Spotify Recommendation System and Playlist Shuffling

Project 2

Silas Stulz
silas.stulz@gmail.com

Advisor: Prof. Dr. Erik Graf

January 9, 2020

Contents

1 Introduction **2**

2 Solution Description **3**
2.1 Problem Description 3
2.1.1 Content-based filtering 3
2.1.2 Collaborative filtering 3
2.1.3 Clustering 4
2.2 Solution 4
2.3 Recommendation 4
2.4 Playlist Shuffling 5

3 Solution Architecture **5**
3.1 Recommendation 5
3.2 Playlist Shuffling 6

4 Data **7**
4.1 Spotify Developer API 7
4.2 Feature Analysis Spotify 8

4.3 Annotation application 8
4.4 Pre-processing and clean up 9
4.4.1 Recommendation 10
4.4.2 Playlist Shuffling 10

5 Results **11**
5.1 Recommendation 11
5.2 Playlist Shuffling 12

6 Conclusion and Future Work **12**
6.1 Recommendation 12
6.2 Playlist Shuffling 13

A Dataset visualization **14**

B Sketches and screenshots **17**

C Organisational **20**

1 Introduction

I personally use Spotify daily, while I'm working or commuting on the train. According to Spotify I listened to 90541 minutes of music in 2019 (status. Dez 2019). This is 1509 hours or 62 days. Averaging 4.1 hours per day! To make sure I am not always listening to the same stuff, I am interested in ways to find new music I enjoy. This is also the task of this Project 2 at the University of Applied Science Bern.

In 2015 Spotify launched a feature called "Discover Weekly", where they provide each user with a personal playlist with unique songs they might like. While the specific algorithm and system that Spotify uses are not public, we know a few things about it. They use a combination of collaborative filtering, Natural Language Processing (NLP) and audio modeling. The probelm is, that they do not exactly know which songs the user likes and which they don't because they rarely like or add the songs to a playlist.

In this project, I want to address the problem in an alternative way. Basing my machine learning algorithm on a personally curated dataset to decide if I like a song or I do not. This model is based on the musical components of a song (accousticness,

tempo etc.). This text serves as a documentation for two applications. Later in the project I pivoted to a different use case, because there was no bearabale way to get access to the Spotify dataset or to replicate it in a proper way.

In this pivot, I built a Spotify playlist shuffle application which shuffles a Spotify playlist based on a reference track in the playlist and positions the other songs accordingly on the similairity to that reference song. This application is based on a clustering algorithm calculating the similairity between two songs. It uses the same datasource as the first application.

2 Solution Description

2.1 Problem Description

Recommender systems are everywhere in the web. Netflix recommends movies to you, Amazon what you could potentially buy and Spotify recommends you new songs. Today, there are basically two main ways to approach recommendation generation. Content-based filtering and Collaborative filtering.

2.1.1 Content-based filtering

Content-based filtering uses item properties (features) to recommend other items similar to what the user likes, based on feedback. So, we look at what kind of items the current user likes (which songs) and calculate with a similarity metric (ex. dot product) the recommendation. These recommendation are fit to a specific user and can not be transfered to another user.
This method doesn't need any data about other users. This makes it easier to scale the system. Additionally the model can recommend specific niche items that very few other users are interested in. This reduces the bubble effect.
On the other hand, this type of recommendation requires a lot of domain knowledge, because feature representations are hand-engineered. Also, the model has a limited ability to expand on the users existing interests.

2.1.2 Collaborative filtering

Collaborative filtering uses similarities between users and items at the same time to provide recommendations. That allows the model to recommend an item to user A based on the interests of a similar user B. In contrast to the content-based filtering

model, there is no need for domain knowledge. The embeddings are automatically learned. One of the biggest benefits, is that the model can help users discover new interests. One of the biggest problems is the "cold-start problem". This is that when an item is not seen during training the system can't create an embedding for it.

2.1.3 Clustering

Clustering is not used for recommendations but to form groups or collections from a dataset. For example if we look at a music dataset, we can organize the dataset into groups based on genre, or decade. Clustering is mostly unsupervised machine learning. Now to group items together you first need to find similar examples. But how do you define what is similar? This is when the similarity measure comes to play. The similarity measure defines how similar two items are to eachother. This was the perfect method for reranking a Spotify playlist.

2.2 Solution

In this section, I will describe two different solutions I developed during this semester. One will include the initial idea, to recommend songs and the other on shuffling existing playlists. Estimated two thirds of the semester went into working on the recommendation engine. The playlist shuffling application was a pivot to produce something that was usable.

