Music Churn Prediction

Team Members

Samyukth Suresh - ss14719

Garima Choudhary - gc2900

VamshiSai Perumandla - vp2361

Dhruv Jain - dj2136

Abstract

A subscription-based business model is currently widely used by music streaming companies. Such businesses can better understand their client retention and how to keep customers by using Churn analysis. Customer turnover rates are decreased via Churn analysis, which also enables businesses to identify the activities that must be taken to avoid significant revenue losses as a result of customer churn. Churn rate is the frequency with which clients leave a business over a predetermined time frame. The amount of subscribers that discontinue their subscriptions or do not renew them may also be considered churn. The more clients who cease making purchases from your company, the greater your churn rate. To assess the success of your marketing initiatives and the general contentment of your clients, it is critical to comprehend your customer churn. Because subscription business models are so common, it's crucial for many companies to comprehend where, how, and why their consumers might be leaving. For a music streaming service to be successful, it is essential to identify consumers who may downgrade their subscription from free to premium or quit it altogether. These users are known as people who are prone to churn.

Background

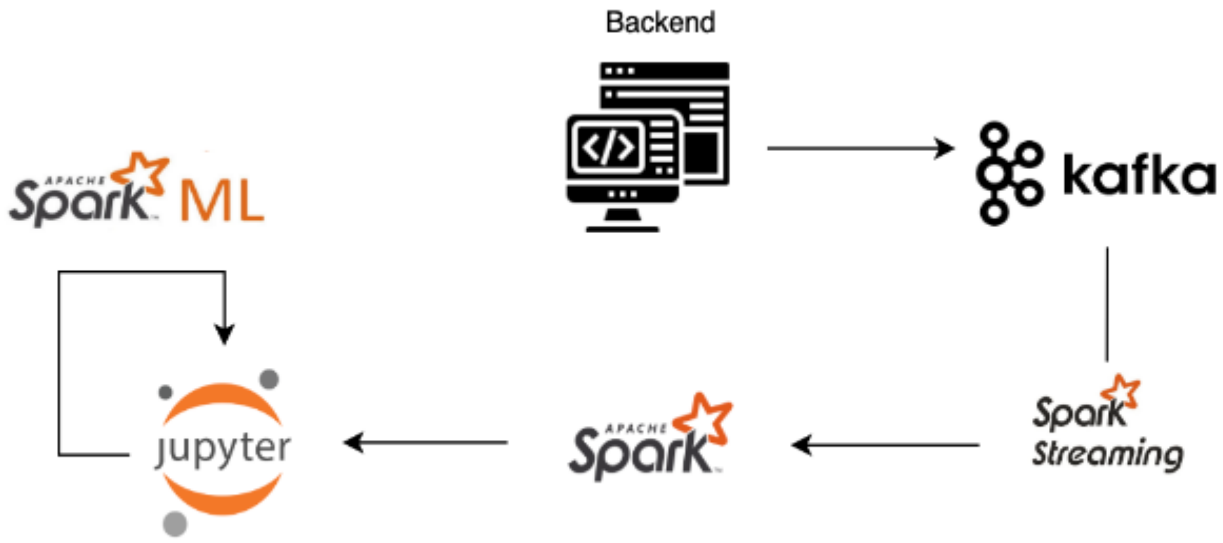
We took a fictional music streaming service called Sparkify for our project (like Spotify, Pandora, etc.). Users of Sparkify have the option of paying for a membership or to listen to music for free. The free users will hear advertisements whereas the subscription subscribers (or paying users) will hear no adverts when listening to music. To listen to music on the service, users must log in.

Users have the option to at any time perform any of the following from a subscription perspective:

1) Upgrade the free plan, move up to the paid plan
2) Downgrade to the paid plan to the free tier
3) Delete their account and stop using the service.

We're looking for users who might delete their accounts and stop using the service. We may motivate this population by offering discounts or other prizes if we were to identify them in advance. This can convince them to stay, giving us a base of devoted clients, which is essential for a business's expansion.

System Design Architecture



We deployed a complete end to end data pipeline that streams data from a log file simulating the application backend to the Kafka cluster. There will be a dedicated Spark streaming application to consume the stream message and ingest to local file storage Spark, stored as json files, and retrieved with Spark Dataframe API via Jupyter to perform Churn Prediction analysis. The experiment's main goal is to identify and forecast the variables affecting turnover rate.

Kafka Streaming

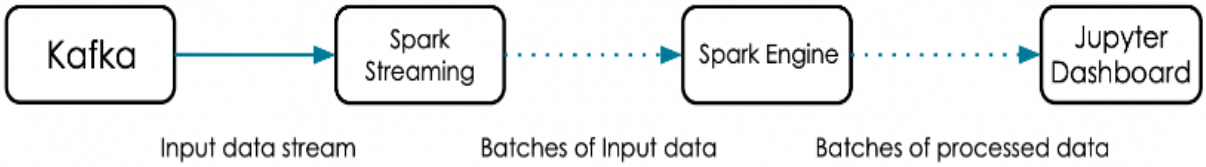
The message is prepared for transmission as a stream of data by the kafka producer server after parsing the json log file. The application's parameters (specific to Kafka and Spark) are listed in the app.cfg file. The reasoning behind sending the kafka message is contained in the producer notebook. The message sent by the kafka producer is tested using a model notebook for a kafka consumer. We take the message from Kafka, convert it, and store it on local file storage in the Spark streaming notebook.

Why Big Data?

Sparkify music streaming log data is so huge, fast or complex that it’s difficult or impossible to process using traditional methods. It produces the user log data in the backend, and this log data is generated continuously in milliseconds. It's so huge that for each user in each click, in the backend it generates a log. This log various features related to the user using the app such as user name, gender, location, activity in app, etc. Large-scale data access and storage for analytics has been practiced for a very long time. Big data technology will be used to manage the music streaming information that users provide when using the app. Although the app regularly interacts with millions of users and offers subscriptions to the majority of them, using big data to grow and manage their business would be quite beneficial. It was discovered that combining big data analytics with machine learning was an effective method for detecting churn. Sparkify log data is generated for N number of users which is massive and difficult to compute. That's why we require big data technologies such as Kafka and Spark to process this data in an efficient way and make churn predictions.

Why Kafka?

Kafka loads our real time user log data from backend with low latency which makes it more reliable than similar messaging services available. Apache kafka is massively scalable because it allows the log data generated by the sparkify streaming app to be distributed across multiple servers. As a message broker, it lets applications publish or subscribe to one or multiple event streams and replicates data to support multiple subscribers.



Why Spark?

Spark/Spark ML and its ML libraries are specifically designed for handling and processing large amounts of data like logs in real-time. It is highly scalable, fault-tolerant, and can process data in parallel, which makes it well suited for handling the high-volume and fast-paced nature of our log data to help with churn prediction.

