COMS E6998 - Cloud computing & Big Data
Spring 2022

Project Report

Instructors:

Sambit Sahu, Rishabh Gupta

Mentor:

Taku Takamatsu

Chaofan Wang(cw3288)
Chen Li(cl4155)
Danmei Wu (dw2943)
Zipei Jiang(zj2321)

Project Proposal

Motivation

Music, unlike web pages that can be obtained by users through search engines, are not written in human recognizable languages. As the number of songs available on the internet increases dramatically in recent years and people can only search music indirectly by its related information like descriptions and lyrics, finding desirable music becomes a problem: People need to listen to the music to decide if they like it or not. It takes a few minutes for each song, which is time-consuming. And to reach a song, people need to search its genre, style, age or the author. For example, to find the song “Heart of the Sunrise” without directly typing its title, people need to search “the progressive rock”, “rock music in the 70s”, or “songs written by ‘Yes’”. This prevents people from finding music in the genre or age they are not familiar with, or written in languages they don’t know. Thus, we believe that instead of letting people search songs manually, techniques that automatically advertise music are a better way for people to discover music they like.

Description

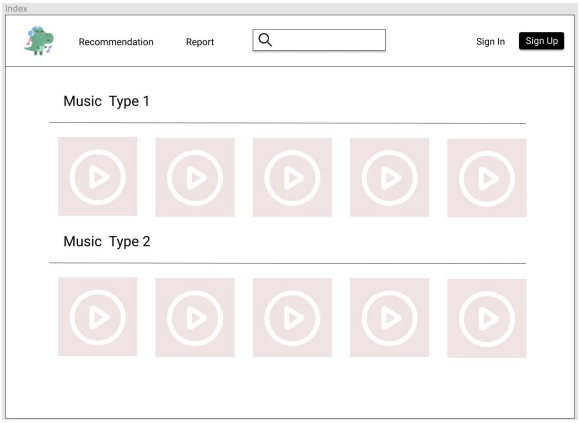
To solve this issue, the team developed a music recommendation platform that automatically generates suggested songs based on either the result of a short test or the user's existing list of liked songs. In specific, users can load the like list from their Spotify account to get the report, which summarizes the preferences of the user and gives a list of recommended music. And if the user decides not to load the Spotify like list, they will take a test to listen to the short pieces of randomly selected songs and tell the system if they like, dislike or are neutral to the song to get the report.

Other features of our project are:

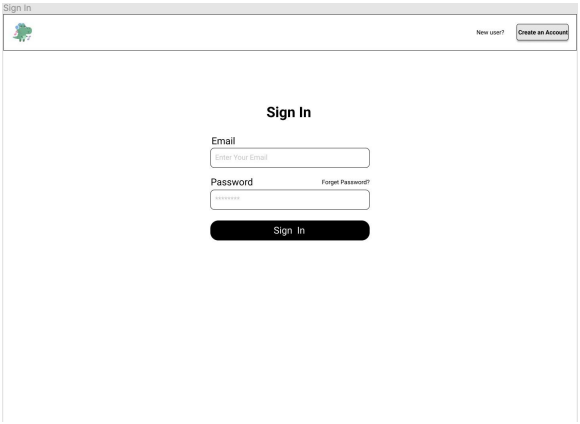
- Registration and login functionalities that supports password resetting by sending email or entering the current password
- Spotify authorization to access the Spotify playlist
- Searching functionality that allows the user to search songs by title or author from Spotify database
- Likelist that stores songs the user “liked”, which can be accessed and updated once the user logs in their account

Project Prototype

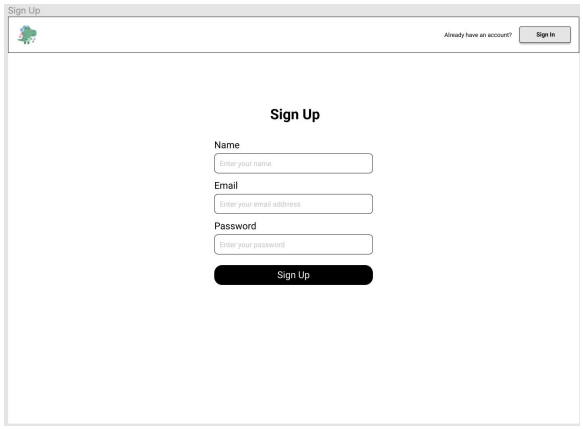
Home page:

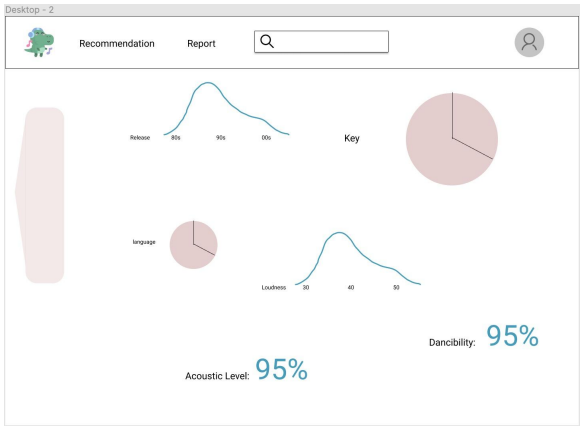


Login Page:

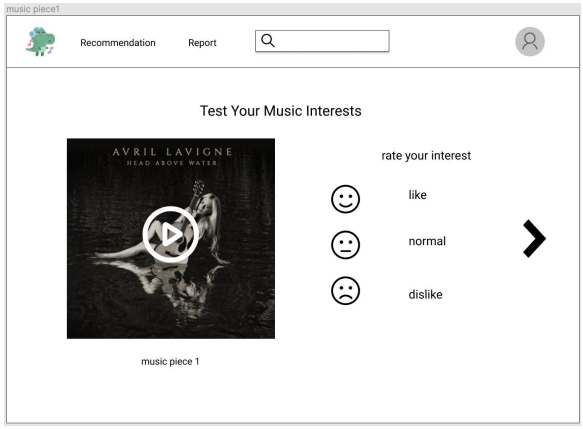


Sign Up Page:

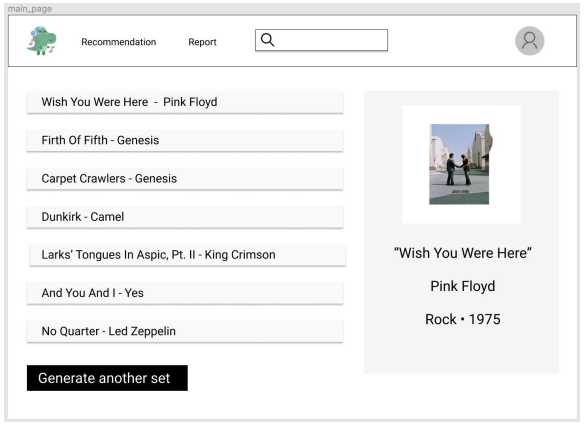




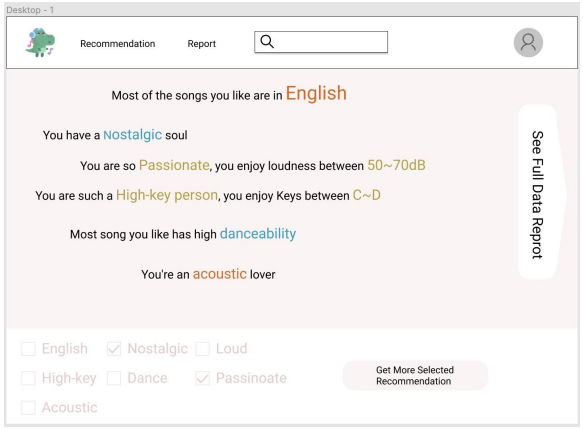
Test Interest page:



Recommendation Page:



Report Page:



Architecture Diagram

See Reference Figure 3.1

Architecture Flow

- **Login/Register/Authorization**

A new user needs to provide username, email address, and password for registration. Once the user clicks the sign-up button, we will request Spotify authorization from the user, if the user agrees to this request, then we call Spotify authorization API to redirect the user to Spotify authorization webpage after the user clicks agree or cancel, we redirect the user to our homepage.

During the whole process, first, we store the user's information in User Table in RDS. Then if the user agrees to our Spotify authorization request, we will get an access token returned by Spotify API, we pass the token to ProcessSpotifyData lambda, then the lambda can use this token to get the user's playlist and spotify id, and we insert the user's playlist to our User Like Table in DynamoDB for our model to recommend songs for the user, and we send user's Spotify id with user's registration information to Amazon SQS, then we have another Lambda function to retrieve the message from this SQS and write to User SpotifyId Table in RDS, we store this information for future use.

As for login, we check whether the user exists and whether the user's password is correct, in addition, users can reset their password if they forget it, we pass the request to ResetPassword lambda, then this lambda sends an email to the user using Amazon SES, once the user receives this email, he/she can reset the password via the link in the email.

- **Test Interest**

This lambda triggers when the user refuses our Spotify authorization request in sign up or the user just wants to update his/her interest, when the user gets to test interest tab, we will give some musicIds that are randomly chosen by certain properties(for example, rap music or popular music), and combine the details using Spotify API (https://developer.spotify.com/documentation/web-api/reference/#/operations/get-several-tracks) then he/she can like/dislike each song, after finishing the test, we will update User Like Table.

- **Recommendation**

When the user clicks the recommendation tab in the navigation bar, or the user clicks the generate another set in the recommendation page or the user chooses several options from report in the checkbox to get more recommendations, it will trigger Recommendation lambda to get to the Recommendation Model in SageMaker, and return the music recommendation id and combine the detailed information get from Spotify API (https://developer.spotify.com/documentation/web-api/reference/#/operations/get-several-tracks) to the frontend.

- **Report**

When the user clicks the report tab in the navigation bar, we first retrieve user's liked songs ID list from User Like Table(To speed up the process, we use OpenSearch), then we use Spotify API(https://developer.spotify.com/documentation/web-api/reference/#/operations/get-audio-features) to search these songs' features, such as acousticness,

danceability, energy, instrumentalness, liveness, key, valence and so on, the Report lambda will send these data to our font end, then our front end will generate a report showing user's preference. For example, when energy>=0.8, the frontend will show "You are so Passionate"

- **Search**

The user can input the search query in the search bar, then our front end will pass the query string to Search lambda, then the lambda calls Spotify search API(https://developer.spotify.com/documentation/web-api/reference/#/operations/search).

- **MusicLikeList**

Users can see their liked songs in their profile, when they are at their liked songs tab, we send requests and pass user id to MusicLikeList lambda, the lambda will search in UserLikeTable with the user id, and we will use Spotify API(https://developer.spotify.com/documentation/web-api/reference/#/operations/get-several-tracks) to get the detailed information, and combine them, return the results to users.