2.3 Recommendation

The initial plan was to play the only strength a self developed music recommendation system could have over a commercial one from Spotify. The strength that I know exactly what kind of music I like and what kind of music I do not like. This lead to a content-based filtering model. This was also the only viable choice, because Spotify does not give access to a lot of user data.

The first ideas, phantasized with a web application, but I thought it would be smart to implement it into the already existing annotation application I created. That is why the plan was a mobile application, where you could get track recommendations, based on your Spotify listening history, as well as annotate new data in the application itself to improve the model. Look at appendix for layout images and drawings.

2.4 Playlist Shuffling

During a meeting with Prof. Graf, I explained him my problem, that I could not further develop the recommendation system because of a limitation in the Spotify API (more detail in results). He came up with the idea to create an application that would not recommend new songs, but rather sort the existing ones in a playlist. I quickly came up with different ideas on how to implement this and decided it would be perfect as a simple web application.

The user can chose a playlist he wants to shuffle and select a reference song, which serves as a comparison object for the other songs. The playlist is then shuffled with the reference song on top, in descending order there are the most similar songs ranked.

3 Solution Architecture

3.1 Recommendation

There were four main components planned for the application. The full architecture was never deployed, but I will nevertheless explain the blueprint behind the application that I had in mind.

1. Spotify Developer API

   The Spotify Developer API is the main ressource for the training data. For each song that is on Spotify, the API provides the musical component data for this song (accousticness, tempo etc.). The model is based on this data, which the REST-API gathers from the API.

2. REST API

   The REST API handles the model training, communicating with the Spotify API, as well has delivering all the data to the mobile application. The model should be retrained every 50 tracks to improve performance.

3. Mobile Application

   The mobile application delivers the recommendations to the end-user. It also serves as a training tool to label new songs for a improved model performance.

Communicates only with the developed REST-API and not directly with the Spotify API.

The machine learning model was trained on a manually pre-curated dataset, containing the musical components for every song (I go more into detail under Data). The model was then trained, first with different Classifiers from the sklearn library. Examples are KNeighbors, RandomForest, SGDClassifier or SVC. There were also some tries with tensorflow and neural networks. But because they did not achieve better results, I stayed with the sklearn library.

3.2 Playlist Shuffling

The playlist shuffling application consist of basically the same components as the recommendation application.

1. Spotify Developer API

   Same as above. Additionally, Spotify provides information about the playlists and songs.

2. REST API

   The REST API handles everything with machine learning. In this case it provides only one endpoint, "/shuffle". Which shuffles the given playlist for a given reference track. All the other information, the web application gets directly from the Spotify API. The REST-API is written in Python, with the Django-rest-framework [1] and deployed on a simple linux server on DigitalOcean [2].

3. Web Application

   The web application delivers an interface to the end-user with the ability to shuffle playlists. The Web application communicates with the REST-API and directly with the Spotify API. It handles the authentication with the Spotify API and receives a token directly, which it will provide to the REST-API for shuffling the playlist. It is written in Typescript using the angular framework [3]. It is deployed on a simple linux server on DigitalOcean.

[1] `https://www.django-rest-framework.org/`
[2] `https://www.digitalocean.com/`
[3] `https://angular.io/`

I created a manual similarity measure, which weighs all features in the dataframe as same important. To calculate the smilarity measure, I first subtract the song data frame from the reference song dataframe, to find the differences in the features. From there I calculate the root mean squared error (RMSE).

$$\sqrt{\frac{f_1^2 + f_2^2 + ... + f_n^2}{2}}$$

Where f is the value of the substraction from the feature from object to the feature from the reference object.

4 Data

In this section, I am going to talk about the used datasets and their structures. The data in this project, all came from the official Spotify API. It was considered to use other available datasets, but their lack in actuality, accuracy and correctness led to using only the Spotify API. Other datasets include:

1. Acoustic Brainz [4]
   Derived from Music Brainz [5] with an estimated 1.8 Million Songs. Uses different algorithms from different University Research Groups. They differ a lot in values for simple things like BPM.

2. One Million song dataset [6]
   As the name describes the dataset contains feature analysis for 1 Million songs. Not updated since 2012 and therefore useless for this project. The feature analysis is done by "The Echo Nest" [7], a company that was acquired by Spotify in 2014.

4.1 Spotify Developer API

The Spotify Developer API [8] is Spotifys only official API. It offers a wide variety of endpoints, gathering and manipulating information about songs, playlists and users. A full list of endpoints can be found in the Web API Reference [9].