Dataset (Schema)

Our dataset is a JSON file which contains 26259199 records and takes 12 GB storage. It contains the following columns and schema

- artist: Artist name (ex. Daft Punk)
- auth: User authentication status (ex. Logged)
- firstName: User first name (ex. Colin)
- gender: Gender (ex. F or M)
- itemInSession: Item count in a session (ex. 52)

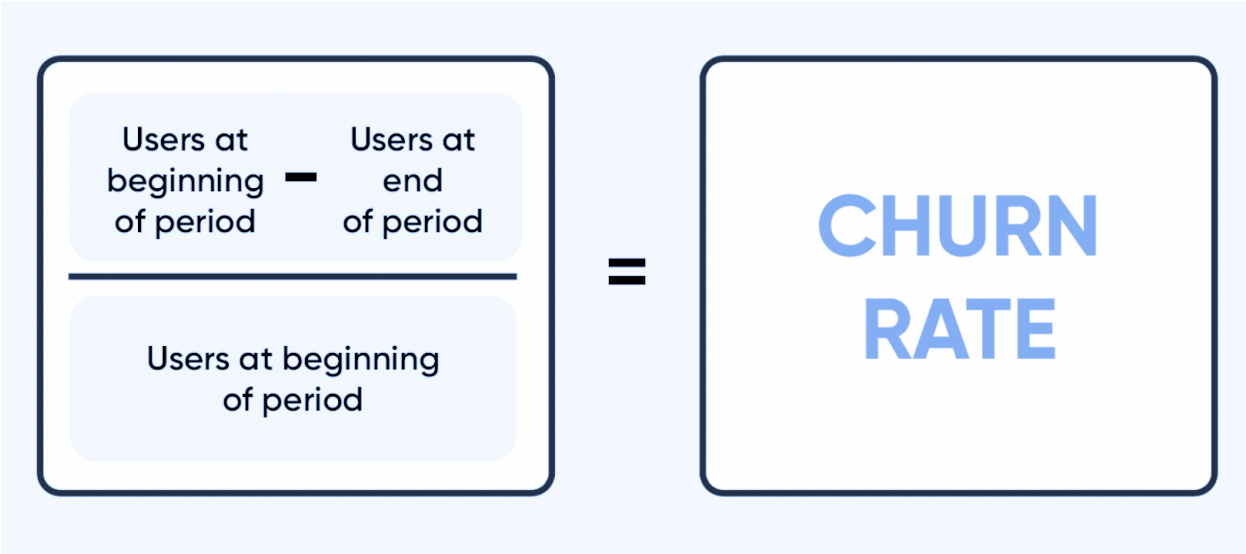
- lastName: User last name (ex. Freeman)
- length: Length of song (ex. 223.60771)
- level: User plan (ex. paid)
- location: User's location (ex. Bakersfield)
- method: HTTP method (ex. PUT)
- page: Page name (ex. NextSong)
- registration: Registration timestamp (unix timestamp) (ex. 1538173362000)
- sessionId: Session ID (ex. 29)
- song: Song (ex. Harder Better Faster Stronger)
- status: HTTP status (ex. 200)
- ts: Event timestamp(unix timestamp) (ex. 1538352676000)
- userAgent: User's browser agent (ex. Mozilla/5.0 (Windows NT 6.1; WOW64; rv:31.0) Gecko/20100101 Firefox/31.0
- userId: User ID (ex. 30)

```
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)
```

Why is Churning Prediction important?

Churn refers to the action of canceling a user's account on the platforms. Another term for a churned user is one that has canceled their account and left the platform. Churn rate, also known as the rate of attrition, is the percentage of users who stop using an app within a given period.

For an app to grow, the number of retained users must be higher than the number of users who leave. Every lost user has a different reason for uninstalling your app: not enough device space, got frustrated with bugs or a confusing UI, or they just lost interest in your app.



Data Investigation

1. Load the Data (Spark)

Pyspark Query

```
def spark_read(spark):
    return spark.read.option('inferSchema', 'true').option('header',
'true').option('encoding', 'utf-8')
```

Pyspark Query

```
def load_full_json_file(spark):
    df =spark_read(spark).json('sparkify_event_data.json')
    return df
```

2. We obtained a total data count of 26 million records.

3. Keeping track of user sessions that are logged in. Consequently, we must remove guest and logged-out users because we won't be able to get future sessions for these user IDs.

We use the following Pyspark Queries,

Pyspark Query - df.select('auth').distinct().collect()

Pyspark Query - df.where(df.auth.isin(['Guest', 'Logged Out'])).count()

4. Checking if records have user IDs present or not ?

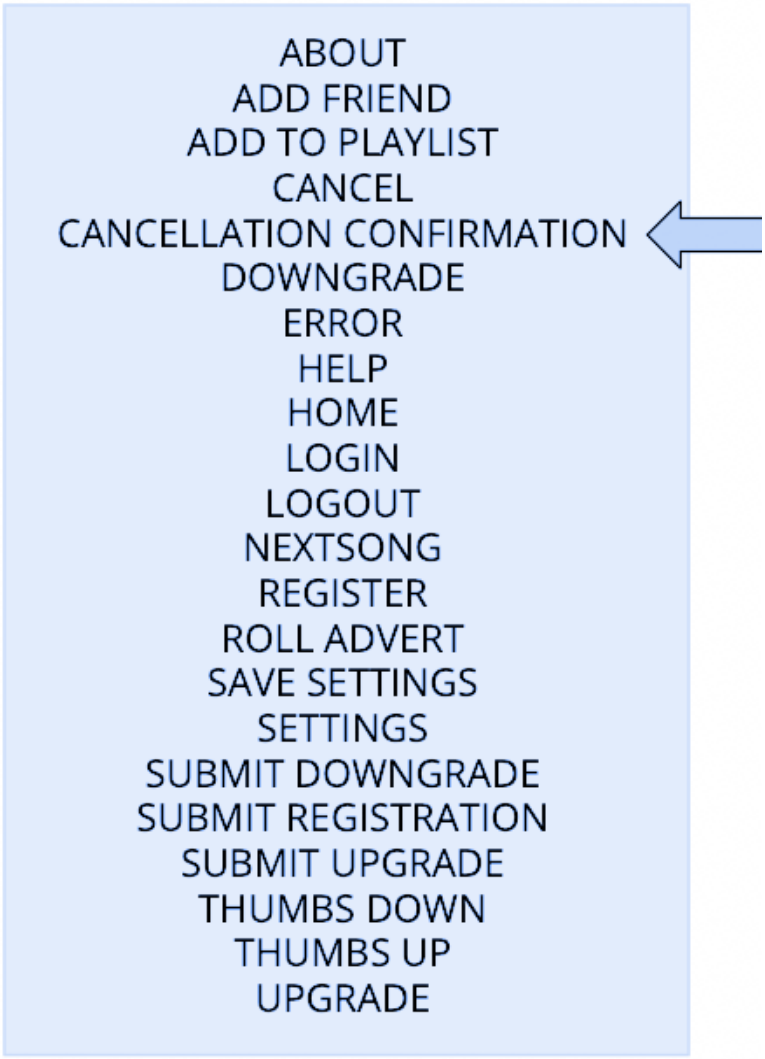
Pyspark Query - df.where(col('userId').isNull()).count()

5. Creating more user columns by using the timestamp and page analysis. By extracting the window and building the feature set, this is accomplished.

length	level	location	method	...	userAgent	userId	date	previous_page	last_event_ts	last_page	register_page	first_ts	ts_elapsed	session_duration
NaN	free	Hartford-West Hartford-East Hartford, CT	GET	...	"Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebK...	1528635	2018-10-01	None	1543412707000	Roll Advert	None	1538400348000	0	1365000
196.28363	free	Hartford-West Hartford-East Hartford, CT	PUT	...	"Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebK...	1528635	2018-10-01	Home	1543412707000	Roll Advert	None	1538400348000	32000	1365000
251.55873	free	Hartford-West Hartford-East Hartford, CT	PUT	...	"Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebK...	1528635	2018-10-01	NextSong	1543412707000	Roll Advert	None	1538400348000	228000	1365000
		Hartford-			"Mozilla/5.0									

