Overview

`**Sparkify**` is a Music platform and has **Customer Churn problem**. Churn defines cancellation of the subscription.

In this case we are going to predict potential churner subscribers. If `**Sparkify**` detects the subscriber who will be churner, they can take actions to labeled subscribers like discount, no ads and so on.

The data provided from `**Sparkify**` Company has

- **User-level information**
  - These columns contain data about users: their names, gender, location, registration date, browser, and account level (paid or free).
- **Log-specific information**
  - Log-specific information shows how a particular user interacts with the service.
- **Song-level information**
  - Information related to the song that is currently playing

First of all we analyzed the dataset and created some features about subscribers. After Feature engineering steps we went to modelling steps and tried some ML alogrithms for predict the potential churner subscribers correctly. In the evaluation step we used **F1 Score** because the dataset is imbalanced. Accuracy is not a good metric for this problem.

Finally We achieved **0.8 F1 Score** with Crossvalidated Random Forest Classifier.

Load and Clean Dataset

In this workspace, the file is `medium_sparkify_event_data.json` provided by Sparkify Company

Import Libraries and Setup Environment

In [1]:

```
import findspark
findspark.init()
findspark.find()
import datetime
import time
import numpy as np
import pandas as pd
import plotly.express as px
import httpagentparser

from pyspark import SparkContext, SparkConf
from pyspark.sql import SparkSession
from pyspark.sql import Window
from pyspark.sql.functions import udf, col, concat, count, lit, avg, lag, first, last, w
hen, from_unixtime, month, year
from pyspark.sql.functions import min as Fmin, max as Fmax, sum as Fsum, round as Fround

from pyspark.sql.types import IntegerType, DateType, TimestampType, StringType
from pyspark.ml.feature import StringIndexer, VectorAssembler, OneHotEncoder, StandardSca
ler
from pyspark.ml.stat import Correlation
from pyspark.ml.evaluation import MulticlassClassificationEvaluator, RegressionEvaluator
from pyspark.ml.classification import GBTClassifier, RandomForestClassifier, LinearSVC, L
ogisticRegression
from pyspark.ml.tuning import CrossValidator, ParamGridBuilder
from pyspark.ml import Pipeline
spark = (SparkSession
```

```
        .builder
        .appName('Sparkify_Churn')
        .getOrCreate())
```

In [2]:

```
#Read the dataframe
df = spark.read.json('medium-sparkify-event-data.json')
#convert pyspark dataframe to pandas dataframe
df.toPandas().head(5)
```

Out[2]:

	artist	auth	firstName	gender	itemInSession	lastName	length	level	location	method	page	registr
0	Martin Orford	Logged In	Joseph	M	20	Morales	597.55057	free	Corpus Christi, TX	PUT	NextSong	1.532064
1	John Brown's Body	Logged In	Sawyer	M	74	Larson	380.21179	free	Houston-The Woodlands-Sugar Land, TX	PUT	NextSong	1.538070
2	Afroman	Logged In	Maverick	M	184	Santiago	202.37016	paid	Orlando-Kissimmee-Sanford, FL	PUT	NextSong	1.535953
3	None	Logged In	Maverick	M	185	Santiago	NaN	paid	Orlando-Kissimmee-Sanford, FL	PUT	Logout	1.535953
4	Lily Allen	Logged In	Gianna	F	22	Campos	194.53342	paid	Mobile, AL	PUT	NextSong	1.535931

Exploratory Data Analysis

In this part I analyzed raw data and I want to show the dataset details

In [3]:

```
df.printSchema()
```

```
root
 |-- artist: string (nullable = true)
 |-- auth: string (nullable = true)
 |-- firstName: string (nullable = true)
 |-- gender: string (nullable = true)
 |-- itemInSession: long (nullable = true)
 |-- lastName: string (nullable = true)
 |-- length: double (nullable = true)
 |-- level: string (nullable = true)
 |-- location: string (nullable = true)
 |-- method: string (nullable = true)
 |-- page: string (nullable = true)
 |-- registration: long (nullable = true)
 |-- sessionId: long (nullable = true)
 |-- song: string (nullable = true)
 |-- status: long (nullable = true)
 |-- ts: long (nullable = true)
 |-- userAgent: string (nullable = true)
 |-- userId: string (nullable = true)
```

```
 |-- userId: string (nullable = true)
```

In [4]:

```
#counts
print ("The number of rows is {}".format(df.count()))
print ("The number of columns is {}".format(len(df.columns)))
print ("The total number of customers is {}".format(df.select("userId").distinct().count
()))
```

```
The number of rows is 543705
The number of columns is 18
The total number of customers is 449
```

Dataset including 543K Rows, 18 Columns and 449 Distinct customers

In [5]:

```
#Type of auths
auths = df.select('auth').distinct().show(truncate=False)
```

```
+----------+
|auth      |
+----------+
|Logged Out|
|Cancelled |
|Guest     |
|Logged In |
+----------+
```

- **There are 4 types auths** Logged Out, Cancelled, Guest, Logged In.

In [6]:

```
#type of levels
levels = df.select('level').distinct().show(truncate=False)
```

```
+-----+
|level|
+-----+
|free |
|paid |
+-----+
```

- **There are 2 types of levels:** free and paid

In [7]:

```
#Genders
genders = df.select('gender').distinct().show(truncate=False)
```

```
+------+
|gender|
+------+
|F     |
|null  |
|M     |
+------+
```

- **There are some missing values in gender**

In [8]:

```
# check duplicates
```

```
df.count() - df.dropDuplicates().count()
```

Out[8]:

0

- **There are no duplicates**

In [9]:

```
#Check null values
def check_nulls(dataframe):
    '''
    Check null values and return the null values in pandas Dataframe

    INPUT: Spark Dataframe
    OUTPUT: Null values

    '''
    # Create pandas dataframe
    nulls_check = pd.DataFrame(dataframe.select([count(when(col(c).isNull(), c)).alias(c
) for c in dataframe.columns]).collect(),
                               columns = dataframe.columns).transpose()
    nulls_check.columns = ['Null Values']
    return nulls_check
check_nulls(df).style.format({'Null Values':"{:,.0f}"}).background_gradient(cmap='Blues'
)
```

Out[9]:

	Null Values
artist	110,828
auth	0
firstName	15,700
gender	15,700
itemInSession	0
lastName	15,700
length	110,828
level	0
location	15,700
method	0
page	0
registration	15,700
sessionId	0
song	110,828
status	0
ts	0
userAgent	15,700
userId	0

As you can see in the null values table. There are some patterns.

- **We have distinct 2 values in null table which are `**110828 and 15700**`**
- **`**artist, length and song**` have a pattern and demographics info has another**

In [10]:

```
#Convert unixtime(in miliseconds) to Standard Time
```

```
df = df.withColumn('tsTime',from_unixtime(col('ts')/1000).cast(TimestampType()))
#Convert unixtime to Standard Date
df = df.withColumn('tsDate',from_unixtime(col('ts')/1000).cast(DateType()))
#Take month from date
df = df.withColumn('tsMonth',month(col('tsDate')))
#Take year from date
df = df.withColumn('tsYear',year(col('tsDate')))
#Create yearmonth(YYYYMM) using year and month columns
df = df.withColumn('tsYearMonth', concat(col('tsYear'),col('tsMonth')))
#Users registration date
df = df.withColumn('registrationDate',from_unixtime(col('registration')/1000).cast(DateTy
pe()))
```

In [11]:

```
#pandas dataframe
df.toPandas().head(5)
```

Out[11]:

	artist	auth	firstName	gender	itemInSession	lastName	length	level	location	method	...	status	
0	Martin Orford	Logged In	Joseph	M	20	Morales	597.55057	free	Corpus Christi, TX	PUT	...	200	15383520
1	John Brown's Body	Logged In	Sawyer	M	74	Larson	380.21179	free	Houston-The Woodlands-Sugar Land, TX	PUT	...	200	15383520
2	Afroman	Logged In	Maverick	M	184	Santiago	202.37016	paid	Orlando-Kissimmee-Sanford, FL	PUT	...	200	1538352
3	None	Logged In	Maverick	M	185	Santiago	NaN	paid	Orlando-Kissimmee-Sanford, FL	PUT	...	307	1538352
4	Lily Allen	Logged In	Gianna	F	22	Campos	194.53342	paid	Mobile, AL	PUT	...	200	1538352

5 rows × 24 columns

In [12]:

```
#min date and max date
df.agg({"tsDate": "min"}).show()
df.agg({"tsDate": "max"}).show()
```

```
+------------+
|min(tsDate)|
+------------+
| 2018-09-30|
+------------+

+------------+
|max(tsDate)|
+------------+
| 2018-11-30|
+------------+
```

- This event lasts for 2 months from ***2018-09-30 to 2018-11-30***.

