Data Analytics 2020

Final Project: *Spotify Song Attributes*
Submitted By: *Bruna Correa & Neha Sharma*
Submitted to: *Fillipa Peleja & CodeOp team*
Date: *28 August 2020*

Table of content:

1. Introduction

“Spotify Song Attributes”: An attempt to build a classifier that can predict whether or not a user likes a song. Here one question arises: what is Spotify?

Picture 1, Spotify logo, Source: [pic:1]

Spotify is a digital music, podcast, and video streaming service that gives you access to millions of songs and other content from artists all over the world. Launched in October 2008, the Spotify platform provides access to over 60 million songs. Users can browse by parameters such as artist, album, or genre, and can create, edit, and share playlists. Spotify is available in most of Europe, Americas, Australia, New Zealand, and parts of Africa and Asia, and on most modern devices, including Windows, macOS, and Linux computers, and iOS, and Android smartphones and tablets. Since February 2018, it has been listed on the New York Stock Exchange. As of July 2020, the company had 299 million monthly active users, including 138 million paying subscribers.

Picture 2, Global Recorded music industry revenues 1999-2017. Source:[pic:2]

Core to Spotify's strategy for winning in this crowded market is its ability to provide personalized recommendations and help users to discover new music, which is enabled by its investments in machine learning. In its IPO prospectus, the company highlighted this strategy stating that it will "continue to invest in our artificial intelligence and machine learning capabilities to deepen the personalized experience that we offer to all of our users" and that "this personalized experience is a key competitive advantage". Given Spotify's deep pool of data (200 petabytes compared to Netflix's 60 petabytes)2, the company is well-poised to create competitive advantage and provide users with a continually improving service.

As Data Analysts, we would like to explore our playlists to discover the features that make us like a given song or not. "Spotify Song Attributes" is an attempt to explore our tastes and preferences in music. With that we would like to develop a recommendation engine based on content.

In the following document we will explore how we collected our data from Spotify API and how we used essential steps of Data Analysis to gain insights about their information. Before that, we will discuss the problem context, define an objective for the task, and will wrap up this document with conclusion and future work.

2. Context of the problem

Spotify has developed algorithms to regulate everything from your personal best home screen to curated playlists like Discover Weekly, and is continuing to experiment with new ways of understanding music, and why people listen to one song or genre over another. Spotify's main differentiating factor from other companies is the level of customization and expansion of music knowledge offered to customers.

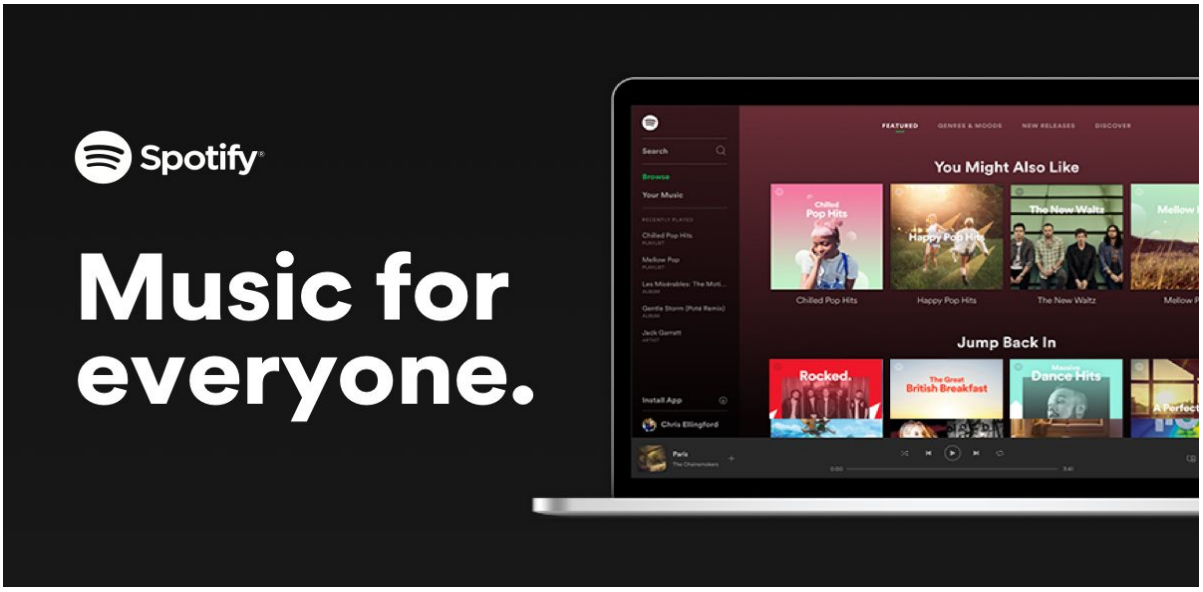
Spotify System works on two concepts mainly, Exploit and Explore. When Spotify exploits, it uses the user information for example user's music history, playlists user made etc. And when it explores it considers the information about the rest of the world, for example playlists and artists similar to artists' taste in music.

Our goal in this task is to focus on "exploit and explore" and create a solution by analysing the audio itself and train an algorithm to learn how to recognize different aspects of the music that might interest an user.

Spotify's strategy has consistently focused on machine learning. First, its machine-generated, personalized playlists such as 'Discover Weekly' and 'Release Radar' account for 31% of all listening on the platform compared to less than 20% two years ago. The company employs three types of machine learning to enhance its recommendation engine: collaborative filtering, natural language processing (NLP), and raw audio models1. Through collaborative filtering, Spotify provides recommendations to users based on the preferences of users with similar tastes. With NLP, the company scours articles, blogs, and song's metadata to generate "tags" associated with each song and compares those tags with those of other songs. The company also analyzes which artists or songs are frequently mentioned along with the song in question to refine the pool of song recommendations. Through raw audio processing, Spotify is able to identify commonalities between songs through their musical elements (e.g. tempo, time signature, key). While collaborative filtering and NLP allow Spotify to point users to popular songs they may enjoy, raw audio processing allows the company to make predictive suggestions for songs with very little user awareness.[1]

With the given strategy, our goal is to create a content based recommendation system so the engine will recognize users' choices from its playlists and return as results the most likely songs.

3. Objective



Picture 3, Music for everyone, source: [pic:3]

Our objective for this project is as following:

- Creating a classifier:
  - The main goal for this task is to use different machine learning algorithms to build a classifier for our given data and predict whether the user will like a song or not.

  This objective is inspired from the kaggle competition called "*Spotify Song Attributes*: An attempt to build a classifier that can predict whether or not I like a song".[2]

- Creating a content-based recommendation system.
  - In this task we will develop a recommender system based on the user's explicit data to create suggestions for that user.

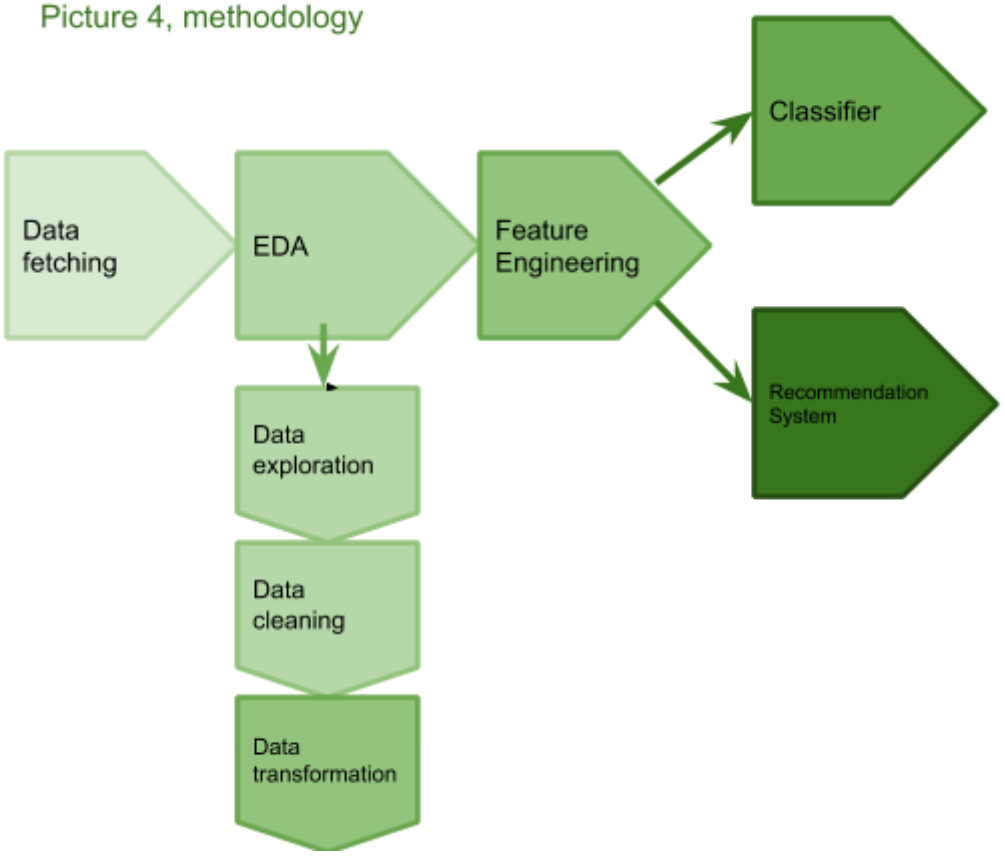
  Spotify, however, developed an engine that used three different recommendation models. Their strategy changed over time, but one of the algorithms that Spotify used to create the 'Discover Weekly' playlists was a mix of the best strategies used by their competitors. Spotify combined three different models to analyze the similarity of songs:[3]
  - Collaborative filtering examines and compares an individual's behavior to other people's taste.
  - Natural Language Processing (NLP) analyzes the text in each song.
  - Audio modeling uses a song's raw audio to understand the tune of the song and compares it to other songs.

We will be seeing these topics in detail as we move forward with the task.

4. Methodology

In this section we are going to see the framework of how the tasks had been performed. In the following picture we can see the steps we took to perform our objectives. We will see the detailed explanation in corresponding sections. Here we will see the overview.

Picture 4, methodology



- For starters we collected our data using the Spotify API with our unique client id and secret.
- After collecting data, we performed exploratory data analysis, in which we explored, cleaned and transformed our data.
- Next we did some feature engineering to benefit our model.
- At this point with our data ready, we performed our main tasks such as classification and recommender engine.
- At the end we wrapped up our project with conclusion and future work.

5. Data Scraping

To be able to perform our objectives for the task, first we need to collect data. Here we are going to use our own personal Spotify data. In this section we are going to see how we fetched our data from Spotify API.
Spotify allows users to access their data from their developers platform. Spotify provides an API service that allows you to access data from their archive of millions of songs. Through the Spotify Web API [4], external applications retrieve Spotify content such as album data and playlists. To access *user-related data* through the Web API, an application must be authorized by the user to access that particular information.

We are going to see the whole process in following steps:

5.1. Authorization

To have the end user approve your app for access to their Spotify data and features, or to have your app fetch data from Spotify, you need to authorize your application.

Your app can be authorized by Spotify in two ways:

- App Authorization: Spotify authorizes your app to access the Spotify Platform (APIs, SDKs and Widgets).
- User Authorization: Spotify, as well as the user, grant your app permission to access and/or modify the user's own data. For information about User Authentication, see User Authentication with OAuth 2.0 [5]. Calls to the Spotify Web API [6] require authorization by your application user. To get that authorization, your application generates a call to the Spotify Accounts Service /authorize endpoint, passing along a list of the scopes for which access permission is sought.

Making authorized requests to the Spotify platform requires that you are granted permission to access data. In accordance with RFC-6749, 3 parties are involved in the authorization process:

- Server: the Spotify server
- Client: your application
- Resource: the end user data and controls

Picture 5, Spotify Web API Authorization, [pic:5]

To be able to obtain authorization following steps were taken:

- Setup Developers account
- Registration for application
- Authorization flow

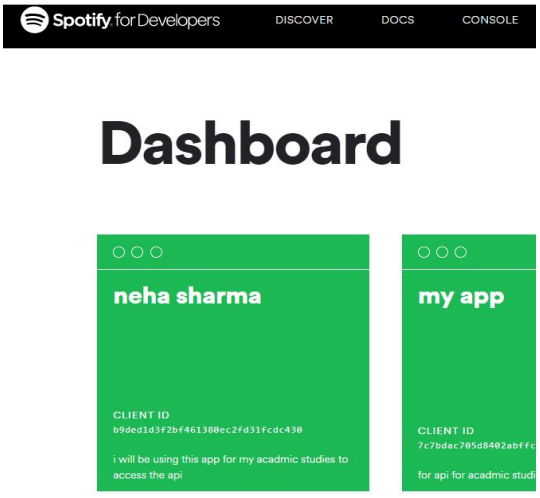
5.1.1. Setup Developers account

To use the Web API, start by creating a Spotify user account (Premium or Free). To do that, simply sign up at www.spotify.com [7]. When you have a user account, go to the Dashboard [8] page at the Spotify Developer website and, if necessary, log in. Accept the latest Developer Terms of Service to complete your account set up.

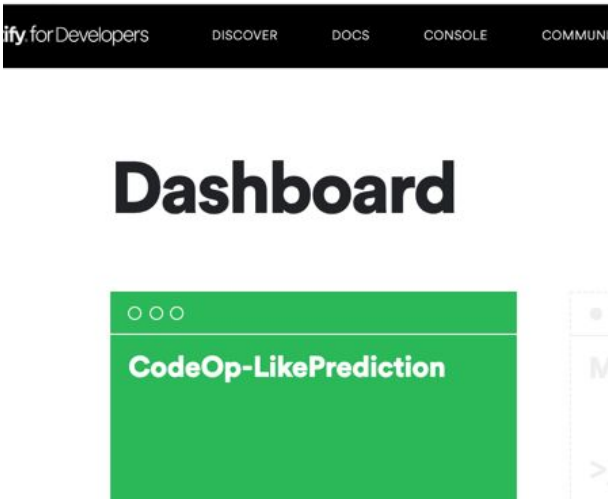
5.1.2. Registration for application

Any application can request data from Spotify Web API endpoints and many endpoints are open and will return data *without* requiring registration. However, if your application seeks access to a user's personal data (profile, playlists, etc.) it must be registered. Registered applications also get other benefits, like higher rate limits at some endpoints. You can register your application, even before you have created it. As you can see in the picture 6 below, user1(Neha) has created an application "neha sharma" on the dashboard and in picture 7, user2(Bruna) has created an application "CodeOp-LikePrediction".

As soon as you register an application it will provide you with client id and client secret. We will be using these two informations to fetch our data using "Spotipy" Python library.



Picture 6, Developers Dashboard Neha



Picture 7, Developers Dashboard Bruna

Before moving to Spotipy library, we will see how authorization flows work.

5.1.3. Authorization flows

There are four optional flows to obtain app authorization:

- Refreshable user authorization: Authorization Code Flow
- Refreshable user authorization: Authorization Code Flow With Proof Key for Code Exchange (PKCE)
- Temporary user authorization: Implicit Grant
- Refreshable app authorization: Client Credentials Flow

Flow	Access User Resources	Required Secret Key(Server Side)	Access Token Refresh
Authorization Code	Yes	Yes	Yes
Authorization Code with PKCE	Yes	No	Yes
Client Credentials	No	Yes	No
Implicit Grant	Yes	No	No

Table 1, Authorization Flow Spotify, Source: [Table:1]

We are going to use Client Credentials Flow for our project.