Data Visualisation

Following are all the events in the application that are being logged for each user:

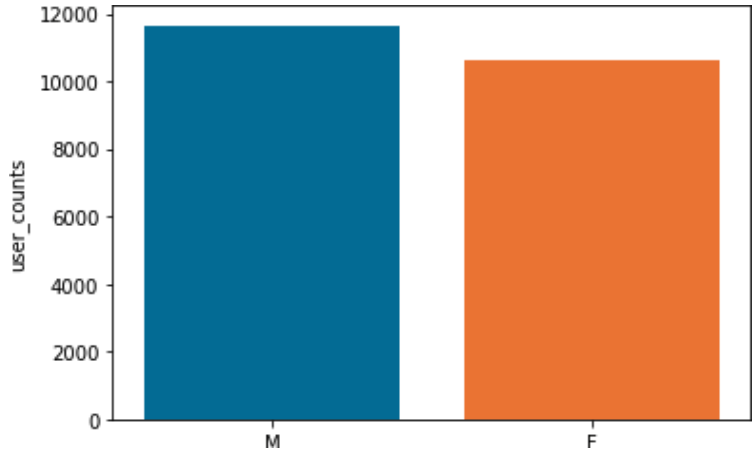


We could pay close attention to Cancellation Confirmation because it's something we never want to see a user do. This event is used to flag and build a churn column.

To determine how many people have truly abandoned the music app and how many are still using it, the data division will examine the user churn ratio

Gender Count

The ratio divide in our dataset is caused by the fact that the dataset includes both male and female users.



As a result, we can observe that in our dataset, there are more male users than female users.

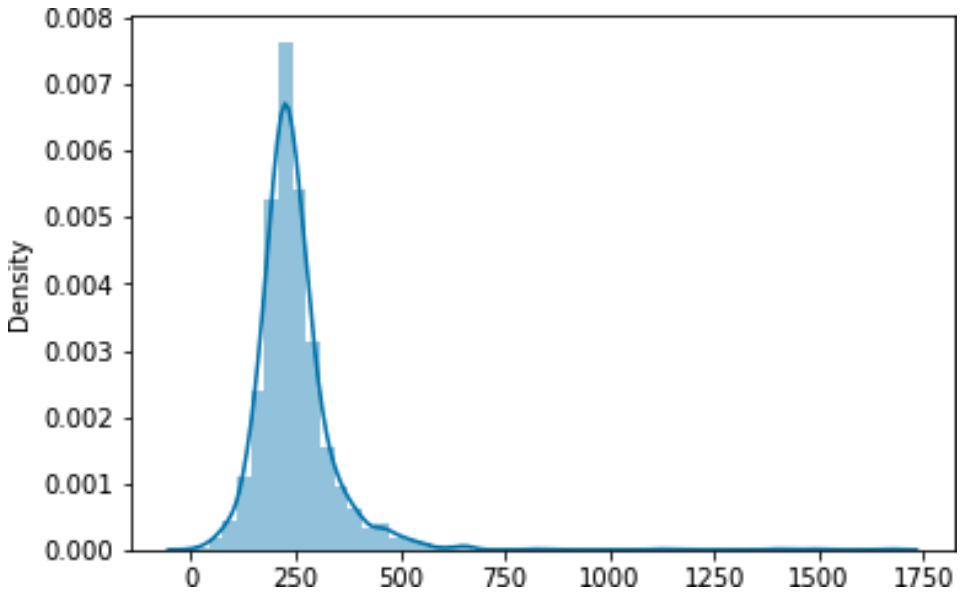
Unique user sessions (Count)

The dataset comprises values for each page visited by each user as well as values for a large number of sessions. As a result, we count the number of unique sessions for each user using the PySpark Query.

```
+-----------+
|item_counts|
+-----------+
|       1429|
+-----------+
```

Unique user sessions (Length)

The dataset includes values for each page visited by each user as well as values for a large number of sessions. As a result, we count the number of unique session length for each user using the PySpark Query. Especially, Sorting the longest period of time a user spent using the app during a single session.



Length looks like a gaussian distribution with a long write tail. Most length values are concentrated between 0 and 500.

User Level (Free Vs Paid)

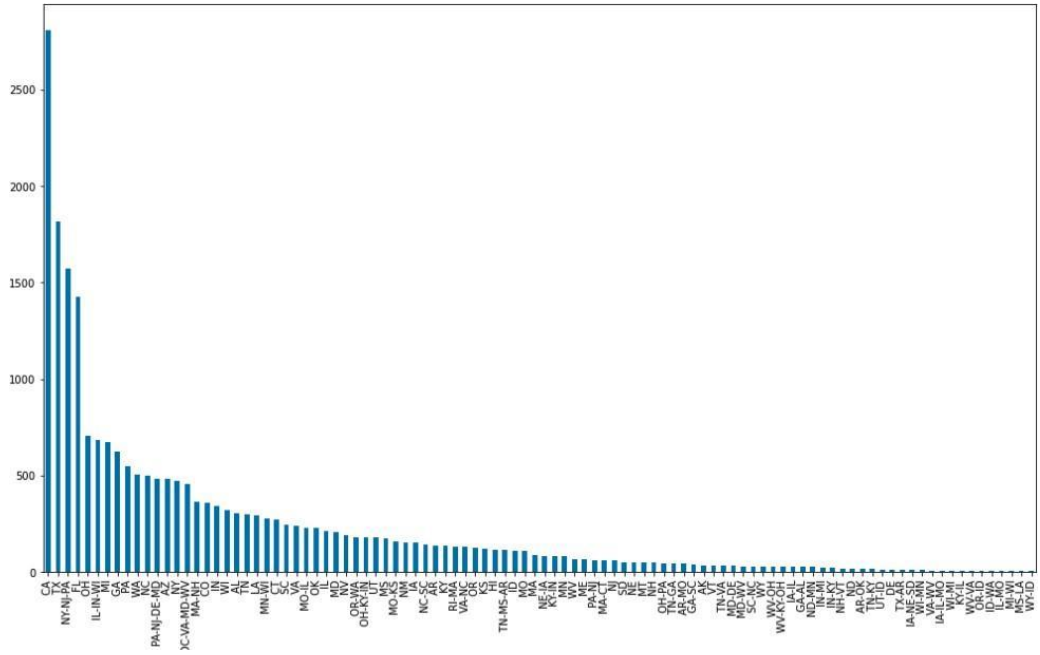
It would be beneficial to ascertain if the user on the app is the cause for the user to churn as they are either a paying subscriber or a free user. In our event log distribution, we would investigate the distribution of the free and paid subscribers.

```
+-----+-----------+
|level|user_counts|
+-----+-----------+
| free|      18793|
| paid|      16185|
+-----+-----------+
```

Additionally, it was discovered that the user interaction figure was 12700, which represents the time that these free app users interacted with those who had paid subscriptions.

