**CS310 Draft Final Report & Notes**

**Structure**

1. Title Page
2. Abstract
3. Table of Contents
4. Keywords
5. Definition of Terms
6. Introduction
7. Existing Solutions
   1. OutLoud
   2. Spotify Party Mode
   3. Some other shit
8. Methodology
9. Design
   1. Overview
   2. Native Bridging Module
   3. REST API
   4. WebSocket API
   5. Database
   6. Recommendation System
      1. Genre Inference Model
      2. Artist Inference Model
   7. User Interface
10. Implementation
    1. Overview
    2. Native Bridging Module
    3. REST API
    4. WebSocket API
    5. Database
    6. Recommendation System
       1. Genre Inference Model
       2. Artist Inference Model
    7. User Interface
11. Testing and Results
    1. Unit Testing
    2. Integration Testing
    3. System Testing
    4. User-Acceptance Testing
12. Evaluation
    1. Functional Evaluation
    2. Non-Functional Evaluation
    3. Legal, Social, Ethical Evaluation
    4. Project Management Evaluation
13. Further Work
14. Conclusions
15. Bibliography
16. Appendices
    1. Project Specification
    2. Progress Report

**Title Page**

**Mobile application for democratic music playback with a hybrid recommender system**

**Harry Verhoef**

**Abstract**

This project entails the creation of a mobile application that allows users to democratically elect tracks to be pushed to a queue on a host device. Alongside this, a hybrid recommender system consisting of genre and artist inference models is developed to recommend tracks to groups of users, which can then be voted for and consequently elected. People in a social setting must typically rely on a single device to control music playback, that is to say it is unifocal, leading to potential conflicts of preference. The application developed [Insert findings from evaluation]

**Keyword List**

**Definitions**

Music Playback [Meriam Webster]

API?

**Introduction**

Too often in any situation where multiple people are listening to music from the same source, there is only one person/device controlling what music playback. As a result, many people are subject to music they are not particularly fond of. This project aims to give everyone in this situation a voice in the form of a vote that can be cast suggesting the next track to be played from a set of recommendations. The song with the most votes will be enqueued at the end of the current song. The idea is to introduce the most democratic music environment possible, one where the majority of people listening are happy with what they’re listening to. With this in mind, users will also have the ability to up-vote and down-vote a current song. A hybrid recommendation system will be used to shortlist the tracks put up for election, from the universe of all possible tracks that Spotify give access to through their API. A good model in this situation will be one that provides recommendations likely to be voted for by the users in the lobby.

**Existing Solutions**

There have been previous attempts at making music playback a democratic process

**Spotify Social Listening System**

* How it works
* Good shit:
  + Very easy to use
  + Can scan a code to join the party (=> This is where QR code inspiration came from)
* Bad shit:
  + Every user is required to have a Spotify account
  + More anarchic than it is democratic

**Festify**

* How it works
* Good shit:
  + Lets users vote for tracks
* Bad shit:
  + “Admin Mode” allows the lobby creator to override the decisions made by the group, by giving them the power to skip and delete tracks in the queue using their own user interface
  + Relies solely on users to search and then add tracks to the queue, leading to potentially one-sided and un-democratic queue, since users can queue as many tracks as they please.
  + Users can vote for more than one track at a time, meaning .

**Outloud**

* How it works
* Good shit:
  + Lets users cast votes for tracks
  + App can be downloaded and used on a large variety of platforms (=> React-Native)
* Bad shit:
  + hyperlink
  + Shortlisting of tracks to be given to users for voting is whack
  + Process of creating a lobby is too long
  + Votes aren’t anonymous, may lead to

**Deductions**

Solution must take the positives from the existing solutions and attempt to omit the negatives.

**Technologies Used**

With the architectural design established, it is possible to deduce the types of technologies that will be required to fulfil such an architecture. Furthermore, it is useful to define the specific technologies that will be used before any design and implementation has begun. It is much more challenging to integrate these technologies if the developer is unaware of any technology-specific limitations that may hinder said integration process.

**WebSocket API**

Certain features of the application such as voting and persisting the lobby state across all users in said lobby are very stateful and therefore a REST solution is infeasible. It would be possible to have a timer on each device sending an HTTPS request over and over until the lobby is terminated. However, this is only mimicking stateful communication and there are protocols that actually accommodate it. WebSocket is a protocol that provides stateful full-duplex communication over a TCP connection. Importantly, WebSocket interactions between client and server allow the server to send data to the client without first receiving a request. React Native comes with built-in WebSocket support, and AWS recently introduced WS APIs as a separate API that can be developed as part of API Gateway. Therefore, the WebSocket and REST APIs are completely separate. Instead of endpoints, the WebSocket API utilises “routes”, but at a high-level they function similarly.

1. $connect – This is a route required by API Gateway, on the opening of a WebSocket TCP connection between the client and the server, this is the route that is called and will serve to store said connection. Once done, the server can communicate with the client without the need for the client to send a request in the first place, until the connection is closed. This route is invoked by any user who has just joined or created a lobby.
2. $disconnect – Contrarily, the purpose of this route is to terminate the connection instantiated by the \$connect route. This route will be invoked once a user has left a lobby or left the application.
3. $default - In the case where a WebSocket connection attempts to transmit a message with a route that is not defined by the API, then it will fall under this route. In such a scenario, the API will simply respond with a message stating that the route is invalid.
4. Vote – When any member of a lobby votes on a track, this needs to be updated for every other member of the lobby too, since it is crucial that the entirety of the lobby can see the state of the votes. Therefore, this route will take a user vote, add it to the current state of votes, then push it as a message to all users part of the same lobby.
5. Next – As previously mentioned, each lobby has state in the form of the currently playing track. Each user will need to have their lobby’s state updated once a new track is being played on the host device. Once the host device detects that a new track is being played, it will send a message to the WebSocket API along the “next” route stating the currently playing track. Then, the API will send a message to each of the users of the lobby with the same information to suggest that the new track is being played.

**Database**

Given that the backend is hosted on an AWS environment, in order to reduce backend internal latency, the decision was made to use an AWS-hosted database too. There are plenty of different Database Systems that the backend could utilise, but due to the application’s scalability requirement (3a), a schema-less database called “DynamoDB” was decided upon [reference to where schema-less databases are quick]. Since DynamoDB is schema-less, it provides the application with much more flexibility in its data storage. Particularly, unique device ids are represented in different ways on different platforms, meaning that only one device id field is required in a schema-less database. Yet, more may be required in one that has a strict schema, to accommodate the variety of id formats. The database will be composed of 4 distinct tables, as follows:

1. Device – This table will store information regarding each device, such as the device id, the device’s vote-weighting, the lobby that the device is in, etc.
2. Lobby - This table will store lobby data, such as the lobby key, the lobby length, the lobby name, etc.
3. Lobby-Connection – As part of the composite primary key, the lobby and connection attributes represent the sort and partition key respectively. The purpose of this table is to be able to group the connections of the lobby together so that WebSocket messages can be sent to the lobby as a whole.
4. Lobby-Track – Similar to the Lobby-Connection table, this table makes use of a composite primary key so that for each lobby, a votes object can be efficiently constructed.

**Recommendation System**

Since the developer has no experience of developing any form of machine-learning system, let alone a recommendation system, the initial design would have to be based off research. In order to know what to look for in terms of the research, it is important to understand the inputs that can be taken from the system design thus far. The key inputs, per lobby, that the recommendation system may be able to take are:

1. User preferences – Using the thumbs-up/thumbs-down feedback system, it is possible to determine user track preferences. Moreover, it is possible to derive user genre preferences and user artist preferences, since both of these are components of a track.
2. Lobby Name – The name that the host user has given to the lobby may be able to provide inferences as to the type of lobby. For example, if a Warwick Gym employee is setting up a lobby for the gym, they may use “Warwick Gym” as their lobby name, leading to potential inferences about the type of music that is relevant for that social setting.

There could be an argument made that another important input is the base playlist. There is lots of useable data in the playlist that the host user links to the lobby; The base playlist will have a name, which could again be useful for inferences, there is also likely to be a variety of tracks as part of the playlist, which can be used to establish an approximation as to what music the lobby will elect. However, the purpose of this project is to make the music playback process as democratic as possible. Making use of a saved playlist in the host user’s library introduces substantial bias. While the lobby name is also written by the host user, it is intended for the entirety of the lobby. The same cannot be said about a saved playlist on the host user’s Spotify account. User preference is a mandatory input of the recommendation system as per requirement 1f.

