Smoothie - Using Machine Learning to Recommend New Music at the Intersection of Multiple Listener's Music Tastes.

by Kai Usher

Student ID: B526637

Loughborough University

18COC257: Computer Science AI Project

Supervisor: Andrea Soltoggio

SUBMITTED 30th APRIL 2019

18,070 words

Abstract

With tens of millions of songs available for consumption, the ability to discover new music to listen to can often be a daunting process. Having to choose music that two people with unique music tastes will enjoy listening to is exponentially more difficult.

The following report outlines my approach and methodology to developing a system which implements a machine learning algorithm to create a playlist which will satisfy the music tastes of more than one person.

The result of this work is a fully-functional website which makes use of the Spotify API to authenticate a user and create a new playlist based on two already existing input playlists. Using a Python script, a neural network is trained to understand the key components of each playlist owner's music taste, and using this information adds tracks which will appeal to both owners to a new playlist.

The finished system is able to produce unique playlists which accurately reflect the music tastes present in the playlists it is trained upon, therefore providing a novel way to discover new music when in the company of others.

Acknowledgements

I'd like to say a special thank you to Dr. Andrea Soltoggio for his advice and openness to letting me complete this project under my own command, and my friends and family for their understanding whilst I was hiding away to ensure I completed this work to the best of my ability.

Contents

1 Introduction **6**
1.1 Motivation 6
1.2 Project Aim 7
1.2.1 Project Objectives 7

2 Literature Review **8**
2.1 Multiple Music Tastes 8
2.2 Recommender Systems 8
2.3 Existing Systems 11

3 System Design **14**
3.1 System Diagram 14
3.2 Development 14
3.2.1 Website 14
3.2.2 Data Processing and Machine Learning 16
3.3 Data 16
3.4 Machine Learning 17
3.5 Visual Identity and Branding 19
3.6 UX/UI Design 20

4 Plan **21**
4.1 Requirements 21
4.1.1 User Stories 22
4.2 Software Engineering Approach 23
4.2.1 Methodology 23
4.2.2 Project Management 23
4.2.3 Version Control 23
4.3 Gantt Chart 24
4.4 Risks 24
4.5 Testing 25

5 Implementation **27**
5.1 Development Environment 27
5.1.1 Website Component 27
5.1.2 Machine Learning Component 28

5.2 Data Collection 29
5.2.1 Website Component 29
5.2.2 Machine Learning Component 33
5.3 Website 36
5.3.1 Front-End 36
5.3.2 Back-End 46
5.4 Machine Learning Implementation 46
5.5 Recommender System 52
5.6 Automated Testing 57

6 Results **58**
6.1 Requirements 58
6.1.1 Authentication 58
6.1.2 Playlist Input 60
6.1.3 Playlist Options 62
6.1.4 Machine Learning 62
6.1.5 Prediction 64
6.1.6 Recommender System 67
6.1.7 Playlist Creation 72
6.2 Project Objectives 72

7 Discussion **76**
7.1 Personal Development 76
7.2 Project Management 76
7.3 Future Improvements 77
7.3.1 Automated Testing 77
7.3.2 Recommender System 77
7.3.3 Overall Website Functionality 80
7.3.4 Mobile 81
7.3.5 User Research 81

8 Conclusion **82**

1 Introduction

1.1 Motivation

On November 1st 2018, Spotify released their Q3 2018 results to shareholders in a publicly available press release [1]. The figures in the report show that, by the end of September 2018, Spotify had 191 million Monthly Active Users ('MAUs') with just over 45% of these MAUs being paid premium subscribers. In an article published by Forbes on May 15th 2018, Spotify's position as a market leader in the music streaming industry was justified with statistics showing that, at the time of writing, Spotify had 25 million more users than its closest competitor; Apple Music reported 50 million users vs Spotify's 75 million [2]. With such a vast user base and consequently a huge amount of data (which is accessible through their APIs [3]) Spotify is an insightful centrepiece for a machine learning project.

Spotify users have created over 3 billion playlists from a catalogue of over 40 million tracks [4]. Due to the individuality of music taste and the sheer number of tracks a user has access to, it is undeniable that a user's playlists are an interesting insight into their personality and mood at any given time. This also means that it is a near impossible task to create a playlist for more than one person that will perfectly appeal to all listeners in question.

The Spotify API can be used in order to retrieve the numerical classification of a given track by extracting audio features such as 'danceability' and 'valence' (the happiness of a track) [5]. As you can do this for any track on Spotify, it is also possible to scale this data gathering to every track in a playlist, average the values, and get a value for the exact 'instrumentalness', for example, of the playlist in question.

By taking such an approach, machine learning can be used to train a network on a given user's music taste to the point where it is possible for the network to predict whether a user would like a random track or not.

The result of this project will determine whether it is possible to use machine learning to create a new playlist that will near-perfectly appeal to multiple users, based on the information derived from existing playlists which are given to the system.

1.2 Project Aim

The main high-level aim of this project is to develop a web-based application in which a user can log into using their Spotify account, provide links to two existing playlists, and as a result get back a new playlist of mostly new tracks (ones which did not exist in either playlists) that fit the music tastes of both owners of the input playlists.

1.2.1 Project Objectives

1. Conduct research on the subject area and compile findings into a literature review.
2. Conduct further research into what related solutions are already available and compare how they differ from this project.
3. Design a web interface in which a user can log in, provide two playlists and view the playlist which is generated by the system.
4. Develop the web interface defined in the previous objective.
5. Develop a system using the Spotify API and machine learning which takes the playlists provided, gathers all of the required data from the playlists, trains a neural network and creates a new playlist using a classifier and a recommender system.
6. Comprehensibly test the system to ensure it is bug-free and fully functional.
7. Write a report on the result of the project and evaluate whether the aim has been met when judged against the criteria of the objectives.

2 Literature Review

2.1 Multiple Music Tastes

McCarthy's paper outlining his work with a group preference agent via the MusicFX system discusses how music selection can be done when the choice has to appeal to multiple listeners [6].

The group preference agent embodied in MusicFX works by using a preference database - each member of the fitness centre, the environment the system is deployed in, must specify their preference for a number of musical genres. A formula is then used to calculate the genres with the highest preference of the members currently in the environment, and that genre is chosen to be played.

This research provides valuable insight into predicting music taste for more than one listener, however differs from the system proposed in this project mostly due to the fact the users of MusicFX have to manually submit their preferences. Aspects from McCarthy's work should be considered, however, in the context of an artificial intelligence system that is able to predict a user's music preference, and thus remove the need for the manual entry preference database. Furthermore, the focus of this project will be on individual tracks culminating in a playlist, and not simply a single genre as seen in MusicFX.

2.2 Recommender Systems

Recommender systems are software tools and techniques providing suggestions for items to be of use to a user [7].

Whilst the research and study of recommender systems is still in its relative infancy (in comparison to more classical information systems such as databases), interest over recent years has rapidly increased. Recommender systems are imperative to the success of nearly all high-traffic websites such as Amazon, Spotify, Netflix and YouTube. One of the more notable events in the history of recommender systems, and one which emphasises the importance of the technology, was when Netflix awarded a $1,000,000 prize to the first team that substantially improved the performance of their existing recommender system [8].

There are six main types of recommender systems, as defined by Burke [9], which are recognised as standard - Content-based, collaborative filtering, demographic, knowledge-based, utility-based and hybrid. Each technique is further explained below.

- **Content-based** - The system recommends items that are similarly featured to items the user has liked in the past.
- **Collaborative filtering** - The system recommends items that users with similar tastes as the active user have liked in the past.
- **Demographic** - The system categorises the active user based on personal attributes and recommends items based on demographic classes.
- **Utility-based** - The system recommends items based on a computation of the utility of each item to user.
- **Knowledge-based** - The system recommends items based on domain knowledge related to how that item will meet the active user's needs and preferences.
- **Hybrid** - The recommender system is a combination of the above techniques; the advantages of one technique are used to mitigate the disadvantages of another.

Recommender systems in relation to music recommendation is a topic which is covered by Schedl et. al. in the Music Recommender Systems chapter of the Recommender Systems Handbook [10]. Below are a few of the key points which may be relevant and interesting in regard to this project.

Recommending music has a number of unique characteristics which are not present in other domains where recommender systems are commonly used (such as movies or books). The time it takes a user to consume a single item of music (approximately three and a half minutes [11]) is significantly shorter than the time it takes for them to consume a book (potentially weeks) or a movie (one to a few hours). A result of this shorter consumption time is that the user is likely to form an opinion on music much faster than other

media domain items. Another differing factor is that where a movie or a book is typically only consumed once, especially in a short period of time, music items may be consumed repeatedly; user's are therefore likely to be appreciative of recommendations of already discovered items.

As stated by Masthoff, recommending to groups is notably more complex than recommending to an individual, however user models can be aggregated effectively to produce successful group recommender systems [12]. Various aggregation methods which are inspired by Social Choice Theory [13] are tested, with the conclusion that the strategies which perform best are Average and Average without Misery. Masthoff also points out how an individual's ratings for a particular item may differ depending on the group they are in, and that modelling affective state is an important factor when developing a group recommender system [14]. This raises the interesting question of what external factors should be considered when recommending music to more than one person; is it enough to simply combine the basic recommendations for two people or should factors such as the relationship between the people be considered? The research also touches on the idea of sequencing the items a recommender system produces. Overall satisfaction of all users can depend on what order the items recommended are consumed, and that to maximise optimal satisfaction the output should have 'a good narrative flow', 'mood consistency' and 'a strong ending'. An example of a music program is given with the comment that the system should consider mood and rhythm, for example, when ordering tracks, but additional information about each tracks content will be needed [12].

Van den Oord et. al. discuss how recommending music is a particularly difficult challenge due to the fact that a large proportion of recommender systems tend to use collaborative filtering, but this method suffers from the 'cold start' problem; new items that have not been consumed before cannot be recommended [15]. This also means that items which appeal to only a niche audience are difficult to recommend because usage data is noticeably more scarce. Whilst content-based recommender systems solve these issues, they can also lead to predictable recommendations if not used with good enough meta-data - recommending tracks by an artist the user is known to enjoy is not very valuable.

A proposed, and developed, solution by Van den Oord et. al. is the use of

deep convolutional neural networks to predict latent factors from audio when they cannot be obtained from usage data. They discovered that even though many characteristics related to listener preference cannot be predicted via audio signals, the system still produced 'sensible' recommendations, particularly when dealing with new and largely unpopular music.

The work by Van den Oord et. al. provides an interesting technique which may be relevant to the system proposed in this report. One notable difference is that using the data available through the Spotify API, the proposed system will have access to the characteristics that form a user's music taste, and therefore applying this information to a similar system as discussed above could enhance the ability to recommend a varied range of music.

2.3 Existing Systems

Built into Spotify as standard are 'Collaborative playlists'. Collaborative playlists allow multiple users to work together to build (add tracks, delete tracks, etc.) a single playlist [16]. This collaboration is very rudimentary compared to that which I am proposing to build in this project. Collaborative playlists require the users collaborating to add each track themselves and it is down to each user to determine whether the other user in question will enjoy the track they are adding. On the contrary, in this proposed system, neither user will have to manually add any tracks, and the tracks that are added will be the outcome of a recommender system that uses machine learning to satisfy both users.

The Developer Showcase on the Spotify for Developers website shows, as described by Spotify, "outstanding apps, all built using our APIs, SDKs and other developer tools."[17]. I went through each entry in the showcase to determine if any were related to my proposed project and my findings were as follows.

Noon Pacific developed an app and website which contains 'curated mixtapes of the best new music handpicked from LA, New York and London' which a user can listen to via their Spotify account [18]. This website and my proposal share the core principle of creating Spotify playlists, but there is no personalization of the content in the playlist, and no use of machine learning or recommender systems in Noon Pacific.

Playlist Souffle, developed by Zach Hammer, offers a more personal ap-

proach by working with a user's existing playlists; the user logs in to Spotify, chooses a playlist from their library through the website interface, and the system will 'souffle up the playlist' by swapping each track for a different track by the same artist [19]. Whilst there is a more personalised approach, such as that which will be at the foundation of my proposal, being used by Playlist Souffle, the lack of machine learning or multiple user collaboration separates it from any potential crossover with my final product.

Musical Data is a web app developed by Rutger Ruizendaal which 'allows users to quickly gather and visualise data of their favourite tracks, albums and artists' [20]. This project relates to my own as it focuses on using the audio features of a track to provide an otherwise unavailable insight into a user's music (tracks are found using a search bar and not the user's music library). However, this web app is built purely for exploration purposes - the user cannot interact with their own music or create playlists/save tracks. I intend to use the data shown by this website 'behind-the-scenes' in my project with the purpose of training the neural network.

A similar project is Klangspektrum - a thesis project by Michael Schwarz - which enables Spotify users the ability to analyse their musical behaviour through data visualisation of track audio features [21]. Much like Ruizendaal's Musical Data and my proposal, Klangspektrum uses audio features of a track as the fundamental data in the system, however, the project doesn't relate to the creation of playlists or collaboration between more than one listener.

Darrell Hanley has developed a web app called Dubolt where a user can 'discover new music with the help of old favourites' [22]. By logging in using the Spotify API, a user can search by artist or track and the website will recommend related tracks (using Spotify's audio features) which the user is likely to enjoy. Following this, the user can save the tracks they like to a playlist. Dubolt is the web app which shares the most similarities with my proposal, however there are a number of things which still separate the two. There is no multi-user/different music tastes aspect to Dubolt, which is the core premise of my proposal, and also the playlist must be handmade by the user, whereas I intend to automatically create the playlist using machine learning.

Magic Playlist, developed by Joel Lovera, is a web app which intelligently

generates playlists based on a given track [23]. This project relates to my proposal on the premise that the user can generate a playlist with minimal input. Magic Playlist bases the entire set of playlist tracks off of the artists detected in the source track, not the user's unique music taste as I propose. Furthermore, there is no collaborative aspect to this project.

C-Listening Room, a website by José Manuel Pérez, is the only item in the Developer Showcase which involves multiple users - the website allows multiple users to create a collaborative Spotify queue by searching and adding tracks through the online interface [24]. Although the website shares the idea of collaboration with my proposal, there is no functionality for playlist creation or generation of suggestions using machine learning; the users adding to the queue in C-Listening Room are doing so based on their own music taste and there is no consideration whether the other users would like the added tracks.