Local Analysis (State)

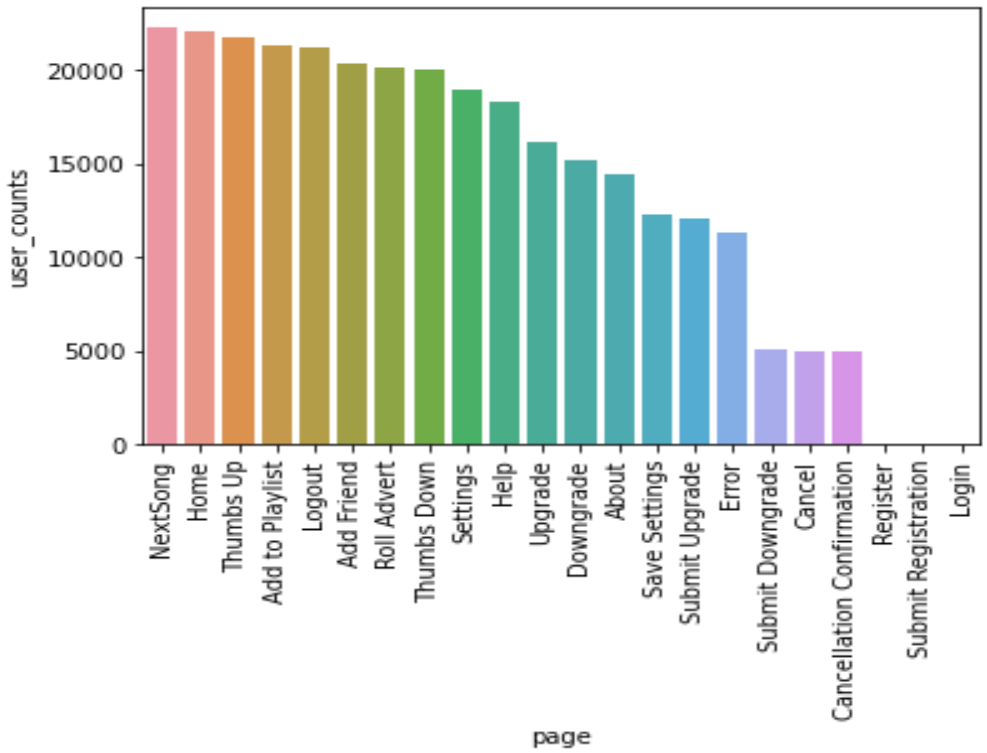
Finding out where the users of our app are physically located is crucial. The marketing staff would benefit from having a better understanding of how to offer discounts or free trials to increase the customer base in those region



The largest user base is in California, followed by Texas and New York State.

Page Count Analysis

The total number of users who accessed various pages or the manner in which they used the app, most likely the page events that were accessed.



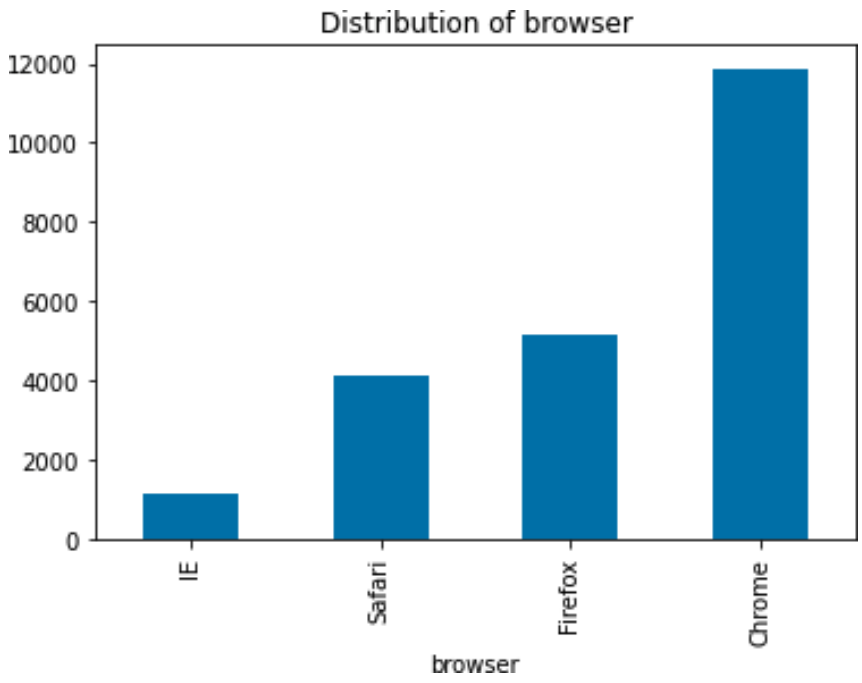
The majority of users are either in NextSong, Home, Thumbs Up, or Add to Playlist. When there aren't many cancellations, or in Submit Downgrade or Cancellation Confirmation.

User Device Analysis

The user device analysis is a crucial component of our prediction of churn since it enables us to determine whether consumers are leaving because of the device they are currently using. Because each device has a unique user interface, this may be influencing platform user behavior.

userAgent	user_counts
"Mozilla/5.0 (Win...	2024
Mozilla/5.0 (Wind...	1661
"Mozilla/5.0 (Win...	1395
"Mozilla/5.0 (Mac...	1340
"Mozilla/5.0 (Mac...	1229
"Mozilla/5.0 (Mac...	1084
Mozilla/5.0 (Maci...	1000
"Mozilla/5.0 (Mac...	998
"Mozilla/5.0 (Mac...	913
"Mozilla/5.0 (Win...	795
Mozilla/5.0 (Wind...	529
"Mozilla/5.0 (iPh...	517
"Mozilla/5.0 (Win...	510
Mozilla/5.0 (Wind...	487
"Mozilla/5.0 (Win...	410
Mozilla/5.0 (X11;...	361
Mozilla/5.0 (Wind...	317
"Mozilla/5.0 (Win...	281
"Mozilla/5.0 (iPa...	259
Mozilla/5.0 (Wind...	239

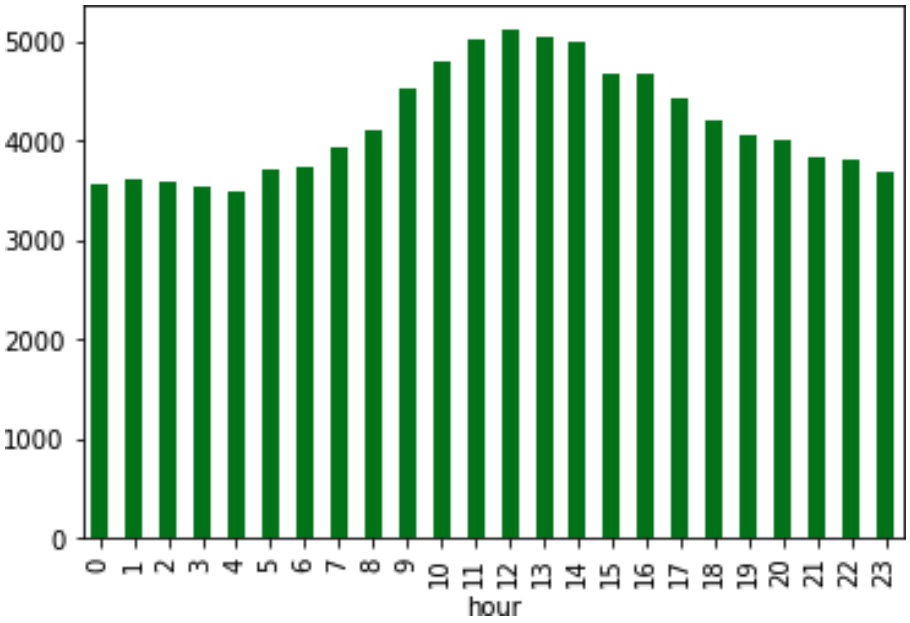
Identifying and Classifying the devices that each user uses when they are present on the platform

