Dissertation on

Music Recommendation System using RNN

Submitted in partial fulfilment of the requirements for the award of degree of

Bachelor of Technology
in
Computer Science & Engineering

Submitted by:

Shashank Saran	**01FB15ECS278**
Siddharth Ganesan	**01FB15ECS290**
Varun V.	**01FB15ECS331**

Under the guidance of

Internal Guide
Samatha R. Swamy
Professor,
PES University

January – May 2019

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING
FACULTY OF ENGINEERING
PES UNIVERSITY
(Established under Karnataka Act No. 16 of 2013)

100ft Ring Road, Bengaluru – 560 085, Karnataka, India

PES UNIVERSITY

(Established under Karnataka Act No. 16 of 2013)
100ft Ring Road, Bengaluru – 560 085, Karnataka, India

FACULTY OF ENGINEERING

CERTIFICATE

This is to certify that the dissertation entitled

Music Recommendation using RNN

is a bonafide work carried out by

Shashank Saran	**01FB15ECS278**
Siddharth Ganesan	**01FB15ECS290**
Varun V.	**01FB15ECS331**

In partial fulfilment for the completion of eighth semester project work in the Program of Study Bachelor of Technology in Computer Science and Engineering under rules and regulations of PES University, Bengaluru during the period Jan. 2019 – May. 2019. It is certified that all corrections / suggestions indicated for internal assessment have been incorporated in the report. The dissertation has been approved as it satisfies the 8th semester academic requirements in respect of project work.

Signature Samatha R. Swamy Professor	Signature Dr. Shylaja S S Chairperson	Signature Dr. B K Keshavan Dean of Faculty

External Viva

Name of the Examiners	**Signature with Date**
1. _____	_____
2. _____	_____

DECLARATION

We hereby declare that the project entitled **"Music Recommendation System using RNN"** has been carried out by us under the guidance of Prof. Samatha R. Swamy and submitted in partial fulfilment of the course requirements for the award of degree of **Bachelor of Technology** in **Computer Science and Engineering** of **PES University, Bengaluru** during the academic semester January – May 2019. The matter embodied in this report has not been submitted to any other university or institution for the award of any degree.

01FB15ECS278	**Shashank Saran**	_____
01FB15ECS290	**Siddharth Ganesan**	_____
01FB15ECS331	**Varun V.**	_____

ACKNOWLEDGEMENT

Firstly, we would like to thank our mentor and guide, Mrs. Samatha R. Swamy. The inputs provided in terms of project progress, methods of inference and documentation were immensely valuable and provided great insight. We would like to thank her for guiding us throughout this internship period.

During the various presentations, the project coordinators, Ms. Preet Kanwal and Ms. Sangeetha V., helped us align in context with all the presentations, both the way the project was implemented and the way the seminars were organized.

We would also like to thank our Chairperson for being a source of motivation to all the students, not just to us. Dr. Shylaja helped in easing the process of progress of presentations and made sure the schedules were clean and effective.

We would also like to thank the dean of faculty for helping with effective and time saver schedules. We would also like to thank Vice Chancellor, PESU, Dr. K.N.B Murthy who helped us with smooth on boarding throughout the process of the final year and for being supportive.

We would also like to acknowledge the contributions of Prof D. Jawahar, Pro- Chancellor, PESU for always motivating and guiding us throughout the course of engineering, for having great vision and providing to quality education to students. We would also like to express our gratitude to the Chairman of PESU, Dr. M. R. Doreswamy for instilling in values of perseverance, excellence and service and help in learning the meaning of various virtues and practice them and apply them in real life.

We would like to give our gratitude to our parents for supporting us, being understanding and helping us to ease our minds even during stressful times. Lastly, we would like to thank our friends, who not only helped us solve issues we were facing but also to keep us smiling through trying times.

ABSTRACT

Music is a vital part of human culture. Listening to music not only allows people to express their personality, but also has several medical benefits. But, how do you pick the music you like, especially when there are millions of songs available to choose from? We can't possibly go through all these songs to choose the few we like! This is where recommender systems come in. They pick a playlist of songs that you might like and serve them up for you to enjoy. However, sometimes these songs are not to our liking or we're just not in the mood to listen to those kinds of songs. This is why we have come up with a music recommender system that analyses what your current listening preferences are based on your playlist and suggest new music according to that. This done with the help of the powerful encoder-decoder neural network architecture, which is primarily used to deal with sequential data. Treating playlists as sequential data rather than simply a set of songs allows us to generate more personalized playlists that capture hidden factors such as mood and commonalities between songs in a playlist. The goal of the project is to be able to take a playlist as input and generate a playlist as output which contains songs which are similar to those given in the playlist. A simple UI will supplement the recommender system as well.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER-1

INTRODUCTION

Recommendation systems have become a vital part of today's technology. They are being used in all fields, from online shopping to bookstores to news feeds, and have become omnipresent in most online applications. They filter items for user's, suggesting only those items that the user would be most likely to purchase or use, saving users the hassle of going through all available items to get what they need.

One key application of recommender systems is in music suggestions. They help users discover new music to listen to. There are several techniques that can be employed to identify what kind of music users might enjoy. The most primitive of techniques was to suggest songs by the same artists that the user has already listened to. This method, while viable, had several drawbacks. One drawback was that no songs from different artists would ever show up in the recommendations. Furthermore, not every song from the same artist may be similar to the other songs of the artist. Another technique was to rank songs by their popularity and suggest the top K most popular songs. While this technique addressed the issues of the above method, it is unable to make useful suggestions for those with niche tastes in music or show music of unpopular artists. Clearly, better techniques were required.

With advancements in big data and machine learning, several new options opened up for recommender systems. The newer techniques can be divided into 2 main categories [1]:

1. *Collaborative filtering-based recommender systems*
2. *Content based recommender systems*

Collaborative filtering systems make suggestions based on analysing commonalities among different user's item history [4]. If A liked an item that B also liked, then any other item that B happened to like will most likely be liked by A as well and will be suggested to A. Thus this technique overcame several issues faced by the earlier techniques. For one, these suggestions were more accurate. The songs that were suggested were not affected solely by same artists or popularity but by popularity among a chain of people with similar tastes in music.

Furthermore, the system does not need to know the nature of items themselves and can easily be adapted for any type of items. One necessity for such systems, is some form of scoring or rating of items (can be explicit, i.e., manual rating of items on a scale or implicit, such as time spent on viewing an item etc.). This tends to become a drawback in environments where explicit or implicit scoring is either not available or difficult to obtain. Another key problem faced by these systems is the *cold-start* problem where newer items or unpopular items are not suggested as they do not have an adequate rating history [2]. This affects users who have a very niche listening history. This problem usually occurs in the beginning when the rating matrix is too small to give useful results. Moreover, as the system does not actually know the nature of items, suggestions are not very personalized. Even scalability and sparsity are big issues since computational needs increase greatly as the user matrix grows. As an example, Facebook uses this technique to suggest friends to users.

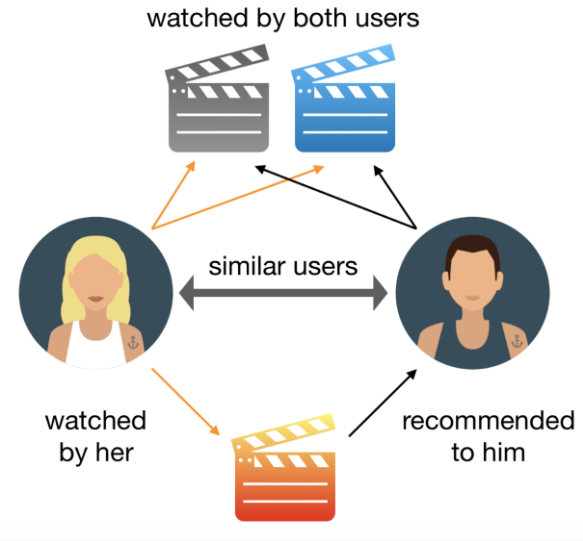

Fig 1.1 Collaborative Filtering recommender systems

Content Based systems, in contrast, work by analysing the nature of items and trying to identify patterns among items in the user's history to make suggestions. This makes choosing of item features a key step in such systems. Choosing the right features of items is thus very important. Traditionally, machine learning algorithms such as Support Vector Machines and K-Nearest Neighbours have been used to make suggestions. Content based recommenders do not suffer from the cold-start problem and can suggest even new or unpopular songs, thereby resulting in more personalized suggestions. However, Collaborative Filtering based

recommender systems generally tend to outperform CB recommender systems when the volume of data available is large [1]. Most of these systems also tend to use song metadata (artist, genre etc.) to make predictions. Pandora is an example of a system that uses CB music recommendations.

There also exist hybrid systems that combine suggestions from both the above mentioned approaches, striking a balance between personalization and. Netflix uses such a system to make TV show recommendations.[1]

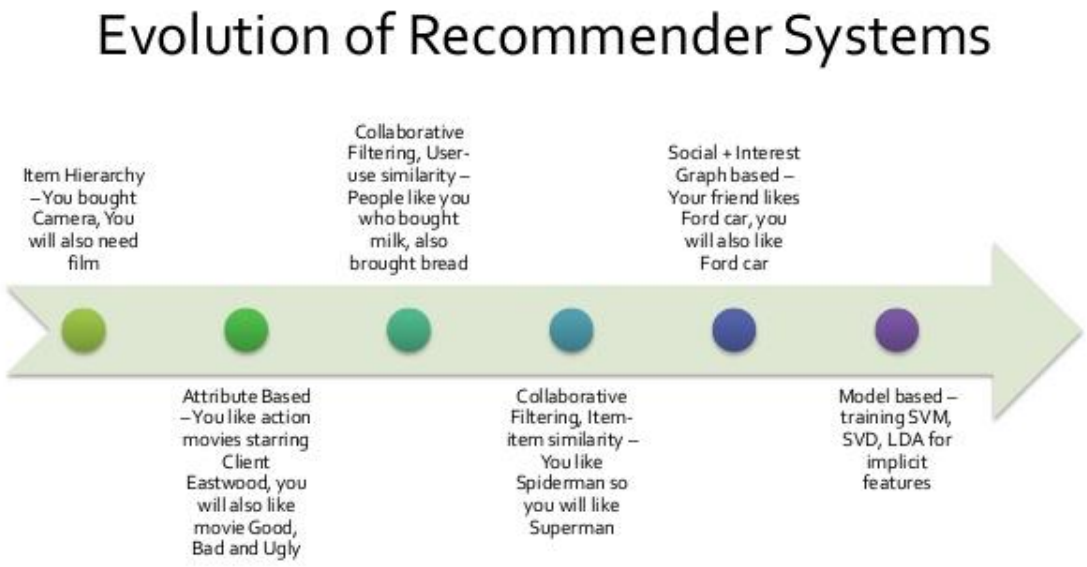

Fig 1.2 Evolution of Recommender Systems

The approach we are suggesting improves upon the current CB recommender systems. We wanted to see if we could improve the performance of these systems by using more modern deep learning approaches rather than traditional ML techniques while preserving the system's ability to give personalized results. This is why we chose to implement a CB system using the power of neural networks. More specifically, we employed Recurrent Neural Networks to implement our system.

RNNs are used to capture dependency in sequences of data. This is done by having a unidirectional connection between hidden units of the same layer. Their main application comes in analysing text data, as sentences tend to have a sequential nature. If we think of

songs in a playlist as items in a sequence, we find that we can capture several time dependent or contextual factors such as the user's current mood, genre based on other songs in the list etc. This is why we opted for RNNs over regular neural networks. RNNs, on their own, suffer from something known as the "vanishing gradient" problem. What this means is that as sequence length grows, the learning of the system gradually decreases. To remedy this, we will be using a variant of RNNs called LSTMs.[6]

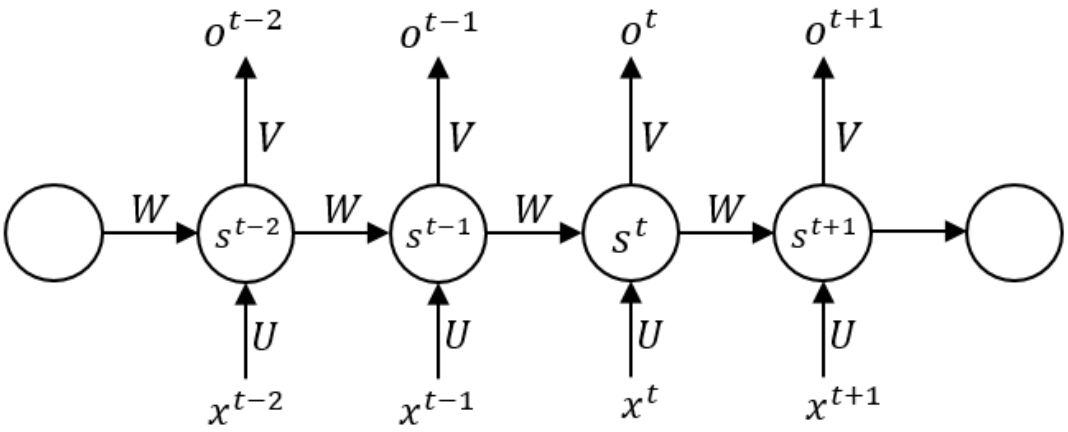

Fig. 1.3 Arrangement of hidden units in RNNs

We will be using a model known as an encoder decoder architecture which comprises of 2 LSTM networks joined together. Our training samples are playlists containing songs. The output of this network will be another set of songs.