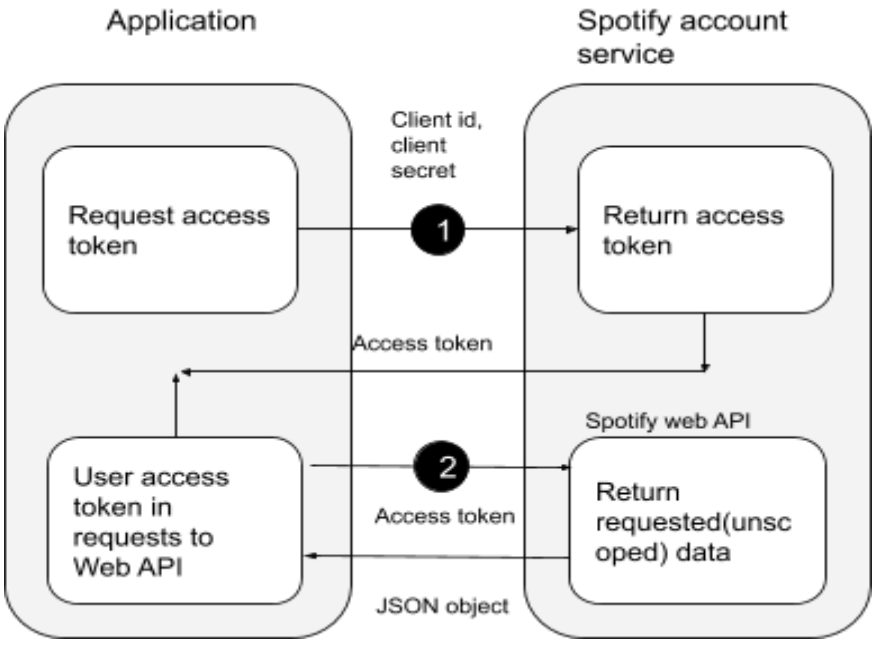
>>> Client Credentials Flow:

The Client Credentials flow is used in server-to-server authentication. Only endpoints that do not access user information can be accessed. The advantage here in comparison with requests to the Web API made without an access token, is that a higher rate limit is applied.

You do: login with your client id and secret key

You get: access token

In the below picture you can see the workflow for client credential flow.



Picture 8, Client Credentials flow, Source: [pic:8]

5.2. Implementation

Up to now we have seen a theoretical explanation. Now we have our client id and client secret. All manual work had been done. It's time to jump into our python environment and do some real programming. We are using “Jupyter Notebook” for our python programming. First things first, we need to import the required library that is Spotipy as shown in the below picture.

```
import spotipy
import spotipy.util as util
import spotipy.oauth2 as oauth2
```

Cell 1, import spotipy, source: [code:1]

Here,

- *Spotipy* [9] is a lightweight Python library for the Spotify Web API. With *Spotipy* you get full access to all of the music data provided by the Spotify platform.
- *Spotipy.util* shows a user's playlists (need to be authenticated via oauth).
- *Spotipy.oauth2* is a class SpotifyOAuth provided by *Spotipy* for Client Authorization Code Flow that can be used to authenticate requests. We will be using "oauth2.SpotifyClientCredentials" and ".get_access_token()" classes to access token from web API. You can see the implementation in picture below:

```
credentials = oauth2.SpotifyClientCredentials( client_id=CLIENT_ID, client_secret=CLIENT_SECRET)
token = credentials.get_access_token()
spotify = spotipy.Spotify(auth=token)
```

Cell 2, Access Token using Spotipy

- Here, client_id and client_secret were taken when we registered our application and saved in a text file called "credentials.txt". We will be calling this file to open in our jupyter cell, something like this:

```
f = open("C:\\Users\Dream\codeop\spotify\credentials","r")
CLIENT_ID = f.readline().replace('\n','')
CLIENT_SECRET = f.readline().replace('\n','')
```

Cell 3, access credentials from text file

- With the help of *Spotipy* we were able to get the access token for our web API. Now with all the requirements fulfilled we need to fetch data in readable form. But before that we need to collect playlist ids from our spotify account.
  In the below table can see the details for playlists:

Playlist names	User1 (Neha)	User2 (Bruna)
liked_songs	0zx34QBjFUJfy5V9Nxnaxy	2Ef3fGdmibLpBC5lYqhTia
disliked_songs	1PRbKkKbUN7O2BJjRL7QRv	74gPVIUFWIYQYA7d7tUM7p'
validation_liked_songs	5z8Feij4DTtjwoAJliL01O	6yeUUCIxul24kft5RbJpRP
validation_disliked_songs	4stUENVtDPmh9tZ54U6eUS	6EL2P68pNWfg9cHdreb5gC

Table 2, playlist from spotify accounts of user1 and user2

- Now as we have our playlists ready, we just need a function to take out all the information we required from them. So for that we created a "get_playlist_songs"

function. Which takes a playlist name and id as attributes and returns a dataframe with audio features and ids for the songs in the playlist given. You can see the function in picture below:

```
# Function to get song features from a playlist
def get_playlist_songs(username, playlist_id):
    # Get songs from playlist
    playlist = spotify.user_playlist_tracks(username, playlist_id)
    songs = playlist['items']
    while playlist['next']:
        playlist = spotify.next(playlist) # use sp.next method to overcome fetch limit of 100 songs
        songs.extend(playlist['items'])
    # Create a list with song ids
    song_ids = []
    for i in range(len(songs)):
        song_ids.append(songs[i]["track"]["id"])
    # Create a dataframe with features for each song
    songs_df = pd.DataFrame()
    for song_id in song_ids:
        features = spotify.audio_features(song_id)
        songs_df = songs_df.append(pd.DataFrame(features), ignore_index=True)
    return songs_df
```

Cell 4, function to return playlist songs features

- Here, we are using “spotify.user_playlist_tracks” method in module spotipy.client to collect track ids from our playlists. As this method has a limit of 100 songs and we have more songs then this limit, we are going to a loop using the “next” method to get all the tracks from our playlists.
- Now as we have all the tracks and ids with us we are going to use “spotipy.audio_features” method to collect audio features of our tracks. At the end we appended all these features into a dataframe. With the above process we were successful to create a function which returns a dataframe with all the features of playlist tracks.

In the next section we are going to see the exploratory analysis of our data.

6. Exploratory Data Analysis

In statistics, exploratory data analysis is an approach to analyze data sets and summarize their main characteristics, often with visual methods also. A statistical model can be used or not, but primarily EDA is for seeing what the data can tell us beyond the formal modeling or hypothesis testing task. In the following section we are going to analyse our data and will see some visualizations to gain insight about song features. We will be doing our EDA in following steps:

- Data Exploration
- Data Visualization
- Data Cleaning and transformation

6.1. Data Preparation and Exploration

In this part, we will explore the shape and size of our dataset, we will be seeing what kind of data types we have and some description about statistics of data. As from the data scraping section we were able to collect data from Spotify API, now it's time to explore it.

We have two users' data with us, User 1 (Neha) and User 2 (Bruna). We will be exploring both users' data. Both users collected data from 4 playlists which are mentioned in "Table 2, playlist from spotify accounts of user1 and user2" already. Playlists are as follows: liked_songs, disliked_songs, validation_liked_songs, validation_disliked_songs.

- User1:

Playlists	Shape	Total features	Audio features	String features	Numeric features
Liked_songs	(599,18)	18	13/18	5/18	13/18
Disliked_songs	(400,18)	18	13/18	5/18	13/18
Validation_liked_songs	(30,18)	18	13/18	5/18	13/18
Validation_disliked_songs	(30,18)	18	13/18	5/18	13/18

Table 3, user1 data shape and size

So user1 has 1000 songs to prepare models and 60 songs to validate the models. With a total 18 features.

❏ User2 :

Playlists	Shape	Total features	Audio features	String features	Numeric features
Liked_songs	(1000,18)	18	13/18	5/18	13/18
Disliked_songs	(250,18)	18	13/18	5/18	13/18
Validation_liked_songs	(50,18)	18	13/18	5/18	13/18
Validation_disliked_songs	(50,18)	18	13/18	5/18	13/18

Table 4, user1 data shape and size

So user2 has 1,250 songs to prepare models and 100 songs to validate the models. With a total 18 features.

We need 2 datasets for each user and a target column with binary values where 1 represents like and 0 represents dislike. Before concatenating datasets together we will add new target features in each dataset. In order to do that we will add a new column called “target” in liked_songs and disliked_songs with value 1 for liked songs and value 0 for disliked songs. And similarly a “discover” name column for validation sets where value 1 for liked songs and 0 value will be for disliked songs. Then we will merge liked_songs and disliked_songs datasets together and validation_liked_songs and validation_disliked_songs together for both users. You can see the code below:

```
disliked_songs['target'] = 0
liked_songs['target'] = 1
validation_liked_songs['Discover'] = 1
validation_disliked_songs['Discover'] = 0
```

Cell 5, new feature addition

```
df = pd.concat([liked_songs,disliked_songs], axis = 0).reset_index(drop=True)
val = pd.concat([validation_set,random_set], axis = 0).reset_index(drop=True)
```

Cell 6, data concatenation

Now we have our data prepared in a likely manner. Let's have a look what it contains:

❏ User 1:

Dataframes	Shape	Total features	Audio features	String features	Numeric features
df (training_set)	(1000,19)	19	13/19	5/19	14/19
val(validation_set)	(60,19)	19	13/19	5/19	14/19

Table 5, prepared datasets user 1

❏ User 2:

Dataframes	Shape	Total features	Audio features	String features	Numeric features
df (training_set)	(1250,19)	19	13/19	5/19	14/19
val(validation_set)	(100,19)	19	13/19	5/19	14/19

Table 6, prepared datasets user 2

Let's explore the dataframes. We will be using dataframe.info(), dataframe.head(), dataframe.describe() function to gain insight.
Here, dataframe.info() will give the size, datatype, contained_values of all features. dataframe.head() will show the first 5 entries of datasets. And dataframe.describe() will show the statistics values of numeric features for example mean, median, 25% quantile etc.

❏ User 1:

```
print(df.shape)
df.head()
```

```
(999, 19)
```

	danceability	energy	key	loudness	mode	speechiness	acousticness	instrumentalness	liveness	valence	tempo	type	
0	0.358	0.920	6	-2.843	1	0.2180	0.00013	0.011800	0.0765	0.493	99.919	audio_features	3HtCaD8SkNbjy3TrEc
1	0.599	0.845	10	-4.100	0	0.0446	0.30200	0.000000	0.0734	0.485	150.032	audio_features	1EBDy7MWZ3GRyFBduF
2	0.505	0.710	3	-3.015	1	0.0321	0.19000	0.000250	0.3290	0.428	89.938	audio_features	5VC29kHMkzcaorzPH
3	0.635	0.926	0	-5.589	1	0.1510	0.08930	0.000005	0.0928	0.761	100.048	audio_features	05p7wWoGLZIhU0xqV
4	0.672	0.655	10	-5.021	0	0.0311	0.03620	0.000000	0.1170	0.556	134.945	audio_features	6Qn5zhYkTa37e91HC

Cell 7, user 1 data's first five entries

❏ User 2:

```
print(songs_df.shape)
songs_df.head()
```

```
(1250, 19)
```

	danceability	energy	key	loudness	mode	speechiness	acousticness	instrumentalness	liveness	valence	tempo	type	
0	0.601	0.757	2	-3.341	1	0.0280	0.0139	0.000088	0.0727	0.462	82.001	audio_features	3SWqGa1J0M7hSBUDM
1	0.557	0.747	4	-4.329	0	0.0261	0.0617	0.002480	0.1220	0.672	91.001	audio_features	65wl0TYoyrBFJtot
2	0.920	0.914	7	-4.555	1	0.0444	0.0222	0.000076	0.1140	0.974	135.923	audio_features	73mlvsfJM2qwlDU
3	0.582	0.962	4	-3.037	1	0.0542	0.0096	0.128000	0.2830	0.506	123.971	audio_features	7aOor99o8NNLZYEIC
4	0.713	0.565	7	-7.800	0	0.0351	0.0252	0.000003	0.6330	0.705	130.047	audio_features	3DCU0R5FFaB9GKxZE

Cell 8, user 2 data first 5 entries

Cell 7 and 8 represent the shape of dataframes and the first five rows in the datasets for different users.

```
df.info()
```

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 999 entries, 0 to 998
Data columns (total 19 columns):
 #   Column            Non-Null Count  Dtype
---  ------            --------------  -----
 0   danceability      999 non-null    float64
 1   energy            999 non-null    float64
 2   key               999 non-null    int64
 3   loudness          999 non-null    float64
 4   mode              999 non-null    int64
 5   speechiness       999 non-null    float64
 6   acousticness      999 non-null    float64
 7   instrumentalness  999 non-null    float64
 8   liveness          999 non-null    float64
 9   valence           999 non-null    float64
 10  tempo             999 non-null    float64
 11  type              999 non-null    object
 12  id                999 non-null    object
 13  uri               999 non-null    object
 14  track_href        999 non-null    object
 15  analysis_url      999 non-null    object
 16  duration_ms       999 non-null    int64
 17  time_signature    999 non-null    int64
 18  target            999 non-null    int64
dtypes: float64(9), int64(5), object(5)
memory usage: 148.4+ KB
```

Here we can see that, data has:

- ❏ Number of rows: 0 to 999
- ❏ Number of columns: 19
- ❏ Total types of data: 3: float, int, object
- ❏ So basically we have numeric and string columns. Out of all the features 13 columns are audio features, 5 are string features with track id and href and one column with our target values.
- ❏ Similarly except for the number of rows everything is the same for user 2 also. Number of rows for user 2 is 1250.

Cell 9, user 1 data information

In the below picture you can see the user 1 data description.

	count	mean	std	min	25%	50%	75%	max
danceability	999.0	0.606435	0.176492	0.000000	0.4860	0.62000	0.744500	0.968
energy	999.0	0.633995	0.226952	0.026000	0.4665	0.67200	0.820000	0.991
key	999.0	5.150150	3.618794	0.000000	1.0000	5.00000	8.000000	11.000
loudness	999.0	-7.061579	3.499657	-29.889000	-8.6620	-6.31200	-4.681500	-0.223
mode	999.0	0.572573	0.494953	0.000000	0.0000	1.00000	1.000000	1.000
speechiness	999.0	0.098867	0.095723	0.000000	0.0376	0.05850	0.127500	0.875
acousticness	999.0	0.297204	0.314886	0.000005	0.0357	0.15200	0.520000	0.992
instrumentalness	999.0	0.042817	0.159865	0.000000	0.0000	0.00001	0.000824	0.961
liveness	999.0	0.216269	0.191428	0.020100	0.0992	0.13700	0.274000	0.994
valence	999.0	0.502039	0.243549	0.000000	0.2960	0.49500	0.706000	0.977
tempo	999.0	119.273883	30.861348	0.000000	94.3600	117.75000	140.067500	207.806
duration_ms	999.0	228962.161161	62874.234141	14914.000000	190947.0000	224493.00000	264213.500000	536053.000
time_signature	999.0	3.928929	0.390203	0.000000	4.0000	4.00000	4.000000	5.000
target	999.0	0.599600	0.490225	0.000000	0.0000	1.00000	1.000000	1.000

Cell 10, user 1 data description

	count	mean	std	min	25%	50%	75%	max
danceability	1250.0	0.601390	0.153422	0.137000	0.488000	0.612000	0.716000	0.975
energy	1250.0	0.570028	0.228428	0.010700	0.393000	0.581500	0.751000	0.990
key	1250.0	5.347200	3.573691	0.000000	2.000000	6.000000	9.000000	11.000
loudness	1250.0	-8.316358	3.954397	-32.770000	-10.357500	-7.692000	-5.538250	1.509
mode	1250.0	0.624800	0.484368	0.000000	0.000000	1.000000	1.000000	1.000
speechiness	1250.0	0.066688	0.066730	0.022700	0.032800	0.042300	0.068875	0.868
acousticness	1250.0	0.402143	0.321407	0.000036	0.090625	0.338000	0.707000	0.995
instrumentalness	1250.0	0.099536	0.233635	0.000000	0.000000	0.000149	0.018025	0.967
liveness	1250.0	0.211841	0.207438	0.021700	0.096750	0.119000	0.234750	0.990
valence	1250.0	0.489464	0.252136	0.033600	0.278250	0.474500	0.686750	0.983
tempo	1250.0	121.090996	29.216355	41.237000	97.998500	119.964500	139.440250	219.981
duration_ms	1250.0	223134.695200	62103.047472	18514.000000	187043.000000	215000.000000	247526.750000	578041.000
time_signature	1250.0	3.910400	0.400125	0.000000	4.000000	4.000000	4.000000	5.000
like	1250.0	0.800000	0.400160	0.000000	1.000000	1.000000	1.000000	1.000

Cell 11, user 2 data description

These are the following conclusions drawn from above description:

- There are no missing/null values for both users.
- Categorization of features:
    - Class Labels: target
    - Interval: acousticness, danceability, energy, instrumentalness, liveness, loudness, speechiness,valence
    - Time: duration_ms
    - Numerical: tempo
    - Ordinal: key, time_signature
    - Binary: mode
    - String: id, 'uri, track_href, analysis_uri, type.

We will discuss each feature in detail in the next section.

- Users average liked song and disliked song:

To find out what is a user's average liked or disliked song, we need to consider two things:

- Average +/- one standard deviation: for continuous features
- Mode: for categorical features

For a liked song: we created a dataframe with target value = 1, and for each continuous feature, we are calculating the mean and standard deviation. For categorical features, the mode of each feature. Then we filter only the songs that have values between the average -1 and +1 standard deviation for continuous features; and equals the mode values for the categorical features.

danceability	0.612
energy	0.748
key	0
loudness	-5.014
mode	1
speechiness	0.0445
acousticness	0.617
instrumentalness	0
liveness	0.167
valence	0.574
tempo	120.052
duration_ms	223267
time_signature	4

Cell 12, user 1 average liked song

danceability	0.483
energy	0.721
key	0
loudness	-6.836
mode	1
speechiness	0.032
acousticness	0.729
instrumentalness	0
liveness	0.189
valence	0.562
tempo	95.261
duration_ms	369600
time_signature	4

Cell 13, user 2 average liked song

You can see the code in the cell below:

```
# Get an 'average song' that represents what I like
like_avg = df[df['target'] == 1].describe().loc[['mean','std']]

average_song = df[
    (df['danceability'] < like_avg.danceability[0]+like_avg.danceability[1]) & (df['danceability'] > like_avg.danceability[0]-lik
    (df['energy'] < like_avg.energy[0]+like_avg.energy[1]) & (df['energy'] > like_avg.energy[0]-like_avg.energy[1]) &
    (df['loudness'] < like_avg.loudness[0]+like_avg.loudness[1]) & (df['loudness'] > like_avg.loudness[0]-like_avg.loudness[1]) &
    (df['speechiness'] < like_avg.speechiness[0]+like_avg.speechiness[1]) & (df['speechiness'] > like_avg.speechiness[0]-like_avg
    (df['acousticness'] < like_avg.acousticness[0]+like_avg.acousticness[1]) & (df['acousticness'] > like_avg.acousticness[0]-lik
    (df['instrumentalness'] < like_avg.instrumentalness[0]+like_avg.instrumentalness[1]) & (df['instrumentalness'] > like_avg.ins
    (df['liveness'] < like_avg.liveness[0]+like_avg.liveness[1]) & (df['liveness'] > like_avg.liveness[0]-like_avg.liveness[1]) &
    (df['valence'] < like_avg.valence[0]+like_avg.valence[1]) & (df['valence'] > like_avg.valence[0]-like_avg.valence[1]) &
    (df['tempo'] < like_avg.tempo[0]+like_avg.tempo[1]) & (df['tempo'] > like_avg.tempo[0]-like_avg.tempo[1]) &
    (df['mode'] == 1) &
    (df['key'] == 0) &
    (df['time_signature'] == 4)
]

# Pick one average LIKE song
average_song.iloc[1].to_frame()
```

Cell 14, code for returning average liked song for each user

For a disliked song: set target value to 0 and for each continuous feature, we are calculating the mean and standard deviation. For categorical features, the mode of each feature. Then we filter only the songs that have values between the average -1 and +1 standard deviation for continuous features; and equals the mode values for the categorical features.

danceability	0.716
energy	0.676
key	7
loudness	-4.762
mode	1
speechiness	0.0618
acousticness	0.000301
instrumentalness	0.00234
liveness	0.289
valence	0.402
tempo	98.01
duration_ms	179278
time_signature	4

Cell 15, user 1 avg. disliked song

danceability	0.691
energy	0.628
key	7
loudness	-6.868
mode	1
speechiness	0.0612
acousticness	0.304
instrumentalness	0
liveness	0.0511
valence	0.938
tempo	161.668
duration_ms	194508
time_signature	4

Cell 16, user 2 avg. disliked song

You can see the code implementation in below cell:

```
# Get an 'average song' that represents what I don't like
nlike_avg = df[df['target'] == 0].describe().loc[['mean','std']]

average_song = df[
    (df['danceability'] < nlike_avg.danceability[0]+nlike_avg.danceability[1]) & (df['danceability'] > nlike_avg.danceability[0]-
    (df['energy'] < nlike_avg.energy[0]+nlike_avg.energy[1]) & (df['energy'] > nlike_avg.energy[0]-nlike_avg.energy[1]) &
    (df['loudness'] < nlike_avg.loudness[0]+nlike_avg.loudness[1]) & (df['loudness'] > nlike_avg.loudness[0]-nlike_avg.loudness[1
    (df['speechiness'] < nlike_avg.speechiness[0]+nlike_avg.speechiness[1]) & (df['speechiness'] > nlike_avg.speechiness[0]-nlike
    (df['acousticness'] < nlike_avg.acousticness[0]+nlike_avg.acousticness[1]) & (df['acousticness'] > nlike_avg.acousticness[0]-
    (df['instrumentalness'] < nlike_avg.instrumentalness[0]+nlike_avg.instrumentalness[1]) & (df['instrumentalness'] > nlike_avg.
    (df['liveness'] < nlike_avg.liveness[0]+nlike_avg.liveness[1]) & (df['liveness'] > nlike_avg.liveness[0]-nlike_avg.liveness[1
    (df['valence'] < nlike_avg.valence[0]+nlike_avg.valence[1]) & (df['valence'] > nlike_avg.valence[0]-nlike_avg.valence[1]) &
    (df['tempo'] < nlike_avg.tempo[0]+nlike_avg.tempo[1]) & (df['tempo'] > nlike_avg.tempo[0]-nlike_avg.tempo[1]) &
    (df['mode'] == 1) &
    (df['key'] == 7) &
    (df['time_signature'] == 4)
]