User-level information

These columns contain data about users: **their names, gender, location, registration date, browser, and account level (paid or free).**

- ***userId*** **(string): user's id**
- ***firstName*** **(string): user's first name**
- ***lastName*** **(string): user's last name**
- ***gender*** **(string): user's gender, 2 categories (M and F)**
- ***location*** **(string): user's location**
- ***userAgent*** **(string): agent (browser) used by the user**
- ***registration*** **(int): user's registration timestamp**
- ***level*** **(string): subscription level, 2 categories (free and paid)**

In [13]:

```
#spark dataframe
df.select(['userId', 'firstName', 'lastName', 'gender', 'location', 'registration', 'reg
istrationDate', 'userAgent', 'level']).show(5)
```

```
+------+---------+--------+------+--------------------+-------------+----------------+-
-------------------+-----+
|userId|firstName|lastName|gender|            location| registration|registrationDate|
userAgent|level|
+------+---------+--------+------+--------------------+-------------+----------------+-
-------------------+-----+
|   293|   Joseph| Morales|     M|  Corpus Christi, TX|1532063507000|      2018-07-19|"Mo
zilla/5.0 (Mac...| free|
|    98|   Sawyer|  Larson|     M|Houston-The Woodl...|1538069638000|      2018-09-27|"Mo
zilla/5.0 (Mac...| free|
|   179| Maverick|Santiago|     M|Orlando-Kissimmee...|1535953455000|      2018-09-02|"Mo
zilla/5.0 (Mac...| paid|
|   179| Maverick|Santiago|     M|Orlando-Kissimmee...|1535953455000|      2018-09-02|"Mo
zilla/5.0 (Mac...| paid|
|   246|   Gianna|  Campos|     F|          Mobile, AL|1535931018000|      2018-09-02|Moz
illa/5.0 (Wind...| paid|
+------+---------+--------+------+--------------------+-------------+----------------+-
-------------------+-----+
only showing top 5 rows
```

In [14]:

```
df.where('userId ==""').count()
```

Out[14]:

```
15700
```

We have 15700 rows of empty strings in **userId column which means we don't have any information about these users.**

In [15]:

```
df.where(df.userId == "").select(['userId', 'firstName', 'lastName', 'gender', 'location
', 'registration', 'userAgent', 'level', 'tsDate', 'page', 'auth',]).show(10)
( df.where(df.userId == "").groupby('auth')
    .count()
    .toPandas().style.format({'count':"{:,.0f}"})
    .bar(color='#d65f5f', align='zero') )
```

```
+------+---------+--------+------+--------+------------+---------+-----+----------+----
-+-----------+
|userId|firstName|lastName|gender|location|registration|userAgent|level|    tsDate| page|
auth|
```

```
+------+---------+--------+------+--------+------------+---------+-----+----------+----
-+----------+
|      |     null|    null|  null|    null|        null|     null| paid|2018-09-30| Home
|Logged Out|
|      |     null|    null|  null|    null|        null|     null| paid|2018-09-30| Home
|Logged Out|
|      |     null|    null|  null|    null|        null|     null| paid|2018-09-30| Home
|Logged Out|
|      |     null|    null|  null|    null|        null|     null| paid|2018-09-30|Login
|Logged Out|
|      |     null|    null|  null|    null|        null|     null| free|2018-09-30| Home
|Logged Out|
|      |     null|    null|  null|    null|        null|     null| free|2018-09-30| Home
|Logged Out|
|      |     null|    null|  null|    null|        null|     null| free|2018-09-30| Help
|Logged Out|
|      |     null|    null|  null|    null|        null|     null| free|2018-09-30| Home
|Logged Out|
|      |     null|    null|  null|    null|        null|     null| free|2018-09-30|About
|Logged Out|
|      |     null|    null|  null|    null|        null|     null| free|2018-09-30|Login
|Logged Out|
+------+---------+--------+------+--------+------------+---------+-----+----------+----
-+----------+
only showing top 10 rows
```

Out[15]:

	auth	count
0	Logged Out	15,606
1	Guest	94

- **Rows with an empty string in the `**userId**` column correspond to logs in which the user has not been logged in (`**Logged Out or Guest**`). `**Logged out**` accounts for more missing values.**

Log-specific information

Log-specific information shows how a particular user interacts with the service.

- ***ts*** **(int): timestamp of the log**
- ***page*** **(string): type of interaction associated with the page (NextSong, Home, Login, Cancellation Confirmation, etc.)**
- ***auth*** **(string): authentication level, 4 categories (Logged In, Logged Out, Cancelled, Guest)**
- ***sessionId*** **(int): a session id**
- ***itemInSession*** **(int): log count in the session**
- ***method*** **(string): HTTP request method, 2 categories (GET and PUT)**
- ***status*** **(int): HTTP status code, 3 categories (200, 307 and 404)**

In [16]:

```
df.select('ts','tsDate','page', 'auth', 'sessionId', 'itemInSession', 'method', 'status'
).show(5)
```

```
+-------------+----------+--------+---------+---------+-------------+------+------+
|           ts|    tsDate|    page|     auth|sessionId|itemInSession|method|status|
+-------------+----------+--------+---------+---------+-------------+------+------+
|1538352011000|2018-09-30|NextSong|Logged In|      292|           20|   PUT|   200|
|1538352025000|2018-09-30|NextSong|Logged In|       97|           74|   PUT|   200|
|1538352118000|2018-09-30|NextSong|Logged In|      178|          184|   PUT|   200|
|1538352119000|2018-09-30|  Logout|Logged In|      178|          185|   PUT|   307|
|1538352124000|2018-09-30|NextSong|Logged In|      245|           22|   PUT|   200|
+-------------+----------+--------+---------+---------+-------------+------+------+
only showing top 5 rows
```

In [17]:

```
def groupBy_count_pct(col_name):
    '''
    group by a specified column and return count and percentage in pandas Dataframe

    INPUT: column name, string
    OUTPUT: count and percentage
    '''
    grouped_df = df.groupby(col_name)\
                    .agg((count(col_name)).alias('count'),\
                      (count(col_name) / df.count()).alias('percentage'))\
                    .sort('percentage', ascending = False)\
                    .toPandas().style.format({'count':"{:,.0f}",\
                                          'percentage': "{:,.2%}"})
    return grouped_df
```

In [18]:

```
# convert spark dataframe to pandas dataframe
groupBy_count_pct('auth').bar(subset='count', color='#d65f5f', align='zero')\
                         .bar(subset='percentage', color='#FFA07A', align='zero')
```

Out[18]:

	auth	count	percentage
0	Logged In	527,906	97.09%
1	Logged Out	15,606	2.87%
2	Cancelled	99	0.02%
3	Guest	94	0.02%

In [19]:

```
# using sql with pandas
df.createOrReplaceTempView('df')

spark.sql(
    """
    SELECT auth,
           COUNT(1) AS count,
           COUNT(1) / (SELECT COUNT(1) FROM df) AS percentage
    FROM df
    GROUP BY auth
    ORDER BY percentage DESC
    """
    ).toPandas().style.format({'count':"{:,.0f}",
                               'percentage': "{:,.2%}"})\
     .background_gradient(cmap='Blues')
```

Out[19]:

	auth	count	percentage
0	Logged In	527,906	97.09%
1	Logged Out	15,606	2.87%
2	Cancelled	99	0.02%
3	Guest	94	0.02%

- **97.09% Customers are `**logged in**` .**

In [20]:

```
groupBy_count_pct('method').bar(subset='count', color='#d65f5f', align='zero')\
```

```
                          .bar(subset='percentage', color='#FFA07A', align='zero')
```

Out[20]:

	method	count	percentage
0	PUT	495,143	91.07%
1	GET	48,562	8.93%

- **91.07% of HTTP request method is `**PUT**`**

In [21]:

```
groupBy_count_pct('status').bar(subset='count', color='#d65f5f', align='zero')\
                          .bar(subset='percentage', color='#FFA07A', align='zero')
```

Out[21]:

	status	count	percentage
0	200	493,269	90.72%
1	307	49,917	9.18%
2	404	519	0.10%

- **HTTP status code, 3 categories `**(200, 307 and 404)**`. 90.72% of HTTP status code is `**200**`. `**404**` (which means error page) accounts for 0.10%.**

In [22]:

```
# '404' status corresponds to 'Error' in the page column.

df.where(df.status == '404').select(['userId', 'tsDate', 'sessionId', 'status', 'page'])
\
  .toPandas().head()
```

Out[22]:

	userId	tsDate	sessionId	status	page
0	232	2018-10-01	477	404	Error
1		2018-10-01	166	404	Error
2	295	2018-10-01	294	404	Error
3	92	2018-10-02	561	404	Error
4	212	2018-10-02	494	404	Error

In [23]:

```
groupBy_count_pct('page').bar(subset='count', color='#d65f5f', align='zero')\
                        .bar(subset='percentage', color='#FFA07A', align='zero')
```

Out[23]:

	page	count	percentage
0	NextSong	432,877	79.62%
1	Home	27,412	5.04%
2	Thumbs Up	23,826	4.38%
3	Add to Playlist	12,349	2.27%
4	Add Friend	8,087	1.49%
5	Roll Advert	7,773	1.43%

	page	count	percentage
6	Login	6,011	1.11%
7	Logout	5,990	1.10%
8	Thumbs Down	4,911	0.90%
9	Downgrade	3,811	0.70%
10	Help	3,150	0.58%
11	Settings	2,964	0.55%
12	About	1,855	0.34%
13	Upgrade	968	0.18%
14	Save Settings	585	0.11%
15	Error	519	0.10%
16	Submit Upgrade	287	0.05%
17	Submit Downgrade	117	0.02%
18	Cancel	99	0.02%
19	Cancellation Confirmation	99	0.02%
20	Register	11	0.00%
21	Submit Registration	4	0.00%

- **The most visited page is the `**Next Song**` and followed by `**Home and Thumbs Up**`**

In [24]:

```
# session pandas dataframe
session_df = df.where(df.userId !='').groupby(['userId', 'sessionId'])\
                .agg(((Fmax('ts')-Fmin('ts'))/1000).alias('sessionLength'))\
                .sort('sessionLength', ascending = False).toPandas()
session_df
```

Out[24]:

	userId	sessionId	sessionLength
0	105	1052	206766.0
1	246	2860	197858.0
2	153	4131	189368.0
3	86	3425	189274.0
4	244	2470	177224.0
...	...	...	...
6075	230	2398	0.0
6076	87	777	0.0
6077	233	4058	0.0
6078	95	1683	0.0
6079	183	4246	0.0

6080 rows × 3 columns

In [25]:

```
# maximum session length and minimum session length
print('The maximum session length in hour is {}'.format(session_df.loc[0,'sessionLength']
/3600))
print('The minimum session length in hour is {}'.format(session_df.loc[6079,'sessionLengt
h']/3600))
```

```
The maximum session length in hour is 57.435
The minimum session length in hour is 0.0
```