CHAPTER-2

PROBLEM DEFINITION

The aim of this project is to build a recommender system using more modern deep learning techniques. While there are several ways of implementing recommender systems out there, we aim to improve performance of CB recommender systems using RNNs while preserving the personalization these systems provide. There are three major phases of this project that need to be carried out:

1. Finding a suitable dataset to use for training the model and pre-processing it for the neural network to use.
2. Designing and implementing the model that will be used to make recommendations.
3. Creating a UI that will allow user interaction and integrating the results from the recommender to this frontend.

The main idea here is to provide an alternative to current recommender systems and see how this one performs.

CHAPTER-3

LITERATURE STUDY

3.1 "Current challenges and visions in music recommender systems research(2018) - Markus Schedl , Hamed Zamani , Ching-Wei Chen, Yashar Deldjoo , Mehdi Ela"

As the title suggests, this paper discusses the present challenges faced by MRS and outlines the future possibilities for these systems. It begins by discussing the particularities of musical data as opposed to other data such as movies and shopping items, stating that musical data is more sequential in nature, tends to be listened to passively, is a very large-scale collection and is emotion dependent. It then states the 3 major challenges faced by such systems:

- Cold-Start problem ⇒ Making recommendations is difficult in the beginning due to lack of initial user data. A sub problem of this is the sparsity problem where the number of given ratings is less than the number of possible ratings. This is pretty common when the amount of items and users is large.
- Automatic Playlist Generation => Finding a way to generate the recommendations in such a way that they fit in with the user's original playlist
- Evaluation of the MRS => As music itself is a very subjective topic for each user, the traditional metrics such as accuracy, precision and recall etc. may not be the best metrics to estimate the MRS. Furthermore, those metrics that are tailored to MRS often incorporate other data that cannot be mathematically represented.

For each of these problems, the paper suggests possible solutions and their pros and cons. It goes on to describe the different metrics that can be used to evaluate MRS and weighs their usefulness in different scenarios. The paper concludes by talking about future possibilities in the field. It speculates that in the future there might be psychologically aware MRS that are able to make predictions based on user's moods and body language, situationally aware MRS that are capable of understanding external

factors such as location and social context etc. to make predictions and even culturally aware MRS. [2]

3.2 "A comparative study of music recommendation systems (2018) - Ashish Patel, Dr. Rajesh Wadhvani"

The paper compares various different MRS and outlines their pros and cons. It talks about three kinds of recommender systems:

- *Collaborative Filtering* ⇒ Recommend music listened to by other users that share common music with user. Uses a matrix of users and songs with some form of rating as matrix values.
- *Content based MRS* ⇒ Try to recommend music based on perceptual similarity of songs with respect to music listened to by user.
- *Hybrid systems* ⇒ Combines the suggestions from the above approaches.

Apart from these approaches, the paper outlines two novel approaches in detail (as examples for CF and CB based MRS):

- *Preference-linked and Positive Graph based algorithm* ⇒ A CF based approach that replaces matrices with two graphs. One graph is a directed graph with items as nodes and preference correlations as links between the nodes. The second is an undirected graph with items as nodes and positive correlations as edges. It combines these 2 graphs to generate predictions
- *Context Aware recommendations using Incremental Regression Tree* ⇒ A lightweight CB algorithm that was designed for mobile environments. Uses a modified decision tree. The modification is that each time a node is added to the tree, the entire tree is not re-evaluated. Only the leaf node and parent node are changed, thus reducing the load on the system. Thus, the name incremental regression tree.[1]

3.3 "Music recommendation based on semantic audio segment similarity (2008) - Alessandro Bozzon, Giorgio Prandi, Giuseppe Valenzise and Marco Tagliasacchi Politecnico di Milano"

This paper uses a content-based model which incorporates a similarity function that, instead of considering entire songs, analyses audio similarities between semantic segments from different audio tracks. A ranking was obtained by extracting the song's timbre (tone of the song) and comparing it with the others. This ranking was used to recommend new songs to the user. This model achieved an accuracy or around 51% which was higher than other novel methods at the time.

3.4 "Collaborative Filtering for Music Recommender Systems (2017) - Elena Shakirova"

In this paper, Shakirov gives an in-depth explanation of how Collaborative Filtering recommenders work. The paper speaks about the goals of a CFRS and investigates the various aspects to designing such systems. The aspects discussed were:

- Similarity Measure ⇒ Choice of similarity function between user/item pairs
- Scoring function ⇒ The function to be used to assign a score based on similarities between an item and other items in a user's collection.
- Ranking aggregation ⇒ The strategy to follow when recommending songs to a user. Shakirov suggests using an ensemble of recommendation strategies to make suggestions rather than any single strategy.
- Evaluation Metrics ⇒ The two metrics discussed here were Root Mean Squared Error (RMSE) and K Mean Average Precision (K-MAP) as they are the most popular metrics for recommender systems.

3.5 "What to play next? A RNN-based music recommendation system (2017) - Miao Jiang, Ziyi Yang, Chen Zhao"

The paper talks about an approach of using a regular LSTM network to generate the next song based on a playlist. The training set consists of song pairs as inputs to the network and outputs a similarity score as the output. The similarity score is then used to rank the songs and the most similar one is suggested. The songs are split into a sequence of musical slices and passed to the network as input. They performed the experiment for both Audio Data (1000 song pairs) and Lyrical Data (28000 song pairs using a convolutional layer) and managed to get accuracies of 76.5% and 86.4% respectively.

CHAPTER-4

SYSTEM REQUIREMENT SPECIFICATION

4.1 Introduction

Our project aims at designing a Music Recommendation System using the power of modern deep learning approaches. The MRS will be implemented with the help of RNN based Sequence-to-Sequence model. The MRS will predict a next song for the user based on their playlist.

4.1.1 Scope

Music listening activity typically spans over a sequence of songs rather than a single song in isolation. The relative position of each song in the user's playlist has significance, and the user satisfaction can significantly change if the same songs are played but in a different order. So this project mainly caters to users that need recommendations to songs that are similar in nature to the songs in their playlist.

4.2 Product Perspective

- A method that differs from ours introduces a new Reinforcement Learning based method for music recommendation task, and concludes that modelling music recommendation as a sequence decision making task gives the proposed method a small but significant boost in performance compared to only reason about song preferences.
- Another method of doing this is to apply convolutional neural networks to learn latent factors of music audio and use these to predict songs to users. The results of this method showed that recent advances in deep learning methods along with the suggested approach translates very well to the music recommendation task.

- The product is independent and totally self-contained.
- It is intended to run as a web application and therefore will be platform independent only requiring network connectivity.
- Deployment of the application would require a web server.

4.2.1 User Characteristics

Our project will be used by anyone that require songs to be recommended to them. This can vary from day to day use of the product to professional use by disc jockeys and radio jockeys.

4.2.2 General Constraints, Assumptions and Dependencies

- The recommendation system's accuracy will depend upon the amount of data given to it. As we are taking only a subset of 10000 song vectors from our dataset, the accuracy won't be optimal. This is because the original dataset size is very large(280 Gb).
- Due to certain licensing issues, we may not be able to play the entire song but only a small portion of it.
- We assume that the songs in a user's playlist will be available in the subset of 10000 songs.
- The model used to make predictions is a neural network-based solution and will face the general risks associated with neural networks such as overfitting, vanishing gradient problems etc.

4.2.3 Risks

As we are working with large quantities of audio data, we'll need a large amount of computing power otherwise the model will not work.

4.3 System Architecture

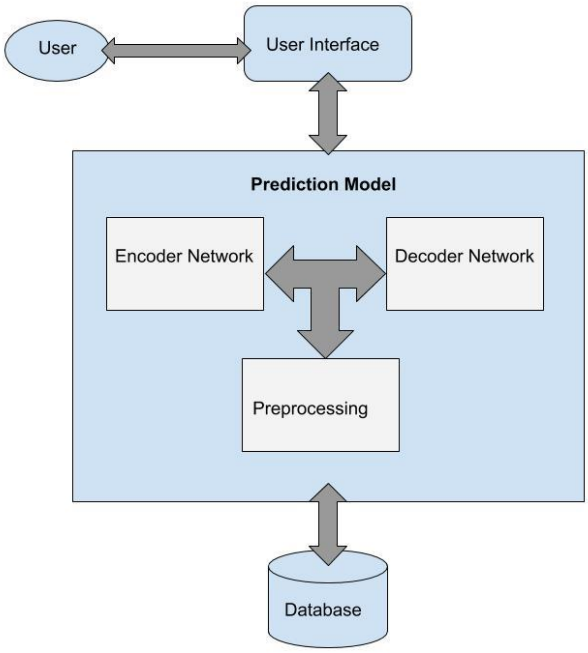

Fig 4.1 System Architecture

The System consists of 5 Main Components:

1. User interface module ⇒ deals with user interactions
2. Pre-processing module ⇒ handles the pre-processing of the musical data, including conversion to a vector space.
3. Encoder Network ⇒ captures the sequence in the playlist
4. Decoder Network ⇒ generates a playlist based on the learning from the encoder.
5. Data Generation ⇒ Converts the data from raw json to usable data

4.4 Requirements List

4.4.1 Front End UI

Reqmt #	Requirement
CRS – 1	The UI is needed by the user to upload songs and get the next song recommended

4.4.2 Data Generation

Reqmt #	Requirement
CRS – 1	Responsible for converting the raw json playlist-song pairs to CSV containing audio features for all the pairs

4.4.3 Data Pre-processing Module

Reqmt #	Requirement
CRS – 1	The pre-processing module is responsible for mapping the musical data to the vector space.

4.4.4 Prediction Module

Reqmt #	Requirement
CRS – 1	The prediction module is needed to recommend the next song depending on the user's playlist

4.5 External Interface Requirements

4.5.1 Hardware Requirements

We are using Google Colab as the platform to train our model as we lack computational power. Apart from that, as our product does not contain any hardware components, we do not have any hardware requirements.

4.5.2 Software Requirements

We require tools to create the front-end User Interface. We also need a database storage for the songs we will store. To build our model we require a machine learning application to process our data.

Front End

HTML, CSS and JavaScript along with a Bootstrap Framework and Flask framework for linking with prediction module.

Prediction Model

Tensorflow and Keras

4.6 User Interfaces

This module is responsible for allowing interaction of the user with the recommendation system. It is also responsible for displaying the recommendations generated by the prediction module and allow the user to control the playback of songs. When the user clicks the recommend button the user's playlist is passed to the server as a list of Spotify IDs to the server using Node.js. The server is implemented as a flask application. It runs as a RESTful service which takes a POST request containing JSON data. The JSON data is a list of Spotify IDs. The flask service runs the prediction model on this input, generates a list of Spotify IDs, passes them to the Spotify Tracks API and passes the resulting JSON as response to the POST request. This response is then handled by Node.js on the front end.

4.7 Performance Requirements

The software response time should be in such a way that the user should not wait for anything

The software should be scalable enough to support a large number of users at the same time

The software should be reliable at all times and should not crash due to unforeseen circumstances

The user interface should be simple so that users do not have to spend too long in understanding the structure of the product

4.8 Special Characteristics

Each user will have access to only their own playlist

They will have a history of the songs recommended to them

Future scope can include user's looking into their friend's playlists

4.9 Help

The User Guide will include a tutorial on how to operate the system and the functionality involved including uploading a song to their playlist and the recommendation of a new song that suits their taste.

4.10 Other Requirements

4.10.1 Site Adaptation Requirements

The layout of the interface should be adaptive in such a way that it can be operated on most types of screens

Browsers should be used in their latest versions otherwise all the features may not be compatible

4.10.2 Safety Requirements

There will be use of existing protocols (like the FTP (File Transfer Protocol)) to ensure secure transfer of data and the encryption will act as a bonus layer.

4.11 Packaging

This product will be packaged as a web app using HTML, CSS and JavaScript along with a Bootstrap Framework and Flask framework for linking with prediction module. This web page

will connect to Tensorflow's Datasets API which will store the entire dataset. Tensorflow and Keras will be used to process the input of user preferences when it comes to favourite songs and use this input to recommend songs that the user will most probably like.

