Music Rating Website

COMP 4905 – Computer Science Honours Project

Final Report

Carleton University

Author: Khaled Banjaki

Student ID: 101058264

Supervisor: Lou Nel

April 25th, 2022

Abstract

In traditional music applications or websites, the number of music is so large that it is difficult for the average user to find the best music that will suit their current mood and needs. It may not be the case for people who have a relatively small catalog of music they listen to. However, since music is so large and the raw number of categories that exist today is steadily increasing, it would be more efficient to have a personalized list rather than manually go through such a vast array of music. It can prove even more difficult with a poorly designed music website lacking attractiveness. As a solution, I have designed and implemented a recommendation system for our music rating website, which will help increase the beauty of the website and predict the appropriate songs for the user based on key attributes and qualities of music that has been recently played.

Acknowledgments

I want to thank my friends and family for supporting me throughout the development of this project. I would also like to thank Professor Louis Nel for the opportunity to work under him and for the advice and guidance that was provided.

Table of Contents

List of Figures

List of Tables

Main Body

Overview

The digital age has made it easier to access music thanks to the tools and technologies humans have created today. It can be heard on your mobile device, computer, or even at your favorite restaurant. Arguably being the best form of media entertainment today, it is thanks to the creativity of humans that have expressed their ideas and emotions in the form of sounds that consist of melody, harmony, and rhythm. Millions of songs exist compared to our previous era, which will continue to grow throughout the future, and sorting out all this music is mentally exhausting and time-consuming. Moreover, it is crucial for music service providers to efficiently help their users in discovering, organizing, and managing their music by providing high-caliber recommendations. With popular music streaming platforms like Spotify and SoundCloud, users can receive recommendations based on their history of interaction with music. This website will have a similar goal in mind, except it will focus solely on recommending music to a user by providing actions a user will perform to help establish what music and genres a user prefers.

The recommender system is the core of our application and where the bulk of the work rests. It is defined as an input machine that feeds off data gathered from users. The system aggregates this data and allocates it to the appropriate user in a personalized way to further expand the possibility of their options. A recommendation system plays a vital role in upholding the streaming music business, by reducing the frustration and amount of time it takes a user to find pleasing content. Our recommender system will help improve the efficiency and accuracy of music being played by forecasting the data specified by the user. Top-played tracks on a user's

Spotify account will help generate these suggestions and create a list of appropriate songs the user would like. The Spotify API will be an important factor in the development of our recommender system. Selective authorization granted by the user will first have to be granted before our recommender system can begin processing data. Using the Spotify API and granted access we will fetch user-related data such as recently played songs, playlists, and more. Users will no longer need to go through an extensive list of music to find their musical tastes. Once our application has access to the user's Spotify account, we will display their favorite songs on our website. For the complete picture, we will use additional data such as the "Like" and "Add to playlist" ratings which will be given by the users. We can rate "Like" on the website for the user's favorite songs to help mark the tracks a user did not like. Typically, people appreciate music that has generated a strong emotional response in them, whether positive or negative. Therefore, the assessments accurately reflect the addiction of a person. But assessments alone are not enough: firstly, people do not always put them, and secondly, the scale lacks semitones - there is only either "good" ("Like") or "bad" ("Don't like"). Therefore, in addition to ratings and auditions, we pay attention to another possible action: Adding tracks to the playlist.

We group together all these actions to compose a popularity score. Since the "Like" and "Add to playlist" sentiments are not enough to determine whether a song should be suggested to a user or not. Hence, I attempt to engineer a public sentiment feature called popularity score. This popularity score will determine what songs a user should have recommended to them, it will help filter out the songs that are popular on the website and have a good public sentiment.

All actions are of course unequal, the “Like” rating and “Adding to a playlist” have different weights associated with them. The popularity score is just as important and will be used to determine what recommendations a user will receive.

Goals

The goal of this project is to provide musical recommendations to a user based on their history of interactions on the Spotify application. This project will consist of both web development and artificial intelligence, the latter being the most challenging. For our website, I plan to design and implement a collaborative filtering approach – one of many types of recommendation systems that exist today and work primarily in the media and entertainment industry. The learning model that will help fulfill this goal is the nearest neighbor unsupervised model. After much research, I have decided this model is the most feasible to help achieve the goal for our music rating website. Using the open-source python machine learning library scikit-learn which provides functionality for unsupervised neighbors-based learning methods. The algorithm will be trained using the Million Song Dataset as a parameter, to help fine-tune it.

For the web development part of our application, our goal is to take advantage of Node and its powerful scalability for our back end alongside MongoDB and Redis for our database and fast caching. For the front-end and user interface development, we will use React.js in conjunction with HTML and CSS to add user ratings, single tracks, playlists, and preference features. I also plan to integrate the Spotify SDK into the back end of the project to help fetch the user’s metadata from Spotify.

With all this in mind, it is expected that a user should be able to generate recommendations based on certain actions performed on single tracks available on the website:

Liking a song, disliking a song, or adding a song to a playlist. The public sentiment feature will be included in our algorithm by gathering data from across our website with high popularity scores.

Objectives

The principal way of achieving the goals is to design the machine learning model for our recommender system and use the right tools for both the front end and back end of our project for maximum efficiency. Below is a list of steps that were taken to complete the project in its entirety including the tools and technologies used.

1. Researched the following AI algorithms and chose the best recommender algorithm for our website out of the following three:
    a. Collaborative filtering that analyzes the user's behavior.
    b. Natural language processing for text analysis.
    c. Audio models that analyze audio files.
2. Developed the front end of the project and created a login page using HTML, CSS, and React.js
3. Developed the back end of the project with Node, Express.js, MongoDB & Redis.
4. Developed the user interface by adding user ratings, dislikes, single tracks, and playlist preference features after integrating the Spotify SDK into the back end of the project.
5. Designed and developed the collaborative filtering approach using the unsupervised nearest neighbor machine learning model with the Million Song Dataset as a parameter to train our model.

6. Tested our application using real data from several users with Spotify accounts.

Python will be used as a language for implementing our recommender algorithm & JavaScript for everything else. Restful APIs will be used for communicating data throughout the application.

