1e5,
    solver="saga",
    penalty="elasticnet",
    l1_ratio=0.5,
    max_iter=2000
)

more_iterations_train_scores, more_iterations_val_scores = custom_cross_val
    model_more_iterations,
    X_train,
    y_train
)

print("Previous Model (Less Regularization)")
print("Train average:      ", -less_regularization_train_scores.mean())
print("Validation average:", -less_regularization_val_scores.mean())
print("Previous Model with Solver")
print("Train average:      ", -alternative_solver_train_scores.mean())
print("Validation average:", -alternative_solver_val_scores.mean())
print("Current Model")
print("Train average:      ", -more_iterations_train_scores.mean())
print("Validation average:", -more_iterations_val_scores.mean())
```

```
Previous Model (Less Regularization)
Train average:      0.44178653201297
Validation average: 0.38898240401938755
Previous Model with Solver
Train average:      0.4422776314871618
Validation average: 0.3756181789946792
Current Model
Train average:      0.44222827836175227
Validation average: 0.37773423494482605
```

### Determine the baseline model

In [69]:
```
baseline_model = model_less_regularization
```

### Preprocessing the full dataset

In [70]:
```
# Instantiate StandardScaler
scaler = StandardScaler()
# Fit and transform X_train
X_train_scaled = scaler.fit_transform(X_train)
# Transform X_test
X_test_scaled = scaler.transform(X_test)

# Instantiate SMOTE with random_state=42
sm = SMOTE(random_state=42) #sampling_strategy=0.28
# Fit and transform X_train_scaled and y_train using sm
X_train_oversampled, y_train_oversampled = sm.fit_resample(X_train_scaled,
```

#### Fitting the baseline model on train dataset

In [71]:
```
baseline_model.fit(X_train_oversampled, y_train_oversampled)
```

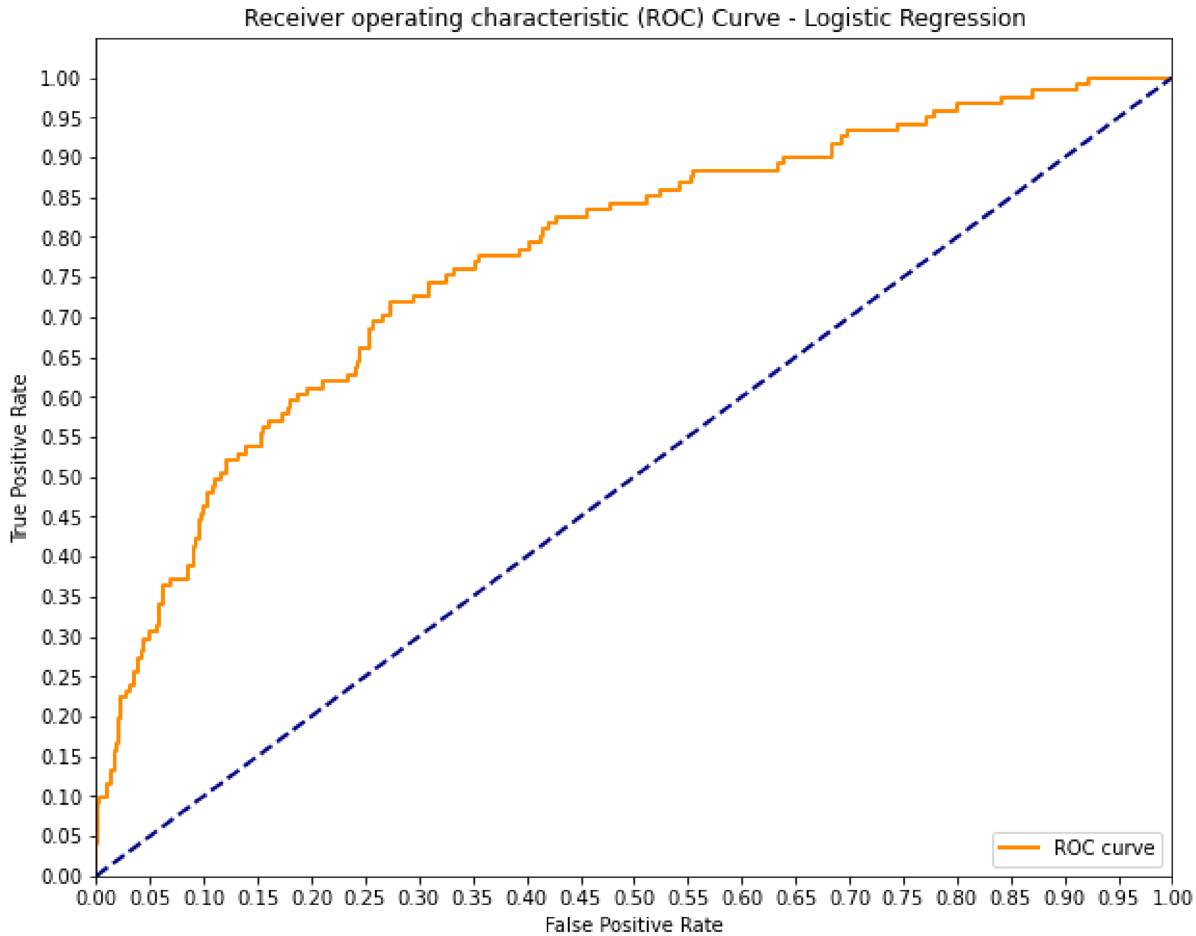
Out[71]:
```
LogisticRegression(C=100000.0, class_weight={1: 0.28}, random_state=42)
```

#### Evaluating the baseline model on test dataset

In [72]:

```
# Probability scores for test set
y_score = baseline_model.fit(X_train_oversampled, y_train_oversampled).deci

# False positive rate and true positive rate
fpr, tpr, thresholds = roc_curve(y_test, y_score)

# Print AUC

print('AUC: {}'.format(auc(fpr, tpr)))

# Plot the ROC curve
plt.figure(figsize=(10, 8))
lw = 2
plt.plot(fpr, tpr, color = 'darkorange',
         lw=lw, label='ROC curve')
plt.plot([0,1], [0,1], color = 'navy', lw=lw, linestyle ='--')
plt.xlim([0.0,1.0])
plt.ylim([0.0, 1.05])
plt.yticks([i/20.0 for i in range(21)])
plt.xticks([i/20.0 for i in range(21)])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic (ROC) Curve - Logistic Regress
plt.legend(loc='lower right')
plt.savefig('charts/Receiver operating characteristic (ROC) Curve - Logisti
plt.show()
```

```
AUC: 0.7764074507667522
```




In [73]:
```
#log loss
log_loss_bs = log_loss(y_test, baseline_model.predict_proba(X_test_scaled))
log_loss_bs
```
Out[73]: 0.37722779834958026

In [74]:
```
#acuracy score
accuracy_bs = accuracy_score(y_test, baseline_model.predict(X_test_scaled))
accuracy_bs
```
Out[74]: 0.8369304556354916

In [75]:
```
#precision score
precision_bs = precision_score(y_test, baseline_model.predict(X_test_scalec
precision_bs
```
Out[75]: 0.42857142857142855

In [76]:
```
#recall score
recall_bs = recall_score(y_test, baseline_model.predict(X_test_scaled))
recall_bs
```
Out[76]: 0.371900826446281

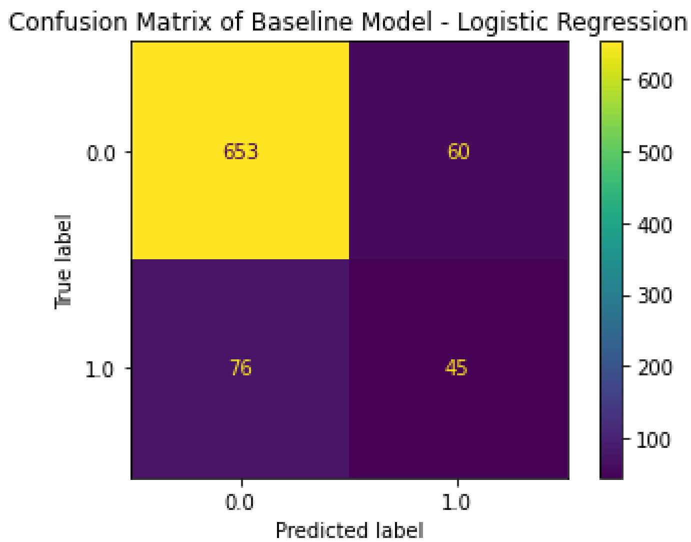
In [77]:
```
#f1 score
f1_bs = f1_score(y_test, baseline_model.predict(X_test_scaled), average='we
f1_bs
```

Out[77]: 0.832062767789279

In [78]:
```
confusion_matrix(y_test, baseline_model.predict(X_test_scaled))
```

Out[78]:
```
array([[653,  60],
       [ 76,  45]])
```

In [79]:
```
plot_confusion_matrix(baseline_model, X_test_scaled, y_test)
plt.title('Confusion Matrix of Baseline Model - Logistic Regression')
plt.savefig('charts/Confusion Matrix of Baseline Model - Logistic Regressic
plt.show()
```




## 3. Build a classifier using supervised machine learning algorithms

Because we have imbalanced classes (84%:16%) we want to focus more on how well the model performed on the Churn cases (the minority class).

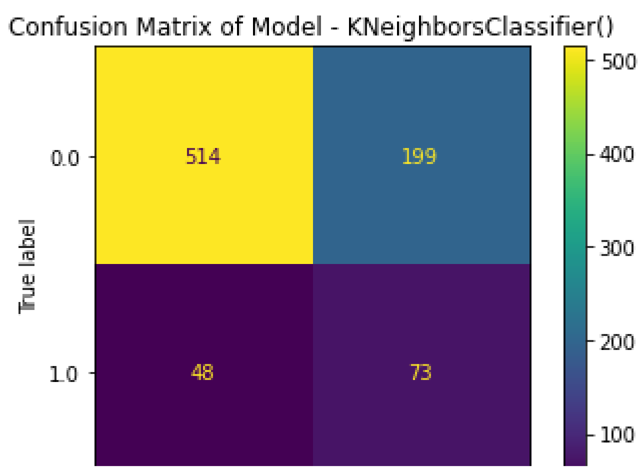
- The F1 Score is the harmonic mean of Precision and Recall. It helps give us a balanced idea of how the model is performing on the Churn class.
- The Recall Score is mainly focusing on the customers who actually churn but we fail to predict.
- AUC refers to Area Under the Receiver Operating Characteristic curves. Perfect classifiers would have an AUC score of 1.0 while an AUC of 0.5 is deemed trivial or worthless.

Therefore, we will choose the model with the highest value of F1 Score, Recall Score and AUC.

### Model Iteration

In [80]:
```
knn = KNeighborsClassifier()
dt = DecisionTreeClassifier()
rf = RandomForestClassifier()
adaboost = AdaBoostClassifier()
gboost = GradientBoostingClassifier()
xgboost = XGBClassifier()
svm = SVC(probability=True)

