Recommender System
For
Automatic Playlist Continuation

CS 6220: Data Mining Techniques
Summer 2018
Prof. Sara Arunagiri

Team Members:
Abhishek Karan
Devanshi Gariba
Kshitij Mourya
Rutva Rajdev

Abstract:

Main idea is to develop a recommender system for the task of automatic playlist continuation. Given a set of playlist features, recommender system should be able to generate a list of recommended tracks that can be added to that playlist, thereby 'continuing' the playlist. Spotify has released a dataset called "Million Playlist Dataset" which would be used as a dataset to train our model. Accurate prediction and recommendation of tracks to the respective playlists accomplishes the task of automatic playlist continuation.

The main problem here is to identify the various way on how to recommend next song to the user or the playlist. Whether it should be based on type of music or language or genre or artists or popularity is a very big question to be discussed before working on the recommendation system for music platforms. Better the recommended songs, more likely the user to keep using that music service. Since user cannot search for all songs manually, if the music service provider is able to provide the recommendations of the songs that user would highly like, then there are higher chances of the music service staying relevant and becoming successful.

Since the dataset is very large and has lots of properties, there cannot be a single method which would work as a recommendation system for such large dataset and giving satisfactory results. Therefore, we tested many methods to achieve the goal. Two main methods are user ratings system where the user rated various songs and then the recommended songs were provided to them resulting in highly favorable outputs from the model. But since, you cannot always ask a user to rate the songs but rather songs are liked or disliked in any music service. Therefore, the second method to model the system was based on this idea. The songs were then recommended on basis of various properties of the song liked by the user.

The generated results of the model were highly favorable, although the models could be trained better with respect to time and better system architectures. The generated results were in accordance with the user's choices as well as on the statistical viewpoints. More information about the project and the methods used are detailed in the given report.

Introduction:

Music recommender systems (MRS) have recently exploded in popularity thanks to music streaming services like Spotify, Pandora and Apple Music. By some accounts, almost half of all current music consumption is by the way of these services. While recommender systems have been around for quite some time and are very well researched, music recommender systems differ from their more common siblings in some characteristically important ways: the duration of the items is less (3-5 min for a song vs 90 minutes for a movie or months/years for a book or shopping item), the size of the catalog of items is larger (10s of millions of songs), the items are consumed in sequence with multiple items consumed in a session, repeated recommendations have a different significance. Music Recommender Systems then require different approaches from traditional recommender systems.

This project's goal is to provide automatic playlist continuation which would enable any music platform (here Spotify) to seamlessly support their users in creating and expanding the playlists by making recommendations based on their choices and preferences. Furthermore, the recommender system does not require any rich and varied supply of user data, instead requiring only basic information as input such as the title of the playlist, the tracks currently in the playlist, and the artists associated with those tracks.

First, we need to determine the business objectives.

i) Objective of the project is to create a recommender system for automatic playlist continuation.

ii) Given a set of playlist features, recommender systems should be able to generate a list of recommended tracks that can be added to that playlist, thereby 'continuing' the playlist.

Then we need to decide on the plan to approach this dataset to train our model. Processing the dataset and finding correlations between the various tracks provided. Using various classification techniques, generating the next song for the playlist which is based on the preferences of the user

There may be different approaches such as:

Non-Personalized Approaches:

It includes recommending which is popular across the whole system. Simple and relatively easier to implement.

Collaborative Filtering:

Personalized recommendations, based on similarity between either users or items. User-based CF assumes that users who behaved similarly (buying patterns, preference in movies or music, job application) in the past will behave similarly again in the future.

Content Based Filtering:

The actual content of item refers to the actual characteristics of an item and these attributes are broken down into 'tags' and items are then matched to user preferences (as modelled by the tags) accordingly.

Methodology:

This section deals with the methodology followed to accomplish and achieve the goals of the project. Code snippets are shown here for the reference purposes, but the entire code is available in the zip files. The methodology we followed is divided into main four parts: Data Collection, Data Understanding, Data Preprocessing and Data Modelling.

i) Data Collection:

The data provided by Spotify is a Million Playlist Dataset (MPD) which contains 1 Million playlists created by the Spotify users. This dataset is quite huge (about 5.4 GB) and thus it has been divided into 1000 files, where each file contains 1000 playlists comprising to 1,000,000 playlists in total. Due to its humongous nature, the idea of scraping the entire dataset was not viable so sized down to using about 6 json files (limitations of hardware) and also decided to work with only certain important features that are extracted with Spotify Api calls. Spotify structures its data pertaining to each song in four objects: a track object, an artist object, an album object, and an audio features object. They are all relatively self-explanatory in terms of the information stored except audio features - this object stores a wide variety of quantitative data that Spotify assigns songs, like energy, acousticness, etc. So, the general method for data collection was to request all four objects that corresponded with each song in the playlist, referenced via the track URI (a unique identifier for each track). This method of data collection presented several problems in terms of scalability.

1. To get any of these objects requires an authentication token, but a temporary one can be easily requested from Spotify's API. This does, however, require us to manually get new tokens periodically, which significantly limits our timeframe for scraping because we cannot leave a computer to scrape while we are at work, sleeping, etc.
2. Spotify will block tokens that are hitting API endpoints too frequently, so on quick Wifi and computers, the scraping was ironically significantly more difficult, with new tokens required for as little as every 50 songs. We had to work around this by using slower mobile hotspots, which allowed us to avoid this blocking mechanism. This, unfortunately, comes with a side effect of a much lower request speed, albeit for less human maintenance.
3. There were often random stops in the scraping due to request errors; these did not occur with any pattern and we are unsure as to why they did occur.