In [26]:

```
session_df.query('sessionLength!=0')
```

Out[26]:

	userId	sessionId	sessionLength
0	105	1052	206766.0
1	246	2860	197858.0
2	153	4131	189368.0
3	86	3425	189274.0
4	244	2470	177224.0
...	...	...	...
5931	222	1787	1.0
5932	100024	194	1.0
5933	40	672	1.0
5934	178	3929	1.0
5935	114	4451	1.0

5936 rows × 3 columns

- The `**maximum session length**` is **more than 57 hours.** The `**minimum session length**` is **0 second.** Other than 0, the `**second minimum session length**` is **1 second**

Song-level information

Information related to the song that is currently playing

- **song** (string): song name
- **artist** (string): artist name
- **length** (double): song's length in seconds

In [27]:

```
# Song-level information
df.select(['artist', 'song', 'length']).toPandas().head()
```

Out[27]:

	artist	song	length
0	Martin Orford	Grand Designs	597.55057
1	John Brown's Body	Bulls	380.21179
2	Afroman	Because I Got High	202.37016
3	None	None	NaN
4	Lily Allen	Smile (Radio Edit)	194.53342

In [28]:

```
# counts
print('The number of distinct artists is {}'.format(df.filter(df.artist.isNotNull()).sele
ct('artist').distinct().count()))
print('The number of distinct songs is {}'.format(df.filter(df.song.isNotNull()).select(
'song').distinct().count()))
print('The number of songs including full duplicates is {}'.format(df.select(['artist','
```

```
song','length']).distinct().count()))
```

```
The number of distinct artists is 21247
The number of distinct songs is 80292
The number of songs including full duplicates is 92097
```

In [29]:

```
AllPages = df.select('page').distinct().toPandas()
AllPages
```

Out[29]:

	page
0	Cancel
1	Submit Downgrade
2	Thumbs Down
3	Home
4	Downgrade
5	Roll Advert
6	Logout
7	Save Settings
8	Cancellation Confirmation
9	About
10	Submit Registration
11	Settings
12	Login
13	Register
14	Add to Playlist
15	Add Friend
16	NextSong
17	Thumbs Up
18	Help
19	Upgrade
20	Error
21	Submit Upgrade

In [30]:

```
SongNullPages = df.where('song is null').select('page').distinct().toPandas()
SongNullPages
```

Out[30]:

	page
0	Cancel
1	Submit Downgrade
2	Thumbs Down
3	Home
4	Downgrade
5	Roll Advert
6	Logout
7	Save Settings

	page
8	Cancellation Confirmation
9	About
10	Submit Registration
11	Settings
12	Login
13	Register
14	Add to Playlist
15	Add Friend
16	Thumbs Up
17	Help
18	Upgrade
19	Error
20	Submit Upgrade

In [31]:

```
pd.concat([AllPages, SongNullPages]).drop_duplicates(keep=False)
```

Out[31]:

	page
16	NextSong

Only `**NextSong page**` has song information.

In [32]:

```
df.filter('page == "NextSong"').groupby('page').count().show()
```

```
+--------+------+
|    page| count|
+--------+------+
|NextSong|432877|
+--------+------+
```

- **`**NextSong**` has 432,877 pages in total**

Data Visualization

We'll start explore the behaviors of users who stayed and who left.

In [33]:

```
# delete empty strings in userid
df = df.filter('userId !=""')
```

In [34]:

```
print('The number of rows after deleting empty strings in userid is {}'.format(df.count()
))
```

```
The number of rows after deleting empty strings in userid is 528005
```

In [35]:

```
# add downgrade flag
```

```
df = df.withColumn('downgrade', when(df.page =='Submit Downgrade', 1).otherwise(0))
df = df.withColumn('user_downgrade', Fmax('downgrade').over(Window.partitionBy('userId')
))
df = df.withColumn('churn', when(df.page == 'Cancellation Confirmation', 1).otherwise(0)
)
df = df.withColumn('user_churn', Fmax('churn').over(Window.partitionBy('userId')))
```

In [36]:

```
def churn_df(df, col):
    """
    return pandas dataframe that calculate counts by user_churn and gender
    INPUT: df, dataframe
           col, colname, string
    OUTPUT: return pandas dataframe
    """
    churn_df = df.drop_duplicates(['userId'])\
                 .groupby(['user_churn', col]).count()\
                 .sort(['user_churn','count'], ascending = False).toPandas()
    return churn_df
```

In [37]:

```
churn_gender_df = churn_df(df, 'gender')
churn_gender_df
```

Out[37]:

	user_churn	gender	count
0	1	M	54
1	1	F	45
2	0	M	196
3	0	F	153

In [38]:

```
# churn_gender_df['percentage'] = (churn_gender_df['count']/churn_gender_df.groupby('gend
er')['count'].transform('sum')).map('{:.2%}'.format)
```

In [39]:

```
def churn_stack_bar(df, x, y):
    '''
    return 100% stack bar chart by not churn and churn group

        INPUT:  df, dataframe
                x, column name, string
                y, column name, string
        OUTPUT: 100% stack bar chart
    '''
    fig = px.histogram(df, x, y,
                 barmode='stack',
                 barnorm='percent',
                 text_auto = True,
                 color=df['user_churn'].map({0:"Not Churned", 1:"Churned"}),
                 color_discrete_map={
                        "Not Churned": '#3366CC',
                        "Churned":   '#b02d0e'},
                 title='churn rate by {}'.format(x),
                 template="simple_white")
    fig.update_layout(xaxis=dict(ticks=""),
                      yaxis=dict(ticksuffix="%", tickformat=".2f", title='churn rate%'),
                      width=800, height=500)
    return fig
```

In [40]:

```
churn_stack_bar(churn_gender_df, x='gender', y='count')\
.update_layout(xaxis=dict(tickvals = [0,1], ticktext=['Female', 'Male']))
```

- **From above chart we can see that `**Female**` churn rate (22.73%) is slightly higher than `**Male**` (21.60%)**

In [41]:

```
# function for generating pandas dataframe
def df_chisq(df, index, columns, values):
    '''
    return pandas dataframe for chi-square test
    INPUT: df, dataframe
           index, string
           columns, string
           values, string
    OUTPUT: dataframe
    '''
    df = df.pivot(index= index, columns = columns, values = values)
    df = df.rename(columns={0:'Not Churned', 1:'Churned'}).fillna(0)
    return df
```

In [42]:

```
churn_gender_chisq = df_chisq(churn_gender_df, 'gender', 'user_churn', 'count')
churn_gender_chisq
```

Out[42]:

user_churn	Not Churned	Churned
gender		
F	153	45
M	196	54

In [43]:

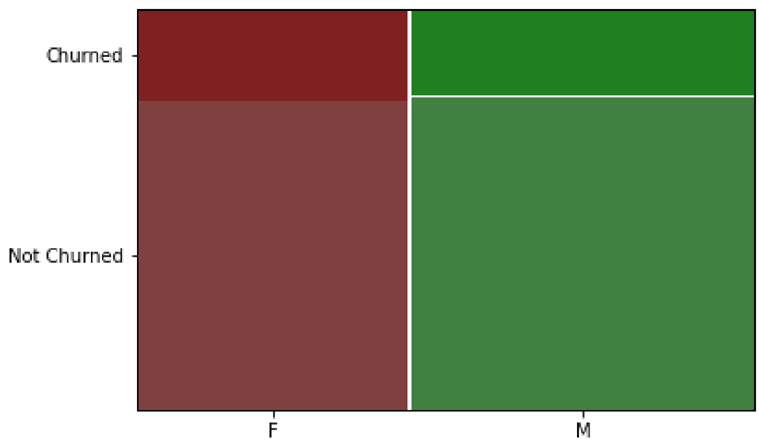
```
# run chi-square test for independence
import bioinfokit
from bioinfokit.analys import stat
bioinfokit.analys.stat.chisq(churn_gender_chisq)
```

```
Chi-squared test

Test               Df    Chi-square    P-value
---------------  ----  ------------  ---------
Pearson             1     0.0292217   0.864268
Log-likelihood      1     0.0291979   0.864323

Expected frequency counts

      Not Churned    Churned
--  -------------  ---------
 0        154.246    43.7545
 1        194.754    55.2455
```



- **From chi-square test, we can see that `**p-value > 0.05**` so we accept null hypothesis that gender and user_churn are independent.**

In [44]:

```
# churn analysis by level
churn_level_df = churn_df(df, 'level')
churn_level_df
```

Out[44]:

	user_churn	level	count
0	1	free	73
1	1	paid	26
2	0	free	269
3	0	paid	80

In [45]:

```
churn_stack_bar(churn_level_df, x='level', y='count')\
.update_layout(xaxis=dict(title='user level'))
```

- From above chart, **paid users** churn rate (24.53%) is **slightly higher** than **free users** (21.35%).

