


T.C.

ÇUKUROVA UNIVERSITY

ENGINEERING FACULTY

COMPUTER ENGINEERING DEPARTMENT

CEN438 - Graduation Thesis

Playlist Recommendation System

by

Aygül DİKMEN **Büşra ORAN**

2017555015 **2017555043**

⟨dikmena@student.cu.edu.tr⟩ ⟨busraoran043@gmail.com⟩

Advised by

Res.Assist.Dr. Barış ATA

10 June, 2022

Abstract

In this work, a Recommendation System (RS) is being used in order to make a playlist for current users. RS uses “spotipy”, “pandas”, “sckitlearn”, and finally “textblob” libraries and cosine similarity in order to make the best recommendations. In the backend of this project, Django and Python, in the frontend React.Js, Material UI, and Semantic UI, finally for the database MongoDB is being used. The whole work is divided by the sections, first one is introduction, after the introduction part the explanation of the Spotify Web API is following the first section, in the third section there is the whole implementation of the project and finally as the 4th section there is a conclusion to briefly covers whole work.

Contents

FIGURE LIST

1 INTRODUCTION

Observing the needs of the user in sales and marketing and taking action according to these needs is the most important and most profitable area of marketing. From the past to the present, people who want to sell something use information about buyers. These needs can be estimated more accurately using available physical and behavioral data. The estimation of user needs is used in advertising today to increase profits. The main reason for using recommendation systems is to recommend the most suitable product to the user or to provide the best user experience.

In this project, Spotify API, and Spotipy open source code library was used to retrieve user data.

2 PRELIMINARIES

2.1 Spotify

Spotify is an application that allows its users to listen to music or podcasts. There are millions of tracks and episodes on Spotify. By creating an account, you can listen to music for a monthly fee or for free, with ads. In this application, which allows users to create their own music lists, daily recommendation lists are also made with user analysis.

In this project, Spotify's API for developers and the Spotipy library were used. The rest of this section explains Spotipy and Spotify Web API.

2.1.1 Spotify API

Spotify API is an interface that programs can use to retrieve and manage Spotify data over the internet. The Spotify Web API is based on REST principles. Data resources are accessed via standard HTTPS requests in UTF-8 format to an API endpoint.

2.1.1.2 Spotify Web API Authorization Code Flow

In order to do some complicated operations, Client Credentials Flow is not advanced enough to handle this kinds of situations since almost all of them requires an access token which is unique for users. Therefore in complex circumstances, authorization code flow must be managed in projects. For Spotify API, there are some list of rules to follow in order to gain access tokens and refresh token mechanism. In the documentation of Spotify Web API, there is an virtual demonstration how the authorization flow works like in figure 1.

In this flow, there is a bunch of instructions for a developer to apply them for redirecting its users to Spotify and asks if they allow the app for using their personal datas. Whenever user allows app to reach its informations, the project will be started to get and post some datas from it.

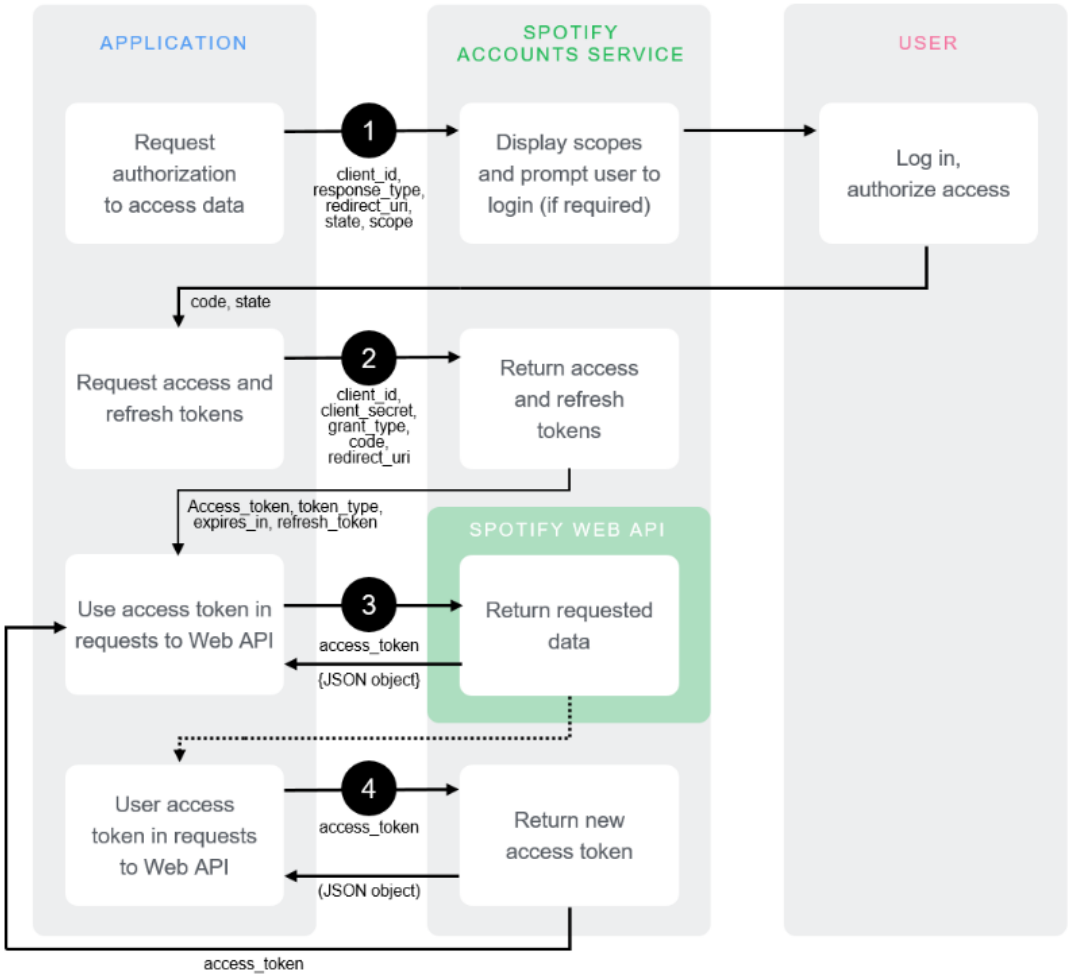

Figure 1: Authorization Code Flow Representation[1]

Basically, when a user gives permission to a service for managing and processing its datas it is called authentication but when a user gives permission to an application to manage its personal datas without communicating directly through the app but via a service such as Spotify accounts service in this case it is so called authorization. Simply the whole idea behind this flow is asking user to login with Spotify and after getting the token informations and keeping them into database redirecting to the main application and starting to getting or posting data with these tokens.

Where possible, Web API uses appropriate HTTP verbs for each action:

METHOD	ACTION
GET	Retrieves resources
POST	Creates resources
PUT	Changes and/or replaces resources or collections
DELETE	Deletes resources

Responses: Web API responses normally include a JSON object.

Timestamps: Timestamps are returned in ISO 8601 format as Coordinated Universal Time (UTC) with a zero offset: YYYY-MM-DDTHH:MM:SSZ. If the time is imprecise (for example, the date/time of an album release), an additional field indicates the precision; see for example, release_date in an album object.

Conditional Requests: Most API responses contain appropriate cache-control headers set to assist in client-side caching:

- If you have cached a response, do not request it again until the response has expired.
- If the response contains an ETag, set the If-None-Match request header to the ETag value.
- If the response has not changed, the Spotify service responds quickly with '304 Not Modified' status, meaning that your cached version is still good and your application should use it.

Spotify URIs and IDs: In requests to the Web API and responses from it, you will frequently encounter the following parameters:

- **Spotify URI:** The resource identifier that you can enter, for example, in the Spotify Desktop client's search box to locate an artist, album, or track. To find a Spotify URI simply right-click (on Windows) or Ctrl-Click (on a Mac) on the artist's or album's or track's name.
- **Spotify ID:** The base-62 identifier that you can find at the end of the Spotify URI (see above) for an artist, track, album, playlist, etc. Unlike a Spotify URI, a Spotify ID does not clearly identify the type of resource; that information is provided elsewhere in the call.
- **Spotify category ID:** The unique string identifying the Spotify category.
- **Spotify user ID:** The unique string identifying the Spotify user that you can find at the end of the Spotify URI for the user. The ID of the current user can be obtained via the Web API endpoint.
- **Spotify URL:** An HTML link that opens a track, album, app, playlist or other Spotify resource in a Spotify client (which client is determined by the user's device and account settings at play.spotify.com.

Response Status Codes:

HTTP status codes are three-digit responses from the server to the browser-side request. The Hypertext Transfer Protocol (HTTP) is an application-level protocol for distributed, collaborative, hypermedia information systems. It is a generic, stateless, protocol which can be used for many tasks beyond its use for hypertext, such as name servers and distributed object management systems, through extension of its request methods, error codes and headers.(RFC 2616).[3] Web API uses the following response status codes, as defined in the RFC 2616.

200: OK - The request has succeeded. The client can read the result of the request in the body and the headers of the response.

201: Created - The request has been fulfilled and resulted in a new resource being created.

202: Accepted - The request has been accepted for processing, but the processing has not been completed.

204: No Content - The request has succeeded but returns no message body.

304: Not Modified

400: Bad Request - The request could not be understood by the server due to malformed syntax. The message body will contain more information; see Response Schema.

401: Unauthorized - The request requires user authentication or, if the request included authorization credentials, authorization has been refused for those credentials.

403: Forbidden - The server understood the request, but is refusing to fulfill it.

404: Not Found - The requested resource could not be found. This error can be due to a temporary or permanent condition.

429: Too Many Requests - Rate limiting has been applied.

500: Internal Server Error. You should never receive this error because our clever coders catch them all … but if you are unlucky enough to get one, please report it to us through a comment at the bottom of this page.

502: Bad Gateway - The server was acting as a gateway or proxy and received an invalid response from the upstream server.

503: Service Unavailable - The server is currently unable to handle the request due to a temporary condition which will be alleviated after some delay. You can choose to resend the request again.

Response Schema

Web API uses two different formats to describe an error:

Authentication Error Object: Whenever the application makes requests related to authentication or authorization to Web API, such as retrieving an access token or refreshing an access token, the error response follows RFC 6749 on the OAuth 2.0 Authorization Framework. It contains error and error_description which are strings.

Regular Error Object: Apart from the response code, unsuccessful responses return a JSON object. It contains status and a message.[4]

Authentication

All requests to Web API require authentication. This is achieved by sending a valid OAuth access token in the request header. Spotify implements the OAuth 2.0 authorization framework:

Figure 2: Web API Authentication Flow[17]

Where:

- End User corresponds to the Spotify user.
- My App is the client that requests access to the protected resources.
- Server which hosts the protected resources and provides authentication and authorization via OAuth 2.0.

2.1.2 Spotipy

Spotipy is a lightweight Python library for the Spotify Web API. With Spotipy you get full access to all of the music data provided by the Spotify platform.

Spotipy supports all of the features of the Spotify Web API including access to all end points, and support for user authorization. All methods in Spotipy library requires user authentication. A developer who needs to use Spotipy, needs to register "My Dashboard" page in the Spotify API and log in to get the credentials necessary to make authorized calls.

There are two authorization flows:

1. Authorization Code Flow
2. Client Credentials Flow

Some of the Functions

- album(album_id) : returns a single album given the album's ID, URIs or URL
- artist(artist_id) : returns a single artist given the artist's ID, URI or URL
- current_user_playlists(limit=50, offset=0): Get user playlists without his/her profile parameters.
- current_user(): Get detailed information about user.

- user_playlist_create(user, name, public=True, collaborative=False, description=”): Creates a playlist for user.
- user_playlist_add_tracks(user, playlist_id, tracks, position=None): Get a list of tracks and add them to the given playlist.
- audio_features(tracks[]): Get audio features for one or multiple tracks.
- user_playlist_unfollow(user, playlist_id): Deletes a playlist for a user.
- playlist_tracks(playlist_id, fields=None, limit=100, offset=0, market=None, additional_types=('track',)) : Get full details of the tracks of a playlist.[14]

2.2 Recommendation System

This part contains the techniques that are used in our recommendation system. Firstly, the recommendation system, then one hot encoding technique, and lastly sentiment analysis with Term Frequency-Inverse Document Frequency (TF-IDF) is explained.

