
New York University

CS-GY 9223 A

# Large Scale Music Recommender Web App


| *Name:* | *NetId:* |
|---|---|
| Yi Zhou | yz4029 |
| Suyang Jiang | sj2364 |
| Zetong Wang | zw1465 |
| Xioajie Shi | xs857 |

# Contents

# 1 Overview

We implemented a music recommendation web app based on Amazon Web Services. Basically, we implemented three main parts. One part is music recommendation based on user behaviors using PySpark MLlib. The second part is a chatbot named Tara. User can chat with it to listen to music and get music recommendations using LastFM dataset and Spotify API. The third part is search function. User can search for specific songs, musician and tags.

# 2 Architecture

The following picture is the architecture of our back-end and all AWS services we used.

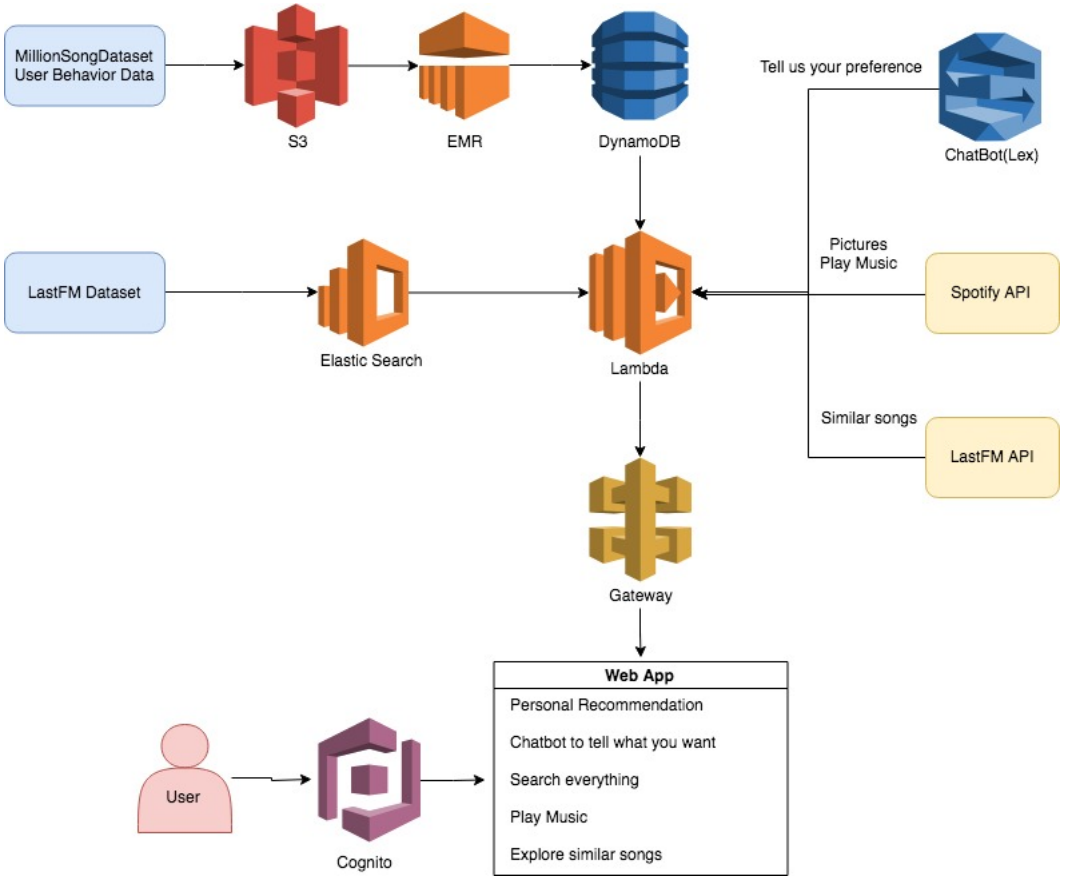


Figure 1: Architecture

Amazon Web Services: Lambda, S3, Cognito, API Gateway, Lex, Elastic Search, EMR, DynamoDB

# 3 Process Large Datasets and Recommend

In this project, two large size data set are used. One is taste profile subset[1], which has 1,019,318 unique users, 384,546 unique MSD songs and 48,373,586 user-song-play count triplets. We use this file as the user-behavior dataset to analyze user preference and give recommendations. The other is lastFM dataset [2], which includes information of 943,347 tracks. We use the first dataset to analyze user behaviors ad give recommendations, and use the second dataset to search related informations about the songs.

