Spotify Playlist Reordering Website

Leo Wang
Department of Computer Science
University of Virginia
Charlottesville, Virginia
yw7uc@virginia.edu

Yifan Wang
Department of Computer Science
University of Virginia
Charlottesville, Virginia
yw5ma@virginia.edu

Isaac Li
Department of Computer Science
University of Virginia
Charlottesville, Virginia
il5fq@virginia.edu

ABSTRACT

With over 200 million monthly active users, Spotify is one of the best streaming music service nowadays. However, Spotify lacks the functionality of rearranging users' playlists based on users' needs. In this paper we first introduces several music datasets we found that are suitable for related studies, then we design different approaches that help users rearrange their playlists. Finally we host the approaches we designed on a website and allow Spotify users to login and use.

1 INTRODUCTION

Music playlist rearrangement has gained importance over the past decades accompanying the increasing popularity of online music streaming app and platform such as ITunes, Spotify, Google play etc. There already exist a few playlist organizing websites or API. Music playlists reordering are more and more personalized based on listeners' preferences. Several properties are commonly utilized in slicing and croping the playlist, including genre, year, BPM (Beats per Minute), energy, loudness etc. The problem our project intends to fix is to create web application integrating spotify API and machine learning models to rearrange existing user playlist. The current algorithms and web applications put more emphasis on track recommendation utilizing collaborative filter, making cross-references between users with similar interests and tastes, or content-based filters. Researches have been conducted attempting to make recommendations according to the audio signals of the track that the users have previously listened to. [12] [11]

We will explore different machine learning models fitting the playlist. We will then rearrange and generate the list according to the models and make comparisons between the results in order to find an optimal approach and host it on a website. The project is beneficial in several ways, as not only it works on large data sets to train models, but also the results of the projects are deployed into daily-life use. Playlist reordering can be immensely convenient for music listeners. Compared to music recommendation, it has more down to ground exploration over the existing tracks and focuses more on personalizing and feature digging. After applying on platform like Spotify with a large number of user basis, the playlist rearrangement system can be further studied and improved with user feedback and experience.

2 BACKGROUND

Since our project aims to help users reorder their playlists, we need to find a music streaming service that has a decent amount of users and offers supportive developer APIs. After doing some research, we found Spotify and Apple Music to be our best options. Although Apple Music has 60 million songs, which is 10 million songs more than Spotify, Apple Music only had 72 million subscribers for Q1 2020, while Spotify had 130 million paid subscribers, not to mention those users who use the free tier. In addition to that, Spotify also has better Developer APIs. After Spotify acquiring The Echo Nest in 2014, which is a music intelligence and data platform for developers and media companies, they migrated Echo Nest APIs including the one that provide audio features which is essential for our research. Therefore, we choose Spotify because of its huge user base and its well-developed Developer APIs.

3 RELATED WORK

There are a lot of existing work that does music recommendation and playlist recommendation, while there are few that does playlist reordering. Previous works mainly rearrange tracks in playlists by mapping tracks in a playlist into a graph, treating them as vertices in the graph, with some self-defined similarity as distances between them and modeling the problem as finding a Hamiltonian path in the graph. Bittner *et al.* [4] follows this pattern by first constructing the feature space for a track by including several acoustic aspects, including acoustic vectors, key, mode and tempo. Then they measure distances using the Euclidean distance between track features, and find the Hamiltonian path with some approximations, such as the greedy approximation which iteratively finds the closest non-visited vertex starting from a seed vertex. The authors of [8] does similar work, while they also introduce the divergence ratio from the start and end song of a playlist to determine the position of a track in a playlist. Despite rearranging lists based on similarities, diversity and dissimilarities between items are also used to rearrange lists [3, 5]. In addition to audio features, several works also emphasize the importance of user feedback. Logan [10] used the graph approach to find song trajectories by measuring similarities based content features of tracks, and the author also uses relevance feedback to improve the performance of the system. Jannach *et al.* [9] optimize a sequence of songs by minimizing the difference between the characteristics of a user's playlist history and the characteristics of the sequence of songs.

These approaches generally use small datasets, with most of them only covering a small range of tracks, and there is also not a existing tool that utilizes these approaches to rearrange Spotify playlists. Therefore, to analyze and propose solutions to the playlists rearranging problem, we explore existing music datasets that are big and compatible with Spotify, develop algorithms and models based on them and host our different music rearranging approaches on a website for Spotify users to use.