CHAPTER-5

HIGH-LEVEL DESIGN

5.1 Introduction

5.1.1 Overview

Music lovers are always on the lookout for the next song to listen to. Finding music to listen to is easy. But finding music personalized to a listener's taste is difficult. Furthermore, choice of song is affected by several factors such as genre, tempo of the song, whether it is suitable for dancing, mood etc. Our project aims at designing a Music Recommendation System using the power of modern deep learning approaches. The MRS will be implemented with the help of RNN based Sequence-to-Sequence model. The MRS will predict a whole new playlist based on the current playlist provided. This recommendation is made by looking at several audio features of songs as well as the sequence of songs in the playlist. The combination of these two techniques capture several implicit factors that affect choice of music.

5.1.2 Scope

Music listening activity typically spans over a sequence of songs rather than a single song in isolation. The relative position of each song in the user's playlist has significance, and the user satisfaction can significantly change if the same songs are played but in a different order. So this project mainly caters to users that need recommendations to songs that are similar in nature to the songs in their playlist. There are several possible improvements over the method we will be using, such as combination with actual audio rather than simply audio features or taking into account user emotion data when they listened to a particular song. Here we are limiting ourselves to simply make prediction based on the sequence in the playlist.

5.2 Design Constraints, Assumptions and Dependencies

- Ideally we would be using the user's last K songs listened to make our predictions. Due to lack of such a dataset and the absence of means to create such a dataset we are using user playlists.
- Predictions that we will be making will be limited to only the set of songs that are available in our unique songs data.

5.3 Design Description

5.3.1 Master Class Diagram

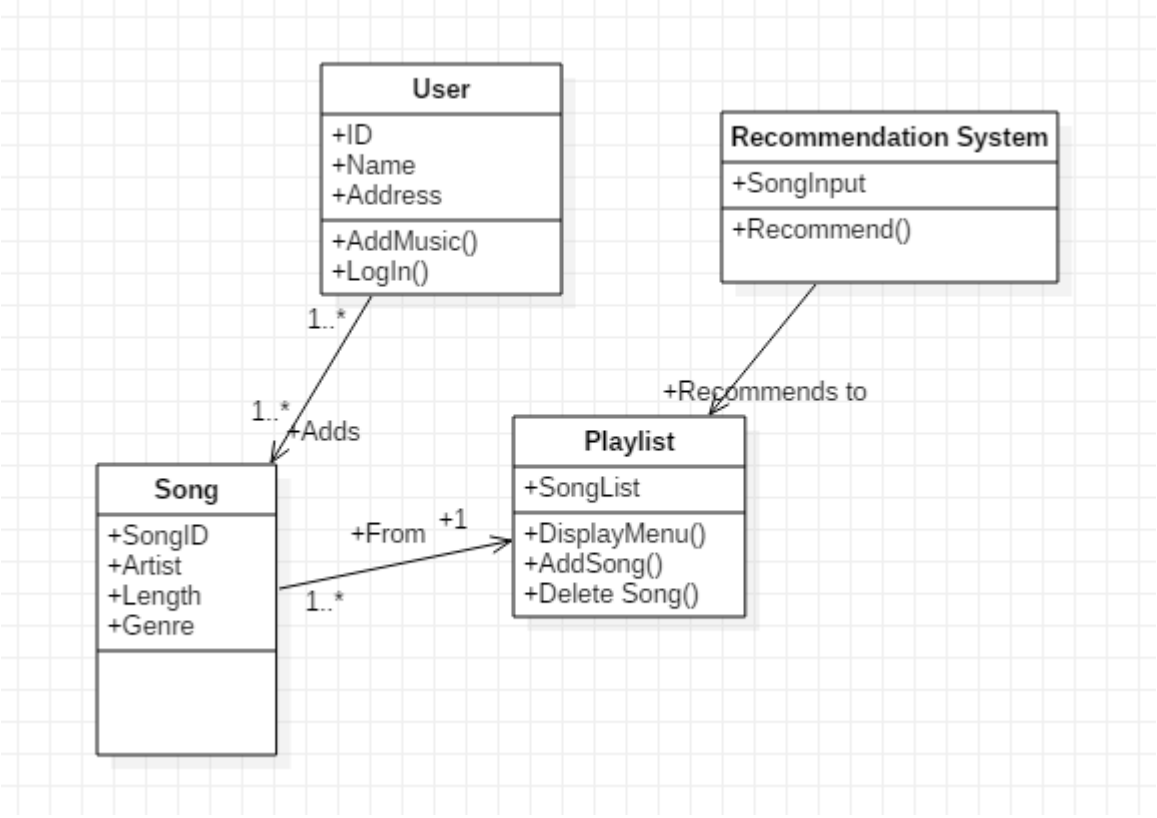

Fig. 5.1 Master Class Diagram

5.3.2 Modules

User:

The User is the person that operates the application. They can play a song of their choice with all the features of usual music players such as fast forward, rewind, pause etc.

Playlist:

The playlist contains songs that are in the database. The user can select the song of their choice from here.

Recommendation System:

This is for providing a song to the user based on the inputs given to it. It consists of an encoder-decoder neural network architecture. The encoder and decoder use LSTM units which are ideal for dealing with sequences. If S is a playlist of songs and N is the number of songs in the playlist [6]:

1. Encoder inputs = S[1...N-1]
2. Decoder inputs = S[2...N]
3. Decoder targets =S[3...N] + End-Of-Sequence

5.3.2.1 Use Case Diagram

:

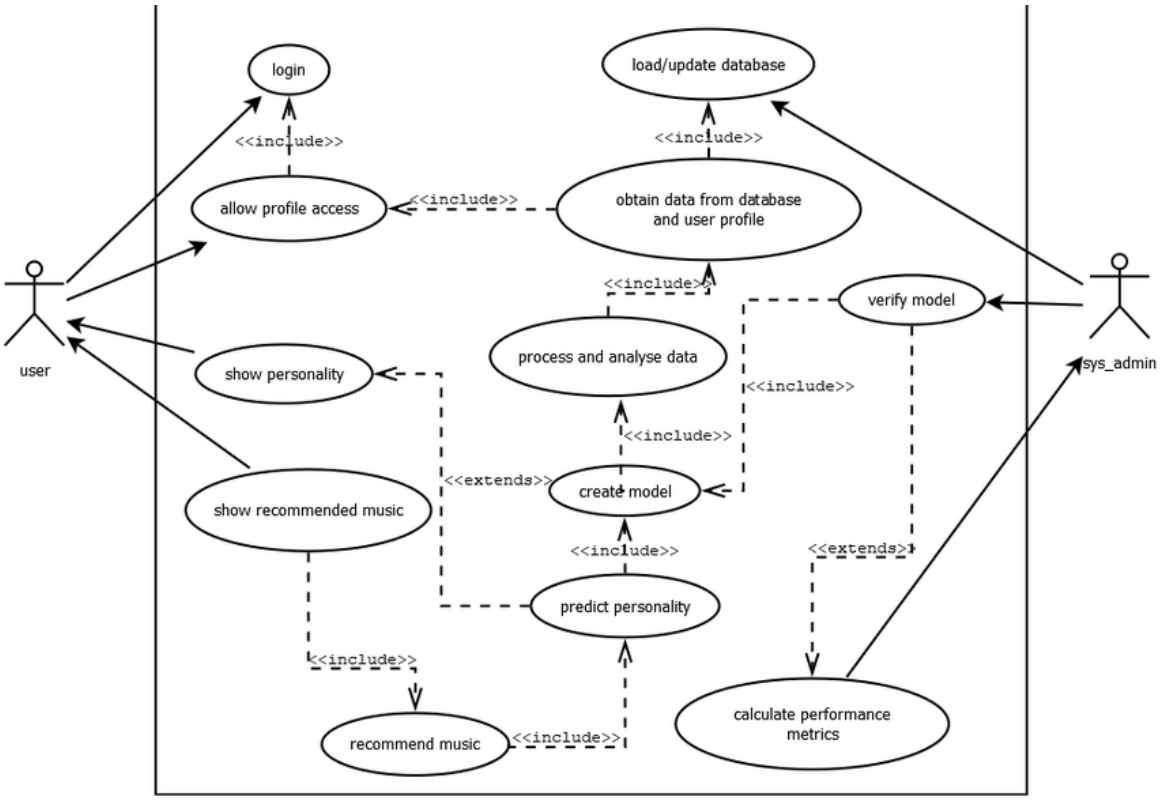

Fig. 5.2 Use Case Diagram

5.3.2.2 Class Diagram

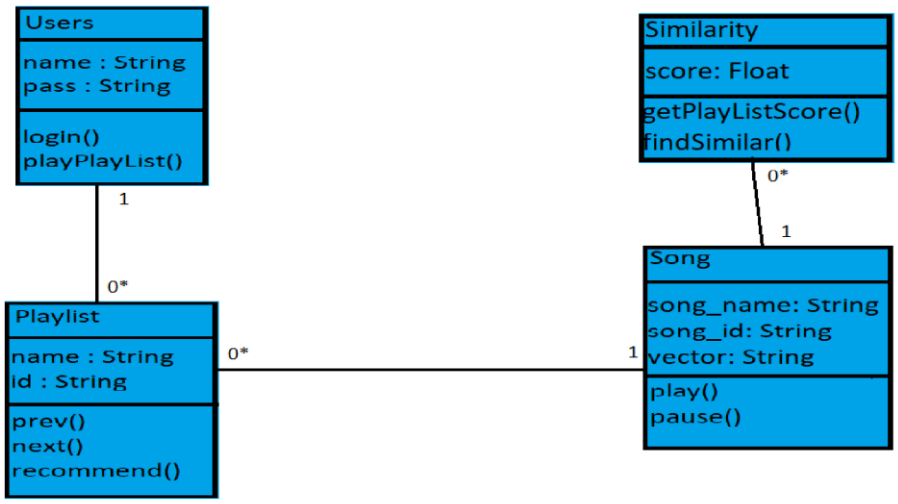

Fig. 5.3 Class Diagram

5.3.2.3 Sequence Diagram

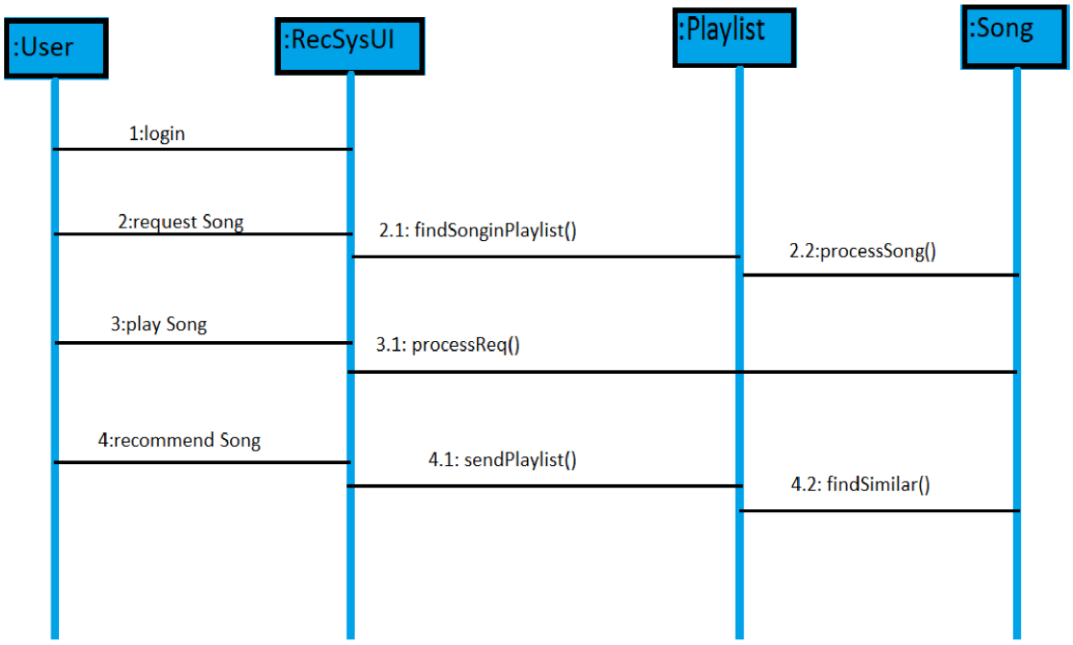

Fig. 5.4 Sequence Diagram

5.4 ER Diagrams

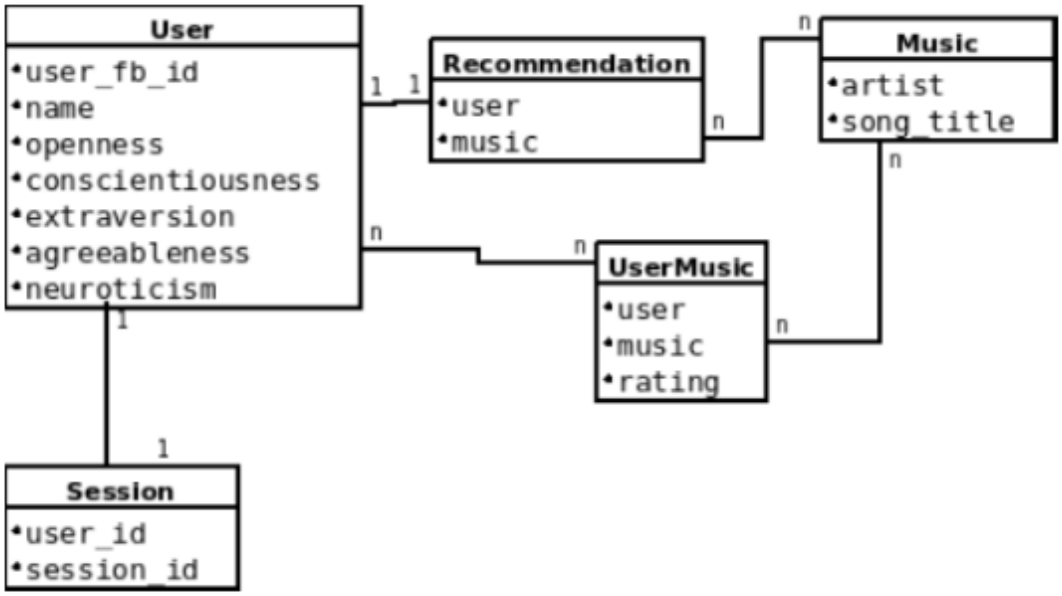

Fig. 5.5 ER Diagram

5.5 User Interface Diagrams

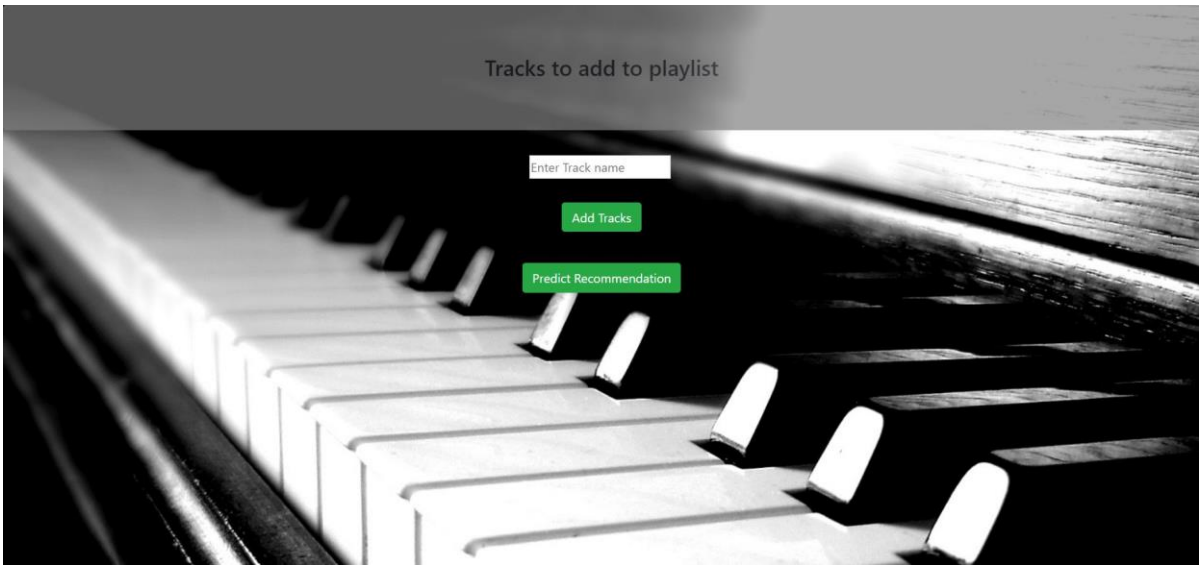

Fig. 5.6 UI Design

5.6 Report Layouts

- Problem Definition
- Literature Survey
- Requirements Specification
- High Level Design
- Low Level Design
- Test Strategy
- Implementation Details
- Conclusion

5.7 External Interfaces

The front page will contain the current song that is playing with the features like pause, rewind and fast forward. There will also be a playlist nearby that the user can see all the songs that exist, and a new song will get added here when it is recommended.

5.8 Packaging and Deployment Diagrams

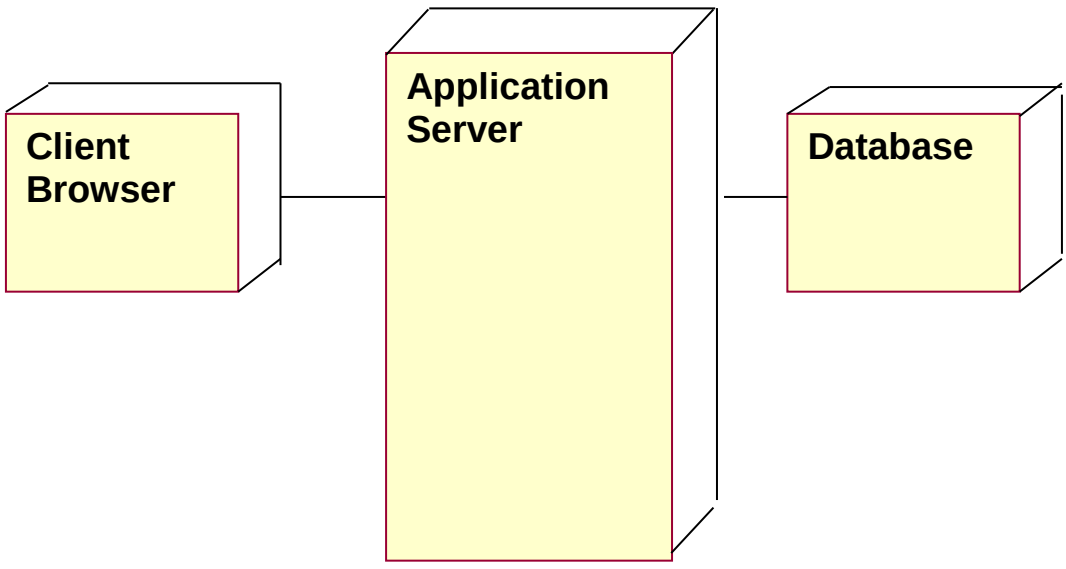

Fig. 5.7 Packaging and Deployment Diagram

5.9 Help

We will have a document that describes all the features of the system and all the steps on how to use it.

5.10 Alternate Design Approach

- Popularity based Model:

  It is the most basic and simple algorithm. We find the popularity of each song by looking into the training set and calculating the number of users who had listened to this song. Songs are then sorted in the descending order of their popularity. For each user, we recommend top most popular songs except those already in his profile. This method involves no personalization and some songs may never be listened in future.

- Collaborative Filtering Model:

  It is a basic Filtering model that recommends you songs based on people that have a similar music taste as you do. Songs are assigned ranks based on the number of users listening to them and the similarity of the users that are listening to the song. This method allows for some personalization but also suffers from the slow start problem.

- Content Based MRS:

  It is an algorithm based on recommending songs based on raw metadata based on similarity of songs. This metadata is what affects the ranking of songs here. There is no personalization but there is no slow-start problem.

5.11 Reusability Considerations

- There exists a music player in the music recommendation system which is made using HTML, CSS and JavaScript. This can be used in multiple scenarios where music needs to be played.
- The recommender can be used for any Sequence prediction scenario.

CHAPTER-6

LOW-LEVEL DESIGN

6.1 Introduction

6.1.1 Overview

The low-level design will outline the various modules and functions of the recommender system that will be implemented. We will first be discussing the design constraints faced by the systems along with all the assumptions we made and dependencies of the system. Following this we discuss the various modules of the system and the various functions that are being carried out by them.

6.1.2 Purpose

The purpose of this low-level design document is to give a more detailed view of how exactly we will go about implementing the music recommendation system. This document will further elaborate upon the functionalities of the various modules present in our system.

6.1.3 Scope

The scope of this document is limited to in-depth explanation of all the various functionalities of the various modules that make the system. It will not discuss implementation details of these systems.

6.2 Design Constraints, Assumptions and Dependencies

- Ideally, we would be using user listening history (last 50 songs played) as the sample data to pass to the model but due to lack of availability we are limited to using playlists as our training data.
- As our song vocabulary is very large (around 34000 unique songs) we do not generate a one-hot encoding on the output layer of the decoder network, as this overflows the available memory.

- We assume that the playlists that are passed to the recommender system contain songs from the song vocabulary.
- The system relies on an external API to generate the song vectors for each song in the playlist.

6.3 Design Description

6.3.1 Dataset Generation

This module is responsible for bringing the training dataset into a form that can be used by the prediction module. It is also responsible for obtaining audio features for all tracks.

6.3.1.1 json_to_csv(path)

Converts the JSON playlistID-track pairs to CSV files.

6.3.1.2 get_audio_features(path,auth_token)

Reads the playlistID-track CSV and appends audio features for all tracks in the CSV using the Spotify API. Requires a spotify authentication token to use the API. Generates another CSV as output.

6.3.1.3 get_song_vocab(path)

Generates the song vocabulary on the above CSV and generates a CSV for the same.

6.3.2 Preprocessing Module

This module is responsible for converting the data into a format that can be understood by the prediction module. As part of preprocessing it performs Normalization, One-Hot encoding categorical variables, generate dataset split and perform padding.

6.3.2.1 read_csv(path)

Reads the Spotify_Data.csv into a Pandas Dataframe.

6.3.2.2 get_max_seq_len(dataframe)

Obtains the length of the playlist with the largest number of songs. This is necessary for making sure all playlists are of same length.

6.3.2.3 norm_numeric_cols(dataframe)

Performs normalization on the numeric features of the song vector. The normalization used is a Min-Max Scalar normalization.

6.3.2.4 one_hot_cat_cols(dataframe)

Converts the categorical features of the song vector to a one-hot encoding.

6.3.2.5 get_data_arrays(dataframe)

Generates the processed datasets that need to be passed as inputs to the Encoder and the Decoder as well as the targets for the Decoder. This function also performs padding on the playlists to get evenly shaped training samples and drops unused columns from the dataframe. Outputs 1 encoder input array, 1 decoder input array and 4 decoder target arrays (one for each head).

6.3.2.6 train_val_test_split(train_samples,val_samples,test_samples)

Takes proportions as inputs and divides the data arrays into train,validation and test data splits based on the given proportions.

6.3.3 Prediction Module

This module is responsible for creation and training of the neural networks employed for making predictions. It is also responsible for making predictions given an input sequence.

6.3.3.1 data_generator(enc_input, dec_input, dec_target_arrays, batch_size)

Generates batches of data and feeds these batches to the model during training. This is done to prevent memory overflow.

6.3.3.2 define_training_model()

Creates the model that will be used for training. Comprises of an encoder network and a decoder network that generates sequences.

6.3.3.3 train_model(model)

Trains the model on data from the data generator. Has several parameters that can be tweaked to improve performance.

6.3.3.4 define_inference_model()

Creates the model that will be used during prediction phase. Uses the same encoder from the training model but alters the decoder to generate only one song vector at a time.

6.3.3.5 make_predictions(model,input_playlist,k)

Uses the inferencing model to generate one prediction at a time from the input sequence. This prediction is then passed as decoder input back into the inferencing model to generate the next prediction. Generates K such predictions. Outputs the recommendations as a list of Spotify track IDs

6.3.3.6 find_most_similar(song_vector)

Finds the most similar song vector from the song vocabulary that matches the predicted song vector. This is done with the help of a KDTree on the song vocabulary.

6.3.4 Frontend and UI module

6.3.4.1 mainFunction()

Generates a search bar to add tracks to playlists and to call the prediction function.

6.3.4.2 add_list()

Adds Each track that the user searches for to the playlist for predictions

6.3.4.3 dist_songId()

Used to display the set of songs that have been currently added to the database

6.3.3.4 recommendations ()

Calls the recommendations api to find out what songs have been predicted and displays links to play each predicted song externally

CHAPTER-7

IMPLEMENTATION

The core language for implementation is python. We have used the Keras library (with tensorflow backend) along with scikit-learn to build our recommender. The backend server is done with the Flask library. The frontend has been implemented using HTML, CSS and Node.js. We have used Google Colab as the platform for training the network as we lacked the computational power on our systems.

7.1 Dataset Generation

We first begin by converting our dataset, which is a json file containing spotify track uris and playlist id into a CSV.