[4] `https://acousticbrainz.org/`
[5] `https://musicbrainz.org/`
[6] `http://millionsongdataset.com/`
[7] `http://the.echonest.com/`
[8] `https://developer.spotify.com`
[9] `https://developer.spotify.com/documentation/web-api/reference/`

4.2 Feature Analysis Spotify

The main data for my project came from the Audio Features endpoint [10]. In the table below, I summarized the data available for every track. The lines in blue were used in the dataset.

Feature	Description	Type
duration_ms	Duration of the track	int;milliseconds
key	Estimated overall key [11] of the track	int[-1..11]; -1 if no key
mode	Modality (major or minor) Major = 1 Minor = 0	int[0,1]
time_signature	How many beats in each bar	int[0..∞]
acousticness	Confidence value whether the track is acoustic	float[0..1]
danceability	Confidence value whether suitable for dancing	float[0..1]
energy	Confidence value of intensity, activity	float[0..1]
instrumentalness	Confidence value wheter a track contains vocals	float[0..1]
liveness	Detects presence of an audience	float[0..1]
loudness	Overall loudness in decibels (dB)	float[-60..0]
speechiness	Detects presence of spoken words	float[0..1]
valence	Musical positiveness	float[0..1]
tempo	Tempo in beats per minute (BPM)	float[0..∞]

Table 1: Spotify API Audio Analysis

4.3 Annotation application

To implement a content-based recommendation system, I basically had to create a dataset from scratch. To let my model know which songs I like and which I dislike, I had to annotate every song I was going to feed to the model. To make this whole process a little easier, I created an android application which can be used to do binary annotation.

The application gathers the last 20 songs I listened to on Spotify via Spotify Developer API. It displays all the songs in card form stacked on to eachother. From there you can either swipe the track right (like), swipe it left (disklike) or press it to play the song (needs Spotify installed on device). There is also an overview, a list where

[10] `https://developer.spotify.com/documentation/web-api/reference/tracks/get-audio-features/`

every song you swiped on is recorded and the value can also be changed (liked to disliked or vis versa). It is also possible to play the songs from the overview.

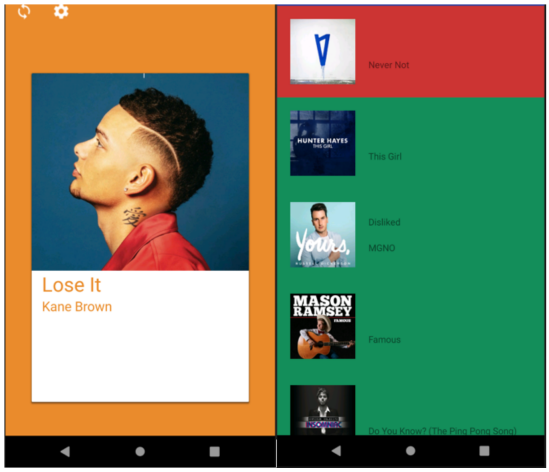

Figure 1: left: Main screen to swipe; right: Overview screen

Another feature, that has later been removed again, is to add liked songs automatically to your library in Spotify. As well as to add the diskliked songs to a playlist, which can be choosen in the settings screen.

The application is connected to a Realtime Database on Firebase [12]. The database saves all the data about the songs swiped, and avoids having to swipe multiple times on the same track.

I wrote a seperate blog post [13] about my application. Also the Github Repository can be found here [14].

4.4 Pre-processing and clean up

After the initial annotation of the songs, the next step is to prepare the data and do some clean up. Because preparing the dataset, differs for the two application, I will talk about them seperately.

[12] https://firebase.google.com/

[13] https://towardsdatascience.com/machine-learning-making-binary-annotations-a-little-less-boring-51537497af3b

[14] https://github.com/Hyferion/SpotifyRecLabelingAndroid

4.4.1 Recommendation

For a better overview I will explain the process step by step.

1. Getting the data
   From the Realtime Database on Firebase we can pull all data in JSON format which makes it pretty easy to process it. Currently my personal dataset contains estimated 800 entries. From there I get the audio analysis for every song from the Spotify API and create a .csv file. From this .csv file I can easily generate a Dataframe to feed the model.

2. Defining relevant features
   As it does not make sense to train the model on all data the Spotify API provides, on basis of domain knowledge and some experimenting I found the most releveant features and cut everything else out. Look at table 1 to see the relevant features.

3. Undersampling or oversampling the dataset
   Because the dataset is heavily unbalanced, the model tended to always chose to say "liked" on a song, because it was more present in the dataset. I tried oversampling the dataset by generating duplicates and also generating synthetic copies of the underrepresented class. Undersampling was simple to just reduce the dataset to the count of the underrepresented class. I aimed for a 50/50 class ratio.