Front-end Development

React.js

To create the front-end of my project, I made use of the popular JavaScript library React.js and used it to generate UI components. I was focused on using Angular for the front-end but React seemed more feasible and attractive for my goal which is why I opted for it. I've had experience using React and it is very flexible, not as complex as Angular, and written in JavaScript. React is basically a layer on top of JavaScript, which builds on top of and relies on other tools like HTML, CSS, and JS for its full performance. React helped play an important role in how the user interface of my website was structured. However, React was not enough on its own to fully develop the website. A webpage consists of many important qualities such as the layout, the content of the page, the design or style of the page, and the logic used for retrieving data is a combination of HTML, CSS, and JS. HTML/CSS is used to display the contents of a web page; however, the results are largely static on their own.

React enabled me to create multiple components for my application, with each of the components having its own logic and controls. React works by creating/modifying and deleting elements that are in the DOM (Document Object Model). It takes advantage of the virtual DOM to calculate potential changes, then implements these changes on the screen via traditional DOM manipulation. It was responsible for handling all the manipulation of our DOM and managing the HTML from under the hood. I was able to maintain the UI of our website using the virtual DOM thanks to its ease of modularization and performance.

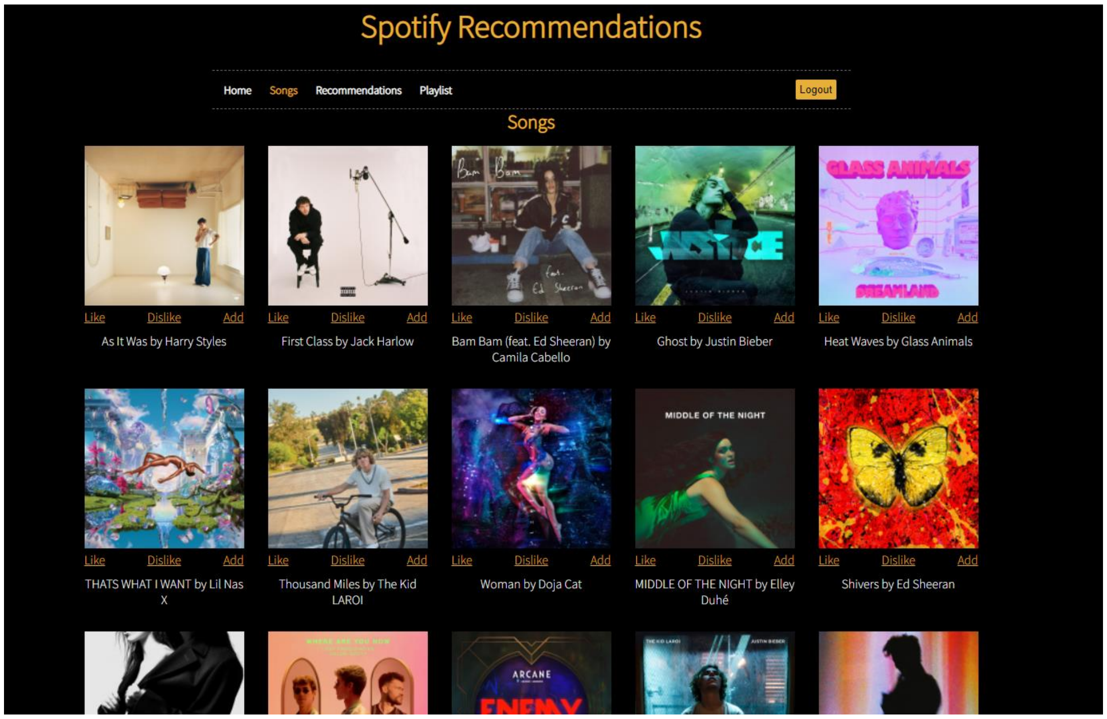

Spotify Recommendations
Home
Songs
Recommendations
Playlist
Logout
Songs
Like Dislike Add
As It Was by Harry Styles
First Class by Jack Harlow
Bam Bam (feat. Ed Sheeran) by Camila Cabello
Ghost by Justin Bieber
Heat Waves by Glass Animals
THATS WHAT I WANT by Lil Nas X
Thousand Miles by The Kid LAROI
Woman by Doja Cat
MIDDLE OF THE NIGHT by Elley Duhé
Shivers by Ed Sheeran

FIGURE - 1. UI for songs tab

To make it even easier to implement my React application, I made great use of JSX (JavaScript XLM) which allowed me to write HTML elements in JavaScript without the need to use methods like createElement() which creates the HTML element specified by the given tagName. It is an XML-like syntax extension used for JavaScript to make it easier and more intuitive to design our user interface.

For managing the state of my application, I made use of the Redux JavaScript library. In simple words, it helps React in building our user interface and is the official React binding for Redux. I was able to have the React components read data from the redux store I set up to dispatch actions to our store and update data when needed. Redux is simple to use and helped scale my React app by providing a sensible way to take care of my application state through a unidirectional data flow model.

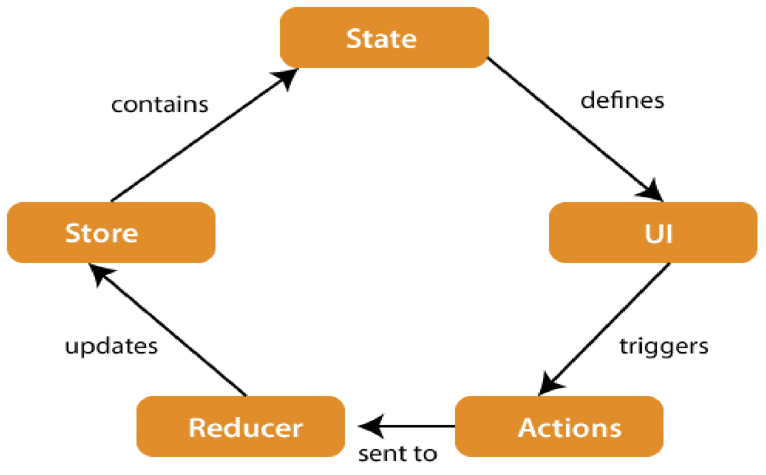
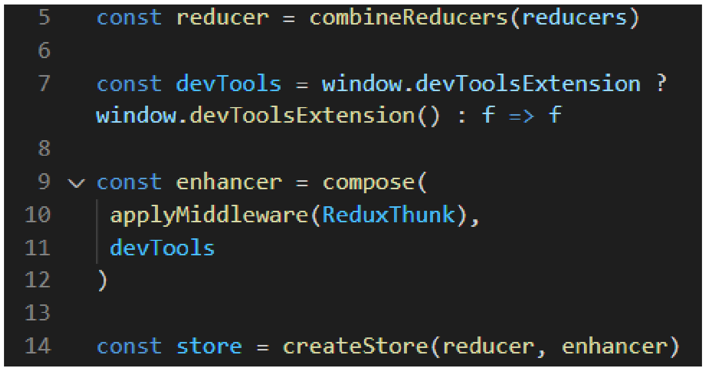