# Pick one average NOT_LIKE song
average_song.iloc[-1].to_frame()
```

Cell 17, code for returning average disliked song for each user

In the following section we are going to use matplotlib and seaborn library to visualize the data features.

6.2. Data Visualization

Data visualization is the graphic representation of data. It involves producing images that communicate relationships among the represented data to viewers. This communication is achieved through the use of a systematic mapping between graphic marks and data values in the creation of the visualization. This mapping establishes how data values will be represented visually, determining how and to what extent a property of a graphic mark, such

as size or color, will change to reflect changes in the value of a datum. We will use “matplotlib” and ‘seaborn” libraries to visualize our data.

Matplotlib [10] is a comprehensive library for creating static, animated, and interactive visualizations in Python. It helps users to:

- Create: Develop publication quality plots with just a few lines of code. Use interactive figures that can zoom, pan, update.
- Customize: Take full control of line styles, font properties, axes properties. Export and embed to a number of file formats and interactive environments
- Extend: Explore tailored functionality provided by third party packages. Learn more about Matplotlib through the many external learning resources

Seaborn [11] Seaborn is a library for making statistical graphics in Python. It is built on top of matplotlib and closely integrated with Pandas [12] data structures.

Here is some of the functionality that seaborn offers:

- A dataset-oriented API for examining relationships between multiple variables
- Specialized support for using categorical variables to show observations or aggregate statistics
- Options for visualizing univariate or bivariate distributions and for comparing them between subsets of data
- Automatic estimation and plotting of linear regression models for different kinds of dependent variables
- Convenient views onto the overall structure of complex datasets
- High-level abstractions for structuring multi-plot grids that let you easily build complex visualizations
- Concise control over matplotlib figure styling with several built-in themes
- Tools for choosing color palettes that faithfully reveal patterns in your data

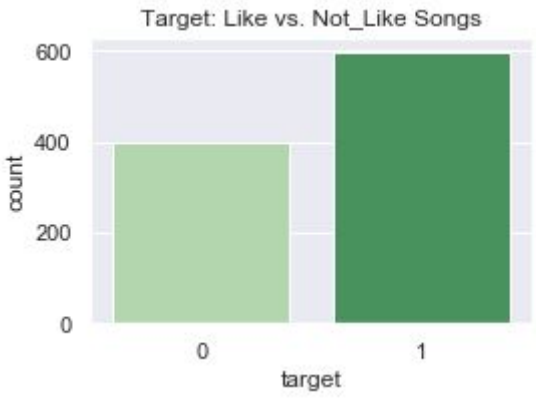
Seaborn aims to make visualization a central part of exploring and understanding data. Its dataset-oriented plotting functions operate on dataframes and arrays containing whole datasets and internally perform the necessary semantic mapping and statistical aggregation to produce informative plots.

It was a theoretical explanation for this section now let's dig in to some programming. With the visualization we will understand each and every feature in our datasets and analyse what importance they hold for our model.

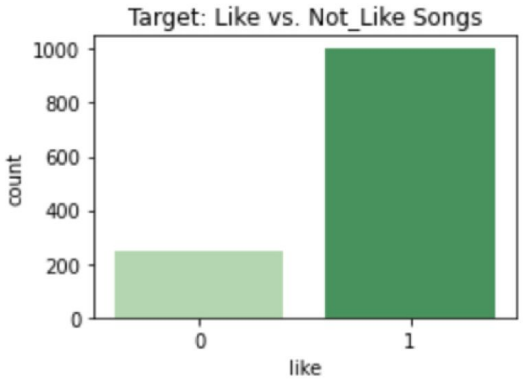
We will go feature by feature here. Which features will we explore? We will be exploring audio features of songs which are 13 in count with 10 continues and 3 categorical types and our target column too. Reference for feature description [13]

>>>features**:**

- “target” [like/dislike]: it explains with the value 1, user like that song and with value 0 user dislike that song.



Graph 1, user1, target



Graph 2, user2, target

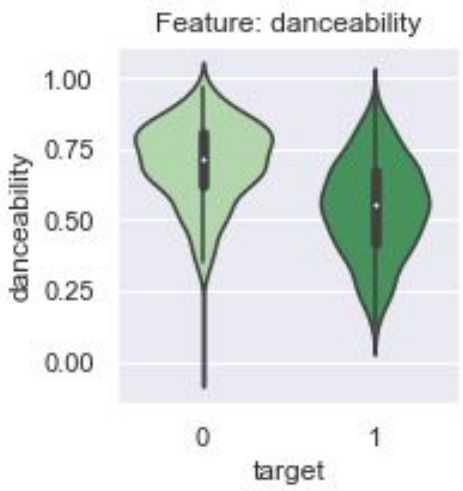
In graph 1 we can see that user1 has around 400 songs which she does not like and 600 songs she likes. Similarly in graph 2 user2 has liked 1,000 songs while disliked 250 songs.

note*: graph 3 to graph 22 are violin plots from seaborn.

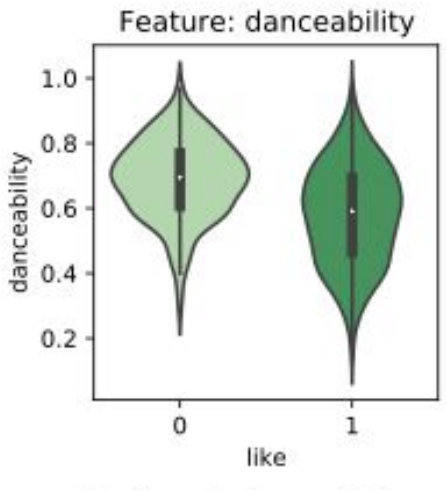
seaborn.violinplot(x=None, y=None, hue=None, data=None, order=None, hue_order=None, bw='scott', cut=2, scale='area', scale_hue=True, gridsize=100, width=0.8, inner='box', split=False, dodge=True, orient=None, linewidth=None, color=None, palette=None, saturation=0.75, ax=None, **kwargs)

It draws a combination of boxplot and kernel density estimate. A violin plot plays a similar role as a box and whisker plot. It shows the distribution of quantitative data across several levels of one (or more) categorical variables such that those distributions can be compared. Unlike a box plot, in which all of the plot components correspond to actual data points, the violin plot features a kernel density estimation of the underlying distribution. This can be an effective and attractive way to show multiple distributions of data at once, but keep in mind that the estimation procedure is influenced by the sample size, and violins for relatively small samples might look misleadingly smooth.

- Danceability [0 to 1]: Describes how suitable a track is for dancing based on a combination of musical elements including tempo, rhythm stability, beat strength, and overall regularity.
  In the below graphs you can see the distribution of liked and disliked data for danceability for both users. In both graphs the median value for both distributions is different from each other. And distribution for disliked songs is more wide compared to distribution of liked songs containing some outliers in the bottom part for user1.

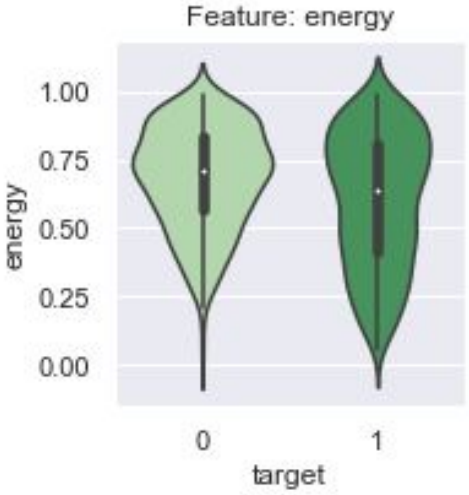


Graph 3, user1, danceability with target

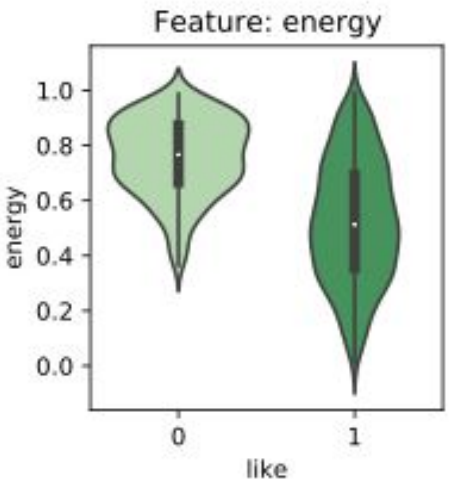


Graph 4, user2, danceability with target

- Energy [0 to 1]: Represents a perceptual measure of intensity and activity. Typically, energetic tracks feel fast, loud, and noisy.
  Distribution for liked songs and disliked songs for energy is different for both users. With different median values and distribution amplitude. For user1's disliked songs it contains some outliers.



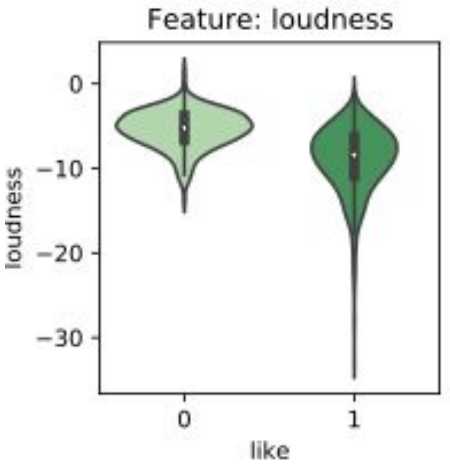
Graph 5, user1, energy with target



Graph 6, user2, energy with target

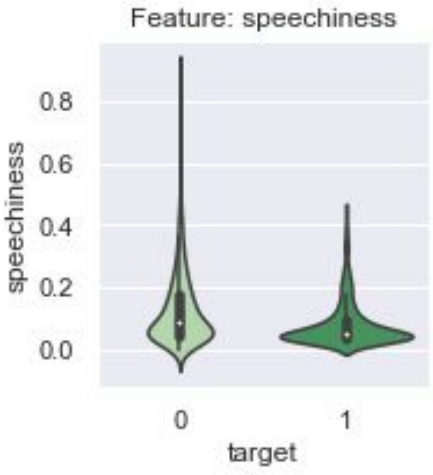
- Loudness (dB) [-60 to 0]: Overall loudness of a track in decibels (dB). Loudness values are averaged across the entire track. Loudness is the primary psychological correlate of physical strength (amplitude).
  In the following distribution representation of target with loudness, we can see that user1 has almost the same median values but user2 has somewhat different, while both distributions have larger outliers present and disliked songs with larger width.

Graph 7, user1, loudness with target

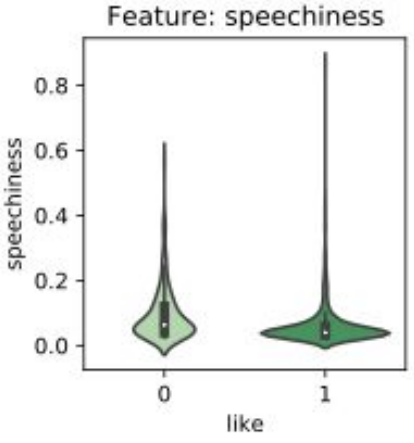


Graph 8, user2, loudness with target

- Speechness [0 to 1]: Detects the presence of spoken words in a track. The more exclusively speech-like the recording (e.g. talk show, audio book, poetry), the closer to 1.0 the attribute value
  Speechness contains large values in the 3rd quartile.

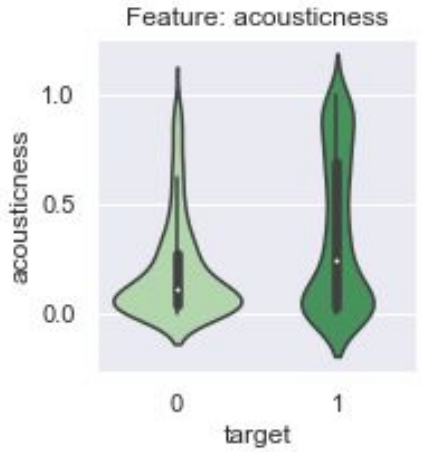


Graph 9, user1, speechiness with target

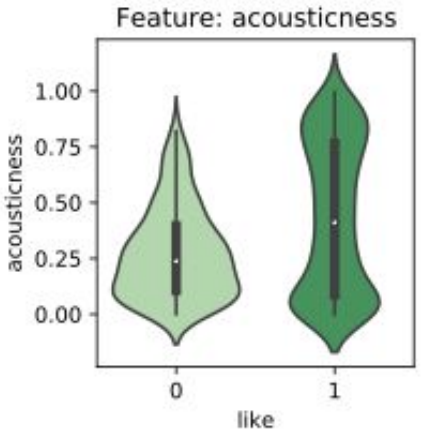


Graph 10, user2, speechiness with target

- Acousticness [0 to 1]**:** A confidence measure of whether the track is acoustic. 1.0 represents high confidence the track is acoustic.

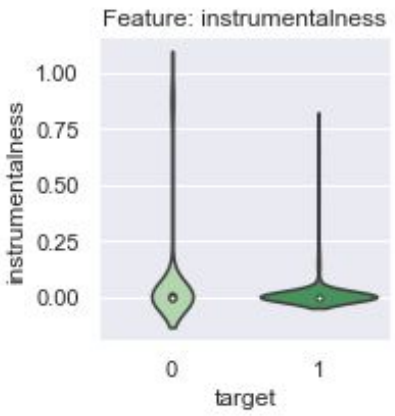


Graph 11, user1, acousticness with target

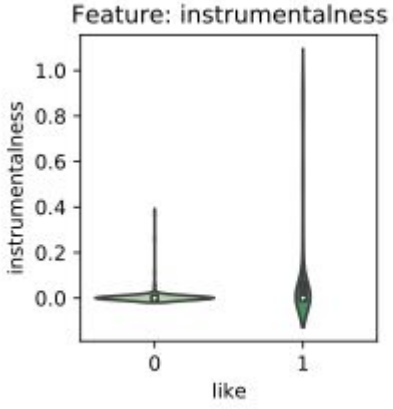