   Because of the above-mentioned factors, we were able to scrape over 32,000 songs from playlists, which amounts to 19,000 songs after removing duplicates. This is a significant amount of data, but without data for all songs that exist on Spotify, which would be a nearly impossible task, we had to design our models with certain workarounds.

ii) Data Understanding:

As stated in the earlier, all the data have been divided into 1000 files, each of which contains 1000 playlists. These are all json files and in the form of a dictionary. These files follow the naming convention:

mpd.slice.*STARTING_PLAYLIST_ID_-_ENDING_PLAYLIST_ID*.json.

The ids start from 0 and go upto 999999. The json dictionaries have two fields: ‘info’ and ‘playlists’.

```
%%bash
head 'mpd.v1/data/mpd.slice.4000-4999.json'
```

```
{
    "info": {
        "generated_on": "2017-12-03 08:41:42.057563",
        "slice": "4000-4999",
        "version": "v1"
    },
    "playlists": [
        {
            "name": "skate",
            "collaborative": "true",
```

The above image shows the ‘head’ command run on of the json files after importing it in a notebook file. As stated already it is in a dictionary format and contains two fields, or say, keys: ‘info’ and ‘playlists’. The ‘info’ is a dictionary which contains general information about the data in that particular file and contains three keys:

- slice: The range of playlists that are contained within the files. E.g. 0 to 999
- version: The current version of MPD (It is v1 at present)
- generated_on: The time stamp on which the file, or slice was generated

```
%%bash

grep -E '^ {4,16}"' 'mpd.v1/data/mpd.slice.4000-4999.json'
```

```
    "info": {
        "generated_on": "2017-12-03 08:41:42.057563",
        "slice": "4000-4999",
        "version": "v1"
    "playlists": [
            "name": "skate",
            "collaborative": "true",
            "pid": 4000,
            "modified_at": 1432252800,
            "num_tracks": 70,
            "num_albums": 62,
            "num_followers": 1,
            "tracks": [
            "num_edits": 55,
            "duration_ms": 16539160,
            "num_artists": 53
            "name": "Work out",
            "collaborative": "false",
            "pid": 4001,
            "modified_at": 1493510400,
```

The above image shows ‘grep’ command executed on the json file. It shows the lines with 4 to 16 leading spaces, which would help us identify the dictionary structure.

The ‘playlists’ is an array of 1000 playlists. Each playlist is in the form of a dictionary which contains the following keys. It contains a field ‘track’, which in itself is an array of dictionaries.

- pid: It represents the id of playlist. It is an integer between 0 and 999999.
- name Name of the playlist
- description: Description of the playlist, as and if any, provided by the user. It is an optional field.
- modified_at: Timestamp, in seconds since the epoch when the playlist was last modified. The times are rounded to midnight GMT of the date when the playlist was last updated.
- num_artists: The total number of unique artists in the playlist
- num_albums: The total number of unique albums in the playlist
- num_tracks: Number of tracks in the playlist
- num_followers: Number of the followers of the playlist, excluding the creator.
- num_edits: The number of edits. All the changes within a two hour window are considered to be a single edit.
- duration_ms: The total duration, in milliseconds, of all the tracks combined.
- collaborative: It is a boolean field. True means that the playlist was created by just a single user.
- tracks: It is an array of dictionaries and it contains the data about the tracks contained in that particular playlist. It contains the following fields:
    - ❏ track_name: Name of the track
    - ❏ track_uri: Spotify URI of the track. It can be used to make API calls, or for other such developer functions.
    - ❏ album_name: Name of the track's album
    - ❏ album_uri: Spotify URI of the album of the track
    - ❏ artist_name: The name of track's artist
    - ❏ artist_uri: The URI of the artist of the track
    - ❏ duration_ms: The duration, in milliseconds, of the track
    - ❏ pos: The position, starting from 0, of the track in playlist

```
column_names = [col["tracks"] for col in columns]
column_names
```

```
[[{'pos': 0,
   'artist_name': 'Ty Segall',
   'track_uri': 'spotify:track:53yXbISmPFewWiFP2kgal8',
   'artist_uri': 'spotify:artist:58XGUNsRNu3cVOIOYk5chx',
   'track_name': 'Circles',
   'album_uri': 'spotify:album:5bXUYhd6yvmry49a8WpEqS',
   'duration_ms': 186176,
   'album_name': 'Mr. Face'},
  {'pos': 1,
   'artist_name': 'Creedence Clearwater Revival',
   'track_uri': 'spotify:track:47atuTch0AN7NvP9vp42a5',
   'artist_uri': 'spotify:artist:3IYUhFvPQItj6xySrBmZkd',
   'track_name': 'Ramble Tamble',
   'album_uri': 'spotify:album:4GLxEXWI3JiRKp6H7bfTIK',
   'duration_ms': 431226,
   'album_name': "Cosmo's Factory"},
  {'pos': 2,
   'artist_name': 'Fugazi',
   'track_uri': 'spotify:track:7uzeJ1HasqTqqB2anNRFCs',
   'artist_uri': 'spotify:artist:62sC6lUEWRibFqXpMmOk4G',
```

The above command extracts all the tracks' data. As discussed previously, 'tracks' is a dictionary which has fields like pos, artist_name, track_uri, artist_uri, track_name, album_uri, duration_ms and album_name. It always displays a couple of examples of that.