FIGURE - 2. Redux Architecture

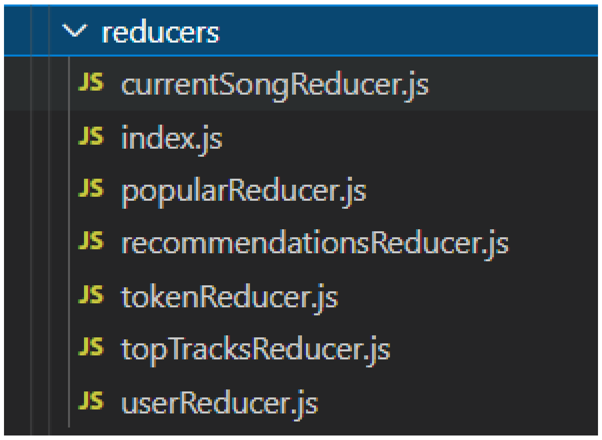
We start by subscribing to the redux store, and from there it does frequent checks to make sure the data from components of the website wants to be changed and re-renders these components. The redux store in place will hold that state tree, and we specify the different parts of that state tree so they can respond to specific actions. Below we combine several of our reducers into a single reducer function by calling combineReducers(), and this function returns a reducer function that will basically invoke every one of our reducers inside of the object to build our state object with the exact shape.



```
const reducer = combineReducers(reducers)

const devTools = window.devToolsExtension ?
window.devToolsExtension() : f => f

const enhancer = compose(
  applyMiddleware(ReduxThunk),
  devTools
)

const store = createStore(reducer, enhancer)
```

FIGURE - 3. How our store is created

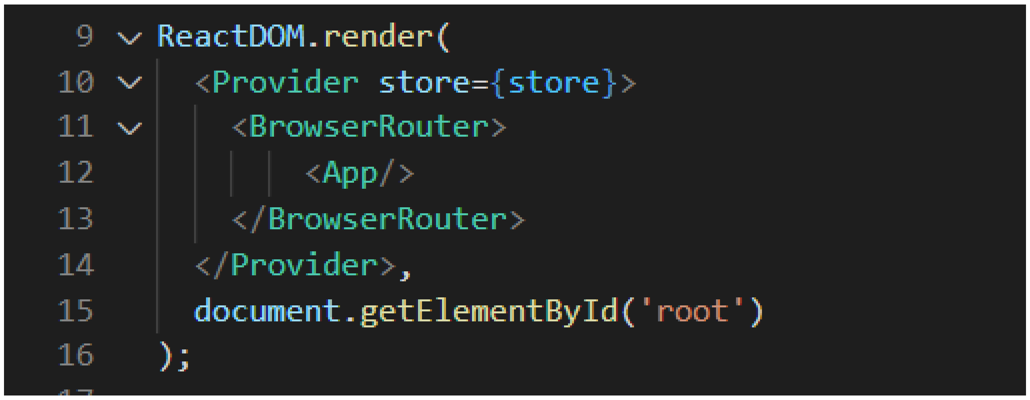
We pass this function to the createStore() with two parameters; The reducer function that was returned from calling combinedReducers() and a store enhancer which applies middleware. Rexud Thunk is the middleware that is applied, and lets you call action creators that will return a function instead of the action object. It will receive our store's dispatch method and will be used to dispatch regular synchronous actions inside our function's body once the asynchronous operations have been completed. There are several reducers I have implemented that are responsible for storing user details, fetching top tracks, storing tokens, fetching recommendations, and storing current played songs with its index.



```
reducers
  currentSongReducer.js
  index.js
  popularReducer.js
  recommendationsReducer.js
  tokenReducer.js
  topTracksReducer.js
  userReducer.js
```

FIGURE - 4. Current reducers being used

Once we have everything in place, we must create a file to handle our app start-up, routing, and other functionality our application requires, placed in our index.js file in the root directory of our project. For this part, I made use of the render function part of the ReactDOM package that helped provide DOM specific methods such as render(), which displays the specified HTML code inside the specified HTML element.



```
ReactDOM.render(
  <Provider store={store}>
    <BrowserRouter>
      <App/>
    </BrowserRouter>
  </Provider>,
  document.getElementById('root')
);
```

FIGURE - 5. Displaying the specified HTML

It takes two arguments, the HTML code which is the React element invoked as a JSX interpretation, and the HTML element id ‘root’. The provider tag makes the Redux store available to the connect() calls to the nested components that will need access to the Redux store. This function call will help control the contents of the container Node that we have passed in, with any existing DOM elements already inside being replaced when first called.

Our React application (front-end) will make requests to the server and it will get a response with data from it. We call the APIs using Node from our React application and it will take this data stored in our database to perform the associated business logic inside it. React will then render this output for the user. Below is an example of how data is moving back in forth on our website.