To summarise my findings from the Spotify for Developers Developer Showcase, there are a wide range of projects which contain some aspect(s) of the functionality I intend to include in my project, but there is not one system which uses them all in harmony to achieve the same aim I have proposed. The most notable feature which is not prevalent in any of the above projects is the use of machine learning, the feature which will be at the core of my project, and thus separate it from the current existing solutions.

3 System Design

This section will outline the tools used to develop the system and explain why they were chosen over potential alternatives. The final two subsections will cover the visual design of the system, including user experience design and the creation of a suitable visual identity.

3.1 System Diagram

Figure 1 shows a high-level view of how the components of the system will interact, and what data will move between them.

3.2 Development

There are two main components to the system - the website component and the data processing and machine learning component.

3.2.1 Website

The front-end of the website is developed using React. React is a JavaScript library designed to enhance the creation of user interfaces for web applications [25].

The main advantage of using React is the Virtual DOM that enables the development of highly dynamic UIs with faster performance. As React is a JavaScript library, and not a framework such as Angular or Vue.js, the learning curve associated with it is much more manageable. This allows the time between starting the implementation of a feature and having working functionality to be significantly reduced.

In addition to the aforementioned advantages, since being published under an open-source license in 2013, the React GitHub repository has been starred 117,223 times and has 1,270 contributors (at the time of writing) [26]. These statistics show that there is a substantial community supporting the development of React and it has a well secured place in the web development environment.

The back-end server aspect of the website is developed using Node.js - an event-driven JavaScript runtime environment [27]. The decision to use

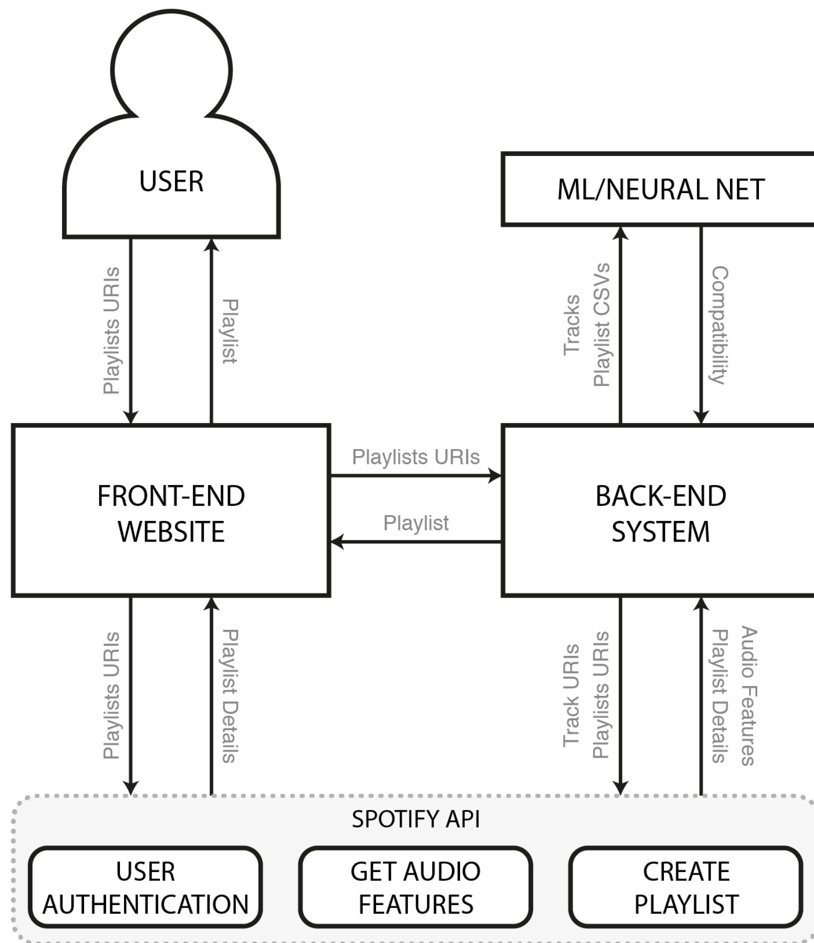

Figure 1: System diagram.

Node.js was mostly made on the basis that Spotify provide a Web API tutorial which uses Node.js to create a simple server-side application to interface with the Spotify API [28]. By using this tutorial and the associated example code for setting up OAuth for Authentication Flow, I was able to progress quickly with getting my development environment set-up and therefore focus on the functionality of the site and not the server/back-end.

3.2.2 Data Processing and Machine Learning

The data processing and machine learning component of the system is developed in Python. Python was chosen over other similar programming languages, most notably R, as it has the ability to perform a wide range of tasks across the whole software development suite and not just in-depth data science. Combining its versatility with an easy-to-learn syntax and plethora of useful libraries, Python was an easy choice to use for this project.

Once the data processing and machine learning algorithms are developed, they will need to be packaged in a way which is accessible for the website component of the system. Python doesn't have any native functionality for web development as such, so therefore I needed to use some sort of framework to expose my Python code as an API. After some research, I came across Flask; Flask is a Python microframework for web development [29]. Furthermore, Flask-RESTful is an extension for Flask that adds support for the quick building of REST APIs [30]. Using Flask-RESTful I will be able to easily transform my Python functionality into an API that can be utilised via simple API requests from the website.

3.3 Data

The Spotify API is the main source of data for the system. The Spotify API was chosen due to the company's status as a market leader, as discussed in Section 1.1, and the plentiful resources available online to support development using the API.

Whilst it is the Spotify API which handles the authentication of the user, the main focus is retrieving playlist data and audio features of tracks from the Spotify catalogue. As shown in Table 1, there are a total of thirteen audio features which make up every track on Spotify [5].

I will utilise spotify-web-api-js, a client-side JS wrapper for the Spotify Web API [31], when retrieving data for the front-end website portion of the system, and spotify-web-api-node, a Node.js wrapper for the Spotify Web API [32], when retrieving data for the back-end. The libraries provide a set of helper functions which cover the breadth of the Spotify API and make it more efficient and understandable to access the API endpoints.

For the machine learning portion of the system, the Spotipy library is used to retrieve the necessary information using the provided URIs. Spotipy is a lightweight Python library which provides full access to the entirety of the music data available on the Spotify platform [33].

Data is handled within the system using the pandas Python library; more specifically, the system uses DataFrames (2D, tabular data structures) when storing and working with the data retrieved from the Spotify API. As the data in question is only a few hundred songs and their respective details, and it is not required to persist the data, it is unnecessary to use an alternative storage option such as a relational database management system. The pandas library allows for fast handling of data, comes with a wide range of added utilities and due to the fact it is native to Python, works well in unison with other libraries used in the system [34].

No data is persistently stored within the system. Whilst there is an argument that this could add value to the UX and generate an increased number of return visitors (for example, a user could save and work with a generated playlist on the website itself) it is not essential to the proposed functionality and does not fit within the necessary time requirements.

3.4 Machine Learning

The machine learning portion of the system is implemented using the scikit-learn Python library - an open-source library which provides simple and efficient tools for machine learning and data mining [35].

I chose to use scikit-learn over other options, such as TensorFlow or PyTorch, simply due to its ease of use and high-level nature. TensorFlow, for example, is a much more low-level library designed for the purpose of building machine learning models, and therefore has a steeper learning curve which could be considered excessive for the implementation in question [36]. In comparison, scikit-learn provides a much higher-level solution with pre-existing

KEY	DESCRIPTION
duration_ms	The duration of the track in milliseconds.
key	The estimated overall key of the track.
mode	Mode indicates the modality (major or minor) of a track, the type of scale from which its melodic content is derived.
time_signature	An estimated overall time signature of a track. Time signature is a notational convention to specify beats in a bar.
acousticness	A confidence measure from 0.0 to 1.0 of whether the track is acoustic.
danceability	Danceability describes how suitable a track is for dancing based on a combination of musical elements including tempo, rhythm stability, beat strength, and overall regularity.
energy	Energy is a measure from 0.0 to 1.0 and represents a perceptual measure of intensity and activity.
instrumentalness	Predicts whether a track contains no vocals.
liveness	Detects the presence of an audience in the recording.
loudness	The overall loudness of a track in decibels (dB).
speechiness	Speechiness detects the presence of spoken words in a track.
valence	A measure from 0.0 to 1.0 describing the musical positiveness conveyed by a track. Tracks with high valence sound more positive.
tempo	The overall estimated tempo of a track in beats per minute (BPM).

Table 1: Table of the audio features of a Spotify track.

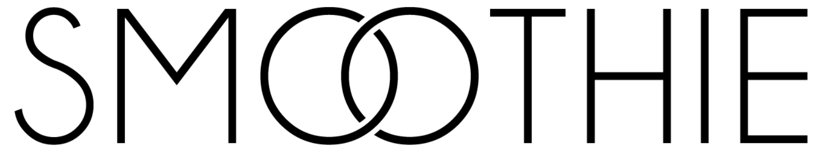

Figure 2: Smoothie logo.

implementations of machine learning algorithms.

3.5 Visual Identity and Branding

As I designed the system to be a complete product I wanted to design a visual identity which complements the concepts within and adds a level of marketability and authenticity to the website.

The name 'Smoothie' was used as it is both short and memorable, and allows for a fun metaphor to explain the system. The target audience for the system is anyone and everyone who has a Spotify account and wants to listen to music in a social situation. This means that not everyone who uses the system will want to try and understand, or even care about, the theory behind it. Therefore, explaining the system using the terms found in this report (machine learning, group recommender systems, intersections of satisfaction, etc.) was not a viable option. The core idea of the system is to take two playlists and blend them together into something new. 'Smoothie' allows the verb 'blend' to be obscured into a fun metaphor to walk the user through the process of generating a playlist without confusing them with technical jargon.

The logo for the system, as seen in Figure 2, was designed to reflect the intersection of two playlists or two music tastes.

This idea continues into the colour scheme chosen for the website. The overall design is kept simplistic to channel the user's focus on the journey of progressing through the required steps (more detail on this in the following section, UX/UI Design), so any use of colour had to be subtle as to not overpower any other elements in the interface. To achieve this I used the

background as my focal point for colour.

At it's core, the background is a gradient. I chose a gradient to further emphasise the notion of blending and deepen the smoothie metaphor which runs through the system. To make the background gradient less intrusive, the three colours (purple, blue and pink) are blended over an area much larger than the screen size and then this area is slowly moved through the user's viewport. The overall effect of this is that the background of the website is continually changing and blending the more time the user spends on the site. The effect is subtle enough to not distract and be a focus point, but effective enough to improve the user experience and make the website more enjoyable to use.

3.6 UX/UI Design

I wanted to design the system to be a genuinely enjoyable and engaging experience for the end users. In theory, the core functionality of the system could be stand-alone and use the command line for input and output, however, this is not what I wanted to achieve with the system. By wrapping the system in a well-designed website I was able to provide a start-to-finish experience that has a hugely positive impact on the user experience.

As mentioned previously, the user is walked through each step of the process of providing the required information and generating a playlist. Each step is stripped back to the bare-bones and presented in a way which is simplistic yet informative. The user isn't just presented with a form to complete, instead they are prompted through each input field one at a time, and each of the fields dynamically change as they provide the information. For example, when a playlist URI is pasted into the respective field a summary element is made visible which shows the playlist title, the playlist author, the playlist thumbnail image, the first three songs of the playlist and the playlist track count. This transition is demonstrated in Figure 3.

It's this immediate feedback that greatly enhances the UX without being obtrusive or even overtly noticeable. It's more common for a user to have to provide all of the information in a form, submit the form, and only then they are told something is wrong and they need to go back and change it. Using this approach, the user can instantly see if they have provided the wrong URI and alter it before progressing to the next step.

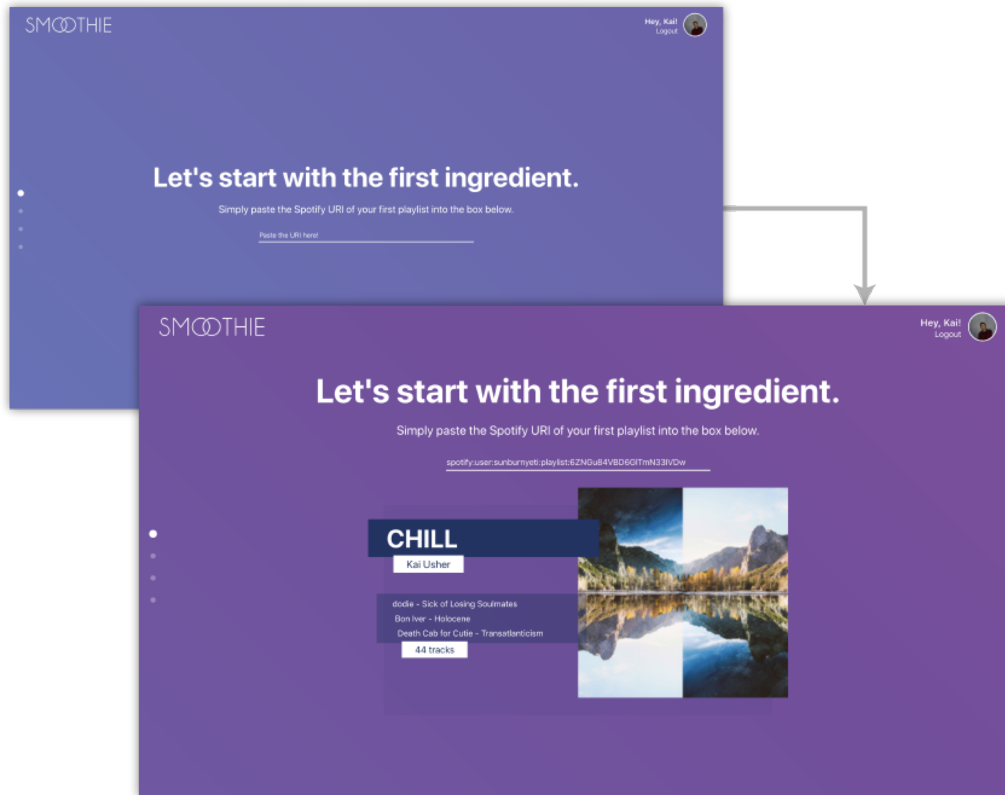

Figure 3: Playlist preview.