Analysing the data

Until now we have looked at the json file formats and the data at a basic level. We would now format the data to get it into the dataframe and evaluating some of the features.

```
# Reading csv data into the pandas dataframe
import pandas as pd
df = pd.DataFrame(columns)
```

```
# A sneekpeak into the 'playlists' data
df.head()
```

	collaborative	description	duration_ms	modified_at	name	num_albums	num_artists	num_edits	num_followers	num_tracks	pid	tracks
0	true	NaN	16539160	1432252800	skate	62	53	55	1	70	4000	[{'pos': 0, 'artist_name': 'Ty Segall', 'track...
1	false	NaN	3053039	1493510400	Work out	11	10	9	1	12	4001	[{'pos': 0, 'artist_name': 'Eminem', 'track_ur...
2	false	NaN	4419474	1462665600	Study	14	13	9	2	19	4002	[{'pos': 0, 'artist_name': 'Old Dominion', 'tr...
3	false	NaN	23594821	1493424000	HOLA	89	68	22	2	104	4003	[{'pos': 0, 'artist_name': 'Reggaeton Latino',...
4	false	NaN	30240074	1508889600	House	70	47	30	2	79	4004	[{'pos': 0, 'artist_name': 'Ben Watt', 'track_...

The above picture shows the head of the data frame generated from the dataset. As expected, the data frame contains the features such as collaborative, description, duration_ms, etc. as those were the fields of 'playlists' dictionary in the json file.
We would now try to analyse and understand some of these features by generating some statistics, histograms and pie charts, as required and appropriate.

num_albums feature

The 'playlists' contains number of albums as one of the fields which represents the total number of unique albums in the playlist.

```
: # Summary Statistics for 'num_albums'

  print("Summary Statistics for number of albums\n")

  # Using in-built describe method of pandas that calculate a summary of given statistics
  describe = df['num_albums'].describe()
  print(str(describe)+'\n')

  # Using functions 'var' and 'ptp' to calculate variance and range that the 'decsribe' function did not give us
  print('Variance\t' + str(df['num_albums'].var()))
  print('Range\t\t' + str(df['num_albums'].ptp()))
```

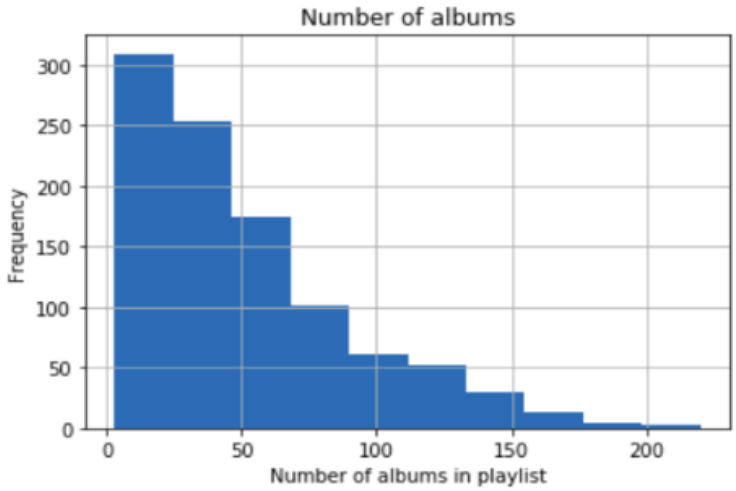
```
Summary Statistics for number of albums

count    1000.000000
mean       51.270000
std        39.539632
min         3.000000
25%        20.000000
50%        41.000000
75%        70.000000
max       220.000000
Name: num_albums, dtype: float64

Variance        1563.3824824824826
Range           217
```

Above are some of the statistics for 'num_albums' feature of the playlist. They were generated using 'describe' built-in functions provided by pandas library. The 'describe' function calculates and generates count, mean, standard deviation, minimum, 25th percentile, 50th percentile, 75th percentile and maximum. The describe function does not calculate variance and range and thus they were calculated individually using 'var' and 'ptp' functions respectively, which are provided by pandas.
The count is 1000, as expected since there are 1000 playlists. The mean is 51.2, which means that each playlist, on average, contains about 51 albums. The standard deviation is 39.5 here. Also, the minimum is 3, which means that all the playlists contain minimum 3 albums and maximum 220. This makes sense as we already saw in section 2 that all the playlists contain minimum 2 unique albums. The 25th percentile, 50th percentile, and 75th percentile values are 20, 41 and 70 respectively. The variance and range values are 1563 and 217 respectively.

```
NAlbumsHist = df['num_albums'].hist()
xlabel = NAlbumsHist.set_xlabel("Number of albums in playlist")
ylabel = NAlbumsHist.set_ylabel("Frequency")
title = NAlbumsHist.set_title("Number of albums")
```



Above is the histogram generated for the number of albums feature. The graph is a kind of decreasing step function, which means that the number of playlists decrease as the corresponding frequency increases. Majority of the playlists contain 1 to 25 albums, while a very few of them contain more than 150 albums.