4 SYSTEM DESIGN

This section is divided into five parts, Spotify APIs, audio features, datasets, approaches we tried, and the website. First we explore Spotify APIs to find what data we can fetch from Spotify in order to decide how we are going to design our approaches. Then we find datasets that are suitable for playlists rearranging problem. After that, we develop different approaches to rearrange playlists. Finally, we design a website that allows user to login through Spotify, and rearrange their playlists based on our proposed approaches.

4.1 Spotify APIs

The Spotify APIs we used can basically be divided into three parts: authentication, personal user data, and track data.

4.1.1 Autorization. The basic Spotify authorization process is handled by the python package python-social-auth-django. Although we have to request the following API to refresh user's access token, since access token expires after a short time.

- https://accounts.spotify.com/api/token: A new access token will be granted for a valid refresh token.

4.1.2 User Data. This part includes the core APIs we need to call in order to get a user's recent play history, playlists, etc.

- https://api.spotify.com/v1/me/player/recently-played: Get current user's recently played tracks. This API can at most request 50 recently played tracks, and a track needs to be played for at least 30 seconds to be included in play history. This feature is quite important for our assumption that the tracks in user's play history are user's recently favorite songs.
- https://api.spotify.com/v1/me/playlists: Get a list of current user's playlists.

4.1.3 Track Data. This part includes the APIs we need to get specific information about tracks and playlists.

- https://api.spotify.com/v1/playlists/`{playlist_id}`: Get specific information about a playlist, including the tracks in the playlist.
- https://api.spotify.com/v1/audio-features/`{id}`: Get audio features of a single track. This and the following API are the main APIs we use, and the audio features fetched from this API are the main features we use for the tracks.
- https://api.spotify.com/v1/audio-features: Get audio features of several tracks.

4.2 Audio Features

In this part we introduce the audio features we can get from Spotify APIs.

- Acousticness: A confidence measure of whether the track is acoustic ranging from 0.0 to 1.0. 1.0 represents the track is highly acoustic.
- Danceablility: A combination of musical elements such as tempo, rhythm stability, beat strength and overall regularity to measure how suitable the track is for dancing . It ranges from 0.0 to 1.0, and 1.0 is the most danceable.
- Energy: This measurement ranges from 0.0 to 1.0, lowest to highest, representing a perceptual measure of intensity and activity.
- Instrumentalness: Ranging from 0.0 and 1.0, this feature predicts whether the track contains vocals. The closer to instrumentals value is 1.0 and any value above 0.5 represent instrumental tracks. This audio feature is not considered in our API because most of our testing tracks are songs containing vocal therefore the feature would make no impact in prediction.
- Liveness: Ranging from 0.0 and 1.0, liveness detects the presence of an audience in the recording. Higher value represents an increased probability that the track is live.
- Loudness: The overall loudness in decibels(dB)
- Speechiness: A measurement ranging from 0.0 to 1.0 that detects the presence of spoken words in a track. Not selected as a feature in our model because of higher values are related to talk shows, audio books or poetry, which are irrelevant to music, the main focus of our project.
- Valence: Ranging from 0.0 and 1.0, valence measures the musical positiviness conveyed by the track. 1.0 represent high positive, such as happiness, cheerfulness etc.
- Tempo: The overall estimate temp of a track in beats per minute (BPM).

4.3 Datasets

There are several datasets we found that are suitable for our study, while only some of them are easy to use without mfurther preprocessing.

4.3.1 Spotify Dataset 1921-2020. A kaggle dataset that contains more than 160,000 songs collected from spotify API. The data is stored in a csv file and the tracks are grouped by artistic, year or genre. The primary id is the entries are the ids the tracks, and the numerical statistics of the entries are audio features mentioned in section 4.2. [1]

4.3.2 The music streaming sessions Dataset. The dataset are divided into two table, the session table contains all the sessions with session id as primary key. Each session contains 10-20 tracks, each with unique track id. skip1, skip2, skip3 and notskipped are boolean values indicating how long the track is played before skipped. Several other user behavior features are also included in the table. The second table contains all the tracks as entries, primary key is the track ids utilized in the first table, and the spotify audio features of the tracks are also included in this table.[6] This dataset has roughly 130 million listening sessions, while we only use a mini subset which has 10000 sessions to train our models due to limited resources and time. One downside of this dataset is that all the track ids are anonymized, therefore we don't know the real names of the tracks.