4. Split dataset into training and test
   This was the shortest step, to split the whole dataset into two pieces. The training-test ratio was about 80/20.

The finished datasets looks like this:

Liked	Acousticness	Danceability	Energy	Instrumentalness	Tempo
0	0.28	0.496	0.639	0.0	147.764
1	0.0491	0.707	0.642	0.0	96.021
...	...	...	...	...	...

4.4.2 Playlist Shuffling

In terms of Pre-processing the shuffling application is much simpler.

1. Getting the data
   The application fetches the information about the playlist and all songs that are in that playlist. There is no annotation or additional data required.

2. Defining relevant features and scaling them (Feature scaling)
   Here it is also required to define the relevant parameters. I chose the exact same ones like in the recommendation application. Additionally for the similarity algorithm to work, the features have to be in the same range, luckily all except the tempo are already in the range zero to one.

3. Define reference track The user defines a track from a playlist that serves as a reference track. The model only has to take that track and put it in a seperate data frame to do the comparison to all the other tracks.

5 Results

5.1 Recommendation

As a first result, I developed a fully functional Android application for binary annotation that can further be improved. When I got to develop the model, I learned what algorithms that work on my dataset and which did not work quite that good. In the end, I achieved the best results with a RandomForestClassifier (parameters can be found in the code). Others like SGD or SVC did not work that well. With the RandomForestClassifier I was able to beat randomness with an estimated accuracy of around 60% on a very diverse dataset. This means that there were very different types of music in this dataset. It depends heavily on the training and test data. If the model would be tested on a more polarised dataset, it could even have a perfect score.

Unfortunately, I was not able to develop the whole system. It was not possible to search the Spotify API for songs that would match special criterias like tempo, accousticness etc. I tried to scramble the whole Spotify API with a search algorithm, but recognized it was not that trivial to do that. Because I could only look through songs with the "search function" and songs would not be found with random character combinations.

Another solution I tried was to take a database like musicbrainz and to use every track in the musicbrainz database as a search term for the Spotify API search function. Sadly, I recognized that even this method would not be easy for 30 million songs. So I decided to not build the whole system and just let the machine learning

model rest and rather pivot and develop a system that I could build in the amount of time that was left.

5.2 Playlist Shuffling

The result is a fully functional web application that is deployed under reordr.li [15]. As the pivot to a playlist shuffling application came late in the process, and the fact that the whole development and deployement process of the different parts of the system consumed substential time, I was not able to develop the clustering algorithm like I wanted to. Currently the algorithm just does basic comparison of two tracks and orders them in descending order of value.

The result is a funny little application that displays what is possible with the Spotify Developer API and could even be formed into a useful application with a little more work on the similarity model.

6 Conclusion and Future Work

6.1 Recommendation

I knew from the beginning that the recommendation application would be a really big project. And I was aware, that I would run into some major problems, because I was heavily dependend on what the Spotify API could do. In the end I was not able to complete the whole system due to limitations on the available data. But I still got a lot of experience developing a recommendation system, learning the pro's and con's of the different learning algorithms. And I can say, that I developed a system, that was able to predict, whether I like a specific song or not. It just was not possible to develop a whole application around that. Maybe one day Spotify will expand its API and solve this problem. I haven't yet found a smart solution, to solve the problem from my end. Maybe if someone would look at the problem from a different angle, they would be able to solve this. One way I thought of would be to use the musicbrainz database, and from there search for every song on Spotify and get the audio analysis, or even generate the audio analysis on your own.

Otherwise, I think there is still potential to improve the recommendation model

[15] `https://reordr.li`

on its own. Maybe even go in a different direction and start clustering the songs into genres, or speed (BPM). I think there is a lot of potential for future applications.

6.2 Playlist Shuffling

There was not enough time to test and improve the system further. One reason for that is that the system is really hard to test. How do you verify if a playlist is shuffled correctly? Which features are more important? Does a song feel more similar if the tempo is off by -.2 or the accousticness? I think there is a lot of room for experimenting and trying out this stuff, Unfortunately, I was not able to do that in the amount of time I invested in this project.

I think with a lot of time and effort invested in the clustering algorithm, this application could really be useful. I think there has to be a lot of domain knowledge acquired, to know when a track sounds similar to another for a human being. From there it could be possible to not only shuffle playlists, but to generate new ones.