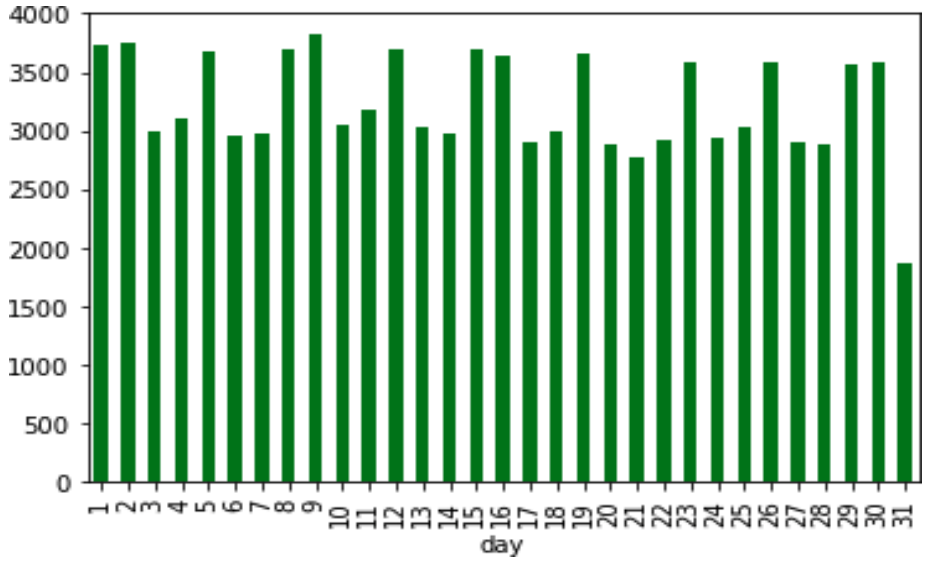


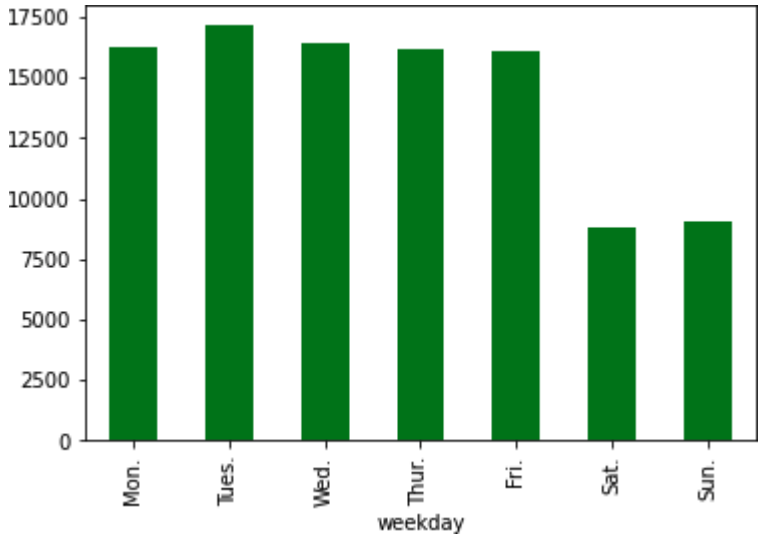
It is clear that Chrome is by far the most popular browser, followed by Firefox and Safari. IE is seldom ever used.

User Analysis (Day & Time)

Users may understand user timestamp behavior on the platform with the aid of the charts below. We can monitor when the user app has been most active with its assistance. How frequently does the customer utilize the music app?







Users' actions are cyclical, and they utilize Sparkify more frequently throughout the week than on the weekends. They utilize Sparkify more frequently after 14 o'clock in a single day.

Effect of each feature on churn (Analysis)

Let's see how these characteristics affected the churn. The references listed below show how each characteristic in our event dataset was utilized to influence user churn. For example, Pyspark queries were used to determine whether male users left the site more frequently than other female users or whether user session length had any bearing on churn prediction.

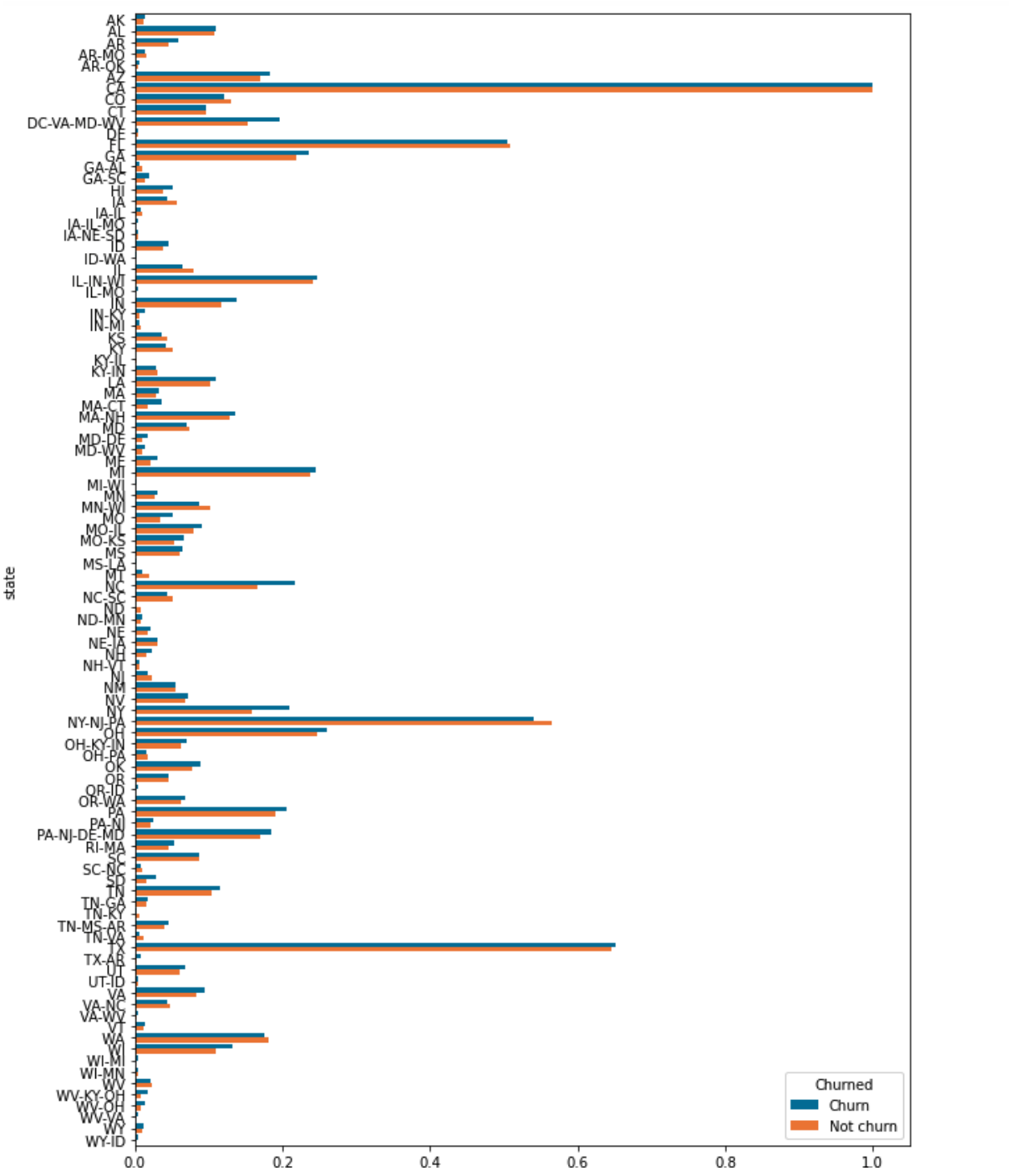
Churned	level	count
Churn	paid	1239
Churn	free	3764
Not churn	free	13484
Not churn	paid	3791