Recommendation System is an intelligent system that makes suggestions about items to users that might interest them. Some of the practical applications that use such systems include recommending books, cd etc. on Amazon.com, movies by Movielens, music by last.fm and news at VERSIFI technologies.

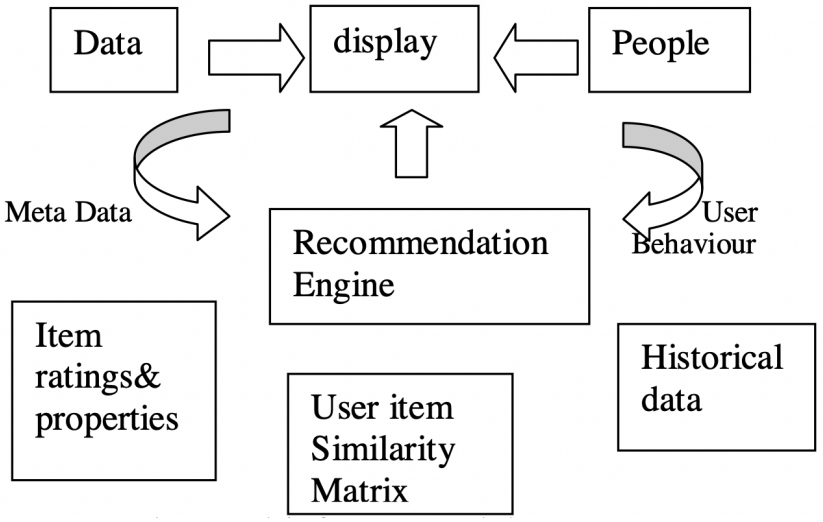

Figure 3 : Model of Recommendation Process [18]

In a recommendation-system application there are two classes of entities, which we shall refer to as users and items. The formal definition of recommender system is:

• C: The set of all users

• S: The set of all possible items that can be recommended, such as books, movies, or restaurants.

• U: A utility function that measures usefulness of a specific item s ∈S to user c ∈C

2.2.1 RECOMMENDATION SYSTEM TECHNIQUES

2.2.1.1 Collaborative Filtering Process:

The Collaborative filtering (CF) systems work by collecting user feedback in the form of ratings for items in a given domain and exploiting similarities in rating behaviour amongst several users in determining how to recommend an item. CF systems recommend an item to a user based on opinions of other users.

Item User	Item1	Item2		Item n	Item j
User1					
User 2					
User i					
User m					

Figure 4: Collaborative Filtering Matrix

- CF algorithms represent the entire user-item space as a rating matrix ‘R’. Each entry Rij in the matrix represents the preference score (rating) if the ith user on the jth item. Each individual rating is within a numerical scale and it can be 0 as well, indicating that a user has not yet rated this item.
- CF problem includes the estimation or prediction of rating for the yet unrated item. For the prediction of rating, similarities between items and users are calculated using different approaches. Thus, the two related problems consist in finding a set of K users that are most similar to a given user and finding a set of K items that are most similar to a given item.

- Finally using these similarities, recommendations that are produced at the output interface can be of two types: Prediction and Recommendation.
- Prediction is a numerical value, Rij, expressing the predicted score of item j for the user i. The predicted value is within the same scale that is used by all users for rating.

2.2.1.2. Content - Based Process:

Content based recommendation systems recommend an item to a user based upon a description of the item and a profile of the user's interests. Such systems are used in recommending web pages, TV programs and news articles etc. All content based recommender systems have few things in common like means for description of items, user profiles and techniques to compare profile to items to identify what is the most suitable recommendation for a particular user.

In content-based recommendation methods, the utility u(c, s) of item s for user c is estimated based on the utilities u(c, si) assigned by user c to items si ∈S that are "similar" to item s. Content-Based recommender systems make suggestions upon
item features and user interest profiles. Typically, personalized profiles are created automatically through user feedback, and describe the type of items a person likes. In order to determine what items to recommend, collected user information is compared against content features of the items to examine.

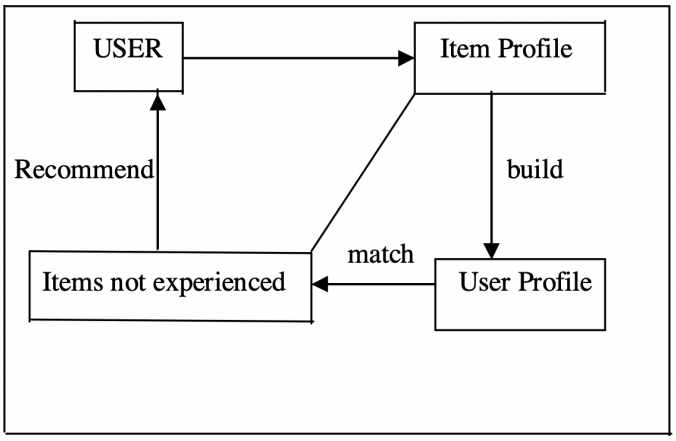

Figure 5: Content - Based Process Schema

- System has a huge database consisting of the items to be recommended and the features of these items and it is termed as Item Profile.

- The users provide some sort of information about their preferences to the system. Combining the item information with the user preferences, the system builds a profile of the users.
- According to the information existing in a target user's profile, the system recommends suitable items to the user.

We are using the content - based process for our recommendation system.

2.2.1.3 Hybrid Process:

Hybrid recommenders are systems that combine multiple recommendation techniques together to achieve a synergy between them. Several researchers have attempted to combine collaborative filtering and content based approaches in order to smoothen their disadvantages and gain better performance while recommending. Depending on domain and data characteristics, several hybridization techniques are possible to combine CF and CB techniques which may generate different outputs. [5]

2.2.1.4 One - Hot Encoding

One - Hot Encoding (OHE) is highly used with machine learning to convert the field of interest into unique label encoding. It represents categorical variables as binary vectors. First, the categorical values are mapped to integer values. Then each integer value is represented as a binary vector where everything has zero value except the index of the integer, which is marked as 1.

The advantage of OHE is a vector representation where all the elements of the vector are zero except one which has 1 as its value. In other words, it transforms the category feature to a format that works better with a classification algorithm. OHE is implemented at the pre-processing state of the dataset. OHE creates high dimension data with a large number of variations in a feature. When the number of variations are high, the representation size grows with the corpus which requires extensive computation and it does not contain any contextual or semantic information embedded in this approach. Here, the limitation is the speed of execution in encoding techniques that are used in machine learning analysis. [6]

2.2.1.4.5 Sentiment analysis using TF–IDF weighting

1. Term frequency–inverse document frequency analysis method

IDF is a logarithmically scaled fraction of the total number of documents in a corpus divided by the number of documents that contain that term. Hence, a term found in only a small amount of documents will have a very high IDF value while an IDF value of 0 indicates that the term is present in all documents. This means that stop words (e.g. ‘‘the’’, ‘‘and’’, ‘‘a’’) will have a low IDF value. TF–IDF value for a term in a document is calculated by multiplying the frequency of that term within the document with the IDF value of that term. A list containing terms ordered by TF–IDF score presents terms frequently used in the document (but uncommon in the corpus) at the top of the list, and terms common in all documents at the bottom of the list. This removes common stop words from our analysis, which is useful as stop words do not convey semantic information but are instead used to structure a sentence.[3]

Term Frequency (TF): The number of times a term appears in each document divided by the total word count in the document.

Inverse Document Frequency (IDF): The log value of the document frequency. Document frequency is the total number of documents where one term is present.[4]

Based on the above, the TF–IDF in a document d, in a corpus of documents D, for a term t is formulated as:

tfidf(t,d,D)=tf(t,d)∗idf(t,D)

Term frequency tf (t , d) is the number of times term t appears in document d. Inverse document frequency idf (t , D) for a term t in a corpus of documents D is:

$$idf(t, D) = log \frac{N}{|(d \in D : t \in d)|}$$

where N is the total number of documents in the corpus D; and |(d ∈ D : t ∈ d)| is the number of documents where the term t appears.[3]

2.2.1.4.6 TextBlob

TextBlob is a Python (2 and 3) library for processing textual data. It provides a simple API for diving into common natural language processing (NLP) tasks such as part-of-speech tagging, noun phrase extraction, sentiment analysis, classification, translation, and more.

Features

- Noun phrase extraction
- Part-of-speech tagging
- Sentiment analysis
- Classification (Naive Bayes, Decision Tree)
- Tokenization (splitting text into words and sentences)
- Word and phrase frequencies
- Parsing
- n-grams
- Word inflection (pluralization and singularization) and lemmatization
- Spelling correction
- Add new models or languages through extensions
- WordNet integration

TextBlob allows you to specify which algorithms you want to use under the hood of its simple API.

Sentiment Analyzers

Sentiment analysis is the process of computationally identifying and categorizing the opinions expressed in a piece of text, especially in order to determine the writer's attitude towards a particular topic, product, etc. is positive, negative or neutral[5]. Sentiment analysis can be highly useful in several cases. The best example is the marketing methodology. Marketing teams can use sentiment analysis to launch a new product or to determine the existing product popularity and preference.[6]

The textblob.sentiments module contains two sentiment analysis implementations, PatternAnalyzer (based on the pattern library) and NaiveBayesAnalyzer (an NLTK classifier trained on a movie reviews corpus). The default implementation is PatternAnalyzer, but you can override the analyzer by passing another implementation into a TextBlob's constructor.

2.2.1.4.7 Similarity and Recommendation

After retrieving the playlist summarized vector and the non-playlist songs, the similarity between each individual song in the database and the playlist can be found. The similarity metric chosen in this project is cosine similarity.

Cosine similarity is a mathematical value that measures the similarities between vectors. Imagining the song vectors as only two-dimensional, the visual representation would look similar to the figure below.

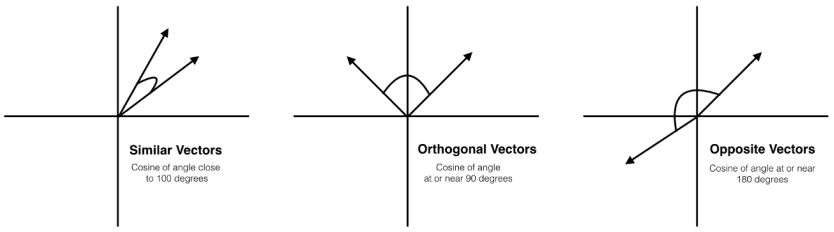

Figure 6: Cosine Similarity measure illustration. Image by DeepAI [19]

The Cosine Similarity measurement begins by finding the cosine of the two non-zero vectors. This can be derived using the Euclidean dot product formula which is written as:

$$\mathbf{A} \cdot \mathbf{B} = \|\mathbf{A}\| \, \|\mathbf{B}\| \cos\theta$$

Then, given the two vectors and the dot product, the cosine similarity is defined as:

$$\text{Cosine Sim}(A, B) = \frac{A \cdot B}{||A|| \times ||B||} = \frac{\sum_{i=1}^{n} A_i \times B_i}{\sqrt{\sum_{i=1}^{n} A_i^2} \times \sqrt{\sum_{i=1}^{n} B_i^2}}$$

The output will produce a value ranging from -1 to 1, indicating similarity where -1 is non-similar, 0 is orthogonal (perpendicular), and 1 represents total similarity.[7]

2.3 Django

Django is basically a high-level Python web framework which helps to build secure, rapid, scalable, portable, maintainable projects. Since it was open-sourced in 2005, now it has numerous contributors to improve it. One of the reasons why it is so popular is its versatility. Because with Django it is possible to build almost any type of website such as management systems or social networks, etc. [2] Therefore Django is one of the most popular web frameworks throughout the world and it looks like it will continue to be so.