In [46]:

```
churn_level_chisq = df_chisq(churn_level_df, 'level', 'user_churn', 'count')
churn_level_chisq
```

Out[46]:

user_churn	Not Churned	Churned
level		
free	269	73
paid	80	26

In [47]:

```
# run chi-square test for independence
bioinfokit.analys.stat.chisq(churn_level_chisq)
```

```
Chi-squared test

Test              Df    Chi-square    P-value
--------------  ----  ------------  ---------
Pearson            1      0.309351   0.578079
Log-likelihood     1      0.305075   0.580718

Expected frequency counts

      Not Churned    Churned
--  -------------  ---------
 0        266.424    75.5759
 1         82.5759   23.4241
```

- From chi-squared test, we can see that **p-value > 0.05** that **we accept null hypothesis that level and user_churn are independent.**

In [48]:

```
avg_songs_df = df.where(df.song.isNotNull()).groupby(['user_churn', 'userId', 'sessionId
']).count()\
                 .withColumn('average', avg('count').over(Window.partitionBy('userId', '
user_churn'))).toPandas()
avg_songs_df
```

Out[48]:

	user_churn	userId	sessionId	count	average
0	1	100010	62	49	48.000000
1	1	100010	166	47	48.000000
2	1	200002	2	40	62.000000
3	1	200002	91	42	62.000000
4	1	200002	164	22	62.000000
...	...	...	...	...	...
5916	0	300029	249	7	51.142857
5917	0	300029	407	35	51.142857
5918	0	300029	421	46	51.142857
5919	0	300029	459	21	51.142857
5920	0	300029	524	106	51.142857

5921 rows × 5 columns

In [49]:

```
avg_songs_df1 = avg_songs_df[['user_churn','userId','average']].drop_duplicates()
avg_songs_df1
```

Out[49]:

	user_churn	userId	average
0	1	100010	48.000000
2	1	200002	62.000000
7	1	296	22.400000
12	0	125	20.666667
15	1	124	114.125000
...	...	...	...
5880	0	216	84.750000
5900	0	119	34.375000
5908	1	100001	32.000000
5911	0	200049	8.666667
5914	0	300029	51.142857

448 rows × 3 columns

In [50]:

```
def box_plot_churn(df, x, y):
    """
```

```
    return box plot by not churned and churned group
    INPUT: df, dataframe
           x, colname, string
           y, colname, string
    OUTPUT: boxplot
    """
    fig = px.box(df, x, y,
             points='all',
             color = df['user_churn'].map({0:"Not Churned", 1:"Churned"}),
             color_discrete_map={
                                 "Not Churned": '#3366CC',
                                 "Churned":   '#b02d0e'},
             template="simple_white")
    fig.update_layout(xaxis=dict(tickvals=[0,1],ticktext=['Not Churned','Churned'], ticks
="", title = 'User Churn'),
                      yaxis=dict(title='average'),
                      width=800, height=500)
    fig.update_traces(width=0.6)
    return fig
```

In [51]:

```
# boxplot for number of songs played per session
box_plot_churn(avg_songs_df1, x="user_churn", y='average').update_layout(title='average n
umber of songs per session')
```

- **Notice that we have some outliers for both groups(**Not Churned and Churned). Outliers of churned are more than not churned. The range of Not Churned ** is **broader and denser** than ** churned **.**

In [52]:

```
avg_session_length = df.groupby(['user_churn', 'userId', 'sessionId'])\
                      .agg(((Fmax('ts')-Fmin('ts'))/1000).alias('sessionlength'))\
                      .groupby(['user_churn', 'userId']).agg({'sessionlength':'avg'}).t
oPandas()
avg_session_length
```

Out[52]:

	user_churn	userId	avg(sessionlength)
0	1	100010	12622.500000
1	1	200002	15783.600000
2	1	296	5822.200000
3	0	125	5099.000000
4	1	124	26740.058824
...	...	...	...
443	0	216	21447.900000
444	0	119	8321.375000
445	1	100001	8259.666667
446	0	200049	2534.333333
447	0	300029	13259.000000

448 rows × 3 columns

In [53]:

```
box_plot_churn(avg_session_length, 'user_churn','avg(sessionlength)').update_layout(title
='average session length')
```

- **Notice that we have some outliers for both groups(**Not Churned and Churned). The maximum of outliers of churned are higher than** not churned. **The range of Not Churned ** is **similar** but **denser** than ** churned **.**

In [54]:

```
life_time_df = df.groupby(['user_churn', 'userId', 'ts', 'registration']).count() \
                .withColumn('LifetimeInSec', (df.ts - df.registration)/1000)\
                .withColumn("LifetimeInMin",Fround(col('LifetimeInSec')/60)) \
                .withColumn("LifetimeInHour",Fround(col('LifetimeInSec')/3600)) \
```

```
                    .withColumn("LifetimeInDay",Fround(col('LifetimeInSec')/(24*3600)))\
                    .groupBy('userId', 'user_churn').agg({'LifetimeInDay' : 'max'}).toPanda
s()
life_time_df
```

Out[54]:

	userId	user_churn	max(LifetimeInDay)
0	100010	1	14.0
1	200002	1	53.0
2	296	1	27.0
3	125	0	105.0
4	124	1	113.0
...	...	...	...
443	216	0	115.0
444	119	0	193.0
445	100001	1	45.0
446	200049	0	112.0
447	300029	0	66.0

448 rows × 3 columns

In [55]:

```
life_time_df.describe()
```

Out[55]:

	user_churn	max(LifetimeInDay)
count	448.000000	448.000000
mean	0.220982	82.814732
std	0.415372	40.551666
min	0.000000	-1.000000
25%	0.000000	61.000000
50%	0.000000	76.000000
75%	0.000000	99.250000
max	1.000000	390.000000

- **Notice that `**minimum value**` of LifetimeInDay is -1 which is unqualified should be cleaned.**

In [56]:

```
life_time_df.query('`max(LifetimeInDay)` == -1.0')
```

Out[56]:

	userId	user_churn	max(LifetimeInDay)
245	100051	1	-1.0

In [57]:

```
life_time_df.drop(life_time_df.index[245], inplace=True)
```

In [58]:

```
life_time_df.describe()
```

Out[58]:

	user_churn	max(LifetimeInDay)
count	447.000000	447.000000
mean	0.219239	83.002237
std	0.414195	40.402210
min	0.000000	2.000000
25%	0.000000	61.000000
50%	0.000000	76.000000
75%	0.000000	99.500000
max	1.000000	390.000000

- **Notice that `**minimum value**` of LifetimeInDay is 2 now which means we have sucessfully deleted the dirty data point.**

In [59]:

```
box_plot_churn(life_time_df, 'user_churn','max(LifetimeInDay)')\
.update_layout(title='Days since registration', yaxis = dict(title='days',tickformat='0.f
'))
```

- **Notice that overall `**not churned**` group stays longer than `**churned**` .**

In [60]:

```
# dataframe with friends
friends_df = df.where('page =="Add Friend"').groupby(['user_churn','userId']).count()\
               .groupby(['user_churn','userId']).agg({'count': 'avg'}).toPandas()
friends_df
```

Out[60]:

	user_churn	userId	avg(count)
0	1	100010	3.0
1	1	200002	2.0
2	1	296	2.0
3	0	125	3.0
4	1	124	26.0
...	...	...	...
404	0	200036	3.0
405	0	216	37.0
406	0	119	12.0
407	1	100001	1.0
408	0	300029	7.0

409 rows × 3 columns

In [61]:

```
box_plot_churn(friends_df, 'user_churn','avg(count)')\
.update_layout(title='Number of friends')
```

- Notice that overall **not churned** group **has more friendsr** than **churned** . For both groups their friends numbers mainly fall **between 0 and 20.**

In [62]:

```
df.select('userAgent').distinct().show(5, truncate=False)
```

```
+-----------------------------------------------------------------------------------------
------------------------------------+
|userAgent
```

```
|
+--------------------------------------------------------------------------------------------
-----------------------------------+
|"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_8_5) AppleWebKit/537.36 (KHTML, like Gecko) C
hrome/36.0.1985.143 Safari/537.36"|
|"Mozilla/5.0 (Windows NT 5.1) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/36.0.1985.14
3 Safari/537.36"                  |
|Mozilla/5.0 (X11; Ubuntu; Linux i686; rv:31.0) Gecko/20100101 Firefox/31.0
|
|"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_4) AppleWebKit/537.36 (KHTML, like Gecko) C
hrome/36.0.1985.125 Safari/537.36"|
|"Mozilla/5.0 (Macintosh; Intel Mac OS X 10_9_4) AppleWebKit/537.36 (KHTML, like Gecko) C
hrome/35.0.1916.153 Safari/537.36"|
+--------------------------------------------------------------------------------------------
-----------------------------------+
only showing top 5 rows
```

In [63]:

```
#Function for users os information
udf_os = udf(lambda x: httpagentparser.simple_detect(x)[0].split(' ')[0]) # The default type of the udf() is StringType which can be ignored
#Function for users browser information
udf_browser =  udf(lambda x: httpagentparser.simple_detect(x)[1].split(' ')[0])
#Function for users Location
udf_location = udf(lambda x: x.split(',')[1])
```

In [64]:

```
#Users Os, Browser and location informations Using UDF's
df = df.withColumn('os', udf_os('userAgent'))
df = df.withColumn('browser', udf_browser('userAgent'))
df = df.withColumn('location', udf_location('location'))
```

In [65]:

```
#Show the result after UDF's
df.select(['os', 'browser', 'location']).show(truncate=False)
```

```
+-------+-------+--------+
|os     |browser|location|
+-------+-------+--------+
|MacOS  |Chrome | TX     |
|MacOS  |Chrome | TX     |
|MacOS  |Chrome | FL     |
|MacOS  |Chrome | FL     |
|Windows|Firefox| AL     |
|MacOS  |Chrome | MN     |
|MacOS  |Chrome | FL     |
|MacOS  |Chrome | TX     |
|Windows|Firefox| TX     |
|Windows|Firefox| TX     |
|MacOS  |Chrome | MN     |
|Windows|Firefox| AL     |
|MacOS  |Chrome | FL     |
|Windows|Chrome | CA     |
|MacOS  |Chrome | MN     |
|MacOS  |Chrome | FL     |
|MacOS  |Chrome | TX     |
|MacOS  |Chrome | TX     |
|MacOS  |Chrome | TX     |
|Windows|Firefox| TX     |
+-------+-------+--------+
only showing top 20 rows
```

In [66]:

```
#os
os_df = churn_df(df, 'os')
```

```
os_df.style.bar(subset='count', color='#d65f5f', align='zero')
```

Out[66]:

	user_churn	os	count
0	1	Windows	48
1	1	MacOS	35
2	1	iPhone	11
3	1	Ubuntu	3
4	1	Linux	2
5	0	Windows	174
6	0	MacOS	138
7	0	Ubuntu	11
8	0	Linux	10
9	0	IPad	9
10	0	iPhone	7