4 Plan

4.1 Requirements

The core requirements of the system are detailed below. For the system to be classed as a success, all of these requirements must be met.

- The system must be able to authenticate a user using the credentials of a pre-existing Spotify account.
- The system must accept two pre-existing Spotify playlist URIs and retrieve the necessary data related to them.
- The system must present the user with options to manage the playlist which will be generated.

- The system must use machine learning to train neural networks on the contents of each playlist - i.e. understand the playlist author's music taste.
- The system must be able to accurately predict if a track fits the playlist a network was trained upon.
- The system must utilise a recommender system to recommend tracks that fit into both playlist authors' music tastes.
- The system must generate a playlist of recommended tracks (with the previously input options) and save it to the authenticated user's Spotify account.

4.1.1 User Stories

In this section, the requirements are further explored to produce user stories. These user stories are written from the perspective of a generic user of the system and will form the basis of the components to be delivered in development sprints.

- As a user, I should be able to use my existing Spotify account to log in to the system.
- As a user, I should be able to paste a link (Spotify URI) for an existing playlist.
- As a user, I should be able to paste a second link (Spotify URI) for a different existing playlist.
- As a user, I should be able to set the options for the playlist that will be generated.
  - As a user, I should be able to choose a name for the generated playlist.
  - As a user, I should be able to write the description for the generated playlist.
- As a user, upon submission of the required information, I should be shown some form of progress status as the playlist is generated.

- As a user, I should receive a playlist which fits mine and the other playlist author's music taste.
  - As a user, I should be able to view the playlist on the website.
  - As a user, I should be able to view the playlist on my Spotify account, via some Spotify client.

4.2 Software Engineering Approach

4.2.1 Methodology

I used an Agile methodology to complete the project, which allowed the project to adapt to changing requirements and alterations in the direction taken. The Agile approach focuses on working software as a measure of progress, with component parts of the working software being delivered regularly and in a short period of time; this resulted in the project seeing constant improvement which was always aligned with the requirements.

4.2.2 Project Management

Project management is vital to ensuring the development process remains on track, and in this instance, Trello was used as the primary project management tool. As described on their website, Trello is a flexible and visual way to manage projects and collaborate effectively using boards, lists and cards [37]. Trello is primarily aimed at collaborative teams, however the tool is still effective for deploying the Kanban methodology for any project, regardless of the number of contributors. Kanban is a widely-used framework for implementing Agile software development [38]. Overall, the use of Trello allowed for the logical division of development tasks into smaller sprints that focused on single components prevalent in user stories.

4.2.3 Version Control

Version control is managed through a private GitHub repository. Git was chosen for this project as it is the version control system which is commonly regarded as industry standard, and furthermore, the GitHub Education Pack provides a number of benefits - most notably, a private repository free of charge [39].

4.3 Gantt Chart

A Gantt chart, as seen in Figure 4, was produced to aid the planning of the project and help organise the distinct sections and deliverables within the allocated time frame.

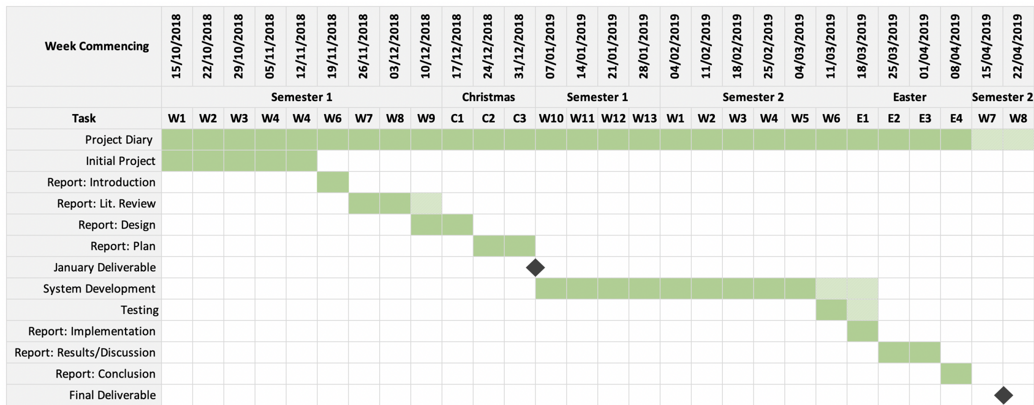

Figure 4: Gantt chart.

This Gantt chart is only intended to be a loose basis for how the project progression will flow, and is likely to be subject to some change. Furthermore, it is highly unlikely that each task will go from start to finish independently; for example, whilst system development is taking place, testing and writing of the implementation as part of the report are likely to be woven together where appropriate.

4.4 Risks

There are a few potential risks that have been identified for the proposed project.

- Machine learning algorithms need a relatively high amount of data to be able to effectively develop a model for classification. Naturally, playlists don't tend to be of an excessive length (let's assume around 100 tracks in a playlist) and this means that the system could struggle to accurately train a model to represent the music tastes of the individual authors. Should this become a problem, one way to mitigate the risk

would be to generate similar data, effectively extend the playlist, using averages of the track features, for example, to produce more training data for the ML algorithm.

- Due to the fact the system is entirely reliant on the availability of the Spotify API, should anything prevent access to the API the system will outright fail. Whilst it is highly unlikely Spotify would suddenly remove public access to the API or the API will be technically unavailable, it is still a risk that should be considered.
- As music is inherently a very personal form of media, there is a possibility that the two playlists provided to the system are so distinctly dissimilar it isn't feasible to recommend tracks that fit the music tastes of both owners. The system should still provide a playlist to the best of its ability; even if the number of joint recommendations is limited, instead a good mix of tracks for both owners should be present.

4.5 Testing

Testing is vital to the success of any product, and in the context of software development allows for the developer to be assured that the system is fully functional and ready for consumer use. By extensively testing the code produced for Smoothie it will help ensure the system is operating as intended and has limited scope for the introduction of potentially code-breaking bugs.

The main testing I will be conducting throughout the development process will be unit testing. Unit tests will be completed as features are implemented; due to the the project being developed using Agile methodology, testing each unit of functionality as and when it is being added to the code base makes the most sense. I will focus on manually unit testing the features as they are added to ensure that they meet the needs they set out to achieve and operate well in unison with the other existing features (a very rudimentary form of integration testing).

Should time permit it, I intend to write and integrate unit tests into the code base to automatically help test functionality as it is added. Using features such as mock objects, method stubs, etc. would result in a code base that is easier to maintain and simpler to expand upon in the future. Furthermore, embedding a test framework in the code base will help to prevent

potentially serious bugs being commit to the code as merges to the master branch on the Git repository could be restricted unless a certain code coverage threshold is met.

System testing will be completed at the end of the project to determine whether or not the system is compliant with the requirements set out at the beginning of this section. This testing will be the main indicator of the success of the system/project.

5 Implementation

The following section of the report outlines and explains the implementation of the proposed system. It covers the environment I used to develop the system, the technical implementations, the testing, and everything else which played a role in shaping the end product.

5.1 Development Environment

Due to the fact I used Trello for my project management (available as an online website), I didn't need to download and install any software. I simply created a Trello Board online and added the tasks I needed to complete into logically separated sections that made it visually very easy to track progress as the development progressed.

To make use of Git for version control, I installed Git on my machine and created a private repository on GitHub. The most notable advantages of this addition to my development environment was that Git allows me to create branches for code features, reliably backup the code as it was being developed and, should it be needed, have access to the code from any machine.

As explained in Section 3.2, the actual development of the project can be divided into two logical components - the website component and the machine learning component.

5.1.1 Website Component

The website can be further divided into two more components - the front-end with which the user interacts, and the back-end which handles most of the data requests and hosts the server. The front-end is primarily developed using React, HTML and CSS, and the back-end using Node.js. There are a few different libraries, packages and tools that I made use of to aid my development of the front and back-end, so I will briefly explain those now.

The back-end server of the website is built using Node.js, which runs in tandem with React, which powers the front-end. Both of these technologies are regarded as industry standard in their own right and add significant functionality to core web technologies such as HTML and JavaScript. More specifically, the server is built using Express, which is a framework that runs on top of Node.js for added web application capability.

The Spotify Web API was an essential component of the project, and without it the MVP simply wouldn't be viable as I required functions of the API on both the front-end and the back-end. For the front-end I used the spotify-web-api-js library, which is a simple client-side JS wrapper for the Spotify Web API. For the back-end I used the spotify-web-api-node library, which much like it's front-end comparison, is simple a Node.js wrapper for the Spotify Web API. Whilst it would have been possible to use the raw Spotify API in both of these scenarios, the spotify-web-api-js and spotify-web-api-node libraries streamlined the process and resulted in a faster development period with tidier and more understandable code.

Both the front-end and back-end were developed in the Visual Studio Code code editor due to it's intelligent syntax highlighting, code suggestions and ability to be customised depending on the packages being used; I could adapt the development environment to best help me write error-free and efficient Node.js and React code on the first pass.

5.1.2 Machine Learning Component

As explained prior, Python, alongside a number of libraries, was chosen as the development language for the machine learning component to effectively implement the logic behind the Smoothie proposal.

I developed the system using Mac OS, so therefore Python comes pre-installed and no further work was required in this aspect. Making use of the pip package installer I could then begin to download the libraries which would help me to build the system.

One of the main libraries I used was the Spotipy library - a Python library for accessing the Spotify Web API. The library was installed with a simple, one-line pip command and then imported in the Python file. From this point, I was able to directly interface with the Spotify API from the Python portion of the code. The main benefit to this is the fact I could now develop the data collection and machine learning algorithms independently from the front-end and avoid having to send the API data from the website component to the machine learning component every time an API request was required.

Inherently, Python is not a programming language that is used for web related projects. However, due to the reasons further described in Section 3.2.2, I wanted to make use of Python as it was undoubtedly the best choice

for the task at hand. This meant I had to find a way to enable interaction between the Python code and the website component code - a task which Flask was built for. Therefore, I installed Flask and Flask-Restful and imported them into the Python code to easily wrap my Python functionality into a Rest API that could be interfaced with from the web component.

PyCharm was my IDE of choice for this aspect of the development as it provides easy project management and enables Python code to be run directly in the editor interface. This was a big quality-of-life feature that meant I could simply develop and test my code at a far more rapid pace than would have otherwise been possible if I had to compile and run from the command line each time I made a change.

5.2 Data Collection

The core of the system is heavily reliant on user data to be able to operate at a sufficient quality level. Without information about a user's music taste, the system will simply fail to provide recommendations that will be relevant and well received; as with any machine learning problem, a good amount of data is also needed to effectively train the classifier.

All of the data used to train the classifier is retrieved from the Spotify API. This connection between the components of the project and the Spotify API is present is various key places throughout the code base, as explained below.

5.2.1 Website Component

The first instance of any data being retrieved is when the user provides the app with approval for access to both read, and modify, their Spotify account data. This process is completed using Spotify's 'authorisation code' which means the user only has to grant access permission only once, and then a refreshable access token is stored (temporarily) within the application. In a nutshell, the process works by the application sending a request to the Spotify Accounts Service, which in turns prompts the user to log in to their Spotify account, and upon a successful login will return the access token. The full process of interfacing with the Spotify Accounts Service can be seen in Figure 5.

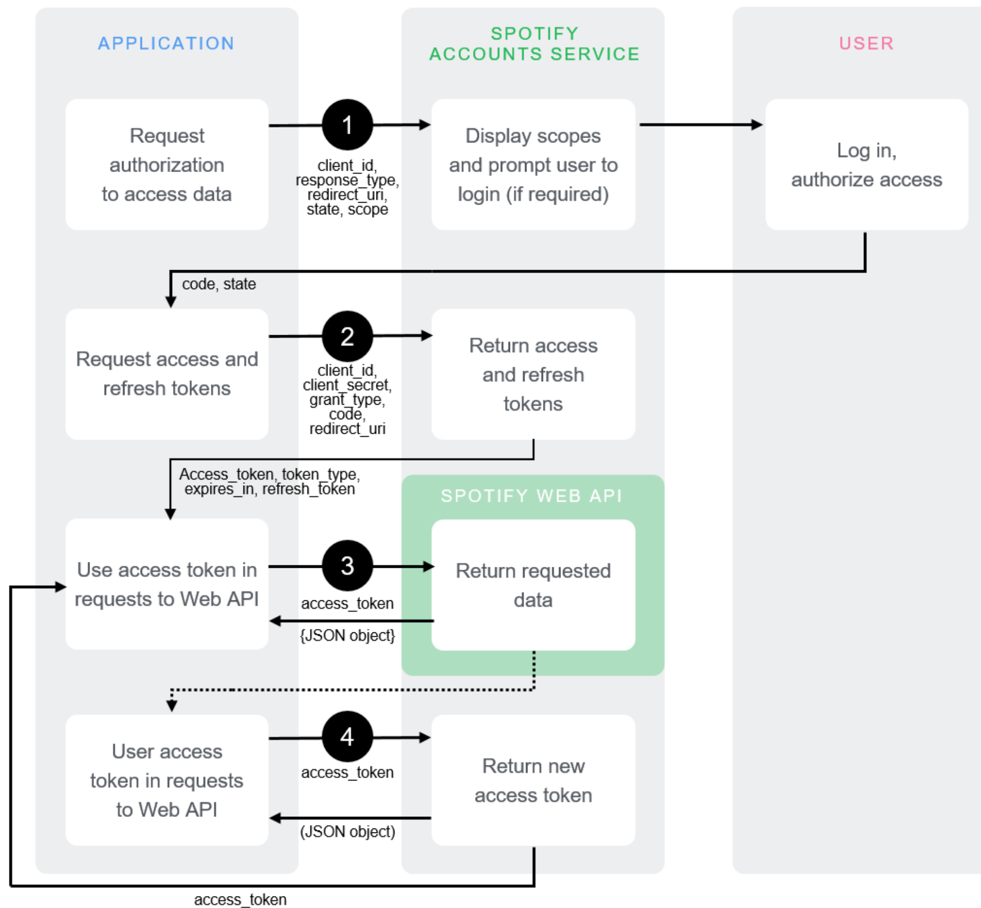

Figure 5: Spotify authorisation code flow [40].

Once the user has successfully logged into their account and the application is authorised, the gathering of the data required for the machine learning and playlist generation begins.

The first two items of data the user is prompted to provide are links to existing Spotify playlists. The two playlists are provided independently to aid with a better user experience, however, both instances of the inputs for these playlists are identical, so the following explanation is applicable to both inputs.