Looking at Fig 7.1, we use the spotify credentials (obtained after creating an app on the spotify developer dashboard) for authentication with the spotify API. We then define the get_audio_features() function which takes a batch of 100 rows, extracts the spotify IDs, makes a call to the spotify API to get audio features and then concatenates it with the other fields of a json and returns this dictionary .

```
client_creds_mgr = SpotifyClientCredentials(client_id="<your client id>",client_secret="<your secret key>")
spotify = spotipy.Spotify(client_credentials_manager=client_creds_mgr)
basedir = "C:/Users/Shashank/Documents/notes/4th year notes/8th Sem/minor project/"
readdf = pd.read_csv(os.path.join(basedir,"spotify playlist dataset/data_CSV/0-999.csv"))

track_features_list = []

def get_audio_features(rows):
    track_ids = []
    for index,row in rows.iterrows():
        #print(row)
        track = row['trackid']
        tokens = track.split(':')
        track_id = tokens[2]
        track_ids.append(track_id)
    features = spotify.audio_features(track_ids)
    for tf,(index,row) in zip(features,rows.iterrows()):
        tf.update({'artist_name':row['artist_name'],'track_name':row['track_name'],'pid':row['pid']})
    return features
```

Fig 7.1 spotify_data.py - authentication and feature extraction function

We then run this function repeatedly for all of the data and make this into a CSV file. We also get all the distinct songs to generate the song vocabulary and write this as a CSV as well.

```
for i in range(100,len(readdf.index),100):
    print(i)
    track_features_list.extend(get_audio_features(readdf.iloc[(i-100):i]))

writedf = pd.DataFrame(track_features_list)
writedf.to_csv(os.path.join(basedir,"spotify_data.csv"),index = False)

df = pd.read_csv(os.path.join(basedir,"spotify_data.csv"))
df = df.loc[:, df.columns != 'pid']
df = df.drop_duplicates()
df.to_csv(os.path.join(basedir,"spotify_songs.csv"),index = False)
```

Fig 7.2 spotify_data.py - feature extraction and CSV generation

7.2 Preprocessing and Model Training

Now we need to load the generated CSVs, process them and use the data to train our machine learning model.

We load the audio feature dataset into our program and define some program parameters. We then find the maximum playlist length by grouping by ‘pid’ and iterating through all the groups. Next, we One-hot-encode all the categorical variable in the feature vectors using the dummyPy library. This is followed by normalization of all the real valued variables to bring them between 0 and 1. The normalization used is MinMax Scalar normalization.

```
[ ]  basedir = "/content/gdrive/My Drive/Minor Project Data/"
     #songlistdf = pd.read_csv(os.path.join(basedir,'spotify_songs.csv'))
     playlistdf = pd.read_csv(os.path.join(basedir,'spotify_data.csv'))

     NUM_SAMPLES = 1000
     train_samples = 600
     val_samples = 200
     test_samples = 200

     #vocab_len = len(songlistdf)
```

```
[ ]  max_seq_len = 0
     playlistdf['pid'].apply(int)

     for name,group in playlistdf.groupby('pid'):
       max_seq_len = max(max_seq_len,len(group))

     onehotencoder = OneHotEncoder(["time_signature", "key", "mode"])
     onehotencoder.fit(playlistdf)
     proc_playlistdf=onehotencoder.transform(playlistdf)
     cols_to_norm = ['duration_ms','loudness','tempo']
     proc_playlistdf[cols_to_norm] = proc_playlistdf[cols_to_norm].apply(lambda x: (x - x.min()) / (x.max() - x.min()))
```

```
[ ]  print(max_seq_len)
     print(proc_playlistdf.shape)
     print(proc_playlistdf.columns.values)
```

```
245
(67500, 37)
['acousticness' 'analysis_url' 'artist_name' 'danceability' 'duration_ms'
 'energy' 'id' 'instrumentalness' 'key_0' 'key_1' 'key_2' 'key_3' 'key_4'
 'key_5' 'key_6' 'key_7' 'key_8' 'key_9' 'key_10' 'key_11' 'liveness'
 'loudness' 'mode_0' 'mode_1' 'pid' 'speechiness' 'tempo'
 'time_signature_0' 'time_signature_1' 'time_signature_3'
 'time_signature_4' 'time_signature_5' 'track_href' 'track_name' 'type'
 'uri' 'valence']
```

Fig 7.3 MRS_Colab.ipynb - Normalization and categorical variables to One-Hot

Then, we seperate the real value features from the categorical ones. This is important as we are doing regression and we will perform multi-head regression to generate the vector and the real value features will use a different output layer as opposed to the categorical variables.

The get_data_arrays function organises the preprocessed dataframe into 3D numpy arrays for encoder input, decoder input and decoder targets of shape (num_playlists,max_seq_len,num_features) . The decoder target features are split into 4 arrays along the 3rd axis. These are the regression features, key features, mode features and time signature features. As our network is multi-headed, we need to split the targets as such. This results in the function returning a total of 6 3D numpy arrays.

```
reg_features = ['acousticness','danceability','duration_ms','energy','instrumentalness','liveness','loudness','speechiness','tempo','valence']
key_features = ['key_0','key_1','key_2','key_3','key_4','key_5','key_6','key_7','key_8','key_9','key_10','key_11']
mode_features = ['mode_0','mode_1']
timesig_features= ['time_signature_0', 'time_signature_1', 'time_signature_3', 'time_signature_4', 'time_signature_5']

def get_data_arrays(groupbyuser):

  enc_input_data = []
  dec_input_data = []
  dec_target_reg = []
  dec_target_key = []
  dec_target_mode = []
  dec_target_timesig = []

  for name,group in groupbyuser:
      grp = group.drop(['analysis_url','artist_name','id','track_href','track_name','type','uri','pid'],axis=1)

      grouplist = grp.values
      grouplist_reg = grp[reg_features].values
      grouplist_key = grp[key_features].values
      grouplist_mode = grp[mode_features].values
      grouplist_timesig = grp[timesig_features].values

      xgroup = grouplist[0:len(grouplist)-1]
      ygroup = grouplist[1:len(grouplist)]
      ytarget_reg = grouplist_reg[2:len(grouplist_reg)]
      ytarget_key = grouplist_key[2:len(grouplist_key)]
      ytarget_mode = grouplist_mode[2:len(grouplist_mode)]
      ytarget_timesig = grouplist_timesig[2:len(grouplist_timesig)]

      for i in range((max_seq_len-len(xgroup))):
          xgroup=np.vstack([xgroup,np.zeros(xgroup.shape[1])])

      for i in range((max_seq_len-len(ygroup))):
          ygroup=np.vstack([ygroup,np.zeros(ygroup.shape[1])])

      for i in range((max_seq_len-len(ytarget_reg))):
          ytarget_reg=np.vstack([ytarget_reg,np.zeros(ytarget_reg.shape[1])])

      for i in range((max_seq_len-len(ytarget_key))):
          ytarget_key=np.vstack([ytarget_key,np.zeros(ytarget_key.shape[1])])

      for i in range((max_seq_len-len(ytarget_mode))):
          ytarget_mode=np.vstack([ytarget_mode,np.zeros(ytarget_mode.shape[1])])

      enc_input_data.append(xgroup)
      dec_input_data.append(ygroup)
      dec_target_reg.append(ytarget_reg)
      dec_target_key.append(ytarget_key)
      dec_target_mode.append(ytarget_mode)
      dec_target_timesig.append(ytarget_timesig)

  enc_input_data= np.array(enc_input_data)
  dec_input_data = np.array(dec_input_data)
  dec_target_reg = np.array(dec_target_reg)
  dec_target_key = np.array(dec_target_key)
  dec_target_mode = np.array(dec_target_mode)
  dec_target_timesig = np.array(dec_target_timesig)

  return enc_input_data,dec_input_data,dec_target_reg,dec_target_key,dec_target_mode,dec_target_timesig
```

Fig 7.4 MRS_Colab.ipynb - Fetching data arrays

We then run this function on the preprocessed dataframe and perform train-val-test split on the dataset.

```
groupbyuser = proc_playlistdf.groupby('pid')
enc_input,dec_input,dec_target_reg,dec_target_key,dec_target_mode,dec_target_timesig= get_data_arrays(groupbyuser)

enc_input_train,dec_input_train,dec_target_reg_train,dec_target_key_train,dec_target_mode_train,dec_target_timesig_train = enc_input[:train_samples],dec_input[:train_samples],dec_target_reg[

enc_input_val,dec_input_val,dec_target_reg_val,dec_target_key_val,dec_target_mode_val,dec_target_timesig_val = enc_input[train_samples:train_samples + val_samples],dec_input[train_samples:tr

enc_input_test,dec_input_test,dec_target_reg_test,dec_target_key_test,dec_target_mode_test,dec_target_timesig_test = enc_input[train_samples + val_samples:train_samples + val_samples + test_

print(enc_input_train.shape)
print(dec_input_train.shape)
print(dec_target_reg_train.shape)
print(dec_target_key_train.shape)
print(dec_target_mode_train.shape)
print(dec_target_timesig_train.shape)
```

Fig 7.5 MRS_Colab.ipynb - train-test split

Now that we have our data arrays, we write the generator class that will feed the data to the network in batches.

```
# Generator function to generate
class my_generator(Sequence):

    def __init__(self, enc_inp,dec_inp,dec_tar_reg,dec_tar_key,dec_tar_mode,dec_tar_timesig,batch_size):
        self.enc_inp = enc_inp
        self.dec_inp = dec_inp
        self.dec_tar_reg = dec_tar_reg
        self.dec_tar_key = dec_tar_key
        self.dec_tar_mode = dec_tar_mode
        self.dec_tar_timesig = dec_tar_timesig
        self.batch_size = batch_size

    def __len__(self):
        return ceil(self.enc_inp.shape[0] / float(self.batch_size))

    def __getitem__(self, idx):
        enc_inp_batch = self.enc_inp[idx*batch_size:(idx+1)*batch_size]
        dec_inp_batch = self.dec_inp[idx*batch_size:(idx+1)*batch_size]
        dec_tar_reg_batch = self.dec_tar_reg[idx*batch_size:(idx+1)*batch_size]
        dec_tar_key_batch = self.dec_tar_key[idx*batch_size:(idx+1)*batch_size]
        dec_tar_mode_batch = self.dec_tar_mode[idx*batch_size:(idx+1)*batch_size]
        dec_tar_timesig_batch = self.dec_tar_timesig[idx*batch_size:(idx+1)*batch_size]

        return [enc_inp_batch,dec_inp_batch],[dec_tar_reg_batch,dec_tar_key_batch,dec_tar_mode_batch,dec_tar_timesig_batch]
```

Fig 7.6 MRS_Colab.ipynb - Data Generator class

Now it's time for us to define the model. To build the encoder we use An input layer that accepts a sequence of song vectors of shape (245,29) and an LSTM layer. We capture the hidden states h and c and set these as initial states for the decoder and discard the encoder outputs. We then define the decoder which follows a similar structure as the encoder but adds 4 heads. 1 regression head and 3 classification heads. the regression head uses a sigmoid activation while the classification heads use a softmax activation. Notice that return sequences = true in the decoder LSTM. This indicates that we get the entire predicted sequence once all predictions have been made.

```
# training model
latent_dim = 256

encoder_inputs = Input(shape=(None, enc_input.shape[2]))
encoder = LSTM(latent_dim, return_state=True)
encoder_outputs, state_h, state_c = encoder(encoder_inputs)
encoder_states = [state_h, state_c]

decoder_inputs = Input(shape=(None, dec_input.shape[2]))
decoder_lstm = LSTM(latent_dim, return_sequences=True, return_state=True)

decoder_outputs, _, _ = decoder_lstm(decoder_inputs,initial_state=encoder_states)

reg_layer = Dense(dec_target_reg.shape[2],activation = 'sigmoid', name = 'reg_output')
key_layer = Dense(dec_target_key.shape[2],activation = 'softmax', name = 'key_output')
mode_layer = Dense(dec_target_mode.shape[2],activation = 'softmax', name = 'mode_output')
timesig_layer = Dense(dec_target_timesig.shape[2],activation = 'softmax', name = 'timesig_output')

decoder_outputs_reg = reg_layer(decoder_outputs)
decoder_outputs_key = key_layer(decoder_outputs)
decoder_outputs_mode = mode_layer(decoder_outputs)
decoder_outputs_timesig = timesig_layer(decoder_outputs)

model = Model([encoder_inputs, decoder_inputs], [decoder_outputs_reg,decoder_outputs_key,decoder_outputs_mode,decoder_outputs_timesig])
```

Fig 7.7 MRS_Colab.ipynb - Training model

Finally we compile and train our model. We define the losses for each of the heads and the loss weights. We use rmsprop as the optimizer and run the training using the training data and validation data.

```
batch_size = 32
epochs = 20

train_gen = my_generator(enc_input_train,dec_input_train,dec_target_reg_train,dec_target_key_train,dec_target_mode_train,dec_target_timesig_train,batch_size)
val_gen = my_generator(enc_input_val,dec_input_val,dec_target_reg_val,dec_target_key_val,dec_target_mode_val,dec_target_timesig_val,batch_size)

loss_funcs = {
    'reg_output':'mse',
    'key_output':'categorical_crossentropy',
    'mode_output':'categorical_crossentropy',
    'timesig_output':'categorical_crossentropy'
}

loss_wts = {
    'reg_output': 1.0,
    'key_output': 1.0,
    'mode_output': 1.0,
    'timesig_output': 1.0
}

model.compile(optimizer='rmsprop', loss=loss_funcs,loss_weights=loss_wts )

model.fit_generator(generator=train_gen,
                    validation_data=val_gen,
                    use_multiprocessing=True,
                    workers=4,
                    epochs=epochs)
```

Fig 7.8 MRS_Cola.ipynb - Model training

One final thing that we need to do is to define the inference model. The model works similar to the training model except that we separate the encoder and decoder models to get 2 independent models. We initialize the encoder and decoder with the states of the training model. One more change we need to make is to set return sequences to false in the decoder and add the encoder states as an additional input to the decoder. We save these 2 models with the .h5 files to be used later.

```
#inferencing model

encoder_model = Model(encoder_inputs, encoder_states)

decoder_state_input_h = Input(shape=(latent_dim,))
decoder_state_input_c = Input(shape=(latent_dim,))
decoder_states_inputs = [decoder_state_input_h, decoder_state_input_c]
decoder_outputs, state_h, state_c = decoder_lstm(decoder_inputs, initial_state=decoder_states_inputs)
decoder_states = [state_h, state_c]

decoder_outputs_reg = reg_layer(decoder_outputs)
decoder_outputs_key = key_layer(decoder_outputs)
decoder_outputs_mode = mode_layer(decoder_outputs)
decoder_outputs_timesig = timesig_layer(decoder_outputs)

decoder_model = Model([decoder_inputs] + decoder_states_inputs,[decoder_outputs_reg,decoder_outputs_key,decoder_outputs_mode,decoder_outputs_timesig] + decoder_states)
```

Fig 7.9 MRS_Colab.ipynb - Inference model

7.3 Making predictions and Backend Server

The predictions are done using the inference encoder and decoder models we used earlier. We also use the normalized song vocabulary we used earlier to build a KDTree which is used for efficient similarity computation. The backend uses flask and acts as a RESTful API. Making a POST request call to url/recommends gives us predictions for a particular playlist.