num_tracks feature

We will now try to understand the num of tracks feature of the playlist. It represents the total number of unique tracks in playlist.

```
# Summary Statistics for 'num_tracks'

print("Summary Statistics for number of tracks\n")

# Using in-built describe method of pandas that calculate a summary of given statistics
describe = df['num_tracks'].describe()
print(str(describe)+'\n')

# Using functions 'var' and 'ptp' to calculate variance and range that the 'decsribe' function did not give us
print('Variance\t' + str(df['num_tracks'].var()))
print('Range\t\t' + str(df['num_tracks'].ptp()))
```

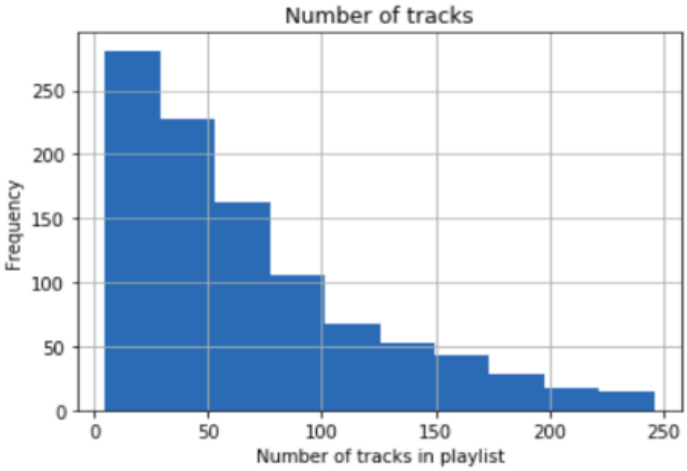
```
Summary Statistics for number of tracks

count    1000.000000
mean       68.101000
std        53.532612
min         5.000000
25%        26.000000
50%        52.000000
75%        95.000000
max       246.000000
Name: num_tracks, dtype: float64

Variance        2865.740539539543
Range           241
```

Above are some of the statistics for number of tracks feature. The count is 1000, as expected since there are 1000 playlists. The mean is 68.1, which means that each playlist contains about 68 tracks on average. The standard deviation is about 53 here. Also, the minimum is 5, which means that all the playlists contain minimum 5 tracks and maximum 246. This makes sense as we already saw in section 2 that all the playlists contain minimum 5 and maximum 250 unique tracks. The 25th percentile, 50th percentile, and 75th percentile values are 26, 52 and 95 respectively. The variance and range values are 2865 and 241 respectively.

```
NTracksHist = df['num_tracks'].hist()
xlabel = NTracksHist.set_xlabel("Number of tracks in playlist")
ylabel = NTracksHist.set_ylabel("Frequency")
title = NTracksHist.set_title("Number of tracks")
```



Above is the histogram generated for number of tracks feature. It is again a decreasing step function and most of the playlists have about 1 to 25 unique tracks in them. Just as the number of albums feature, here too the frequency of playlists decrease as the number of tracks increase. The major difference among the number of tracks and number of albums features is their range, which is 217 for number of albums, while 241 for number of tracks.

The ‘duration_ms’ feature

Each playlist contains the duration in milliseconds feature. It represents the total duration, in milliseconds of all the tracks combined in that playlist.

```
# Summary Statistics for 'duration'

print("Summary Statistics for duration\n")

# Using in-built describe method of pandas that calculate a summary of given statistics
describe = df['duration_ms'].describe()
print(str(describe)+'\n')

# Using functions 'var' and 'ptp' to calculate variance and range that the 'decsribe' function did not give us
print('Variance\t' + str(df['duration_ms'].var()))
print('Range\t\t' + str(df['duration_ms'].ptp()))
```

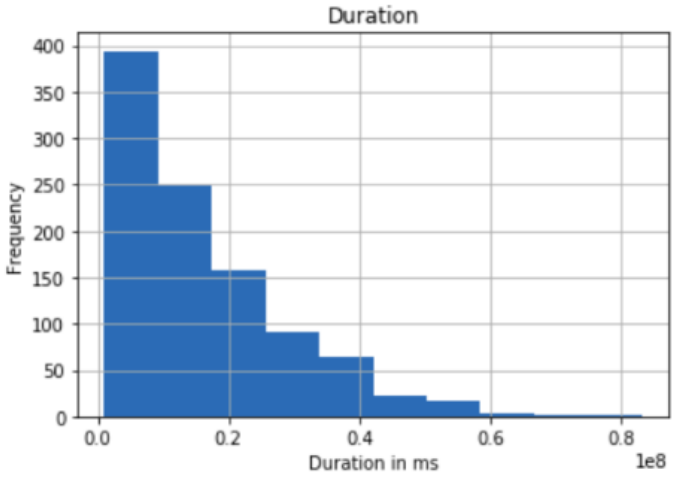
```
Summary Statistics for duration

count    1.000000e+03
mean     1.604941e+07
std      1.280911e+07
min      1.044377e+06
25%      6.118938e+06
50%      1.228655e+07
75%      2.234913e+07
max      8.309966e+07
Name: duration_ms, dtype: float64

Variance        164073286096730.56
Range           82055288
```

The count here is 1000, as there are 1000 playlists. The mean is $1.6*10^7$, which is about 266 minutes and means that each playlist contains about 68 tracks on average. The standard deviation is about 200 minutes here. Also, the minimum is 166 minutes, while the maximum is about 1383 minutes. The variance and range values are 164073286096730 milliseconds and 82055288 milliseconds respectively.

```
Duration = df['duration_ms'].hist()
xlabel = Duration.set_xlabel("Duration in ms")
ylabel = Duration.set_ylabel("Frequency")
title = Duration.set_title("Duration")
```



Here is a histogram for the duration in milliseconds feature. Just like number of albums and number of tracks, it is a decreasing step functions which means that as the duration increases, the associated playlist frequency decreases. Majority of the playlists have about the total duration of 166 to 255 minutes of all the tracks combined.

iii) **Data Pre Processing:**

Dataset needed to be preprocessed before modelling was done.