More specifically, the user is asked to provide a Spotify URI for a playlist. A Spotify URI (Uniform Resource Indicator) is a string that can be used to

find an item of data stored on the Spotify platform, and will follow a structure something along the lines of:

spotify:user:spotify:playlist:37i9dQZF1DWY4lFlS4Pnso

The most important part of the URI is the base-62 identifier at the end of the string (*37i9dQZF1DWY4lFlS4Pnso*, in the case of the above example). This part of the URI is called the Spotify ID and can be used to identify any track, artist, playlist, album, etc. This is the part of the URI that is utilised in calls to the Spotify API to retrieve data.

Therefore, the point can be argued that making the user input the entirety of the Spotify URI for a playlist when only the Spotify ID is needed to retrieve its details from the API is excessive. My decision to implement the input in this manner again comes down to a UX decision. From the Spotify interface on desktop or mobile, a user can right click (or navigate to the share options) on a playlist or track and choose 'Copy Spotify URI'. By simply accepting this information it saves the user the effort and extra step of extracting the ID from the string themselves. Furthermore, this reduces the potentiality for errors as a simple copy and paste is far easier than relying on the user to correctly extract the ID from the URI themselves.

From the perspective of the code, when the user pastes the URI into the text field a function called '`extractURI(URI)`' is triggered, with the user's input being passes as the 'URI' parameter. This function splits the URI on the ':' characters and stores the final component (the ID, which is always at the end of the URI) in a new variable for later use.

With the ID of the playlist now stored in the system, a call to a function called '`getPlaylist(id)`' is made, with the ID as the parameter. The purpose of the `getPlaylist` function is to make a call to the Spotify API, retrieve the corresponding playlist object and store it in the system. Thanks to the spotify-web-api-js library, this is a very simple process. Using the `spotifyApi` object defined at the start of the code, the `getPlaylist` helper function is called with the ID of the playlist the user provided. This will make a request to the Spotify API and, if successful, will return a response containing a playlist object. The component state is updated to contain the retrieved playlist.

Now that there is the possibility for playlist information to be stored in the system, a check can be performed within the `render()` component of

the app to see whether further information can be displayed. The check will determine whether the playlist variable in the state is empty (indicating that the user has no provided input yet and a playlist has not been retrieved) or contains a playlist object (indicating the input has been provided and a playlist retrieved). If the check determines a playlist is stored in the variable a number of components are defined and displayed to the user, acting as a playlist preview.

The playlist preview contains the title of the playlist, the author/owner of the playlist, the playlist cover image/thumbnail, the first three tracks in the playlist, and finally the length of the playlist - as seen in Figure 6. This information appears almost immediately after the user pastes the URI in the text field, and this near immediate feedback helps the user to identify any errors before progressing to the next stage - it'll be clear if the wrong information has been input.

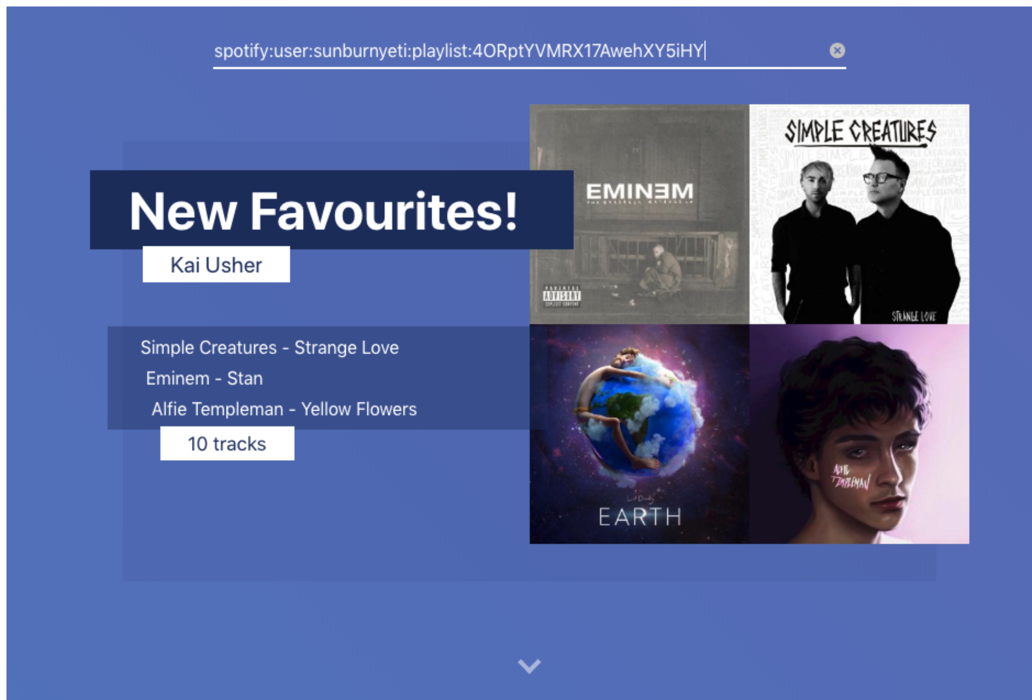

Figure 6: Screenshot of the playlist preview that is displayed following the input of a playlist.

As mentioned above, this is the same process for both of the input playlists.

Once the playlists have been provided by the user and the respective data loaded into the system, the next stage of the process is configuring the settings of the playlist that will be generated. This is a much simpler process for data collection than the playlist, as no external API calls need to be carried out.

First, the user will input a name for the playlist that will be generated. One thing to note here is that upon inputting the chosen name, the system will store the string with the addition of ' - blended by Smoothie' on the end. This has no impact on anything other than retaining branding in the end product.

The second option in this section is a toggle switch to determine whether the generated playlist will be public (visible to anyone on Spotify) or private (only visible to the author - in this case, the account the user logged in/authenticated with). By default, the value is set to public as this is the more likely choice due to the collaborative nature of the system.

Finally, the user can then select how long they would like the playlist to be. This is done by positioning an indicator on a slider; the value selected is displayed on a label below the slider.

Once all of this information is gathered the system is ready to generate a playlist. It has the playlists required to train the classifier, a title for the playlist, a length for the playlist and knowledge whether the playlist should be public or not.

5.2.2 Machine Learning Component

The data collection for the machine learning component is much more focused on the Spotify API than the data collection on the website - the user doesn't interact with this component of the system or explicitly provide any more information than they already have.

There are four pieces of data that need to be passed to the Rest API in order for it to execute successfully: the user ID, the ID of the first playlist, the ID of the second playlist and the desired length of the generated playlist.

Much like loading the playlist data on the front-end, the code explained below must be completed for both playlists, and therefore the process is

logically identical.

The first aspect of data retrieval occurs when the code makes a call to the Spotify API, using the playlist ID provided, to request the data about said playlist. In this situation, only the tracks themselves are required, so the tracks portion of the returned object is saved into an object and the rest is dropped.

This object containing all of the track information is still unnecessarily verbose, containing information such as the date that each track was added to the playlist and the global regions in which the track is available. To extract the useful information, each track is passed into a function that returns an array containing the ID of the track, the name of the track, and the track artist; each song array is then compiled into one main 'tracks array'.

Following the first implementation of the code to create the tracks array it soon became clear there was an error in the way the code executed. I discovered that the Spotify API returns paginated results, so when a playlist was provided that contained more than 100 songs, the tracks with a position greater than 100 were simply being missed off. To solve this (after investigation in the API documentation) I discovered that the return of the API call contains a paging object, and within that object is a key defined as 'next' - the value of this key will either be the URI of the next page of items (i.e. the tracks past position 100) or 'null' if there is no next page. Using this information I implemented a simple loop that executed while the value of the 'next' key was not 'null'; the contents of the loop simply updated the tracks object with the next page of tracks and then repeated the above paragraph, appending the new tracks to the end of the tracks array.

```
1 results = sp.user_playlist(username, playlist_id , fields='tracks,
    next')
2
3 tracks = results['tracks']
4
5 #Strip unnecessary information using track_info function.
6 playlist = extract_track_features(tracks)
7
8 #Handle pagination of API response.
9 while tracks['next']:
```

```
10      tracks = sp.next(tracks)
11      playlist += extract_track_features(tracks)
```

Listing 1: Code showing the generation of the tracks array.

Now that all the the tracks have been extracted from the user input playlist, the features of the tracks need to be retrieved in order to train the classifier. In order to extract the track features, each track (URI, track name, artist name and artists ID) is passed into a '`get_audio_features()`' function, one after another.

The first action in this function is a request to the Spotify API - '`audio_features (uri)`' will send a call to the Spotify API with a track identifier, and in return will get an object containing a breakdown of the tracks features. The response is then looped through and each features extracted, and appended to a 'features array'. Once this loop has been completed the system will have an array containing all required information about a track - danceability, tempo, valence, etc.

The final step of the audio features retrieval is to store the data in a Panda's dataframe; this is a simple process of just appending each feature into the correct column of a defined dataframe. Once this has been done, the above two paragraphs are repeated with the rest of the tracks in the user's playlist. Upon completion, the system will have a dataframe which completely represents the user's playlist and shows all of the features for each track, ready to be used for training a classifier.

With both playlists having been processed and stored in their independent dataframes, they then need to be compiled into a single data source to train the classifier. This is achieved by defining a new dataframe that is initialised with the column headers, appending the 'playlist1/user1' dataframe and appending the 'playlist2/user2' dataframe. To finish the process, this dataframe is then stored as a .csv file ready for classifier training. An example file is seen in Figure 7.

owner	id	name	artists	danceability	energy	key	loudness	mode	speechiness	acousticness	instrumentalness	liveness	valence	tempo	duration_ms	time_signature
user1	51kzBRf8LFVFybAs7n71y	Strange Love	Simple Creatures	0.59	0.6	6.0	-7.25	1.0	0.03	0.01	0.0	0.12	0.39	91.96	151632.0	4.0
user1	0xAhbD6lCf5re1RXl5yiT	Sweet Talkin' Woman	Electric Light Orchestra	0.61	0.61	0.0	-9.21	1.0	0.03	0.51	0.0	0.09	0.92	121.84	229867.0	4.0
user1	4QVOTT9CM2ftSLwnYGNDj	Stan	Eminem	0.78	0.74	6.0	-5.07	0.0	0.21	0.04	0.0	0.45	0.53	80.06	404427.0	4.0
user1	1eQBEelI2NCy7AUTerX0K	Ultralight Beam	Kanye West	0.6	0.39	8.0	-8.71	1.0	0.38	0.64	0.0	0.54	0.36	107.42	320680.0	4.0
user1	5HJFKE5LsThIH0RpCr8Hl	Yellow Flowers	Alfie Templeman	0.44	0.92	4.0	-4.05	1.0	0.08	0.14	0.3	0.32	0.81	123.72	186259.0	4.0
user2	1CDVadnneswMxi6gBqJTt	Legend	Twenty One Pilots	0.72	0.57	0.0	-6.21	1.0	0.04	0.09	0.0	0.08	0.8	85.94	172560.0	4.0
user2	0UvVZwaKrd4JRXS03D59p	Lake Effect Kid	Fall Out Boy	0.57	0.95	5.0	-2.29	1.0	0.05	0.01	0.0	0.31	0.66	142.01	220533.0	4.0
user2	3YKptz29AsOlm7WAVnztB	Kiss From A Rose	Seal	0.58	0.53	10.0	-7.11	0.0	0.03	0.68	0.0	0.31	0.22	131.74	288427.0	3.0
user2	6r20M5DWYdIoCDmDViBxu	You Got It	Roy Orbison	0.65	0.62	9.0	-10.73	1.0	0.03	0.6	0.0	0.19	0.67	114.69	210267.0	4.0
user2	33glxrKoHmtZryGnIZ5vM	Something Good	Jule Vera	0.47	0.65	10.0	-4.52	1.0	0.07	0.37	0.0	0.12	0.34	171.97	217800.0	4.0

Figure 7: An example .csv file.

5.3 Website

5.3.1 Front-End

The main component of the front-end portion of the system is built using React. One of the main features of React is the ability to render views based on states; given a particular state of the website/application, React will render a corresponding interface [25]. The use of states is key to the implementation of Smoothie and I will explain a few of the most notable instances in the following section.

The first instance of a state determining the interface being displayed is the differentiating state of 'is the user logged in?' - `loggedIn` is a Boolean state that is true if the user has successfully logged in and an access token received, or false if the user has not logged in. Whilst `loggedIn` is false, the user is presented with an interface that briefly explains what Smoothie is and a button that prompts them to 'connect with Spotify', aka, log in to the website. When `loggedIn` is true, the user will either see the form to input the data required for the system, or if they have already completed this step, the generated playlist.

Using these states, React dynamically generates the HTML/DOM depending on conditions. For example, the header of the website needs to change depending on whether the user is logged in or not; if they are logged in, their profile image and a 'log out' button should be displayed in the header, but this element is of course unrequired/impossible to display if the user has not logged in. With a simple condition based on the state of `loggedIn`, React knows whether to display the log out element or not.

```
<header className="header">
    <div className="logo">
        <img src={require('./assets/logo_white.svg')} alt="logo
    "></img>
    </div>

    <!-- If the state of loggedIn is true, display the user
    summary element. -->
    { this.state.loggedIn &&
        <div className="login">
            <div className="login-left">
                <div className="username">
                    Hey, {userName}!
                </div>
                <div className="logout">
                    <a href="/logout">Logout</a>
                </div>
            </div>
                <div className="login-right">
                    <img src={profileURL} alt="pp" className="
    profilepic"></img>
                </div>
        </div>
    }

          </header>
```

Listing 2: Code showing the rendering of the header elements based on the current state of `loggedIn`.

A similar implementation is used to display a loading indicator/animation whilst the playlist is being generated. Upon the user pressing the 'generate' button, the `loading` state is set to true and the interface is updated to show a loading indicator.

The loading indicator itself is designed to represent the bars of a music equaliser as I felt this was more on-brand and visually engaging than the generic loading wheel or progress bar. The animation of the bars is achieved in pure CSS animations; the height of each bar is rotated through different heights and each bar of the visualiser is triggered at a varying time offset to the previous one, thus resulting in an animation which represents a music

visualiser, as seen in Figure 8 and Figure 9.