Once again, we pass our spotify credentials to authenticate with the spotify API. We also build the KDTree using the song vocabulary.

```
app = Flask(__name__)

# Builds the KDTree
def build_sim_tree(song_vocab):
    sim_tree = spatial.KDTree(song_vocab_vecs)
    return sim_tree

#On start setup
max_seq_len = 245
max_playlist_len = 3
max_pred_iterations = 500
song_vocab = pd.read_csv("norm_songs.csv")
song_vocab_vecs = song_vocab.drop(['id'],axis=1).values
setrecursionlimit(10000)
sim_tree = build_sim_tree(song_vocab)
client_creds_mgr = SpotifyClientCredentials(client_id="<your client id>",client_secret="<your client secret>")
spotify = spotipy.Spotify(client_credentials_manager=client_creds_mgr)
```

Fig 7.10 app.py - Initializations

Whenever we send a POST request to localhost:5000/recommends, the recommendations function gets activated. In it we obtain the normalized vectors for the spotify ids passed to the server. we then proceed to perform similar preprocessing that we did on the training data, to the normalized vectors.These are then passed to the decode_sequence() function which performs the predictions.

```
# RESTful service to get recommendations
@app.route('/recommends',methods=['POST'])
def recommendations():
    if not request.json or not 'idlist' in request.json:
        abort(400)

    #recieves a list of spotify ids as JSON
    idlist = request.json['idlist']

    #get audio features and convert to df
    playlist = song_vocab[song_vocab['id'].isin(idlist)]
    playlist = playlist.drop(['id'],axis=1)

    # print(playlist.columns.values)
    # print("Features:{0}",len(playlist.columns.values))

    #perform preprocessing on df
    playlist_inp = playlist.values

    for i in range(max_seq_len-len(playlist_inp)):
        playlist_inp = np.vstack([playlist_inp,np.zeros(29)])
    playlist_inp = np.reshape(playlist_inp,(1,-1,29))

    #Load inferencing model
    inf_enc,inf_dec = get_models("inf_encoder.h5","inf_decoder.h5")

    #get predictions
    pred_ids = decode_sequence(playlist_inp,inf_enc,inf_dec,idlist)
```

7.11 app.py - recommendations function

The decode sequence function first generates the start-of-sequence vector. This vector is taken as the mean of all the input vectors. The decoder generates song vectors(along with decoder states) one step at a time. This generated vector is queried on the KDTree to generate the most similar vector in the vocabulary. The ID of this song vector is outputted along with the vector and is added to the recommendations list if it is not already there. The the generated states and recommended vector are passed to the next iteration of the decoder. This loop runs until we reach the prediction limit. Returns the list of recommended IDs.

```
def decode_sequence(input_seq,encoder_model,decoder_model,idlist):

    states_value = encoder_model.predict(input_seq)

    target_seq = prep_sos_seq(input_seq)
    print(target_seq)

    checklist = idlist

    stop_condition = False
    rec_song_ids = []
    c = 0

    while not stop_condition:

        reg, key, mode, timesig ,h, c = decoder_model.predict(
            [target_seq] + states_value)

        songid,song_vec = getclosestsong(reg[0,-1,:],key[0,-1,:],mode[0,-1,:],timesig[0,-1,:])

        if songid not in checklist:
            rec_song_ids.append(songid)
            print("added a song : "+songid)
        checklist.append(songid)
        if len(rec_song_ids) >= max_playlist_len:
            stop_condition = True
        target_seq = song_vec
        states_value = [h, c]
        c = c + 1

    return rec_song_ids
```

Fig 7.12 app.py - decode_sequence() function

Finally we run the Spotify Tracks API on the list of recommended IDs to generate all the details of the songs (name,album,artist,preview_url) and send this as response to the POST request.

```
#use spotify API and spotify ID to get JSON data
pred_dict = spotify.tracks(pred_ids)

#return JSON as response to the POST request
return jsonify({'preds':pred_dict}),201
```

Fig 7.13 app.py - returning response

7.4 Front End

The frontend is made using flask to create a web page that can take input directly from users and call the prediction model to retrieve predictions and uses these predictions and creates a list of songs along with their playable ids to help the user try each song to identify what they like out of the three given options. There are a few functions that help process all necessary information for front-end which will be explained below

The necessary imports that are needed for the same are flask, spotipy and json

```
from flask import Flask,jsonify,request,abort,make_response,render_template,redirect
import pandas as pd
import numpy as np
from keras.models import load_model
from scipy import spatial
from sys import setrecursionlimit
import json
import spotipy
from spotipy.oauth2 import SpotifyClientCredentials
```

The mainFunction() calls the index.html file which is used to search for tracks and store them in a global variable

```html
<html>
    <head>
        <title>Music Recommender System</title>
    </head>
    <body>
        <h3>Tracks to add to playlist</h3>
        <form method="POST" action="/songQ">
            <input type="text" name="track" placeholder="Enter Track name">
            <br>
            <input type="submit" value="Add Tracks">
        </form>
        <br>
        <form method="POST" action="/recommends">
            <input type="submit" value="Predict Recommendation">
        </form>
    </body>
</html>
```

The index.html then sends the tracks to the add_list() which adds the track id to the songs array and redirects back to the index.html home page to accept more songs or predict songs using current playlist

```
@app.route('/songQ',methods = ['POST'])
def add_list():
    # print(request.form)
    # return jsonify({
    #     "message": request.form["track"]
    # })
    # name = request.json['title']
    name = request.form["track"]
    results = spotify.search(q=name, type='track')

    res = json.dumps(results)
    main = json.loads(res)
    if (len(main['tracks']['items'])>0):
        songid = main['tracks']['items'][0]['id']
        songName = main['tracks']['items'][0]['name']
        print(songName)
        songs.append(songid)
        names.append(songName)
    songid = -1
    return redirect(url_for('mainFunction'))
    # return jsonify({
    #     "message" : "Sucess",
    #     "url": url_for(mainFunction)
    # })
```

We can choose to predict songs once we add enough songs to the playlist. This is taken care of by the recommend() function which then processes the data and returns three songs to the user

```
# RESTful service to get recommendations
@app.route('/recommends',methods=['POST'])
def recommendations():
    # if not request.json or not 'idlist' in request.json:
    #     abort(400)

    #recieves a list of spotify ids as JSON
    # idlist = request.json['idlist']
    # print(idlist)
    # print()
    print(songs)
    idlist = songs

    #get audio features and convert to df
    playlist = song_vocab[song_vocab['id'].isin(idlist)]
    playlist = playlist.drop(['id'],axis=1)

    # print(playlist.columns.values)
    # print("Features:{0}",len(playlist.columns.values))

    #perform preprocessing on df
    playlist_inp = playlist.values

    for i in range(max_seq_len-len(playlist_inp)):
        playlist_inp = np.vstack([playlist_inp,np.zeros(29)])
    playlist_inp = np.reshape(playlist_inp,(1,-1,29))
```

```
#load inferencing model
inf_enc,inf_dec = get_models("inf_encoder.h5","inf_decoder.h5")

#get predictions
pred_ids = decode_sequence(playlist_inp,inf_enc,inf_dec,idlist)

#use spotify API and spotify ID to get JSON data
pred_dict = spotify.tracks(pred_ids)
# pred_dict = jsonify(pred_dict)
# return pred_dict
# return JSON as response to the POST request
# return jsonify({'preds':pred_dict}),201
return render_template('predictions.html',pred = pred_dict)
```

CHAPTER-8

TESTING

8.1 Introduction

The Test Plan document archives and tracks the necessary information required to effectively define the approach to be used in the testing of the product. This allows the final product to work as efficiently as possible.

8.1.1 Scope

This document encompasses the testing plan for our Music Recommendation System. Testing will be done across the modules that exist in our project.

8.2 Test Strategies

User Friendliness:

The product should be easy and simple to use so that the users do not have any issue in navigating

Stress Test:

The product should be tested beyond the normal operation in order to see how it will perform in extreme cases.

Disaster Recovery:

Once an issue has been found, the product has to be fixed in order to get it in working condition again

8.3 Performance Criteria

1	Availability	The system will be available all the time.
2	Performance	The user interface screens respond within 3 seconds
3	Maintainability	The system developed can be easily maintained with the added benefit of easy updating of information Development standards such as structured programming, recognizable nomenclature, clear documentation ensure maintainability of code
4	Usability	Use of Hallway testing on random users have led to the conclusion that the website developed is user friendly which ensure efficient use of software and reduced cost of training.
5	Accuracy	The system ensures that data transmission, storing, maintaining and accessing of data will be accurate.

Table 8.1 Performance Criteria

8.4 Test Environment

The product's front end was made using the latest version of chrome so it should be tested in other versions.

We have used Google Colab to run our model. As our model requires high resources in terms of RAM and GPU, we have to try running it on other machines

8.5 Risk Identification and Contingency Planning

This section identifies the risks that are associated during the testing

Risk #	Risk	Nature of Impact	Contingency Plan
1	While performing integration testing, each unit test has to be performed again	The product as a whole will not work	Intense testing after integration testing will be carried out
2	Dependencies across different units	Specific units might not work with each other	Test different units with each other

Table 8.2 Risks

8.6 Roles and Responsibilities

Unit Testing is done by the programmer who develops a particular functionality.

Integration Testing is done by our team when we combine all the modules together

System testing is done by our team who evaluates the system's compliance with the specified requirements.

8.7 Test Schedule

The model used to recommend the playlist was built first so the unit testing of this model was carried out initially.

The front-end to play the music was built after this and then the backend integration was done.

After these modules were built, integration testing was done to see if they work with each other.

8.8 Acceptance Criteria

The matching of the expected output and the actual output is the criteria that is taken into consideration for determine the passing of a test case.

If that doesn't match, then the test case would have failed in its approach.

8.9 Test Case List

This section shall clearly define the test cases that are planned for testing.

Test Case Number	Test Case	Required Output
1	Pre-processing of data	The data should be pre-processed into a form that can be passed into the model
2	Encoder data passed to decoder	The data that the encoder provides should be read by the decoder
3	Decoder Output	The decoder should provide a new recommendation
4	Music Player	The music player should have all the features working

Table 8.3 Test Cases

8.10 Test Data

For testing the music recommender, we give the model song data and then test the accuracy of the output of the model.

CHAPTER-9

RESULTS AND DISCUSSION

9.1 Dataset

The dataset we chose was the spotify playlist dataset. This dataset consisted of 4000 playlists of which we make use of a 1000 of them. We make a train-validation-test split of 600:200:200 samples. Each sample consists of playlistID-songID pair. We make use of the Spotify Developer API to extract audio features from the SongIDs. The Encoder Input, Decoder Input and Decoder Targets are all 3D numpy arrays of shape (Number of training samples, Maximum playlist length,Number of features) which in this case comes out to (600,245,29). The feature descriptions are given below:

Audio Feature	Description
Duration	Duration in milliseconds
Key	Specifies the overall key
Mode	Major or minor scale
Time Signature	Measure of beats in one bar
Acousticness	Measure of whether the track is acoustic
Danceability	Suitability to dance
Energy	Measure of intensity and activity
Instrumentalness	Detects presence of vocals
Liveness	Determines if the audio was played live

Loudness	Loudness in decibels
Speechiness	Presence of spoken words
Valence	0-1 describing musical positiveness
Tempo	Speed of the track in beats per minute
ID	Spotify ID
URI	Spotify uri
Track_href	link to API that provides full details
Analysis_URL	url to access the song's full analysis
Type	Object Type

Table 9.1 Dataset Features

9.2 Training parameters

The model uses 4 separate loss functions for each of the 4 outputs, MSE for the real-valued features and Categorical Cross Entropy for the 3 categorical variables. Each of these losses have equal weightage. We use RMSprop as the optimizer for the model. Batch size is set to 32 samples and we run this model for 20 epochs. The number of units in the hidden layer is 256.