2.3.1 Virtual Environments

Before starting a project and working on it, it is suggested to work in a virtual environment to take advantage of its opportunities. Virtual environment is a way for developers to have multiple different versions of Python regardless of what the original version your host machine actually has. Since it provides you an independent development environment you can have necessary software installed in the /bin folder of the virtual environment. Before working in a virtual environment, it is required to go to the directory that keeps our django project and run the “pip -m venv ‘your_virtual_env_name’ ” command. After that it can be seen a folder that will be created in the same folder with the project. And since it is now installed, now it is time to activate the virtual environment by going to the Scripts directory in the virtual environment and run the “activate ” command. If the virtual environment is successfully activated, its name should be added in front of the project directory as it shown in the figure 7.

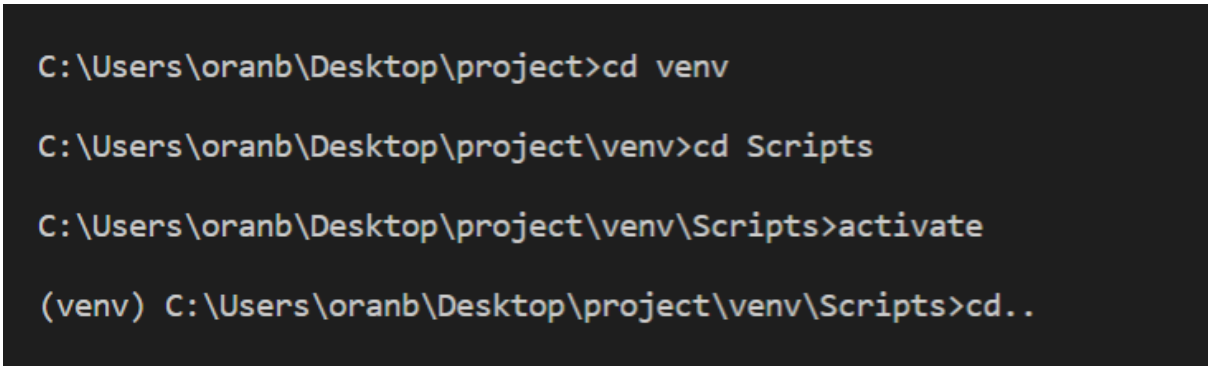

Figure 7: Virtual Environment Implementation

2.3.2 Basic Components of Django

To start a Django project, it is required to command “django-admin startproject ‘your_project_name’ ” on terminal. After that Django will automatically creates the projec t structures. As shown in the figure 8, there are some files belongs in the projects file such as urls.py to store projects urls. In the settings.py file, there are all the settings for whole project such as database settings or application settings. So if there is some other changes that is not attached to the settings.py file it is not possible to see them. It is obligatory to arrange the settings correctly according to the project files. In this case, since the database was changed to MongoDB, the settings file was also changed accordingly like in figure 9.

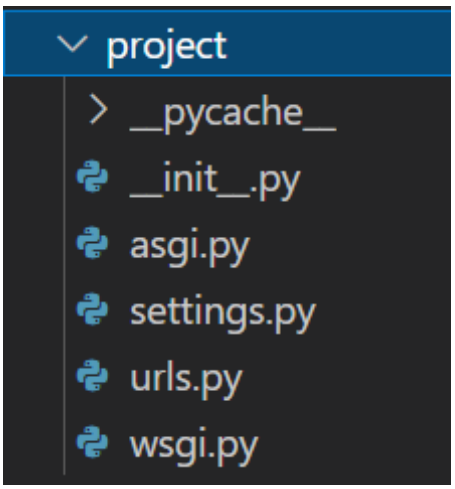

Figure 8: Simple Django Project Content

```
DATABASES = {
    'default': {
        'ENGINE': 'djongo',
        'NAME': 'projectDb',
        'ENFORCE-SCHEMA': False,
    }
}
```

Figure 9: Database settings

For running the server and running the project, use manage.py file which is created when the project was created. With using “py manage.py runserver” command, the localhost server will be automatically start working and when going to the link, it will be seen a page like in figure 10.

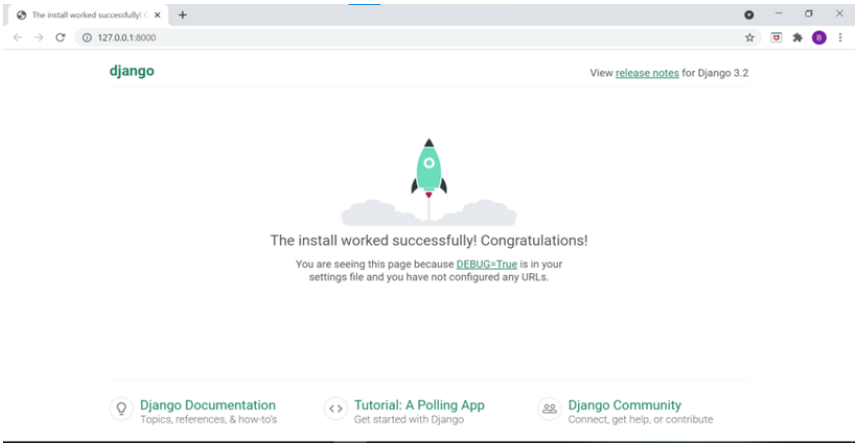

Figure 10: Running Django Project

Every Django project has its own unique secret key and it is not a good idea to share it with others. Since the security issues may happen, in this project, an .env file to store all of our secretive informations was created. Whenever it needs to use a variable in .env file using ‘os.gentenv()’ attribute and also loading the enviroment variables in needed files is enough.

2.3.2.1 Django Apps

With Django apps, the functionality and other features will be added to the project such as database models or views. To create an app, the command “py manage.py startapp ‘your_app_name‘ ” should be runned on the terminal. Like in the figure 11, there will be some separated python files attached to this app.

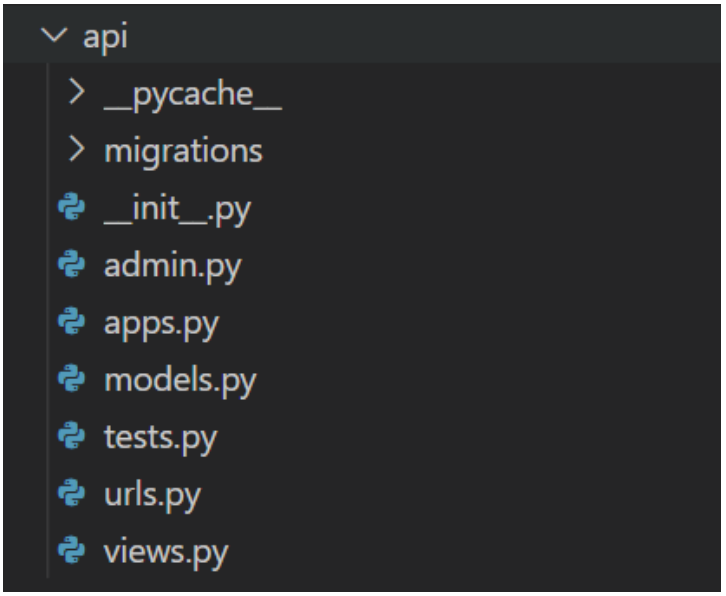

Figure 11: Content of a Basic Django App

After creating an application, its name should be added to the INSTALLED_APPS list in settings.py file otherwise it will not work with this project even though it places in the same directory and folder as project. As shown in figure 12, the project’s structure is divided 2 parts which are backend and frontend apps. For frontend, there is an app called frontend and for backend of the project an app called spotify was created since it will handle all of the functional work of the project.

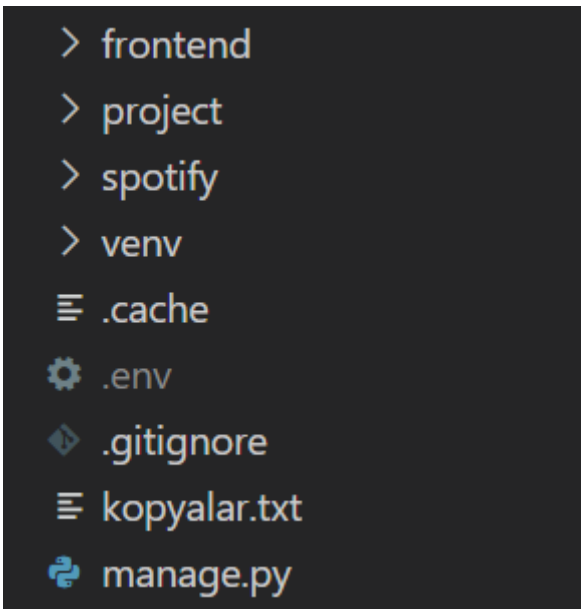

Figure 12: Playlist Recommendation System Structure

2.3.2.2 Django Rest-Framework

Django REST framework is a beneficial and powerful toolkit for building RESTful APIs. For using this we should also add the rest-framework name to the INSTALLED_APPS list. In this project there are some usage of APIView which is imported from rest-framework.

2.4 React

For frontend development of the project, React was used. React is a JavaScript-base UI development library which was created by Facebook and open-sourced in 2013. Compare to the other libraries which have similarities with React.js, there are some reasons behind why it is chosen for usage such as its flexibility and broader community support. Also React is an excellent tool to create interactive applications for web, mobile and other platforms. Because of these reasons, its popularity is increasing day by day.

2.4.1 Connecting React to the Project

After creating the frontend app, it is required to go to the frontend directory in the terminal and wrote a bunch of commands in order for using React and some tool kits works well with it effectively such as Material UI. First of all Node.js must be installed in your

device. And then wrote these following commands for setting the React on your machine correctly.

- npm init -y
- npm i webpack webpack-cli - - save - dev

And there is a need to set up the babel which will help the project to work well with any browser.

- npm i@babel/core babel-loader @babel/preset-env @babel/preset-react - - save - dev
- npm i react react-dom - - save - dev

Now, it is time for setting the Material UI. By using the pages itself can easily be styled.

- npm install@material-ui/core
- npm install @babel/plugin-proposal-class-properties
- npm install react-router-dom
- npm install @material-ui/icons

After running these, adding components to the application became possible. To make it run the React code it also should be run the React server instead of just running the backend server by running “npm run dev” command. It should be noted that whenever something on code changes and the file is being saved, the server will be started again. Thats how the developers can use the rapidness of the React.js.

2.5 MongoDB

NoSQL(not only SQL) databases are non-tabular and therefore they store data differently than relational tables. They provide flexible schemas and scale easily with large amounts of data and high user loads. Furthermore, the Agile Manifesto was rising in popularity, and developers were reconsidering the way they developed softwares. They recognized the importance of the rapid adaption to the new requirements and that’s why the ability to iterate quickly and make changes throughout their software stack, all the way down to the database, became a need. So NoSQL databases gave them this flexibility. MongoDB is the most popular NoSQL database with its features throughout the world. Since NoSQL databases are becoming more and more popular day by day because of the reasons above, in this project the database is chosen as MongoDB. Because for this projects’ case there is only one database table known as model in the Django models.py file.

2.5.1 Connecting Django with MongoDB

When Django projects are created, there is a default database connection with Sqlite, but for this project MongoDB was connected with the Django backend. As first step in models.py file in the spotify app layer, SpotifyToken model was created. For connecting the Django and MongoDB it is required to install some extensions such as djongo and pymongo. And it should be noted that Django has some default models in it and they should be migrated after connecting the database. Like in figure 4 above, settings should be arranged for the MongoDB connection. Also MongoDB and MongoDB compass should be installed to the computer. After the installation, it is required to create a new db on the compass. Since this project's database named as projectDb, the settings was arranged according to it as in figure 4. For transfering the changes, connecting django and db andmaking the database migrations the following two commands must be run in the terminal:

- py manage.py makemigrations
- py manage.py migrate

```
class SpofityToken(models.Model):
    user = models.CharField(max_length=50, unique=True)
    created_at = models.DateTimeField(auto_now_add=True)
    access_token = models.CharField(max_length=150)
    refresh_token= models.CharField(max_length=150)
    token_type = models.CharField(max_length=50)
    expires_in = models.DateTimeField()
```

Figure 13: Spotify Token Database Model

As it is shown in the figure 13, one and only database table in this project is written and its fields are given. The purpose of this table to keep session_id and other fields for using while handling the authorization flow for Spotify Web API such as access_token and expiry date.