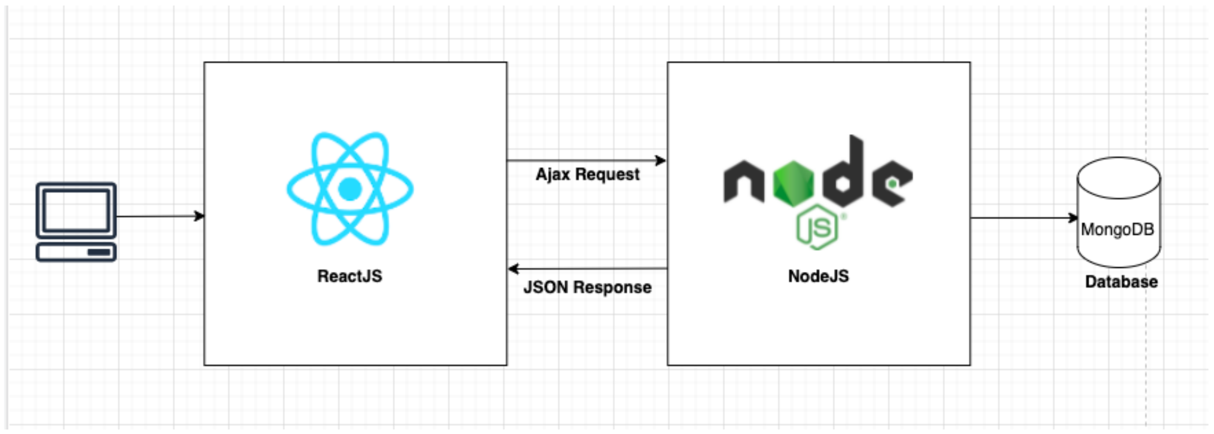
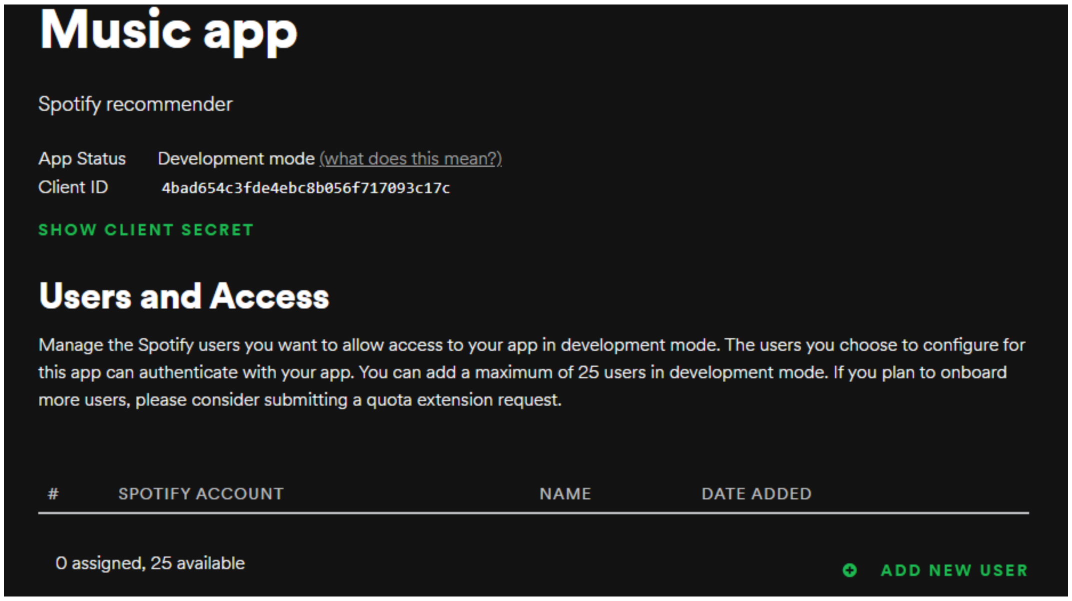


FIGURE - 6. App Architecture

For authentication, I used MongoDB to store the email address and encrypted password for the user by using JWT (JSON Web Token) authentication. I used as a secure way to authenticate users and share information, typically using a private key or secret to sign the JWT. To make sure the token hasn't been compromised the receiver will have to verify the signature to make sure it is valid. JWT provides two types of tokens: access_token and refresh_token. The access token should be expired after an hour for security reasons. Redis does a good job handling this, by deleting this access token after a certain period. Authentication is also done with Node on the back end with MongoDB. A user would have to sign in to see their recommendations or top tracks. After logging in, the user will be asked to authorize their Spotify accounts by using the Spotify API. Their access token will be saved in the database where the user would be registered. Users will then be redirected to our React app after authorization is successful. Data from their Spotify will be returned and only the user's top played tracks will be displayed under the "Songs" tab. The Spotify API uses the redirect_uri to help return users back to the previous page on our website. This is how I was able to create the authorization process on the server-side of my application, and in turn, return user-related data which will be useful for our recommender algorithm. Using the Spotify API requires you to create and use a Developer dashboard on the Spotify website. This is where the developer will fetch their client id and secret which will be stored in .env files for configuration. Granting access and whitelisting yourself along with other users is required to be able to use the Spotify API. Here is an example of what it looks like when setting up your developer dashboard and creating an app:



Music app

Spotify recommender

App Status Development mode (what does this mean?)
Client ID 4bad654c3fde4ebc8b056f717093c17c

SHOW CLIENT SECRET

Users and Access

Manage the Spotify users you want to allow access to your app in development mode. The users you choose to configure for this app can authenticate with your app. You can add a maximum of 25 users in development mode. If you plan to onboard more users, please consider submitting a quota extension request.

#	SPOTIFY ACCOUNT	NAME	DATE ADDED

0 assigned, 25 available

ADD NEW USER

FIGURE - 7. Developer Dashboard tools

REST API

Rest stands for Representational State Transfer. Rest is an architectural style for creating websites using the HTTP protocol. Suppose we have a web client and a web server (API). The client requests a resource, and the server fetches the response from its database.

FIGURE – 8. Taken from: https://developer.spotify.com/documentation

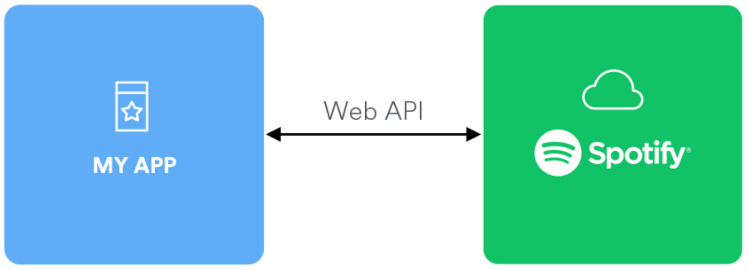


FIGURE - 8. Spotify API

Using simple REST principles, the Spotify web API was used to return JSON metadata from the user's Spotify history which consists of top tracks, music artists, albums, and more. Resources refer to something that belongs exclusively to the server. When a client requests a resource via the restful API, the server sends a representation of that resource. i.e., the representation is what the server sends as a response. When we fetch the metadata from Spotify, instead of our web server, we call the Spotify web server and fetch the information of songs. I have integrated the Spotify SDK via REST API. Our server will first authorize the user as discussed above when signing into their Spotify account. Once the user is authorized, we will call the REST API of Spotify to get metadata information on the user.

Back-end Development

Node

Node is a powerful server-side JavaScript run time environment used to develop fast and highly scalable server-side applications. Built on Google's V8 engine, it is very lightweight and includes a unique I/O model which enables it to handle big flows of data such as heaps of requests in real-time. Each incoming request by a user is handled on a single thread, thanks to its unique I/O model. When we receive a request from a user, it is handled on this thread and is coupled with a callback function which will be called once a task is completed. We make use of several of JavaScript's fundamentals such as asynchronous operations, events, and call-backs. The circle in the figure below is code written by the developer, and the squares are outside of the developer's realm.