➔ Some cleaning was done in Microsoft Excel - the following:

a) Deleted rows with headers (because they were all merged, every file that was merged had its own header row)
b) Removed duplicates based on track_id - there were obviously many duplicates because playlists don't all have different songs

A merged CSV file was created which was then processed.

➔ Then Album released date were in different formats and all those formats were converted to YYYY format. How did we know there wouldn't be a problem where e.g. MM/DD/(19)18 and MM/DD/(20)18 would be mistaken as the same? There are no years before 1939 and therefore this problem simply wouldn't arise. Thus consistency to the album released date was achieved.

➔ Then non quant and garbage columns were dropped. There were many columns with N/A values which were also dropped. This was done due to the fact they those columns would not help regarding the modelling and would only hinder the overall process.

➔ All the quantitative columns were standardized for consistent range. There were columns which were having values in decimals and there were some which were having values in multiples of 10 and 100. Thus for proper analysis, these values would have created problems. So they were standardized so that there comparison and collaboration could be done properly.

➔ Correlation Matrix was plotted for the data. With the matrix, analysis was done and many data columns were dropped to reduce redundancy. Dropped feature ‘energy’ and ‘album_popularity’ since they are highly related to ‘loudness’ and ‘track_popularity’ respectively and would just increase redundancy in model.

Based on the correlation plot we are going to drop one of all pairs of predictors that have higher than 0.75 correlation in magnitude. This is to ensure that when we fit a regression model, the weights/coefficients of two nearly collinear predictors aren't arbitrarily distributed, which would ruin interpretability for both those predictors. The pairs that meet that criteria are Loudness and Energy, and Album Popularity and Track Popularity. These high correlations make sense and the both predictors in each pair should probably not go into the same model from a logical standpoint as well. We decide which one to drop based on the one that has the next highest correlation with a different predictor, i.e. we drop energy over loudness because energy has a stronger correlation with acousticness, and we drop album over track popularity because album pop has a stronger correlation with artist popularity.

iv) **Data Modelling**

A single data model design for this humongous spotify dataset is difficult. Primarily, we tested two main approaches to handle the problem and generate recommendations. The first approach is based on a **user rating system**. We provide the 3 different users randomly generated songs (20 in our case) from our database, and in turn, they give back ratings on a scale of 1-10 for the songs (with 10 being the highest). We then trained a model to learn their preferences and predict what ratings they would give for any other song, based on the characteristics of those given songs and how they rated them. Our ultimate goal is to give them back a list of any number of top rated songs from our mode (we chose to gave them their 5 top favourites)

For the second approach, we wanted to address the fact actual music recommendation systems base their recommendations off songs in playlists. When users add particular songs to their playlist, they are assumed to consider those songs "good" and to their taste, so it is quite unorthodox to base a model off songs that users rated poorly as well (as we did in the first model). The issue with this, however, is that user-provided playlists could very easily have songs that were not in our smaller database. Thus we would be unable to actually test this model on real users, but we made it nonetheless.

The models (see IPYNB for much more detail):

Model 1: The beauty of the first model is that we could test it on real users to see if they would like the output songs. This definitely did require user intervention, but we were able to find three people that were willing to take the time to do so. This user feedback did help us a lot. We randomly selected 20 songs (as many songs as we felt we could do without burdening users too much) from our database, and got ratings from 1-10 on a continuous scale (decimals allowed). With so many predictors in comparison to observations we begin falling victim to some degree of the curse of dimensionality, but there was really no way to avoid this due to real life limitations. Each model was fit on those 20 songs, with quantitative attributes as predictors, and the user-provided ratings as the response variable. Treating the response variable as continuous, we were aware we would get predictions that went under 1 and over 10, but that would not impact our ability to take, say, the highest 5 predictions, no matter how high they were. We tried various model types for Model 1 which are simple linear regression, random forest and neural networks for which we got the best results for simple regression model.

Model 2: Rather than training on a dataset with only 20 observations, this model is trained on the entire database. The response variable is a binary classification, with 0 indicating that the song is not in the user-provided playlist, and 1 indicating that it is present. The issue with this model is, that depending on the size of the playlist, the number of classification 1's can be very low in comparison to classification 0's. It was an unbalanced dataset due to which we did not really get desirable results.