4.3.3 Spotify million playlist dataset. This dataset has 1 million playlists consist of over 2 million unique tracks by nearly 300,000 artists [7]. Comparing to the music streaming sessions dataset, the track ids in this dataset are real Spotify Track IDs, therefore tasks such as training track embeddings and playlists embeddings can be performed on this dataset. The original task of this dataset is to generate a list of recommended tracks that can be added to a playlist given a set of playlist features, including some existing seed tracks in the playlist. We tried to train track embeddings similar to how word2vec is trained, and train playlist embeddings using autoencoder, but we didn't find good ways to prove that our embeddings make sense, and therefore we don't include them in the report.

4.4 Playlists Rearranging Approaches

The goal of our website is to provide users with approaches to rearrange their playlists, while it is hard to know how a user wants to order a playlist. Therefore we develop different approaches to accommodate users' needs. We start with basic approaches like sorting playlists by name to provide basic functionalities. Then we design approaches that utilizes track features provided by Spotify or the datasets we found. After that, we train some traditional machine learning models on those big datasets mentioned in the previous section with some relating tasks, and use these models to rearrange users' playlists. Finally, we try to tackle the problem with neural networks.

4.4.1 Basic Approaches. To provide users with basic functionalities to rearrange a playlist, we select several features from the audio feature API, including danceability, energy, acousticness, speechiness, instrumentalness, liveness, valence and tempo, and also track name for users to sort the playlists by. In addition to basic sort, we also provide the naive cosine similarity measure. We first select the eight features mentioned above, then we retrieve user's recently played 50 tracks and calculate the arithmetic mean of the 50 tracks to try to encode a user's recent preferences. Finally we compute the cosine similarity between each track in user's playlist and the arithmetic mean of recently played tracks and sort playlists by the computed cosine similarities.

4.4.2 K-means. In this section, the Spotify Dataset 1921-2020 is used to train a K-means clustering algorithm, which is later used to rearrange the tracks based on how many times their classification groups has shown up in the playlist. Before the actual training steps, the dataset is preprocessed dropping irrelevant features and scaling into an appropriate scale using the StandardScaler. A K-means model is then initiated to classify all the dataset into a certain number of clusters. To apply the results to the playlist rearranging tasks, the cluster information would be stored into a separate file as the reference for track scoring. Next, in order to score the tracks, we would generate scores for every cluster. Every cluster initially has a score of 0. When a playlist is given as an input, a program goes over every track in it, refers to the previous training results and finds the cluster it belongs to, and increments the score of that cluster by 1. After going over the playlist we end up with different scores assigned to each cluster. Since these scores are also one-to-one mapped to the tracks, we are eventually able to rearrange the tracks in the playlist by the descending order of the cluster scores.

4.4.3 Random Forest Classifier. The music streaming sessions data set is used in this section. Before conducting actual random forest regressor's fit and prediction over the dataset. Prepossessing is required. The two tables of the dataset are natural joined over the track ids. It is inferred that the sole prediction of skip2 is competent in competent.[2] After dropping several irrelevant values and making several combinations, we create a new table that has entries containing the session id, the id of the 10th track within the session, and selected feature (skip, acousticness, danceability, energy, liveness, speechiness, tempo and valence) of the first 10 tracks of the session. Each feature has a suffix number indicating which track it belongs to, and the features of the 10th track has no prefix. For instance, skip9 stands for whether the 9th track of the session is skipped, and acousticness5 stands for the acousticness of the 5th track in the session. In our project we conduct random forest regression over 1000 sessions, and the test size is 0.1, in other word the train sample size is 900 leaving 100 sessions for testing purpose.

4.4.4 Deep Learning Models. We use the music streaming sessions dataset to train our deep learning models since they generally require large datasets. The task is to predict if the next song is skipped, given the previous $n - 1$ songs in a session. Although the dataset provide a wide range of features for a music playing session, some of the interesting ones including a boolean value indicating if the user encountered this track while shuffle mode was activated and the user action which led to the current track being played, the Spotify APIs that get audio features and retrieve user play history don't provide these features. Therefore, we only choose danceability, energy, acousticness, speechiness, liveness, and valence for each track played in a session. Since the input data is a sequence of songs a user played in a session, recurrent neural networks is a great choice to deal with sequential data. Therefore, we first try the gated recurrent unit (GRU) RNN, then we try long short-term memory (LSTM) RNN with attention.