One major component to work on is the similarity measure. Currently it is manually crafted. From there maybe there would be better results if we weight different features on a different scale. Maybe the tempo (BPM) is more important for a song too feel similar then accousticness. From there it is a possibility to switch to a supervised similarity measure. With the help of a supervised deep neural network (DNN), the embeddings on the feature data itself could be generated.

A Dataset visualization

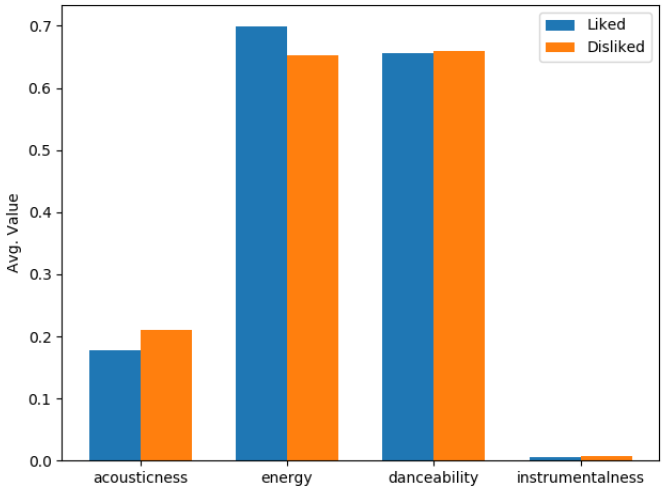

Figure 2: Difference in relevant features for classes

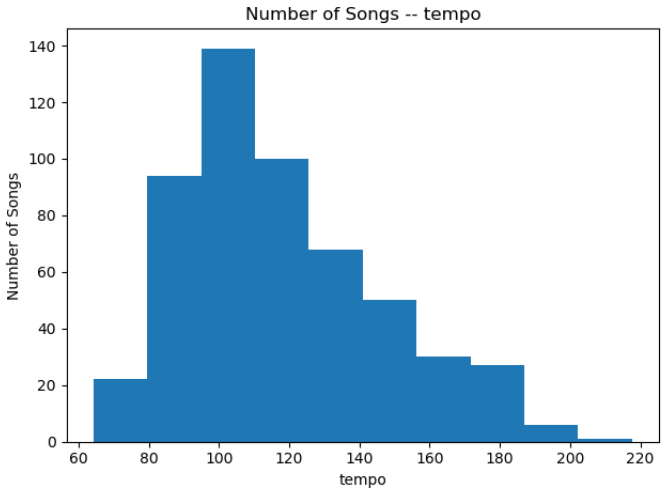

Figure 3: Distribution of tempo for number of songs

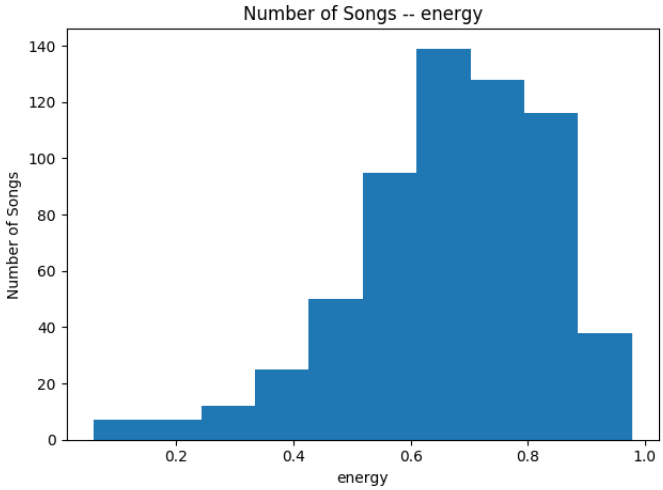

Figure 4: Distribution of energy for number of songs