The Last FM tracks and MSD songs can be matched using a matching file from taste_profile_song_to_tracks.txt [3] provided by Lab ROSA.

## 3.1 Recommend Using PySpark MLlib

Since the dataset is large, we deal with the user behavior in a scalable way. We first upload the user behavior file into S3, then launch a AWS EMR cluster to load the data from S3 and do analysis.

Our user behavior data is user-song-play count triplets. Since it's implicit feedback data, we use the implicit feedback collaborative filtering method to train our model and then give every user a recommend song list.

In detail we use pyspark.mllib.recommendation.trainImplicit to do a collaborative filtering on the behavior and then save recommendations as many generated text files on S3. After that, we load the recommended list to DynamoDB for future usage.

## 3.2 Load Large Datasets into DynamoDB

Loading small datasets into the DynamoDB is easy, but loading large datasets is kind of tricky. We succeed to load large dataset into DynamoDB efficiently and without error by using batch write API. The batch write API in DynamoDB is fast

[1]Taste profile, https://labrosa.ee.columbia.edu/millionsong/tasteprofile
[2]Lastfm Dataset, https://labrosa.ee.columbia.edu/millionsong/lastfm
[3]taste_profile_song_to_tracks.txt: https://www.kaggle.com/c/msdchallenge/data

but also easy to fail. So we log all the failures and use item-wised write to backup this failures. That's because the item-wised write is stable although slow. So by taking advantages of these two types of write APIs, we can write into DynamoDB in an efficient way without errors.

One feature of our app is that we use short encoding for user_id and track_id for the purpose of saving storage. The origin id is a 40-character-strings, and we map them into 6-character-strings so that we can save a lot of storage when we have to store a lot of ids as recommendation results in our DynamoDB.

Another trick here is to set "write capacity units" in DynamoDB to be very large for a short amount of time period when you are uploading the dataset. And after you finished, you can set it to be a small value so that it won't cost much. This can help you to avoid most of errors when you are writing to the DynamoDB. For us, we set "write capacity units" to be 150 when loading our data.

# 4 Search

## 4.1 Load LastFM into Elasticsearch

In our web application there is a search box where user can search songs by song's title, musician's name and the tags. This is achieved by using AWS Elasticsearch Services.

The original dataset is a csv file which is not suitable for ElasticSearch, so we generate a 943,347 json entries from the original LastFM dataset. For the purpose of saving time and stability, we split them into many smaller batch files. And then we post them into our Elasticsearch Instance in AWS.

## 4.2 Search APIs

We provide three search APIs in our backend. The first API is /get-track-info-by-track-id , which enable users to search song information by songId. By using this API, we can search the information of target song by its id. The second one is the /get-track-info-by-tag, which enable users to search song information by tag. So we can show related tags in our web app, and when user click a tag, he will be able to explore the songs under this tag. The third one is /general-search, which enable the search box in our app. We use "multi-match" function of the Elasticsearch to match

artist, title and tags. So that when user type the keyword in the search box, he can get the results he want as long as the keyword is included in the LastFM dataset.

# 5 Chatbot

## 5.1 Overview

An intelligent chatbot is embedded in our website. The chatbot name is Tara. She basically provides three functions, play music, recommend music, have short conversation. The chatbot mainly depends on the AWS service, such API Gateway, S3, Lambda, lex etc. The procedure of the chatbot roughly as following: the lex get the input of users, and pass the value to lambda function. After getting values, the lambda function first calls the LastFM API to get similar songs of the searched keyword. Then it will pass the similar songs to the Spotify API to get information of the recommended music, for example image url, preview url, musician name etc. Finally, the lambda function format the information and return to the front end.

## 5.2 Amazon Lex

For the lex part, We mainly create Greeting Intent, Play music Intent, Song Recommendation Intent, Conversation Intent, ThankYou Intent. For each intent, we train the bot, such give it utterance, slot value, response and so on.

## 5.3 Dependency

For the lambda function connected with lex, We request LastFM and Spotify API to get all information we want. We parse the response from these API. Then format them into more elegant style. However, there maybe some value in these response is null. If the recommend list is null, We provide compatible warning. For some songs, Spotify does not provide the preview URL, We assign these URL to a 404 page.