For the input of the model, assuming batch size is B, the session length is N and the number of features is 6, the input consists of the features of the previous $N - 1$ tracks, which is of size $(N - 1, B, 6)$, and the features of the final track, which is of size $(1, B, 6)$. The output is the $skip\_2$ feature of the final track, indicating if the track was only played briefly. Originally, we separate the input because we assume we have access to the $skip$ features provided in the dataset, but we later found that we can't retrieve this data from Spotify's recently-played-tracks API.

The six features we choose for each track don't have enough dimensions, therefore for both models we first use a fully connected (FC) layer to transform the input track features into higher dimensions, which we choose to be 300. We also add an activation function for this FC layer to add non-linearity. Both the input of previous $N - 1$ tracks and the last track are fed into the same FC layer to get higher dimension representations.

For the GRU model, we choose a hidden size of 300, number of layers to be 1, and bidirectional to $True$. The sequential input of previous tracks and the last track are fed together into the GRU model as one sequence, and GRU outputs the output features of each track and the hidden states of the final track. Since this is a bidirectional GRU, we concatenate the hidden states of both directions, and feed that into a FC layer that transforms this into the size of the number of classes to be predicted. Finally, we use Cross Entropy Loss as the loss function, and Adam as the optimizer with a learning rate of 0.0001.

For the LSTM model with attention, we use a similar setting, except that only the previous $N - 1$ tracks are fed into the LSTM model to produce the output states of each track and the final hidden state. Then we compute attention based on the hidden state and the output states, and produce a context vector based on attention and the output states. Finally, we concatenate the embedding of the last track with this context vector, and feed this into a FC layer that transforms this to the output dimension.

To use this trained model on our website, we retrieve the user's recently-played 19 tracks and all the tracks in the specific playlist the user is willing to rearrange with the corresponding audio features. Then for each track in the playlist, we feed its features and the recently-played tracks' features into the model, and apply a softmax function to the output of the model. Finally, we use the not-skip-class's re-scaled value which is between 0 to 1 to determine how likely a user will skip the track, and sort the tracks from high to low based on this to rearrange the playlist.

4.5 Website

We choose Django as our backend, use bootstrap4 as the UI framework, and deploy the website on Heroku. The project uses sqlite as the database locally, and uses Postgresql on Heroku. To use the website, users must login first through Spotify. We provide this feature by using the python package Python-social-auth for Django. First, we login to Spotify's developer dashboard to create a project, then we set the client id and client secret from this project in Django, and add the following authorization scopes to ensure that only limited access to a user's private and public information are given to our website.

- user-read-private: Gives access to user's subscription details
- user-library-read: Gives read access to a user's "Your Music" library
- user-read-recently-played: Gives read access to current user's recently played tracks
- playlist-read-private: Gives read access to current user's private playlists
- playlist-read-collaborative: Includes collaborative playlists when requesting a user's playlists
- user-read-currently-playing: Gives read access to a user's currently playing content
- user-modify-playback-sate: Allows access to APIs that modifies user's current playback state like pausing and skipping to next or previous track.
- playlist-modify-private: Gives write access to a user's private playlists. This includes adding, removing items from a playlist, creating a playlist and reorder a playlist's items
- playlist-modify-private: Gives write access to a user's public playlists.
- user-library-modify: Gives write/delete access to a user's "Your Music" library.

Once a user logs in, a Django User object will be constructed with Spotify User ID as username and Spotify email as email. A corresponding user social auth object will also be constructed, which has a one-to-one relationship with the user object, and has three other fields: provider, Uid, and extra data. Provider is Spotify in this case, Uid is user's Spotify User ID, and extra data stores the auth time, refresh token which will be used to get new access tokens, access token, which is used when calling Spotify Developer APIs, and the token type, which is Bearer.

All requests are handled with a function that sends get or post request to a specific url with data, headers and parameters. It will automatically fetch current user's access token from the database, add them in the request and send the request. This function makes sure that if the response from Spotify shows error, like invalid access token, it will automatically include user's refresh token in the data field and send a POST request to Spotify's endpoint that grants new access tokens. Once the new access token is received, it will be saved to the extra data field of current user's corresponding social auth object, and the original request will be sent again.