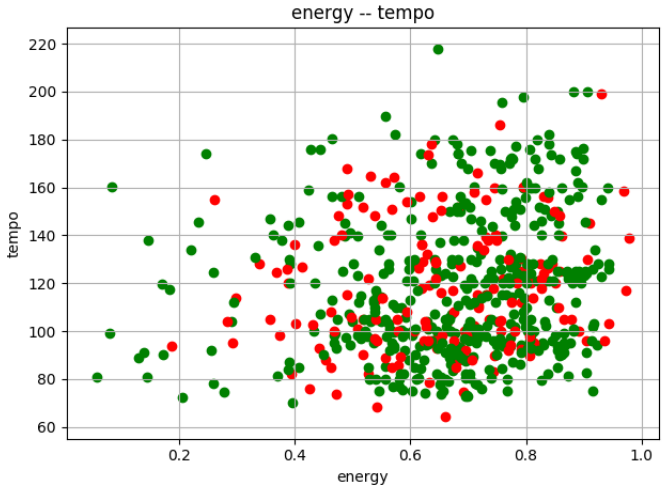

Figure 5: Green:Liked, Red:Disliked

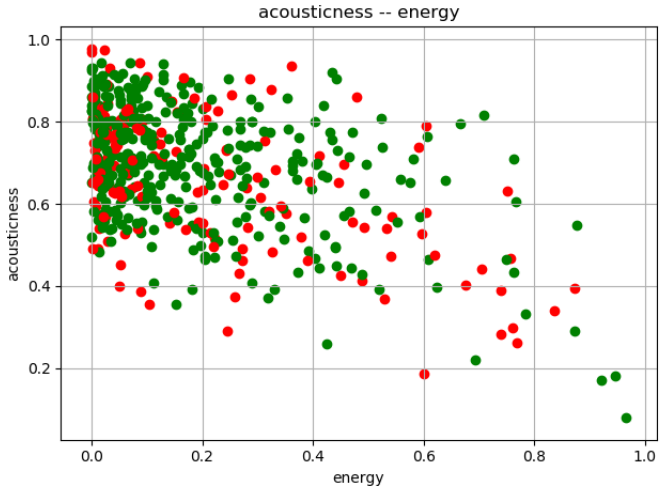

Figure 6: Green:Liked, Red:Disliked

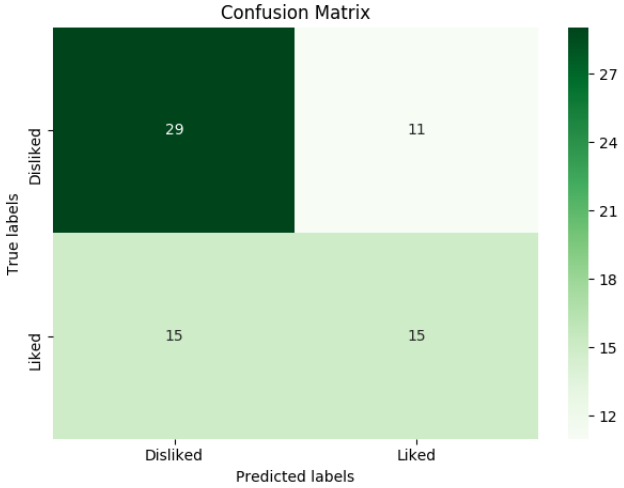

Figure 7: Confusion matrix of example run

B Sketches and screenshots

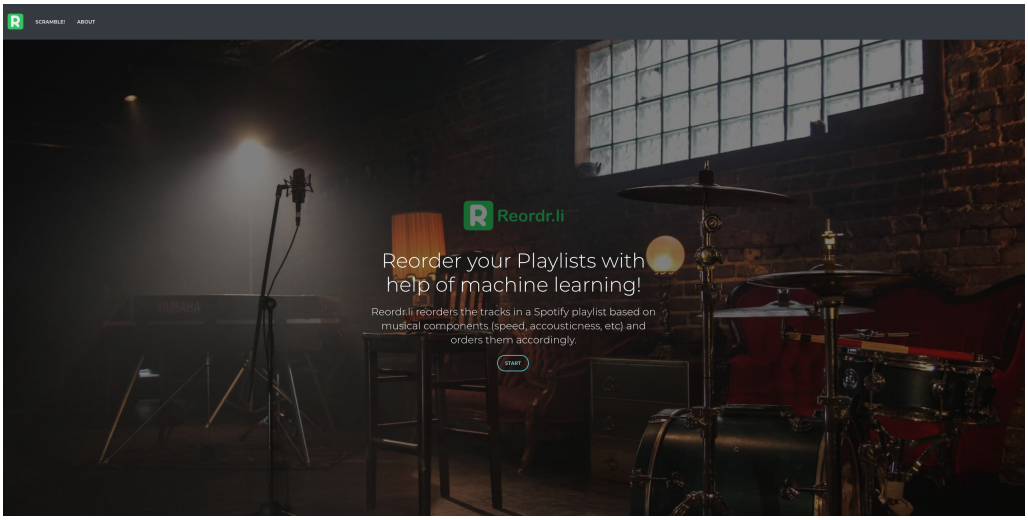

Figure 8: Home view of Reordr.li

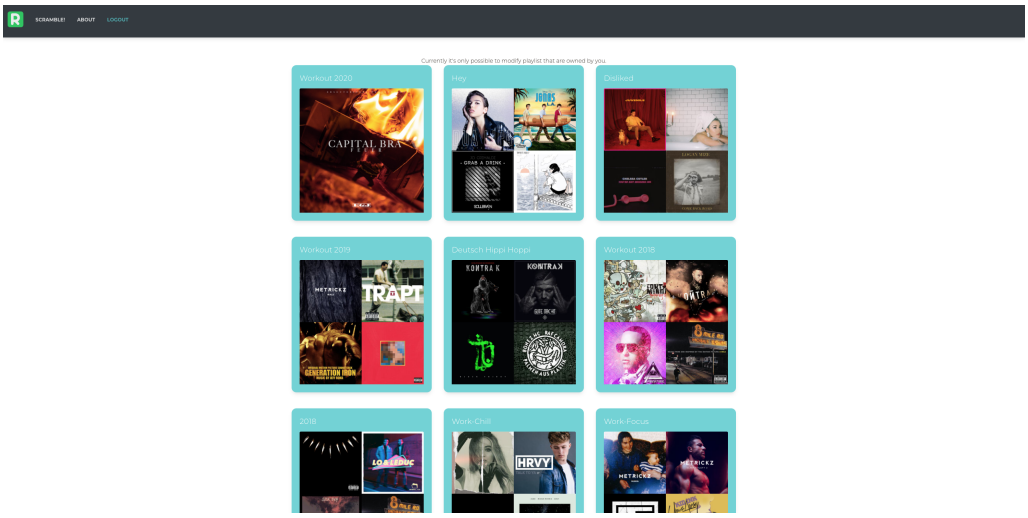

Figure 9: Playlistoverview view of Reordr.li