Graph 12, user2, acousticness with target

- Instrumentalness [0 to 1]: Predicts whether a track contains no vocals. “Ooh” and “aah” sounds are treated as instrumental in this context. Rap or spoken word tracks are clearly “vocal”. The closer the instrumentalness value is to 1.0, the greater likelihood the track contains no vocal content.

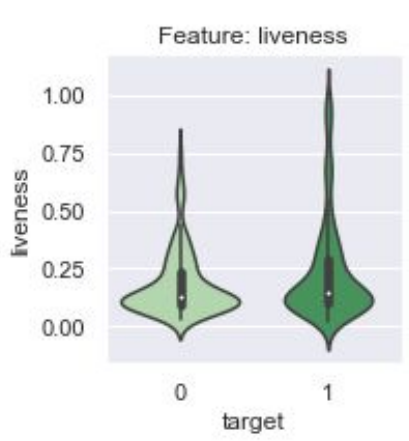


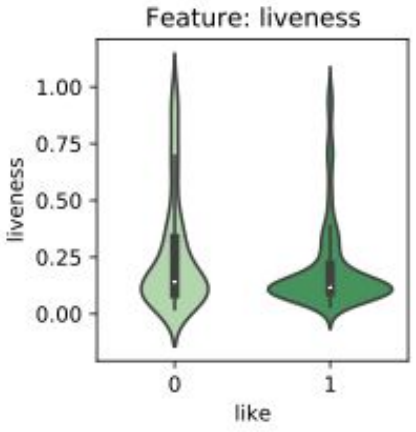
Graph 13, user1, instrumentalness With target



Graph 14, user2, instrumentalness with target

- Liveness [0 to 1]: Detects the presence of an audience in the recording. Higher liveness values represent an increased probability that the track was performed live. A value above 0.8 provides strong likelihood that the track is live.

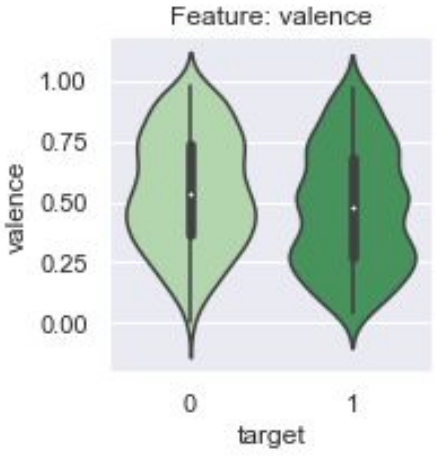


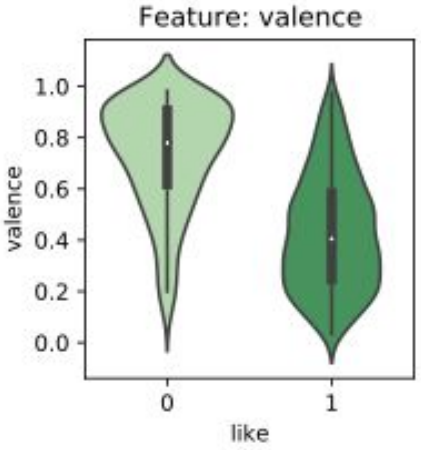


Graph 15, user1, liveness with target

Graph 16, user2, liveness with target

- Valence [0 to 1]:Describes the musical positiveness conveyed by a track. Tracks with high valence sound more positive (e.g. happy, cheerful, euphoric), while tracks with low valence sound more negative (e.g. sad, depressed, angry).

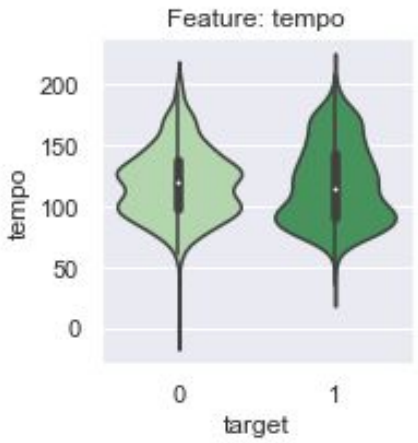




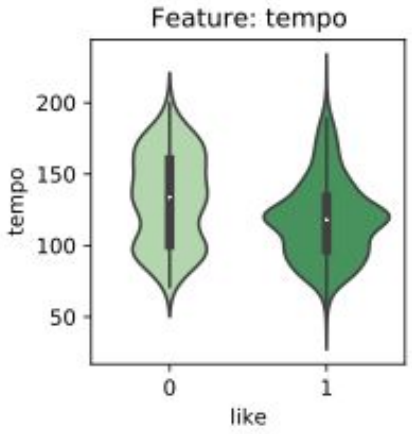
Graph 17, user1, valence with target

Graph 18, user2, valence with target

- Tempo (BPM) [~0 to ~210]: The overall estimated tempo of a track in beats per minute (BPM). In musical terminology, tempo is the speed or pace of a given piece and derives directly from the average beat duration.

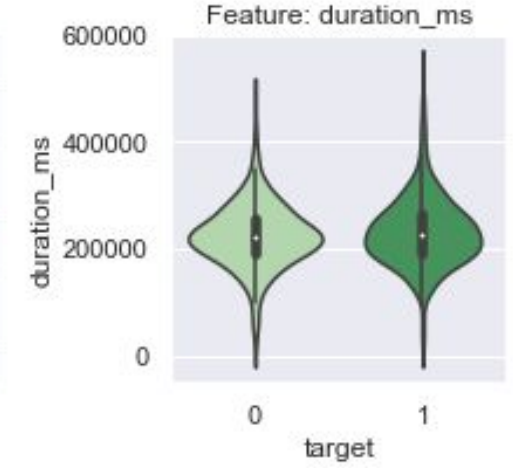


Graph 19, user1, tempo with target

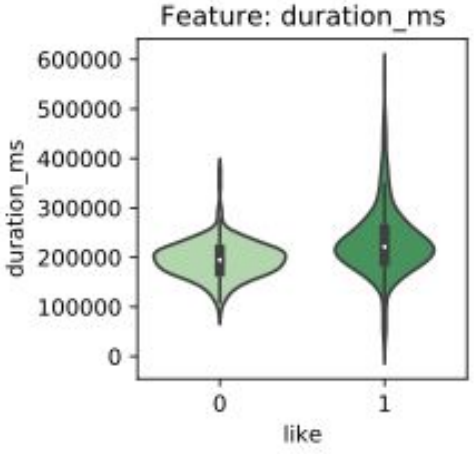


Graph 20, user2, tempo with target

- Duration (ms): The duration of the track in milliseconds.
-

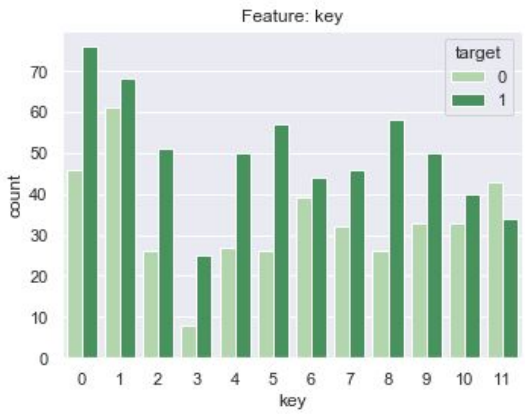


Graph 21, user1, duration with target

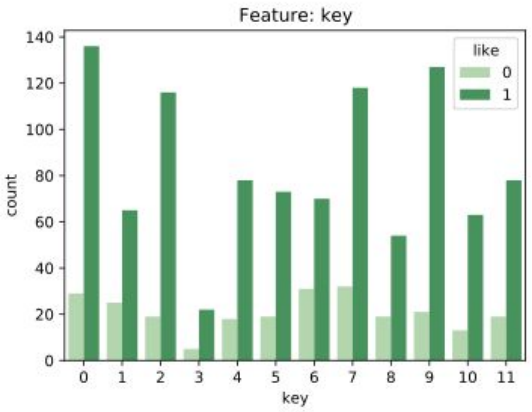


Graph 22, user2, duration with target

- Key [Categorical]: The estimated overall key of the track. Integers map to pitches using standard Pitch Class notation . E.g. 0 = C, 1 = C♯/D♭, 2 = D, and so on.
  Here you can see user1 has most liked songs on key 0 and disliked songs on key 1 too, while user2 has most liked songs has key 0 and disliked songs has key 6 or 7. It is somewhat clear from visualization that user1 data is hard to distinguish while user2 has more clear distinguished data.



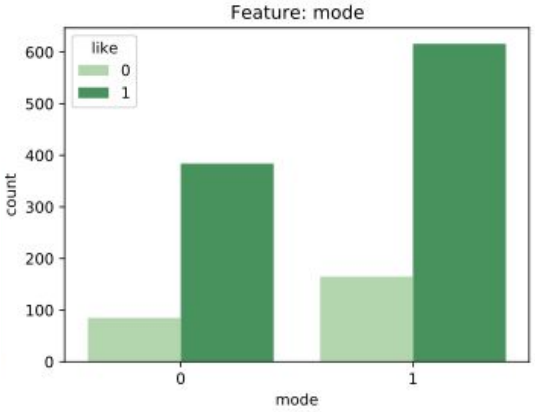
Graph 23, user1, key with target



Graph 24, user2, key with target

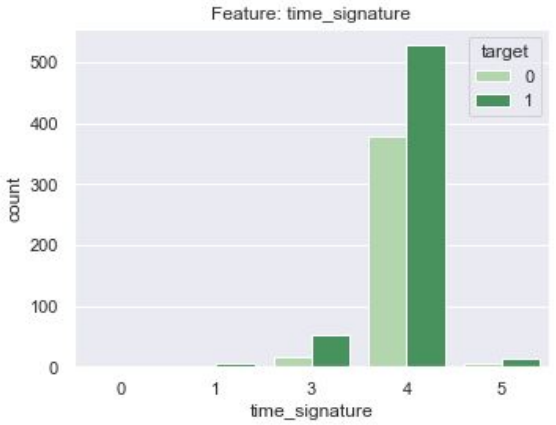
- Mode [Binary]:Mode indicates the modality (major or minor) of a track, the type of scale from which its melodic content is derived. Major is represented by 1 and minor is 0.
  As we know we have more liked songs in our data , we can see that we have liked songs values higher for both modes.

Graph 25, user1, mode with target

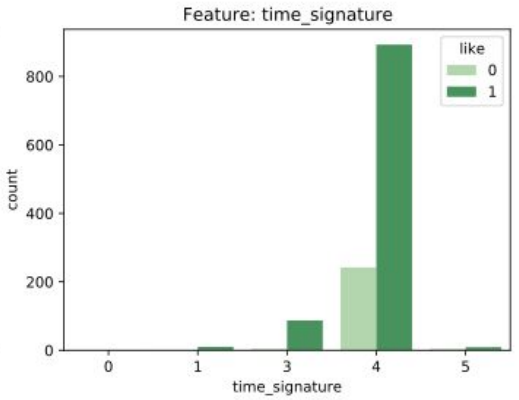


Graph 26, user2, mode with target

- Time Signature [Categorical]: An estimated overall time signature of a track. The time signature (meter) is a notational convention to specify how many beats are in each bar (or measure). E.g. 4/4; 3/4.
  Both users have same time signature higher for liked songs and disliked songs, which is 4.



Graph 27, user1, time_signature with target



Graph 28, user2, time signature with target

As we were able to visualize our data and get some insight information for all features, it is time to see if these features are correlated or not.

>>>Features correlation:

Data and feature correlation is considered one important step in the feature selection phase of the data pre-processing especially if the data type for the features is continuous. So what is data correlation?

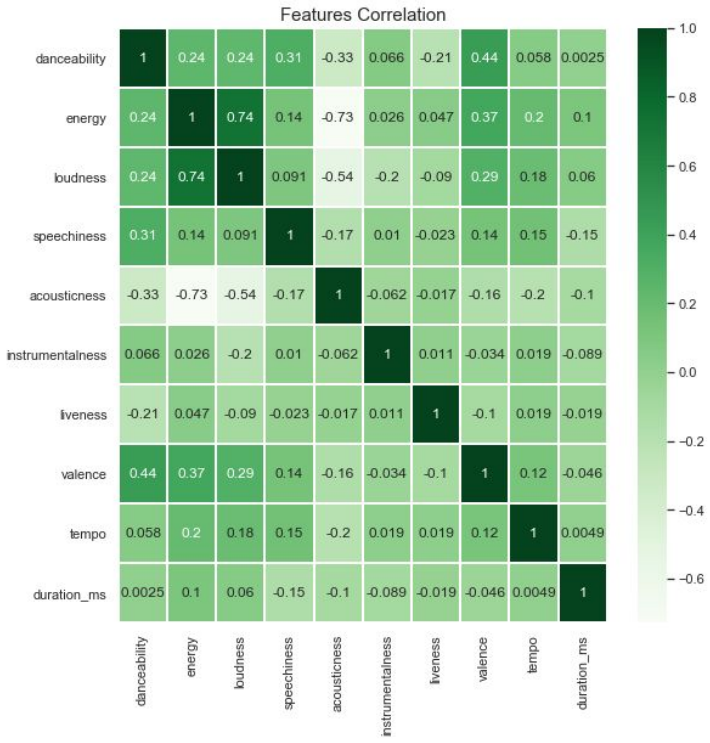
Data Correlation is a way to understand the relationship between multiple variables and attributes in your dataset. Using correlation, you can get some insights such as:

- One or multiple attributes depend on another attribute or a cause for another attribute.
- One or multiple attributes are associated with other attributes.

So, why is correlation useful?

- Correlation can help in predicting one attribute from another (great way to impute missing values).
- Correlation can (sometimes) indicate the presence of a causal relationship.
- Correlation is used as a basic quantity for many modelling techniques

❑ Complete correlation heatmap: correlation heatmap is a graphic way to represent correlation between features. Correlation varies from -1 to 1, where closer to 1 means higher positive correlation and closer to -1 indicates higher inverse correlation. As you can see in the graph below darker color green represents positive correlations except diagonal values because diagonal values represent correlation of a feature within itself and obviously it is gonna give correlation 1. And lighter the colour green, closer it moves to inverse correlation.

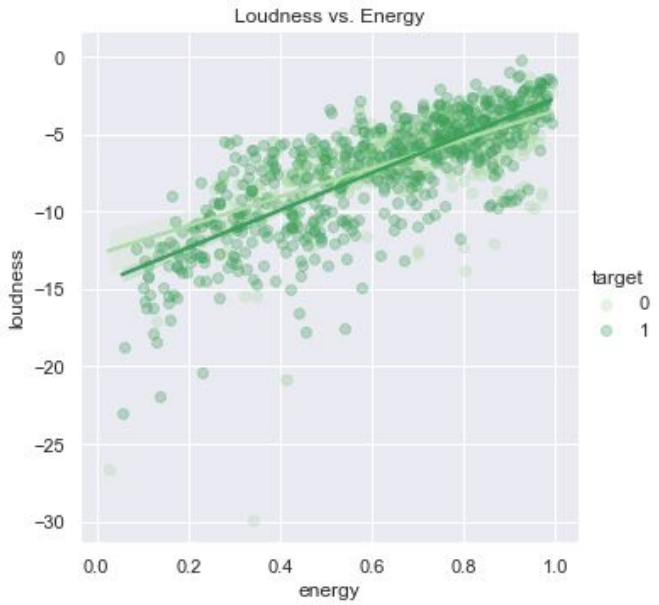


Graph 29 correlation heatmap

From the above heatmap it is clear that energy and loudness have the highest positive correlation with value 0.74, and highest inverse correlation with value -0.73 appears between energy and acousticness.

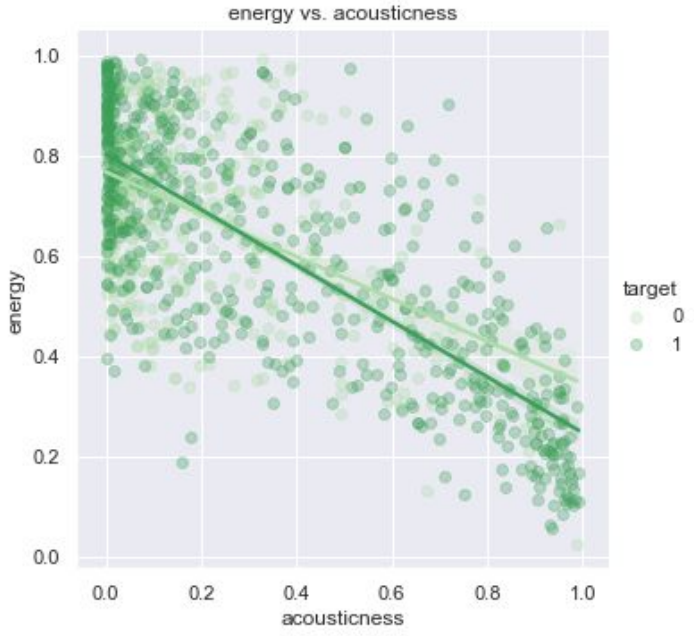
- Correlation positive: means that if feature A increases then feature B also increases or if feature A decreases then feature B also decreases. Both features move in tandem in a linear manner.

  In the below graph you can see that loudness and energy have linear relationships which can be seen in graphs with straight lines in dark green color. It means energy increases as loudness increases and it decreases as loudness decreases.



Graph 30, positive correlation(loudness vs energy)

- Correlation negative: Negative correlation: means that if feature A increases then feature B decreases and vice versa. As you can see in the bellow graph energy and acousticness have an inverse relationship which means energy is decreasing when acousticness is increasing or vice versa.

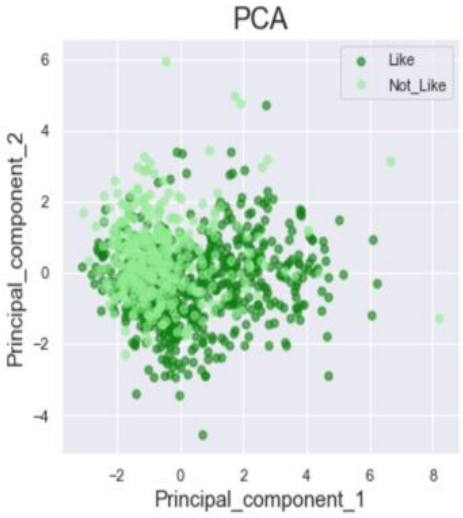


Graph 31, inverse correlation(energy vs acousticness)

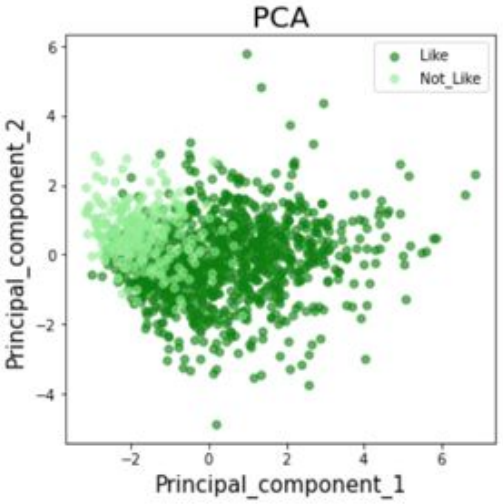
After feature correlation we are going to see our both user's data with PCA to decide whether our target classes are distinguishable or not.

>>> Principal Component Analysis(like vs. don't like) :

Principal component analysis [14] is linear dimensionality reduction technique that can be utilized for extracting information from a higher dimensional space by projecting it into a lower dimensional subspace. The dimensions we are talking about are the features of our songs here



Graph 32, user1, PCA



Graph33, user2, PCA

In the graph 32 for user1 you can see that like/not_like data is hardly distinguishable. User1 contains songs with mixed taste or you can say mixed features so it is quite hard for us to create a linear separation between our target classes. While for user 2, in Graph33 it is more clear that classes for user2 are distinguishable and we could create a better linear separation between liked songs and disliked songs.

In the below cell you can see the implementation of PCA:

```
# PCA - Data projection in 2D
pca = PCA(n_components=2)
principal_components = pca.fit_transform(songs_df_std)
principal_df = pd.DataFrame(data = principal_components,
                            columns = ['principal component 1', 'principal component 2'])
# Concatenate std features and target ('like')
songs_df_std_target = pd.concat([principal_df, songs_df[[song_target]]],
                                axis = 1)

# Plot 2D PCA
fig = plt.figure(figsize = (5,5))
ax = fig.add_subplot(1,1,1)
ax.set_xlabel('Principal_component_1', fontsize = 15)
ax.set_ylabel('Principal_component_2', fontsize = 15)
targets = [1, 0]
colors = ['green', 'lightgreen']
for target, color in zip(targets, colors):
    idx = songs_df_std_target['like'] == target
    ax.scatter(songs_df_std_target.loc[idx, 'principal component 1'],
               songs_df_std_target.loc[idx, 'principal component 2'],
               c = color,
               s = 30,
               alpha=0.6
              )
ax.legend(['Like','Not_Like'])
plt.title('PCA', fontsize = 20)
plt.tight_layout()
```

Cell 18, PCA implementation

>>> Mann-Whitney test and get p-value:

In statistics, the Mann–Whitney U test [15] (also called the Mann–Whitney–Wilcoxon (MWW), Wilcoxon rank-sum test, or Wilcoxon–Mann–Whitney test) is a nonparametric test of the null hypothesis that the probability that a randomly selected value from one population is less than a randomly selected value from a second population is equal to the probability of being greater.

This test can be used to investigate whether two independent samples were selected from populations having the same distribution. The Mann-Whitney U test is often used when the assumptions of the independent samples t-test are violated. A similar nonparametric test used on dependent samples is the Wilcoxon signed-rank test.

For user2, all continuous features distributions were significantly different when comparing liked vs. not liked songs; while for user1 tempo and duration were not significantly different.

Significant features	P-value (user1)	P-value (user2)
Valence	4.255038e-06	3.809726e-59
Loudness	2.724966e-05	6.398669e-50
Danceability	2.005800e-45	8.823610e-25
Instrumentalness	3.129358e-02	2.107401e-43
Energy	5.366387e-07	9.897383e-50
Duration	> 0.05 (Not Significant)	4.523344e-21
Speechiness	2.708129e-17	2.388415e-19
Tempo	> 0.05 (Not Significant)	4.779544e-11
Liveness	1.778741e-03	4.265676e-02
Acousticness	3.108409e-08	5.291756e-08

Table 7, Mann-Whitney test, p-values

6.3. Data Cleaning and Transformation

In this section we are going to clean our data and transform it in our likely format. As from the section 6.1 it is clear that:

- There is no missing and null values in our data
- We are not adding extra features at this point
- We are not modifying features

We do not have things to do in this section as our data is already cleaned and in the form our model required.

7. Models and evaluation metrics

In this section we are going to describe the implementation and evaluation of our classification and recommendation models. But before diving into each model we will be discussing what evaluation metric is and how and what parameters we are using to evaluate the models.

>>> Evaluation metrics:

What do we want to optimize for? Most of the businesses fail to answer this simple question. Every business problem is a little different and it should be optimized differently. We all have created classification models. A lot of time we try to improve our models by increasing accuracy. But do we really want accuracy as a metric of our model performance? What if we are predicting the number of asteroids that will hit Earth. Just say zero all the time. And you will be 99% accurate. My model can be reasonably accurate, but not at all valuable. What should we do in such cases?
In this part of the section we are going to see different evaluation metrics for our classification models. These are the the most common used metrics so far:

- Accuracy
- Precision
- Recall
- F1 score
- Log loss
- AUC

		Actual	
		Positive	Negative
Predicted	Positive	True Positive	False Positive
	Negative	False Negative	True Negative

Picture 9, TP,TN,FP,FN Representation, Source: [pic:9]

In the above picture you can see the parameters to evaluate our metrics.
A “True Positive” is an outcome where the model correctly predicts the positive class. Similarly, a “True Negative” is an outcome where the model correctly predicts the negative class. A “False positive” is an outcome where the model incorrectly predicts the positive outcome and a “False Negative” is an outcome where the model incorrectly predicts the negative class.

- Accuracy: Accuracy is one metric for evaluating classification models. It is the fraction of predictions our model got right. Or we can represent Accuracy in this way:

  $$Accuracy = (number\ of\ correct\ predictions) \div (total\ number\ of\ predictions)$$

  For binary classification accuracy can also be calculated in terms of positives and negatives as follows:

  $$accuracy = (TP + TN)/(TP + TN + FP + FN)$$

  Here TP = True positive, TN = true negative, FP = false Positive, FN = False negative

  Accuracy alone does not tell the full story when you are working with a class imbalanced dataset, like our one, where there is a significant disparity between the numbers of positives and negative labels. It is a valid metric in the scenario where we have well balanced problems and not skewed or no class imbalance.

- Precision: when the question arises: what proportion of the predicted positives is truly positive, the answer is precision. We can represent precision with this equation down here:

  $$precision = (TP)/(TP + FP)$$

  Precision is a valid choice of evaluation metric when we want to be very sure of our prediction. For example: if we are building a system to predict if we should decrease the credit limit on a particular account, we want to be very sure about our prediction or it may result in customer dissatisfaction. Limitation for this metric is “being very precise means our model will leave a lot of credit defaulters untouched and hence lose money”.

- Recall: Another very useful measure is *recall*, which answers a different question: what proportion of actual positives is correctly classified?

  $$recall = TP/(TP + FN)$$

  When to use? Recall is a valid choice of evaluation metric when we want to capture as many positives as possible. For example: if we are building a system to predict if a person has cancer or not, we want to capture the disease even if we are not very sure. Caveats: *Recall is 1 if we predict 1 for all examples.* And thus comes the idea of utilizing tradeoff of precision vs. recall — *F1 Score*.

- F1 Score: it is a number between 0 and 1, and it is the harmonic mean of precision and recall. This is the formula presentation for f1 score:

  $$F1 = 2 * (precision * recall)/(precision + recall)$$

Let us start with a binary prediction problem. *We are predicting if an asteroid will hit the earth or not.* So if we say “No” for the whole training set, our precision here is 0. What is the recall of our positive class? It is zero. What is the accuracy? It is more than 99%. And hence the F1 score is also 0. And thus we get to know that the classifier that has an accuracy of 99% is basically worthless for our case. And hence it does not solve our problem.

When to use it? When we want to have a model with both good precision and recall.

Simply stated the *F1 score sort of maintains a balance between the precision and recall for your classifier*. If your precision is low, the F1 is low and if the recall is low again your F1 score is low.

The main issue with F1 score is that it gives equal weight to precision and recall. We might sometimes need to include domain knowledge in our evaluation where we want to have more recall or more precision.

In our problem we will be looking for precision, recall and f1 score. But our main evaluation metric will be F1 score as we want the balance of precision and recall.

Moving on, now it is time to finally prepare our models but before going completely hands on we will be understanding what each model is and how it works.

7.1. Classification

In machine learning we see two most common methods which are supervised learning and unsupervised learning. Classification and regression comes under the supervised learning where our target is observable. Difference between classification and regression is that classification is used where target is discrete and regression is used where target is continuous.

The goal of classification problems is to take an input vector X and classify it into one of K possible classes of our target y. Most often, these classes are mutually exclusive. And often, the target is simply binary. We can divide a classification problem into two steps:

- Estimate the probability of our vector of features X belonging to every one of the K possible classes.
- Translate the vector of probabilities into a final class assignment. A common and simple approach is to take the class with maximum probability. This, for a binary classification problem leads to the following decision rule: X_i = 1 if P(Y = 1|X) >= 0.5 else 0.

Classification basically categorizes a set of data into classes and the classes we are using for our data are “liked_songs as 1” and “disliked_songs as 0”.

$$y = \alpha + \beta x$$

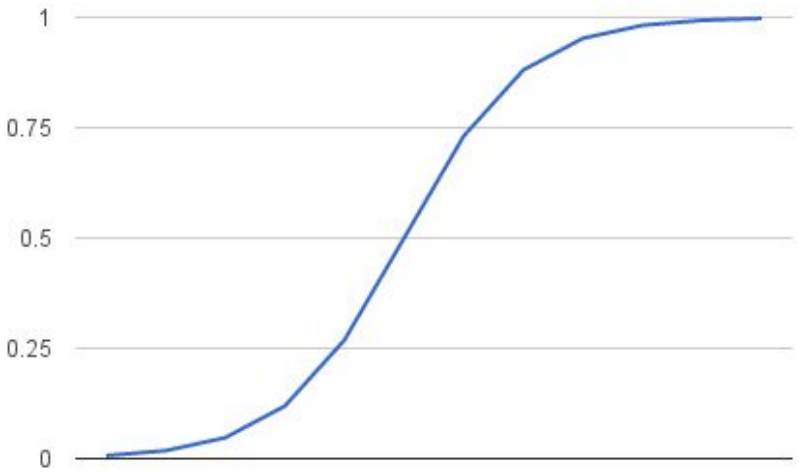
Imagine we have a linear problem for our classification. By default, linear regression is not a good method for classification. It can lead to values outside of the [0, 1] range and it’s not robust to outliers.
For modeling a classification problem with a linear model, we can make use of logistic regression. We will be using Logistic Regression as a baseline model. The idea behind the outline is that we will be moving from the simplest model to more complex ones.

7.1.1. Logistic Regression (Baseline):

Logistic regression [15] is named for the function used at the core of the method, the logistic function. The logistic function is an S-shaped curve that can take any real-valued number and map it into a value between 0 and 1, but never exactly at those limits.

$$f(x) = 1 / (1 + e^x)$$

Where e is the base of the natural logarithms and value is the actual numerical value that you want to transform. Below is a plot of the numbers between -5 and 5 transformed into the range 0 and 1 using the logistic function.



Graph 34, Logistic function

Let’s see how it is used in logistic regression. Logistic regression uses an equation as the representation, very much like linear regression. Input values (x) are combined linearly using weights or coefficient values to predict an output value (y). A key difference from linear regression is that the output value being modeled is a binary value (0 or 1) rather than a numeric value.

$$y = f(X) = \alpha + \beta_1 x_1 + \beta_2 x_2 + \ldots + \beta_n x_n$$

Logistic regression that passes our original linear equation through a standard logistic function

$$y = f(X) = logistic(\alpha + \beta_1 x_1 + \beta_2 x_2 + ... + \beta_n x_n) = 1 / (1 + e^{x\beta})$$

Basically it describes and estimates the relationship between one dependent binary variable (target 'like'; 0 or 1) and independent variables (song features).
Now that we know what logistic regression is, lets build this model for our classification problem where inputs are our audio features and target is the classes songs we like and songs we do not like.

```
from sklearn.linear_model import LogisticRegression
lr = LogisticRegression()
lr.fit(X_train, y_train)
y_pred = lr.predict_proba(X_test)[:, 1]
```

Cell 19, Logistic regression model

So here we imported our model from Scikit-Learn library. We instantiated the model with lr, then we fit the model using our data, and we get the probability of each song being classified as liked in this case. If the probability is greater or equal 0.5, we classify a song as "liked", if it is smaller than 0.5, as "not liked". Evaluation metrics look like this for both users:

Logistic Regression	**User**	
Metrics	**User1(Neha)'s songs**	**User2(Bruna)'s songs**
Precision	**0.61500**	**0.80952**
Recall	**1.00000**	**0.96480**
F1-Score	**0.74893**	**0.88014**

Table 7, evaluation results for Logistic regression

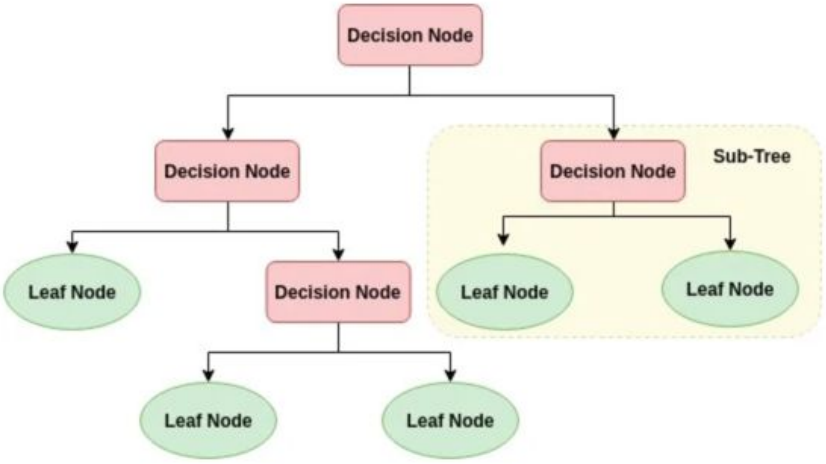
So from the above table we can see that this model is working well for user2 while for user1 is not that good in terms of precision. It is quite fascinating to see user1 has a perfect recall of 1 which indicates that she likes all the songs which is not true in reality. F1 score is balancing both precision and recall, which sometimes can be misleading as in user1 case it has perfect recall but very low precision, highlighting the importance of evaluating multiple metrics.

Lets move to another model. We will be seeing Decision Trees in the next section.

7.1.2. Decision Tree

Decision-tree [16] methods are a family of models that generate predictions by making sequential splits to the input variables based on their values and assign a prediction to each leaf - usually through a simple model. In the case of regression that simple model can be just an average or the median. In the case of a classification problem, that model can be the mode or the median. A flowchart-like tree structure:

- Internal node represents feature
- Branch represents a decision rule
- Each leaf node represents the outcome



Picture 10, Decision Tree, source: [pic:10]

As a rule, a decision tree starts with a root node, after which it branches into various possible outcomes. These outcomes then lead to additional nodes until one reaches the right outcome. Therefore, the purpose of using a decision tree is to present the required information to the user in the most efficient way, using a logical question structure, providing relevant information, and avoiding irrelevant questions and unnecessary detours.

A classification decision tree is where the outcome was a variable like ‘fit’ or ‘unfit’. Here the decision variable is categorical/discrete. Such a tree is built through a process known as binary recursive partitioning. This is an iterative process of splitting the data into partitions, and then splitting it up further on each of the branches. Binary recursive partitioning approach is also commonly known as divide and conquer because it splits the data into subsets, which are then splitted repeatedly into even smaller subsets, and so on and so forth until the process stops when the algorithm determines the data within the subsets are sufficiently homogenous, or another stopping criterion has been met.

>>> Advantages of Classification with Decision Trees:

- Inexpensive to construct.
- Extremely fast at classifying unknown records.
- Easy to interpret for small-sized trees

- Accuracy comparable to other classification techniques for many simple data sets.
- Excludes unimportant features.

>>> Disadvantages of Classification with Decision Trees:

- Easy to overfit.
- Decision boundary restricted to being parallel to attribute axes.
- Decision tree models are often biased toward splits on features having a large number of levels.
- Small changes in the training data can result in large changes to decision logic.
- Large trees can be difficult to interpret and the decisions they make may seem counter intuitive.

Now we know what decision tree is, it is time to build our model. In the cell below you can see the implementation:

```
from sklearn.tree import DecisionTreeClassifier
model2=DecisionTreeClassifier()
model2.fit(X_train,y_train)
y_pred = model2.predict_proba(X_test)[:, 1]
```

Cell 20, decision tree classifier implementation

So here we imported our model from Scikit-Learn library. We called the model with model2 and we fit our data and get the probabilities, similarly as described above. Evaluation metrics looks like this for both users:

Decision Tree	**User**	
Metrics	**user1(Neha)'s songs**	**user2(Bruna)'s songs**
Precision	**0.75652**	**0.90899**
Recall	**0.70731**	**0.91760**
F1-Score	**0.75738**	**0.91398**

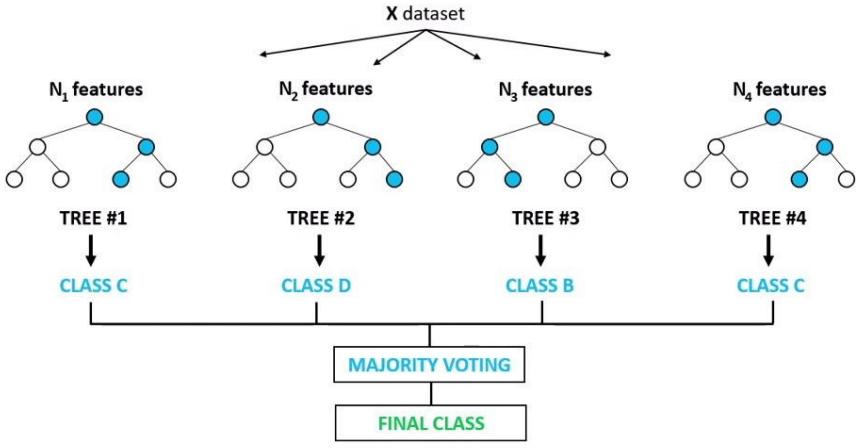
Table 8 evaluation of decision tree for both users

From the above table we can see that the model performance is better for both users compared to logistic regression. User1 is scoring around 75% of f1 score while user2 is doing a score around 91%.

In the next section we will be seeing the Random Forest model for our classification problem.

7.1.3. Random Forest

Random forest,[17] like its name implies, consists of a large number of individual decision trees that operate as an ensemble. Each individual tree in the random forest outputs a class



prediction and the class with the most votes becomes our model's prediction.

Picture 11,random forest classifier,source:[pic:11]

Random forest is an ensemble algorithm that aims at overcoming the limitations of decision-tree methods. It has gained much popularity due to its high performance on predictive tasks, although new gradient boosting methods have taken a toll on its usage.

Random forest tries to overcome the overfitting issues of decision trees through bagging and subsampling. Bagging is a technique that consists in taking random subsamples of data, training decision-tree algorithms on those subsamples of data and averaging their splits out. Similarly, subsampling or sampling random subspaces randomly samples from the feature space and trains a decision-tree algorithm on those features.

>>> Advantage of Random forest:

- It can come out with very high dimensional (features) data, and no need to reduce dimension, no need to make feature selection
- It can judge the importance of the feature
- Can judge the interaction between different features
- Not easy to overfit
- Training speed is faster, easy to make parallel method
- It is relatively simple to implement
- For unbalanced data sets, it balances the error.
- If a large part of the features are lost, accuracy can still be maintained.

>>> Disadvantages of Random Forest:

- Random forests have been shown to fit over certain noisy classification or regression problems.
- For data with different values, attributes with more values will have a greater impact on random forests, so the attribute weights generated by random forests on such data are not credible.

Now we know what random forest is, it is time to build our model. In the cell bellow you can see the implementation:

```
from sklearn.ensemble import RandomForestClassifier

rf = RandomForestClassifier()
rf.fit(X_train, y_train)
y_pred = rf.predict_proba(X_test)[:, 1]
```

Cell 21, random forest classifier implementation

So here we imported our model from Scikit-Learn library. We called the model with rf and we fit our data, and get probabilities of each song being liked. Evaluation metrics looks like this for both users:

Random Forest	User	
Metrics	**user1(Neha)'s songs**	**user2(Bruna)'s songs**
Precision	**0.82258**	**0.90305**
Recall	**0.82927**	**0.97859**
F1-Score	**0.80065**	**0.93859**

Table 9, evaluation of random forest classifier for both users.

From the above table we can see that model performance is better for both users compared to previous models. User1 is scoring around 80% of f1 score while user2 is doing a score around 93%.

In the next section we will be seeing a boosting model for our classification problem.

7.1.4. XGBoost

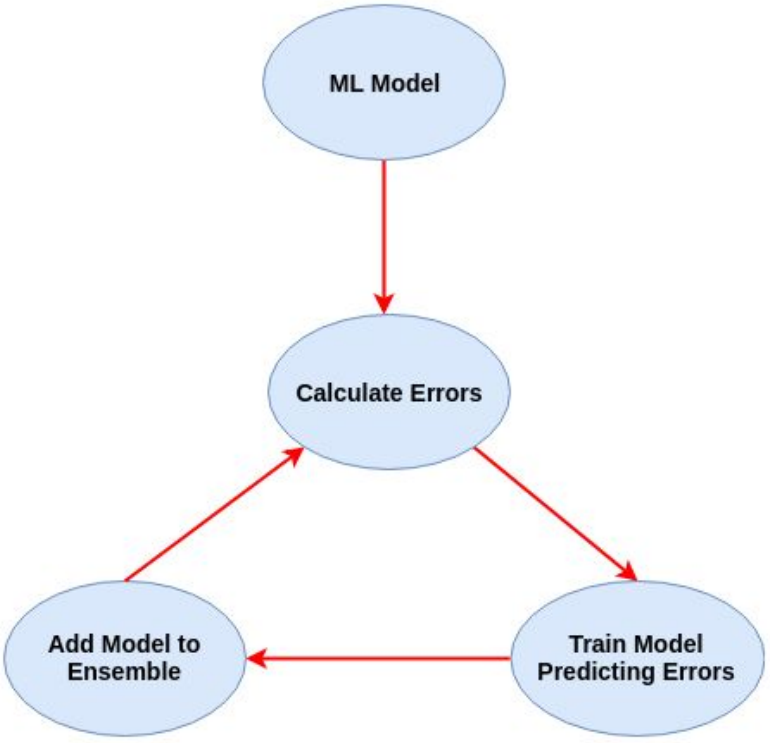
With a regular machine learning model, like a decision tree, we'd simply train a single model on our dataset and use that for prediction. We might play around with the parameters for a bit or augment the data, but in the end we are still using a single model. Even if we build an ensemble, all of the models are trained and applied to our data separately.

Boosting,[18] on the other hand, takes a more iterative approach. It's still technically an ensemble technique in that many models are combined together to perform the final one, but takes a more clever approach.

Rather than training all of the models in isolation of one another, boosting trains models in succession, with each new model being trained to correct the errors made by the previous ones. Models are added sequentially until no further improvements can be made.

The advantage of this iterative approach is that the new models being added are focused on correcting the mistakes which were caused by other models. In a standard ensemble method where models are trained in isolation, all of the models might simply end up making the same mistakes!

Gradient Boosting specifically is an approach where new models are trained to predict the residuals (i.e errors) of prior models. You can see the outline in the picture below:



Picture 12, outline of XGBoost Classifier, Source: [pic:12]

7.1.4.1. XGBoost default

"State-of-the-art" machine learning algorithm to deal with structured data.

- Speed and performance
- Core algorithm is parallelizable
- Consistently outperforms other algorithm methods
- Wide variety of tuning parameters
- Boosting: a sequential technique which works on the principle of an ensemble.
    - Combines a set of weak learners
    - Default base learners: tree ensembles.
    - Trees are grown one after another
    - Outcomes predicted correctly are given a lower weights and the ones miss-classified are weighted higher
    - Attempts to reduce the misclassification rate are made in subsequent iterations

We will be using default parameters first to check how our model behaves and in the next part we will be tuning those parameters to improve our model's performance. In the cell below you can see the implementation:

```
from xgboost import XGBClassifier
xgb = XGBClassifier()
xgb.fit(X_train, y_train)
y_pred = xgb.predict_proba(X_test)[:, 1]
```

Cell 22, implementation of XGBoost default

In the table below you can see the performance of our model for both users:

XGBoost	User	
Metrics	**user1(Neha)'s songs**	**user2(Bruna)'s songs**
Precision	**0.82031**	**0.92810**
Recall	**0.85366**	**0.95820**
F1-Score	**0.80129**	**0.94280**

Table 10, XGBoost default model evaluation for both users

From the above table we can see that model performance is better for both users compared to previous models. User1 is scoring around 80% of f1 score while user2 is doing an almost perfect score around 94%.

That just sums up about the basics of XGBoost. But there are some more cool features that'll help you get the most out of your models.

- The gamma parameter can also help with controlling overfitting. It specifies the minimum reduction in the loss required to make a further partition on a leaf node of the tree. I.e if creating a new node doesn't reduce the loss by a certain amount, then we won't create it at all.
- The booster parameter allows you to set the type of model you will use when building the ensemble. The default is gbtree which builds an ensemble of decision trees. If your data isn't too complicated, you can go with the faster and simpler gblinear option which builds an ensemble of linear models.

In the next section we will be seeing a boosting model with hyperparameter tuning for our classification problem.

7.1.4.2. XGBoost hyperparameter tuning

Setting the optimal hyperparameters of any ML model can be a challenge. So why not let Scikit-Learn do it for you? We can combine Scikit-Learn's random grid search with an XGBoost classifier quite easily. In the below cell you can see how we used grid search for hyperparameter tuning:

```
# Hyperparameter tunning for XGBoost
ratio = (1 - y).sum() / y.sum() # proportion of like not_like
param_grid = {
    'max_depth': [4,5,6,7],
    'n_estimators': list(range(200, 600)),
    'learning_rate': list(np.logspace(np.log10(0.005), np.log10(0.1), base = 10, num = 1000)),
    'reg_alpha': list(np.linspace(0.5, 1.5)),
    'reg_lambda': list(np.linspace(0.5, 1.5)),
    'gamma': [0,5,10,15,20],
    'colsample_bytree': list(np.linspace(0.5, 0.9, 10)),
    'colsample_bylevel':list(np.linspace(0.5, 0.9, 10)),
    'min_child_weight': [5,10,20,30,40],
    'subsample': list(np.linspace(0.5, 1, 100)),
    'scale_pos_weight': np.linspace(ratio-3, ratio+3),
}

rs = RandomizedSearchCV(XGBClassifier(),
                        param_grid,
                        scoring='f1',
                        cv=3,
                        verbose=1,
                        n_jobs=-1,
                        n_iter=1000)
rs.fit(X, y)
best_params = rs.best_params_
rs.best_params_
```

Cell 23, grid search for hyperparameter tuning for XGBoost

For the randomized grid search we choose 1,000 iterations and some range for our interesting parameters of model. In the cell below you can see the tuned parameters:

```
Fitting 3 folds for each of 1000 candidates, totalling 3000 fits
[Parallel(n_jobs=-1)]: Using backend LokyBackend with 4 concurrent workers.
[Parallel(n_jobs=-1)]: Done  76 tasks      | elapsed:    5.2s
[Parallel(n_jobs=-1)]: Done 376 tasks      | elapsed:   21.5s
[Parallel(n_jobs=-1)]: Done 876 tasks      | elapsed:   54.0s
[Parallel(n_jobs=-1)]: Done 1656 tasks      | elapsed:  1.5min
[Parallel(n_jobs=-1)]: Done 2744 tasks      | elapsed:  2.4min
[Parallel(n_jobs=-1)]: Done 3000 out of 3000 | elapsed:  2.6min finished
```

```
{'subsample': 0.8838383838383839,
 'scale_pos_weight': 2.7602040816326525,
 'reg_lambda': 1.0918367346938775,
 'reg_alpha': 1.2959183673469385,
 'n_estimators': 223,
 'min_child_weight': 5,
 'max_depth': 4,
 'learning_rate': 0.05047405878882374,
 'gamma': 0,
 'colsample_bytree': 0.8111111111111111,
 'colsample_bylevel': 0.9}
```

Cell 24, tuned hyperparameter

After using the tuned parameters this is what our evaluation metric looks like:

XGBoost HyperparamOpt	User	
Metrics	**user1(Neha)'s songs**	**user2(Bruna)'s songs**
Precision	**0.72623**	**0.90984**
Recall	**0.90718**	**0.97960**
F1-Score	**0.80631**	**0.94331**

Table 11, evaluation of XGboost model with hyperparameter tuning

As we can see from the table that our xgboost model with hyperparameters is doing very well compared to all the other models. We will be using this model for validation. Before doing that lets see the summary table of all the models' evaluation metrics:

User1(Neha)'s Songs	Models				
Metrics	Logistic Regression	Decision Tree	Random Forest	XGBoost	XGBoost Hyperparam
Precision	0.61500	0.75652	0.82258	0.82031	0.72623
Recall	1.00000	0.70731	0.82927	0.85366	0.90718
F1-Score	0.74893	0.75738	0.80065	0.80129	0.80631

Table12, user1(neha)'s models performance comparison

User2(Bruna's) Songs	Models				
Metrics	Logistic Regression	Decision Tree	Random Forest	XGBoost	XGBoost Hyperparam
Precision	0.80952	0.90899	0.90305	0.92810	0.90984
Recall	0.96480	0.91760	0.97859	0.95820	0.97960
F1-Score	0.88014	0.91398	0.93859	0.94280	0.94331

Table 13, user2(Bruna)'s models performance comparison

From the tables 12 and 13 we can see that the hyperparameter XGBoost model has the best performance for both users in terms of f1 score.

7.2. Recommender System

In this section we are going to build our final model that is the recommender system. Before going hands on we will be seeing in depth what a recommendation system is and how it works.

During the last few decades, with the rise of Youtube, Amazon, Netflix and many other such web services, recommender systems have taken more and more place in our lives. From e-commerce (suggest to buyers articles that could interest them) to online advertisement (suggest to users the right contents, matching their preferences), recommender systems are

today unavoidable in our daily online journeys. In a very general way, recommender systems are algorithms aimed at suggesting relevant items to users (items being movies to watch, text to read, products to buy or anything else depending on industries).

Recommender systems are really critical in some industries as they can generate a huge amount of income when they are efficient or also be a way to stand out significantly from competitors. As a proof of the importance of recommender systems, we can mention that, a few years ago, Netflix organised a challenge (the “Netflix prize”) where the goal was to produce a recommender system that performs better than its own algorithm with a prize of one million dollars to win.

In this section, we will go through different paradigms of recommender systems. For each of them, we will present how they work, describe their theoretical basis and discuss their strengths and weaknesses. Our main focus will be on the models we will be using for our project.

Types of recommendation system:

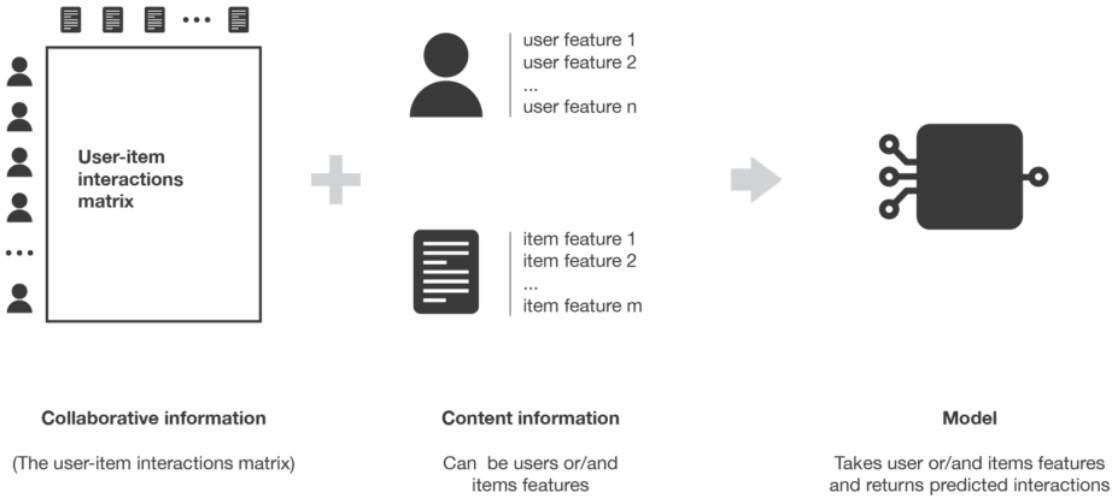
- Content-based filtering methods
- Collaborative filtering methods
  - Model based
  - Memory based
- Hybrid methods

The method we are using is content-based. In this part of section we will elaborate how content-based filtering works.

- Content-based filtering method:

Unlike collaborative methods that only rely on the user-item interactions, content-based approaches use additional information about users and/or items. If we consider the example of a movies recommender system, this additional information can be, for example, the age, the sex, the job or any other personal information for users as well as the category, the main actors, the duration or other characteristics for the movies (items).

Then, the idea of content-based methods is to try to build a model, based on the available “features”, that explain the observed user-item interactions. Still considering users and movies, we will try, for example, to model the fact that young women tend to rate better some movies, that young men tend to rate better some other movies and so on. If we manage to get such model, then, making new predictions for a user is pretty easy: we just need to look at the profile (age, sex, …) of this user and, based on this information, to determine relevant movies to suggest.



Picture 13, content based filtering, source: [pic:13]

Content-based methods suffer far less from the cold start problem than collaborative approaches: new users or items can be described by their characteristics (content) and so relevant suggestions can be done for these new entities. Only new users or items with previously unseen features will logically suffer from this drawback, but once the system is "old" enough, this has few to no chance to happen.

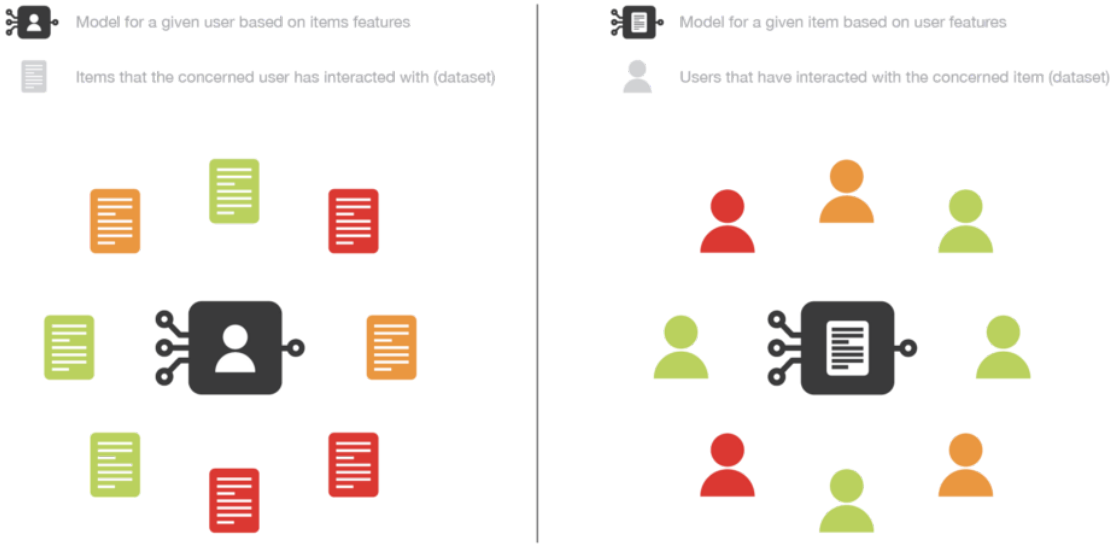
>>>Concept of content-based methods:

In content-based methods, the recommendation problem is casted into either a classification problem (predict if a user "likes" or not an item) or into a regression problem (predict the rating given by a user to an item). We are going to set a model that will be based on the user and/or item features at our disposal (the "content" of our "content-based" method).

If our classification is based on item features, we say the approach is item-centred: modelling, optimisations and computations can be done "by item". In this case, we build and learn one model by item based on the item's features trying to answer the question "what is the probability for each user to like this item?" (or "what is the rate given by each user to this item?", for regression). The model associated with each item is naturally trained on data related to this item and it leads, in general, to pretty robust models as a lot of users have interacted with the item. However, the interactions considered to learn the model come from every user and even if these users have similar characteristics (features) their preferences can be different. This means that even if this method is more robust, it can be considered as being less personalised (more biased) than the user-centred method thereafter.

If we are working with user features, the method is then user-centred: modelling, optimisations and computations can be done "by user". We then train one model by user based on user's features that tries to answer the question "what is the probability for this

user to like each item?" (or "what is the rate given by this user to each item?", for regression). We can then attach a model to each user that is trained on its data: the model obtained is more personalised than its item-centred counterpart as it only takes into account interactions from the considered user. However, most of the time a user has interacted with relatively few items and, so, the model we obtain is far less robust than an item-centred one.



Picture 14, Illustration of the difference between item-centred and user-centred content based methods, source: [pic:14]

Now we know what recommendation system is and how it works, we will be explaining the implementation of our system.

We merged both users data for this part of our project and in the end we got 2,460 songs in total.

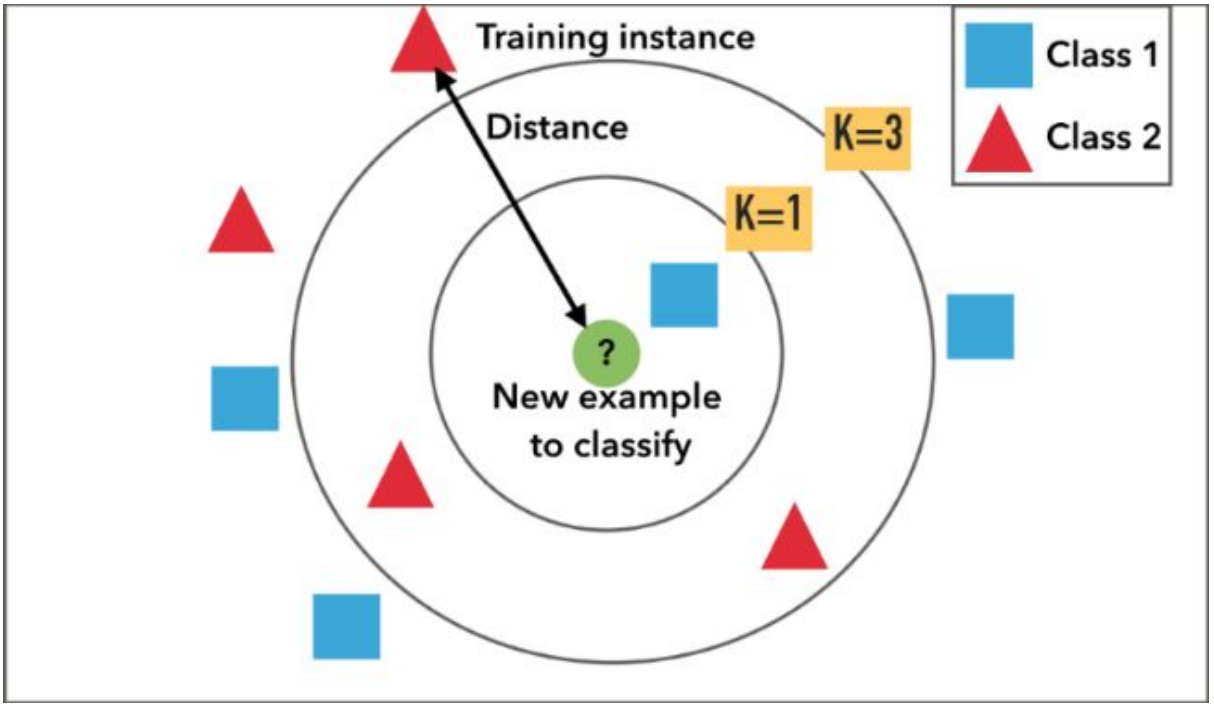
We are using the K-nearest neighbour model to implement our model. In this part of section we are going to see how KNN works and what output we get with it.

>>> K-Nearest Neighbour model:

To implement an item-based filtering, KNN is a perfect go-to model and also a very good baseline for recommender system development. But what is the KNN? KNN is a non-parametric, lazy learning method. It uses a database in which the data points are separated into several clusters to make inference for new samples.

KNN does not make any assumptions on the underlying data distribution but it relies on item feature similarity. When KNN makes inference about a song, KNN will calculate the "distance" between the target song and every other song in its database, then it ranks its distances and returns the top K-nearest neighbor movies as the most similar song

recommendation. In our case, we selected the top 5 most similar songs of a given query song as recommendations.



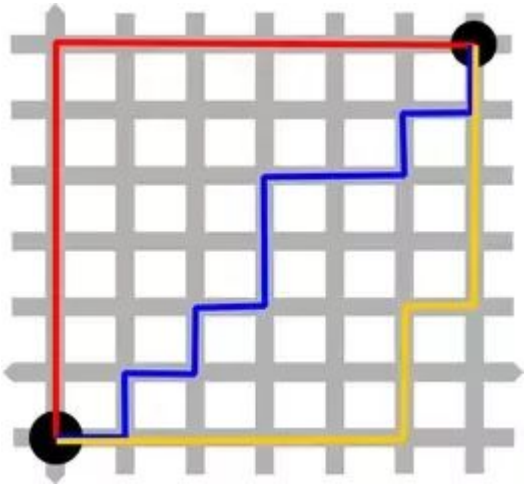
Picture 15, Illustration of how KNN makes classification about new sample, source: [pic:15]

>>> Manhattan evaluation metric:

We will be using Manhattan distance to measure the similarity between our songs. Each song is represented by the set of 13 features described previously. The Manhattan distance between two vectors (or points) a and b is defined as

$$\sum_i = |a_i - b_i|$$

over the dimensions of the vectors. This is known as Manhattan distance because all paths from the bottom left to top right of this idealized city have the same distance:



Picture 16, illustration of manhattan distance, source:[pic:16]

- If the Euclidean distance marks the shortest route, the Manhattan distance marks the longest route, resembling the directions of a taxi moving in a city. (The distance is also known as taxicab or city-block distance.)

- For instance, the Manhattan distance between points (1,2) and (3,3) is abs(3-1) and abs(3-2), which results in 3.

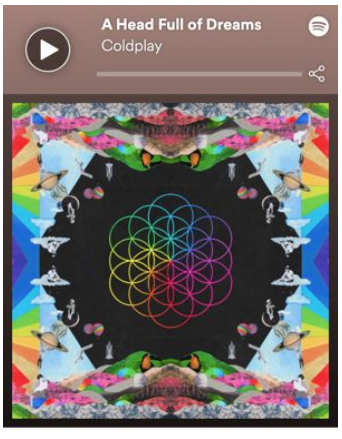
In the cell below you can see the implementation of our recommendation system:

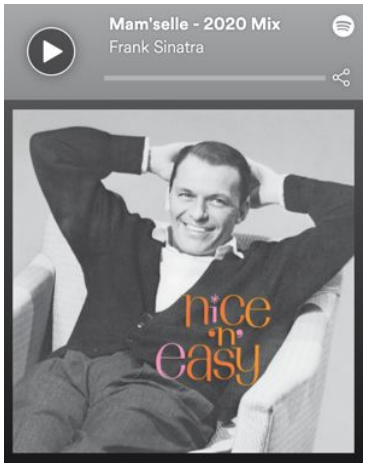
```
# Get song recommendations
# Instantiate knn model
knn = NearestNeighbors(n_neighbors=5,
                       algorithm='auto',
                       metric='manhattan')