Churned	mean_length	stdev_length	max_length	min_length
Churn	248.680031572098	97.31322630186042	3024.66567	0.522
Not churn	248.73752863664356	97.28014262879445	3024.66567	0.522

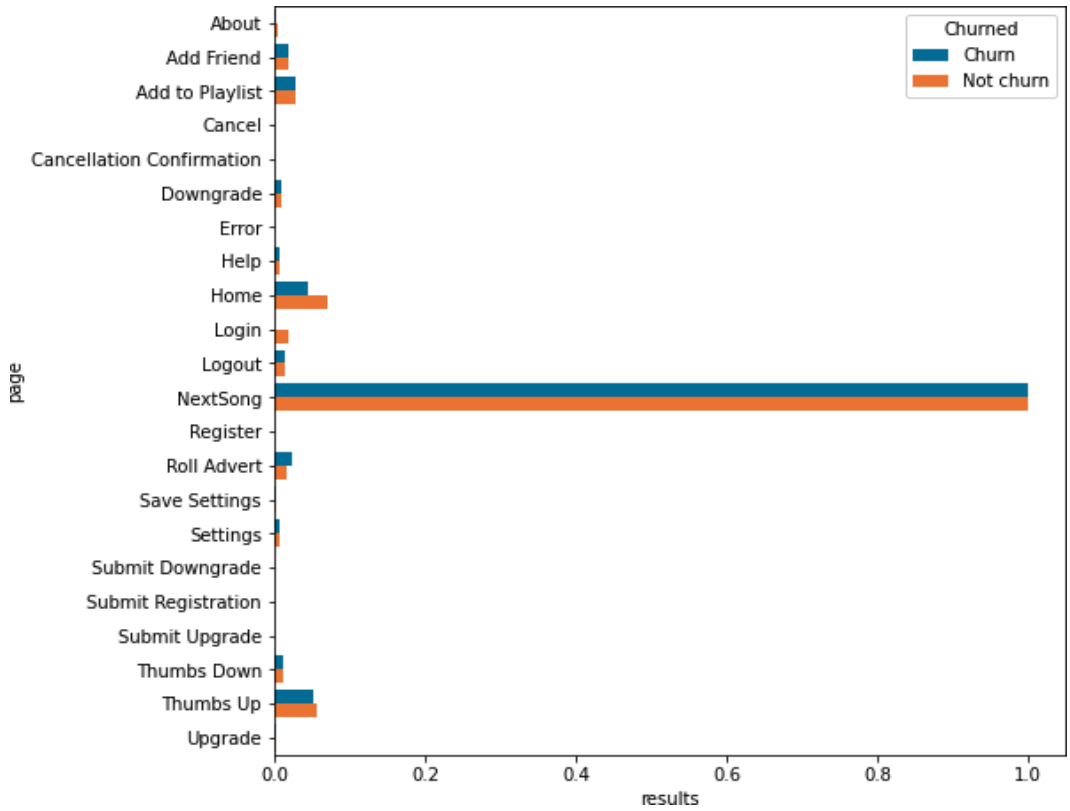
Churned	gender	count
Churn	F	2347
Churn	M	2656
Not churn	null	1
Not churn	M	8995
Not churn	F	8279

Churned	auth	count
Churn	Logged In	5003
Not churn	Logged In	17274
Not churn	Logged Out	1

Location (Churn Effect)



Page (Churn Effect)



The pages for NextSong, Thumbs Up/Down, Home, and Add to Playlist appear to influence churn or not.

Feature Engineering

The practice of applying domain expertise of the data to develop features that enable machine learning algorithms is known as feature engineering. If feature engineering is done properly, it can boost the predictive capacity of machine learning algorithms by generating features from raw data that aid in the machine learning process. It takes art to design features. In order to create a new collection of features that were more useful in predicting user churn, we altered the original set of features in the ways that are described below.

In this section, we build new features on top of the ones that already exist. For instance, the quantity of times a user has an event on the Submit Downgrade page becomes the quantity of reductions. Some features, such as firstName and lastName, are disregarded because churn makes them seem inapplicable. Others, such as location and userAgent, are difficult to extract crucial information.

User Session (Count of each page visit as feature set for new data)

Based on the user activities recorded in the event logs, new column features may be generated using the PySpark order by feature.

```
df_sessions = df.orderBy(df.sessionId).groupBy('sessionId', 'userId').agg(
        smax(df.ts).alias('max_event_ts'),
        smin(df.ts).alias('min_event_ts'),
        ssum(df.length).alias('session_n_total_playback'), # Based on songs length
        count(when(df.page == 'Thumbs Up', True)).alias("session_n_likes"),
        count(when(df.page == 'Thumbs Down', True)).alias("session_n_dislikes"),
        count(when(df.page == 'NextSong', True)).alias("session_n_songs"),
        count(when(df.page == 'Add Friend', True)).alias("session_n_friends"),
        count(when(df.page == 'Add to Playlist', True)).alias("session_n_add_playlist")
        count(when(df.page == 'Home', True)).alias("session_n_home"),
        count(when(df.page == 'Roll Advert', True)).alias("session_n_ads"),
        count(when(df.page == 'Help', True)).alias("session_n_help"),
        count(when(df.page == 'Error', True)).alias("session_n_error"),
        count(when(df.page == 'Settings', True)).alias("session_n_sets"),
        count(col('page')).alias('session_n_actions'),
        first(col('session_duration')).alias('session_duration')
    )
```

Interval until next session (Adding the interval between sessions as an feature)

w_user_sessions_interval = Window.partitionBy('userId').orderBy('min_event_ts')
df_sessions = df_sessions.withColumn('interval_to_session', col('min_event_ts') - lag(col('max_event_ts')).over(w_user_sessions_interval)

Adding the average playback time as a feature to check how users have played songs per session

df_session_time = df_sessions.groupBy('userId').agg(
(avg(df_sessions.session_n_total_playback) / minutes_to_hours).alias('avg_playback_time'))

df_sessions = df_sessions.join(df_session_time, on = 'userId')

User Profile

Creating user dimension features for subscription, streaming, community and page visit counts based on user interaction

CHURN_CANCELLATION_PAGE = 'Cancellation Confirmation'

REGISTRATION_PAGE = 'Submit Registration'