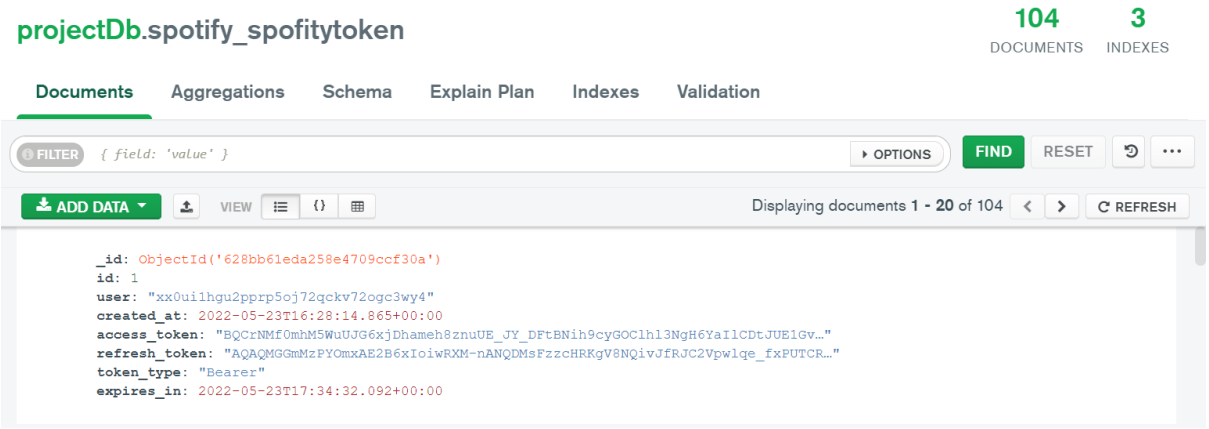

Figure 14: Demonstration of SpotifyToken model

In the Django Compass, it is provided to make a connection on the localhost when the server is running the database will also be upgraded. Also with this application the datas can be filtered and queried in an easy way. In the figure 14 there is a demonstration of how a table data look alike in MongoDB compass.

3 Implementation

In this section, the implementation of the project is explained. Firstly backend, then frontend, and finally Machine Learning had been placed.

3.1 Backend

For handling the backend development of this project, there is an app called spotify and it contains some specific python files as in figure 15 to contain related functionalities.

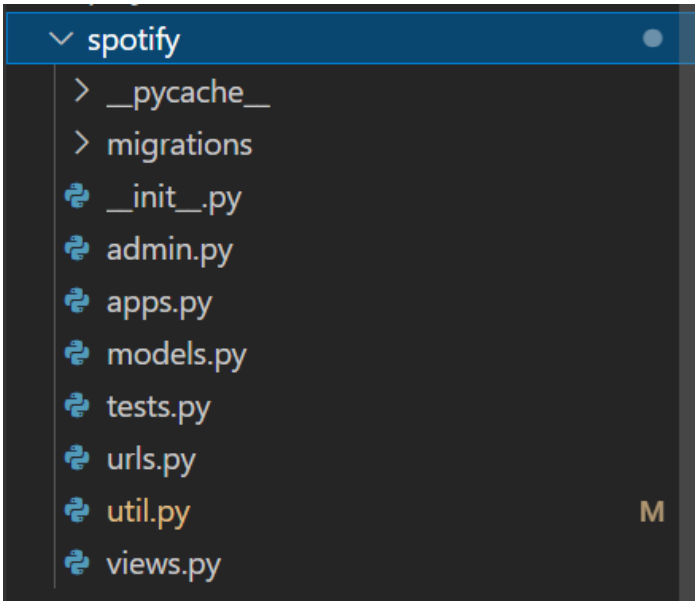

Figure 15: Spotify App Content

Except for the util.py file, rest of the files automatically created when the app is created. In utils.py file there are all of the functions that our application needs in order to use them in the views.py. According to the util.py file, views.py is used for handling the data and requests for rendering the frontend pages in background. In urls.py, all of the views are associated with urls and for using this app's urls it is not enough to just write these urls on the spotify app, it is also necessary to define these app's urls with the projects' urls.py file. After that it is now possible to open them on the browser.

3.1.1.Create an App on Spotify Developers Dashboard

Spotify provides for developers an interface that is different from the normal application which allows software developers to create apps and manage their client

credentials through this site. Since it is needed to have client_id and client_secret informations, the app is created to its informations be used like in figure 16.

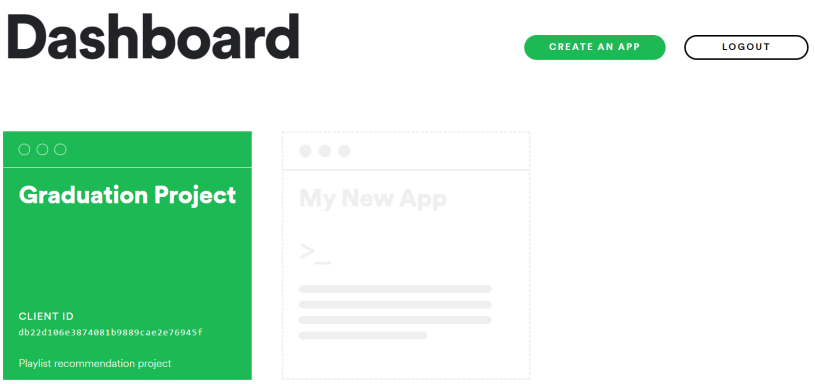

Figure 16: Dashboard app for the project

3.1.2 Content of Util.py

Util.py file consists of all kinds of functions to be used in the views file as mentioned before. This is the only python file that is added to the spotify app after creation just to keep views simple in order to prevent complex codes.

```
def get_user_tokens(session_id):
    user_tokens = SpofityToken.objects.filter(user=session_id)
    if user_tokens.exists():
        return user_tokens[0]
    else:
        return None
```

Figure 17: get_user_tokens() Method

In order to use access token to use spotipy libraries' functions, there is a method called get_user_tokens() was written. In this function, it takes session_id as a parameter and querying the database model calles SpotifyToken if there is any token data is kept by the session in the database. If it finds any entry it returns the latest one if it does not exist then returns None. In this project almost every view is using this function and instead of rewriting the same method multiple times, define it as a function helps not to waste time.

```python
def update_or_create_user_tokens(session_id, access_token, token_type, refresh_token, expires_in):
    tokens = get_user_tokens(session_id)
    expires_in = timezone.now() + timedelta(seconds=expires_in)

    if tokens:
        tokens.access_token = access_token
        tokens.refresh_token = refresh_token
        tokens.expires_in = expires_in
        tokens.token_type = token_type
        tokens.save(update_fields=['access_token',
                    'refresh_token', 'expires_in', 'token_type'])

    else:
        tokens = SpofityToken(user=session_id, access_token=access_token,
                              refresh_token=refresh_token, expires_in=expires_in, token_type=token_type)

        tokens.save()
```

Figure 18: update_or_create_user_tokens() Method

It should be noted that the access token that Spotify gives to use is expiring after 1 hour, so the implementation of a refresh token mechanism is a must for this project. And the method called update_or_create_user_tokens() was written to handling this problem. In this function the token entry is getting and checks if its expired or not by converting the date information into seconds. So if there is any tokens by the session id the function updates the fields for this entry but if there is no such token it creates a brand new token in the database and saves it.

```python
def is_spotify_authenticated(session_id):
    tokens = get_user_tokens(session_id)
    if tokens:
        expire_date = tokens.expires_in
        if expire_date <= timezone.now():
            refresh_spotify_token(session_id)
            return True
    return False
```

Figure 19: is_spotify_authenticated() Method

As shown in figure 19, there is a method called is_spotify_authenticated to check the expiry date and if it turns out that the token is expired, it will call the refresh_spotify_token method and return True but if it is not expired then return False.

```python
def refresh_spotify_token(session_id):
    refresh_token = get_user_tokens(session_id).refresh_token

    response = post('https://accounts.spotify.com/api/token', data={
        'grant_type': 'refresh_token',
        'refresh_token': refresh_token,
        'client_id': os.getenv('SPOTIFY_CLIENT_ID'),
        'client_secret': os.getenv('SPOTIFY_CLIENT_SECRET'),
    }).json()

    access_token = response.get('access_token')
    token_type = response.get('token_type')
    expires_in = response.get('expires_in')
    refresh_token = response.get('refresh_token')

    update_or_create_user_tokens(
        session_id, access_token, token_type, refresh_token, expires_in)
```

Figure 20: refresh_spotify_token() Method

For setting up a refresh token mechanism, there is a method called refresh_spotify_token as shown in the figure 20. In this function, it gets the refresh token information from the specified user with a session id and makes a post request to the spotify api to get a refresh token and getting the response's new fields. After that it calls the update_or_create_user_tokens with sending these informations as parameters.

```python
def execute_spotify_api_request(session_id, endpoint, post_=False, put_=False):
    tokens = get_user_tokens(session_id)
    headers = {
        'Content-Type': 'application/json',
        'Authorization': 'Bearer ' + tokens.access_token
    }

    if post_:
        post(BASE_URL + endpoint, headers=headers)
    if put_:
        put(BASE_URL + endpoint, headers=headers)

    response = get(BASE_URL + endpoint, {}, headers=headers)

    try:
        return response.json()
    except:
        return {'Error': 'We are having issues with request'}
```

Figure 21: execute_spotify_api_request() Method

In the function shown in figure 21, it takes endpoint, session_id, post and put values and creates post or put requests whether which boolean value is True, and gets the response of this request. For preventing the errors, it gets the response in a try except block and returns response of the error message according to the code.

```
def get_playlist(session_id):
    tokens = get_user_tokens(session_id)
    spotify = spotipy.Spotify(auth=tokens.access_token)

    playlists = spotify.current_user_playlists()

    return playlists
```

Figure 22: get_playlist() Method

For the project to get user's playlists there is a function called get_playlist() which basically uses the access token and authenticates the Spotify class to use spotipy function called current_user_playlists() and returns the playlists.

```
def get_playlist_songs(session_id, playlist_id):
    tokens = get_user_tokens(session_id)
    spotify = spotipy.Spotify(auth=tokens.access_token)

    songs = spotify.playlist_tracks(playlist_id=playlist_id)

    return songs
```

Figure 23: get_playlist_songs() Method

Similar to the function above, in figure 23 there is function to get the songs of a specific playlist of a user by using spotipy's playlist_tracks() method.

```python
def get_current_user_id(session_id):
    tokens = get_user_tokens(session_id)
    spotify = spotipy.Spotify(auth=tokens.access_token)

    user_id = spotify.current_user()

    return user_id['id']
```

Figure 24: get_current_user_id() Method

In the method is demonstrated figure 24, with using spotipy's current_user() method the informations of the current user is taken and from whole list, the id information will be returned.

```python
def create_playlist(session_id):
    tokens = get_user_tokens(session_id)
    spotify = spotipy.Spotify(auth=tokens.access_token)

    user = get_current_user_id(session_id)

    spotify.user_playlist_create(user, name="Your Recommended Playlist", public=False, collaborative=True,
                                 description="This is the playlist to reflect your selected playlist. Hope you enjoy listen it.")
```

Figure 25: create_playlist() Method

In this project since its aim to create a recommendation playlist to the user according to the chosen playlist it should be made a function of this feature. So as in figure 25, there is a function to create in user's Spotify account, and its name and description are given by default as a parameter in spotipy's user_playlist_create function.

```python
def add_tracks_to_playlist(session_id, user_id, playlist_id, tracks):
    tokens = get_user_tokens(session_id)
    spotify = spotipy.Spotify(auth=tokens.access_token)

    spotify.user_playlist_add_tracks(user_id, playlist_id, tracks)
```

Figure 26: add_tracks_to_playlist() Method

In the figure 26 the playlist is created but is empty, now its time to add recommended songs that comes from the similarity algorithms to this created playlist. For doing this user_playlist_add_tracks function of spotipy is used and user_id playlist_id and a list that contains recommended songs id's is given as parameters of the function.

```python
def delete_recommended_playlist(session_id):
    tokens = get_user_tokens(session_id)
    spotify = spotipy.Spotify(auth=tokens.access_token)

    playlists = get_playlist(session_id)
    playlist_id = playlists['items'][0]['id']

    user_id = get_current_user_id(session_id)

    spotify.user_playlist_unfollow(user_id, playlist_id)
```

Figure 27: delete_recommended_playlist() Method

Since it is a possibility that the user does not like the playlist the project provides and wants to delete the playlist there is a function is shown figure 27 to manage this feature of the project. In Spotify when a playlist is created it will locates as the top of the playlists and its index is always 0 in the items list. So after getting the playlist the latest playlist is chosen and with using user_playlist_unfollow function it will simply be deleted from the user's profile.