models = [knn, dt, rf, adaboost, gboost, xgboost, svm]

for model in models:
    model.fit(X_train_oversampled, y_train_oversampled)
    y_score = model.predict_proba(X_test_scaled)[:, 1]
    fpr, tpr, thresholds = roc_curve(y_test, y_score)
    y_preds_test = model.predict(X_test_scaled)
    y_preds_train = model.predict(X_train_oversampled)
    print('Model:', model)
    print('Training Recall:', recall_score(y_train_oversampled, y_preds_tra
    print('Testing Recall:', recall_score(y_test, y_preds_test))
    print('Testing F1 Score', f1_score(y_test,y_preds_test))
    print('Testing AUC', auc(fpr,tpr))
    plot_confusion_matrix(model, X_test_scaled, y_test)
    plt.title(f'Confusion Matrix of Model - {model}')
    plt.show()
    print('\n ---------------------------------- \n')
```

```
Model: KNeighborsClassifier()
Training Recall: 0.99719232569022
Testing Recall: 0.6033057851239669
Testing F1 Score 0.3715012722646311
Testing AUC 0.689352404576171
```




*Comments: According to the recall score, weighted f1 score and especially the amount of AUC, XGBoost is the best classifier we want to choose for the churn prediction model.*

## 4. Best Classifier - XGBoost (eXtreme Gradient Boosting)

In [81]:
```
clf = XGBClassifier()

clf.fit(X_train_oversampled, y_train_oversampled)
```

Out[81]: XGBClassifier()

#### Evaluation on XGBoost model before tuning

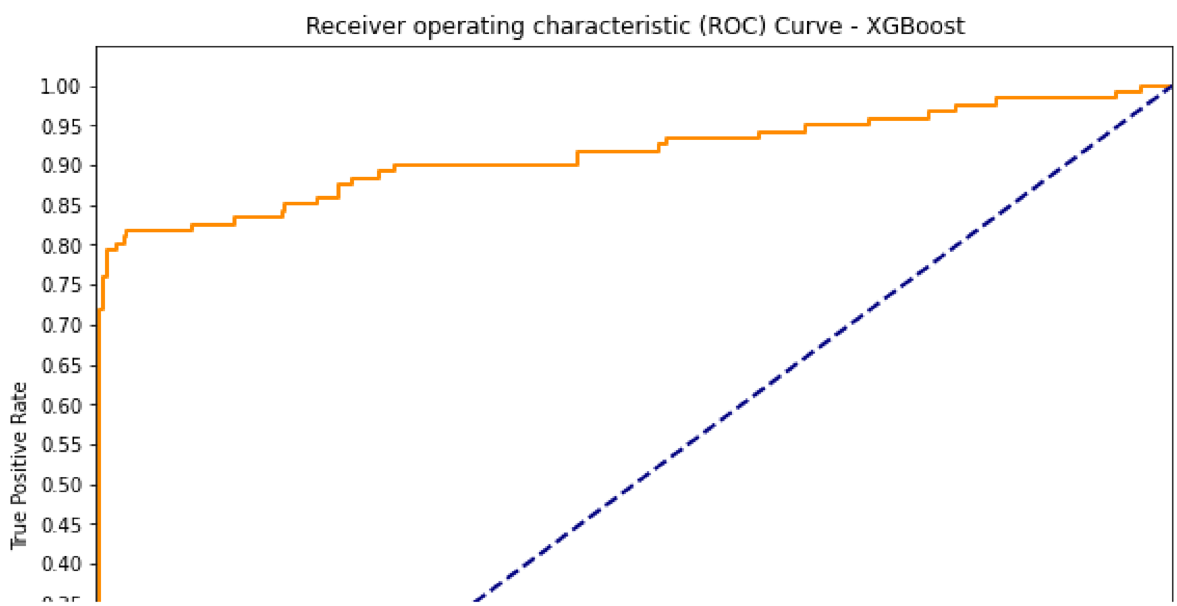
In [82]:
```
y_preds_test = clf.predict(X_test_scaled)