9.3 Metrics

Recommender systems cannot use accuracy as a metric because music suggested is very subjective in nature. Instead the metrics most recommender systems use is Mean Average Precision (MAP). MAP requires the knowledge of precision.

$$mAP = \frac{1}{|classes|} \sum_{c \in classes} \frac{\#TP(c)}{\#TP(c) + \#FP(c)}$$

Fig 9.1 Mean Average Precision

Where c is the cutoff point (precision on top-c). We will be using a variation of the AP formula called AP@N to evaluate our recommendation system (as there is no measure of relevance).

9.4 Results

The recommender system was able to make successful predictions based on the input sequence. We set a prediction limit of 30 songs and ran the model. We took 5 playlists from the test set and set a cut off value(c) of 10 and calculated average precision for each playlist and took mean over these playlists. Comparing these results to those in [6]

	Matrix Factorization	Regular LSTM (single layer)	Our Seq2Seq network (single layer)
MAP	0.0017	0.0028	0.0030

Table 9.1 MAP Comparison

CHAPTER-10

SNAPSHOTS

trackid	artist_nam	track_nam	pid	
spotify:tra	Missy Ellio	Lose Contı	0	
spotify:tra	Britney Spe	Toxic	0	
spotify:tra	BeyoncÃ©	Crazy In Lc	0	
spotify:tra	Justin Timl	Rock Your	0	
spotify:tra	Shaggy	It Wasn't N	0	
spotify:tra	Usher	Yeah!	0	
spotify:tra	Usher	My Boo	0	
spotify:tra	The Pussyc	Buttons	0	

Fig 10.1 Original data CSV

acousticne	analysis_u	artist_nam	danceabili	duration_r	energy	id	instrument	key	liveness	loudness	mode	pid	speechine	tempo	time_signa	track_href	track_nam	type	uri	valence
0.0311	https://ap	Missy Ellio	0.904	226864	0.813	0UaMYEvV	0.00697	4	0.0471	-7.105	0	0	0.121	125.461	4	https://ap	Lose Contı	audio_fea	spotify:tra	0.81
0.0249	https://ap	Britney Spe	0.774	198800	0.838	6l9VzXrHx	0.025	5	0.242	-3.914	0	0	0.114	143.04	4	https://ap	Toxic	audio_fea	spotify:tra	0.924
0.00238	https://ap	BeyoncÃ©	0.664	235933	0.758	0WqIKmW	0	2	0.0598	-6.583	0	0	0.21	99.259	4	https://ap	Crazy In Lc	audio_fea	spotify:tra	0.701
0.202	https://ap	Justin Timl	0.891	267267	0.714	1AWQoqb	0.000234	4	0.0521	-6.055	0	0	0.14	100.972	4	https://ap	Rock Your	audio_fea	spotify:tra	0.818
0.0561	https://ap	Shaggy	0.853	227600	0.606	1Izr43nnX	0	0	0.313	-4.596	1	0	0.0713	94.759	4	https://ap	It Wasn't N	audio_fea	spotify:tra	0.654
0.0212	https://ap	Usher	0.881	250373	0.788	0XUfyU2Q	0	2	0.0377	-4.669	1	0	0.168	104.997	4	https://ap	Yeah!	audio_fea	spotify:tra	0.592
0.257	https://ap	Usher	0.662	223440	0.507	68vgtRHr7	0	5	0.0465	-8.238	1	0	0.118	86.412	4	https://ap	My Boo	audio_fea	spotify:tra	0.676
0.158	https://ap	The Pussyc	0.544	225560	0.823	3BxWKCI0	0	2	0.268	-4.318	1	0	0.32	210.75	4	https://ap	Buttons	audio_fea	spotify:tra	0.434
0.273	https://ap	Destiny's C	0.713	271333	0.678	7H6ev70W	0	5	0.149	-3.525	0	0	0.102	138.009	4	https://ap	Say My Na	audio_fea	spotify:tra	0.734
0.103	https://ap	OutKast	0.728	235213	0.974	2PpruBYCc	0.000532	4	0.175	-2.261	0	0	0.0665	79.526	4	https://ap	Hey Ya! - R	audio_fea	spotify:tra	0.965
0.0569	https://ap	Nelly Furta	0.808	242293	0.97	2gam98EZ	6.13E-05	10	0.154	-6.098	0	0	0.0506	114.328	4	https://ap	Promiscuo	audio_fea	spotify:tra	0.868
0.00206	https://ap	Jesse McC	0.71	211693	0.553	4Y45aqo9	5.48E-05	4	0.0469	-4.722	0	0	0.0292	99.005	4	https://ap	Right Whe	audio_fea	spotify:tra	0.731
0.0759	https://ap	Jesse McC	0.66	214227	0.666	1HwpWwa	0	9	0.0268	-4.342	1	0	0.0472	89.975	4	https://ap	Beautiful S	audio_fea	spotify:tra	0.933
0.0352	https://ap	Jesse McC	0.693	216880	0.709	20ORwCJu	3.40E-06	9	0.0688	-5.787	1	0	0.0608	79.237	4	https://ap	Leavin'	audio_fea	spotify:tra	0.889
0.352	https://ap	Cassie	0.803	192213	0.454	7k6IzwMG	0	8	0.0655	-4.802	0	0	0.0294	99.99	4	https://ap	Me & U	audio_fea	spotify:tra	0.739
0.189	https://ap	Omarion	0.775	256427	0.731	1Bv0Yl01x	0	8	0.129	-5.446	1	0	0.134	131.105	4	https://ap	Ice Box	audio_fea	spotify:tra	0.821
6.79E-05	https://ap	Avril Lavig	0.487	204000	0.9	4omisSITk	0	0	0.358	-4.417	1	0	0.0482	149.937	4	https://ap	Sk8er Boi	audio_fea	spotify:tra	0.484
0.0246	https://ap	Chris Brow	0.846	229867	0.482	7xYnUQlgl	0	1	0.393	-6.721	0	0	0.129	100.969	4	https://ap	Run It!	audio_fea	spotify:tra	0.212
0.0708	https://ap	BeyoncÃ©	0.705	210453	0.796	6d8A5sAxS	0	7	0.388	-6.845	1	0	0.267	166.042	4	https://ap	Check On I	audio_fea	spotify:tra	0.864
0.00543	https://ap	Destiny's C	0.771	230200	0.685	4pmc2AxS	0.00157	1	0.0537	-4.639	1	0	0.0567	88.997	4	https://ap	Jumpin', Ju	audio_fea	spotify:tra	0.683

Fig 10.2 playlist-song feature vectors

```
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/tensorflow/python/ops/math_ops.py:3066: to_int32 (from tensorflow.python.ops.math_ops) is deprecated and will be remov
Instructions for updating:
Use tf.cast instead.
WARNING:tensorflow:From /usr/local/lib/python3.6/dist-packages/tensorflow/python/ops/math_grad.py:102: div (from tensorflow.python.ops.math_ops) is deprecated and will be removed in
Instructions for updating:
Deprecated in favor of operator or tf.math.divide.
Epoch 1/20
19/19 [==============================] - 22s 1s/step - loss: 1.0144 - reg_output_loss: 0.0712 - key_output_loss: 0.6451 - mode_output_loss: 0.1822 - timesig_output_loss: 0.1159 - va
Epoch 2/20
19/19 [==============================] - 17s 895ms/step - loss: 0.9169 - reg_output_loss: 0.0186 - key_output_loss: 0.6424 - mode_output_loss: 0.1737 - timesig_output_loss: 0.0822 -
Epoch 3/20
19/19 [==============================] - 18s 929ms/step - loss: 0.9028 - reg_output_loss: 0.0120 - key_output_loss: 0.6408 - mode_output_loss: 0.1695 - timesig_output_loss: 0.0804 -
Epoch 4/20
19/19 [==============================] - 17s 915ms/step - loss: 0.9101 - reg_output_loss: 0.0185 - key_output_loss: 0.6407 - mode_output_loss: 0.1690 - timesig_output_loss: 0.0819 -
Epoch 5/20
19/19 [==============================] - 17s 891ms/step - loss: 0.8949 - reg_output_loss: 0.0103 - key_output_loss: 0.6390 - mode_output_loss: 0.1664 - timesig_output_loss: 0.0793 -
Epoch 6/20
19/19 [==============================] - 17s 894ms/step - loss: 0.8950 - reg_output_loss: 0.0099 - key_output_loss: 0.6390 - mode_output_loss: 0.1670 - timesig_output_loss: 0.0790 -
Epoch 7/20
19/19 [==============================] - 17s 890ms/step - loss: 0.8927 - reg_output_loss: 0.0094 - key_output_loss: 0.6389 - mode_output_loss: 0.1658 - timesig_output_loss: 0.0787 -
Epoch 8/20
19/19 [==============================] - 18s 952ms/step - loss: 0.8913 - reg_output_loss: 0.0104 - key_output_loss: 0.6382 - mode_output_loss: 0.1648 - timesig_output_loss: 0.0779 -
Epoch 9/20
19/19 [==============================] - 17s 892ms/step - loss: 0.8877 - reg_output_loss: 0.0089 - key_output_loss: 0.6373 - mode_output_loss: 0.1644 - timesig_output_loss: 0.0770 -
Epoch 10/20
19/19 [==============================] - 17s 885ms/step - loss: 0.8862 - reg_output_loss: 0.0086 - key_output_loss: 0.6369 - mode_output_loss: 0.1640 - timesig_output_loss: 0.0766 -
Epoch 11/20
19/19 [==============================] - 17s 917ms/step - loss: 0.8867 - reg_output_loss: 0.0085 - key_output_loss: 0.6372 - mode_output_loss: 0.1643 - timesig_output_loss: 0.0767 -
Epoch 12/20
19/19 [==============================] - 17s 903ms/step - loss: 0.8837 - reg_output_loss: 0.0083 - key_output_loss: 0.6367 - mode_output_loss: 0.1626 - timesig_output_loss: 0.0761 -
Epoch 13/20
19/19 [==============================] - 18s 932ms/step - loss: 0.8858 - reg_output_loss: 0.0090 - key_output_loss: 0.6371 - mode_output_loss: 0.1636 - timesig_output_loss: 0.0761 -
```