The recommendation system that is required for this project falls under a similar category to APC (Automatic Playlist Continuation) systems, which suggest appropriate tracks to add to playlists. The ACM RecSys Challenge 2018 led by Spotify gave teams a dataset of 1 million playlists, along with metadata and asked them to build an APC system [reference to the task outline paper]. In 2019, a paper analysing the solutions by each team was published [reference]. Interestingly, in this paper it is described that most successful solutions consisted of a combination of different models. Lots of successful teams used a model for cold-start situations, attempting to infer tracks to recommend based on only the playlist title. Specifically, the paper states that convolutional neural networks can be useful for “extracting useful information from playlist titles”. Also, the paper suggests that a number of successful teams utilised recurrent neural networks for “modelling the sequence of tracks in a playlist”. From this it is possible to link these two networks to the 2 previously established inputs. A convolutional neural network could be used to extract information from the lobby name, and a recurrent neural network could be used to model a sequence of tracks ordered by user preference, since music is also a sequential process. Using user-preference for approaches such as matrix factorisation is out of the question, since it would be impossible to match users from the project application to random Spotify users.

The 1 million playlist dataset is no longer public, and no publicly available datasets of playlists contain un-hashed Spotify ids of tracks, playlists, artists, etc. Therefore, it is infeasible to build a model trained on public datasets that can interact with the Spotify Web API as is required. For this reason, the models built in this section are trained and evaluated on data scraped from the Web API using a custom Python script.

**Genre**

Using the lobby name, it is infeasible to suggest that specific tracks can be inferred. There are over 50 million tracks [<https://newsroom.spotify.com/company-info/>], yet there are only 126 genres that can be used in the Spotify Web API. From there, using the inferred genre, tracks can be generated using the Spotify Web API’s /recommendations endpoint. The endpoint can take a total of 5 genres, tracks, or artists and produce a list of 100 recommendations.

The basic architectural design plan of the network is to include a 1-dimensional convolutional layer which will extract the defining features of the lobby name, in this case it will extract the words that the resultant inference depends on most. Before the convolutional layer, an embedding layer will be required to produce a vector representation of the input. Whether or not the vector representation will be domain-specific or attained using a pre-trained model is a detail that can be tested and tuned at the point of implementation. In order to increase generality, the model is likely to include a hidden dropout layer. Since this is a Natural Language Processing (NLP) model, prediction inputs will need to be tokenised, and the output may need to be one-hot-decoded.cAny other specific details of the model are superfluous to outline pre-implementation, since it is incredibly likely that said details will change once the model is tuned.

**Artist**

Unlike genres, the number of artists available through Spotify is massive, yet artist inference is still preferred over specific track inference. This is because the output of the model will be used as a seed for the Spotify /recommendations endpoint and a track is too specific. Moreover, an artist inference compliments a genre inference better than a track inference, since a track belongs to one genre and an artist can have many tracks belonging to many different genres. However, while distinct identifiers for each track may be too specific, track-features such as acousticness, valence, and danceability produce a form of latent representation of a track. This latent representation can be used alongside an artist identifier to describe a track relatively accurately, with the benefit of less data being required. Additionally, when using track-features to describe a sequence of tracks, general user mood can be modelled and specific track features may render themselves helpful in the inference process. For example, if a user is listening to tracks with high tempo, danceability and liveness, it would make sense to infer artists that generally contain tracks with similar features. The model would be able to learn to attribute certain artists with the latent representations of their tracks, making the concept plausible. The Spotify Web API contains an endpoint /audio-features that takes a track id and returns a set of audio features, including those mentioned already.

User preference is a vague concept and there is no definitive way to calculate it. In the context of this specific application, user preference represents the collective feedback of all users in a lobby. One feature of RNNs that use back-propagation is that elements at the beginning of the input stream have less effect on the output. This is known as the vanishing gradient problem and is mitigated through the use of a Long Short-Term Memory (LSTM) network, a solution that many teams successfully employed in the 2018 RecSys challenge [reference]. While most see this as a problematic inescapable feature of RNNs, it can be exploited to help process user preference in a manner such that the best received tracks are placed at the end of the input stream and worst at the front. Input streams can be represented as vectors, maintaining the order.

<Calculation for vote weighting>

Equation (5,1) represents the reassignment of the vote-weight w for a user i. The implication of this vote weight is that when a user I votes for a track, wi is how much their vote is worth. Equation (5,2) demonstrates an example vector warrow of user weights in a lobby, and r is the vector of track feedback in the same lobby. For each user i in a lobby, wi is the user’s vote-weight and ri is the user’s feedback of the currently playing track. Ri = 1 if thumbs-up, -1 if thumbs-down and 0 if neutral.

Vectors w and r as shown in (5,1) represent examples of vote-weight and rating vectors in a given lobby. For each user i in said lobby, wi denotes the user’s vote-weight. That is the amount that their vote is worth, but can also be used as a measure of user credibility, determined by the rest of the lobby. For the same user i, ri denotes the rating that user i has given to the currently playing track, determined by the thumbs-up/thumbs-down feedback system. The vote-weight adjustment demonstrated by equation (5,2) is applied to any user i who voted for the track that is currently playing, and is applied at the end of the track. In (5,2), n represents the total number of users in the lobby. (5,3) Demonstrates the derivation of ri, it is worth noting that that the thumbs-up/thumbs-down status is reset once each new track begins, meaning ri is also reset.

Once a track t has concluded, the adjustment \ref{eq:readjustment} is applied to the users that voted for track t, and then the track weighted-ranking xt is calculated using equation \ref{eq:weightedranking}. The weighted-ranking is then compared with the history of track weighted-rankings for that lobby. As demonstrated by \ref{eq:order}, the artist id and track features of the 5 tracks that have had the highest weighted-ranking in the history of the lobby are used to construct a vector a and matrix f. Each ft is a set of 8 track features: acousticness, danceability, energy, instrumentalness, liveness, loudness, speechiness and valence. Each at is a Spotify artist id. a and f are ordered in ascending order of track weighted-ranking to exploit the aforementioned vanishing gradient problem. A heterogeneous tuple will be constructed using a and f, which will be taken by the RNN to generate a prediction in the form of an artist id.

**Combining Models**

The genre inference model is static by nature of its input, the lobby name is not going to change after the lobby is created, so neither will the inference. Since lobby state is dynamic, the genre inference will only be used in cold-start scenarios, specifically where less than 3 tracks have already been played. The artist inference can be utilised any time after 1 track has been played, so for the 2nd and 3rd track of the lobby, both of the inferences will be used to generate recommendations. The decision making process can be seen in figure x in the form of a flow chart.

**User Interface**

The User Interface is the last fundamental component that is to be fully developed, seeing as it does not in essence add any functionality to the application. For this purpose, an initial, temporary UI is to be developed that’s sole purpose is to support the development of more crucial functionality. At the final stage of the development of the application, a new and more complex design will be adopted by the UI.

Before beginning the design process of the 2 outlined UIs, it is first important to establish a constant foundation upon which both will be developed. This can be shown in the form of a component map, since the UI will be composed of react components.

Component Map:

<https://www.gloomaps.com/QZhwP9PiwZ>

**Implementation**

In this chapter, each section describes the implementation of a fundamental component, following its design.

**Overview**

<Design and Implementation Timeline>

**Native Bridging Module**

**MAKE SURE TO TALK ABOUT HOW THIS IS IOS AND HOW YOURE DOING IOS FIRST**

**Implementation**

As shown in figure x, the implementation of the native bridging module took much longer than expected. This is partly due to time that was allocated for learning Objective-C, but namely since the machine that was used to build the application on to the device took approximately 15 minutes to do so. Meaning that each time the bundle server (which is required to build react-native applications) needs to be restarted, it takes 15 minutes. The bundle server needs to be restarted when any changes to the module need to be made. This, when combined with the fact that the language is unknown to the developer, made it a very timely process. Nonetheless, each of the designed methods were implemented successfully.

The first task was to create a module that could be accessed by the client react-native application, utilising the NativeModules built-in library. This meant writing a simple Objective-C class and header file, then exporting a simple void method that outputs a string to stdout, as a macro. Once done, the method can be invoked, passing in an anonymous callback function, as this is the recommended way to receive data from a natively-written method. It’s a non-conventional form of integration, and for the purpose of clarity, code examples are demonstrated by figures x, y and z.