Data Model

- **User Table(RDS)**

Name	Type	Constraint
UserId	string	Primary Key
Username	string	NOT NULL
Password	string	NOT NULL, must include at least 6 letters
Email	string	NOT NULL,UNIQUE

- **User Like Table(DynamoDB)**

userId(partition key), likelist
likelist is a string, containing a list of music id, separated by ","

UserId	Likelist
test888	4dkoqJrP0L8FXftrMZongF, 0Ajr6kJrlYgAl5gBKvTThV, 3pa1wdHEFWgdRfLcrtUv48

- **User SpotifyID Table(RDS)**

Name	Type	Constraint
UserId	string	Primary Key
SpotifyID	string	NOT NULL

APIs

- POST /login

Description: user logins
Input: (the password will be encrypted)

```
{
   "email":"xxxx@xxx.xxx",
   "password":"xxxxxxx"
}
```

Output:

```
{
   "statusCode":200,
   "body":{
      "userId":"xxxx",
      "message":"login successfully"
   }
}
```

- POST /signup

Description: user signups
Input:

```
{
   "userName":"Joann",
   "password":"xxxxxxx",
   "email":"xxxx@xxx.xxx"
}
```

Output:

```
{
   "statusCode":200,
   "body":{
      "userId":"xxxx",
                  "message":"create user successfully"
   }
}
```

- PUT /password/{userId}

Description: update the password, forget=true means when a user forgets the password, and after he/she clicks the email that receives from us to reset the password, forget=false means to update the password in the profile
Input:

```
{
   "newPassword":"xxxxxxx",
   "forget":true
}
```

Or

```
{
   "newPassword":"xxxxxxx",
   "oldPassword":"xxxxxxx",
   "forget":false
}
```

Output:

```
{
   "statusCode":200,
   "body":{
            "message":"update password successfully"
   }
}
```

- POST /forget

Description: When a user forgets the password, we will send an email to let him/her reset the password.
Input:

```
{
   "email":"dw2943@columbia.edu"
}
```

Output:

```
{
   "statusCode":200,
   "body":{
                  "message":"send email successfully"
   }
}
```

- GET /interest/{userId}

Description: Triggers when the first time when a new user comes in, and when she/he takes the music interest test automatically, or a user plans to take the music interest test manually by clicking the “Test Interest” in the menu.
Output:

```
{
   "statusCode":200,
   "body":{
      "count":4,
      "music":[
         {
            "musicId":"xxxx",
            "musicName":"xxxx",
            "artistName":"xxxx",
            "imageUrl":"xxxx",
            "musicUrl":"xxxx"
         }
      ],
            "message":"fetch test interest successfully"
   }
}
```


- POST /interest/{userId}

Description: Triggers when the user submits the music interest test result. It can also trigger when the user clicks the like icon of a song.

Input: like parameter: 1 means like 0 means normal, -1 means dislike


```
{
   "music":[
      {
         "musicId":"xxxx",
         "like":1
      }
   ]
}
```


Output:


```
{
   "statusCode":200,
   "body":{
            "message":"send interest test successfully"
   }
}
```


- GET /search?userId=&q=&page=

Description: Triggers when the user clicks the search, the user can search the artist name, and music name, we support obscure search.

When the query parameters can userId, which means that the user logins, the result will show whether the user likes the song or not.

Output:


```
{
   "statusCode":200,
   "body":{
      "count":5,
      "music":[
         {
            "musicId":"xxxx",
            "musicName":"xxxx",
            "artistName":"xxxx",
            "imageUrl":"xxxx",
            "musicUrl":"xxxx",
            "like":0
         }
      ],
            "message": "get search data successfully"
   }
}
```


- GET /recommendation/{userId}

Description: Triggers when the user clicks on the recommendation menu bar.

Output:


```
{
   "statusCode":200,
   "body":{
      "count":5,
      "music":[
         {
            "musicId":"xxxx",
            "musicName":"xxxx",
            "artistName":"xxxx",
            "imageUrl":"xxxx",
            "musicUrl":"xxxx",
            "like":0
         }
      ],
         "message":"fetch recommendation data successfully"
```

```
  }
}
```


- GET /report/{userId}

Description: Triggered when the end of the month, or the user clicks on the navigator report bar

Output


```
{
  "statusCode":200,
  "body":{
    "acousticness":0.00242,
    "danceability":0.1,
    "energy":0.8,
    "instrumentalness":0.00686,
    "liveness":0.9,
    "loudness":0.7,
    "speechiness":0.2,
    "key":2,
    "mode":0,
    "valence":0.428,
    "release":{
      "1980":0.1,
      "1990":0.85,
      "2000":0.05
    },
          "message": "fetch report data successfully"
  }
}
```

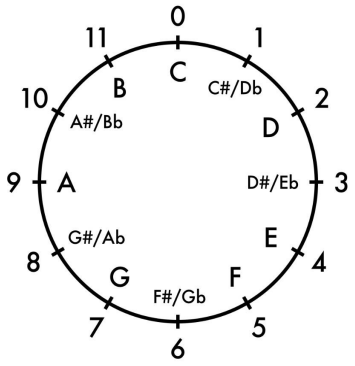

Description of report data:

Acousticness: A confidence measure from 0.0 to 1.0 of whether the track is acoustic. 1.0 represents high confidence the track is acoustic.

Danceability: Danceability describes how suitable a track is for dancing based on a combination of musical elements including tempo, rhythm stability, beat strength, and overall regularity. A value of 0.0 is the least danceable and 1.0 is the most danceable.