# Probability scores for test set
y_score = clf.predict_proba(X_test_scaled)[:, 1]

# False positive rate and true positive rate
fpr, tpr, thresholds = roc_curve(y_test, y_score)
```

In [83]:
```
auc_xgb = auc(fpr, tpr)
auc_xgb
```

Out[83]: 0.9128464293579682

In [84]:
```
# Print AUC
print('AUC: {}'.format(auc_xgb))

# Plot the ROC curve
plt.figure(figsize=(10, 8))
lw = 2
plt.plot(fpr, tpr, color = 'darkorange',
         lw=lw, label='ROC curve')
plt.plot([0,1], [0,1], color = 'navy', lw=lw, linestyle ='--')
plt.xlim([0.0,1.0])
plt.ylim([0.0, 1.05])
plt.yticks([i/20.0 for i in range(21)])
plt.xticks([i/20.0 for i in range(21)])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic (ROC) Curve - XGBoost')
plt.legend(loc='lower right')
plt.show()
```

```
AUC: 0.9128464293579682
```




In [85]:
```
#log loss
log_loss_xgb = log_loss(y_test, y_score)
log_loss_xgb
```

Out[85]: 0.18669417489301057

In [86]:
```
#acuracy score
accuracy_xgb = accuracy_score(y_test, y_preds_test)
accuracy_xgb
```

Out[86]: 0.9580335731414868

In [87]:
```
#precision score
precision_xgb = precision_score(y_test, y_preds_test)
precision_xgb
```

Out[87]: 0.9056603773584906

In [88]:
```
#recall score
recall_xgb = recall_score(y_test, y_preds_test)
recall_xgb
```

Out[88]: 0.7933884297520661

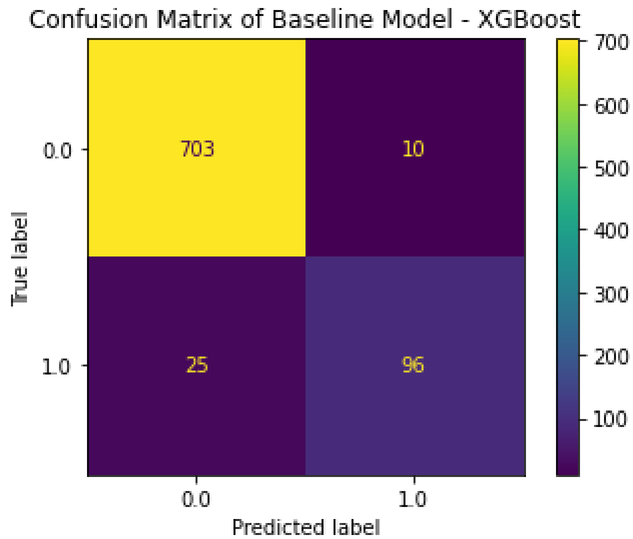
In [89]:
```
#f1 score
f1_xgb = f1_score(y_test, y_preds_test, average='weighted')
f1_xgb
```

Out[89]: 0.9568654406449436

In [90]:
```
confusion_matrix(y_test, y_preds_test)
```

Out[90]:
```
array([[703,  10],
       [ 25,  96]])
```

In [91]:
```
plot_confusion_matrix(clf, X_test_scaled, y_test)
plt.title('Confusion Matrix of Baseline Model - XGBoost')
plt.show()
```




**Tuning XGBoost with GridSearchCV**

In [92]:
```
param_grid = {
    'learning_rate': [0.1, 0.2],
    'max_depth': [6],
    'min_child_weight': [1, 2],
    'subsample': [0.5, 0.7],
    'n_estimators': [100],
}
```

In [93]:
```
grid_clf = GridSearchCV(clf, param_grid, scoring='recall', cv=None, n_jobs=
grid_clf.fit(X_train_oversampled, y_train_oversampled)

best_parameters = grid_clf.best_params_

print('Grid Search found the following optimal parameters: ')
for param_name in sorted(best_parameters.keys()):
    print('%s: %r' % (param_name, best_parameters[param_name]))
```