Use the images from the progress report, although maybe a more high res version of the react code

Once the integration between these two fundamental components had been successfully established, it was important to establish communication between the module and the Spotify SDK, so that development on the authorisation process could commence. As per the Spotify SDK “Getting Started guide” [reference], the Spotify SDK framework was installed as part of the XCode project. Then, the Spotify framework can be accessed simply by including the classes required.

Figure x shows the OAuth2 “authorisation code” flow, obtained from the Spotify Web API documentation. The authorisation code flow was chosen over other authorisation flows such as “client credentials” and “implicit grants”, since it accommodates both the refreshing of access tokens and the ability to access user credentials. Refreshing access tokens means users do not need to be prompted to grant access to the application each session (thereby skipping step 1), and accessing user credentials is important to acquire the base playlist. Step 1 of figure x is done by the Spotify SDK and does not require any authorisation endpoint hosted on the application’s backend. Step 2 invokes the /swap endpoint, which will consequently call the relevant lambda function and from there the Spotify Accounts Service will return access and refresh tokens. The backend API will act as an intermediary proxy throughout the authorisation process. However, the backend API had not been developed at this stage, and so a node.js script was being locally hosted which used the express library [reference] to handle HTTP requests. Once the /swap endpoint was functional, the module was able to store the access and refresh tokens as session attributes, and then when session methods such as playURI were invoked (to play a track), the access token would be used alongside the request to the Spotify application. The /refresh endpoint was also implemented locally which meant if a session had expired (as it does after 1 hour), then the module could retrieve a new access token using the refresh token. It is clear that going forward, in order to implement a comprehensive authorisation utility, the backend API hosted on AWS would need to persist the refresh token for each host user.

**Difficulties Faced**

The implementation procedure was timely and there were many bumps in the road regarding the development in the native bridging module. Other than those previously mentioned, the specific implementation problems included Objective-C’s lazy function evaluation, the fragility of the Spotify SDK appRemote connection, and the intertwining of different threads.

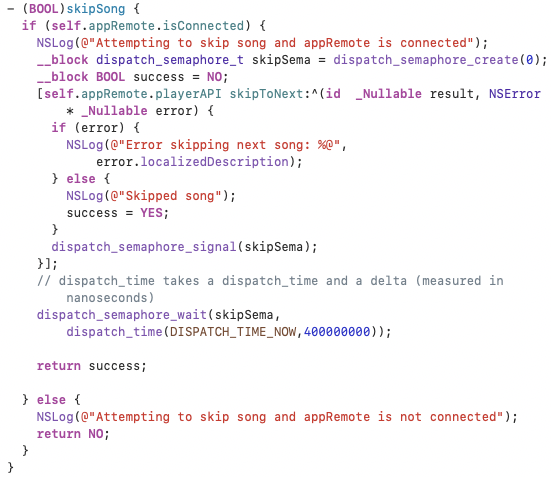
Lazy function evaluation

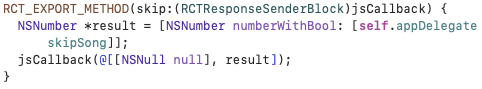
* If a compile-time constant is to be returned by a function, then when it is called the function may return the constant before the body has been executed. This feature of the Objective-C compiler, LLVM [reference], renders the callbacks of such functions completely futile. The callback can be invoked before the method has finished executing, meaning any parameters to the callback may be undefined. This problem was first encountered when the authorisation method of the module was called which returned a “YES” (Objective-C equivalent of a true boolean). As a result, the error and result parameters would always be null, making it impossible to know if the authorisation had succeeded. More importantly it had meant that any consequent body of code, that may rely on the successful completion of authorisation, would execute before any such authorisation had taken place.
* In order to fix this problem, semaphores were utilised in order to force the thread to wait until the semaphore had been signalled before proceeding through the block of function code. The semaphore would be instantiated at the beginning of the function and once the evaluation had concluded, it would be signalled. Placed just before the return statement of the method, the dispatch\_semaphore\_wait function told the thread to wait for the signal before returning. Once the function returned, the callback that had been passed into the invocation in the client application could be evaluated with the correctly assigned error and result parameters.

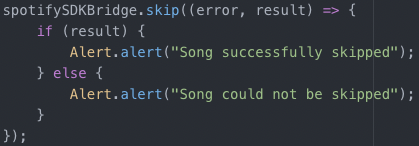
Fragility of the Spotify appRemote connection

The appRemote is an object that provides an API which can be used to easily interact with the Spotify application functionalities such as controlling playback or retrieving user information. It has 2 possible states, connected and unconnected, to the Spotify application. The appRemote can only be connected if a non-expired access token is stored as part of the session. If the user is not using the application, the appRemote must be disconnected, and if they are to reload the application then a new connection attempt must be made. In order to connect the appRemote, there must be music playing on the device, due to iOS security restrictions. It is inconvenient for the user to be required to have music playing in the background while the project application connects to the Spotify application.

To avoid this inconvenience, during the authorisation process the appRemote will attempt to connect. If music is playing, it will connect, otherwise it will not. However, when the host user first joins the lobby they will be asked to start the lobby music playback by selecting a track among the collection of recommendations. At this point, if the appRemote is connected, it will simply call the playURI method part of it’s API. Conversely, if it is not, the module will invoke the authorizeAndPlayURI method. This method will attempt to authorise the user again, and when the app-switch is made to the Spotify application for authorisation, it will play the requested track before attempting to connect the appRemote.







Spotify authorisation Flowchart

**REST API**

1. As mentioned in chapter x, the rest API was initially implemented as a single node.js file that was hosted locally, in order to support the development of the authorisation process for the native bridging module.
2. Once the native bridging module had been developed the rest API could be migrated to the AWS servers. Migrating the API to the AWS servers was very simple since the API at the time consisted of 2 endpoints, /swap and /refresh. Nonetheless, they provided adequate functionality to test the interaction between the client application and the REST API. Therefore, if these two endpoints could successfully be migrated, then the process of implementing the other endpoints would become much simpler, since the architecture that supports each endpoint will be the same.
3. Firstly, a new API was created as part of the API Gateway service provided by AWS. In doing so, a Uniform Resource Locator (URL) is allocated to the API. Following this, resources can be created, these represent the endpoints of the API, so the /swap and /refresh endpoints at this stage. Alongside the resources, the API Gateway interface allows API engineers to associate a resource with a set of HTTP methods. In this case, OAuth2 dictates that POST is the required method for both /swap and /refresh. Then, a node.js lambda function is created in the AWS Lambda interface, making up the code that is to be executed upon successful invocation of the corresponding endpoint. As a result, both swap and refresh lambda functions were created and linked to the API resources using proxy integration, as shown in figure x. With this, the swap and refresh lambda functions will execute once a POST /swap or POST /refresh request is sent respectively to the API. The code used to implement the endpoints in the local node.js server only required minor tweaking to do with sending back the response, since express is no longer needed.
4. However, the code was also dependent on another node package, axios. When using a combination of lambda functions, they can only interact with each other by using the API they themselves compose. That is to say, they’re completely modularised, and each lambda function needs its own node\_modules folder if it requires a collection of node packages. In order to perform the installation of a node package to lambda function, the simplest method is to write the lambda function code locally as part of an index.js file contained within its own directory. Then, initiate node package manager, install the required packages to the node\_modules directory, zip the entire lambda function directory and upload it to the online AWS Lambda interface. This was the method employed in the implementation of all the lambda functions. All of the endpoints of the API were developed alongside their associated lambda functions, but not all of the lambda functions were fully implemented, since they may depend on functionality that is yet to be developed. For example, the /get\_recommendations endpoint cannot be fully implemented until both the artist and genre inference models are designed, trained and deployed.