## 5.4 Server-less

Because we use the third party library. The whole development is using the serverless framework. Server-less is a toolkit for deploying and operating serverless architectures. Serverless help us to develop, test and deploy in a single environment. We

first create our local serverless service. We code all lambda functions locally, and edit configure file to every lambda functions. After completing coding, I install the virtual environment. Virtualenv is a lightweight, self-contained Python installation. It allows us to make isolated python environments. In the virtual environment, We install all dependencies into it. Finally, at this stage, our function is working locally. So we add requirements to the serverless.yml and deploy them to Amazon Web Service.

# 6 Integrating with Spotify and LastFM API

LastFM API has the feature that it collected the physical feature of most of songs. It is extremely suitable for our recommendation based on items. But the LastFM does not provide relevant information of songs. In order to get information of songs. We use another API named Spotify. The Spotify API has large and detailed information of songs. So all the visual information are fetched from it. We use these two API for different purpose of providing both visual as well as phonic experience to our users.

# 7 User Credential

We use AWS Cognito to provide user credentials to our users. Amazon Cognito provides authentication, authorization, and user management for web and mobile APP. After user login, Cognito will include a code as a query-string parameter in the redirect URI. Then we use the code to exchange for an Access and ID Token. After that, we can get the identity ID from AWS.config. The identity ID can be used to identify a unique user. So we use the information to give recommendations to our users.

# 8 Front End

## 8.1 Sign in and Sign up

User can sign in and sign up before enjoying the music on our website. Note that, for now, we the signing up is not yet perfect, because we need to match the credential with our database to give recommendations.

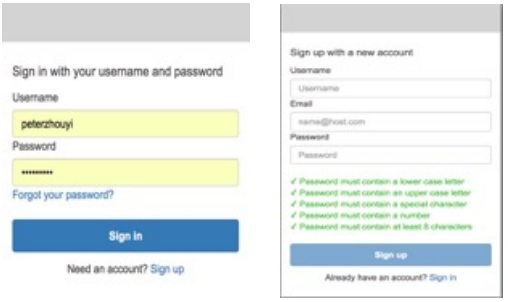


Figure 2: Sign In and Sign Up Page

## 8.2 Home Page

When a user log in, the home page will show up. In this page, we have three parts. The first part is the chatbot named Tara. User can click it to chat with her and get music recommendations. The second part is search box on the top. User can search for musicians, songs, and tags. The third part is recommendations for users which based on user's behavior. We also provided the most popular music nowadays for different music types.

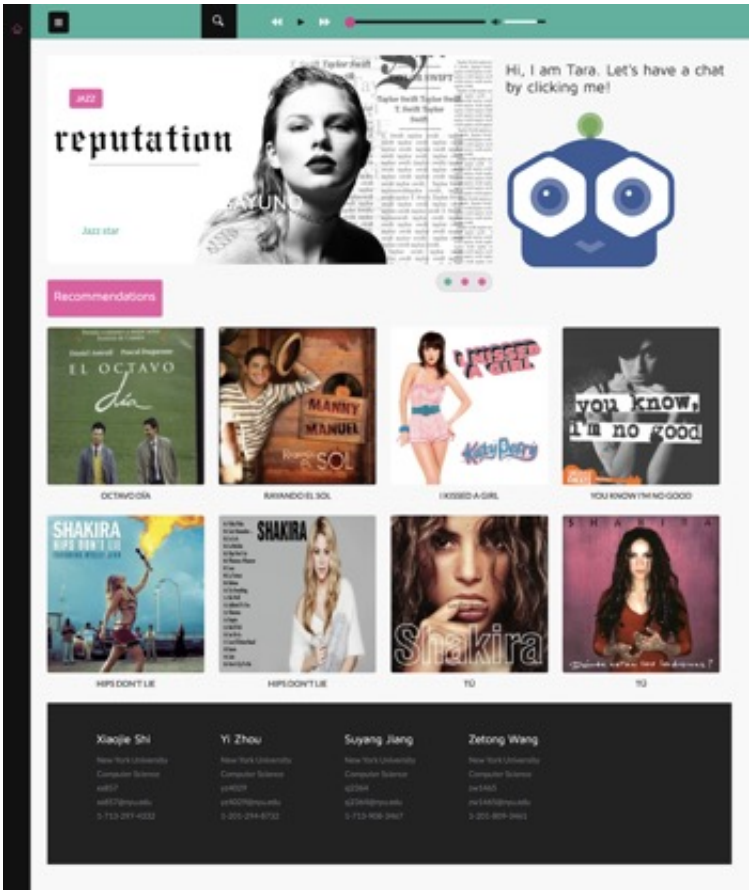


Figure 3: Home Page

## 8.3 ChatBot Page

The following image is our ChatBot page. User can chat with our ChatBot to play music and recommend similar music. After user offers the music name and artist to our ChatBot for recommendation, 'Tara' will give back the similar songs' preview for 30 seconds.