FIGURE - 9. Taken from: https://www.manning.com/books/express-in-action

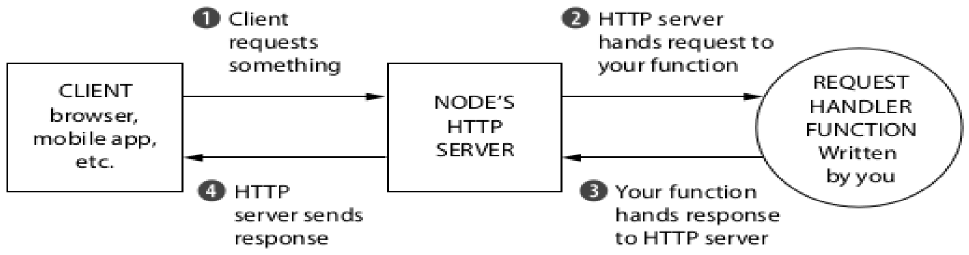

1 Client requests something
2 HTTP server hands request to your function
CLIENT browser, mobile app, etc.
NODE'S HTTP SERVER
REQUEST HANDLER FUNCTION Written by you
4 HTTP server sends response
3 Your function hands response to HTTP server

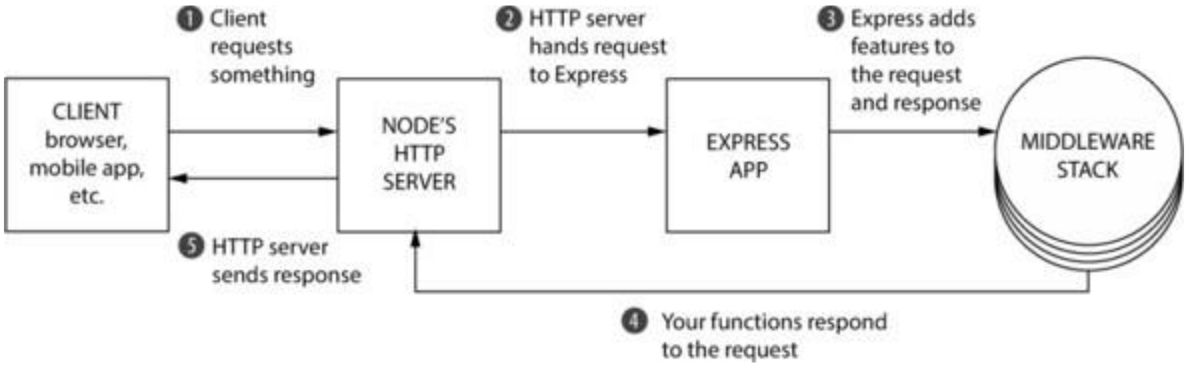
FIGURE - 9. Node's HTTP server

To further develop our music website, we made use of the many frameworks and programs available to us by using Node. The popular MEAN (MongoDB, Express.js, Angular.js, Node) stack was used apart from Angular.js, as I decided to use React.js instead.

Express.js

Since Node won't be too useful on its own, we have taken advantage of Express.js which is a minimal and very flexible Node framework that provides write handlers for requests with distinct HTTP verbs (GET, POST, DELETE, etc.) at different ULR paths (routes). Express is easy to configure and customize. Express was used to define routes of my application based on the various HTTP methods and URLs. It also includes various middleware modules which can be used to perform additional tasks on requests and responses. The middleware used is a type of body parser that parses the body of the incoming HTTP requests with the form submission. This will enable me to access the parameters of the request directly. Middleware end with next(), and can continue calling the next middleware if needed. Additional reasons Express was used was for building an HTTP server by wrapping the backend code of Node and helping make the building of our RESTful APIs simple. To process HTTP requests, our application waits for them in the web browser, and once it is received, our application will determine what action to take based on the given URL pattern. Other information could also be received, which is contained in the POST or GET Data. We then read or write information from our database if needed to complete our request. Once we have processed our request, we will return a response to the user's web browser to create an HTML page. This happens by placing the retrieved data into placeholders embedded into an HTML template. Below is a diagram that explains the flow of requests that happens through an Express application. Circles, once again, are code written by the developer and squares are outside of your scope.

FIGURE - 10. Taken from: https://www.manning.com/books/express-in-action



1 Client requests something
2 HTTP server hands request to Express
3 Express adds features to the request and response
CLIENT browser, mobile app, etc.
NODE'S HTTP SERVER
EXPRESS APP
MIDDLEWARE STACK
5 HTTP server sends response
4 Your functions respond to the request

FIGURE - 10. Flow of requests

MongoDB and Redis

When the user starts performing actions on the retrieved metadata (their top played tracks) such as Like/Dislike or adding a song to a playlist, we store the Spotify id in our database with the performed action and user information. This approach is feasible as Spotify provides REST API which we can integrate into our application. All our data is stored in MongoDB, below is a diagram depicting data storage of user information and collections. Our collections contain various information that will be important when performing a recommendation to our users.