Energy: Energy is a measure from 0.0 to 1.0 and represents a perceptual measure of intensity and activity. Typically, energetic tracks feel fast, loud, and noisy.

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



Key: The key the track is in Integers map to pitches using standard Pitch Class notation. E.g. 0 = C, 1 = C♯/D♭, 2 = D, and so on(see above image). If no key was detected, the value is -1

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

Loudness: The overall loudness of a track in decibels (dB). Loudness values are averaged across the entire track and are useful for comparing relative loudness of tracks. Loudness is the quality of a sound that is the primary psychological correlate of physical strength (amplitude). Values typically range between -60 and 0 db.

Mode: Mode indicates the modality (major or minor) of a track, the type of scale from which its melodic content is derived. Major is represented by 1 and minor is 0.

Speechiness: Speechiness detects the presence of spoken words in a track. The more exclusively speech-like the recording (e.g. talk show, audio book, poetry), the closer to 1.0 the attribute value. Values above 0.66 describe tracks that are probably made entirely of spoken words. Values between 0.33 and 0.66 describe tracks that may contain both music and speech, either in sections or layered, including such cases as rap music. Values below 0.33 most likely represent music and other non-speech-like tracks.

Valence: A measure from 0.0 to 1.0 describing the musical positiveness conveyed by a track. Tracks with high valence sound more positive (e.g. happy, cheerful, euphoric), while tracks with low valence sound more negative (e.g. sad, depressed, angry).

- GET /like/{userId}?page=

Description: Triggers when user clicks the profile->liked songs
Output:


```
{
   "statusCode":200,
   "body":{
      "count":20,
      "music":[
         {
            "musicId":"xxxx",
            "musicName":"xxxx",
            "artistName":"xxxx",
            "imageUrl":"xxxx",
            "musicUrl":"xxxx",
            "like":1
         }
      ],
            "message": "fetch user like data successfully"
   }
}
```


- GET /morerecom?userId=&q=

Description: Triggers when user clicks the report->get more recommendation or home
Input:
query may contain
max_acousticness, min_acousticness
max_danceability, min_danceability
max_energy, min_energy
max_instrumentalness,
min_instrumentalness
max_key, min_key
max_liveness, min_liveness
max_loudness, min_loudness
max_speechiness, max_speechiness
max_valence, min_valence
max_popularity, min_popularity
Output:


```
{
   "statusCode":200,
   "body":{
      "count":5,
      "music":[
         {
            "musicId":"xxxx",
            "musicName":"xxxx",
            "artistName":"xxxx",
            "imageUrl":"xxxx",
            "musicUrl":"xxxx",
            "like":0
         }
      ],
                  "message": "fetch more recommendation data successfully"
   }
}
```


- POST /authorize

Description: Triggers when user clicks the signup then authorizes Spotify
Input:

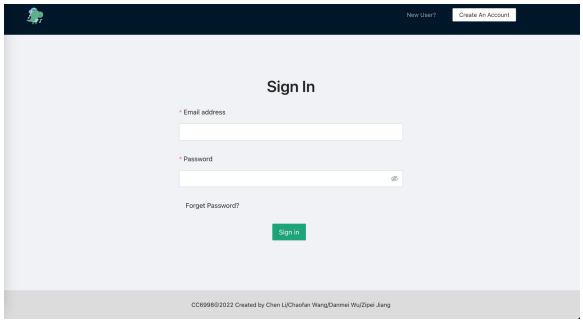
```
{
   "userId":"xxxx",
   "accessToken":"xxxx"
}
```

Output:

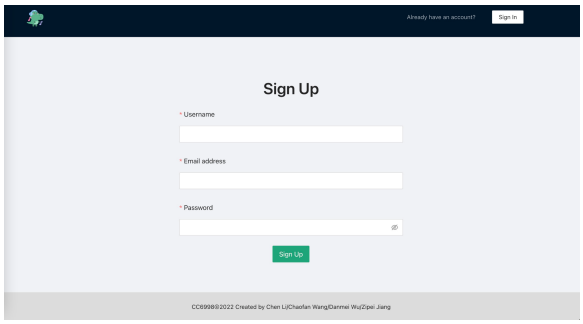
```
{
   "statusCode":200,
   "body":{
                  "message":"authorize  spotify successfully"
   }
}
```