```
Grid Search found the following optimal parameters:
learning_rate: 0.1
max_depth: 6
min_child_weight: 1
n_estimators: 100
subsample: 0.7
```

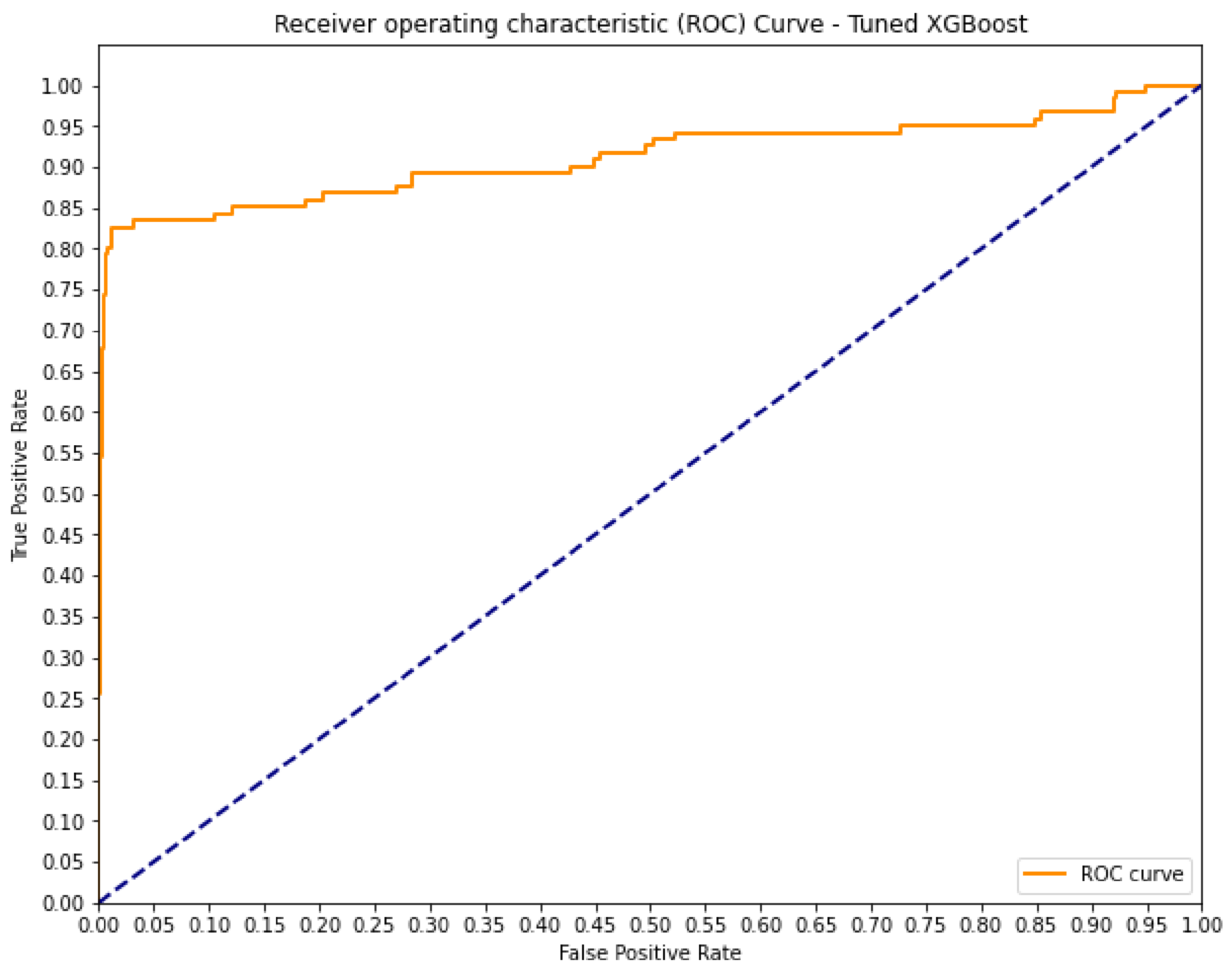
In [94]:
```
training_preds = grid_clf.predict(X_train_oversampled)
tuned_y_preds = grid_clf.predict(X_test_scaled)
tuned_y_score = grid_clf.predict_proba(X_test_scaled)[:, 1]
fpr_tuned, tpr_tuned, thresholds_tuned = roc_curve(y_test, tuned_y_score)
```

In [95]:
```
auc_xgb_tuned = auc(fpr_tuned, tpr_tuned)
auc_xgb_tuned
```

Out[95]:
```
0.9116177714928193
```

In [96]:
```
# Print AUC
print('AUC: {}'.format(auc_xgb_tuned))