3.1.3 Content of Views.py

In the Django framework, views are Python functions or classes that receive a web request and return a web response. The response can be a HTTP response, HTML template response or just HTTP redirect response. Besides them, there are a bunch of types of views that can be used. In this project both class and function views are written. For providing the data for the frontend of the project APIViews that are imported from rest framework was used. Since the first thing must be handled was authorization through the Spotify WebAPI, as it is mentioned before in section 2.1.1.2, there was a set of rules to follow in order to authorize the project for taking tokens to use them in other views and functions. For the first step, like in figure 28 the user's authentication must be checked if it returns True or False. If user is not authenticated, in the frontend part the function will call the AuthURL view as shown in figure 29.

```python
class IsAuthenticated(APIView):
    def get(self, request, format=None):
        is_authenticated = is_spotify_authenticated(
            self.request.session.session_key)

        return Response({'status': is_authenticated}, status=status.HTTP_200_OK)
```

Figure 28: IsAuthenticted View

```python
class AuthURL(APIView):

    def get(self, request, format=None):
        scopes = 'playlist-read-collaborative playlist-modify-public playlist-read-private playlist-modify-private'

        url = Request('GET', 'https://accounts.spotify.com/authorize', params={
            'scope': scopes,
            'response_type': 'code',
            'redirect_uri': os.getenv('REDIRECT_URI'),
            'client_id': os.getenv('SPOTIFY_CLIENT_ID')
        }).prepare().url

        return Response({'url': url}, status=status.HTTP_200_OK)
```

Figure 29: AuthURL View

After this view the data url is prepared which includes the authorization code. So after taking this data the view will redirect to the redirect page and from the view in figure 25 making a post request to the spotify api for receiving access token and refresh token, etc. it is checked if the session exists with the same session key. If there is no session key it will create a session and call the update_or_create_user_tokens() method and give all of the data it received from the spotify api call. The last step in this authorization view is redirecting to to frontend's my-playlists page to show the user's current playlists.

```python
def spotify_callback(request, format=None):
    code = request.GET.get('code')
    error = request.GET.get('error')

    response = post('https://accounts.spotify.com/api/token', data={
        'grant_type': 'authorization_code',
        'code': code,
        'redirect_uri': os.getenv('REDIRECT_URI'),
        'client_id': os.getenv('SPOTIFY_CLIENT_ID'),
        'client_secret': os.getenv('SPOTIFY_CLIENT_SECRET')
    }).json()

    access_token = response.get('access_token')
    token_type = response.get('token_type')
    refresh_token = response.get('refresh_token')
    expires_in = response.get('expires_in')
    error = response.get('error')

    if not request.session.exists(request.session.session_key):
        request.session.create()

    update_or_create_user_tokens(
        request.session.session_key, access_token, token_type, refresh_token, expires_in)

    return redirect('frontend:my-playlists')
```

Figure 30: spotify_callback View

After the user authentication, the project is handling the other functionalities such as getting user's playlists, making a recommendation playlist. As shown in the figure 30, there is a view called GetPlaylist for getting both playlist and songs that belong to it as an APIView. Since the spotify api gives us a very detailed list of playlists and songs it is not necessary to reflect all of these information. Hence in the view with using nested loops to iterate the items of the songs and playlists, there is a custom list was providing for each user and returns the response to the frontend app of the project to display these data effectively.

```
class GetPlaylist(APIView):
    def get(self, request, format=None):
        session_id = self.request.session.session_key
        response = get_playlist(session_id)

        playlist = []
        pl_songs = []

        for i in range(len(response['items'])):
            playlist_id = response['items'][i]['id']
            songs_list = get_playlist_songs(session_id, playlist_id)
            for j in range(len(songs_list['items'])):
                pl_songs.append(
                    {
                        'name': songs_list['items'][j]['track']['name'],
                        'artist': songs_list['items'][j]['track']['album']['artists'][0]['name'],
                        'img': songs_list['items'][j]['track']['album']['images'][0]['url']
                    }
                )

            playlist.append(
                {'name': response['items'][i]['name'],
                 'cover': response['items'][i]['images'][0]['url'],
                 'id': response['items'][i]['id'],
                 'owner': response['items'][i]['owner']['display_name'],
                 'songs': pl_songs,
                }
            )

            pl_songs = []

        return Response(playlist, status=status.HTTP_200_OK)
```

Figure 31: GetPlaylist View

In MakePlaylistRecommendation view both includes machine learning algorithms and the usage of the functions that are mentioned previously in the util.py section defined as 3.1.2. Therefore the machine learning part will be explained in the section 3.3. And since there is also an api view for this part, after getting the recommended songs with its features their id column is separated as a list and is used to create a recommended playlist with these list in user's account as it is demonstrated in figure 31.

```
create_playlist(session_id)
user_playlists = get_playlist(session_id)
new_playlist_id = user_playlists['items'][0]['id']

add_tracks_to_playlist(session_id, get_current_user_id(
    session_id), new_playlist_id, pl_songs)

new_pl_songs = []

songs_list = get_playlist_songs(session_id, new_playlist_id)
total_songs = songs_list['total']
for j in range(total_songs):
    new_pl_songs.append(
        {
            'name': songs_list['items'][j]['track']['name'],
            'artist': songs_list['items'][j]['track']['album']['artists'][0]['name'],
            'img': songs_list['items'][j]['track']['album']['images'][0]['url']
        }
    )

return Response(new_pl_songs, status=status.HTTP_200_OK)
```

Figure 32: Creating New Playlist with Recommended Tracks

It also should be considered if the user does not like the created playlist. Therefore there is a view to delete the latest created playlist of user's account like in figure 33.

```
def DeletePlaylistView(request):
    session_id = request.session.session_key

    delete_recommended_playlist(session_id)

    return redirect('frontend:my-playlists')
```

Figure 33: DeletePlaylistView

3.1.4 Content of Urls.py

Without defining the paths of the views in the urls.py file of the spotify app in our project, they are not reachable by the browser. So in order to handle this situation, all of the views must be associated with related named paths in the url patterns list as shown in the figure 34.

```python
urlpatterns = [
    path('get-auth-url', AuthURL.as_view()),
    path('redirect', spotify_callback),
    path('is-authenticated', IsAuthenticated.as_view()),
    path('playlists', GetUserPlaylists.as_view()),
    path('pl', GetPlaylist.as_view(), name='playlists'),
    path('songs', GetPlaylistSongs.as_view()),
    path('get-recommendation/<str:id>', MakePlaylistRecommendation.as_view()),
    path('delete-playlist', DeletePlaylistView)
]
```

Figure 34: Urls.py of Spotify App

3.2 Frontend

As mentioned in the previous sections, this project contains React.js components to managing the frontend part of the pages. Since frontend is also a django app it has also the build-in python files inside of itself but also consists some other folders for static files and component files also the configuration files to handle the implementation of these components as desired. In the figure 35 there is the demonstration of how the frontend app file structure looks alike.

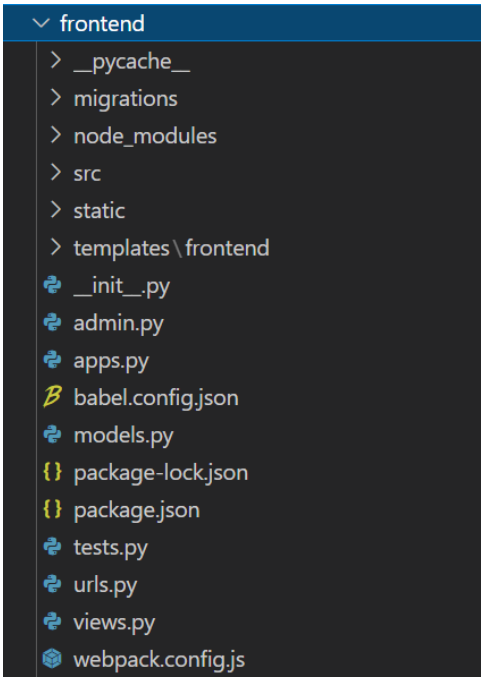

Figure 35: Content of Frontend App

Node-modules file contains all of the modules that has been using in the frontend implementation such as react, react-router-dom, babel and material ui, etc. src folder consists of components of the projects. In the static folder, there are static folders such as images and css files. In the templates file there is a folder named also frontend and inside this folder there is an index.html file that applies to whole pages of the project. In index.html file there is navigation bar and footer components of the project and also since semantic ui and some other frontend toolkits are being used in the project there are source to apply their designed tools. It also loads the static files and the main.js file properties which makes the whole frontend work. Additionally there are some json files to configure the versions of the packages. When the frontend react server is started running with the "npm run dev" command the application first changes the main.js according to the changes that was happened in the project so far. And changes to the main.js, since main.js file is being read by the index.html file and this file affects the whole project, it is really easy to apply those differences to the project and reflects them to the browser. As it is shown in the figure 36, all of the projects url paths are using the same view function called index.

```python
from django.shortcuts import render

# Create your views here.
def index(request, *args, **kwargs):
    return render(request, 'frontend/index.html')
```

Figure 36: Index View of Frontend

3.2.1 React Components

Components are independent and reusable bits of code that serve the same purpose as JavaScrpt functions but work in isolation an return HTML.[8] In React there are two options to define a component which are functional components and class components. And inevitably there are some differences between them such as syntax. Because a functional component is just a plain Js function which accepts props as an argument and returns a React element where a class component requires you to extend from React.[9]Component and creates a render function which returns a React element. In this project all of the components are class components.

```
export default class App extends Component{
    constructor(props){
        super(props);

    }
    render(){
        return(<Router>
          <Routes>
            <Route exact path='/' element={<HomePage />} />
            <Route path='/my-playlists' element={<ShowPlaylistPage />} />
            <Route path='/recommendation/:id' element={<ShowRecommendationPage />} />
            <Route path='/about-us' element={<AboutUsPage />} />
          </Routes>
          </Router>
      );
    }
}

const appDiv = document.getElementById("app");
render(<App />, appDiv);
```

Figure 37: App Component

For all components to work well with the project, React's routing was used. To route the components, their paths should be given to the Router as Route and all Routes must be placed in Routes element. All of the components in the Route have to define as element. Since the project first looks to the index.js file, in this file App.js router component was imported otherwise it will not work. In the project there are 4 main components that renders 4 different pages of the project as shown in figure 37 above.

In the HomePage component there are a simple typography and button which are imported from Material UI. For adding an action to the button for redirecting to the authentication page of Spotify a JavaScript function called authenticateSpotify() was created as in the figure 38. In this function, the is-authenticated page is fetching to get the status data and setting spotifyAuthenticated state which is defined in the constructor of the class component. After setting this state the fucntion checks whether it is false or true and it is false that means user should be redirecting to the Spotify authorization page. So to use this method the state is binded to the constructors property. To use this method with the button an OnClick property was added to the button and it calls to the authenticateSpotify function.

```
export default class HomePage extends Component {
  constructor(props) {
    super(props);
    this.state = {
      spotifyAuthenticated: false,
    };
    this.authenticateSpotify = this.authenticateSpotify.bind(this);
  }

  authenticateSpotify() {
    fetch("/spotify/is-authenticated")
      .then((response) => response.json())
      .then((data) => {
        this.setState({ spotifyAuthenticated: data.status });
        console.log(data.status);
        if (!data.status) {
          fetch("/spotify/get-auth-url")
            .then((response) => response.json())
            .then((data) => {
              window.location.replace(data.url);
            });
        }
      });
  }
```

Figure 38: authenticateSpotify Function and Constructor of HomePage Component

After all of the style designing for the HomePage, the final look of the page is the figure 39.