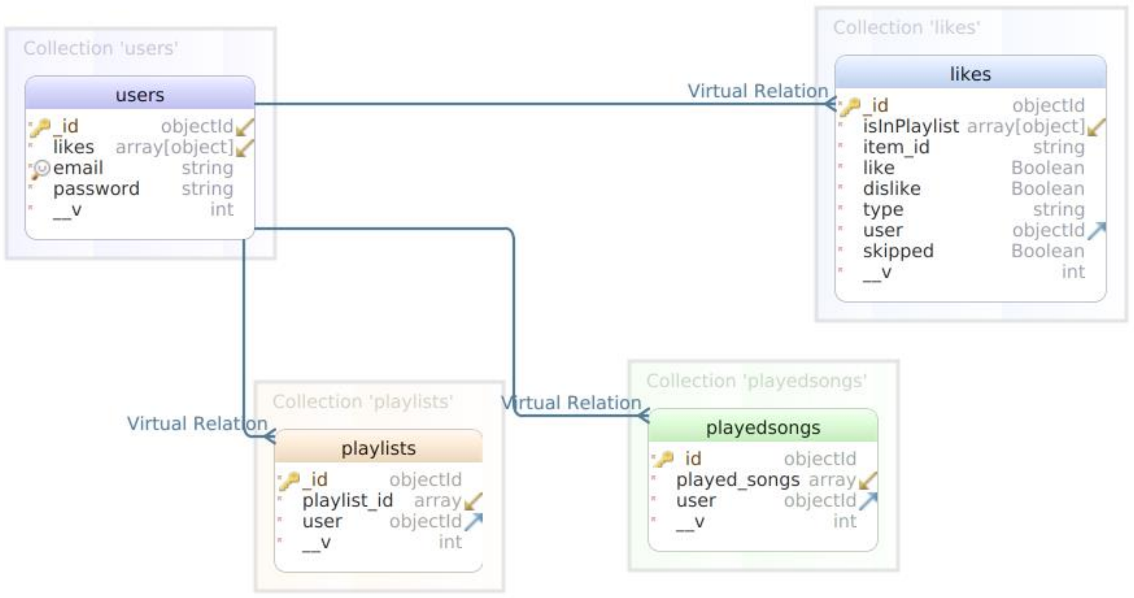


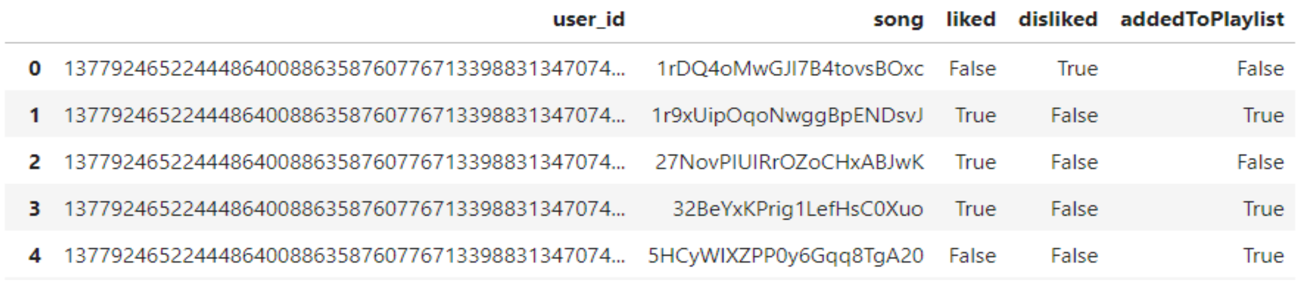
FIGURE - 11. Database Collections

I found it more essential to use a JSON and NoSQL-based database such as MongoDB instead of the originally planned MySQL database after much testing and research. By having a database in place, I was able to leverage the benefits of using JavaScript across the rest of our stack when similar objects stored in the database could be processed by the server and the front end without any additional conversion. Having a flexible schema made it easier for data storage and querying. A big advantage was indexing, a popular MongoDB feature where fields that are indexed because they are 'required' in a collection will be added to a sorted array with the values of this field in all documents. Collections were much easier to search and perform operations on when needed. Redis was an interesting choice to pair with my application, specifically MongoDB. Initially used for storing our authentication tokens such as the refresh_token, validating them and blacklisting them. I also used Redis as it is a great choice for implementing a highly available in-memory cache to decrease the data access latency, increase throughput, and ease the load off our NoSQL database and application. Redis can also serve frequently requested items at sub-millisecond response times and enables easier scaling for higher loads without growing the costlier backend. Providing a flexible data model, horizontal scaling, and fast querying, MongoDB was a perfect fit for my database.

Because python was used for developing the recommendation system, we had to make use of REST API to have Node communicate with our python scripts. Once our data is stored in MongoDB, we must import the data from mongo using PyMongo which is a python distribution that contains many tools to facilitate the process of retrieving data from a mongo database. To achieve this, I first created a connection to the MongoDB where all our data is stored. Data consists of user activities such as what songs a user has liked, disliked, or added to a playlist. We focus on the positive sentiments such as the likes and songs that were added to a playlist by all users that have been stored in our database. Once we are connected, we start querying to the specific database and collections (Likes, playlists…etc.) and expand the cursor to construct the DataFrame. DataFrame is a pandas function which is a 2D labeled data structure that contains rows and columns.

Recommender Algorithm and dataset

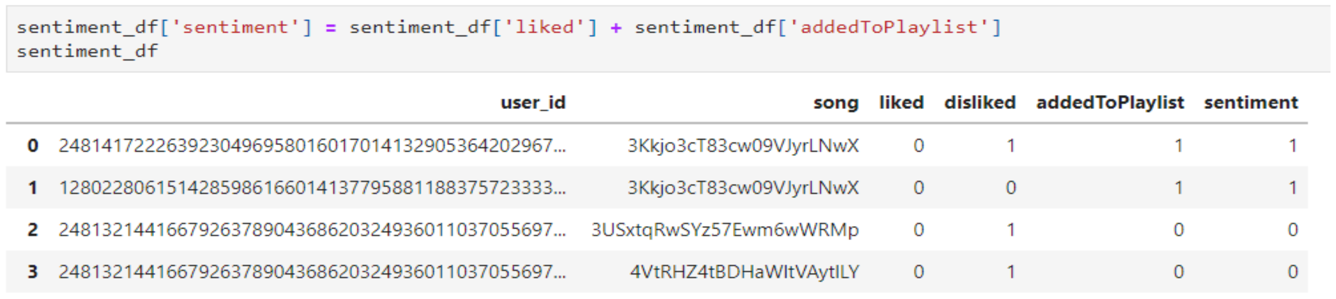
The parameter used to train our algorithm was the Million Song Dataset. It is a collection of audio features and metadata for one million music songs. It enabled me to work with a large-scale dataset and train my recommender algorithm accordingly. I was able to accurately predict what recommendations a user should have when registered on our website. The more data we can gather from user activity the better we can predict what songs a given user would like. The dataset which is registered comprises the clickstream activity of each user i.e., whether that user likes or dislikes a song. Also, if that user likes a song, then we also find up to what degree. Does the user add that song to a playlist or not?



	user_id	song	liked	disliked	addedToPlaylist
0	1377924652244486400886358760776713398831347074...	1rDQ4oMwGJI7B4tovsBOxc	False	True	False
1	1377924652244486400886358760776713398831347074...	1r9xUipOqoNwggBpENDsvJ	True	False	True
2	1377924652244486400886358760776713398831347074...	27NovPIUIRrOZoCHxABJwK	True	False	False
3	1377924652244486400886358760776713398831347074...	32BeYxKPrig1LefHsC0Xuo	True	False	True
4	1377924652244486400886358760776713398831347074...	5HCyWIXZPP0y6Gqq8TgA20	False	False	True