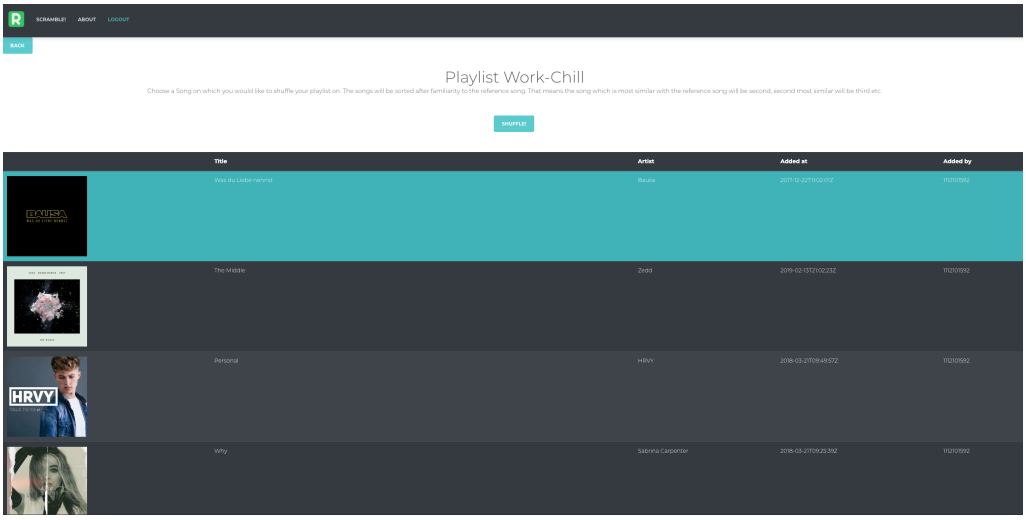

Figure 10: Playlist view of Reordr.li

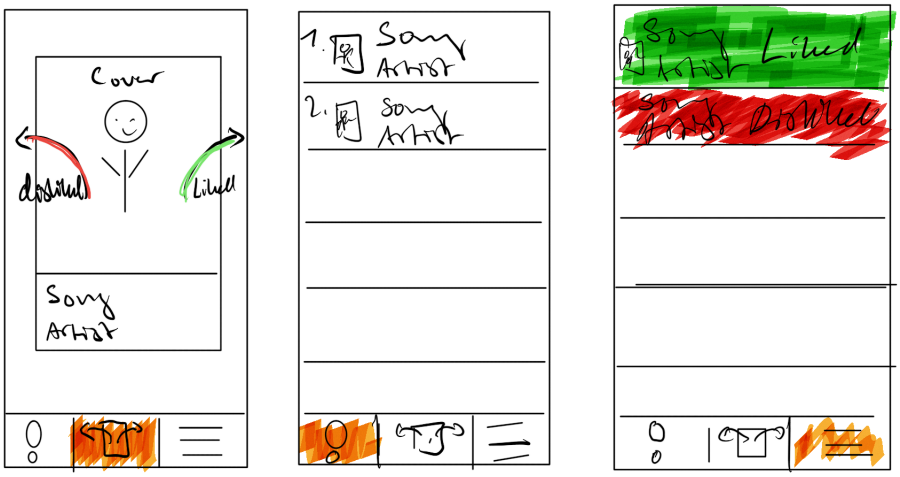

Figure 11: Mobile application UI sketches

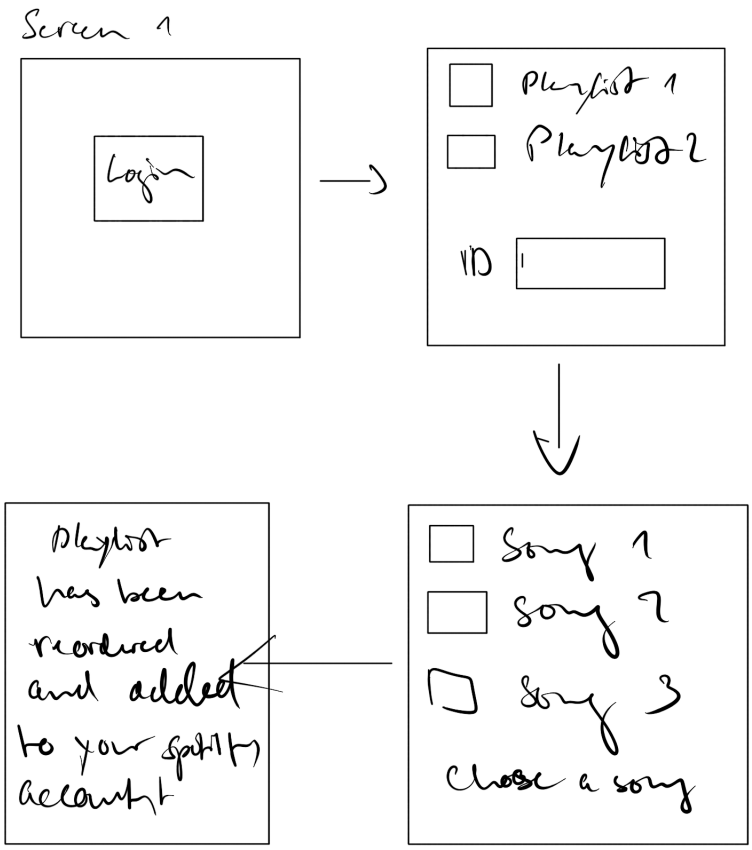

Figure 12: Playlist shuffling sketch

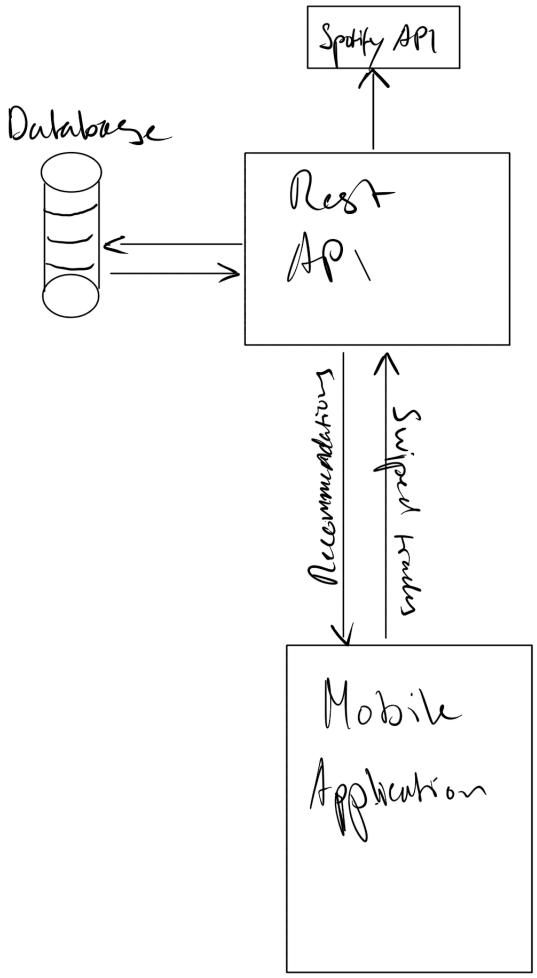

Figure 13: Recommendation architecture sketch

C Organisational

teamgantt
Created with Free Edition