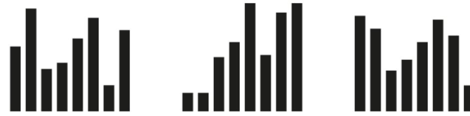

Figure 8: Three possible instances/frames of the loading animation.

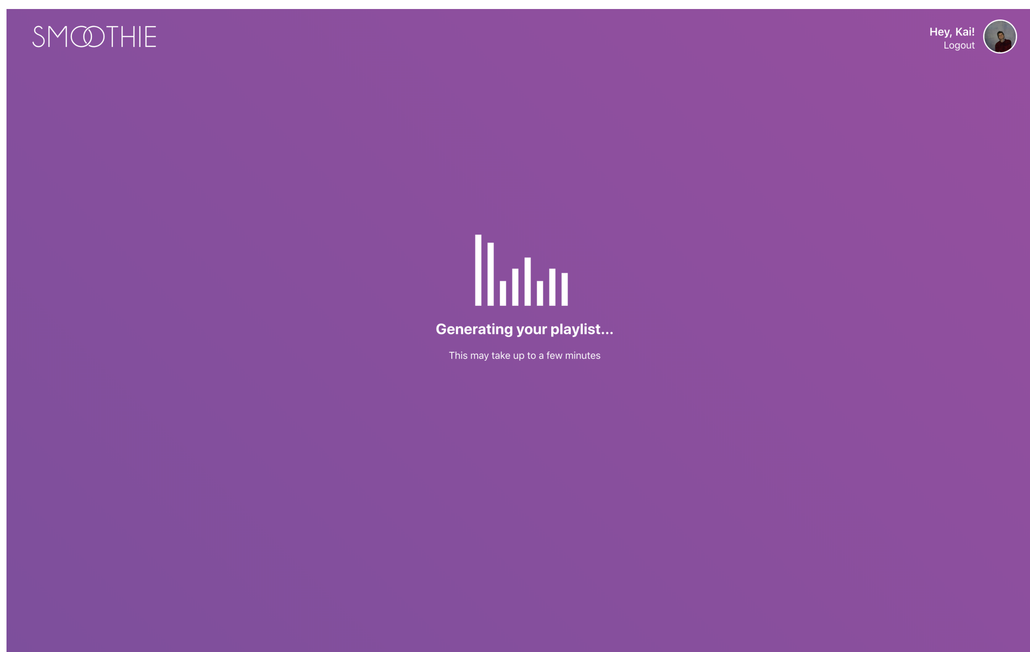

Figure 9: Screenshot of the loading screen.

As well as using the states to determine which of the main components, or 'pages', should be displayed to the user (log in interface, form interface, loading interface or generated playlist interface), states are also used to store and manage the important information that is utilised by the system. Most notably, the playlists are stored in states.

By storing the playlist objects in states, the data can then be accessed to render dynamic elements depending upon the information the user has provided. A good example of this is the rendering of the 'playlist preview'

which is displayed when the user pastes a playlist URI in either one of the playlist input fields.

```
{ !this.isEmpty(this.state.playlist1) &&

    <div className="playlist-display">
        <div style =
            {{backgroundImage: "url(" + this.state.playlist1.
    images[0].url + ")"}}
            alt="album"
            className="album-art"
        ></div>
        <div className="playlist-details">
            <h2 className="playlist-name">
                {this.state.playlist1.name}
            </h2>
            ...
```

Listing 3: Code showing a section of the rendering of the playlist preview element.

Utilising states allows for far more adaptive code reuse due to the absence of hard-coded content. This style of DOM element rendering is prevalent throughout the entirety of the code base wherever dynamic content needs to be displayed to the user.

Errors are another example where states are used to dynamically determine what information should be provided to the user. Each input field (playlist 1, playlist 2 and playlist name) have an error element which is made visible when there is an issue with the user input. An example of such an error message can be seen in Figure 10.

A playlist is only saved to a state in the system if the return from the API call to retrieve the playlist details is successful - i.e., doesn't return an error. If the user has input an invalid playlist URI the API call will return an error and the respective playlist state will remain an empty object. This means that a similar implementation as seen in the code to render the playlist

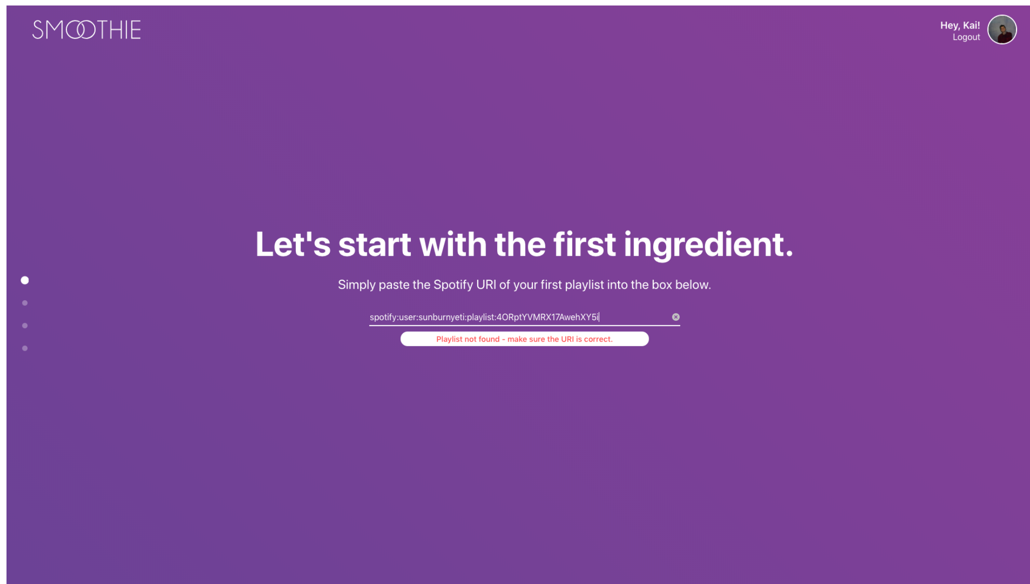

Figure 10: Example of a Smoothie error message.

preview can be used to render an error message.

```
<div className="playlist-input-section">
    <input className="playlist-input"
           type="search"
           placeholder="Paste the URI here!"
           spellCheck="false"
           onChange={(e) => this.extractURI(e, 1)}>
    </input>

    { this.playlistErrorCheck(1) && this.state.playlist1Input &&

        <p className="input-error">Playlist not found - make sure the URI is correct.</p>

    }

</div>
```

Listing 4: Code showing rendering of the conditional error.

The error message rendering uses two conditions to determine whether the element should be displayed or not - `this.playlistErrorCheck()`, which will return true if there is no playlist stored in the system, and `this.state.playlistXInput`, which is true if the user has interacted with the input field. The second condition is used to ensure the error messages aren't displayed when the user first visits the website and is yet to input a URI (in the initial state of the system the playlist objects will be empty). Upon entering text into the input field, regardless of whether it is a valid URI or not, the `playlistXInput` state will be set to `true` and the error messages will then have the opportunity to be displayed.

The same conditions are used to display the chevron at the bottom of the screen upon successful input of a playlist URI. The chevron, which when clicked on will scroll the user to the next input section, is only displayed when there is a playlist object present and thus no error message.

Combining all of the error handling states allows for the gating of the generation triggering; if any one of the inputs has an error the user will be unable to click the 'generate playlist' button. Should they try to click the button when an error is present, the system will highlight the respective error to provide a prompt of what the user needs to do. This prompting can be seen in Figure 11.

The glow seen in Figure 11 fades in from zero when the user clicks the generate button with an error present, and persists until the error is rectified. To further aid the user in fixing the error, the errors shown beneath the generate button can be clicked on. Upon clicking an error the user will be navigated to the respective section so it is clear to them what needs to be fixed.

Another notable implementation of the front-end is the dynamic generation and rendering of the playlist which is returned from the machine learning/recommender system API call. The length of the playlist is determined by the value the user inputs in the initial data collection, and therefore the length of the playlist display needs to be able to easily adapt; each track in the playlist has its own element which needs to be rendered.

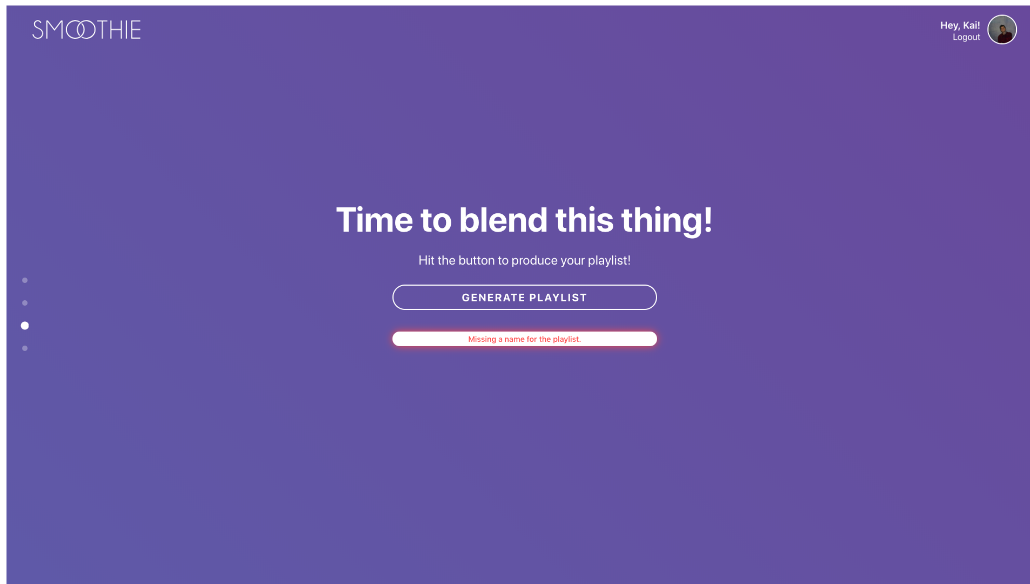

Figure 11: Example of a Smoothie error message being highlighted.

To achieve the desired functionality, a function called `displayPlaylist()` was written. The function contains a loop which executes the same number of times as there are tracks in the playlist (47 track long playlist = 47 loops), and within each loop a HTML element is generated and pushed to an array.

By using the index of the loop in conjunction with the state which stores the generated playlist, only a single version of the HTML element needs to be defined to determine the structure and the variable content is dynamically updated.

```
1 displayPlaylist(){
2     var playlistHolder = [];
3     for (var x=0; x < this.state.genPlaylist.tracks.items.length
    ; x++){
4         playlistHolder.push(
5             <div class="track-cont" key={x}>
6                 <div class="track-art" style=
7                     {{backgroundImage: "url(" this.state.
  genPlaylist.tracks.items[x].track.album.images[1].url + ")"}}
```

```
8                 ></div>
9                 <div class="track-details">
10                    <b class="track-title">
11                        { this.state.genPlaylist.tracks.items[x
    ].track.name.substring(0,75) }
12                    </b>
13                    <ul class="artist-list">
14                        { this.displayArtists(x) }
15                    </ul>
16                </div>
17                <div class="track-info">
18                    <span class={ this.display2Prediction(x) }
    style=
19                        {{backgroundImage: "url('" + this.
    displayPrediction(x) + "')"}}
20                    ></span>
21                    <span class="track-length">
22                        { this.millisToMinutesAndSeconds(this.
    state.genPlaylist.tracks.items[x].track.duration_ms) }
23                    </span>
24                </div>
25            </div>
26        )
27    }
28    return playlistHolder
29 }
```

Listing 5: Code showing the generation of the elements to display the generated playlist.

Being able to define HTML elements within JavaScript code also allows for cleaner segmentation and more readable code. For example, if an element of the DOM requires logical conditioning, instead of defining the HTML inline, a function can be defined that implements the logic and returns the correct HTML element, or component of a HTML element. This is seen when defining the image to use for the background of the element which displays to the user what owner is most likely to enjoy the song. The url of the image depends on the value of the item in the list of predictions, so a simple switch

statement can be implemented to return the correct value.

```
displayPrediction(index){
  switch (predictions[index]) {
    case 0:
      var doubleURL =
          this.state.user1_img + "'),
          url('" + this.state.user2_img;
      return doubleURL;
    case 1:
      return this.state.user1_img;
    case 2:
      return this.state.user2_img;
  }
}
```

Listing 6: Code showing the logic to dynamically define the background image for HTML elements.

The result of the aforementioned implementation is a relatively slimline code base and an adaptive, dynamic front-end that can be flexible enough to suit all potential needs of the system. Figure 12 shows the display of a playlist that has been generated by Smoothie.

One problem I encountered when developing the website was related to Cross-Origin Resource Sharing (CORS) - 'A web application executes a cross-origin HTTP request when it requests a resource that has a different origin (domain, protocol, and port) than its own origin' [41].

When making calls from the front-end to the Spotify API to add tracks to the generated playlist I was receiving the error: `no access-control-allow -origin header is present on the requested resource`.

To solve this problem I enabled CORS on the back-end of the website and moved the functionality to add tracks to the back-end file.

```
app.get('/addSong', function(req, res) {
    var tracksToAdd = JSON.parse(req.query.songs);
    var playlistId = req.query.playlistId
```

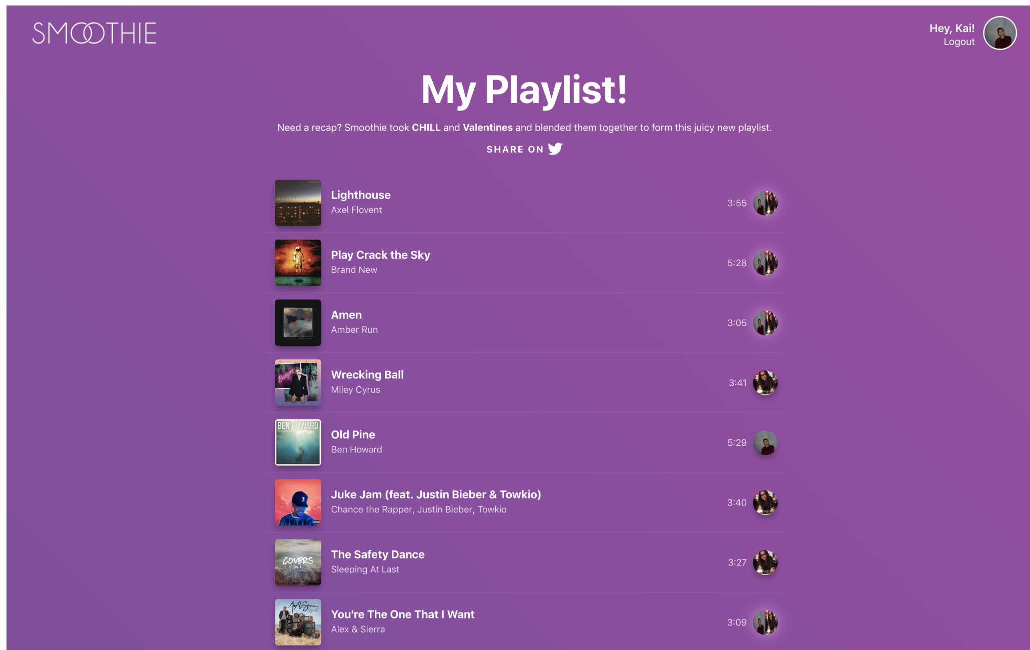

Figure 12: Screenshot of the generated playlist screen.

```
    spotifyWebApi.addTracksToPlaylist(playlistId, tracksToAdd,
    {
      position : 0
    })
    .then(function(data) {
      res.send(playlistId);
    }, function(err) {
      console.log(err);
    });
});
```

Listing 7: Code showing the back-end function to add tracks to a playlist.