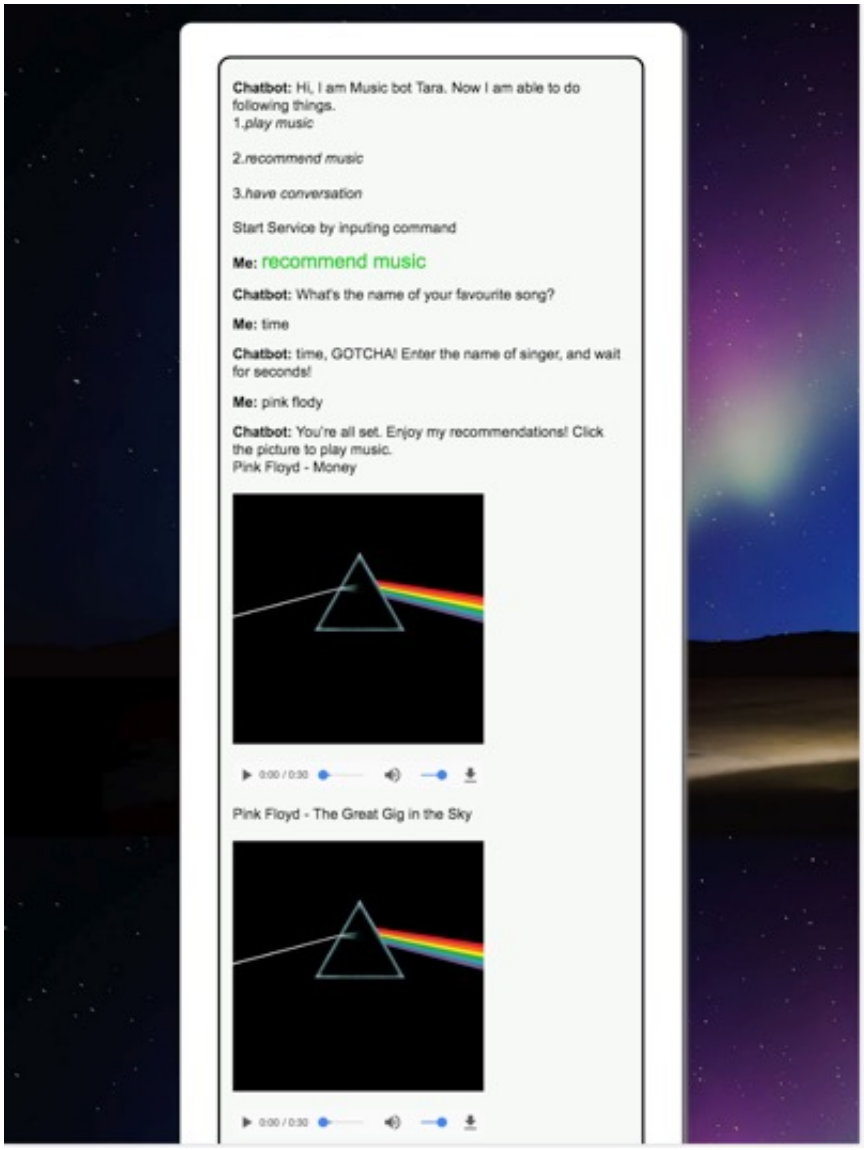


Figure 4: ChatBot Page

## 8.4 Search Page

Search results will show on this page. User can click one of them to play and jump to play page to enjoy the music.