TABLE - 1. User data stored in rows/columns

An important feature can be engineered using such clickstream data: **Sentiment**

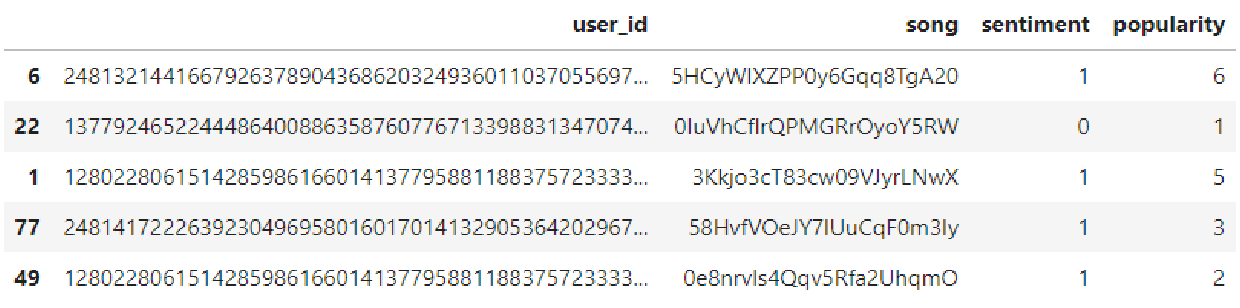


```
sentiment_df['sentiment'] = sentiment_df['liked'] + sentiment_df['addedToPlaylist']
sentiment_df
```

	user_id	song	liked	disliked	addedToPlaylist	sentiment
0	2481417222639230496958016017014132905364202967...	3Kkjo3cT83cw09VJyrLNwX	0	1	1	1
1	1280228061514285986166014137795881188375723333...	3Kkjo3cT83cw09VJyrLNwX	0	0	1	1
2	2481321441667926378904368620324936011037055697...	3USxtqRwSYz57Ewm6wWRMp	0	1	0	0
3	2481321441667926378904368620324936011037055697...	4VtRHZ4tBDHaWItVAytILY	0	1	0	0

TABLE - 2. User sentiment table

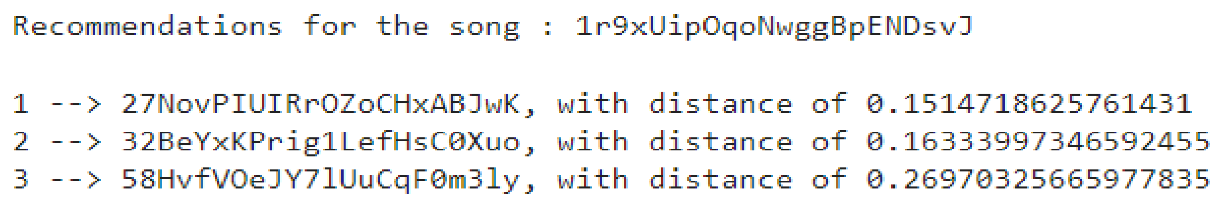
Alone this sentiment can't determine whether a song should be recommended or not. Hence, we try to engineer a public sentiment feature: popularity_score. Based on this popularity_score, I can filter out some songs which are famous as well as the ones that have a good public sentiment.



	user_id	song	sentiment	popularity
6	2481321441667926378904368620324936011037055697...	5HCyWIXZPP0y6Gqq8TgA20	1	6
22	1377924652244486400886358760776713398831347074...	0IuVhCfIrQPMGRrOyoY5RW	0	1
1	1280228061514285986166014137795881188375723333...	3Kkjo3cT83cw09VJyrLNwX	1	5
77	2481417222639230496958016017014132905364202967...	58HvfVOeJY7lUuCqF0m3ly	1	3
49	1280228061514285986166014137795881188375723333...	0e8nrvls4Qqv5Rfa2UhqmO	1	2

TABLE - 3. Sentiment and Popularity score

I then trained an unsupervised Nearest Neighbors Machine Learning model which attempts to find songs with similar user activities. e.g., If a new user shows positive sentiment towards K-pop songs, then the user is highly likely to get recommended songs that are liked by the K-pop fan community. The technical evaluation metric used to train and evaluate the model is called cosine similarity.



```
Recommendations for the song : 1r9xUipOqoNwggBpENDsvJ

1 --> 27NovPIUIRrOZoCHxABJwK, with distance of 0.1514718625761431
2 --> 32BeYxKPrig1LefHsC0Xuo, with distance of 0.16333997346592455
3 --> 58HvfVOeJY7lUuCqF0m3ly, with distance of 0.26970325665977835
```

TABLE - 4. Distance using evaluation metric cosine

Implementation of Nearest Neighbours ML model

This type of recommendation (item-based collaborative filtering algorithm) system is trained on songs having quality sentiment and popularity score above a threshold value carefully selected using hypothesis testing on the user data.

Statistical Thresholding

```
In [30]:
fig, ax = plt.subplots(figsize=(17,8))
plt.axvline(x=100,ymax=0.95, c='red', label = 'Threshold (100)');
sns.histplot(ax=ax, data = album_sentimentCount['popularity'], log_scale=True);
plt.legend(fontsize=25);
```