From the corresponding front-end function where I needed to add tracks to a playlist I could then make a call to this function in the following manner: `fetch('http://localhost:8888/addSong?' + songString + '&playlistId =' + generatedId)`. This would result in all of the track IDs contained in the variable `songString` to be added to the playlist with the ID stored in the variable `generatedID`.

By performing this task on the back-end with CORS enabled, the request to the Spotify API has the appropriate CORS headers and successfully adds the tracks with no problems.

5.3.2 Back-End

The majority of the server component of the website is provided by Spotify's Web API Tutorial [28]. Other than the aforementioned functionality to add tracks to the generated playlist, the only other functionality is to handle the logging in and authentication of a user.

There are two main calls to achieve the authentication. The focus of the first call is to prompt the user to provide authentication for the services that the website is providing and allow access to the required information - this call is done via the `/authorize` endpoint of the web API. The second call is to the `/api/token` endpoint of the API and results in the API returning an access token. Following these calls, the web-app should be successfully authorised and subsequent calls to access user data will be permitted.

5.4 Machine Learning Implementation

I am using scikit-learn for the machine learning aspect of the project.

On the sci-kit learn website there is a useful graphic for determining which algorithm is most appropriate for the job, which I have recreated in Figure 13 [42]. Working through this flowchart results in a few different algorithms being recommended for the machine learning problem at hand:

- Linear SVC
- K-Neighbors Classifier
- Ensemble Classifiers
- SVC

With this information in mind, I ran some tests using each of the algorithms to see which of them performed the best (had the highest level of accuracy) on the sample data I produced. In regards to the ensemble classifier, I chose to use a random forest. From initial tests, the results of the algorithms can be seen in Table 2.

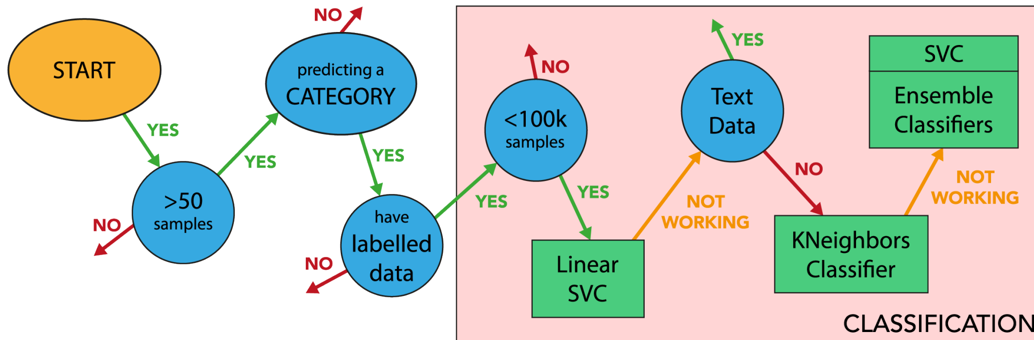

Figure 13: Recreation of the scikit-learn flowchart for algorithm selection [42].

Algorithm	Accuracy	Scikit-Learn Algorithm
Linear SVC	74.24%	LinearSVC()
K-Neighbors	69.69%	KNeighboursClassifier()
Random Forest	95.45%	RandomForestClassifier()

Table 2: Comparison of the the selected classification algorithms.

Following these tests, it was clear that the best choice for the task at hand was the Random Forest Classifier.

Scikit-learn makes implementation of these models very simple. First of all, a basic model is initialised using the `RandomForestClassifier()` function within the library and then passed into the `GridSearchCV()` function with a set of parameters in the form of a parameter grid. The parameter grid contains a `n_estimators` value (number of trees in the forest) and a `max_features` value (the size of the random subsets of features to consider when splitting a node). `GridSearchCV()` will take the parameters found within the parameter grid and exhaustively search through them to find the best parameter combination for the model.

```
# Create a random forest object.
randomforest = RandomForestClassifier()
```

```
3
4 # Define the parameters for the search.
5 param_grid = {
6     'n_estimators': [200, 800],
7     'max_features': ['auto', 'sqrt', 'log2']
8 }
9
10 # Combine all components of the model.
11 randomforest = GridSearchCV(estimator=randomforest, param_grid=
       param_grid)
```

Listing 8: Code showing the definition of the model.

The values used for the parameter grid were decided upon following a number of tests to determine what would be the most effective setup. For example, Figure 14 shows the testing of values to determine which parameters should be used as the `n_estimators` value.

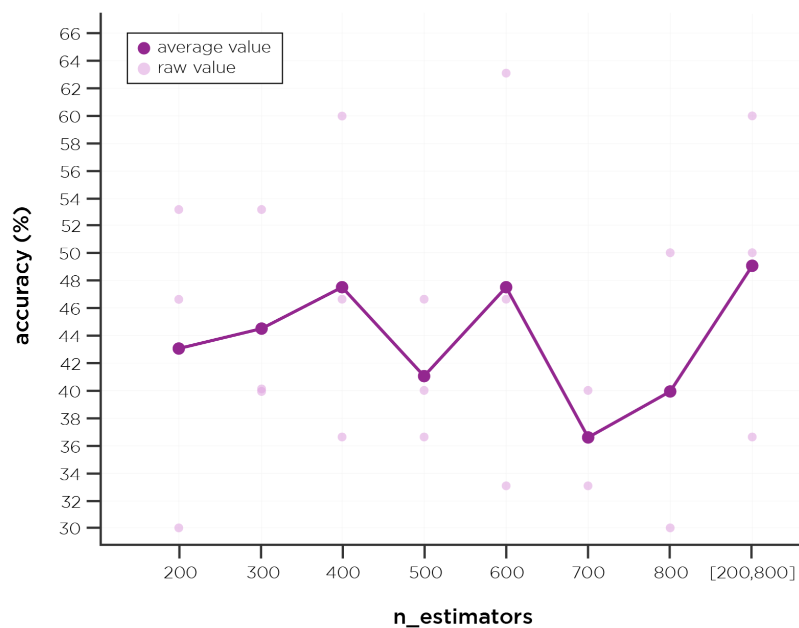

Figure 14: Results of testing different values for `n_estimators` of the classifier.

The test involved passing two playlists into the system, training a model

using them and recording the resulting accuracy of said model. From this test, it showed that the option to use a range of possible values - `[200,800]` - is the best choice. As stated above, using this option means that the algorithm will use parameters in this range to determine which is the best parameter to use for the model.

Once the model had been defined, the data to be used for training needed to be prepared.

The dataframe containing all of the information for both provided playlists is loaded and split into two separate tables - one for each of the playlists. As the tables contain a combination of both numerical and non-numerical values, all of the columns containing the non-numerical (such as, 'artist' and 'track title') are dropped from both tables.

By utilising the scikit-learn `train_test_split()` function, the data is randomly split into training and testing subsets. The parameters used for the function are the features to be trained on (all numerical features in the tables remaining after the dropping of non-numerical values), the target of the classifier (the playlist owner) and the size of the percentage of the data to be used as test data (0.15 in this case). This returns four variables: a training data set, a testing data set, a training target set and a testing target set.

The accuracy of the classifier can then be tested by passing the testing data set into the model and storing the outcome in a variable; by comparing this variable with the target testing set, using the scikit-learn `accuracy_score()` function, a percentage value can be produced which represents how accurate the model is.

In order to fulfil the requirements of the project, it is required that the system knows how confident the classifier is in predicting the owner of a given track; if the owner cannot distinctly be predicted by the classifier, the track should be recommended to the user.

The confidence ratings of the classifier are determined through the `predict_proba()` method of the model. The list of probabilities has an item for each item in the list of predictions produced by the model, and therefore they can be compared side by side to see each prediction and the probability it is correct.

```
# Pass a table of tracks (testing_df) into the classifier and
    get predictions for owner/user.
prediction = randomforest.predict(testing_df).tolist()

# Get the probability for each track prediction.
confidenceRatings = randomforest.predict_proba(testing_df).
    tolist()

# Simple test to check predictions and the probility they are
    correct
for x in range(len(prediction)):
    print(x, ' - PRED/CONF - ', prediction[x], '/',
    confidenceRatings[x])

>>> 1  - PRED/CONF -  user1 / [0.52, 0.48] # 52% sure user1
>>> 2  - PRED/CONF -  user2 / [0.24, 0.76] # 76% sure user2
```

Listing 9: Code showing the prediction and probability generation.

These values then need to be converted into values than can be directly correlated with either owner 1, owner 2, or both/unable to predict a clear owner. The logic I implemented to achieve this is based around the mappings seen in Table 3.

Value	Predicted Owner
0	Unable to choose distinct owner.
1	Most likely to be enjoyed by owner 1.
2	Most likely to be enjoyed by owner 2.

Table 3: Table showing mappings between values and predicted owner.

Using this mapping, a single integer can be used to represent the prediction of the classifier, and therefore only a minimal amount of information

needs to be returned from the API - for example, [0,1,2,2,0] represents [both, owner1, owner2, owner2, both].

In order to retrieve these values from the information returned by the classifier the list of probabilities is passed into a function called confidence_values(). This function simply determines if the two probability components of a prediction are similar (if so, the prediction is assigned '0' to represent both users would enjoy the track), or if there is a relatively large difference between the values then the larger one is determined and the respective value assigned to the prediction (1/2).

```
# Declare variable to store the predicted owner.
    predicted_owner = []

    # Loop through each probability item and determine which user is most likely to enjoy the song.
    for x in range(len(probabilities)):
        # If the first value is largest then user 1 is the predicted owner and vice versa.
        if probabilities[x][0] > probabilities[x][1]:
            # If the difference between the values is small (<0.25) then the owner can't distinctly be predicted.
            if probabilities[x][0] - probabilities[x][1] < 0.25:
                predicted_owner.append(0)
            else:
                predicted_owner.append(1)
        else:
            if probabilities[x][1] - probabilities[x][0] < 0.25:
                predicted_owner.append(0)
            else:
                predicted_owner.append(2)

    return predicted_owner
```

Listing 10: Code showing the assigning of values to the predictions.

5.5 Recommender System

Now that the required data is collected and the machine learning system built, the final aspect of development that needed to be covered was the process of using the two aforementioned components to recommend suitable tracks to be compiled into the final playlist.

The Spotify API contains an endpoint for getting recommendations from the Spotify music library, so I chose to use this as my method of retrieving tracks. Following some initial testing, I came across some problems with the endpoint of the API which will be discussed throughout this section.

The endpoint is officially titled 'Get Recommendations Based on Seeds'; tracks are generated based on a number of provided seed entities that are used to create matches against similar tracks and artists [43]. There are various seeds that can be passed into the API, three of which can be considered the 'main seeds': seed_artists, seed_genres and seed_tracks. A value for at least one of these seeds must be provided in order for the API to return a response object with recommended tracks.

Initially, I was hoping to simply use the track attributes (danceability, tempo, etc.) to gather recommendations, however after gaining a better understanding of the API endpoint it became clear I would not be able to do this due to the aforementioned reason of needing either a seed artist, genre or track. This raised a potentially big problem as the required seeds are not numerical, and therefore I could not simply add them to the machine learning algorithm/classifier. Therefore, I would have to treat them in an almost mutually exclusive fashion.

With the knowledge that I needed to work with more data related to the provided playlists I had adapt the data collection of the Python scripts. Prior to the adaptation, the dataframe contained the artist name, and not the artist ID, which the recommendations endpoint requires. It was a simple fix to add the corresponding artist ID to each entry in the dataframe as the data was already being gathered from the initial call to the API to retrieve the track details, and thus just needed to be appended to the information being saved in the dataframe. Following this change, the dataframes now have an additional column which stores the artist ID for each track.

Gathering the genre information for a track was slightly more complex as this information was not already being gathered by the system. Spotify works

by associating genres to artists and not individual tracks. This means that in order to determine the genre of a track you have to retrieve the details of the artist to whom the tracks belongs using the 'Get Artist Details' endpoint of the Spotify API. Thanks to the fact the artist ID was now being stored in the system, I could easily utilise it in order to make a call to get the respective genres. This call to the API will return a list of all genres associated with an artist; as I only required one genre for each artist I simply took the first element in the list, with the reasoning that this is the one which best represents the artist and also that I had no available information or data to justify a better choice. This genre information was then appended to the dataframe for each entry.

Following these changes to the data collection, the dataframes now contained all of the information that was required to gather recommendations. Table 4 shows the seeds passed to the API endpoint.

Seed	Description
limit	The target size of the list of recommended tracks.
target_*	For each of the tunable track attributes a target value may be provided; tracks with the attribute values nearest to the target values will be preferred.
seed_genres	A comma separated list of any genres in the set of available genre seeds.
seed_artists	A comma separated list of Spotify IDs for seed artists.

Table 4: Comparison of the the selected classification algorithms.

With all of the data stored, it then needed to be manipulated in a way that would be reflective of the user's music tastes as a whole whilst also being in a form that could be passed to the API.

The limit seed was easily the most simple, with the value the user entered in the web-app being used for this seed.