Features Implementation

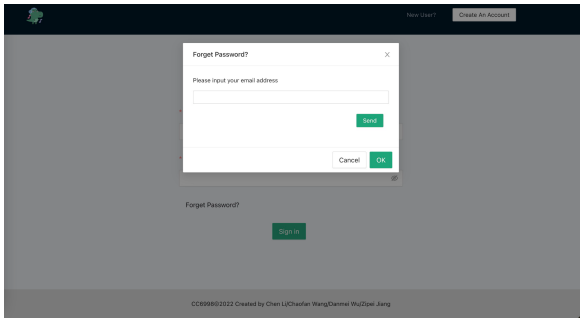
- Login Page

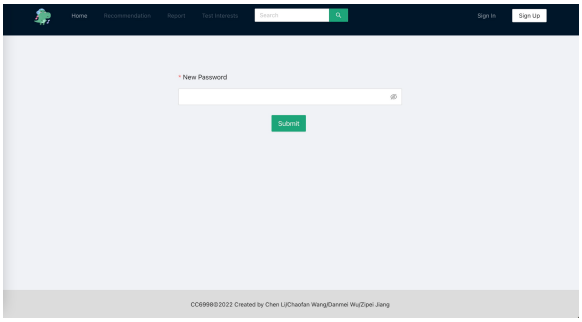


- SignUp Page

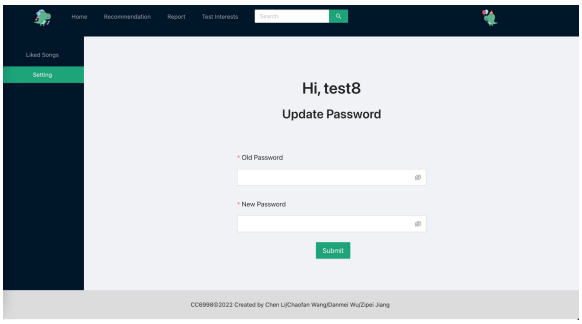


- Forget Password Page

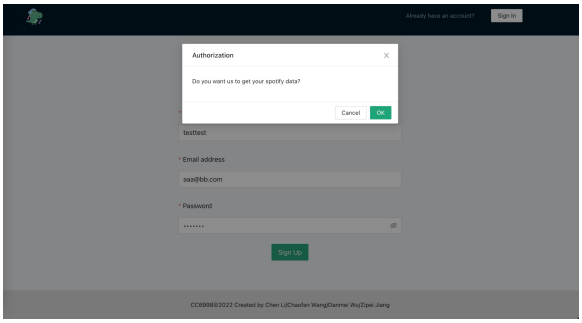




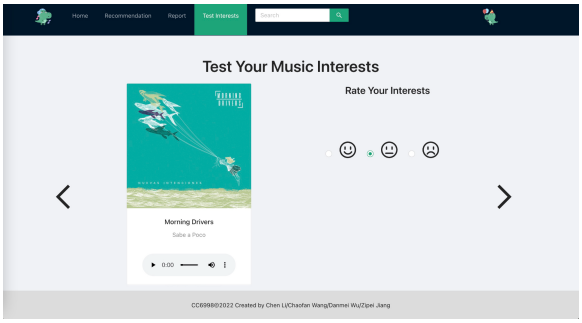
- Update Password Page



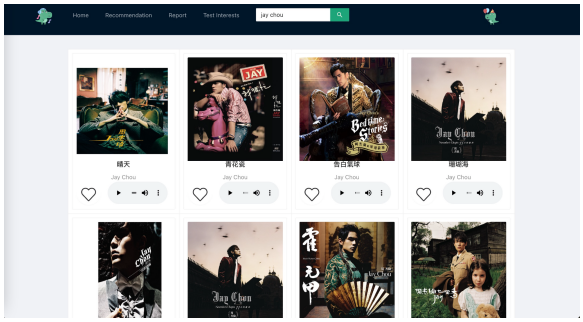
- Authorize Spotify Page

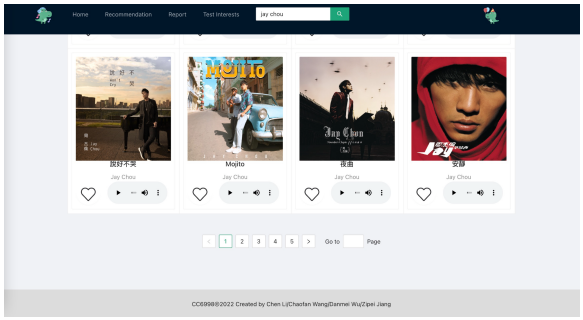


- Test Interest Page

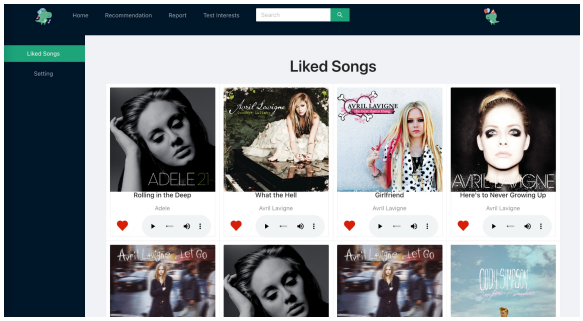


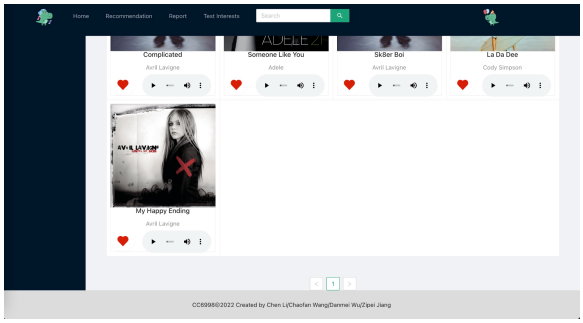
- Search Page



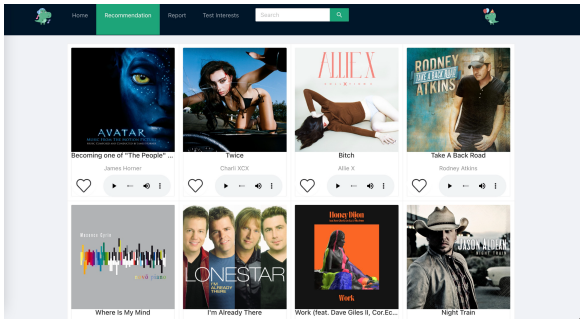


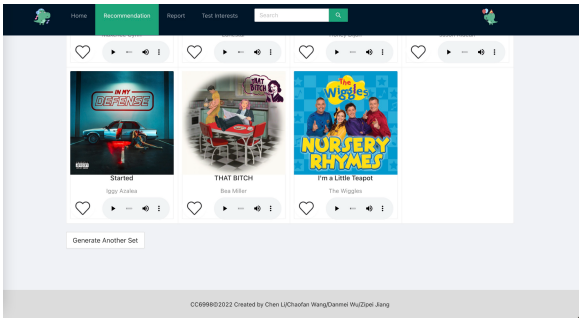
- Music Like Page



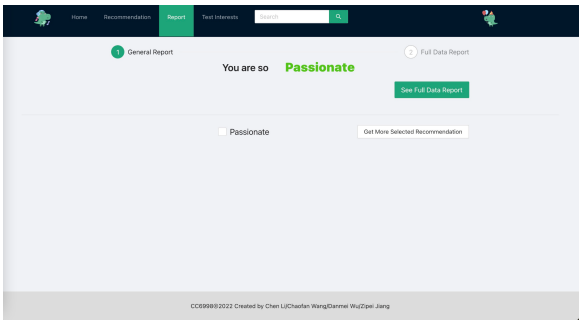


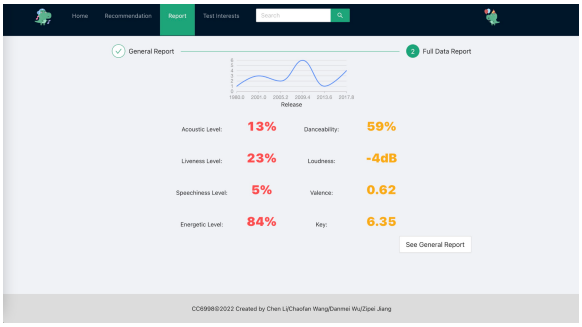
- Recommendation Page



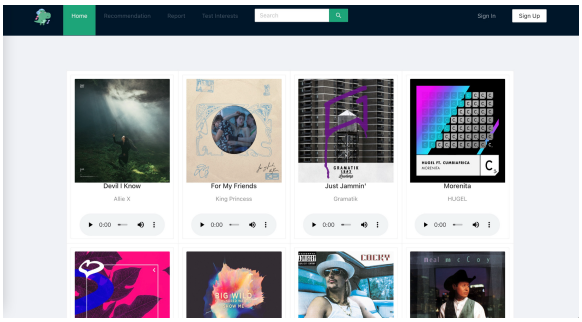


- Report Page





- Home Page



Recommendation System implementation

We construct a recommendation system based on song similarity. We define the similarity of two songs as the probability that they appear in a user's like list at the same time. We calculate the similarity of two songs by the cosine similarity of their embedding.

In order to generate embedding for each song that meets the above requirements, we build a self-supervised contrastive learning pipeline to train the deep encoder. After trying various encoders, we found that the best result can be achieved when user preference data is expressed in the form of a gigantic graph and graph neural network is chosen as the encoder.

- **User Song Similarity for Recommendation**

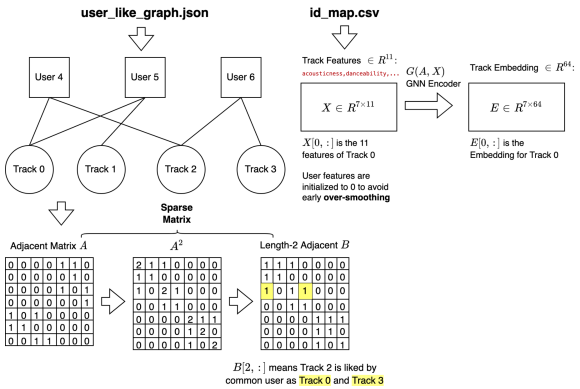
Initially, we generated recommendations using the similarity of users and songs. However, we soon found that such a structure could not be implemented in real time. Each time a new user appeared, we needed to generate embedding for the user, a process that required us to add the user to the huge graph