In [67]:

```
churn_stack_bar(os_df, x='os', y='count')
```

In [68]:

```
os_chisq = df_chisq(os_df, 'os', 'user_churn', 'count')
os_chisq
```

Out[68]:

user_churn	Not Churned	Churned
os		
IPad	9.0	0.0

user_churn	Not Churned	Churned
os		
Linux	10.0	2.0
MacOS	138.0	35.0
Ubuntu	11.0	3.0
Windows	174.0	48.0
iPhone	7.0	11.0

In [69]:

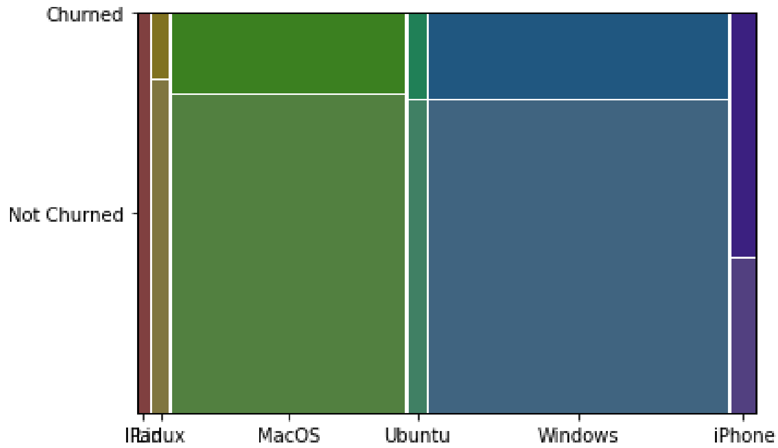
```
# run chi-square test for independence
import bioinfokit
from bioinfokit.analys import stat
bioinfokit.analys.stat.chisq(os_chisq)
```

```
Chi-squared test

Test              Df    Chi-square     P-value
--------------  ----  ------------  ----------
Pearson            5       19.0561  0.00187646
Log-likelihood     5       17.7574  0.00326629

Expected frequency counts

      Not Churned     Churned
--  -------------  ----------
 0        7.01116     1.98884
 1        9.34821     2.65179
 2      134.77       38.2299
 3       10.9062      3.09375
 4      172.942      49.058
 5       14.0223      3.97768
```



- **From chi-square test, we can see that $p < 0.05$ which means we refuse null hypothesis and accept alternative hypothesis that os and user_churn are correlated.**

In [70]:

```
#browser
browser_df = churn_df(df, 'browser')
browser_df.style.bar(subset='count', color='#d65f5f', align='zero')
```

Out[70]:

	user_churn	browser	count
0	1	Chrome	44
1	1	Firefox	26
2	1	Safari	22
3	1	Microsoft	7
4	0	Chrome	185

	user_churn	browser	count
5	0	Firefox	73
6	0	Safari	67
7	0	Microsoft	24

In [71]:

```
churn_stack_bar(browser_df, x='browser', y='count')
```

- **Most** of users use **windows** for `**os**` and **most** of users use **Chrome** `**web browser**`
- `**Ipad**` has **0** churn rate but proportion of `**Ipad**` users are **very low** (18)
- `**Linux, MacOs, Ubuntu and Windows**` users has nearly **20%** churn rate but `**iPhone**` users **61%** churn rate. Maybe there is a problem with user experience.
- Users who are using `**Firefox**` web browser churn rate is **highest** (26%)

In [72]:

```
browser_chisq = df_chisq(browser_df, 'browser', 'user_churn', 'count')
browser_chisq
```

Out[72]:

user_churn	Not Churned	Churned
browser		
Chrome	185	44
Firefox	73	26
Microsoft	24	7
Safari	67	22

In [73]:

```
# run chi-square test for independence
```

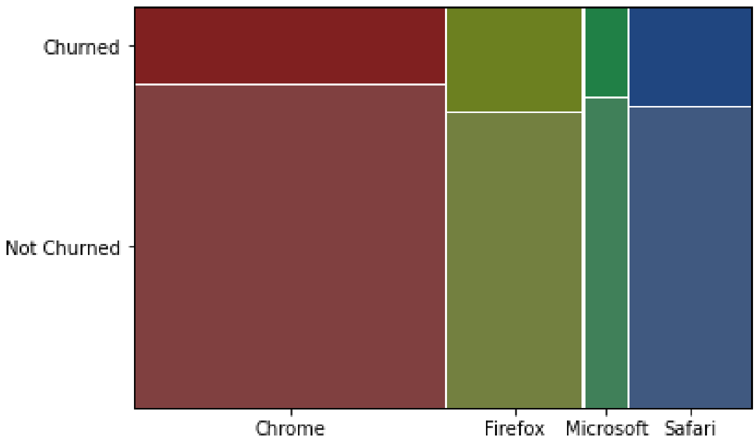
```
import bioinfokit
from bioinfokit.analys import stat
bioinfokit.analys.stat.chisq(browser_chisq)
```

```
Chi-squared test

Test              Df    Chi-square    P-value
--------------  ----  ------------  ---------
Pearson            3       2.46325   0.481969
Log-likelihood     3       2.45036   0.484329

Expected frequency counts

      Not Churned    Churned
--  -------------  ---------
 0        178.395    50.6049
 1         77.1228   21.8772
 2         24.1496    6.85045
 3         69.3326   19.6674
```



- **From chi-square test, we can see that p > 0.05 which means we accept null hypothesis that **browser and user_churn** are independent**

In [74]:

```
#location
location_df = churn_df(df, 'location')
```

In [75]:

```
location_df
```

Out[75]:

	user_churn	location	count
0	1	CA	10
1	1	NY-NJ-PA	8
2	1	IL-IN-WI	7
3	1	FL	6
4	1	TX	5
...	...	...	...
107	0	HI	1
108	0	KY-IN	1
109	0	NM	1
110	0	SC-NC	1
111	0	ND-MN	1

112 rows × 3 columns

112 rows × 3 columns

In [76]:

```
location_df['location'].unique()
```

Out[76]:

```
array([' CA', ' NY-NJ-PA', ' IL-IN-WI', ' FL', ' TX', ' PA-NJ-DE-MD',
       ' OH', ' GA', ' MI', ' KY', ' CT', ' RI-MA', ' GA-AL', ' WA',
       ' IN', ' OK', ' NM', ' AL', ' AZ', ' LA', ' PA-NJ', ' NY', ' TN',
       ' NJ', ' NC', ' CO', ' MD', ' TN-VA', ' MS', ' OR-WA', ' NH',
       ' UT', ' MA-NH', ' VT', ' VA', ' DC-VA-MD-WV', ' PA', ' OR',
       ' KY-IN', ' NC-SC', ' SC', ' MO-KS', ' MO-IL', ' VA-NC', ' WI',
       ' MN-WI', ' IL', ' OH-KY-IN', ' NV', ' TN-MS-AR', ' KS', ' MD-WV',
       ' WV', ' MT', ' SD', ' NE-IA', ' AK', ' MN', ' IA', ' IL-MO',
       ' ID', ' MA-CT', ' ME', ' IA-IL-MO', ' AR', ' UT-ID', ' AR-OK',
       ' HI', ' SC-NC', ' ND-MN'], dtype=object)
```

In [77]:

```
location_df['location'].nunique()
```

Out[77]:

```
70
```

In [78]:

```
churn_stack_bar(location_df, x='location', y='count')
```

In [79]:

```
location_chisq = df_chisq(location_df, 'location', 'user_churn', 'count')
location_chisq
```

Out[79]:

user_churn	Not Churned	Churned
location		

user_churn	Not Churned	Churned
location		
AK	2.0	0.0
AL	7.0	2.0
AR	1.0	0.0
AR-OK	1.0	0.0
AZ	8.0	2.0
...	...	...
VA-NC	3.0	1.0
VT	0.0	1.0
WA	8.0	2.0
WI	7.0	0.0
WV	2.0	0.0

70 rows × 2 columns

- `**Location**` can be good estimator because some locations have **100% churn rate**.

Feature Engineering

Based on the above analysis, we need the following features:

- total song listened
- Number of thumbs down
- Number of thumbs up
- Total time since registration (lifetime in day)
- Average songs played per session
- Number of songs added to the playlist
- Total number of friends
- Help page visits
- Settings page visits
- Downgrade
- os
- location

Target:

- churn

In [80]:

```
features = []
```

total song listened

In [81]:

```
song_df = df.select('userId', 'page').where(df.page=='NextSong').groupBy('userId').count
()\
            .withColumnRenamed('count', 'songs')
song_df.show(5)
features.append(song_df)
```

```
+------+-----+
|userId|songs|
+------+-----+
|   296|  112|
|100010|   96|
|200002|  310|
```

```
|   125|   62|
|    51|  266|
+------+-----+
only showing top 5 rows
```

Number of thumbs down

In [82]:

```
thumbs_down_df = df.select('userId', 'page').where(df.page=='Thumbs Down').groupBy('user
Id').count()\
                     .withColumnRenamed('count', 'thumbs down')
thumbs_down_df.show(5)
features.append(thumbs_down_df)
```

```
+------+-----------+
|userId|thumbs down|
+------+-----------+
|100010|          3|
|200002|          5|
|   125|          1|
|   124|         15|
|    51|          1|
+------+-----------+
only showing top 5 rows
```

Number of thumbs up

In [83]:

```
thumbs_up_df = df.select('userId', 'page').where(df.page == 'Thumbs Up').groupBy('userId
').count() \
                   .withColumnRenamed('count', 'thumbs_up')
thumbs_up_df.show(5)

features.append(thumbs_up_df)
```

```
+------+---------+
|userId|thumbs_up|
+------+---------+
|   296|        8|
|100010|        4|
|200002|       15|
|   125|        3|
|    51|       16|
+------+---------+
only showing top 5 rows
```