# Recommendation function
def get_recommendations(input_df=all_songs_df_std,
                        song_idx=None):

    # Select a wine randomly if none is provided
    if song_idx is None:
        query_index = np.random.choice(input_df.shape[0])

    # Use input song name to get index
    else:
        song_name = input_df.index == song_idx
        query_index = [int(idx) for idx in song_name.nonzero()][0]

    # Fit knn model
    model_knn = knn.fit(input_df.iloc[:,:])

    # Get distances and indexes from the top 5 neighbors
    distance, index_neigh = model_knn.kneighbors(input_df.iloc[query_index,].values.reshape(1,-1),
                                                 n_neighbors=6)

    # Return top 5 recommendations
    for i in range(0, len(distance.flatten())):
        if  i == 0:
            print(f'Top 5 recommendations for: \n\n###{input_df.index[query_index]}###\n')
        else:
            print(f'#{i}: {input_df.index[index_neigh.flatten()[i]]}\n')
```

Cell 25, KNN implementation

>>> Recommendation results:

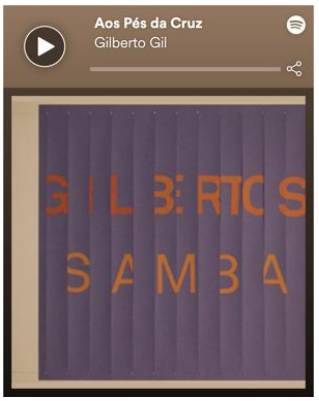
- ❏ Top 5 recommendations for: “Coldplay:A Head Full of Dreams”
  - ❏ Coldplay: Charlie Brown
  - ❏ Bombay Bicycle Club: Evening/Morning
  - ❏ The Fray: Heartbeat
  - ❏ Imagine Dragons: Amsterdam
  - ❏ Coldplay: Hurts Like Heaven





- ❏ Top 5 recommendations for: “Frank Sinatra:Mam'selle”
  - ❏ Frank Sinatra: You forgot all the words
  - ❏ BOY: Into The Wild
  - ❏ Coldplay: Up with the Birds
  - ❏ Frank Sinatra: High Society: You're sensational
  - ❏ Adele: Hometown Glory (Live at Hotel Cafe)

- Top 5 recommendations for: “Gilberto Gil:Aos Pés da Cruz”
  - Rio Bossa Trio: Crazy in Love
  - MAR ABERTO: Dois corpos - Acústico
  - Scubba: Something
  - Maria Gadú: Lounge
  - Elephant Revival: Furthest Shore



With all the recommendations we can conclude that our model results had similarity and diversity at the same time and so far we are happy with the recommendations.

8. Conclusion

The conclusions we drew from our work are as follows:

- For our classification model XGBoost with hyperparameter tuning was the best performance model in terms of F1 score.
- Models' performance depends on the users' likes and dislikes as we saw user1 has a more undistinguishable taste in music while user2 has a more clear separation between liked songs and disliked songs.
- We calculated multiple metrics such as precision, recall and f1 score because the best evaluation depends on the use case. For example in user1 best evaluation is f1 score because it has mixed data so we would like to balance our precision and recall while for user2 we have a clear separation for our given classes so we might give more weight to precision in the evaluation.
- In our recommendation model we have seen the similarity and diversity combination which we consider as a good outcome. The reason is that you do not want your recommendations with very similar songs only, because it might bore you in some time and with the touch of diversity it might be creating a better user experience in the long-term.

9. Future Work

With our classification models and recommendation model we can say that we created a fence for this project. To develop version 2.0 of this project we have few steps in our mind, which are the following ones:

- We would like to add more song likes/not_likes, either manually or crowdfunding.
- We would like to use Feature engineering: create new meta-features combining current features, as we saw correlation between features currently.
- We would like to build a Collaborative model for recommendation:
    - Merge our classification and recommendation models
    - Add song's rankings

We would be developing this project until it satisfies us completely.

10. References

- Picture 1, Spotify logo, Source: [https://www.spotify.com/]
- Picture 2, Global music industry revenue ,Source: [https://core.ac.uk/download/pdf/216833222.pdf]
- Picture 3, music for everyone, Source: [https://open.spotify.com/]
- Picture 5, Spotify Web API Authorization, Source: [https://developer.spotify.com/documentation/general/guides/authorization-guide/]
- Picture 8, clent Credential Flow, Source: [https://developer.spotify.com/documentation/general/guides/authorization-guide/#client-credentials-flow]
- Picture 9, TP,TN,FP,FN representation, Source: [https://towardsdatascience.com/the-5-classification-evaluation-metrics-you-must-know-aa97784ff226]
- Picture 10, Decision Tree, source: [https://www.datacamp.com/]
- Picture 11,random forest classifier,source: [https://m.youtube.com/watch?v=goPiwckWE9M]
- Picture 12, outline of XGBoost Classifer, Source: [https://miro.medium.com/max/922/1*A9myadIB_CqJv-EJA-G_bA.png]
- Picture 13,content based filtering, source: [https://miro.medium.com/max/1000/1*ReuY4yOoqKMatHNJupcM5A@2x.png]
- Picture 14,Illustration of the difference between item-centred and user-centred content based methods, source: [https://miro.medium.com/max/1000/1*DSMBC24ZnJNViN7ownzZAA@2x.png]
- Picture 15,Illustration of how KNN makes classification about new sample, source: [https://miro.medium.com/max/650/1*OyYyr9qY-w8RkaRh2TKo0w.png]
- Picture 16, illustration of manhattan distance, source:[https://qph.fs.quoracdn.net/main-qimg-8d64c8344fc8364e46b9712e2c51dca4.webp]
- How-machine-learning-finds-your-new-music?, source:[https://medium.com/s/story/spotifys-discover-weekly-how-machine-learning-finds-your-new-music-19a41ab76efe]
- Spotify classification, source: [https://www.kaggle.com/geomack/spotifyclassification]
- Behind spotify recommendations engine, source: [https://medium.com/datadriveninvestor/behind-spotify-recommendation-engine-a9b5a27a935]
- Spotify web API: [https://developer.spotify.com/documentation/web-api/]
- Spotify authentication: [https://oauth.net/articles/authentication/]
- Developers dashboard: [https://developer.spotify.com/dashboard]
- Matplotlib: [https://matplotlib.org/]
- Seaborn: [https://seaborn.pydata.org/]
- Pandas: [https://pandas.pydata.org/]
- Audio_features details: [https://developer.spotify.com/web-api/get-audio-features/]

- Principal component analysis: [https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html]
- Mannwhitneyu test: [https://docs.scipy.org/doc/scipy/reference/generated/scipy.stats.mannwhitneyu.html]
- Classification models: [http://scikit-learn.org/stable/auto_examples/classification/plot_classifier_comparison.html]

Appendix 1: For pictures appeared in document:

Index	Abbreviations	Description
1	Picture 1	Spotify logo [section1]
2	Picture 2	Global recorded music industry revenues 1999-2017 [section1]
3	Picture 3	Music for everyone [section3]
4	Picture 4	Methodology for project [section4]
5	Picture 5	Spotify web aPI authorization [section 5.1]
6	Picture 6	Developers dashboard user1 [section 5.1.2]
7	Picture 7	Developers dashboard user2 [section 5.1.2]
8	Picture 8	Client credential flow outline [section 5.1.3]
9	Picture 9	TP, TN, FP, FN representation [section 7]
10	Picture 10	Decision tree illustration [section 7.1.2]
11	Picture 11	Random forest illustration [section 7.1.3]
12	Picture 12	XGBoost illustration [section 7.1.4]
13	Picture 13	Content based filtering [section 7.2]
14	Picture 14	Illustration of the difference between item-centred and user-centred content based methods [section 7.2]
15	Picture 15	Illustration of how KNN makes classification about new sample [section 7.2]
16	Picture 16	Illustration of manhattan distance [section 7.2]

Appendix 2: For tables appeared in document:

Index	Abbreviations	Description
1	Table 1	Authorization flow spotify [section 5.1.3]
2	Table 2	Playlist details for both users on spotify [section 5.2]
3	Table 3	User 1 data shape and size [section 6.1]
4	Table 4	User 2 data shape and size [section 6.1]
5	Table 5	Spotify web API authorization [section 5.1]
6	Table 6	Developers dashboard user1 [section 5.1.2]
7	Table 7	Evaluation for logistic regression [section 7.1.1]
8	Table 8	Evaluation for decision tree [section 7.1.2]
9	Table 9	Evaluation of random forest [section 7.1.3]
10	Table 10	Evaluation of XGBoost with default parameters [section 7.1.4]
11	Table 11	Evaluation of XGBoost with hyperparameter tuning [section 7.1.4]
12	Table 12	user1(Neha)'s models performance comparison [section 7.1]
13	Table 13	user2(Bruna)'s models performance comparison [section 7.1]

Appendix 3: For cells and graphs appeared in document:

index	abbreviation	description
1	Graph [1 to 33]	Graph for features of data
2	Cell [1 to 25]	Coding implementation

Appendix 4: technologies used for project:

index	technologies	description
1	python	Programming language used for implementation
2	Github	Open source available for code repositories
3	Google drive	To share work between partners
4	spotify developers	For application authorization
5	Jupyter notebook	Framework for python