**Difficulties Faced**

Latency, high modularisation increases readability of the API as a whole but drastically increases overhead too, since for each endpoint the dependent node modules need to be loaded into memory. As a result, if a sequence of endpoints need to be called in quick succession, each reliant on the previous invocation, then the latency accumulates. To decrease this overhead, the API could employ a monolithic structure which would mean overhead is significantly reduced. However, while the migratioan of the API to a monolithic structure would be relatively simple, it would cost much more to have a constantly running AWS S3 instance act as the server, and low latency isn’t specifically a requirement. Requirement x specifices that the API should be scalable, and it could be argued that AWS Lambda is more scalable than its monolithic counterpart, since each lambda function satisfies a specific purpose, and it therefore requires less computational power to load a new lambda function instance than to handle a multitude of requests simultaneously, which is a strategy that assumes the request could utilise the entire universe of functionality that the API can offer. This is backed up by the AWS documentation, as is shown in the quote taken directly from the article on lambda scalability [reference].

Requirement 3c specifies that the lyrics for the currently playing track should be displayed as the track is being played, if the lobby setting is enabled. When it came to the implementation of this functionality, the Spotify Web API could not be utilised, since there is not a /get\\_lyrics resource. Spotify use the Genius API [reference] to obtain their lyrics data. Therefore, a /get\\_lyrics endpoint was created as part of the REST API and a lyrics lambda function was instantiated. The idea was that the lambda function would take the Spotify track id and output the lyrics for the track. This seemed simple enough, but it turns out the Genius API do not offer a lyrics accessor resource either, and after further inspection, it became clear that this was for copyright reasons. Instead of lyrics, however, the Genius API do offer “annotations” for tracks. These annotations are user-written descriptions of the track, which could include the meaning behind the track, or how and when was written/made. While this is a completely different feature and certainly doesn’t satisfy requirement 3c, it does offer functionality and information that the user can interact with, for each track, which is essentially the purpose of the lyrics feature.

The Genius API does not offer a service that maps a Spotify track id to a Genius track id, but does store the Spotify id as part an external URL within the track object of some tracks. Once again, the /search endpoint had to be utilised, this time with the Genius API. The Spotify id is used to determine the track name (using the Spotify Web API), and then the track name is used to attempt to find the track as stored on the Genius API, which can then be used to acquire the annotations. As shown in figure x, a small adaptation was made to the UI so that the user could double tap the album cover of the currently playing track and it would be replaced with the fetched annotation. This worked for most tracks, but there are inconsistencies in the names of tracks as stored on the different APIs, meaning that it is often the case that the annotations cannot be found, and since they’re user-written, the actual annotations could be inaccurate, profane, meaningless, etc. For this reason, the feature was disabled but the UI adaptation remained since it offered the possibility for users to double-tap to obtain the lobby QR code, once this functionality had been developed.

**Dynamo Synchrony**

* Dynamo interactions by default not asynchronous, have to use await and promise method

Similar to that of certain Objective-C methods, some DynamoDB API interactions such as updateitem and getitem evaluate to null before the action is completed, which is updating and getting an item respectively. This is not much of an issue when using the updateitem method, where the action is still completed and what the invocation evaluates to is not important. However, this can and did lead to confusion when using methods such as getitem, where the purpose of the method is that it returns the attributes of a specific item upon invocation. This is due to the fact that database queries are mostly asynchronous. As a result, and discovered through experimentation, the query can be converted to a promise, and then the compiler can be asked to “await” the fulfilment of the promise. Using the Javascript ES6 async/await syntax means that the whole process is wrapped in a try-catch statement, and therefore in the event where an error is thrown after invoking a database query, the lambda function will respond with a HTTP 500 (Internal server error) status code. The process of converting asynchronous queries into promises and awaiting their fulfilment is demonstrated twice in figure x.

**Landing**

1. First UI Component, as such, the reader will notice it has a different colour scheme to the others. The wireframe and the implementation differ in resolution since the wireframe uses the iPhone X as a canvas and the device used for development is an iPhone 8. As per the designed wireframe, the implemented component serves only 2 functions: Joining a lobby and creating a lobby. Users can join a lobby by typing the lobby key into the built-in TextInput component, then pressing the built-in Touchable component which sends an HTTPS POST /join\_lobby request with the unique device id and lobby key as part of the body. The QR code method of joining a lobby was not implemented at this stage since it is not a high priority requirement.

**CreateLobby**

1. The CreateLobby component was implemented strictly according to the wireframe. What isn’t shown in figure x is that before the carousel of playlists is displayed to the user, the user must first connect to Spotify, by pressing a Touchable, then press another touchable which triggers the application to fetch the user’s playlists using the/get\\_playlists endpoint. Upon receival of this HTTPS request, the playlist objects will be loaded into the carousel. The carousel component was not developed by the developer and is instead part of the react-native-carousel node package [reference]. Users can enter a lobby name, tweak the lobby settings, choose a base playlist, see playlist information, then create the lobby. Once the lobby is created (after /make\\_lobby invoked), the user will have the HostLobby.js component pushed on top of their navigation stack.

**Recommendation System**

1. No usable dataset that would fit the general design of the model outlined in design section.
2. Wrote a python script that scraped data off the Spotify Web API, using access tokens obtained from the REST API /refresh endpoint through the Postman software [reference]. Used a txt file containing 450k english words.
3. Pseudo code of the python
4. Resultant dataset
   1. Limitations => Non-english => Reduced availability
   2. Lots of the English words used in the search found 0 playlists, they were too long and obscure, which would result in many of predictions being Out of Vocabulary for the model (OOV). Subwords of 3-4 characters yielded many more results, meaning the data scraping process could become much more efficient.
   3. Limitations => Not very domain specific
      1. Used a CSV of playlist names
5. Squeezing Genres, cosine similarity
6. Graphs of train accuracy (Hyperparameter Tuning)
7. Graphs of validation accuracy (Hyperparameter Tuning)

The script was built to save the scraped data every 100 words in a new CSV, at which point the access token used to interact with the Spotify Web API would be refreshed using the /refresh endpoint. The CSVs could then be concatenated using another Python script to produce one large dataset containing all the data that the script has scraped. After 5 hours of the script running, there were 285 CSVs, meaning 28500 words had been used to search for playlists, and a total of 318379 playlists had been scraped, and their genres approximated. On average, each word yields approximately 11 playlists.

It was evident, however, that there were limitations to using a large list of English words. For one, many non-English words will be Out-Of-Vocabulary (OOV) for the model, reducing the availability of the application, which is a requirement. Also, it was evident that shorter, more general words, or even sub-words like “ing” produced plenty more results than large specific words such as “nonsubstitutionally”. The latter is an example of a word that will not find itself belonging to many playlist names. Furthermore, there are no spelling errors in this dataset of English words, and playlist names are written by people therefore subject to error, if the model were to be trained using only correctly spelled words, then a large proportion of playlist names would be OOV. In order to counteract these issues, the script was adapted to generate random permutations of 3 and 4-letter sub-words and use these as the search query. These random subwords have no language and will collect playlist names written in a large variety of languages, and since these random permutations have no requirement to be real words, playlist names with spelling errors will also be found. In addition, a dataset of playlist names [reference] was used to create another adapted script where instead of using English words or random sub-words to search, random playlist names were used, gathering domain-specific data. Then, all 3 of these datasets could be combined to build a much larger dataset, with both general (random sub-words and English words) and domain-specific (using real playlist names) data, catering for correctly and incorrectly spelt words, English and non-English.

Talking about the results from the adapted scripts

With these various datasets processed, there was a total of 1700 different class labels (genres). However, there are only [TODO] genres that can be used as seeds in the /recommendations endpoint as part of the Spotify Web API, which turns the inferences into tracks. Therefore, in order to have a usable model, the number of classes in the dataset needs to be reduced down to [TODO], through the clustering of classes. Cosine Similarity is an algorithm that “gives a useful measure of how similar two documents are likely to be in terms of their subject matter” [<https://en.wikipedia.org/wiki/Cosine_similarity#cite_note-1>]. In this case, 2 class label strings would be converted into vectors so that the cosine similarity between the 2 strings could be calculated.

In essence, genres that share lots of similarity in plaintext are assumed to be of similar genre generally. For example, the genre “Swedish hip hop” shares most plaintext similarity with the more general cluster “hip-hop” than it does with “folk”, “classical”, etc. As a result, after execution of algorithm x, all examples in the dataset who have the class “Swedish hip hop” will now have the class “hip-hop”. This new class can be used in tandem with the Spotify Web API /recommendations endpoint to obtain a set of tracks belonging to that genre, in this case, “hip-hop”. The developer is aware that using a string comparison mechanism to cluster the classes is relatively naïve. It was immediately evident following the execution of algorithm x that there are some erroneous cases; “grime”, a british rap genre, shared most similarity with “anime”, a genre primarily composed of Japanese cartoon soundtracks. A more sophisticated approach could attempt to mitigate these blatantly erroneous cases, maybe by learning embeddings for each or through the utilisation of an autoencoder. However, the decision was made by the developer to employ the cosine similarity clustering method with a catch. If the cosine similarity is less than 0.5 then any playlist belonging to said genre shows no sign of plaintext similarity with the usable set of genres, and thus it will not be used to train the model. This condition helps to ensure that the model is not trained on false conclusions.