Lifetime in day

In [84]:

```
lifetime_df = df.groupby([ 'userId', 'ts', 'registration']).count() \
                .withColumn('LifetimeInDay', Fround((df.ts - df.registration)/(1000*24*3
600)))\
                .groupBy('userId').agg({'LifetimeInDay' : 'max'})\
                .withColumnRenamed('max(LifetimeInDay)', 'days_since_registration')
lifetime_df.show(5)

features.append(lifetime_df)
```

```
+------+-----------------------+
|userId|days_since_registration|
+------+-----------------------+
```

```
|   296|                  27.0|
|100010|                  14.0|
|200002|                  53.0|
|   125|                 105.0|
|   124|                 113.0|
+------+----------------------+
only showing top 5 rows
```

Average songs played per session

In [85]:

```
avg_songs_p_session_df =  df.where(df.song.isNotNull()).groupby(['userId', 'sessionId'])
.count()\
                            .groupby(['userId']).agg(Fround(avg('count'))) \
                            .withColumnRenamed('avg(count)', 'avg_songs_p_session')

avg_songs_p_session_df.show(5)

features.append(avg_songs_p_session_df)
```

```
+------+--------------------+
|userId|round(avg(count), 0)|
+------+--------------------+
|200002|                62.0|
|   296|                22.0|
|100010|                48.0|
|   125|                21.0|
|     7|                31.0|
+------+--------------------+
only showing top 5 rows
```

Number of songs added to playlist

In [86]:

```
songs_in_playlist_df = df.select('userId', 'page') \
                         .where(df.page == 'Add to Playlist').groupBy('userId').count()
\
                         .withColumnRenamed('count', 'songs_in_playlist')

songs_in_playlist_df.show(5)

features.append(songs_in_playlist_df)
```

```
+------+-----------------+
|userId|songs_in_playlist|
+------+-----------------+
|   296|                3|
|100010|                1|
|200002|                6|
|   125|                2|
|    51|                8|
+------+-----------------+
only showing top 5 rows
```

Total number of friends

In [87]:

```
number_of_friends_df = df.where('page == "Add Friend"').groupby(['userId']).count()\
                         .groupby(['userId']).agg({'count': 'avg'}).withColumnRenamed('a
vg(count)', 'number_of_friends')
```

```
number_of_friends_df.show(5)

features.append(number_of_friends_df)
```

```
+------+-----------------+
|userId|number_of_friends|
+------+-----------------+
|   296|              2.0|
|100010|              3.0|
|200002|              2.0|
|   125|              3.0|
|   124|             26.0|
+------+-----------------+
only showing top 5 rows
```

Help page visits

In [88]:

```
help_df = df.groupby('userId').agg(Fsum(when(col('page') == 'Help', 1).otherwise(0)).ali
as('help_visits'))
help_df.show(5)

features.append(help_df)
```

```
+------+-----------+
|userId|help_visits|
+------+-----------+
|   296|          2|
|100010|          0|
|200002|          1|
|   125|          2|
|    51|          0|
+------+-----------+
only showing top 5 rows
```

Settings page visits

In [89]:

```
settings_df = df.groupby('userId').agg(Fsum(when(col('page') == 'Settings', 1).otherwise
(0)).alias('settings_visits'))
settings_df.show(5)

features.append(settings_df)
```

```
+------+---------------+
|userId|settings_visits|
+------+---------------+
|   296|              1|
|100010|              0|
|200002|              2|
|   125|              3|
|    51|              2|
+------+---------------+
only showing top 5 rows
```

Errors

In [90]:

```
errors_df = df.groupby('userId').agg(Fsum(when(col('page') == 'Error', 1).otherwise(0)).
alias('errors'))
errors_df.show(5)
```

```
features.append(errors_df)
```

```
+------+------+
|userId|errors|
+------+------+
|   296|     0|
|100010|     0|
|200002|     0|
|   125|     0|
|    51|     2|
+------+------+
only showing top 5 rows
```

Downgrade

In [91]:

```
downgrade_df = df.select('userId', 'downgrade').dropDuplicates()
downgrade_df.show(5)

features.append(downgrade_df)
```

```
+------+---------+
|userId|downgrade|
+------+---------+
|    73|        0|
|    19|        0|
|    69|        1|
|   288|        0|
|   209|        0|
+------+---------+
only showing top 5 rows
```

os

In [92]:

```
most_frequent_os_df = df.groupby(['userId', 'os']).count()\
                        .orderBy('count', ascending = False)\
                        .groupby('userId')\
                        .agg(first('os').alias('most_frequent_os'))
most_frequent_os_df.show(5)
```

```
+------+----------------+
|userId|most_frequent_os|
+------+----------------+
|100010|          iPhone|
|200002|          iPhone|
|   296|           MacOS|
|   125|           MacOS|
|   124|           MacOS|
+------+----------------+
only showing top 5 rows
```

In [93]:

```
# label encoding
os_indexer = StringIndexer(inputCol='most_frequent_os', outputCol='os_index')
os_indexed = os_indexer.fit(most_frequent_os_df).transform(most_frequent_os_df)
os_indexed.show(5)
```

```
+------+----------------+--------+
|userId|most_frequent_os|os_index|
+------+----------------+--------+
|100010|          iPhone|     2.0|
```

```
|200002|          iPhone|     2.0|
|   296|           MacOS|     1.0|
|   125|           MacOS|     1.0|
|   124|           MacOS|     1.0|
+------+----------------+--------+
only showing top 5 rows
```

In [94]:

```
# OneHotEncoder to os indexer
onehotencoder_os_vector = OneHotEncoder(inputCol= 'os_index', outputCol='os_vec')
os_df = onehotencoder_os_vector.fit(os_indexed).transform(os_indexed)
```

In [95]:

```
# filtered os_df
os_df = os_df.select(['userId', 'os_vec'])
os_df.show(5)

features.append(os_df)
```

```
+------+-------------+
|userId|       os_vec|
+------+-------------+
|100010|(5,[2],[1.0])|
|200002|(5,[2],[1.0])|
|   296|(5,[1],[1.0])|
|   125|(5,[1],[1.0])|
|   124|(5,[1],[1.0])|
+------+-------------+
only showing top 5 rows
```

location

In [96]:

```
location_df2 = df.select('userId', 'location').dropDuplicates()
location_df2.show(5)
```

```
+------+--------+
|userId|location|
+------+--------+
|   170|      MS|
|   209|      LA|
|    43|   MA-NH|
|100041|      FL|
|100036|      OK|
+------+--------+
only showing top 5 rows
```

In [97]:

```
# string index
location_indexer = StringIndexer(inputCol='location', outputCol='location_index')
location_indexed = location_indexer.fit(location_df2).transform(location_df2)
location_indexed.show(5)
```

```
+------+--------+--------------+
|userId|location|location_index|
+------+--------+--------------+
|   170|      MS|          26.0|
|   209|      LA|          25.0|
|    43|   MA-NH|          16.0|
|100041|      FL|           3.0|
|100036|      OK|          42.0|
+------+--------+--------------+
only showing top 5 rows
```

In [98]:

```
# onehotencoder to locationIndex
onehotencoder_location_vector = OneHotEncoder(inputCol="location_index", outputCol="locat
ion_vec")
location_encoded_df = onehotencoder_location_vector.fit(location_indexed).transform(locat
ion_indexed)
location_encoded_df = location_encoded_df.select(['userId', 'location_vec'])
location_encoded_df.show(5)
# add to features
features.append(location_encoded_df)
```

```
+------+---------------+
|userId|   location_vec|
+------+---------------+
|   170|(69,[26],[1.0])|
|   209|(69,[25],[1.0])|
|    43|(69,[16],[1.0])|
|100041| (69,[3],[1.0])|
|100036|(69,[42],[1.0])|
+------+---------------+
only showing top 5 rows
```

In [99]:

```
## Target(churn)
churn = df.select('userId', 'user_churn').dropDuplicates()
churn.show(5)
features.append(churn)
```

```
+------+----------+
|userId|user_churn|
+------+----------+
|100010|         1|
|200002|         1|
|   296|         1|
|   125|         0|
|   124|         1|
+------+----------+
only showing top 5 rows
```

In [100]:

```
features
```

Out[100]:

```
[DataFrame[userId: string, songs: bigint],
 DataFrame[userId: string, thumbs down: bigint],
 DataFrame[userId: string, thumbs_up: bigint],
 DataFrame[userId: string, days_since_registration: double],
 DataFrame[userId: string, round(avg(count), 0): double],
 DataFrame[userId: string, songs_in_playlist: bigint],
 DataFrame[userId: string, number_of_friends: double],
 DataFrame[userId: string, help_visits: bigint],
 DataFrame[userId: string, settings_visits: bigint],
 DataFrame[userId: string, errors: bigint],
 DataFrame[userId: string, downgrade: int],
 DataFrame[userId: string, os_vec: vector],
 DataFrame[userId: string, location_vec: vector],
 DataFrame[userId: string, user_churn: int]]
```

