**Overview**

* We have studied the use of content- based approaches to form playlists from a given seed song. Our major approach would be to follow audio similarity measure. This measure compares songs according to the novelty of their frequency spectrum and the below mentioned audio parameters. In this approach, we investigate extensions to simply choosing the N closest songs to a seed as a playlist. The playlist will be keep on updating as the user will like/ dislike the currently played song in the mobile application and playlist will be formed by trajectories through the distance space using user relevance feedback.

**Progress**

* We are able to implement the backend algorithm which gets the data input in form of JSON which contains the attributes such as Echonest ID and like/ dislike flag. The algorithm works on the following heuristic: Candidate songs are all songs in the collection which have not been played (or skipped) yet. For each candidate song, let da be the distance to the nearest accepted, and let ds be the distance to the nearest skipped. If da < ds, then add the candidate to the set S. From S play the song with smallest da. If S is empty, then play the candidate song which has the best (i.e. the lowest) da/ds ratio.
* We are able to implement the system architecture in which we have shown how data is getting exchanged between the backend, server and Mobile application. The system architecture has been shown below:

**Echonest Million song data set**

**Network Algorithm**

**Flask Server**

**Mobile Application**

**Spotify Web Server**

**Echonest Million Song Data set:** Contains the audio features of million songs and returns the audio features to the Network Algorithm function with song UID so that our algorithm can process the audio features whenever an API call is made, to perform comparisons for recommendations.

**Network Algorithm Function:**  This is the main algorithm which we have developed, which retrieves the URI for every song, the user has liked/ disliked from the flask server with mobile application as the GUI. It stores liked/ disliked songs in an adjacency matrix and also compares the distance between candidate song and like/ dislike songs and then return the UID and name of the recommended songs to the flaskserver.

**Flask Server:**  It gets the user input in the form of JSON, which contains the UID and name of the liked and disliked songs and forwards the same to the backend function/ algorithm. In addition to this, it passes the JSON file to the User Interface in the form of JSON which contains the attributes of recommended songs only and then those songs get added to the playlist currently being played.

**Android Mobile Application:** It has an interface like a music player, has all the functionalities like play/ pause/ next/ playlist generation. There is also an option for like/ dislike song, which sends the UID and song name in JSON format to the server. Whenever the song is liked/ disliked, playlist keeps on updating and it streams the song from the Spotify web server by using Spotify API calls and access songs/ albums/ playlist from Spotify server using track UID.

**Algorithm - Working**

* We imagine a graph of all songs in our database. Each song is a node and links between songs describe how closely the songs are related. The simplest graph uses our distance measure for the link strength. A playlist can be formed using this graph by choosing the shortest path of length N emanating from the seed song. However, this prior work assumes that relevant attributes have already been determined for each song rather than extracting them from the audio as in our case.
* The million songs 2.3GB data set from Echonest, contains all the audio features. We will be using libraries such as hdf5, numexpr, cython, blosc, tables and numy. Our data set is in HDF5 format, we have to use h5py python library to process this data. The h5py package is a Python interface to the HDF5 binary data format. It lets you store huge amounts of numerical data, and easily manipulate that data from NumPy. For example, you can slice into multi-terabyte datasets stored on disk, as if they were real NumPy arrays. Thousands of datasets can be stored in a single file, categorized and tagged however you want.
* MFCC ( Mel-frequency cepstrum coefficient) of our data has short spectrum information and there are lot of segments in the data which contains timbre information, so we will consider timbre as the major comparison feature to match songs and we can also use other audio features like tempo, pitch and loudness as well. The following features will be taken into consideration for comparison purpose and to find the minimum distance to represent the closely matched songs in graph representation:
* ‣ **key**: the estimated overall key of a track. The key identifies the tonic triad, the chord, major or minor, which represents the final point of rest of a piece.
* **tempo**: 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.
* **loudness**: the overall loudness of a track in decibels (dB). Loudness values in the Analyzer are averaged across an entire track and are useful for comparing relative loudness of segments and tracks. Loudness is the quality of a sound that is the primary psychological correlate of physical strength (amplitude).
* **timbre** is the quality of a musical note or sound that distinguishes different types of musical instruments, or voices. The Echo Nest Analyzer’s timbre feature is a vector that includes 12 unbounded values roughly centered around 0. Those values are high level abstractions of the spectral surface, ordered by degree of importance. The actual timbre of the segment is best described as a linear combination of these 12 basis functions weighted by the coefficient values: timbre = c1 x b1 + c2 x b2 + ... + c12 x b12, where c1 to c12 represent the 12 coefficients and b1 to b12 the 12 basis functions as displayed below. Timbre vectors are best used in comparison with each other.

**Fallouts**

Not using social tags.

No web application as promised

No Ios app for the users.

In addition to this, we are going to access our data set locally, as it will be quite expensive to host 2.3GB of data on any server.

Current

we present and evaluate heuristics

to adapt playlists automatically given a song to start with

(seed song) and immediate user feedback.

Instead of rich metadata we use audio-based similarity.

The user gives feedback by pressing a skip button

if the user dislikes the current song. Songs similar to

skipped songs are removed, while songs similar to accepted

ones are added to the playlist. We evaluate the

heuristics with hypothetical use cases. For each use case

we assume a specific user behavior (e.g. the user always

skips songs by a particular artist). Our results show that

using audio similarity and simple heuristics it is possible

to drastically reduce the number of necessary skips.