The randomly selected set of hyperparameters that produced the highest validation accuracy (of 38.5%) were:

* 5 epochs
* 219 Filters
* 23 Kernel Size
* 16% dropout
* 83 Embedding Dimensionality

As is shown by the figures, there is only one hyperparameter that seems to demonstrate correlation despite the random selection of the other hyperparameters; Epochs. It can therefore be deduced that the number of epochs has a large effect on the overall validation accuracy regardless of the other hyperparameters. With this in mind, another hyperparameter tuning job was initiated whereby the number of epochs would vary between 1 and 7, and other hyperparameters remained constant. The results are shown by figure x. As is clear, validation accuracy tends to its global maxima after around 5 epochs.

The set of hyperparameters was used to train the model on the full dataset which lasted x hours, and resulted in a training accuracy of x and a validation accuracy of y, as can be seen measured for each epoch in figure x. It is clear to see that the model is overfitting, Thus, the dropout was increased to 25% and the results can be seen in figure y. The model weights and configuration were saved to h5 and JSON files respectively, so that the model could be deployed on AWS SageMaker.

Train the model on a 10% dataset

Train the model on a 50% dataset

Train the model on 100% dataset

Train the model on a 200% dataset (maybe 2 epochs)

Figure x demonstrates the validation of the accuracy after a number of training epochs. The model was trained on 80% of the total concatenated dataset and validated against the remaining 20%. Once each epoch concluded a callback method was invoked which measured that validation accuracy, thus removing the need to create 5 different models all trained on a different number of epochs to produce the same result. The validation accuracy after 5 epochs was 51%, a substantial increase on the model trained with the same parameters using 10% of the dataset and only 3 epochs. While 51% may not seem like a large accuracy score for a classification model, it is worth noting that the model is trained on user-data, where it is certainly possible that one input maps to various outputs. For example, in the training data there are plenty of cases of playlists that have been found that have the same name, yet are composed of different tracks that yield a different genre when given to the genre approximation heuristic. Moreover, it is evident that the size of the dataset has a profound impact on the accuracy of this model, probably since it is fundamentally a Natural Language Processing (NLP) model, which require around 1 million words to learn embeddings well [reference]. The number of words contained within the dataset is [amount]. With this in mind, the data scraping scripts were loaded up again and much more data was gathered, approximately double what the model whos validation accuracy is shown in figure x was trained and tested on.

**Artist Inference Model**

**Preprocessing**

1. Using the script to gather playlist information
   1. Going through each user
   2. Obtains the track information too, to determine model general mood of the listener.
2. Using these playlists to create an APC model, whereby the model will attempt to guess the main artist of the track that the user next listens to.
3. Over 1 million playlists collected

**Model**

1. Splitting the read data into chunks of 6 tracks, taking the artist ids and track features of the first 5 as an input and the artist id of the 6th as the class label.
2. With over 1 million playlists collected, and with each OHE artist id having 80000 features, attempting to use the conventional model.fit method failed, as 126GB was needed to be loaded into memory.
3. Model Architecture
4. In order to counteract this, the training job would invoke a python generator, requesting the next batch of data. This meant that the training and testing datasets were computed incrementally and that memory wasn’t overloaded.
5. Hyperparameter Tuning
6. Results
   1. Graphs of hyperparameter tuning
   2. Example predictions
7. Training vs Test

The artist inference model will take a sequence of 5 track features, along with their main artists, and predict the main artist of the next track that the user will listen to. In this case, user-built playlists will be used to mimic sequential listening sessions. After all, the competitors for the 2018 RecSys Challenge used playlist data as the foundation for their models too. While the resultant model will be categorised as an APC model, it is perfectly capable of recommendations, due to the fundamentally sequential nature of the tracks played by the lobby, which can themselves be used to compose a playlist.

Before the model was developed or any data was gathered, the mechanism that determines the input to the model (see chapter x) was first implemented into the /get\\_recommendations resource as part of the backend REST API. An ordered array of heterogeneous pairs is stored in each lobby item within the lobby table of the database, demonstrated by figure x, and is updated for when any track concludes within the lobby. The pairs are composed of an xi and a ti, these represent the track\\_weighted\\_ranking and the track\\_id respectively. In the interest of saving database capacity, only the track\\_id is stored in the way of track data. The track id is then used to fetch the main artist and track features, to provide the inputs to the model (see equation x).

Before the model was even to be compiled, it was clear that the entire training process would be very computationally expensive, the training matrix, X, had a shape of (255946, 3, 9). Consequently, when the model was compiled, the training job failed as the script attempted to load approximately 126 GB into memory. Without access to a machine with massive memory, there was no way to use the conventional model.fit method to train any model with this collected data. Instead, a python generator [reference] is used to incrementally determine and return the data for each batch. The generator is invoked when either of the training or testing jobs have finished training/testing on the current batch and require the next batch of data. The generator invocation is passed as a parameter to the tensorflow fit and evaluate methods. With this unorthodox approach, it is possible to train and test the model while developing the data required incrementally and by doing so avoiding the overconsumption of memory.

However, using the generator approach drastically increases CPU overhead, since data needs to be constantly loaded in to and out of memory. Consequently, the training and testing process can be relatively timely, which is unsuitable for a hyperparameter tuning job where 60 or so models are trained and tested. Instead, small subsets of the data collected can be used alongside the conventional model.fit and model.evaluate methods to create a hyperparameter tuning job similar to that of the genre inference model. Then, the parameters that are determined to produce the maximum validation accuracy can be used to train a model on the entire dataset using the generator approach, since the nature of the data doesn’t change with size.

The model architecture designed in section x was implemented using the keras functional API [reference]. The functional API was chosen over the sequential one since the functional API is more able to accommodate multiple inputs.

After executing an initial training job with standard parameters, it was clear that a different training and testing approach to that which is most conventionally adopted was required. Attempting to fit the model by simply passing in the input matrices and ground-truth vector led to the development machine attempting to load 126GB into memory. This, of course, meant that the training job failed as the machine simply could not supply the capacity of memory requested. To counter this, a Python generator was developed that incrementally determined and loaded into memory the data required for training/testing each batch. The artist ids were encoded as positive integers and one-hot-encoded within the generator for each batch. Consequently, the shape of X changed from (255946, 3, 84478) to (255946, 3, 9), making the training and testing of the model much more space efficient. Conversely, CPU overhead is increased, as the computation for each batch has to be done incrementally, instead of all at once.

The resultant large CPU overhead makes hyperparameter tuning a difficult task. On the entire dataset, it takes approximately 4 hours to train the model for each epoch, and it is unknown how many epochs are required to make the validation accuracy converge. Therefore, a Python script was written to produce small subsets of the entire dataset, in the hope that the results of any hyperparameter tuning job could use the subsets to produce an optimum set of parameters that could be scaled up later on to train and test a model using the entire dataset.

With very small subsets of data, the proportion of classes to examples is higher. This is shown by figure x, where the gradient of the graph represents said proportion. Consequently, the model will be less equipped to learn the relations between classes (artists). Moreover, it is much more likely that unseen artist ids will be given to the model if it is trained on a very small subset of data. Thus, validation accuracy is likely to suffer on very small subsets of the data, and should not be used to scale any hyperparameter tuning results to larger datasets .

Training a model on 10% of the dataset with the same initial parameters costs approximately 10 minutes per epoch. While this is a substantial improvement on 4 hours, a tuning approach similar to that of the genre classification model where 60 models are trained with 3 epochs would take approximately 30 hours, making it largely impractical. Instead, the approach adopted by the artist classification model takes the 10% dataset and assumes that each hyperparameter is independent from one another. This approach allows each hyperparameter to be tuned individually, one at a time, in the hope that the perceived optimum hyperparameters can be implemented on a model which is trained and tested on the entire dataset.