Subscription

a.) No of downgrades

b.) No of upgrades

c.) Paid User

d.) User Canceled

Streaming

a.) Next Song

b.) Likes on each song

c.) Dislikes on each song

Community

a.) Add Friend

b.) Add to playlist

A user profile can contain personal data. Most user profiles have a set of parameters which are either mandatory or optional. In some cases, the user profile could have different sections and tabs

```
df_user_profile = df.groupby('userId')\
        .agg(first(when(col('gender') == 'M', TRUE).otherwise(FALSE)).alias('male'),

            smin(col('first_ts')).alias('ts_start'),
            smax(col('last_event_ts')).alias('ts_end'),

            ((smax(col('last_event_ts')) - smin(col('first_ts'))) / milliseconds_to_hours).alias('time_window'),

            # Subscription
            count(when(col('page') == 'Submit Downgrade', True)).alias('n_downgrades'),
            count(when(col('page') == 'Submit Upgrade', True)).alias('n_upgrades'),
            last(when(col('level') == 'paid', TRUE).otherwise(FALSE)).alias('paid'),
            first(when(col('last_page') == CHURN_CANCELLATION_PAGE, TRUE).otherwise(FALSE)).alias('canceled'),

            # Streaming
            count(when(col('page') == 'NextSong', True)).alias('n_songs'),
            count(when(col('page') == 'Thumbs Up', True)).alias('n_likes'),
            count(when(col('page') == 'Thumbs Down', True)).alias('n_dislikes'),

            (count(when(col('page') == 'NextSong', True))/count(when(col('page') == 'Roll Advert', True))).alias('n_ads
            (count(when(col('page') == 'NextSong', True))/count(when(col('page') == 'Thumbs Up', True))).alias('n_likes
            (count(when(col('page') == 'NextSong', True))/count(when(col('page') == 'Thumbs Down', True))).alias('n_dis
            (count(when(col('page') == 'Thumbs Up', True))/count(when(col('page') == 'Thumbs Down', True))).alias('n_li

            countDistinct(col('sessionId')).alias('n_sess'),
            (avg(col('session_duration')) / milliseconds_to_hours).alias('avg_session_duration'),

            # Community
            count(when(col('page') == 'Add Friend', True)).alias('n_friends'),
            count(when(col('page') == ' ', True)).alias('n_added_to_playlist'),

            # Other
            count(when(col('page') == 'Home', True)).alias('n_home'),
            count(when(col('page') == 'Roll Advert', True)).alias('n_ads'),
            count(when(col('page') == 'Help', True)).alias('n_help'),
            count(when(col('page') == 'Error', True)).alias('n_errors'),
            count(when(col('page') == 'Settings', True)).alias('n_settings'),
            count(col('page')).alias('n_actions')
        )
```

User Dimension (Daily Data) - Adding daily

```
df_unique_days = df.groupby('userId').agg(countDistinct('date').alias('n_days'))
df_daily_actions = df.groupby('userId', 'date').agg(count('page').alias('total'))
df_daily_actions = df_daily_actions.groupby('userId').agg(avg('total').alias('avg_daily_actions'))
df_days = df_unique_days.join(df_daily_actions, df_unique_days.userId == df_daily_actions.userId)
df_days = df_days.drop(df_daily_actions.userId)
```

```
df_users = df_user_profile.orderBy(df_user_profile.userId).join(df_days, on = 'userId')
```

The final features

- **male:** 1 — male or 0 — female
- **paid:** 1- paid subscription or 0 — free subscription
- **avg_daily_actions:** Average actions by day
- **avg_session_duration:** Average duration in hours of each session
- **avg_playback_time:** Average time listening to music for each session

- **n_actions:** The number of actions
- **n_added_to_playlist:** Number of songs added to the playlist
- **n_ads:** Number of ads viewed
- **n_days:** Number of days of the observed window,
- **n_dislikes:** Number of dislikes
- **n_downgrades:** Number of downgrades
- **n_errors:** Number of experienced errors
- **n_friends:** Number of friends added
- **n_help:** Number of times has accessed the help page
- **n_home:** Number of times has accessed the home page
- **n_likes:** Number of songs liked
- **n_sess:** Number of sessions
- **n_settings:** Number of times has accessed the help page
- **n_songs:** Number of songs played
- **n_upgrades:** Number of upgrades
- **n_ads_over_songs:** Number of ads viewed divided by Number of songs played
- **n_likes_over_songs:** Number of songs liked divided by the number of Songs
- **n_dislikes_over_songs:** Number of songs disliked divided by the number of Songs
- **n_likes_over_dislikes:** Number of songs liked divided by the number of songs disliked
- **time_window:** Time time in hours of observed data

Data Modeling

Decision Tree Classifier

Decision trees can be used in decision analysis as one of the visual and explicit representations of decision and decision making procedure. The decision tree employs a tree-like model of decisions, with each internal node denoting a test, each branch representing an outcome of the test, and each leaf node holding a corresponding class label. One of the most powerful and widely used tools for classification and prediction is decision trees. Decision trees are widely used in medical diagnosis, failure prediction, credit scoring, and crime risk analysis. There are numerous business analysis examples available, but one of the most common is the detection of fraudulent financial statements.

Gradient Boost and Random Forest

Boosting is a method of machine learning that aims to improve the performance of a weak classifier by combining it with other weak classifiers. A weak classifier is defined as one whose performance is slightly better than random chance. One type of boosting algorithm is called Gradient Boosting Trees, which is an improved version of a technique called AdaBoost. AdaBoost was the first boosting algorithm and was proposed in 1996.

AdaBoost works by constructing a "strong" classifier as a combination of multiple "simple" weak classifiers. It does this by weighting the observations used for training, giving more weight to those that are difficult to classify and less weight to those that are already classified well. It then adds new weak classifiers sequentially, focusing on the more difficult patterns, and makes predictions by taking the majority vote of the weak classifiers' predictions, each of which is weighted by its individual accuracy.

In other words, AdaBoost is a method that takes a set of weak classifiers and combines them in a way that creates a stronger overall classifier. It does this by focusing on the most difficult patterns and adjusting the weights of the observations accordingly, so that the weak classifiers can better handle those patterns. The final prediction is made by considering the majority vote of the weak classifiers, each of which has its own weight based on its accuracy.

Gradient boosting involves three elements:

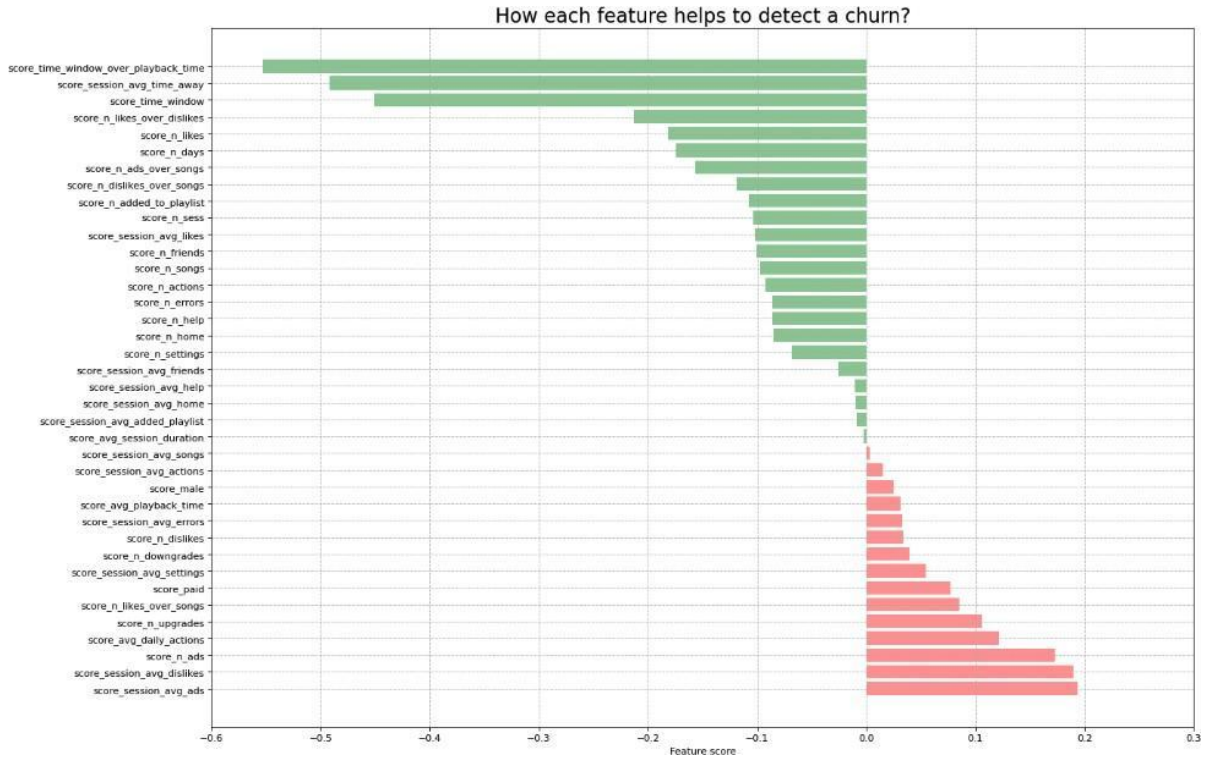
- A loss function that must be optimized. The type of loss function used is determined by the problem being solved. It must be differentiable, but there are many standard loss functions available, as well as the ability to define your own. For example, regression may employ squared error, whereas classification may employ logarithmic loss.
- Making predictions is difficult for a slow learner. In gradient boosting, decision trees serve as the weak learner.
- To minimize the loss function, an additive model is used to add weak learners. Existing trees in the model are not changed, and new trees are added one at a time. When adding trees, a gradient descent procedure is used to minimize loss.