Both the artist and genre seeds come with the caveat that a maximum of 5 seed values can be provided in any combination of artist, track and genre. With potentially hundreds of artist IDs and track genres being stored in the dataframe, I had to determine a way to extract the most reflective information.

For the artist, I used the Pandas '`.value_counts()`' function to create an object containing counts of the unique values in the 'artist ID' column. This resulted in an object that showed the most popular artists in the dataframe (and therefore the user's playlists). With the API imposed limits in mind, I took the top three artists and stored the ID values from the dataframe in a list.

```
1 top_artists = combo_df.loc[:, 'artist_id'].value_counts().
    nlargest(3).index.tolist()
```

Listing 11: Code showing the collection of the top three artists.

For the genres, I approached the situation with the same logic I did for the artists; I counted the unique genres found in the dataframe and stored the top two (five total seeds minus three artist seeds) in a list.

Following some initial tests making calls to the API, it immediately became clear that the genres I was providing were returning an 'invalid request' error. After I dived deeper into the API reference guide I discovered that the recommendations endpoint only accepts a subset of the genres Spotify uses to categorise the artists on the platform.

```
1 possible_genres = ['acoustic', 'afrobeat', 'alt-rock', '
    alternative', 'ambient', 'anime', 'black-metal', 'bluegrass',
     'blues', 'bossanova', 'brazil', 'breakbeat', 'british', '
    cantopop', 'chicago-house', 'children', 'chill', 'classical',
     'club', 'comedy', 'country', 'dance', 'dancehall', 'death-
    metal', 'deep-house', 'detroit-techno', 'disco', 'disney', '
    drum-and-bass', 'dub', 'dubstep', 'edm', 'electro', '
    electronic', 'emo', 'folk', 'forro', 'french', 'funk', '
    garage', 'german', 'gospel', 'goth', 'grindcore', 'groove', '
```

```
grunge', 'guitar', 'happy', 'hard-rock', 'hardcore', '
hardstyle', 'heavy-metal', 'hip-hop', 'holidays', 'honky-tonk
', 'house', 'idm', 'indian', 'indie', 'indie-pop', '
industrial', 'iranian', 'j-dance', 'j-idol', 'j-pop', 'j-rock
', 'jazz', 'k-pop', 'kids', 'latin', 'latino', 'malay', '
mandopop', 'metal', 'metal-misc', 'metalcore', 'minimal-
techno', 'movies', 'mpb', 'new-age', 'new-release', 'opera',
'pagode', 'party', 'philippines-opm', 'piano', 'pop', 'pop-
film', 'post-dubstep', 'power-pop', 'progressive-house', '
psych-rock', 'punk', 'punk-rock', 'r-n-b', 'rainy-day', '
reggae', 'reggaeton', 'road-trip', 'rock', 'rock-n-roll', '
rockabilly', 'romance', 'sad', 'salsa', 'samba', 'sertanejo',
 'show-tunes', 'singer-songwriter', 'ska', 'sleep', '
songwriter', 'soul', 'soundtracks', 'spanish', 'study', '
summer', 'swedish', 'synth-pop', 'tango', 'techno', 'trance',
 'trip-hop', 'turkish', 'work-out', 'world-music']
```

Listing 12: Genres which can be used as genre_seeds for recommendations.

This posed a new problem I had to deal with - the genres stored in the dataframe had to be mapped to the possible genres that could be passed to the API. For example, in a number of the tests I completed on random playlists, the top genres object would contain genres including 'metropopolis', 'steampunk', 'c86', and 'anthem emo', all of which did not feature in the list of genres that could be used for recommendations. With only 126 possible genres to use as seeds, and over 1000 genres in the Spotify platform, the mismatch is very significant.

The currently deployed solution to this problem is as follows. By utilising nested loops, each item in the 'top genres' list (derived from the dataframes of the user input) is compared against each item in the possible genres list. If the item in the possible genres list is a substring in the item from the top genres list, the item from the possible genres list is added to a new list. For some context, if the genre 'anthem emo' was in the top genre list, when compared against 'emo' in the possible genres list it would trigger a match ('emo' is a substring of 'anthem emo') and 'emo' would be added to the list of genres to use as seeds for the API call.

```
# Determine the top five genres from the provided playlists/dataframe.
top_genres = combo_df.loc[:, 'artist_genre'].value_counts().nlargest(2).index.tolist()

# Store all possible seed genres in a list.
possible_genres = sp.recommendation_genre_seeds()['genres'];

# Initialise a list to store genres.
seed_genres = []

# Loop through the top genres and the possible genres, checking each item combination for a substring match.
for x in top_genres:
    for y in possible_genres:
        if y in x:
            # If the possible genre is a substring of the top genre, add the possible genre to list.
            seed_genres.append(y)

# Remove duplicate genres by converting to dictionary and then back to list.
seed_genres = list(dict.fromkeys(seed_genres))
```

Listing 13: Code for determining which genres to use as seeds.

Whilst this works for a large portion of the genres in the Spotify catalogue, acting as an algorithm to convert sub-genres into their parent genre ('underground hip hop', 'southern hip hop', 'hardcore hip hop', would all become 'hip-hop'), it does mean that the more unique genres (such as 'vapor soul' or 'pixie') are lost. As the more unique genres are exactly that, unique, it shouldn't have a large impact on the success of the project. Of course this is not an ideal solution, but due to time constraints and the scope of the project, this implementation is the best possible solution. Without having more contextual information about what makes up the genres on Spotify, aka Spotify releasing a more comprehensive document about their genres/the relationships between them and a better API endpoint for recommendations, there is a limited scope to what I could do to improve this problem.

5.6 Automated Testing

I had intended to implement some form of automated unit and integration tests into the final system. However, time restrictions meant this functionality kept being pushed back as I deemed it of lower priority to the core functionality of the system, and therefore I was unable to complete it in the given time frame.

Whilst it would add significant value to the system for a number of reasons, I believe my decision to prioritise the other requirements was a sensible one and as a result led to a better end product. The system was still tested vigorously throughout the development cycle through manual means so the automation is mostly a quality-of-life feature that was unachieved.

One advantage of automated testing would be that the code base would be maintained to a higher quality and therefore future implementations would be easier and more likely to work smoothly on the first try. This can be taken further to help ensure no serious bugs are introduced to the production code base as contributions to the Git repository could be subject to code coverage checks; only code which is sufficiently tested would be allowed to be merged.

The automated testing is the only functionality that was present in the plan but did not make it into the developed system. I do still deem it an important feature and would make it a priority should I continue to develop the system following this project.

6 Results

6.1 Requirements

As stated in the Requirements chapter of the Plan (Section 4.1), there were seven requirements that must be met for the system to be classed as a success. Below I will explore each of these requirements in more detail to determine whether or not each one was met, and therefore, whether the system as a whole was successful.

6.1.1 Authentication

The system must be able to authenticate a user using the credentials of a pre-existing Spotify account.

Upon navigating to Smoothie, the user is greeted with the log-in screen, as seen in Figure 15. The copy on the button - `CONNECT WITH SPOTIFY` - serves as a prompt to the user that their Spotify account/credentials are going to be used to access the functionality of the website.

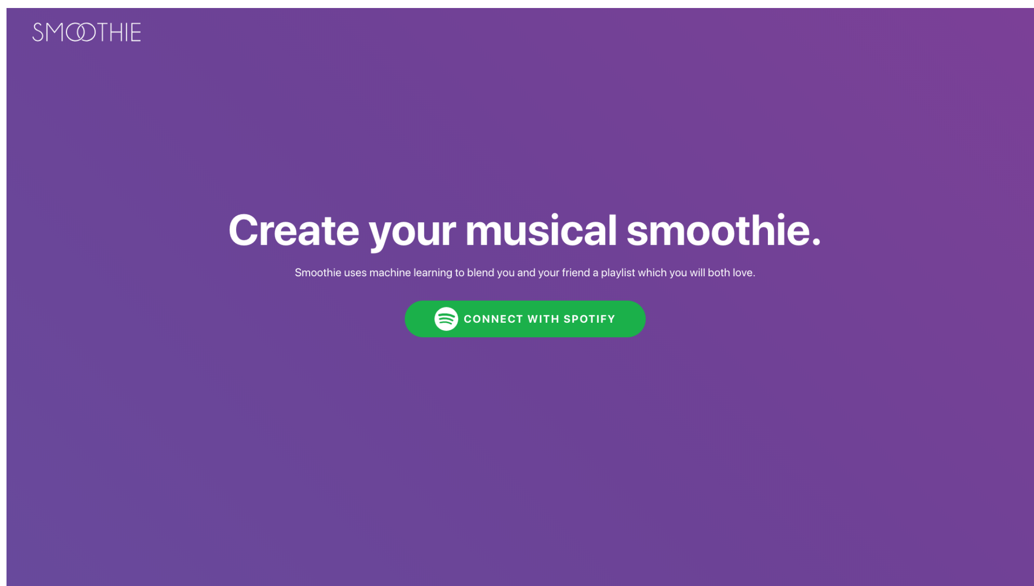

Figure 15: Smoothie log-in screen.

When the user presses the log-in button, the Spotify authentication process is triggered and the user will be presented with the Spotify log-in screen, as seen in Figure 16. From here they can log in to their existing Spotify account using their email and password, log in using their Facebook account (if linked to their Spotify account), or create a new Spotify account if they don't currently have one.

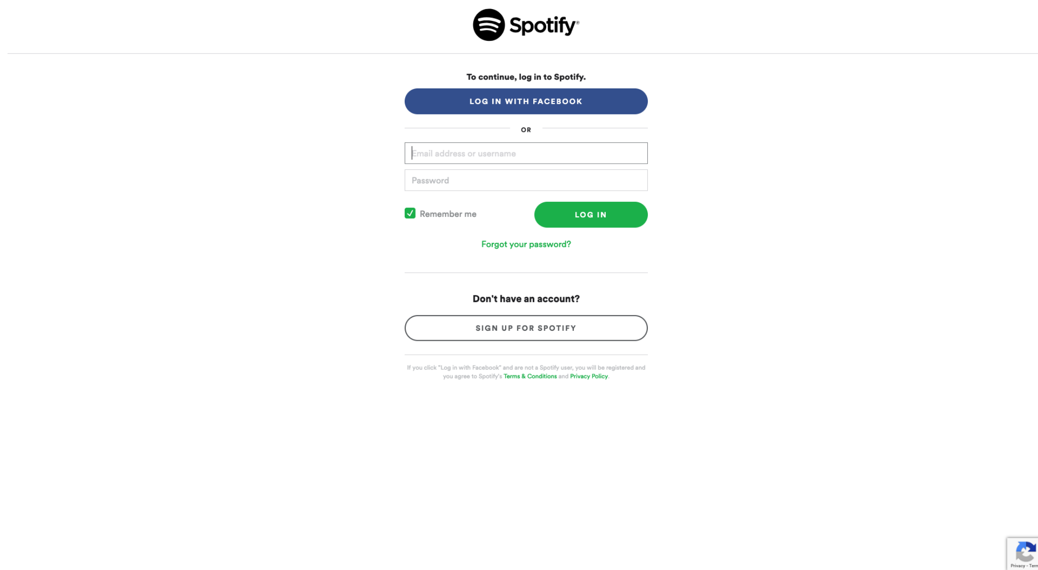

Figure 16: Spotify authentication screen.

Once the Spotify account has been authenticated, the user is returned to the Smoothie website which is now authenticated/logged-in using the Spotify account of the user. The logged-in state is indicated by both the fact the user is now presented with the input form and an element in the navigation bar to show their account details, as seen to the right side of Figure 17.

Figure 17: Smoothie navigation bar when user is authenticated.

With all above information taken into consideration, this requirement has

been met.

6.1.2 Playlist Input

The system must accept two pre-existing Spotify playlist URIs and retrieve the necessary data related to them.

Once the user has logged-in to the Smoothie website, the main call-to-action will be the field to input their first playlist. The design of this interface was kept very minimal to make it as clear as possible to the user what needs to be done and mitigate a large portion of potential errors. Both playlist inputs fields/interfaces are identical in design (with only copy changing) as once the user has successfully learnt the process for the first playlist, they can quickly replicate it for the second without having to learn or navigate anything new. The playlist inputs can be seen in Figure 18.

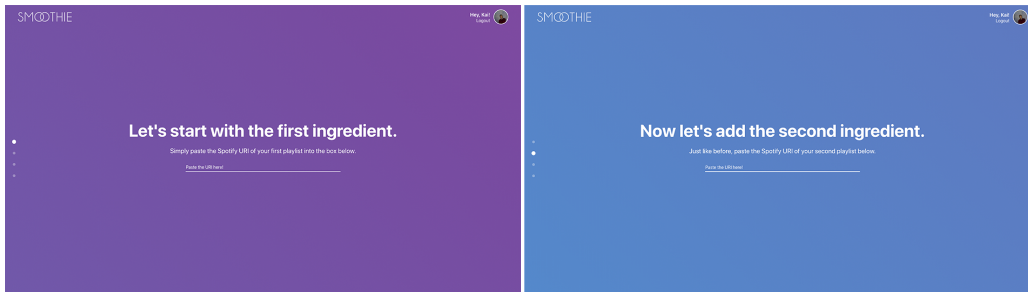

Figure 18: Smoothie playlist input interfaces.

For the remainder of this section all figures will be of the first playlist input interface - the design and user interaction process is identical so further duplicate screenshots would be redundant.

When a user pastes a Spotify URI for an existing playlist on the platform a preview will 'immediately' be displayed, as seen in Figure 19. This preview is displayed to show brief information about the playlist (name, author, no. of tracks, first three tracks, and thumbnail image) and thus allow the user to recognise if the URI they provided is in fact the playlist they wanted.