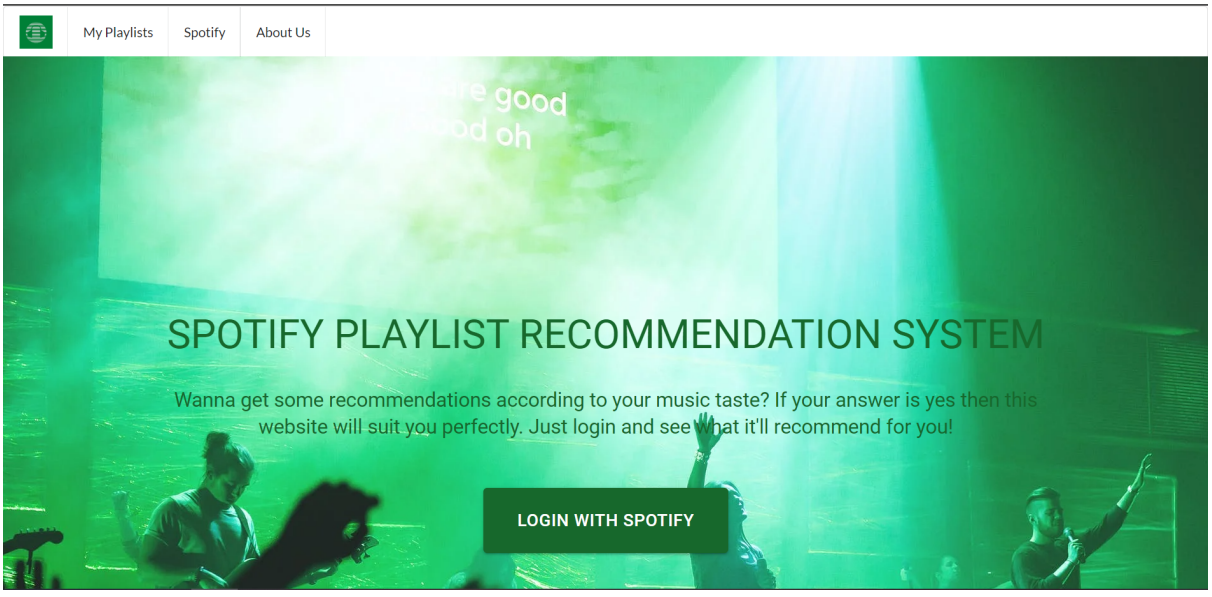

Figure 39: HomePage of the Project

The other component was written is ShowPlaylistPage which basically gets the GetPlaylist APIView's response data and uses them in nested map functions to render on the

web page. It should be noted that map() function is used for manipulating the array's item by iterating them.[11] Since it gives an oppurtunity to access every element in an array, it is used to rendering the songs and playlists.

```
{item.songs?.map((items) => {
  return (
    <Grid
      xs={8}
      justifyContent="center"
      container
      direction="row-reverse"
      style={{
        backgroundColor: "#FFF",
        padding: "30px",
        borderStyle: "solid",
        borderColor: "#88CF28",
        borderRadius: "10px",
      }}
    >
      <Grid
        item
        xs={3}
        align="center"
        style={{ paddingTop: "20px" }}
      >
        <h1>{items.name}</h1>

        <h3>{items.artist}</h3>
      </Grid>
```

Figure 40: Map Function to Iterate Songs in ShowPlaylistPage Component

It is also crucial point to define a parent element for each map function otherwise react will give an error. Also with json data it is not possible to use map function all the elements must be stored in an array and that's why in this component both playlists and songs states were defined as array. And the demonstration of the page on the browser is shown in the figure 41.

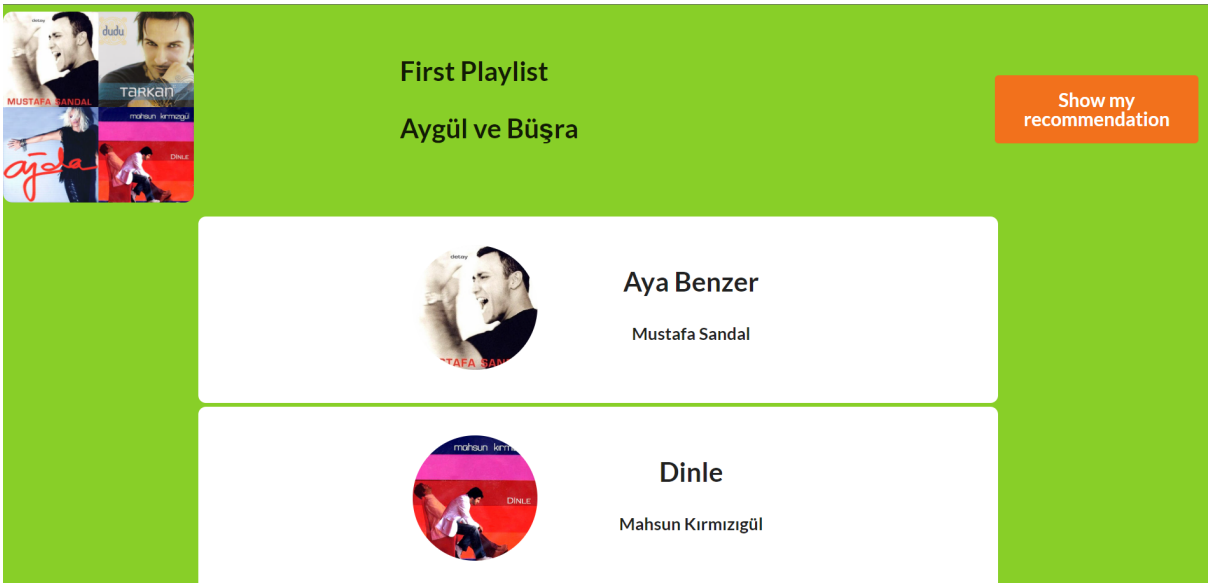

Figure 41: ShowPlaylistPage of the Project

The third component is ShowRecommendationPage component which is used for rendering a list of songs belong to the recommended playlist. So it is fetching to the url and get data from the url and stores them in the songs array with the function called getPlaylistSongs(). It is important to know that the id of the playlist is given by using the useParams hook. useParams() returns an object of key/value pairs from the current url that were matched to route path. But since useParams hook cannot be used with the class components like this one, it should be defined in the props as in the figure 42. And after that it can be used in the class. After getting the data the component is iterating them individually by using map function again. The page is looks like in the figure 43.

```
export default (props) => (
  <ShowRecommendationPage {...props} params={useParams()} />
);
```

Figure 42: useParams Definition

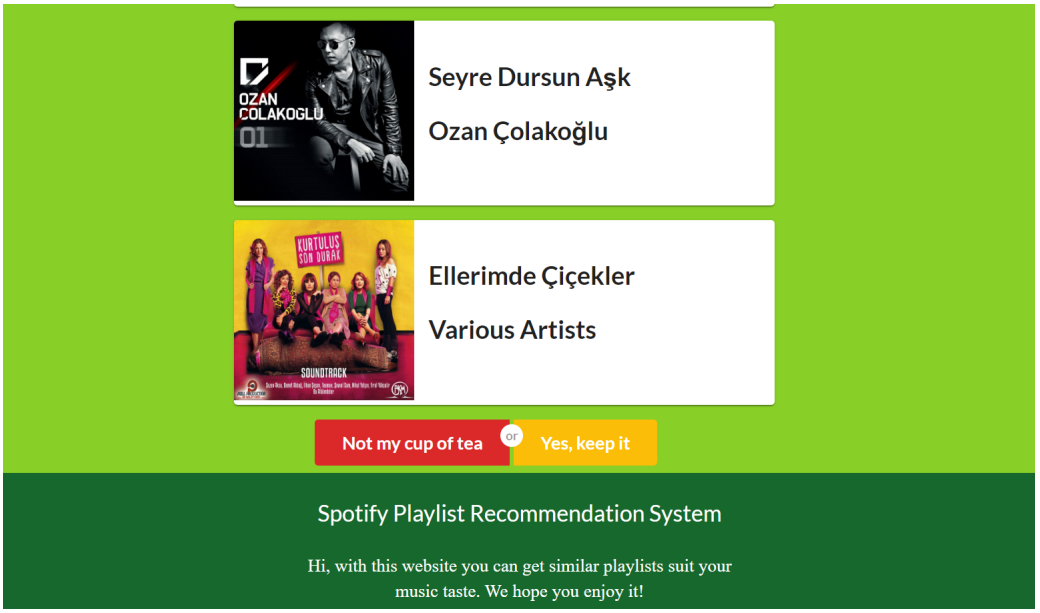

Figure 43: ShowRecommendationPage of the Project

```
constructor(props) {
  super(props);

  this.state = {
    songs: [],
    id: this.props.params.id,
  };
  console.log("deneme");
  this.getPlaylistSongs = this.getPlaylistSongs.bind(this);
  this.getPlaylistSongs();
}

getPlaylistSongs() {
  fetch("/spotify/get-recommendation/" + this.state.id)
```

Figure 44: id State with Using Props

The last component is AboutUsPage, since it is not using any functions but just rendering some Material UI components, it is just used for creating two cards that shows some non-detailed information about us, the developers of this project. The cards contain the contact links that are connected with the IconButton which is also a Material UI component. For giving a href link to the button HTML <a> tag is not enough for this case. Therefore as in the figure 46, the links are defined in the window.open() function and by changing the target value it is possible to manipulate which tab and window is gonna be opened with the link. So the final look of the about us page is shown in the figure 45.