**Epochs**

<fig>

Demonstrated by figure x, the validation accuracy begins to converge after approximately 10 epochs. After which, it fluctuates negligibly. Therefore, the final model will be trained using 10 epochs. However, the following hyperparameters will be tuned using only 1 epoch, since the validation accuracy attained each iteration is only useful when compared to the other iterations.

**Embedding Dimensionality**

**A close up of a map

Description automatically generated**

Embedding dimensionality represents the dimension of the space output by the embedding layer, and it is evident that it shares a strong positive correlation with validation accuracy. The minimum validation accuracy was attained by the model with an embedding dimensionality of 200, which took 43 seconds to train and achieved a validation accuracy of 5.21%. The most accurate of all the 15 trained models was the model with an embedding dimensionality of 2800, which achieved 9.71% and took 427 seconds to train. Both models were trained using a single epoch. While the change in validation accuracy is substantial, so is the difference in time taken to train each model. The final model may not be able to employ an embedding layer with such high dimensionality if it means the training time is impractical.

**RNN Dimensionality**

A screenshot of a cell phone

Description automatically generated

The RNN Dimensionality parameter describes the dimension of the output space of the recurrent layer. It is clear that there is some positive correlation between RNN Dimensionality and validation accuracy, though it is relatively unstable and the difference in validation accuracy between the minimum and the maximum is only approximately 2%. 15 different models were trained, each consisting of a single epoch. The maximum validation accuracy was attained by the model with an RNN Dimensionality of 360, and the minimum with 120. The latter took 217 seconds to train, and the former took 214 seconds, meaning that the RNN Dimensionality of the model has seemingly no effect on the time it takes to train.

**Dropout**

**A screenshot of a cell phone

Description automatically generated**

Interestingly, figure x indicates that there exists some negative correlation between the dropout percentage and validation accuracy, which is seemingly counterintuitive, since the dropout layer serves to specifically increase validation accuracy relative to training accuracy. This could potentially be attributed to the fact that the dropout layer attempts to reduce overfitting, and that after 1 epoch there is very little room for overfitting; The validation accuracy is often greater than the training accuracy at this point, thus underfitting. To investigate this, the relation between validation accuracy and dropout percentage was tested after each epoch for 7 epochs for 5 models, each with different dropout percentages. The results are shown in figure y.

It would seem that increasing the number of epochs did mitigate some negative correlation, but it is also evident that not much positive change in validation accuracy can be made through applying a dropout layer; There is only a 0.86% improvement attained by applying a 30% dropout compared to 0%. Nonetheless, this came at no cost in terms of time taken to train and test. It can also be observed that a 40% dropout is too high for this model, as it resulted in a decrease of validation accuracy for at epoch.

**Testing**

* **Unit & Integration Testing**
  + Jest for frontend
  + Wscat for WebSocket API
  + Postman for REST API & recommendation system + inspecting lambda function logs
* **Integration Testing**
  + How frontend interfaces with backend
  + Vote-weighting was faulty and remained at 1
* **System Testing**
  + XCode reveals that some methods may only work for specific iOS version.
* **User-Acceptance Testing**
  + Below
  + Unclear how to initiate lobby track
  + Could not play same track twice in a row.

**Unit and Integration Testing**

**Frontend**

The frontend and it’s internal integration between the native bridging module and the react-native components was tested using the Jest package [reference]. Throughout development, Jest was used to continuously snapshot-test the UI, ensuring the underlying tree structure of each UI component did not change unexpectedly. Jest could not be used to unit test the exported macros or internal methods of the native bridging module, because the authorisation process is dependent on user interaction. Therefore, the preliminary UI was manually used to test each of these, which is part of what made the development for the native bridging module so unexpectedly long. Both the problems described in section x were discovered during this manual testing procedure.

**Backend**

The REST API was first unit tested using Postman [reference], where each endpoint would be invoked and the responses monitored for a variety of input cases, focusing mainly on correctness instead of security or latency. If the endpoint gave an erroneous response the lambda function would be inspected alongside the corresponding logs. Through this inspection procedure it was possible to detect and mitigate the problem where any database interaction wasn’t returning a promise, as mentioned in section x.

The WebSocket API was tested using wscat [reference] which allows WebSocket connections to be established and messages sent to various routes over the command-line. Similarly to the REST API testing procedure, each route was tested individually and checked for correctness. This involved creating “dummy” lobbies using the DynamoDB UI (so that they were not deleted when the host closed the application), and attempting to send messages from different dummy users in that lobby. Any incorrect responses meant that the lambda function code was inspected and logs checked too. In some cases it was revealed that the lambda function was attempting to send responses to users who were not in the lobby, resulting in a HTTP 410 error. Consequently, the exception handling demonstrated in figure x was written to remove the connection information of any user for which the error is thrown.

The integration between the frontend and the backend was tested using the preliminary UI to invoke react native fetch and ws requests, for the REST and WebSocket APIs respectively. In doing so, it was possible to test the entire user-lifecycle by building the application to a physical iPhone and running an iOS emulator on the development machine, thus creating two distinct users. At this point, the recommendation system had not been developed and the endpoint simply returned the tracks that made up the base playlist, so that the voting functionality could be tested. The frontend integrated seamlessly with the AWS-hosted backend APIs, seemingly accommodating all valid user-lifecycles.

**Recommendation System**

Once the final models had been trained and tested, example predictions were performed to essentially unit-test the models. Notably, the genre inference model performed well when any artist names were used as part of the prediction input, and also did particularly well in detecting language. For example, “Beethoven only” would result in classical music being the most probable genre, and “Classique” would result in French being the most probable genre (with classical the second-most probable). Clearly, the model was functioning in the way intended. The artist classification model was given numerous example inputs and almost always output artists that were similar to those which were included in the input. It was also notable that artists placed at the top of the input matrix had less importance than those at the bottom, as intended (see x). These example predictions were backed-up by the validation accuracy, which is much more telling of the accuracy of the model than example predictions. The example predictions mainly serve to test how to take the model inputs, format and encode them, input them, retrieve the output, and finally decode them.

Once the example predictions had been performed it was possible to use the same input and outputting process in the lambda functions that interfaced between the /get\\_recommendations endpoint and the hosted models themselves. Once these lambda functions had been implemented the integration between the REST API and the hosted models was tested using Postman. The latency for each invocation was massive relative to regular HTTP requests, 8 seconds for genre classification and 15 for artist classification. Nonetheless, they functioned as expected and could be now invoked from within the /get\\_recommendations lambda function. Thus, the full /get\\_recommendations lambda function was implemented and the full user-lifecycle could be tested with a multitude of users which is described in section x.

**User-Acceptance**

The project application was built on to the variety of iOS devices of 4 family members, who were then given the task of creating a lobby and controlling the music playback as a group. No specific instructions on how to do so were given in the hope that the UI would be familiar, and it would therefore be clear. The author was observing the group’s interaction with the application throughout, taking notes on any difficulties they may face. Once they had concluded their usage of the application, they would report any feedback to the author individually and anonymously.

The group quickly and successfully established a host and created a lobby. The lobby was named “Quarantine Party” by the host, leading to predictions of pop, dance and pump-up music (in that order of probability) by the genre classification model. This was discovered through inspection of the lambda function logs after the test had concluded. Conceivably, the word “party” played more of a role than “quarantine” in this prediction. The host user proceeded by playing one of the recommended tracks, and it was discovered afterwards that this track was not part of the base playlist and was thus inferred by the recommendation system. After which, users began to join the lobby using a combination of the QR code and the lobby key. All users voted for their favoured next track and at the stage where votes are counted and the next track deduced, there was a 3:1 majority. The group as a whole gave neutral feedback to the first track of the lobby, meaning the host user’s vote-weight remained 1. With no more users joining the lobby, the number of votes for each track paints an accurate description of the feedback that will entail for such track. As was the case, the second track of the lobby attained a 3:1 positive to negative feedback ratio, meaning those users who voted for the track now had a 1.5 vote-weighting. More importantly, the genre classification model was able to give this track preference over the previous one due to the exploitation of the vanishing gradient problem so that more accurate predictions could be made as to what artist the users would be most likely to listen to next. The next set of recommendations generated contained tracks belonging to the same artist as the well-received track, along with a similar artist to the neutrally-received track. The group continued this cycle of voting and giving feedback for tracks for 3 or so more tracks. At one point, a user closed the application as the playback switched from one track to another, and when they attempted to load the application it crashed. Short after this, the track which was elected was the same track as had just previously been elected, and the UI, specifically the progress bar, demonstrated undefined behaviour as a result.