In [101]:

```
final_df = song_df

def merging_dataframes(a_df, b_df):
    '''
```

```
    INPUT:
    a_df, b_df - dataframes to be merged

    OUTPUT:
    merged_df - merged dataframe

    Description:
    Join dataframes
    '''

    merged_df = b_df.join(a_df, on=['userId'], how='left')

    return merged_df

for names in features[1:]:
    final_df = merging_dataframes(final_df, names)

# fill nans
final_df = final_df.na.fill(0)
final_df.show(5)
```

```
+------+----------+---------------+-------------+---------+------+--------------+-----
------+-----------------+-----------------+--------------------+----------------------
+---------+-----------+-----+
|userId|user_churn|   location_vec|       os_vec|downgrade|errors|settings_visits|help_vi
sits|number_of_friends|songs_in_playlist|round(avg(count), 0)|days_since_registration|thu
mbs_up|thumbs down|songs|
+------+----------+---------------+-------------+---------+------+--------------+-----
------+-----------------+-----------------+--------------------+----------------------
+---------+-----------+-----+
|100010|         1|(69,[11],[1.0])|(5,[2],[1.0])|        0|     0|              0|
0|              3.0|                1|                48.0|                   14.0|
4|          3|   96|
|200002|         1| (69,[4],[1.0])|(5,[2],[1.0])|        0|     0|              2|
1|              2.0|                6|                62.0|                   53.0|
15|          5|  310|
|   296|         1|(69,[45],[1.0])|(5,[1],[1.0])|        0|     0|              1|
2|              2.0|                3|                22.0|                   27.0|
8|          0|    0|
|   125|         0| (69,[1],[1.0])|(5,[1],[1.0])|        0|     0|              3|
2|              3.0|                2|                21.0|                  105.0|
3|          1|   62|
|   124|         1|(69,[39],[1.0])|(5,[1],[1.0])|        0|     0|             15|
10|             26.0|               45|               114.0|                  113.0|
102|         15| 1826|
+------+----------+---------------+-------------+---------+------+--------------+-----
------+-----------------+-----------------+--------------------+----------------------
+---------+-----------+-----+
only showing top 5 rows
```

In [102]:

```
final_df.printSchema()
```

```
root
 |-- userId: string (nullable = true)
 |-- user_churn: integer (nullable = true)
 |-- location_vec: vector (nullable = true)
 |-- os_vec: vector (nullable = true)
 |-- downgrade: integer (nullable = true)
 |-- errors: long (nullable = true)
 |-- settings_visits: long (nullable = true)
 |-- help_visits: long (nullable = true)
 |-- number_of_friends: double (nullable = false)
 |-- songs_in_playlist: long (nullable = true)
 |-- round(avg(count), 0): double (nullable = false)
 |-- days_since_registration: double (nullable = false)
 |-- thumbs_up: long (nullable = true)
 |-- thumbs down: long (nullable = true)
 |-- songs: long (nullable = true)
```

In [103]:

```
final_df.columns
```

Out[103]:

```
['userId',
 'user_churn',
 'location_vec',
 'os_vec',
 'downgrade',
 'errors',
 'settings_visits',
 'help_visits',
 'number_of_friends',
 'songs_in_playlist',
 'round(avg(count), 0)',
 'days_since_registration',
 'thumbs_up',
 'thumbs_down',
 'songs']
```

In [104]:

```
# numeric variables
corr_df = final_df.toPandas()
corr_df = corr_df.iloc[:, np.r_[1, 4:15]]
corr_df
```

Out[104]:

	user_churn	downgrade	errors	settings_visits	help_visits	number_of_friends	songs_in_playlist	round(avg(count), 0)	days_s
0	1	0	0	0	0	3.0	1	48.0	
1	1	0	0	2	1	2.0	6	62.0	
2	1	0	0	1	2	2.0	3	22.0	
3	0	0	0	3	2	3.0	2	21.0	
4	1	0	0	15	10	26.0	45	114.0	
...	...	...	...	...	...	...	...	...	
540	0	0	2	11	12	37.0	52	85.0	
541	0	0	0	2	2	12.0	17	34.0	
542	1	0	0	3	1	1.0	3	32.0	
543	0	0	0	0	0	0.0	0	0.0	
544	0	0	0	2	0	7.0	13	51.0	

545 rows × 12 columns

Correlation Matrix

In [105]:

```
# correlation matrix with numeric variables
corr_df.corrwith(corr_df['user_churn'])
```

Out[105]:

```
user_churn                1.000000
downgrade                -0.004142
errors                   -0.011416
settings_visits          -0.007469
help_visits              -0.034846
number_of_friends        -0.038760
```

```
songs_in_playlist        -0.056953
round(avg(count), 0)      0.080021
days_since_registration  -0.116095
thumbs_up                -0.056938
thumbs_down               0.080583
songs                    -0.031960
dtype: float64
```

In [106]:

```
# categorical variables
corr_df2= df.groupby(['userId', 'user_churn', 'location', 'os']).count()\
            .orderBy('count', ascending = False)\
            .groupby(['userId', 'user_churn', 'location'])\
            .agg(first('os').alias('most_frequent_os'))\
            .orderBy('userId').drop('userId').toPandas()
corr_df2
```

Out[106]:

	user_churn	location	most_frequent_os
0	1	MS	MacOS
1	0	TX	Windows
2	1	FL	MacOS
3	0	CA	Windows
4	1	FL	Windows
...	...	...	...
443	0	AZ	MacOS
444	0	NJ	MacOS
445	0	CA	Windows
446	0	TX	MacOS
447	0	CT	Windows

448 rows × 3 columns

In [107]:

```
# encode categorical variables
corr_df2['location'] =corr_df2['location'].astype('category').cat.codes
corr_df2['most_frequent_os'] =corr_df2['most_frequent_os'].astype('category').cat.codes
```

In [108]:

```
# corr_df2
corr_df2
```

Out[108]:

	user_churn	location	most_frequent_os
0	1	34	2
1	0	61	4
2	1	9	2
3	0	5	4
4	1	9	4
...	...	...	...
443	0	4	2
444	0	41	2
445	0	5	4

	user_churn	location	most_frequent_os
446	0	61	2
447	0	7	4

448 rows × 3 columns

In [109]:

```
# correlation matrix with categorical variables
corr_df2.corrwith(corr_df2['user_churn'])
```

Out[109]:

```
user_churn          1.000000
location            0.016293
most_frequent_os    0.109786
dtype: float64
```

- **From correlation matrix, I decide to keep them all first.**

VectorAssembler

In [111]:

```
# inputCols
inputCols = ['location_vec',
 'os_vec',
 'downgrade',
 'errors',
 'settings_visits',
 'help_visits',
 'number_of_friends',
 'songs_in_playlist',
 'round(avg(count), 0)',
 'days_since_registration',
 'thumbs_up',
 'thumbs_down',
 'songs']

# outputCol
outputCol = 'features'

# assembler
assembler = VectorAssembler(inputCols = inputCols, outputCol = outputCol)

# use VectorAssembler to transform the dataset
data = assembler.transform(final_df).select(['user_churn', 'features'])
data.show(5)
```

```
+----------+--------------------+
|user_churn|            features|
+----------+--------------------+
|         1|(85,[11,71,78,79,...|
|         1|(85,[4,71,76,77,7...|
|         1|(85,[45,70,76,77,...|
|         0|(85,[1,70,76,77,7...|
|         1|(85,[39,70,76,77,...|
+----------+--------------------+
only showing top 5 rows
```

StandardScaler

In [112]:

```
# standardize the feautres
scaler = StandardScaler(
```

```
    inputCol = 'features',
    outputCol = 'scaledFeatures',
    withMean = True,
    withStd = True
).fit(data)

# when we transform the dataframe, the old feature will still remain in it
df_scaled = scaler.transform(data)
df_scaled.show(5)
```

```
+----------+--------------------+--------------------+
|user_churn|            features|      scaledFeatures|
+----------+--------------------+--------------------+
|         1|(85,[11,71,78,79,...|[-0.4143999704435...|
|         1|(85,[4,71,76,77,7...|[-0.4143999704435...|
|         1|(85,[45,70,76,77,...|[-0.4143999704435...|
|         0|(85,[1,70,76,77,7...|[-0.4143999704435...|
|         1|(85,[39,70,76,77,...|[-0.4143999704435...|
+----------+--------------------+--------------------+
only showing top 5 rows
```

In [113]:

```
data = df_scaled.select(['user_churn', 'scaledFeatures'])
data.show(5)
```

```
+----------+--------------------+
|user_churn|      scaledFeatures|
+----------+--------------------+
|         1|[-0.4143999704435...|
|         1|[-0.4143999704435...|
|         1|[-0.4143999704435...|
|         0|[-0.4143999704435...|
|         1|[-0.4143999704435...|
+----------+--------------------+
only showing top 5 rows
```

Modeling

In [114]:

```
# function for printing results
def evaluate_print(model_result, model_name, start, end):
    '''
    INPUT:
    model_result : result
    model_name : string
    start, end : training time start and end

    OUTPUT: list

    Description:
    The function return list of results and prints results and total time of the training

    '''

    evaluator = MulticlassClassificationEvaluator(predictionCol='prediction')
    evaluator.setLabelCol('user_churn')
    accuracy = evaluator.evaluate(model_result, {evaluator.metricName : 'accuracy'})
    f1 = evaluator.evaluate(model_result, {evaluator.metricName : 'f1'})
    time = (end - start)/60

    result = [model_name, round(accuracy,3), round(f1,3), round(time,1)]

    print('{} performance metrics:'.format(model_name))
    print('Accuracy: {}'.format(accuracy))
    print('F-1 Score: {}'.format(f1))
    print('Total training time: {} minutes'.format(time))
```

```
    return result
```

In [115]:

```
# function for printing reports
def reports(model_result):
    '''
    INPUT:
    model_result : list

    OUTPUT: None

    Description:
    The function aggregate and prints results of models
    '''
    print(model_result[0])
    print('Accuracy: {}'.format(model_result[1]))
    print('F-1 Score: {}'.format(model_result[2]))
    print('Total training time: {} minutes'.format(model_result[3]))

    print()
```

Baseline Model

Baseline model will be target values are all 0s which means `**no users has canceled this subscription**`. We compare `**Logistic Regression, Random Forest, Gradient Boosted Trees, and Support Vector Machine**` with baseline models. If we achieve a better score than the baselineline, it is good.