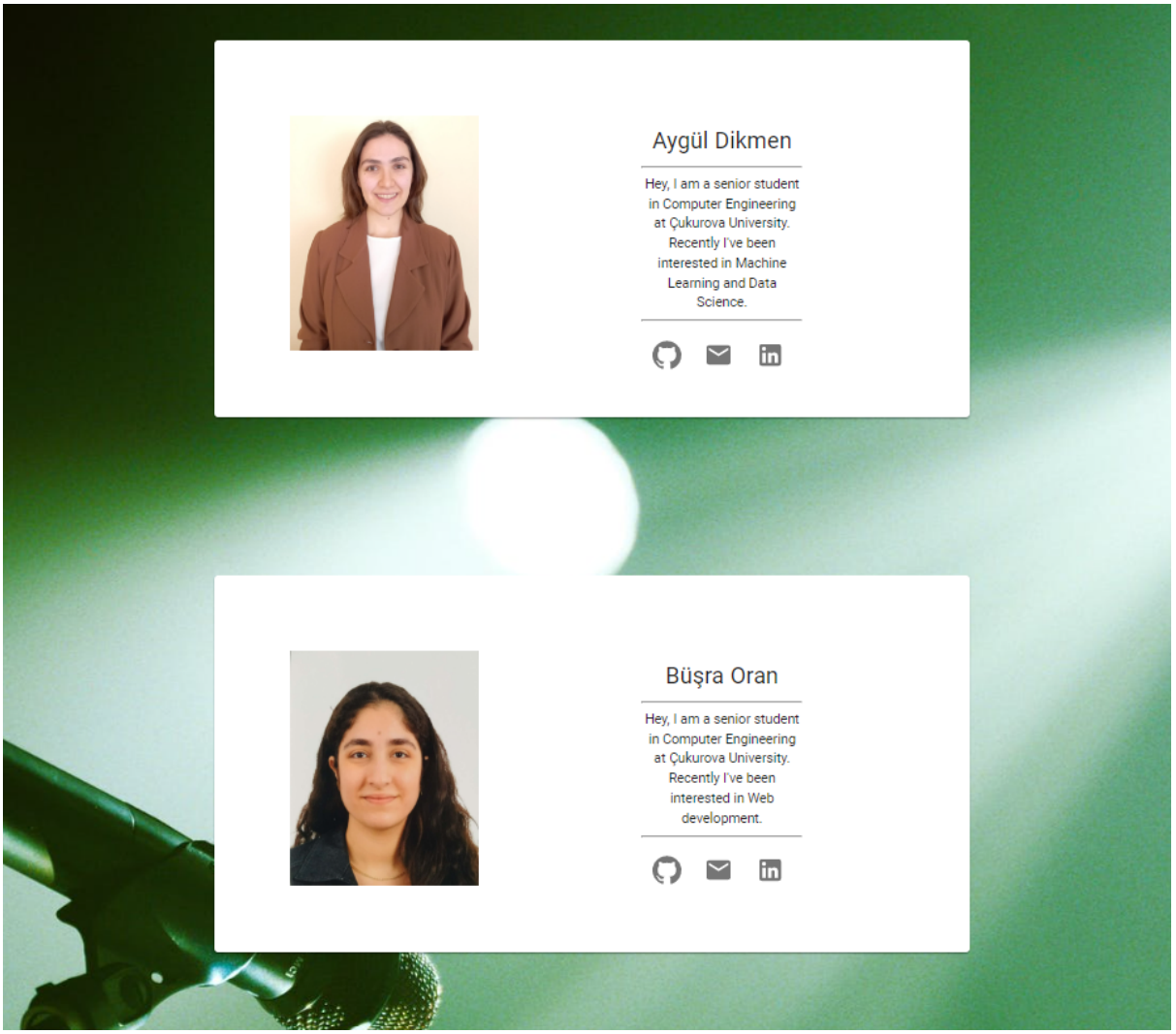

Figure 45: AboutUs Page of the Project

```
<IconButton
  onClick={() =>
    window.open("https://github.com/busraaaoran", "_blank")
  }
>
  <GitHub fontSize="large" />
</IconButton>
```

Figure 46: window.open() Function

As it can be seen, the target values defined as "_blank" which means the link will be opened on another tab. If it was "_self" intead of "_blank" there will not be opened a new window and the link will be opened in the same tab of the current user is in.

3.3 Machine Learning

3.3.1 Extracting Song Data From the Spotify API Using Python

After making the user authentication, data extracting process is applied and making it a pandas dataframe by 'extract_data' function. This function has two parameters: a data frame to add the songs that are extracted, and the URI of the playlist.

```
def extract_data(playlistDF, playlist_URI):
    for track in sp.playlist_tracks(playlist_URI)["items"]:
        try:
            track_id = track["track"]["id"]
            track_name = track["track"]["name"]

            track_pop = track["track"]["popularity"]

            #Main Artist
            artist_uri = track["track"]["artists"][0]["uri"]
            artist_info = sp.artist(artist_uri)
            artist_name = track["track"]["artists"][0]["name"]
            artist_pop = artist_info["popularity"]
            genres = artist_info['genres']

            for feature in sp.audio_features(track["track"]["id"]):
                danceability = feature["danceability"]
                energy = feature["energy"]
                key = feature["key"]
                loudness = feature["loudness"]
                mode = feature["mode"]
                speechiness  = feature["speechiness"]
                acousticness = feature["acousticness"]
                instrumentalness = feature["instrumentalness"]
                liveness = feature["liveness"]
                valence = feature["valence"]  # A measure from 0.0 to 1.0 describing the musical positiveness conveyed
                tempo = feature["tempo"]

            ser = pd.Series([track_id, track_name, track_pop ,artist_name, artist_pop, genres, danceability, energy, ke
            playlistDF = pd.concat([playlistDF, ser],axis = 1, ignore_index = True)


        except:
            continue
```

Figure 47: 'extract_data' Function

In this function, with a loop every item in the playlist is taken and made into a series, and finally added to the data frame. There is a try - except block to prevent the error caused by possible errors. And finally, we determine the data types.

```
playlistDF = playlistDF.transpose()
playlistDF.columns = ['track_id', 'track_name', 'track_pop', 'artist_name', 'artist_pop', 'genres', 'danceability',

playlistDF = playlistDF.astype(dtype = {'track_id' : str,
                                        'track_name' : str,
                                        'track_pop' : int,
                                        'artist_name' : str,
                                        'artist_pop': float,
                                        'genres' : object,
                                        'danceability' : float,
                                        'energy' : float,
                                        'key' : int,
                                        'loudness' : float,
                                        'mode' : float,
                                        'speechiness' : float,
                                        'acousticness' : float,
                                        'instrumentalness' : float,
                                        'liveness' : float,
                                        'valence' : float,
                                        'tempo' : float })


return playlistDF
```

Figure 48: 'extract_data' Function Continued

While extracting the data two functions from spotipy is used: sp.playlist_tracks() which returns all the information belonging to a track, and sp.audio_features which returns the features of the track.

The next function in data preprocessing is drop_duplicates. Via this function it is possible to delete the repetitive songs.

```
# Drop song duplicates
def drop_duplicates(df):
    '''
    Drop duplicate songs
    '''
    df['artists_song'] = df.apply(lambda row: row['artist_name']+row['track_name'],axis = 1)
    return df.drop_duplicates('artists_song')
print(all_playlistDF['genres'].dtype)
songDF = drop_duplicates(all_playlistDF)
playlistDF = drop_duplicates(user_playlistDF)
```

Figure 49: 'drop_duplicates' function

As it is mentioned before, in this project from Recommendation System Techniques, Content Based Process is used. The data has been taken in three ways: **metadata, audio data, and text data** .

Metadata

Metadata refers to the attributes related to the song but not the song itself (e.g., popularity and genres). In this project, I treated the metadata in two ways.
For genre data, one-hot encoding is used. This is done by converting each category into a column so that each category can be represented as either True or False.

The genres in Spotify are not balanced distributed with some genres more prevalent while others are more obscure. In addition, one artist or track could be associated with multiple genres. Hence, we need to weigh the importance of each genre to combat overweighting specific genres while underestimating others. Therefore, TF-IDF measures are introduced and applied to the genre data.

The motivation is to find words that are not only important in each document but also account for the entire corpus. The log value was taken to decrease the impact of a large N, which would lead to a very large IDF compared to TF. TF is focused on the importance of a word in a document, while IDF is focused on the importance of a word across documents.

In this project, the documents are analogous to songs. The most prominent genre in each song and their prevalence across songs is calculated to determine the weight of the genre.

To implement this, the TfidfVectorizer() function from scikit learn has been used.

```
def ohe_prep(df, column, new_name):
    '''
    Create One Hot Encoded features of a specific column
    ---
    Input:
    df (pandas dataframe): Spotify Dataframe
    column (str): Column to be processed
    new_name (str): new column name to be used

    Output:
    tf_df: One-hot encoded features
    '''

    tf_df = pd.get_dummies(df[column])
    feature_names = tf_df.columns
    tf_df.columns = [new_name + "|" + str(i) for i in feature_names]
    tf_df.reset_index(drop = True, inplace = True)
    return tf_df
```

Figure 50: 'ohe_prep' Function

For popularity data, It is considered as a continuous variable and only normalized it into a range between 0 and 1. The idea is songs that are popular are likely to be heard by people who like popular songs, while songs that are less popular are likely to be heard by people who have the same taste.

This is done via the MinMaxScaler() function in scikit learn:

```
# Scale audio columns
floats = df[float_cols].reset_index(drop = True)
scaler = MinMaxScaler()
floats_scaled = pd.DataFrame(scaler.fit_transform(floats), columns = floats.columns) * 0.2
```

Figure 51: Scale Column

Audio Data

Audio data refers to the audio features of the song extracted using the Spotify API. For example, the loudness, tempo, danceability, energy, speechiness, acousticness, instrumentalness, liveness, valence, and duration. In this project, the only manipulation that is conducted on these data was normalization based on the maximum and minimum values of each variable. In addition, one-hot encoding was conducted on several other audio features, such as the key of the track. The constraint of this method is similar to that of one-hot encoding — it is not known if people view equally among different keys. By assuming every key is equally weighted, it is less likely to obtain the best representation of the data mathematically. Therefore, possible hyperparameter tuning could be required to improve the prediction.

```
# One-hot Encoding
subject_ohe = ohe_prep(df, 'subjectivity','subject') * 0.3
polar_ohe = ohe_prep(df, 'polarity','polar') * 0.5
key_ohe = ohe_prep(df, 'key','key') * 0.5
mode_ohe = ohe_prep(df, 'mode','mode') * 0.5

# Normalization
# Scale popularity columns
pop = df[["artist_pop","track_pop"]].reset_index(drop = True)
scaler = MinMaxScaler()
pop_scaled = pd.DataFrame(scaler.fit_transform(pop), columns = pop.columns) * 0.2

# Scale audio columns
floats = df[float_cols].reset_index(drop = True)
scaler = MinMaxScaler()
floats_scaled = pd.DataFrame(scaler.fit_transform(floats), columns = floats.columns) * 0.2
```

Figure 52: Audio Data Extracting

Text

The only text feature that has been used in this project is track name. I conducted sentiment analysis finding the polarity and subjectivity of the track name.

Subjectivity (0,1): The amount of personal opinion and factual information contained in the text.

Polarity (-1,1): The degree of strong or clearly defined sentiment accounting for negation.

The goal of the sentiment analysis is to extract additional features from the tracks. By doing so, sentiment data and other audio features via textual information can be extracted. To conduct sentiment analysis, TextBlob.sentiment was used.

```
def getSubjectivity(text):
  '''
  Getting the Subjectivity using TextBlob
  '''
  return TextBlob(text).sentiment.subjectivity
```

Figure 53: 'getSubjectivity' Function

```
def getPolarity(text):
  '''
  Getting the Polarity using TextBlob
  '''
  return TextBlob(text).sentiment.polarity
```

Figure 54: 'getPolarity' Function

```
def getAnalysis(score, task="polarity"):
  '''
  Categorizing the Polarity & Subjectivity score
  '''
  if task == "subjectivity":
    if score < 1/3:
      return "low"
    elif score > 1/3:
      return "high"
    else:
      return "medium"
  else:
    if score < 0:
      return 'Negative'
    elif score == 0:
      return 'Neutral'
    else:
      return 'Positive'
```

Figure 55: 'getAnalysis' Function

```
def sentiment_analysis(df, text_col):
  '''
  Perform sentiment analysis on text
  ---
  Input:
  df (pandas dataframe): Dataframe of interest
  text_col (str): column of interest
  '''
  df['subjectivity'] = df[text_col].apply(getSubjectivity).apply(lambda x: getAnalysis(x,"subjectivity"))
  df['polarity'] = df[text_col].apply(getPolarity).apply(getAnalysis)
  return df
```

Figure 56: 'sentiment_analysis' Function

Now, every feature engineering method are combined into one function and output the data into a large feature dataframe:

```
def create_feature_set(df, float_cols):

    # Tfidf genre lists
    tfidf = TfidfVectorizer()
    tfidf_matrix =  tfidf.fit_transform(df['genres'].apply(lambda x: " ".join(x)))
    genre_df = pd.DataFrame(tfidf_matrix.toarray())
    genre_df.columns = ['genre' + "|" + i for i in tfidf.get_feature_names()]
    #genre_df.drop(columns='genre|unknown') # drop unknown genre
    genre_df.reset_index(drop = True, inplace=True)

    # Sentiment analysis
    df = sentiment_analysis(df, "track_name")

    # One-hot Encoding
    subject_ohe = ohe_prep(df, 'subjectivity','subject') * 0.3
    polar_ohe = ohe_prep(df, 'polarity','polar') * 0.5
    key_ohe = ohe_prep(df, 'key','key') * 0.5
    mode_ohe = ohe_prep(df, 'mode','mode') * 0.5
```