and re-run the encoder throughout the graph. As the volume of users and songs increases, this graph may contain tens of millions of nodes with even more edges. This results in generating new embedding that takes a lot of time, so real-time is almost impossible. So we decided to bypass the user embedding and use only the song embedding. and periodically generate the embedding for newly added songs and store it in the database. When a user needs a recommendation, we take the embedding of his favorite song out of the database. And we find the songs from the embedding database that are most similar to the embedding of the user's favorite song as a recommendation. Such a recommendation approach allows us to perform time-consuming model operations offline and only query the database when making recommendations, thus achieving the goal of real-time recommendation.

- **Data format**

We use the following criteria to convert user preference data into graph structures.

1. Consider users and songs as nodes in the graph.
2. An edge may exist between a User node and a Song node if the song is liked by the user.
3. Define: Song-1 and Song-2 are neighbor if there exist at least one user, whose like list contains both Song-1 and Song-2

Considering the following mini example with only 3 users and 4 songs



Note that the second order adjacent matrix B naturally indicates if two songs are neighbors, which helps to reduce the sampling time during training.

- **Training**

The following algorithm explains how we sample self-supervised training data.

Sample Training Pair

1. Sample $a, b \in [4]$.
2. Calculate label $y = 2B[a, b] - 1$
3. Return Training example (a, b, y)
4. Query E to get embedding of a and b
5. Calculate $\text{Cos } Sim$ and $Loss$ with given loss function f

The following algorithm explains the implementation of contrastive learning to train the encoder G.