In [116]:

```
# split test and train set
train, test = data.randomSplit([0.6, 0.4], seed=42)
```

In [117]:

```
#baseline model
baseline = test.withColumn('prediction', lit(0.0))
baseline.show(5)
```

```
+----------+--------------------+----------+
|user_churn|      scaledFeatures|prediction|
+----------+--------------------+----------+
|         1|[-0.4143999704435...|       0.0|
|         1|[-0.4143999704435...|       0.0|
|         0|[-0.4143999704435...|       0.0|
|         0|[-0.4143999704435...|       0.0|
|         1|[-0.4143999704435...|       0.0|
+----------+--------------------+----------+
only showing top 5 rows
```

In [118]:

```
# print baseline model
baseline_result = evaluate_print(baseline, 'Baseline', 0, 0)
```

```
Baseline performance metrics:
Accuracy: 0.7788018433179723
F-1 Score: 0.6819560182421622
Total training time: 0.0 minutes
```

In [119]:

```
#start training
START = time.time()
```

Logistic Regression

In [120]:

```
numFolds = 3
lr = LogisticRegression(maxIter=10, labelCol='user_churn', featuresCol='scaledFeatures')
evaluator = MulticlassClassificationEvaluator(labelCol='user_churn')

pipeline = Pipeline(stages=[lr])
lr_paramGrid = (ParamGridBuilder()
                .addGrid(lr.regParam, [0.1, 0.01, 0.001])
                .build())


crossval = CrossValidator(
    estimator=pipeline,
    estimatorParamMaps=lr_paramGrid,
    evaluator=evaluator,
    numFolds=numFolds)

lr_start = time.time()
lr_model = crossval.fit(train)
lr_end = time.time()
```

In [121]:

```
lr_results = lr_model.transform(test)

lr_safe = evaluate_print(lr_results, 'Logistic Regression', lr_start, lr_end)

best_param = list(lr_model.getEstimatorParamMaps()[np.argmax(lr_model.avgMetrics)].value
s())
print('Best regression parameter is {}'.format(best_param[0]))
```

```
Logistic Regression performance metrics:
Accuracy: 0.7926267281105991
F-1 Score: 0.7802268983769177
Total training time: 12.614573101202646 minutes
Best regression parameter is 0.001
```

Random Forest

In [122]:

```
numFolds = 3
rf = RandomForestClassifier(labelCol='user_churn', featuresCol='scaledFeatures', seed =
42)
evaluator = MulticlassClassificationEvaluator(labelCol='user_churn')

pipeline = Pipeline(stages=[rf])
rf_paramGrid = (ParamGridBuilder()
             .addGrid(rf.numTrees, [10,20])
             .addGrid(rf.maxDepth, [10,20])
             .build())

crossval = CrossValidator(
    estimator=pipeline,
    estimatorParamMaps=rf_paramGrid,
    evaluator=evaluator,
    numFolds=numFolds)

rf_start = time.time()
rf_model = crossval.fit(train)
rf_end = time.time()
```

In [123]:

```
rf_results = rf_model.transform(test)
```

```
rf_safe = evaluate_print(rf_results, 'Random Forest', rf_start, rf_end)

best_param = list(rf_model.getEstimatorParamMaps()[np.argmax(rf_model.avgMetrics)].value
s())
print('Best number of trees {}, best depth {}'.format(best_param[0], best_param[1]))
```

```
Random Forest performance metrics:
Accuracy: 0.8341013824884793
F-1 Score: 0.7998007223813676
Total training time: 16.8668750723203 minutes
Best number of trees 20, best depth 10
```

Gradient Boosted Trees

In [124]:

```
numFolds = 3
gbt = GBTClassifier(labelCol='user_churn', featuresCol='scaledFeatures', seed = 42)
evaluator = MulticlassClassificationEvaluator(labelCol='user_churn')

pipeline = Pipeline(stages=[gbt])
gbt_paramGrid = (ParamGridBuilder()
                 .addGrid(gbt.maxIter, [10,20])
                 .addGrid(gbt.maxDepth, [10,20])
                 .build())

crossval = CrossValidator(
    estimator=pipeline,
    estimatorParamMaps=gbt_paramGrid,
    evaluator=evaluator,
    numFolds=numFolds)

gbt_start = time.time()
gbt_model = crossval.fit(train)
gbt_end = time.time()
```

In [125]:

```
gbt_results = gbt_model.transform(test)

gbt_safe = evaluate_print(gbt_results, 'Gradient Boosted Trees', gbt_start, gbt_end)

best_param = list(gbt_model.getEstimatorParamMaps()[np.argmax(gbt_model.avgMetrics)].val
ues())
print('Best number of iterations {}, best depth {}'.format(best_param[0], best_param[1])
)
```

```
Gradient Boosted Trees performance metrics:
Accuracy: 0.7142857142857143
F-1 Score: 0.7163508260447035
Total training time: 51.39637206792831 minutes
Best number of iterations 10, best depth 10
```

Support Vector Machine

In [126]:

```
numFolds = 3
svc = LinearSVC(labelCol='user_churn', featuresCol='scaledFeatures')
evaluator = MulticlassClassificationEvaluator(labelCol='user_churn')

pipeline = Pipeline(stages=[svc])
svc_paramGrid = (ParamGridBuilder()
                 .addGrid(svc.maxIter, [5,10])
                 .build())

crossval = CrossValidator(
    estimator=pipeline,
```

```
    estimatorParamMaps=svc_paramGrid,
    evaluator=evaluator,
    numFolds=numFolds)

svc_start = time.time()
svc_model = crossval.fit(train)
svc_end = time.time()
```

In [127]:

```
svc_results = svc_model.transform(test)

svc_safe = evaluate_print(svc_results, "Support Vector Machine", svc_start, svc_end)

best_param = list(svc_model.getEstimatorParamMaps()[np.argmax(svc_model.avgMetrics)].val
ues())
print('Best number of iterations {}'.format(best_param[0]))
```

```
Support Vector Machine performance metrics:
Accuracy: 0.7788018433179723
F-1 Score: 0.755875542162308
Total training time: 11.776197810967764 minutes
Best number of iterations 10
```

In [128]:

```
# finish training
END = time.time()
```

Evaluate models

In [129]:

```
# performance comparison
results_models_list = [baseline_result, lr_safe, rf_safe, gbt_safe, svc_safe]
# xaxis values, yxais values
F1_score, model, time =[],[],[]
for res in results_models_list:
    reports(res)
    model.append(res[0])
    F1_score.append(res[2])
    time.append(res[3])
print('=='*55)
print('F1_score')
print(F1_score)
print('=='*55)
print('model')
print(model)
print('=='*55)
print('time(minuetes)')
print(time)
```

```
Baseline
Accuracy: 0.779
F-1 Score: 0.682
Total training time: 0.0 minutes

Logistic Regression
Accuracy: 0.793
F-1 Score: 0.78
Total training time: 12.6 minutes

Random Forest
Accuracy: 0.834
F-1 Score: 0.8
Total training time: 16.9 minutes

Gradient Boosted Trees
Accuracy: 0.714
F-1 Score: 0.716
```

```
Total training time: 51.4 minutes

Support Vector Machine
Accuracy: 0.779
F-1 Score: 0.756
Total training time: 11.8 minutes

===========================================================================================================
F1_score
[0.682, 0.78, 0.8, 0.716, 0.756]
===========================================================================================================
model
['Baseline', 'Logistic Regression', 'Random Forest', 'Gradient Boosted Trees', 'Support Vector Machine']
===========================================================================================================
time(minuetes)
[0.0, 12.6, 16.9, 51.4, 11.8]
```

In [130]:

```python
# model dataframe
d = {'model': model, 'F1_score': F1_score, 'time': time}
model_df =pd.DataFrame(d)
model_df1 = model_df.sort_values(['F1_score'], ascending=False).reset_index(drop=True)
model_df1
```

Out[130]:

	model	F1_score	time
0	**Random Forest**	**0.800**	**16.9**
1	**Logistic Regression**	**0.780**	**12.6**
2	**Support Vector Machine**	**0.756**	**11.8**
3	**Gradient Boosted Trees**	**0.716**	**51.4**
4	**Baseline**	**0.682**	**0.0**

- **Since we have imbalanced data, F1 score is choosen to be the metric to evaluate the peformance of models.**

In [131]:

```python
# f1 score by models
colors = ['lightslategray',] * 5
colors[0] = 'crimson'
fig = px.bar(model_df1, x='F1_score', y='model', orientation='h', template='simple_white',
             title='F1 score by models', color='model', color_discrete_sequence=colors)
fig.update_layout(xaxis=dict(title='F1 score'),
                  yaxis=dict(title='model', ticks=''),
                  width=800, height=500,
                  showlegend=False)
```

- **All F1 scores of four models are beyond baseline model. Among of them, **random forest** is the best with F1 score 0.818.**

In [132]:

```
model_df2 = model_df.sort_values(['time'], ascending=False).reset_index(drop=True)
model_df2
```

Out[132]:

	model	F1_score	time
0	Gradient Boosted Trees	0.716	51.4
1	Random Forest	0.800	16.9
2	Logistic Regression	0.780	12.6
3	Support Vector Machine	0.756	11.8
4	Baseline	0.682	0.0

In [133]:

```
# time by models
colors = ['lightslategray',] * 5
colors[0] = 'crimson'

fig = px.bar(model_df2, x='time', y='model', orientation='h', template='simple_white',
             title='time(minutes) by models', color='model', color_discrete_sequence= co
lors)
fig.update_layout(xaxis=dict(title='time'),
                  yaxis=dict(title='model', ticks=''),
                  width=800, height=500,
                  showlegend=False)
```

- In the aspect of time, `**Gradient Boosted Trees**` takes much **longer** than other three models.
- Overall, `Radom Forest` is the **best model** among four models `(Gradient Boosted Trees, Random Forest, Logistic Regression, Support Vector Machine)`

Conclusion

In this project, I have learned how to deal with **big data using Pyspark**. This dataset allows me to practice customized data visualizations and churn analysis by creating predictive feautres. I trained four models: Gradient Boosted Trees, Random Forest, Logistic Regression, Support Vector Machine. `**Random Forest**` is proven to be **the best model** among them. The main chanllenges for me is feature engineering especially feature selection. In the future, I plan to group location variable to reduce dimensions and then perform chisq test on it. Also, more advanced ML algorithms can be applied in this dataset.