Fig 10.3 Model Training

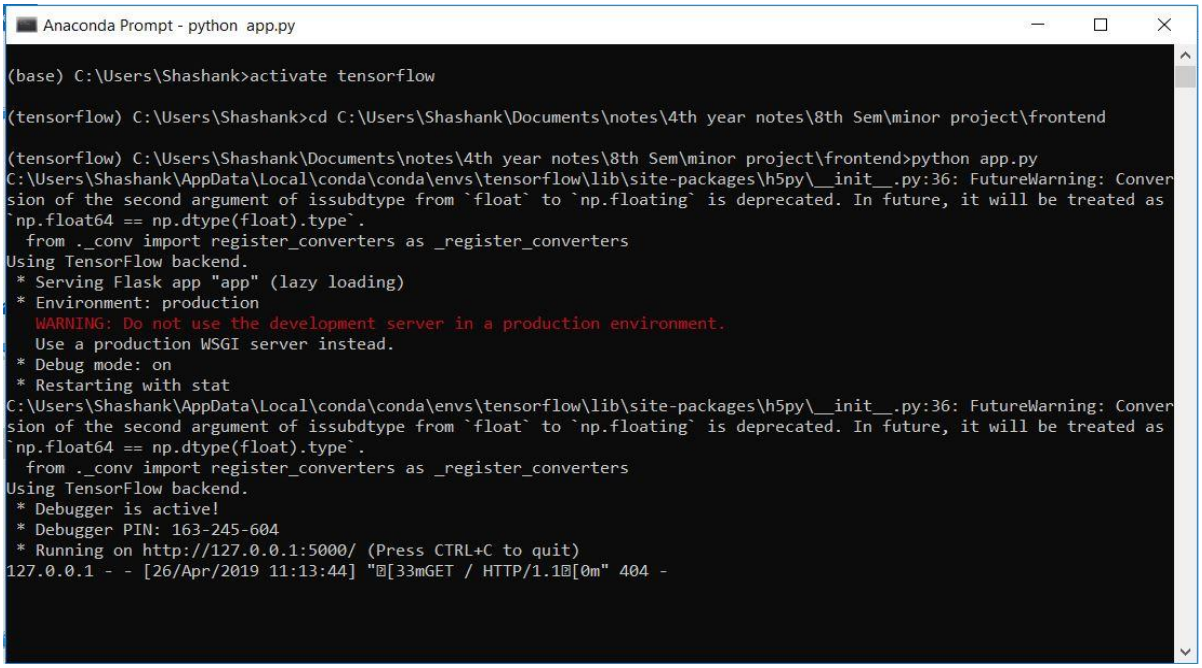

Fig 10.4 Flask Server running

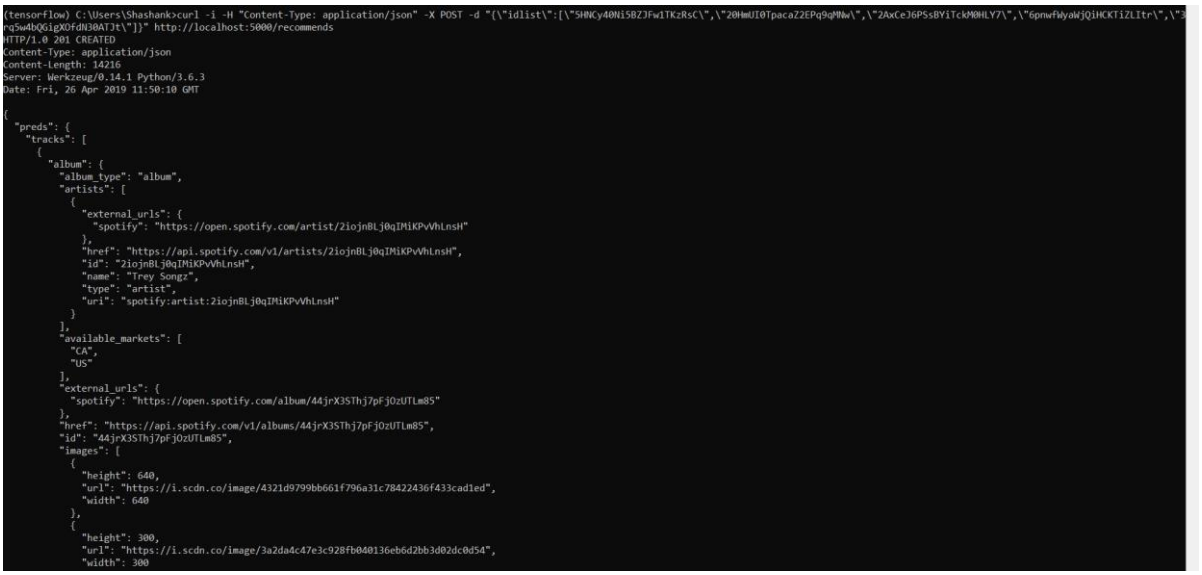

Fig 10.5 Output response from API(song details as JSON)

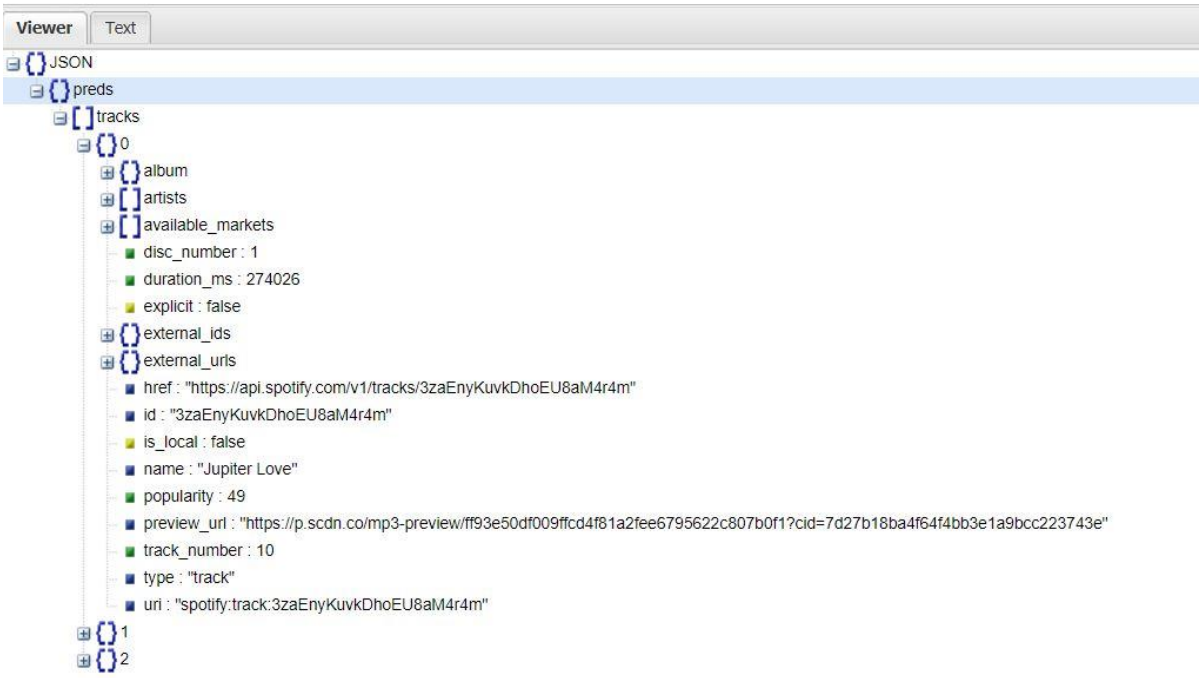

Fig 10.6 Output JSON structure

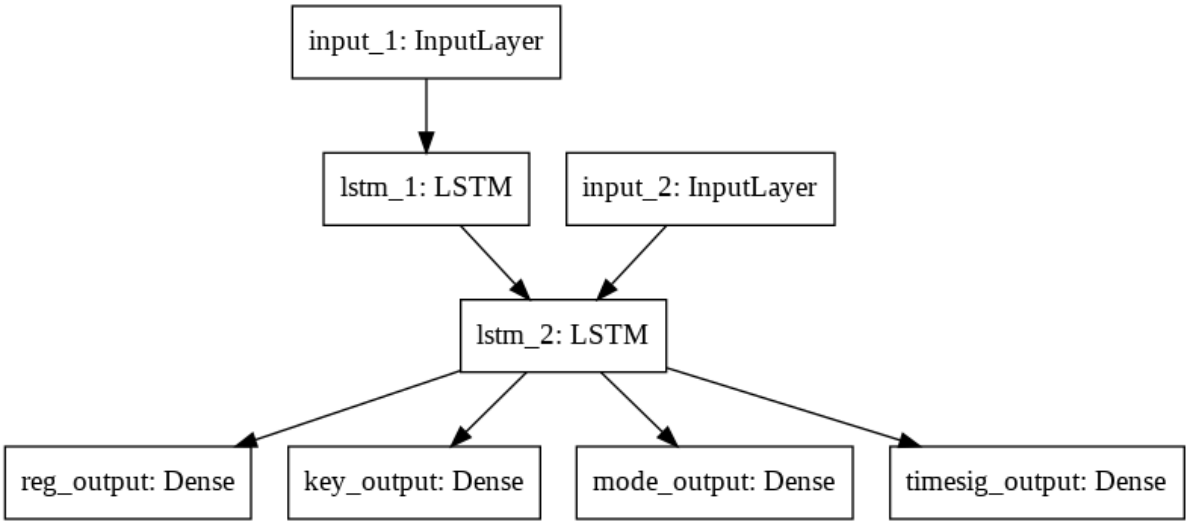

Fig 10.7 Training Model Graph

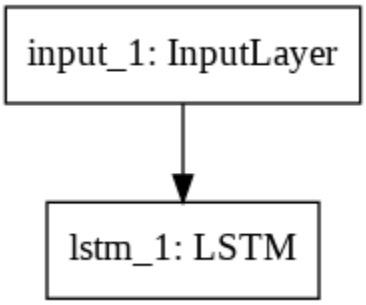

Fig 10.8 Inference Encoder Model Graph

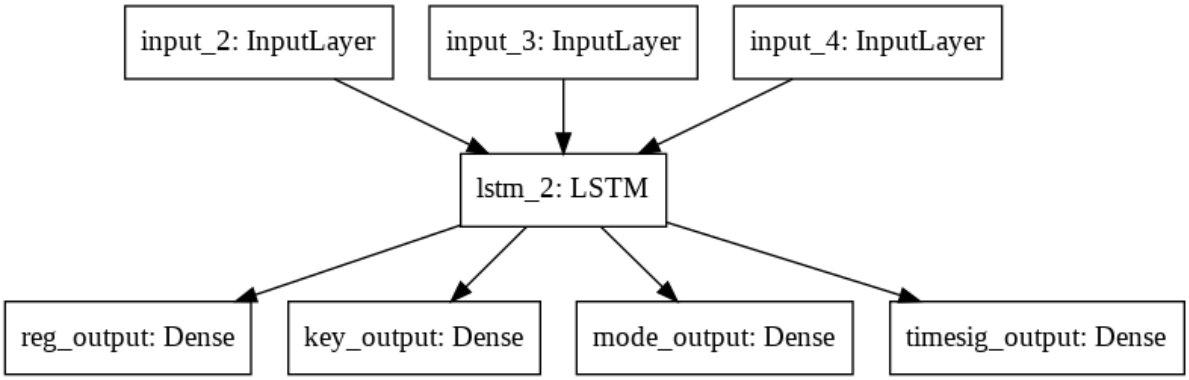

Fig 10.9 Inference Decoder Model Graph

CHAPTER-11

CONCLUSION

The recommender was successful in making predictions based on input data from the users. From the results it can be seen that it has comparable performance to the model presented in [6] while outperforming the regular LSTM and Matrix factorization methods. We feel that the performance can be further improved by using multiple layers. In comparison to [6] our architecture is more scalable as we perform regression to generate recommendations as compared to classification on the entire song vocabulary as this would completely exceed memory requirements with larger song vocabularies. However, the drawback of our model is that the initial predictions are slow as the server has to build the KDTree on the entire song vocabulary. Furthermore, the chance of exceeding recursion limit in building the tree is high with increased song vectors.

CHAPTER-12

FURTHER ENHANCEMENTS

As this is a model based on encoders and decoders, the accuracy of this model can be vastly improved by using Attention mechanism. The model can also be improved by combining both Meta-Data features such as genre,year,popularity,artist etc. and audio features to generate song recommendations. To add to this, we can also take into account user data such as gender, age group, location etc. to further enhance personalization and performance of the system. [2] Even talks about coupling emotion data with the above mentioned techniques to generate recommendations based on mood.

REFERENCES/BIBLIOGRAPHY

Published Papers

1. "A comparative study of music recommendation systems (2018) - Ashish Patel, Dr. Rajesh Wadhvani"
2. "Current challenges and visions in music recommender systems research (2018) - Markus Schedl , Hamed Zamani , Ching-Wei Chen, Yashar Deldjoo , Mehdi Ela"
3. "Music recommendation based on semantic audio segment similarity (2008) - Alessandro Bozzon, Giorgio Prandi, Giuseppe Valenzise and Marco Tagliasacchi Politecnico di Milano"
4. "Collaborative Filtering for Music Recommender Systems (2017) - Elena Shakirova"
5. "What to play next? A RNN-based music recommendation system (2017) - Miao Jiang, Ziyi Yang, Chen Zhao"
6. "Music Recommendation using Recurrent Neural Networks - Ashustosh Choudhary, Mayank Agarwal"

REFERENCES/BIBLIOGRAPHY

Websites

1. https://github.com/vaslnk/Spotify-Song-Recommendation-ML ⇒ Spotify Dataset
2. https://en.wikipedia.org/wiki/Recommender_system
3. https://keras.io/ ⇒ Keras documentation
4. https://machinelearningmastery.com/encoder-decoder-attention-sequence-to-sequence-prediction-keras/ ⇒ Tutorial on encoder decoders
5. https://datascience.stackexchange.com/ ⇒ ML queries
6. https://developer.spotify.com/documentation/web-api/reference/ ⇒ Spotify API documentation
7. http://sdsawtelle.github.io/blog/output/mean-average-precision-MAP-for-recommender-systems.html ⇒ explanation for MAP-K
8. https://docs.scipy.org/doc/numpy/reference/ ⇒ Numpy documentation
9. https://pandas.pydata.org/pandas-docs/stable/ ⇒ Pandas documentation
10. https://stackoverflow.com/questions/tagged/python⇒ Python forum
11. https://github.com/yashu-seth/dummyPy/tree/master/dummyPy⇒ DummyPy Library
12. https://yashuseth.blog/2017/12/14/how-to-one-hot-encode-categorical-variables-of-a-large-dataset-in-python/ ⇒ One Hot encoding tutorial
13. https://blog.miguelgrinberg.com/post/designing-a-restful-api-with-python-and-flask ⇒ Tutorial to build RESTful API with flask