Figure 57: 'create_future_set' Function

```
    # Normalization
    # Scale popularity columns
    pop = df[["artist_pop","track_pop"]].reset_index(drop = True)
    scaler = MinMaxScaler()
    pop_scaled = pd.DataFrame(scaler.fit_transform(pop), columns = pop.columns) * 0.2

    # Scale audio columns
    floats = df[float_cols].reset_index(drop = True)
    scaler = MinMaxScaler()
    floats_scaled = pd.DataFrame(scaler.fit_transform(floats), columns = floats.columns) * 0.2

    # Concanenate all features
    final = pd.concat([genre_df, floats_scaled, pop_scaled, subject_ohe, polar_ohe, key_ohe, mode_ohe], axis = 1)

    # Add song id
    final['track_id']=df['track_id'].values

    return final
```

Figure 58: 'create_future_set' Function Continued

Process

Two steps were implemented in the algorithm. These two steps are needed every time some enters a new playlist query:

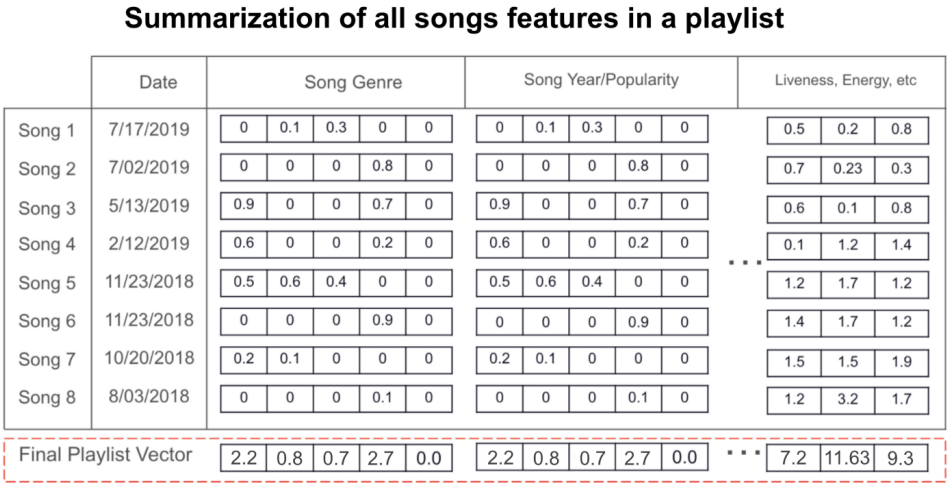

Figure 59: Summarization process illustration. [12]

In this step, all songs in a playlist are summarized into one vector that can be compared to all other songs in the dataset to find their similarities. In other words, this vector describes the whole playlist as if it is one song.

```
def generate_playlist_feature(complete_feature_set, playlist_df):

    # Find song features in the playlist
    complete_feature_set_playlist = complete_feature_set[complete_feature_set['track_id'].isin(playlist_df['track_id'].
    # Find all non-playlist song features
    complete_feature_set_nonplaylist = complete_feature_set[~complete_feature_set['track_id'].isin(playlist_df['track_i
    complete_feature_set_playlist_final = complete_feature_set_playlist.drop(columns = "track_id")
    return complete_feature_set_playlist_final.sum(axis = 0), complete_feature_set_nonplaylist
```

Figure 60: 'generate_playlist_feature' Function

Similarity and Recommendation

In this project, the cosine_similarity() function from scikit learn is used to measure the similarity between each song and the summarized playlist vector.

```
def generate_playlist_recos(df, features, nonplaylist_features):

    non_playlist_df = df[df['track_id'].isin(nonplaylist_features['track_id'].values)]
    # Find cosine similarity between the playlist and the complete song set
    non_playlist_df['sim'] = cosine_similarity(nonplaylist_features.drop('track_id', axis = 1).values, features.values.
    non_playlist_df_top_20 = non_playlist_df.sort_values('sim',ascending = False, ignore_index = True).head(20)

    return non_playlist_df_top_40
```

Figure 61: 'generate_playlist_recos' Function

And finally the using of these functions:

Extracting Data

```
all_playlist_list = ["https://open.spotify.com/playlist/37i9dQZEVXbNG2KDcFcKOF?si=253f12a3e38c4aa1","https://open.spoti
                     "https://open.spotify.com/playlist/37i9dQZF1DX2iRL6tJD46O?si=e1634761eed1487c", "spotify:playlist:
                     "spotify:playlist:37i9dQZF1DX28yPuNTio0f", "spotify:playlist:37i9dQZF1DWSDCcNkUu5tr","spotify:play

# Extracting all data
all_playlistDF = pd.DataFrame()

#for playlist in range(len(all_playlist_list)):
for playlist in range(len(all_playlist_list)):
    temp_playlist = pd.DataFrame()
    playlist_link = all_playlist_list[playlist]
    playlist_URI = playlist_link.split("/")[-1].split("?")[0]
    #track_uris = [x["track"]["uri"] for x in sp.playlist_tracks(playlist_URI)["items"]]

    temp_playlist = extract_data(temp_playlist, playlist_URI)

    all_playlistDF = all_playlistDF.append(temp_playlist)


#Extracting Tracks From a User
playlist_link = "https://open.spotify.com/playlist/0lAwdSzEfiWhSXplPT227g"
playlist_URI = playlist_link.split("/")[-1].split("?")[0]
#track_uris = [x["track"]["uri"] for x in sp.playlist_tracks(playlist_URI)["items"]]

user_playlistDF = pd.DataFrame()

user_playlistDF = extract_data(user_playlistDF, playlist_URI)
```

Figure 62: Usage of Functions

Recommendation Process

```
sentiment = sentiment_analysis(songDF, "track_name")

# One-hot encoding for the subjectivity
subject_ohe = ohe_prep(sentiment, 'subjectivity','subject')

# Save the data and generate the features
float_cols = songDF.dtypes[songDF.dtypes == 'float64'].index.values

floats = songDF[float_cols].reset_index(drop = True)
scaler = MinMaxScaler()
floats_scaled = pd.DataFrame(scaler.fit_transform(floats), columns = floats.columns) * 0.2

# Test playlist:  User's Playlist
playlistDF_test = playlistDF.copy()
playlistDF_test.to_csv("/Users/ayguldikmen/GraduationProject/data/test_playlist.csv")

# Save the data and generate the features
float_cols = songDF.dtypes[songDF.dtypes == 'float64'].index.values

# Generate features
complete_feature_set = create_feature_set(songDF, float_cols=float_cols)

# Generate the features
complete_feature_set_playlist_vector, complete_feature_set_nonplaylist = generate_playlist_feature(complete_feature_set

# Genreate top 10 recommendation
recommend = generate_playlist_recos(songDF, complete_feature_set_playlist_vector, complete_feature_set_nonplaylist)
recommend.head()
```

Figure 63: Recommendation Step

4 Conclusion

Since music is an international thing that has an effect on almost every area in the world, people need to listen to music and feel the vibe that the music provides to them. In this case the usage of the music platforms is rising day by day. In order to be one step ahead from the rest of the companies, Spotify has been providing some recommendations for its users. Hence in this project, the aim is taking the Spotify user's playlists and showing them to choose which playlist is gonna be used for using Content-Based Recommendation System. After this machine learning section, the project's interface provides a brand new playlist. The project's backend is developed by Django and Django REST-Framework , its frontend is developed by React.js and for the database connection the most popular NoSQL database, MongoDB, is used. In brief, this Website helps users to find similar playlists that suit the referenced playlists and make sures if the user likes it or not to keep the playlist in the user's own Spotify account and it can be improved inevitably by time.

References

1. Authorization Code Flow Web Page https://developer.spotify.com/documentation/general/guides/authorization/code-flow/ , Last Visited: 10 May, 2022
2. Django Introduction, https://developer.mozilla.org/en-US/docs/Learn/Server-side/Django/Introduction , Last Visited: 10 May, 2022
3. FIELDING R, IRVINE UC, Hypertext Transfer Protocol -- HTTP/1.1, June 1999, https://www.ietf.org/rfc/rfc2616.txt
4. https://developer.spotify.com/documentation/web-api/
5. Vikas Malik and Amit Kumar. “Sentiment Analysis of Twitter Data Using Naive Bayes Algorithm”, International Journal on Recent and Innovation Trends in Computing and Communication, Vol. 6, No. 4, 2018.
6. A Comprehensive Study on Lexicon Based Approaches for Sentiment Analysis, ISSN: 2249-0701 Vol.8 No.S2, 2019, pp. 1-6
7. Sentiment analysis using TF–IDF weighting of UK MPs’ tweets on Brexit
8. React Components, https://www.w3schools.com/react/react_components.asp#:~:text=Components%20are%20independent%20and%20reusable,will%20concentrate%20on%20Function%20components. , Last Visited: 10 May 2022
9. Functional vs Class-Components in React https://djoech.medium.com/functional-vs-class-components-in-react-231e3fbd7108, Last Visited: 10 May, 2022
10. Building a Song Recommendation System with Spotify, https://towardsdatascience.com/part-iii-building-a-song-recommendation-system-with-spotify-cf76b52705e7, Last Visited: 10 May, 2022
11. How to Use the Map() Function to Export JavaScript in React https://www.pluralsight.com/guides/how-to-use-the-map()-function-to-export-javascript-in-react, 10 May 2022
12. spotify-recommendation-engine.ipynb https://github.com/madhavthaker/spotify-recommendation-system/blob/main/spotify-recommendation-engine.ipynb
13. Lalita Sharma , Anju Gera. "A Survey of Recommendation System: Research Challenges ". International Journal of Engineering Trends and Technology (IJETT). V4(5):1989-1992 May 2013. ISSN:2231-5381. www.ijettjournal.org. published by seventh sense research group.

14. Welcome to Spotipy! https://spotipy.readthedocs.io/en/2.19.0/, Last Visited: 10 May 2022
15. SINGH K.S., MAHURYA N.S., MANI A., YADAV R.S., “Machine learning method using position-specific mutation based classification outperforms one hot coding for disease severity prediction in haemophilia ‘A”, 23 June 202
16. What is Cosine Similarity? - https://deepai.org/machine-learning-glossary-and-terms /cosine-similarity, Last Visited: 10 May, 2022
17. Authorization https://developer.spotify.com/documentation/general/guides/authorization/ , Last Visited: 10 May, 2022
18. SHARMA L., GERA A., “A Survey of Recommendation System: Research Challenges”, Published 2013
19. What is Cosine Similarity? https://deepai.org/machine-learning-glossary -and-terms/cosine-similarity, Last Visited: 10 May, 2022

Curriculum Vitae

Büşra Oran was born in Istanbul, 1999. In 2013, she started to her high school, Zeytinburnu Anatolian High School, as the 1st student. After her high school life finished she decided to go to college outside her hometown. In 2017 she enrolled to the Çukurova University as a preparation student since her department offers 100% English lessons. After finishing the preparation class for 1 year, she started studying Computer Engineering as her subject. When she was a junior student, she participated in the Engineer Girls of Turkey project. With this project she started to follow soft skills certification programmes. With the influence of her mentors and coach she started to give online English speaking lessons voluntarily. The summer holiday follows her junior year, she has done her 2 internships at different two companies, Doğuş Technology and Novumare Technologies. In the first semester of senior year she went to Poland to study as an Erasmus+ student. Her hobbies are watercolor drawing, reading worlds' classics, and watching anime. In April 2022, she participated in an Ideathon competition produced by Wtech(Woman in Technology) and her team's project became 1st place. She has currently been interested in web development. Now she is a 4th grade Computer Engineering student at Çukurova University who is just one step behind graduating.

Aygül Dikmen was born in 1999, Adana, Turkey. She graduated from Borsa Anatolian High School. After she enrolled at Cukurova University's Computer Engineering department. In the first year of her University education she went to English preparation class for an education year. In second grade and fourth grade she had the right to go to University

Politehnica of Bucharest via Erasmus+ Education program. She made two internships, in one of them she made a SQL based hospital automation system, and the other one as a research intern. She has certificates from Agile Thinking, Introduction to Cybersecurity. Currently she is a 4th grade student who is aimed to improve herself in the Machine Learning, Data Science area.