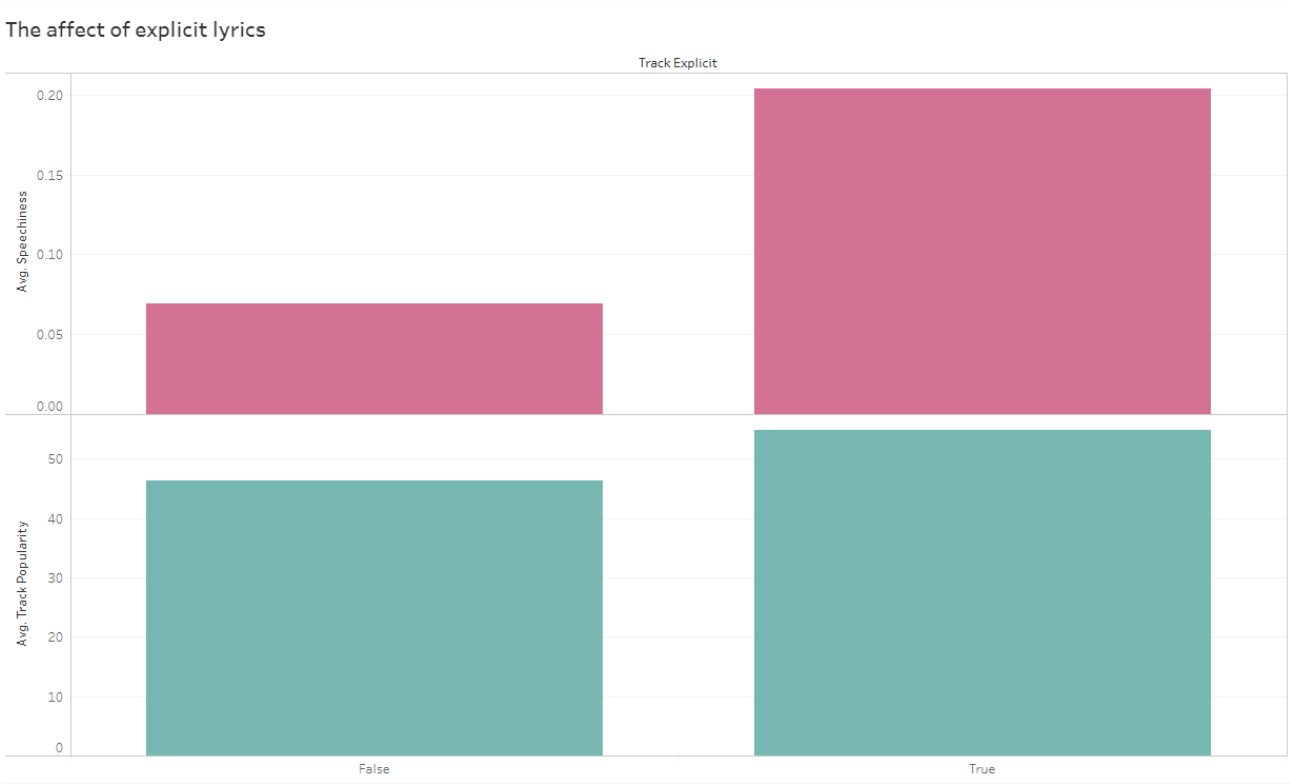
From there we tested a multitude of classification models. Unfortunately, because users could not provide playlists with songs that were not in our database, we could not test

this on user data. We did, however, randomly assign songs to be “in the playlist” as to check our mechanisms.

It is to note that as a result of the extremely low number of observations in Model 1, and low number of classifications in Model 2, we chose not to split our data into a test set for model 1. Rather, (for Model 1) we literally tested the model on the same human users, asking them how they liked to outputted songs.

It is to also note that we chose not to reduce the large number of dimensions relative to observations with PCA. We did eliminate as many predictors as we could through other means, including highly correlated ones, but we wanted to maintain interpretability. Using PCA would prevent us from saying, for example, that a positive coefficient for year means that the user prefers (correlation) more recent songs.

DATA VISUALIZATIONS:

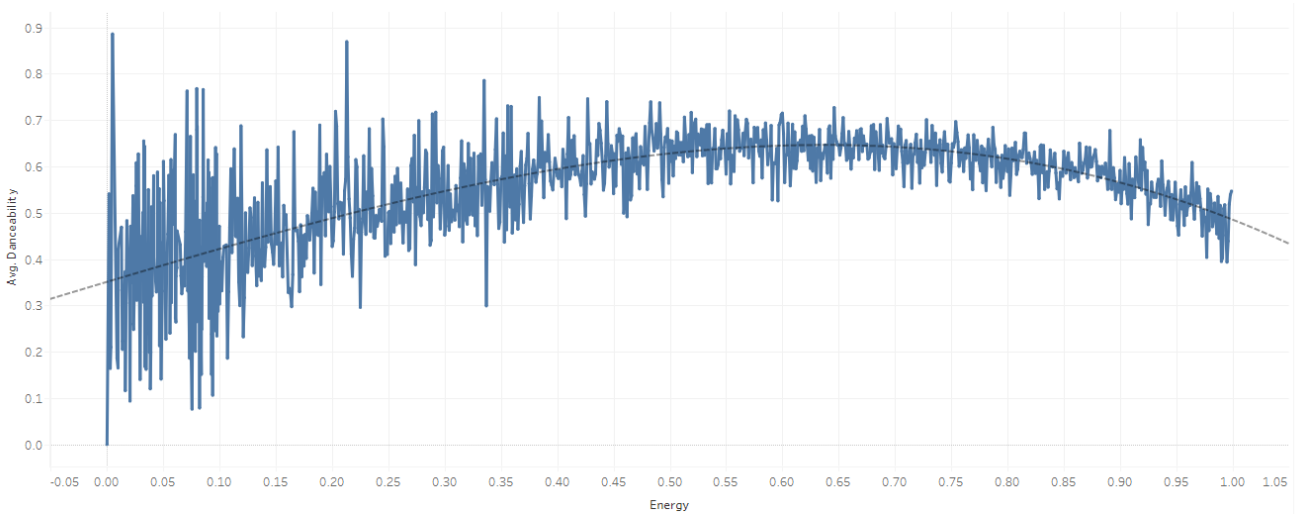


Effects of lyrics on popularity of a track

Interpretation:

- Here, we are plotting the 2 bar graphs comparing track explicit with Average Spechiniess and Average Track Popularity.
- For the Speechiness Vs Explicit, if the track has high speechiness then there are more number of collaborations in the playlist i.e people use these songs and create collaborative playlists.