Training Pipeline (One batch)

1. Calculate Embedding $E = G(A, X)$
2. Sample BS training pairs T
3. For $data$ in T:
   $BatchLoss = BatchLoss + Loss(data)$
4. Backprop to train G

A positive training data example

$a = 2, b = 0.\ y = 2B[2, 0] - 1 = 1.$ sample = $(2, 0, 1)$

$x_1 = E[2, :], x_2 = E[0, :]$

$$Loss = f\left(\frac{x_1 \dot{x}_2}{|x_1||x_2|}, 1\right)$$

A negative training data example

$a = 1, b = 2.\ y = 2B[1, 2] - 1 = -1.$ sample = $(1, 2, -1)$

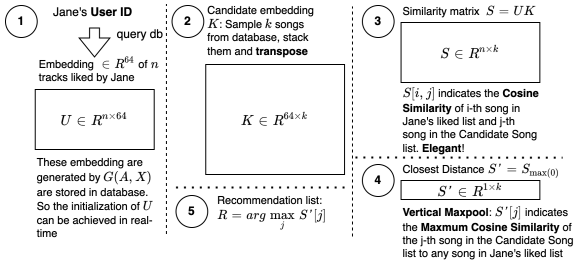
$x_1 = E[1, :], x_2 = E[2, :]$

$$Loss = f\left(\frac{x_1 \dot{x}_2}{|x_1||x_2|}, -1\right)$$

However, two songs are not neighbors in the graph doesn't mean they are necessarily different. This could only be caused by having insufficient user nodes in the graph. To accommodate this problem. We introduced a value p, which will adjust the positive and negative sample ratio. As more and more users join and update their likelist, the model will converge to real neighbor distribution.

- **Inference**

The following diagram shows how to do real-time inference.



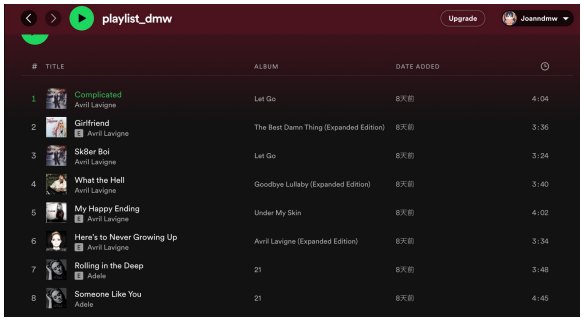
Evaluation

We collected a test dataset from the Spotify API containing over 1,000 users and the 24,000 songs they liked to test our model. When choosing a threshold of 0 (i.e. predict two songs are neighbors if the cosine similarity between their embedding is larger than 0 and not neighbor otherwise,) the embedding generated by the trained encoder was able to correctly determine whether two songs were neighbors 76.3% of the time.

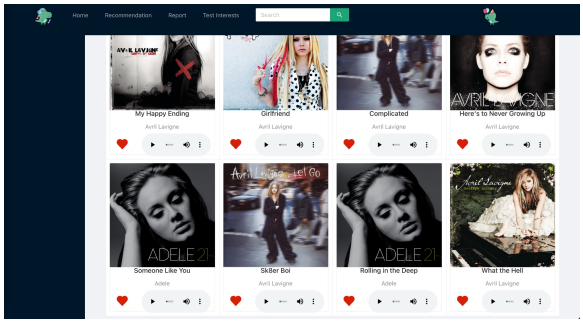
- **Comparison with Spotify's recommendation**

Because we don't know the model data pipeline of spotify's own recommendation algorithm, we can't generate a corresponding test dataset or find a suitable metric for testing. Therefore, we compared the recommended song list of a user's spotify with the recommended song list of our model

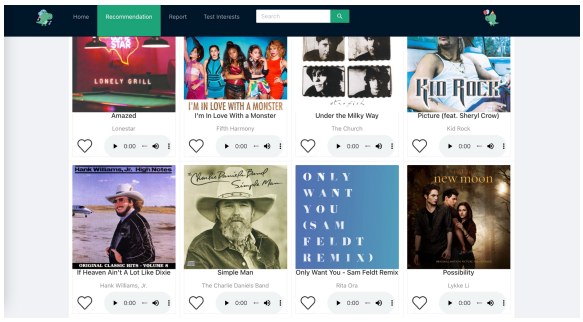
Spotify Account music:



According to our liked song list:

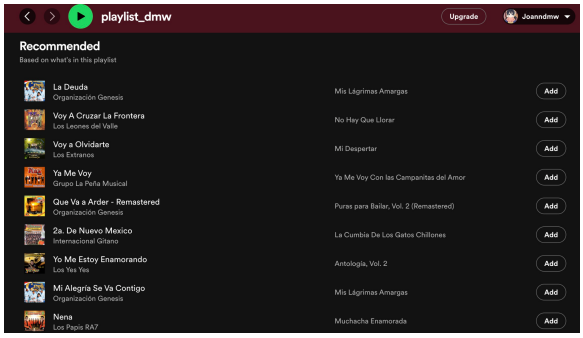


Our Recommendation system gives:



We can see that our playlist has a tendency toward passionate music, and our system also gives us some similar rhyme music.

Comparison with Spotify Recommendation:



We can find that our recommendation is much better than Spotify based on this playlist. Spotify recommends some weird songs, not even in English.

Conclusion

Our service uses artificial intelligence models to provide users with fast and accurate music recommendations. Users can either authorize logging into their Spotify account to get recommendations based on their playlist, or they can get recommendations by doing a quick and easy hobby test.