# Plot the ROC curve
plt.figure(figsize=(10, 8))
lw = 2
plt.plot(fpr_tuned, tpr_tuned, color = 'darkorange',
         lw=lw, label='ROC curve')
plt.plot([0,1], [0,1], color = 'navy', lw=lw, linestyle ='--')
plt.xlim([0.0,1.0])
plt.ylim([0.0, 1.05])
plt.yticks([i/20.0 for i in range(21)])
plt.xticks([i/20.0 for i in range(21)])
plt.xlabel('False Positive Rate')
plt.ylabel('True Positive Rate')
plt.title('Receiver operating characteristic (ROC) Curve - Tuned XGBoost')
plt.legend(loc='lower right')
plt.savefig('charts/Receiver operating characteristic (ROC) Curve - Tuned X
plt.show()
```

```
AUC: 0.9116177714928193
```



In [97]:
```
f1_xgb_tuned = f1_score(y_test, tuned_y_preds, average='weighted')
f1_xgb_tuned
```

Out[97]:
```
0.9616444779034706
```

In [98]:
```
accuracy_xgb_tuned = accuracy_score(y_test, tuned_y_preds)
accuracy_xgb_tuned
```

Out[98]:
```
0.9628297362110312
```

In [99]:
```
precision_xgb_tuned = precision_score(y_test, tuned_y_preds)
precision_xgb_tuned
```

Out[99]:
```
0.9326923076923077
```

In [100]:
```
recall_xgb_tuned = recall_score(y_test, tuned_y_preds)
recall_xgb_tuned
```

Out[100]:
```
0.8016528925619835
```

In [101]:
```
print('Using our tuned XGBoost classiciation model, we will miss {}% of cus
who will soon-to-churn'.format(round(((1 - recall_xgb_tuned) * 100), 2)))
```

```
Using our tuned XGBoost classiciation model, we will miss 19.83% of custo
mers who will soon-to-churn
```

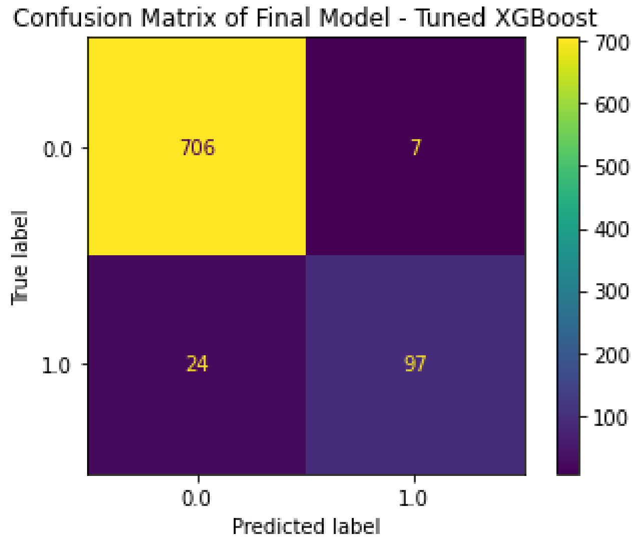
In [102]:
```
print(classification_report(y_test,tuned_y_preds))
```

```
              precision    recall  f1-score   support

         0.0       0.97      0.99      0.98       713
         1.0       0.93      0.80      0.86       121