A user will be redirected to the home page after logging in, and a get request will be sent to get user's playlists. Then users can choose which playlist to rearrange, and go to the specific page for a playlist. On the specific playlist page, all tracks will be retrieved with the corresponding features and artists. The user can select how to sort the playlist, and after it is selected, user will be redirected to this playlist page with the sort by URL parameter equals to the selected sort-by method. If the user is satisfied with this new playlist, user can click on the "Reorder and Create New Playlist" button, and a post request will be first send to a Spotify's endpoint that creates a new playlist, with the name equals to the original playlist name concatenating with current time, and the description indicating that this is a reordered playlist created at current time. Once a success response is received from this request, its playlist id will used in the next request to add the reordered tracks to this new playlist. Finally, the user will be redirected to the home page, and the user should see the newly created playlist. We choose not to use the Spotify's reorder playlist endpoint because we want to make sure that we don't modify a user's existing playlists, so that users can always rename or delete the rearranged playlists if they like or dislike them.

5 PROCEDURE

Our website url is https://reorderplaylist.herokuapp.com. It's deployed on Heroku and since we use a free dyno, it will sleep if it receives no traffic in a 30-minute period, therefore it might take some time to first load the page most of the time. To use our website, the user must have a Spotify account, while it doesn't matter whether the account is free or premium. The basic flow of using this website is described in the above section. The core function of our website is on the playlist page, which retrieves all the tracks of a playlist and allows the user to choose which playlist rearranging approach to use to sort the playlist. All the approaches described in 4.4 are implemented and deployed.

6 RESULTS

In this section we report the result and performance of both the machine learning models and the user feedbacks on the website. For each model described in the section 4.4 we conduct implementation and evaluation in order to visualize the accuracy in predicting user's preference and test their performance in playlist reordering. The performance of website is reflected by the comments and suggestions raised by the users invited as the tool testers.

6.1 K-means

To implement the K-means clustering algorithm on the dataset containing more than 160,000 songs, we decide to include both K-means and Mini-Batch K-means model into our performance analysis and pick the better one for the actual implementation. The goal is to find out the best number of clusters to optimize the model's performance, which is evaluated in two aspects: the sum of squared distances of samples to their closest cluster center (SSE) and the runtime of model training process. We first test both models with the cluster number from 1 to 20. Both models show decreasing SSE values as the cluster number increased, but the runtime of the K-means model goes beyond 10 seconds while the runtime of the Mini-Batch K-means model is still less than 0.3 seconds. In regards of the time efforts, we continue the analysis by incrementing the cluster number on only the Mini-Batch K-means. At the end we reach the cluster number of 100, where the model has the SSE decreased from the initial value around 1,300,000 to the lowest at about 200,000 and reaches the acceptable runtime is around 4.5 seconds. Given this cluster number, on average each cluster should contain about 1,600 songs. We believe this is a moderate size for clusters and further incrementing cluster number may cause overfitting issues. We therefore decide to accept this Mini-Batch K-means model with the cluster number of 100 for the actual implementation of playlist rearranging tool.

6.2 Random Forest Classifier

The 1000 sessions are fitted and predicted using the RandomForestClassifier provided in scikit-learn with a test size split of 0.1, 900 samples in the train set and 100 samples in the validation set. Starting with 100 estimators, the accuracy score is 0.76, as the number of estimators increases from 100 to 500 and eventually to 1000, the accuracy steadies at 0.8. Feature importance of the learned model is plotted as a bar graph to visualize the importance of each feature in predicting whether the current track will be skipped or not. According to Figure.1, whether the most recent track before the current song is skipped or not has a decisive importance of 0.1, while other features ranges from 0 to 0.045. And top four decisive features are the skipping conditions of the four recently played tracks, namely skip9, skip8, skip7 and skip6.

However, since the user history pulled from Spotify web API doesn't have info on whether the history tracks are skipped or not, the another random forest model is trained after taking out the skip features from the data set, leading to an accuracy score of only 0.46 approaching 0.5 as the number of estimators increases. According to the bar plot (Figure.2) of feature importance of this new model, the audio features of the user history doesn't have much correlation with whether the user will skip the current track. What is seen from the graph is a general decaying trend, rather than a dominant feature from previous bar chart (Figure.1). This is because the sessions are imported from different users, who might have entirely diversed tastes in music. However, the habitual inertia seen from the first model exists universally among different users.