For each song users get a 30 second demo. Users can say they like or dislike any song at any time. Any comment a user makes about a song is immediately taken into account by our model in the next recommendation.

As the number of users increases, our model will evolve as the training data will get closer to the real population. Users can search to get songs directly from spotify and add them to their favorite list. If the added song does not exist in our recommendation model, the model will generate embedding for the added songs in regular offline data processing. After that, the newly added songs will be taken into account in the recommendation generation.

We bypass the need to run the model in real time for each recommendation by using the similarity between songs directly to achieve real-time recommendation, so that the speed of recommendation generation is not affected by the increase of data volume. Even if the number of songs in the complete graph grows to tens of millions, the recommendation system can still achieve real-time recommendation if the database is sufficiently complete.

Also, we are able to show users the analysis of their music preferences. And some keywords are proposed in the analysis interface, and users can check these keywords to get the corresponding music recommendations.

Important Links

Website: http://project-frontend222.s3-website-us-east-1.amazonaws.com/build2/index.html
Figma Demo: https://www.figma.com/file/5OYYLoVNKwZsM9lt2gJLu9/demo?node-id=0%3A1
FrontEnd: https://github.com/JoanWu5/cc-music-recommendation-frontend
Recommendation System: https://github.com/CChenLi/Music-Recommend-Model
BackEnd: https://github.com/zipeijiang/Music-Recommend-Lambda
Youtube link: https://www.youtube.com/watch?v=Ybs6k62LuSU

Reference

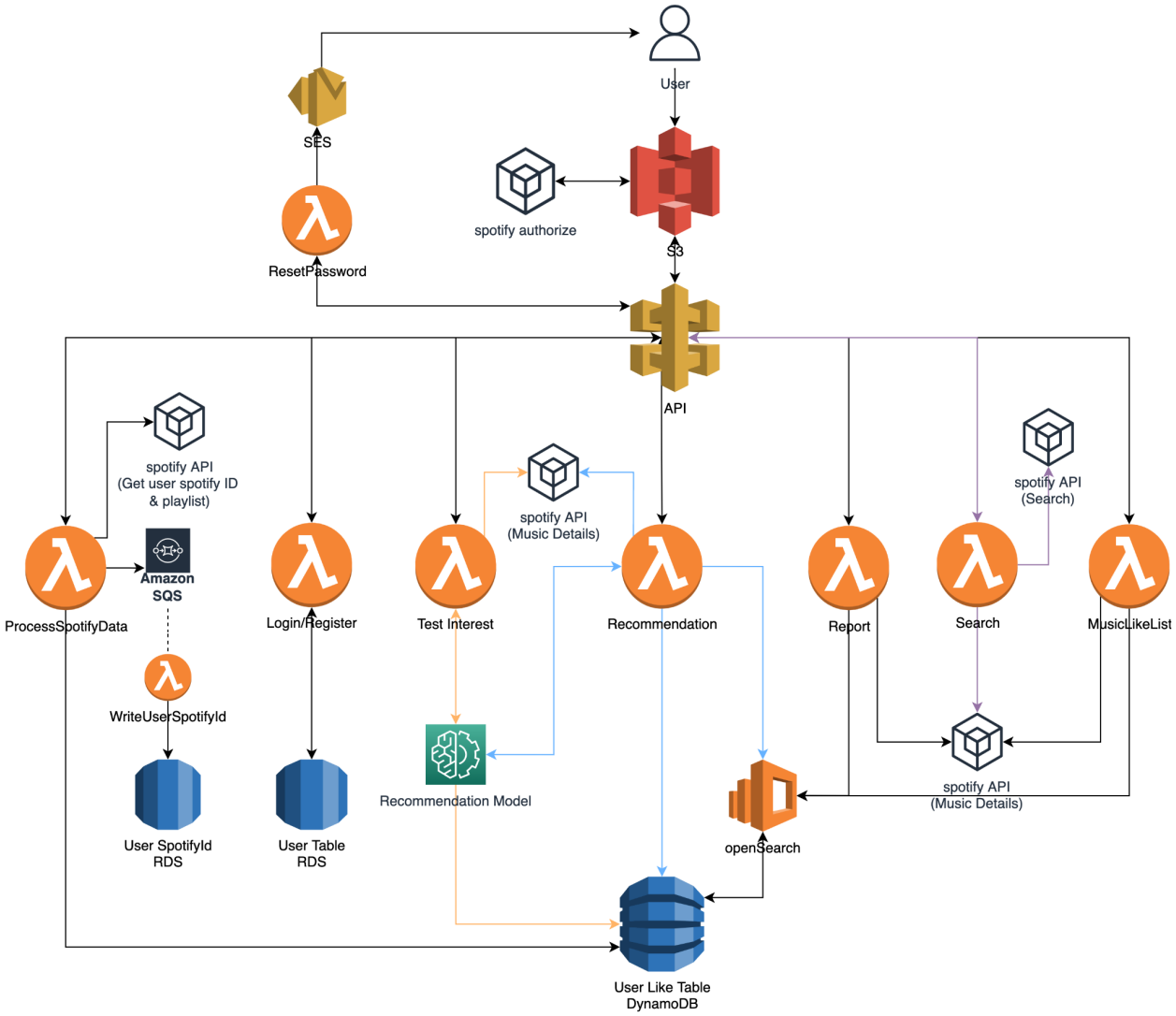


Figure 3.1 Architecture Diagram