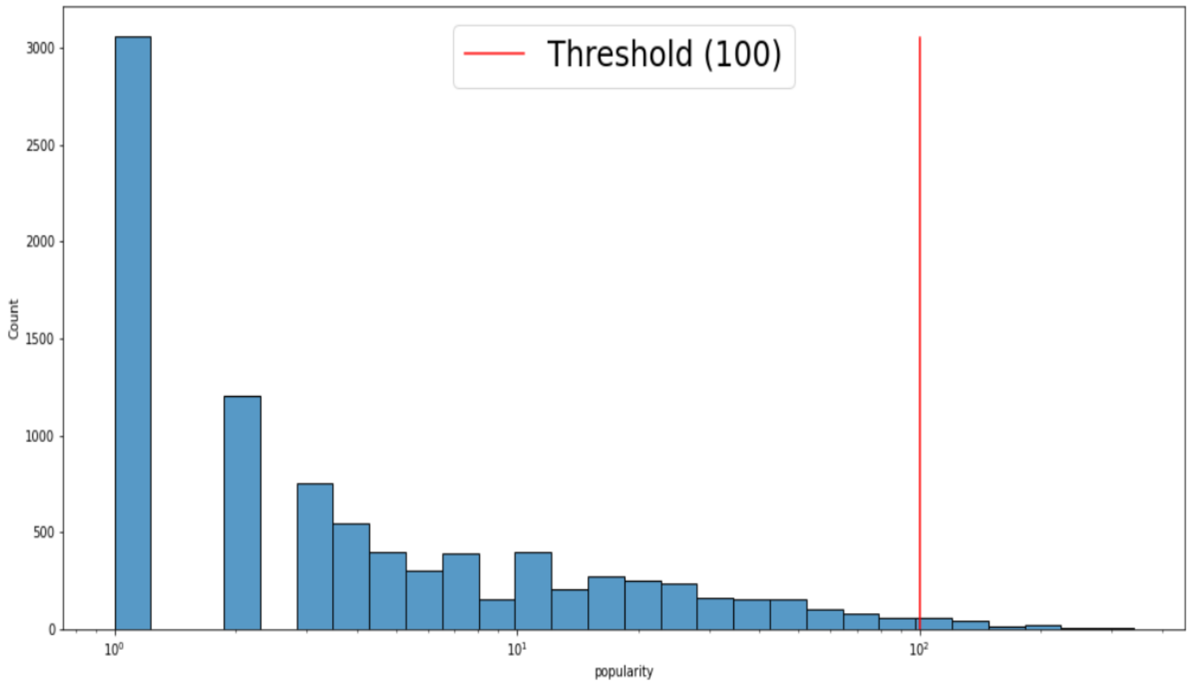
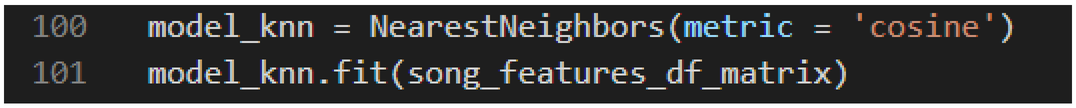


FIGURE - 12. Statistical Thresholding Diagram

Later these filtered songs and their user activity are put in a pivot table (crosstab). What the Nearest neighbor unsupervised learning model does is that given a song id, it tries to find another song whose user activity vector is similar to that of the selected song. This is achieved using the metric cosine similarity. Currently, we can generate two recommendations per action given by the user. If a user likes or adds a song to a playlist, these two recommendations will be produced under the recommendation's header.

All the user activities of each song are considered a vector on a 2D plane. Since the pivot table is very sparse, it is computationally very inefficient to compare songs based on Euclidean distances (as in the case of K-NN classification). Also, there might be a song that is a new addition to the popularity list and there might also be a song of the same genre that has been popular for a long time. All songs are connected by a common theme – the popularity at which they are liked and added to a playlist. My objective here was to try and quantitatively estimate how much similarity exists between the different activity vectors of users.



```
model_knn = NearestNeighbors(metric = 'cosine')
model_knn.fit(song_features_df_matrix)
```

FIGURE - 13. Using the cosine similarity

The code written above is not KNN (K – Nearest Neighbors classifier or regressor). It is a Nearest Neighbors approach, and I am using the distance metric passed as a parameter as the 'cosine score'. Generally, we would use KNN to find similar vectors based on Euclidian distances. But if we rely on Euclidean distances, they would be far apart.

That's the reason why distances between any two songs are compared by the cosine of the angle between their vectors. Songs parallel to each other i.e., having similar user activities throughout would have a strong correlation $cosine(0) = 1$. Hence one song would be recommended when any user shows positive sentiment towards the other song. This type of recommendation system is activated and fueled by user activity data. With an increase in the number of users and their activities, the recommendation system is bound to produce better results.

$$\text{cosine similarity} = S_C(A,B) := \cos(\theta) = \frac{\mathbf{A}\cdot\mathbf{B}}{\|\mathbf{A}\|\|\mathbf{B}\|} = \frac{\sum\limits_{i=1}^{n} A_i B_i}{\sqrt{\sum\limits_{i=1}^{n} A_i^2}\sqrt{\sum\limits_{i=1}^{n} B_i^2}},$$

FIGURE - 14. Cosine similarity algorithm

Conclusion and future work

For my project, I focused on the positive sentiment of music interaction such as liking a song or adding a song to a playlist. At first, I was skeptical about adding a dislike feature because of the negativity behind it. Instead, I thought it would be much more feasible for a user to simply like a song if they enjoy it, and if they don't then simply don't perform an action on the given song. I imagined better sentiments out there that people would rather Express such as liking a song instead of disliking it. It is the appropriate sentiment for a human being and especially for music because of the immense emotion it can generate. That is why I decided to opt for a more appropriate feature: the public sentiment and filter out popular songs liked by all users. This will create a better mood for recommended music – similar to a way one can Express themselves to others without the negativity part. That is why for future work, as a solution to the dislike button, a comment system would be best and more friendly. A user will be able to better Express themselves for the given song by sharing a small comment regarding the song.

While researching for the best recommender algorithm for my website, I came across NLP (Natural Language Processing) for text analysis of music lyrics. I believe this can be a useful algorithm for music lyrics mining and possibly to further analyze user sentiment. This algorithm involves analyzing the elements of any given song such as its lyrics, the tempo, and even the song's beat. We currently make use of the public sentiment in our algorithm, and its presence would prove to be useful for the development of such an algorithm. Since it makes use of accurate classifiers given by music sentiments. To conclude, after much research and testing I was finally able to achieve my goal of developing a music rating website. I successfully integrated and implemented both the front end and back end of my application using JavaScript, followed by the recommender algorithm implemented using python. My results were very promising thanks to the development of my React application. I Integrated the recommendation system into my application using REST API and showed that it is possible to recommend music to users based on positive sentiments such as liking a song or adding a song to a playlist. The entertainment industry is growing exponentially and the need for a music website where users can express their moods and needs has never been more useful.

References

Manning. (n.d.) Using Express.

https://www.manning.com/books/express-in-action

Sciencedirect.(n.d) Cosine Similarity.

https://www.sciencedirect.com/topics/computer-science/cosine-similarity

Towards Data Science. (n.d.) Music recommendation system example.

https://towardsdatascience.com/the-abc-of-building-a-music-recommender-system-part-i-230e99da9cad

Section. (n.d) Building an authentication API with JWT.

https://www.section.io/engineering-education/how-to-build-authentication-api-with-JWT-token-in-nodejs/

Compose. (n.d) Using Redis with MongoDB.

https://www.compose.com/articles/why-and-how-to-redis-with-your-mongodb/

Spotify. (n.d) Spotify API documentation.

https://developer.spotify.com/documentation/web-api/