    accuracy                           0.96       834
   macro avg       0.95      0.90      0.92       834
weighted avg       0.96      0.96      0.96       834
```

In [103]:
```
confusion_matrix(y_test, tuned_y_preds)
```

Out[103]:
```
array([[706,   7],
       [ 24,  97]])
```

In [104]:
```
plot_confusion_matrix(grid_clf, X_test_scaled, y_test)
plt.title('Confusion Matrix of Final Model - Tuned XGBoost')
plt.savefig('charts/Confusion Matrix of Final Model - Tuned XGBoost.png')
plt.show()
```




#### Feature Importance

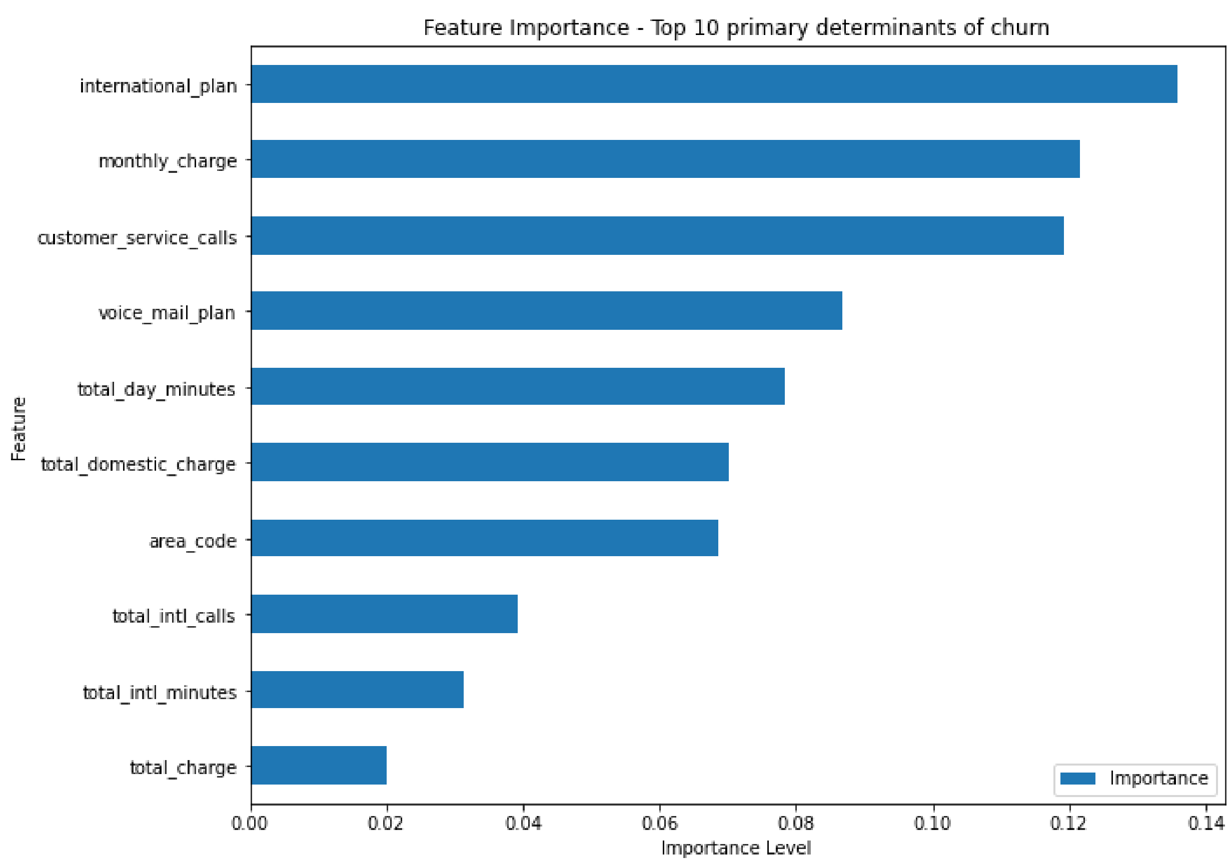
In [105]:
```
importances = list(zip(clf.feature_importances_, X.columns))
df_fi = pd.DataFrame(importances)
```

In [106]:
```
df_fi.columns = ['Importance', 'Feature']
df_fi.set_index('Feature', inplace=True)
df_fi.sort_values('Importance', ascending=False, inplace=True)
df_fi.head(5)
```

Out[106]:

| Feature | Importance |
|---|---|
| international_plan | 0.135862 |
| monthly_charge | 0.121580 |
| customer_service_calls | 0.119144 |
| voice_mail_plan | 0.086774 |
| total_day_minutes | 0.078404 |

In [107]:
```
fig = plt.figure(figsize=(10,8))
ax = fig.add_subplot()

df_fi.head(10).sort_values(by='Importance', ascending=True).plot(y='Importa
ax.set_title('Feature Importance - Top 10 primary determinants of churn')
ax.set_xlabel('Importance Level')
ax.set_ylabel('Feature')

plt.savefig('charts/Feature Importance - Top 10 primary determinants of chu
plt.show()
```



In [108]:
```
pd.DataFrame(df.corr().churn.sort_values(ascending = False)).head(6)
```

Out[108]:

| | churn |
|---|---|
| churn | 1.000000 |
| international_plan | 0.259852 |
| monthly_charge | 0.231549 |
| total_domestic_charge | 0.226962 |
| customer_service_calls | 0.208750 |
| total_day_minutes | 0.205151 |

In [109]:
```
pd.DataFrame(df.corr().churn.sort_values(ascending = False)).tail(5)
```

Out[109]:

| | churn |
|---|---|
| state_AZ | -0.032759 |
| state_VA | -0.034940 |
| total_intl_calls | -0.052844 |
| number_vmail_messages | -0.089728 |
| voice_mail_plan | -0.102148 |

*Comments: We noticed that the features with higher importance in our model are the ones with the higher correlation with churn in the raw datasets.*

In [110]:
```
primary_det = list(df_fi.head(5).index.str.replace('_', ' '))
primary_det
```

Out[110]:
```
['international plan',
 'monthly charge',
 'customer service calls',
 'voice mail plan',
 'total day minutes']
```

In [111]:
```
print(f"The primary determinants of the customer churn are {', '.join(prima
```