The model trained from the dataset can be implemented on the website, and according to the playlist history's audio features, we can classify the current tracks into two categories, the ones that the user tends to skip and the ones the user will not skip. Then the tracks in the latter classification will be rearranged at the front of the new playlist while the former classification will be placed at the end of the new playlist.

Table 1: The Result of deep learning models on the music streaming session dataset without skip features

Model	Training Accuracy(%)	Testing Accuracy(%)
GRU	64.73	63.58
LSTM	65.86	65.04

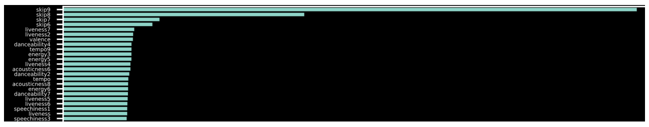

Figure 1: Feature Importance Bar Chart
Random Forest Regression Feature Importance with Skips Bar Plot in Descending Order

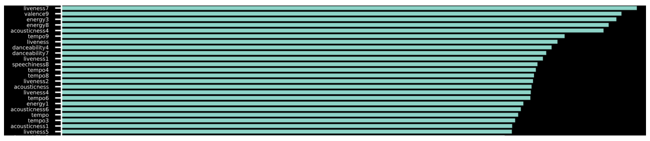

Figure 2: Feature Importance Bar Chart
Random Forest Regression Feature Importance without Skips Bar Plot in Descending Order

6.3 Deep Learning Models

We train both the GRU model and the LSTM with attention model on a small subset of the music streaming session dataset, which has 10000 listening sessions. The testing accuracy and loss stabilize after around 10 epochs with a learning rate of 0.0001 of the Adam optimizer. Both of the models produce similar results, as shown in Table 1. Similarly from previous analysis, we think it is hard for models to predict if a user skips the current track not knowing if the user skips previous tracks. To test how significant the skip features of previous tracks are, we add skip features to all the tracks except for the last track, and got a increase in testing accuracy of 10% as shown in Table 2. This is reasonable because users have different tastes, and we can't simply predict if a user is gonna skip this track, given the audio features of this track and the previous tracks, not knowing user's skip behaviors. Therefore, we think it is necessary to include negative samples in the input to better model user's preferences.

6.4 Playlist Rearranging

To evaluate the website performance, we invite 20 students to be our users. All the students have been Spotify users for at least six months, have listened to at least 50 songs on the platform, and has at least one active playlist. To use our website, each student is asked to login into their Spotify account, select a playlist, rearrange the

Table 2: The Result of deep learning models on the music streaming session dataset with skip features

Model	Training Accuracy(%)	Testing Accuracy(%)
GRU	79.93	74.28
LSTM	82.17	75.3

playlist with the two sort-by methods we assign, review the generated playlist, and tell us which one among the rearranged playlist and the original one is better. We set the minimum requirement for our users since our playlist rearranging algorithm requires playlist as the input. However, such requirement is only for this evaluation to make it effective and is not in our future plan. We decide to have each user test two sort-by methods for efficiency, and we choose to assign methods for them in order to avoid most of them using the same method and leaving the others untested. The comparison between original and rearranged playlist is aimed at collecting the performance of rearranging algorithm. Since the goal of rearrangement is to improve the playlist quality, it is considered to meet the expectation only if user prefers the rearranged playlist to the original one. In addition to these required steps in the evaluation, we also encourage all the users to explore more sort-by methods and provide feedback. Users are encouraged but not required to evaluate the interface and the functionality as well in order to collect suggestions on improving the website structure and mechanisms. Among all invited students, 11 of them showed stronger preference on the reordered playlist, 8 believed the original playlist was still better, and the rest 1 students did not identify obvious differences between the two playlists. Among the feedback, some students preferring rearranged playlists believed the algorithm creates a new playlist without actually changing songs, while some who prefers the original ones said that simply rearranging tracks based on similarity may generate a boring playlist. We believe our website works well in a certain tasks but still needs revision and improvement.