From the observations of the author, there are plenty of positives to take away from this test. In particular that the recommendation system and user vote-weightings seemed to work well, and users seemed to require no guidance using the UI. Nonetheless, there were problems encountered that were unexpected, such as the application crashing when the user closed it as the lobby state changed, and that the UI responded negatively to the same track being played twice. The individual anonymous feedback reiterated the problems that the author was able to observe, but also included a criticism that it was unclear what will happen when the host user has just created a lobby and selects a track from those recommended, as required.

The problem where what the host user’s initial selection entails is a UI problem, since there are no prompts and it is identical to the voting interface. To distinguish the two, a modal was added to the UI upon creation of the lobby that is demonstrated by figure x. Included in this modal is a clear prompt telling the user what their action means; By selecting a track, it will be played as the first track of the lobby.

The UI is also responsible for the problem of the progress bar failing to function expectedly when the same track is played in succession, which is a perfectly valid instance of the applications lifecycle. To detect when a new track has begun playing, the progress bar takes in the lobby state as a property, and once this property changes, it is checked to see if the active track is the same as the one before. If it is not, then it is assumed that a new track has begun playing. This is a valid assumption. However, the assumption that if the lobby state is updated and the same track is playing then a new track can’t have begun is false. If the same track is to be played twice or more in a row, then the lobby state will update as a consequence of the “next” WebSocket message, but the active track property will not have. Thus, the progress bar will not reset. To correct this, a counter was passed as a property to the progress bar, representing the number of “next” messages the user device has received. If this number changes then a new track must have begun playback within the lobby and so the progress bar resets, even if it is the same track as before.

Unfortunately, in the interest of time, the problem where users cannot close the application during a lobby state change without the application crashing was not fixed as at first glance it is a much more complex problem than the other two. Such a fix will be included on the list of future work.

**Evaluation**

Citing tests, how many of the requirements does the application satisfy?

Deploying Models

* Save the configuration of the model in a JSON file, contains information on model such as activation functions, layer types, etc.
  + Version conflicts when deploying model as model was trained on a machine with python 3 yet AWS environment only supports python 2.7.
* Save the weights of the trained model to a h5 file.
* Then use the guide [reference] to export these 2 files into a model in protobuf format [reference].
* Transfer the saved model to an amazon s3 instance, then create an AWS SageMaker endpoint.
* Since the encoding of the inputs and outputs needs to be done for both artist and genre inference, lambda functions are used to handle this. For example, the genre inference lambda function will tokenise the lobby name and then invoke the endpoint using the result as the input. The output from the model will be in the form of a one-hot-encoded representation of a genre, this will need to be decoded using an inverse-map established at the point of training the model. The decoded genre seed can then be returned by the lambda function. In effect, the lambda function acts as the interface between any inference request and the SageMaker runtime.
  + To implement this for the genre inference model, the tokeniser needs to be loaded inside the lambda function. The keras tokeniser object used when training and testing the model was serialised using the Python pickle package [reference]. Once the pickle and keras-preprocessing packages are installed on to the lambda function, the lambda function was able to load and use the tokeniser. While this is largely inefficient and can take up to 8 seconds to do, it is much more efficient that instantiating a new tokeniser by passing in a dataframe of the train/test dataset.
  + For the artist inference model, the matrix of track features needs no altering and can be fed directly in to the model, but the vector of spotify artist ids needs encoding and decoding to and from OHE format. Therefore, only the OHE map and its inverse are required to interface interaction with the model runtime. However, both maps are approximately 80000 entries large and consequently, the time taken to load the serialised python dictionary is substantial. Fortunately, this is not much of a concern since the entire recommendations process has 25\% of the duration of the currently playing track to determine the set recommendations. The average track is approximately 3 minutes long [reference], meaning the entire recommendations process has approximately 45 seconds on average which is more than enough.

Once the models had been trained and tested, their configurations and parameter weights were saved into JSON and h5 files respectively. The JSON file contains data regarding the model’s configuration, including information on the layer types and activation functions thereof. The h5 file consists of model bias and parameter weight data. The combination of these files allows models to be deployed to a large variety of platforms, such as AWS SageMaker [reference]. Using the AWS guide on exporting a pre-trained model to SageMaker [reference], the models were first exported to ProtoBuf or “SavedModel” format [reference], using both the JSON and h5 files. The ProtoBuf format is essentially a serialisation of the models themselves. The ProtoBuf models were uploaded to an S3 instance which allows SageMaker to create endpoints from them. The endpoints simply invoked the model runtime with the passed input and returned the prediction as calculated by the models. No processing of inputs was made at any point by these endpoints, it is expected that any work that needs to be done on the inputs has already been done, and the outputs are precisely that of the model. Consequently, 2 more lambda functions were created to accommodate the handling of inputs and outputs for each model, these lambda functions thus served as interfaces between inference requests and the model runtimes.

To implement this for the genre inference model, the tokeniser needs to be loaded inside the lambda function. The keras tokeniser object used when training and testing the model was serialised using the Python pickle package [reference]. Once the pickle and keras-preprocessing packages are installed on to the lambda function, the lambda function was able to load and use the tokeniser. While this is largely inefficient and can take up to 8 seconds, it is much more efficient that instantiating a new tokeniser by passing in a dataframe of the train/test dataset. The inverse OHE map was used to decode a one-hot-encoded genre seed into a plaintext representation that is usable by the get\\_recommendations lambda function. Since there are only 102 classes (genres), the actual python dictionary could be copy/pasted from the local training machine to the lambda function, drastically reducing overhead that would arise from serialisation.

For the artist inference model, the matrix of track features needs no altering and can be fed directly in to the model, but the vector of spotify artist ids needs encoding and decoding to and from OHE format. Therefore, only the OHE map and its inverse are required to interface interaction with the model runtime. However, both maps are approximately 80000 entries large and consequently, the time taken to load the serialised python dictionary is substantial. Fortunately, this is not much of a concern since the entire recommendations process has 25\% of the duration of the currently playing track to determine the set of recommendations (see chapter [chapter of lifecycle]). The average track is approximately 3 minutes long [reference], meaning the entire recommendations process has approximately 45 seconds on average, which is more than enough.

**WebSocket API**

1. The WebSocket API routes that were designed in chapter x proved to be sufficient, and as such precisely those routes were implemented, and no high-level amendment needed to be made.
2. When a user enters a lobby, either the InLobby or HostLobby UI components will be pushed on to the navigation stack, and one they have mounted, the life-cycle methoed componentDidMount() is called. In the body of this method, a connection is initiated with the WebSocket API using the $connect route. On the client-side, the connection is bound to a WebSocket object ws, and event listeners and handlers can be created, which enable the client to detect incoming messages from the API.
3. When an incoming message comes from the API, the client-side will check a simple flag to determine if the message is regarding voting state or that the next track has begun playback.

A screenshot of a cell phone

Description automatically generated

Figure x shows the architecture of the next route, whereby next track has begun playing on the host device, so it will notify the WebSocket API utilising the next route. Upon receipt of this message, the API will invoke the “next” lambda function, which will identify all users in the same lobby as the host device, obtain their connection ids, then send a “next” message through all of these connections. The message will invoke the event handler as part of the InLobby.js component that manages incoming WebSocket messages, and prompt devices to send a /get\\_next\\_track request to the REST API, and thus acquire details of the track that has recently begun playback on the host device.

A screenshot of text

Description automatically generated

The “vote” route takes the same architecture as the “next” route, except that any device can send a message through the vote route, it is not exclusive to the host device. Figure x shows a snippet of code which is part of the vote lambda function. This code takes a list of users connected to the API which are part of the same lobby as the user who voted, then sends an updated set of votes according to the database to each user apart from the user who voted. The function doesn’t send the vote data back to the user who invoked it, because lambda functions can at times have relatively high latency, and it would be perceived by the user that their vote is taking a lot of time to register with the system. Instead, when a user votes, a client-side variable is assigned the index of the recommendation which was voted for, so that the vote number for that recommendation can be incremented by the user’s vote weight almost instantly. Then, if said user were to receive another message from the API that a different user had voted for another track, the client-side component would disregard the increment made by itself. Instead, it would use the full set of votes as given by the message since it’s a more up-to-date set of votes.