```
The primary determinants of the customer churn are international plan, mo
nthly charge, customer service calls, voice mail plan, total day minutes
```

# Conclusions

## Findings

**1. The final best classification model is XGBoost.**

- F1 Score = 96%. Generally, the higher F1 scores are generally better. F1 scores can range from 0 to 1, with 1 representing a model that perfectly classifies each observation into the correct class and 0 representing a model that is unable to classify any observation into the correct class. Therefore, our model can classifies whether the customers will churn or not. It will help SyriaTel take immediate action to retain its soon-to-churn customers.
- Accuracy Score = 96%. i.e. Our model can correctly identifies the type of customers (churn or not churn) about 96% of the time.
- Precision Score = 93%. i.e. If our model labels a given cell of customer as churn, there is about a 93% chance that he/she will actually churn and about a 7% chance that it is actually not churn.
- Recall score = 80%. i.e. If our model labels a give cell of customer as churn, there is about a 80% chance that our model will correctly label it as soon-to-churn customer and about a 20% chance that our model will incorrectly label it as not-to-churn customer.

We understand that acquiring new customers will cost more than retaining existing customers. If we mistakenly label a soon-to-churn customer as a not-to-churn customer, our client SyriaTel will incur more cost. Therefore, when we determined the best parameters of our classifier, we relied on the 'recall score'. The tuned XGBoost model has the best recall score compared with other classification models.

**2. Primary determinants of whether a customer will churn or not:**

We used the 'feature importance' function to get the primary features that determine whether a customer will churn or not. The top 5 features are whether the customer has international plan or voice mail plan, the monthly bill charges, how many customer service calls he/she had, and how long he/she generally called during the day time.

- International Plan: The customer who has international plan will tend to churn.

  The average international call charge is $0.27 / min, much higher than the domestic plan. If the competitors have less expensive international call charge, the customers with international plan will easily switch to another carrier.

- Monthly Charge: The customer who has higher monthly charge will tend to churn. The high bill is the main concern of customers who will churn.

- Customer Service Calls: The customer who called customer service frequently will tend to churn.

  It makes sense. Generally customers will call often to customer service if they have issues or complaints. These pain points during the interaction with customers are the biggest obstacles of the company to retain customers.

- Voice Mail Plan: The customer who did not enroll voice mail plan will tend to churn. Probably the voice mail is a good service/product provided by SyriaTel compared with its competitors. Customers are satisfied with the voice mail plan, so those will stay.

- Total Day Minutes: The customer who called more minutes during the day will tend to churn. It is associated with the high cost of day call. We noticed that the average call charge per minute for day, eve and night are $0.17, 0.085 and 0.045. Therefore, the customer with more day minutes call will undertake high phone bill, and they tend to churn.

As a summary, the customers with international plan, fewer voice mail, longer day calls, higher monthly bills and more customer service calls, tend to churn.

**3. Other features related to churn:**

- The customers in the following states are more easily to churn: CA, NJ, TX, MD, SC, MS, NV, WA, ME, MT and AR.
- Account length is not a determinant of churn.

## Next steps

**1. Market research on competitors**

Given the primary determinants are mostly related to high charges, SyriaTel need to conduct a market research on competitors to compare and further determine its own pricing strategy. Upon the understanding of the competitors and industry benchmark, we can get more accurate key features that determine the churn of customers.

**2. Customer experience measurement and design**

It is critical to understand the each step when customers interact with the company. The management need to understand and take action on how to measure the customer experience, how to design customer experience journey and how to establish a customer-centric culture. For example, we can dig into the content of customer service calls, to measure the customer experience through voice of customer, surveys and net promoter scores. It is critical to understand what the customers need and what are their biggest concerns when deciding the carrier.

**3. Partnership with local carriers**

The dataset we studied is from SyriaTel's customer in the U.S. over a one month period. SyriaTel is a Syria based cell phone service company. The cost of providing telecom service in the U.S. is higher for an international cell phone service company. The higher cost leads to higher charge to customers. Therefore, partnership with local carriers will help SyriaTel to provide more stable service and more attractive pricing deals to its customers. The customers will be more loyal, not easy to churn.

```
In [ ]:
```