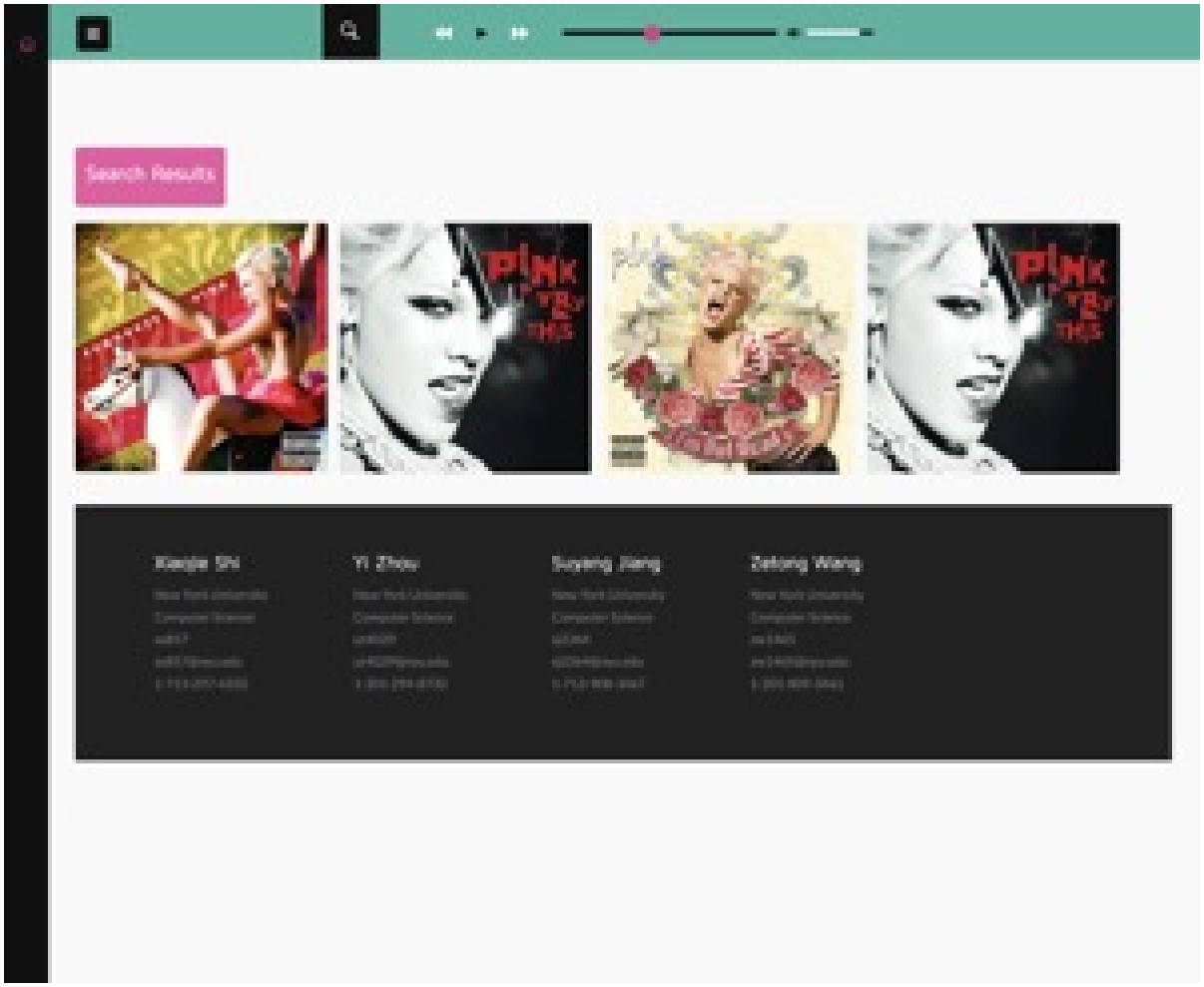


Figure 5: Search Page

## 8.5 Play Page

The following is the play page. User can enjoy the music and click the button on the bottom to see similar music.