GBT constructs trees one at a time, with each new tree assisting in the correction of errors made by previously trained trees. Anomaly detection in supervised learning settings where data is often highly unbalanced, such as DNA sequences, credit card transactions, or cybersecurity, is a great application of GBT.

The distinctions between GBT and RF can be explained as follows: Gradient boosting employs regression trees for prediction, whereas random forest employs decision trees. The random forest is simple to parallelize, but boosted trees are more difficult. Random forests overfit a sample of the training data before simply averaging the predictors to reduce the overfit. Gradient boosting can outperform random forests if the parameters are carefully tuned. However, if there is a lot of noise, gradient boosting may not be the best option because it can lead to overfitting. They are also more difficult to tune than random forests.

Logistic Regression

We want to classify our observations as "will churn" or "won't churn" from the platform when working with our data that accumulates to a binary separation. A logistic regression model will attempt to predict the likelihood of belonging to one of two groups. The logistic regression is essentially an extension of the linear regression, with the only difference being that the predicted outcome value is between [0, 1]. The model will identify relationships between our target feature, Churn, and the remaining features in order to perform probabilistic calculations to determine which class the customer should be assigned to. In Python, we'll be using the 'ScikitLearn' package.



Results

Machine learning modeling has been successful in predicting customer churn. This can assist application owners in increasing user lifetime. Despite the relatively good results of other selected machine learning models, such as SVM and Decision Trees, the GBT algorithm is chosen and hyper parameterized. To overcome overfitting in future work, the Random Forest algorithm can be tweaked with different settings.

Decision Tree Classifier

```
In [87]: pipeline1 = create_decision_tree_pipeline()
         model_pipeline1 = pipeline1.fit(train_df)
         predictions1 = model_pipeline1.transform(test_df)
         show_results_and_save(model_pipeline1, predictions1)

         [[ 841  497]
          [ 139 4411]]

         accuracy..... 0.8920
         precision..... 0.8582
         recall..... 0.6286
         auc..... 0.7990
         beta..... 0.6641
         F1 macro..... 0.8292
         F1 micro..... 0.8920
         F1 weighted..... 0.8857
         F1 binary..... 0.7256
```

Gradient Boost

```
pipeline3 = create_gradient_boost_pipeline()
model_pipeline3 = pipeline3.fit(train_df)
predictions3 = model_pipeline.transform(test_df)
show_results_and_save(model_pipeline3, predictions3)

Number of models to train: 150
[[ 876  401]
 [ 193 4447]]

accuracy..... 0.8996
precision..... 0.8195
recall..... 0.6860
auc..... 0.8222
beta..... 0.7091
F1 macro..... 0.8421
F1 micro..... 0.8996
F1 weighted..... 0.8963
F1 binary..... 0.7468
```

Random Forest

```
In [89]: pipeline2 = create_random_forest_pipeline()
         model_pipeline2 = pipeline2.fit(train_df)
         predictions2 = model_pipeline2.transform(test_df)
         show_results_and_save(model_pipeline2, predictions2)

         [[ 804  534]
          [ 100 4450]]

         accuracy..... 0.8923
         precision..... 0.8894
         recall..... 0.6009
         auc..... 0.7895
         beta..... 0.6426
         F1 macro..... 0.8254
         F1 micro..... 0.8923
         F1 weighted..... 0.8844
         F1 binary..... 0.7172
```

Logistic Regression

```
In [86]: pipeline = create_logistic_regression_pipeline()
         model_pipeline = pipeline.fit(train_df)
         predictions = model_pipeline.transform(test_df)
         show_results_and_save(model_pipeline, predictions)

         [[1085  253]
          [ 609 3941]]

         accuracy..... 0.8536
         precision..... 0.6405
         recall..... 0.8109
         auc..... 0.8385
         beta..... 0.7699
         F1 macro..... 0.8086
         F1 micro..... 0.8536
         F1 weighted..... 0.8592
         F1 binary..... 0.7157
```

Possible Improvements for Current Model

Even though the current model has an accuracy of nearly 89%, there is still room for improvement.

- Increase the dataset size to work with more observations.
- Investigating various parameter settings to improve accuracy.
- Analyzing location information has an impact on churned users because location can provide us with information about lifestyle and allow company owners to provide location-based promotions to retain users.
- Used environments can also provide valuable data. If Android users are more likely to abandon the application, this indicates that it needs to be improved for that environment.

Conclusion

We were able to study the service dataset and create functions for the modeling process through this project. To begin, we examined various levels of the dataset, which were the logs of each user session. The dataset enabled us to investigate churn and develop appropriate predictive features. Furthermore, feature selection was not an easy task. Logistic Regression, Random Forest, Gradient Boosted Trees, and Decision Tree Classifier were all trained.

In real-world applications, feasibility and cost become critical. Depending on the data latency and business requirements, this model can be run weekly or monthly. The operational costs should be tracked, and the model results should be validated through testing (A/B testing). Experiment outcomes (evaluation metrics, KPIs) should be tracked so that our model and subsequent actions can add value to the business.

Acknowledgements

We are grateful to Professor Juan Rodriguez and the teaching assistants for providing the information on big data analysis, for responding to our inquiries, and for sparking our intense interest in this field. They gave us the knowledge and abilities we needed to complete a project of this size and complexity. We also want to express our gratitude to the teaching assistants for helping us improve our conceptual and technical comprehension of the various big data topics.

References

1) https://www.analyticsvidhya.com/blog/2022/06/customer-churn-prediction-using-mlib/
2) https://pub.towardsai.net/this-is-how-you-can-build-a-churn-prediction-model-using-spark-e187b7eca339
3) https://www.kaggle.com/code/mnassrib/customer-churn-prediction-with-pyspark
4) https://leopoldwalther.medium.com/predicting-user-churn-for-a-streaming-service-using-spark-6ef7379c7963
5) https://medium.com/streamthoughts/streaming-data-into-kafka-s01-e04-parsing-log-files-using-grok-expressions-c282d54b3e3f
6) https://medium.com/@Ankitthakur/apache-kafka-installation-on-mac-using-homebrew-a367cdefd273
7) https://udacity-dsnd.s3.amazonaws.com/sparkify/sparkify_event_data.json