**Final User Interface**

The purpose of the final UI is to satisfy those requirements that the preliminary UI does not. Specifically, the final UI will accommodate the ability to join a lobby through the means of a QR code (requirement 1a), and it will consist of components that are both familiar to users and consistent across the entire UI (requirement 3h).

**QR Code Capability**

**Familiarity and Consistency**

1. Buttons & Text Inputs (Familiar to user and consistent across UI)
   1. Buttons and text inputs are consistent across the site in an attempt to reduce any confusion
2. Playback ()
   1. Track Name and Artist
   2. Playlist Name/Lobby Name

User Interaction

User familiarity is important for components that can be interacted with, so that users can quickly deduce the purpose of these components based on previous experiences with other applications, thus reducing confusion. The main components that users can interact with in the developed application are text inputs and touchables (buttons). These components were redesigned to mimic a design popular in the Spotify application as shown in figures x and y.

Plaintext Lobby Name

Tokeniser

Embedding Layer

Convolutional Layer

Regularisation

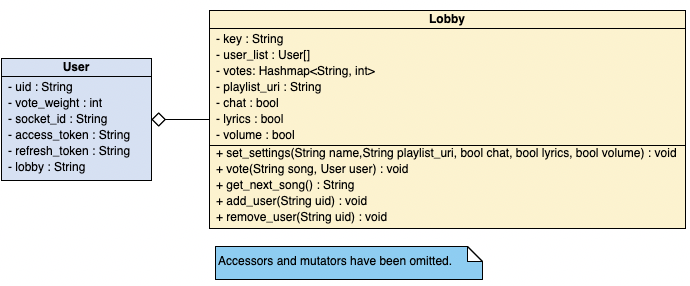
Output Layer

Decoder

Plaintext Genre

**Requirements**

|  |  |  |
| --- | --- | --- |
| **Number** | **Description** | **Importance** |
| **1A** | **The host user will be able to create a lobby and invite users to this lobby through a key or barcode** | **5** |
| **1B** | **The host user will be able to alter the following lobby settings before or after creation:**   1. **Lobby Name** 2. **Base Playlist** 3. **Maximum Users** 4. **Type of Lobby (gym, party, car, etc.)** 5. **Chat Room (enabled/disabled)** 6. **Volume Control (enabled/disabled)** 7. **Lyrics (enabled/disabled)** | **4** |
| **1C** | **The host user will be able to terminate their lobby at any time** | **5** |
| **1D** | **The host user will be able to authenticate their account with the Spotify Accounts Services** | **5** |
| **2A** | **Users will be able to join a lobby if they enter the correct key (or barcode) given that the lobby is not full** | **5** |
| **2B** | **Users will not be able to join a lobby that is full** | **3** |
| **2C** | **Users can only be in one lobby at a time** | **3** |
| **2D** | **Users can leave a lobby at any time** | **5** |
| **2E** | **Users can choose to show or hide the chat room if enabled** | **2** |
| **2F** | **Users can choose to show or hide the track lyrics if enabled** | **2** |
| **2G** | **Users can thumbs-up or thumbs-down the current track** | **4** |
| **2H** | **Each user has 1 vote for the next track (however, their vote may be worth more or less than 1)** | **5** |
| **2I** | **A users vote weight depends on the number of times they have received a significant number of thumbs-up or thumbs-down for a suggestion** | **4** |
| **2J** | **A user cannot obtain a vote weight of 0** | **5** |
| **2K** | **If the volume control is enabled, users can vote to turn the volume up or down** | **2** |
| **2L** | **The user can save any track being played to their Spotify library** | **1** |
| **2M** | **Users on any iOS or Android Device can download the application from their respective app store.** | **4** |
| **3A** | **The backend must run efficiently in order to be scalable, and therefore multithreading may need to be implemented** | **3** |
| **3B** | **The application user interface will be responsive in that it will function as expected on a wide range of devices regardless of size or aspect ratio.** | **2** |
| **3C** | **The lyrics displayed on-screen will be synchronised with the audio be played at the time** | **3** |
| **3D** | **Develop and use a test suite to determine the reliability and overall functionality of the application** | **5** |
| **3E** | **The recommendation system will allow the lobby to get smart recommendations on the next tracks to be played and will be used as a tie-breaker when two songs have an equal number of votes** | **5** |
| **3F** | **The recommendation system will use the thumbs-up/thumbs-down data as feedback on it’s own recommendations so that it learns as the application is being used** | **3** |
| **3G** | **The recommendation system will be hosted on an external application server** | **3** |



**A screenshot of a cell phone

Description automatically generated**

Figure x = inlobby\_sequence

Figure y = hostlobby\_sequence

In the interest of clarity, the sequence diagrams shown as figures x and y have abstracted certain procedures that play a part in the processes demonstrated. The Device lanes encompass that of the user, client application, and native bridging module. Any trivial database or Spotify Web API interactions such as the persistence of voting data or querying user playlists have been omitted from the “Backend API” lane. Moreover, neither sequence diagram denotes the entire domain of functionality available to the subject device in the context of the application, only a subset. The sequence diagrams represent example use-cases; The process of generating artist inferences is omitted since it is very similar to the inference process in figure x.

Figure x demonstrates a sequence diagram for an example non-host user who joins a lobby, gives feedback on the currently playing track in the lobby (thumbs up/down), then votes on the next track to be played. 3 other users in the lobby vote while this user is in the lobby, and at the point where the track concludes, the device receives a “next” message from the WebSocket API (invoked by the host device). Once received, the currently playing track information will update, and the votes, thumbs status, and progress bar will reset to their initial and default state.

Figure y is more complex as it shows an example use-case for a host-user, who creates a lobby. The beginning of the diagram shows the user proceeding through the authorisation process; Authorising the application to use their account, then connecting the application to the Spotify app. Once this is completed, the user can choose the base playlist for their lobby, through the invocation of the /get\_playlists endpoint which queries the Spotify Web API (not shown). Once the user confirms the lobby settings, they create the lobby. At this point, any user can join the lobby and begin voting for tracks. Figure y demonstrates a vote coming in from another user, which becomes the most voted-for track and is thus elected. After the currently playing track is finished, the elected track will play and the host-device will alert the backend API via both the /set\_track endpoint and a WebSocket “next” message. The /set\_track endpoint is for those users who are yet to join the lobby, and the “next” message is to alert those who have already joined.

**Implementation**

**Overview**

This chapter contains sections which describe the implementation process of each fundamental component within the application, including any unexpected problems that arose and the tackling thereof. There were 10 sprint cycles of varying length, starting from 05-Aug-19 and concluding on 09-Mar-20, as shown in table \ref{tab:implementation\_timeline}.

**Adapted requirements**

At the point of the progress report, it was assessed by the author that the time constraint that was on the project at that time was too large considering the remaining development that needed to be completed. In part this was due to the implementation of the native bridging module taking much longer than expected, as shown in table x. Consequently, in alignment with the project planning, the contingency plan described in section x was activated and 3 of the lowest importance requirements were removed from the project backlog in order to ease the pressure of the time constraint. This was done in the hope that the amount of time gained was sufficient so that complete development of higher importance requirements was more likely. Requirements 2e, 2l, and 3c were the 3 lowest importance requirements and their removal meant that the chat room would no longer be implemented, and nor would the application be developed for deployment on android devices.

As described in section x, requirements x and y were adapted so that instead of lyrics being displayed and synchronised for each track, the lyrics were replaced with track information such as artist background information or the motivation behind the creation of the track. The adaptation follows the discovery of legal issues that would possibly be encountered by displaying the lyrics for each track, and there are no legal issues with displaying the track information, which is just as available.

**Evaluation**

Satisfaction of requirements

1. Comparison of requirements in sections of user, host misc, etc.
2. Total percentage of satisfied requirements
3. Weighted on importance

Project Management Evaluation

1. Risk management protocols functioned effectively
2. Dynamic specification was definitely the right choice, since requirements were amended along the way

Legal, Social and Ethical evaluation

1. Only legal issue that was encountered was that lyrics could not be displayed publicly
2. Spotify haven’t revoked right for author to use their API

**Satisfaction of Requirements**

**Project Management asfsad**

Recommendation System

Spotify Web API

Spotify SDK

Backend API

Client Application

Native Bridging Module