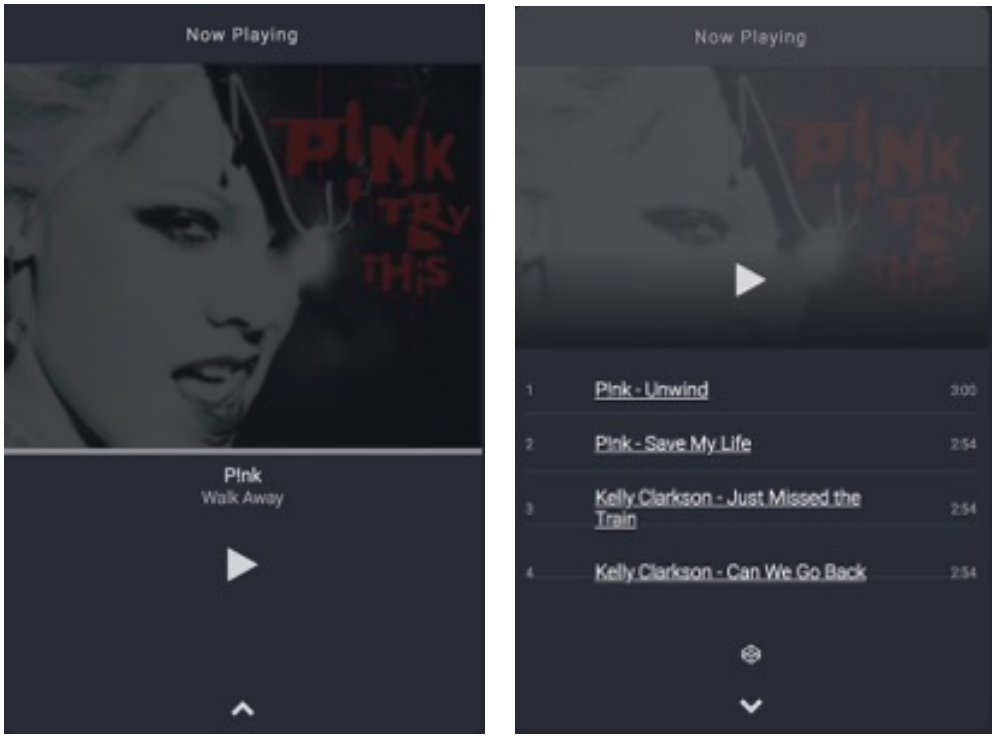


Figure 6: Play Page

# 9 Code Details

## 9.1 Front End Code Details

This is the JavaScript we use to handle the information from back end.

```
function getSongs(searchList) {
let counter = 0;
searchList = searchList.slice(0, 4);
return new Promise((resolve, reject) => {
    searchList.forEach(search => {
        if (counter >= 4) return;
        let params = search.title + "," + search.artist;
        BotRequest(params, (err, result) => {
            if (err) {
                console.log(err);
            } else {
```

```
                    let data = JSON.parse(result.data.messages[0].text);
                    console.log(data);
                    let similarList = data[0];
                    let infoList = data[1].split(",");
                    let song = infoList[0].split("-")[0].trim();
                    let artist = infoList[0].split("-")[1].trim();
                    let info = {
                        song: song,
                        artist: artist,
                        imageUrl: infoList[1],
                        playUrl: infoList[2]
                    };
                    songs.push(info);
                }
                counter++;
                if (counter === 4) {
                    resolve(songs);
                }
            })
        })
    })
}
```

## 9.2 Collaborative Filtering Recommend: Implicit Feedback

This is the python script we use to to generate to recommended songs to our users. First we read the user behavior data from the S3, then we use transform the index. At last, we train an implicit feedback model and then give recommendations to our customers. When all the calculations are finished, we save the index, the recommend list into S3.

```
from csv import reader
from pyspark.mllib.recommendation import *
from pyspark.sql.functions import format_string

tuple_path = "s3://million-song-dataset-yizhou/TasteProfile/train_triplets.txt"

df = spark.read.load(tuple_path, format="csv", sep="\t",
                                 inferSchema="true", header=None)
```

```
# Transform index
from pyspark.ml.feature import StringIndexer
user_indexer = StringIndexer(inputCol="_c0", outputCol="user_index")
song_indexer = StringIndexer(inputCol="_c1", outputCol="song_index")
partial_indexed = user_indexer.fit(df).transform(df)
indexed = song_indexer.fit(partial_indexed).transform(partial_indexed)
indexed.createOrReplaceTempView("indexed")

res_df = spark.sql("select user_index, song_index, _c2 as click from indexed")
res_df = res_df.select(format_string("%.0f,%.0f,%d",res_df.user_index,
                                     res_df.song_index,res_df.click))
rdd = res_df.rdd.flatMap(list).map(lambda x:x.split(","))
model = ALS.trainImplicit(rdd, 25, seed=10)

# Generate userIndex
temp1 = spark.sql("select distinct _c0,user_index from indexed")
user_df = temp1.select(format_string("%s,%.0f",temp1._c0,temp1.user_index))
user_df.write.save("s3://million-song-dataset-yizhou/TasteProfile/userIndex",
                                     format="text")

# Generate songIndex
temp2 = spark.sql("select distinct _c1,song_index from indexed")
song_df = temp2.select(format_string("%s,%.0f",temp2._c1,temp2.song_index))
song_df.write.save("s3://million-song-dataset-yizhou/TasteProfile/songIndex",
                                     format="text")
# Get recommends
recommends = model.recommendProductsForUsers(20)

# Transform
def extractMapper(x):
    user, ratings = x
    products = list(map(lambda y:y.product, ratings))
    return user, products
def formatMapper(x):
    user, products = x
    products = list(map(str,products))
    return str(user) + "\t" + ",".join(products)

recommends.map(ExtractMapper).map(formatMapper).\
  saveAsTextFile("s3://million-song-dataset-yizhou/TasteProfile/recommendList")
```