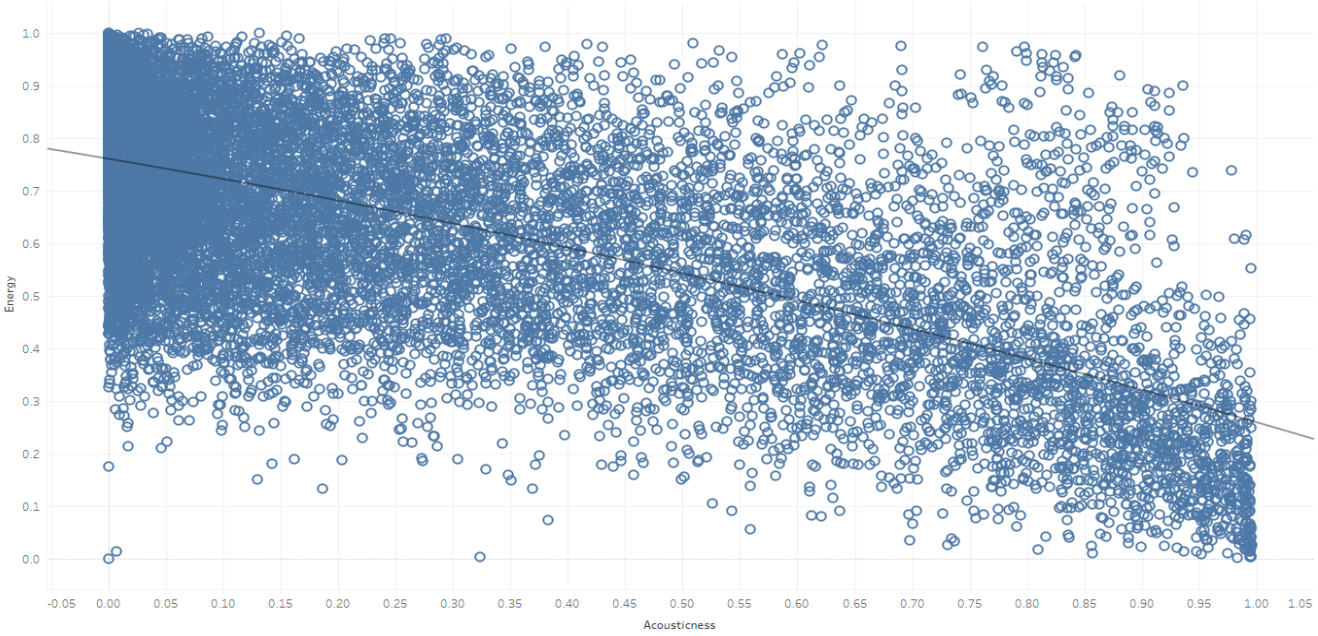
- For the Track Popularity Vs Explicit, there is not much difference between the tracks which are popular or unpopular to be found in a callborative playlists.



Dance Vs Energy Graph

Interpretation:

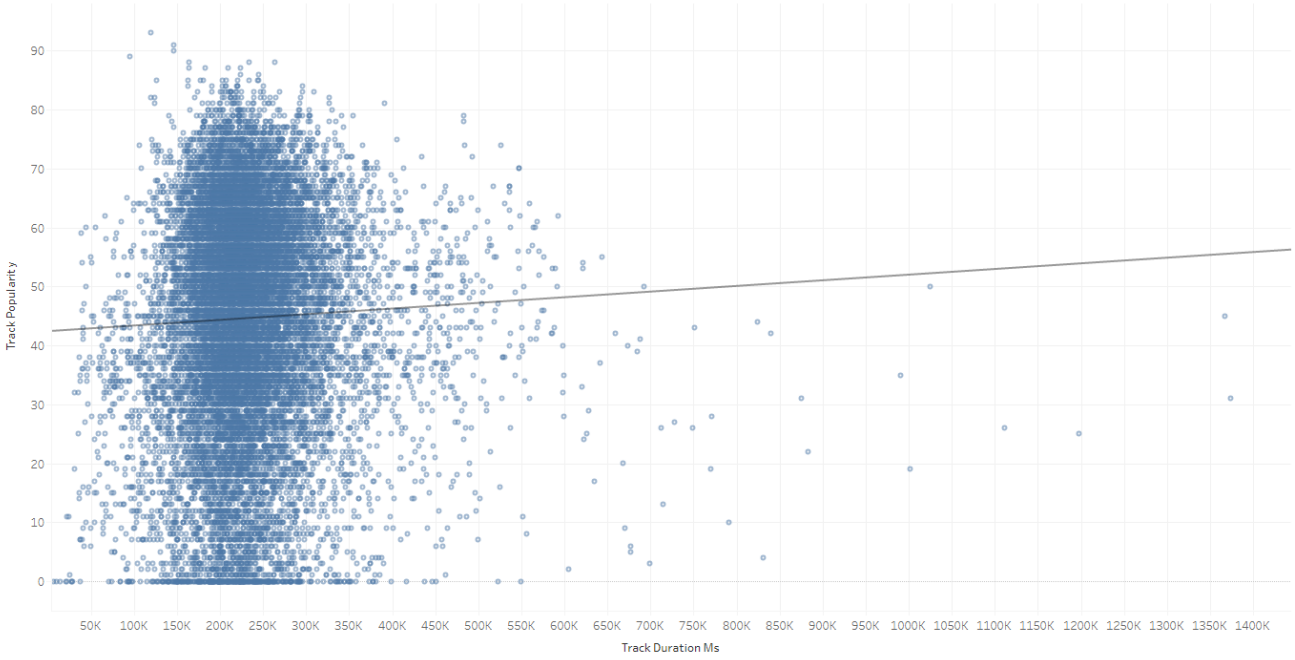
- This is a good plot. It shows how the energy between 0.45 to 0.75 of a track correlates to the dacebility of the track.
- Also, when the energy is too high, there is a sligh decrese in the dancebility factor. So, may be people don't dance a lot if the energy is way high of the track.