7 CONCLUSION

We designed a website to meet music listeners' needs to rearrange their personal playlists. The website is deployed on Heroku, which allows users to login through Spotify and reorder their playlists based on different approaches we developed. We first develop some basic approaches for users to sort their playlists, including cosine similarity, track name and other audio features. Then we use the pre-trained K-means clustering algorithm to group the tracks in a playlist into clusters. Such classification reflects the similarity between tracks and would be used as the reference of playlist rearranging. Pre-trained random forest classifier and deep learning models estimates the likelihood of skipping the current track based on playlist history, including audio features and conditions, therefore the tracks with the lowest estimated probability of being skipped will be rearranged at the very beginning of the new playlist. The systems have created several viable options for users to decide what kind of order their playlists will be, bringing a sense of refreshment to the existing playlists. As sequence can be an essential influence towards the user's perception of the combination of tracks, such system intends to promote better user experience. The overall performance evaluation of our website conducted with the invited users has provided objective assessments and helpful suggestions to our product. The designed testing procedure allows users to present their feedback after a certain level of experience. While we recognize our models have reached an acceptable level of accuracy, as more than half users prefer the reordered playlist more, this would not be the ultimate result. The evaluation has revealed the shortages in both machine learning models and website design. There is still a long way to go for improving both of them.

8 FUTURE WORK

In the future, we will seek to solve the shortcomings of the current system, that is the general low accuracy of the prediction algorithms without information of the history playlist skip conditions. Solutions include debugging and seeking better fitting parameters for the model, finding other machine learning methods that can prospectively yields better prediction or better criterion for rearrangements etc. Moreover, several new features can be added to the web application satisfying more potential needs from the users. The basis of such additions is a reactive deployment, consistently receiving user's personal experience and reaction towards the functionality deployed. By digging into user experiences, more improvements on perfecting the interaction can be conducted. More personalized data such as user's personal emotions towards the reordered playlist can be collected for future usage.

REFERENCES

[1] [n. d.]. Spotify Dataset 1921-2020, 160k+ Tracks. https://www.kaggle.com/yamaerenay/spotify-dataset-19212020-160k-tracks. Accessed: 2020-11-16.

[2] Sainath Adapa. 2019. Sequential modeling of Sessions using Recurrent Neural Networks for Skip Prediction. *arXiv preprint arXiv:1904.10273* (2019).

[3] G. Adomavicius and Y. Kwon. 2012. Improving Aggregate Recommendation Diversity Using Ranking-Based Techniques. *IEEE Transactions on Knowledge and Data Engineering* 24, 5 (2012), 896–911. https://doi.org/10.1109/TKDE.2011.15

[4] Rachel M. Bittner, Minwei Gu, G. Hernandez, Eric J. Humphrey, T. Jehan, Hunter McCurry, and N. Montecchio. 2017. Automatic Playlist Sequencing and Transitions. In *ISMIR*.

[5] K. Bradley and Barry Smyth. 2001. Improving Recommendation Diversity.

[6] Brian Brost, Rishabh Mehrotra, and Tristan Jehan. 2019. The Music Streaming Sessions Dataset. In *Proceedings of the 2019 Web Conference*. ACM.

[7] Ching-Wei Chen, Paul Lamere, Markus Schedl, and Hamed Zamani. 2018. Recsys Challenge 2018: Automatic Music Playlist Continuation. In *Proceedings of the 12th ACM Conference on Recommender Systems (RecSys '18)*. Association for Computing Machinery, New York, NY, USA, 527–528. https://doi.org/10.1145/3240323.3240342

[8] Arthur Flexer, Dominik Schnitzer, Martin Gasser, and Gerhard Widmer. 2008. Playlist Generation using Start and End Songs. 173–178.

[9] Dietmar Jannach, Lukas Lerche, and Iman Kamehkhosh. 2015. Beyond "Hitting the Hits": Generating Coherent Music Playlist Continuations with the Right Tracks. In *Proceedings of the 9th ACM Conference on Recommender Systems (RecSys '15)*. Association for Computing Machinery, New York, NY, USA, 187–194. https://doi.org/10.1145/2792838.2800182

[10] Beth Logan. 2002. Content-Based Playlist Generation: Exploratory Experiments.

[11] Jan Schluter and Christian Osendorfer. 2011. Music similarity estimation with the mean-covariance restricted Boltzmann machine. In *2011 10th International Conference on Machine Learning and Applications and Workshops*, Vol. 2. IEEE, 118–123.

[12] Malcolm Slaney, Kilian Weinberger, and William White. 2008. Learning a metric for music similarity. In *International Symposium on Music Information Retrieval (ISMIR)*, Vol. 148.