## 9.3 Load large dataset into DynamoDB

We implement a putItem function and putBatch function. When in most cases, we use putBatch to load the data every time we find a failure, we record the failure and then backup them when we logged 5 failures using putItem function. Also, we auto extend the time gap so that we can avoid future failures.

```
# putItem
def putIntoDB(x):
    response = table.put_item(
        Item = {
            'user_id': x[0],
            'user_index': x[1]
        }
    )
    return (x[0],response)

# putBatch
client = boto3.client('dynamodb')
def putBatchIntoDB(xlist):
    def transform(x):
        out = dict()
        out['PutRequest'] = dict()
        out['PutRequest']['Item'] =  dict()
        out['PutRequest']['Item']['user_id'] = dict()
        out['PutRequest']['Item']['user_index'] = dict()
        out['PutRequest']['Item']['user_id']['S'] = dict()
        out['PutRequest']['Item']['user_index']['S'] = dict()
        out['PutRequest']['Item']['user_id']['S'] = x[0]
        out['PutRequest']['Item']['user_index']['S'] = x[1]
        return out
    putList = list(map(transform,xlist))
    response = client.batch_write_item(
        RequestItems={
            'UserIdToIndex': putList
        }
    )
    unsuccessful = response['UnprocessedItems']
    if len(unsuccessful) == 0:
        return []
    else:
```

```
        return unsuccessful['UserIdToIndex']

usList = []
times = 1.0 # Auto extending the time gap when failing too many times
for fileName in fileList:
    time.sleep(1.0 *times)
    us = putBatchUserIndexFile(fileName, batchSize=25, times = times)
    if len(us):
        usList.append(us)
    if len(usList) >= 5:
        print("------------------------Backing Up----------------------------")
        time.sleep(5*times)
        Backup(usList)
        print("--------------------------------------------------------------------")
        usList = []
        times *= 1.1
```

## 9.4 ChatBot Recommendation

We implement the recommendation based on item by using python, we get the information of songs and format them put back to front end.

```
def song_suggestion_response(intent_request):
    song_name = get_slots(intent_request)['SongName']
    musician_name = get_slots(intent_request)['MusicianName']
    logger.debug(song_name + musician_name)

    if validate_info(song_name, musician_name):
        similar_songs_list = get_similar_songs(song_name, musician_name)
        recommendations_list = get_name_img_preview_url(similar_songs_list)
        print(recommendations_list)
        html_code = get_content(recommendations_list)

        if html_code is not None:
            return {"dialogAction": {
                "type": "Close",
                "fulfillmentState": "Fulfilled",
                "message": {
                    "contentType": "PlainText",
                    "content": "You're all set. Enjoy my recommendations! /
```

```
                    Click the picture to play music.{0:s}".format(
                        html_code)
                }
            }}
        else:
            return {"dialogAction": {
                "type": "Close",
                "fulfillmentState": "Fulfilled",
                "message": {
                    "contentType": "PlainText",
                    "content": "Whoops! I can not find that. Please check your inpu
                }
            }}
```

## 9.5 Configure virtual environment

```
npm install -g serverless
serverless create --template aws-python3
virtualenv venv --python=python3
source venv/bin/activate

provider:
  name: aws
  runtime: python3.6

functions:
  numpy:
    handler: handler.main

npm install --save serverless-python-requirements
```

## 9.6 Elastic Search

We support our search box in our web app by using the multi_match query in the elastic search.

```
def generalSearch(uri, term):
    """Simple Elasticsearch Query"""
```

```
query = json.dumps({
    "query": {
    "multi_match" : {
        "query":  term,
        "fields": [ "artist", "tags", "title" ]
        }
    }
})
headers = {'Content-Type': 'application/json'}
response = requests.get(uri,headers=headers, data=query)
results = json.loads(response.text)

return results
```