Energy vs Acousticness

Interpretation:

- This is an interesting observation. If the energy of the song is high then the acousticness value is low. Probaby because its a slow or a romantic song.
- Also, most of the songs in the dataset are grouped in low acoustic and high energy bracket.



TrackTime vs TrackPopularity

Interpretation:

- Since, there is a positive slope with increase in playlist time, the popularity of tracks also increases.
- Most of the tracks are of a common length in the playlist as they are grouped together near the 150K -250K milliseconds mark.

Results and Discussions:

Model 1: The first model is validated based on human feedback. The fact that the entire model is based on human intervention made it a little easier for us to actually verify

whether our recommender engine works or not. Our recommendation system worked pretty well with this approach. The idea of making a user to rate songs and then training a model based on those song choices generated a list of probable audio features that the user subconsciously ends up liking is practical, viable and implementable. The practical aspect of this approach made it a very suitable one for at least smaller datasets. Cold start problem has been taken care of with this approach. Whenever a user doesn't have a prior playlist with tracks added to it, this approach would work just fine. However, this recommender engine would definitely work better if the user has some initial playlist with 'rated' tracks added to it to make recommendations even more personal and better.
It shows from the notebook that 'Helen' preferred the predicted playlist from the Simple Linear Regression(SLR) model over the other 2 models(Random Forest and Neural Network), and the other two people didn't want intensive analysis, so we skimmed through the dataset and chose which they might like at random and they both chose the Simple Linear Regression as well. It appears that the SLR works best however with such a low sample size and subjective opinions, the results cannot be taken as exactly conclusive but important enough.

Model 2: The second approach is validated based on CV scores. In this approach, we addressed the problem through a classical binary classification model. We appended an extra column in our dataset called "in_playlist" which is binary with 0 meaning the track is not in playlist and 1 means that track is present in the playlist. We ran Logistic Regression, RandomForest, Gradient Boosting, K-NN & created a neural network as well to classify the tracks. We divided the dataset in train/test and did upto 10-fold cross validations on each of the models. We generated the R^2 Error values and compared them. The cross validated scores were used to validate the errors generated. Also, we used cross validation to get the optimal number of neighbors for our K-NN model. Overall after the comparisons, we found that the Random Forest have an accuracy of ~0.99 on the test set which makes it a stand out model. Since, we did the cross validations, the risk of overfitting the model is also reduced. All the scores and measures are documented well in the jupyter notebooks.

Conclusion:

Although the idea of recommendations work best with e-commerce sites as that enhances purchases and does not bring direct revenue increase to such firms that provides music streaming service, it has about the same indirect tangible effects on user engagement and user sustainability. Implementing and testing a better recommendation engine means the ability to personally interact with every user. Higher the accuracy, higher is the engagement and more is the revenue. Of course, there are a lot of other firms that provide the exact same services as Spotify, but the idea here is to make Spotify a little better as compared to others on the song discovery front. It makes a lot of sense to make song discovery a very natural process to the users utilizing the services. Although there is no right or wrong way to build a recommendation engine because of the enormous data set, lack of appropriate system architecture, lack of enough time in iterating and training the models, our recommendation engine did pretty well in

predicting the tracks under different conditions. These two approaches answered different questions. For the first approach, simple regression model did well and for the second approach, random forest did pretty well. All the figures for all other models ipynb files.

References:

[1] Spotify Challenge: https://recsys-challenge.spotify.com/readme
[2] Yiwen Ding,Chang Liu. Exploring drawbacks in music recommendation systems. Bachelor Thesis in Informatics, 2015.
[3] Yajie Hu. A model-based music recommendation system for individual users and implicit user groups. Dissertation, 2014.
[4] Spotify developer echo next API: https://developer.spotify.com/spotifyecho-nest-api/.
[5] Quartz Spotify: https://qz.com/571007/the-magic-thatmakes-spotifys-discover-weekly-playlists-sodamn-good/
[6] Yifan Hu, Yehuda Koren, Chris Volinsky. Collaborative Filtering for implicit feedback datasets. 2009.
[7] http://benanne.github.io/2014/08/05/spotify-cnns.html
[8] https://papers.nips.cc/paper/5004-deep-content-based-music-recommendation