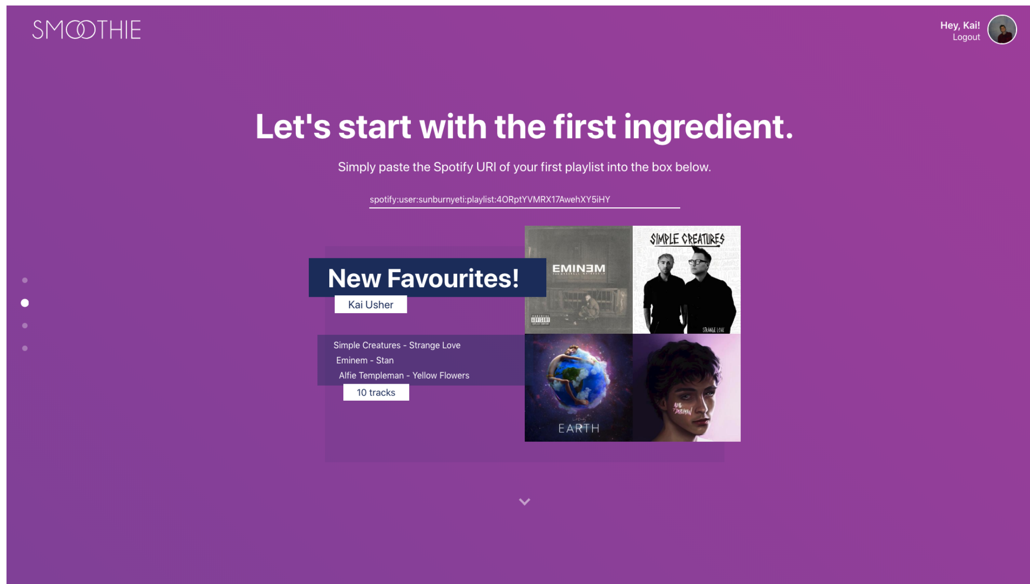

Figure 19: Smoothie playlist preview.

As discussed further in the Implementation section, this playlist is then retrieved from the Spotify platform and stored in the system. If the object returned by the Spotify API call (the object stored in the system) is logged to the console, the information shown in Figure 20 is printed; therefore, this requirement is met as the playlists are accepted and the necessary data is retrieved and stored.


```
▾Object
   collaborative: false
   description: ""
 ▸ external_urls: {spotify: "https://open.spotify.com/playlist/4ORptYVMRX17AwehXY5iHY"}
 ▸ followers: {href: null, total: 0}
   href: "https://api.spotify.com/v1/playlists/4ORptYVMRX17AwehXY5iHY"
   id: "4ORptYVMRX17AwehXY5iHY"
 ▸ images: (3) [{…}, {…}, {…}]
   name: "New Favourites!"
 ▸ owner: {display_name: "Kai Usher", external_urls: {…}, href: "https://api.spotify.com/v1/users/sunburnyeti", id: "sunburnyeti", type: "user", …}
   primary_color: null
   public: true
   snapshot_id: "MTcsZWRmNzE3OGM5N2NlZTU0OTcxZDhhN2NiN2UxODAxMzk4YmQzNDI1Mw=="
 ▸ tracks: {href: "https://api.spotify.com/v1/playlists/4ORptYVMRX17AwehXY5iHY/tracks?offset=0&limit=100", items: Array(10), limit: 100, next: null, offset: 0, …}
   type: "playlist"
   uri: "spotify:playlist:4ORptYVMRX17AwehXY5iHY"
 ▸ __proto__: Object
```


Figure 20: Output displayed in the console when logging a playlist object.

6.1.3 Playlist Options

The system must present the user with options to manage the playlist which will be generated.

Following the input of both of the playlists, the next section the user is directed to is the settings interface, as seen in Figure 21. Here they can input a name for the generated playlist, choose whether the playlist will be public or private and select the number of tracks they want to be added to the playlist.

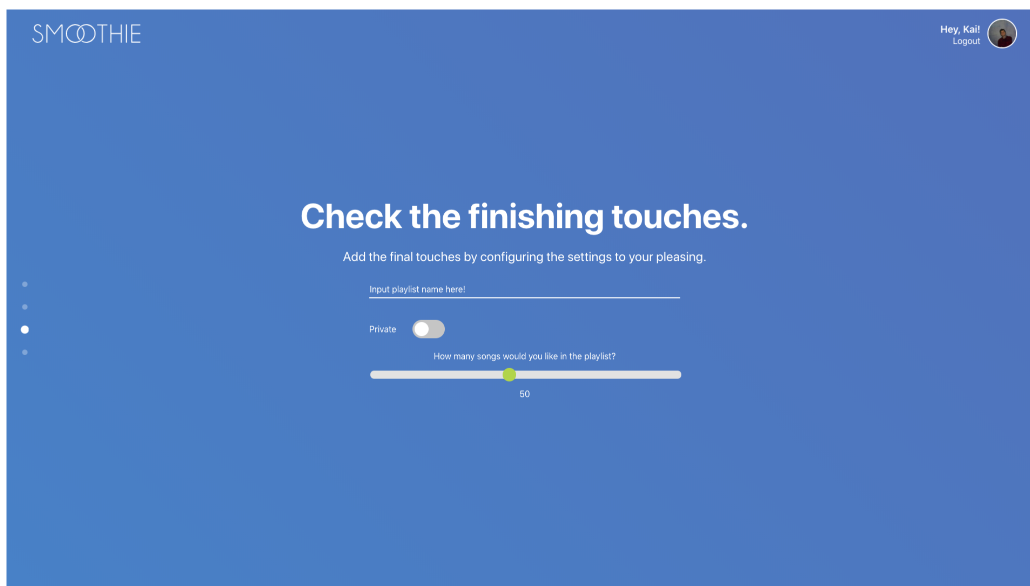

Figure 21: Smoothie settings interface.

With the above information taken into account, this requirement has been met.

6.1.4 Machine Learning

The system must use machine learning to train neural networks on the contents of each playlist - i.e. understand the playlist author's music taste.

For full details and evidence of the machine learning implementation refer to Section 5.4 of this report - the following will summarise the contents of that section.

The system utilises a Random Forest Classifier trained on the details of the two provided playlists to gain an understanding of each owner's music taste. The random forest is successfully able to build a model that allows for the classification of tracks into two categories - owner 1 or owner 2. Proof of this can be seen in Table 5, where the model was trained using various different networks and the resulting accuracy of the model recorded.

PL1 Genre	PL2 Genre	Similarity	Accuracy
Ambient	Alternative	Low	100.0%
Country	Folk	Medium	87.5%
Alternative	Rock	High	80.0%
Pop	K-Pop	High	80.95%
Disney	Disney	Identical	4.76%

Table 5: Results of testing the accuracy of the classifier.

From the results seen in Table 5, there is a clear negative correlation between similarity of the two playlists and the accuracy of the classifier - as the similarity increases, the accuracy decreases. This is to be expected and therefore the machine learning implementation is working well. When the playlists are very similar to one another it should be hard to determine the owner of a given track that would fit in the playlists, and when the playlists are mutually distinct it should be a simple task.

Taking the above information into account, this requirement has been met.

6.1.5 Prediction

The system must be able to accurately predict if a track fits the playlist a network was trained upon.

Given a track, the system is able to make a prediction about whether it is most likely to belong to owner 1 or owner 2. Further information about how this functionality is implemented can be found in Section 5.4, Machine Learning Implementation.

In order to test the functionality of the prediction ability, a number of playlists focused around certain genres were made and the machine learning system trained using them. For example, if owner 1 were to provide a playlist made up of mostly tracks from the classical genre, and owner 2 provided a playlist made up of rock tracks, when a classical track is provided to the system it should predict that owner 1 will like the track more. This scenario was carried out with varying genres to test the success of the machine learning implementation/predictor.

Below are the results of testing the system using a classical playlist and a punk-rock playlist.

Number	Playlist Name	Length	Genre
1	SmooothieTest - Classical	48	Classical
2	SmoothieTest - Punk/Rock	49	Punk-Rock

Table 6: Details of two playlists used for testing the prediction.

Each of the playlists outlined in Table 6 were generated using the Spotify API. By passing a request to the API's recommendation endpoint with just a single genre seed, the resulting return object is a list of tracks from the given genre. Generating test playlists like this meant I was guaranteed to have playlists composed of tracks that perfectly fit specific genres.

Table 7 shows the result of passing various tracks into the prediction algorithm. Each prediction is comprised of the probability values for the

track belonging to each owner, with the higher value being assigned to the owner which is more likely to enjoy the track.

Track Name	Expected Pred.	Actual Pred.	Probability
Canon in D, P.37	1	1	[1.0, 0.0]
The Thieving Magpie	1	1	[0.86, 0.14]
Leave It Alone	2	2	[0.02, 0.98]
Miserable At Best	2	BOTH	[0.59, 0.41]
Tabula Rasa: I. Ludus	1	1	[0.98, 0.02]
How's It Going to Be	2	2	[0.17, 0.83]
The Scientist	BOTH	BOTH	[0.59, 0.41]
Beast and the Harlot	2	2	[0.02, 0.98]
No Surprises	BOTH	BOTH	[0.62, 0.38]
Infra-Red	2	2	[0.0, 1.0]

Table 7: Results of testing the prediction system with various tracks and two distinctly different genre playlists.

From this test alone it shows that the system is exceptionally good at determining the owner of tracks. However, it must be taken into consideration that this test used two highly distinct playlists where it is relatively easy to determine the difference between a classical track and a punk-rock track; one is almost entirely instrumental, which is consequently why acoustic tracks (such as 'Miserable At Best') are predicted to be owned by both playlists.

Conducting a similar test using the punk-rock playlist as both input

playlists yields considerably different results, as shown in Table 8.

Track Name	Expected Pred.	Actual Pred.	Probability
Want You Bad	BOTH	BOTH	[0.4302, 0.5697]
1985	BOTH	BOTH	[0.4542, 0.5457]
The Phoenix	BOTH	2	[0.3534, 0.6465]
Skate or Die	BOTH	BOTH	[0.4097, 0.5902]
So Wrong	BOTH	BOTH	[0.4142, 0.5857]
Undercover Martyn	BOTH	BOTH	[0.4874, 0.5125]
So Far Away	BOTH	1	[0.6517, 0.3482]
Bent but Not Broken	BOTH	BOTH	[0.4564, 0.5435]
Centuries	BOTH	BOTH	[0.4895, 0.5104]
Last Resort	BOTH	BOTH	[0.3967, 0.6032]

Table 8: Results of testing the prediction system with various tracks and two playlists of the same genre.

The system is unable to clearly predict an owner for almost all of the tracks shown in Table 8. However, this is not a bad thing, and as shown by the 'Expected Pred.' column, the system is working as intended. The prediction should not just classify tracks in a mutually exclusive fashion, but instead be able to predict when a track will be liked by both owners as these are the ones Smoothie aims to highlight in the generated playlist.

With the above information taken into account, this requirement has been met. If a given track distinctly fits into a single owner's music taste/playlist the system will accurately predict so, and if a track's owner cannot distinctly be predicted the system will predict that both owner's will enjoy it.

6.1.6 Recommender System

The system must utilise a recommender system to recommend tracks that fit into both playlist authors' music tastes.

By making use of the Spotify Web API 'Get Recommendations based on Seeds' endpoint, the Smoothie system is able to recommend tracks based on the characteristics that are present in the two provided playlists.

The first component that makes up the seeds used to get the recommendations is the genres. To test that the genres were being accurately determined I took a number of existing playlists curated by Spotify that supposedly represent specific genres. Table 9 shows the results of this test.

Playlist Name	Genre	Smoothie Determined Genre(s)
Essential Pop	Pop	['dance pop', 'pop', 'canadian pop', 'hip hop', 'brostep']
The Rock List	Rock	['english indie rock', 'modern alternative rock', 'brighton indie', 'garage rock', 'indie rock']
Classical Essentials	Classical	['classical', 'baroque', 'american modern classical', 'american contemporary classical', 'compositional ambient']
Essential K-Pop	K-Pop	['dance pop', 'k-pop', 'j-pop', 'unknown', 'k-indie']
Massive Dance Hits	Dance	['big room', 'dance pop', 'deep groove house', 'edm', 'house']

Table 9: Results of testing Smoothie's ability to determine the genre(s) of a playlist.

As discovered, and explained further in Section 5.5, the genres that can be used for recommendations are not equal to all of the genres within the

Spotify platform. Therefore, Smoothie must take the genres of a playlist and transform them into genres that can be used as seeds for the recommendations API endpoint. Using the same playlists in Table 9, the genres were then passed through the algorithm to transform them into usable genre seeds - the results are shown in Table 10.

Playlist Name	Smoothie Determined Genre(s)	Seed Genre(s)
Essential Pop	['dance pop', 'pop', 'canadian pop', 'hip hop', 'brostep']	['dance', 'pop']
The Rock List	['english indie rock', 'modern alternative rock', 'brighton indie', 'garage rock', 'indie rock']	['indie', 'rock', 'alternative', 'garage']
Classical Essentials	['classical', 'baroque', 'american modern classical', 'american contemporary classical', 'compositional ambient']	['classical', 'ambient']
Essential K-Pop	['dance pop', 'k-pop', 'j-pop', 'unknown', 'k-indie']	['dance', 'pop', 'k-pop', 'j-pop', 'indie']
Massive Dance Hits	['big room', 'dance pop', 'deep groove house', 'edm', 'house']	['dance', 'pop', 'groove', 'house', 'edm']

Table 10: Results of transforming actual genres into usable genre seeds.

From the results of this test it is clear that the algorithm to transform the genres works, but could definitely be improved. Whilst the 'main genre' of each playlist is present in the genres that are output from the algorithm, it is often not the first genre in the list. This becomes a problem when selecting the genres to use for the seeds in the call to the API.

As the API can only accept 5 total seeds, which I chose to split into 2 genre seeds and 3 artist seeds, only 2 of the genre seeds produced by the algorithm can be used. With two playlists, this means one genre to represent

the playlist. The methodology of choosing a single seed by taking the first item in the list results in a relatively representative genre for the playlist, however there is still room for improvement. Using the results in Table 6.1.6, it can be seen that in the case of K-Pop, for example, the chosen seed will be 'dance', even though 'k-pop' is present in the list and would be a better choice; this is the same for the Pop and Rock playlists, which would be represented as 'dance' and 'indie' respectively.

One potential fix to this problem would be to implement a function/logic that takes each of the possible genre seeds and compares it against the name of the playlist to see if the genre seed is a sub-string of the playlist name - if it is, that genre seed should take priority. Of course this is not a catch-all solution though. Whilst a lot of playlists are named in relation to the genre they represent, there are a significant amount that are not, and therefore this method would not work for them.

Another solution could be to order the list of potential genre seeds based on the amount of times that genre appears as a sub-string of the actual genres of the playlist. For example, using the 'The Rock List' playlist seen in Table 6.1.6 and Table 6.1.6, when logging the process of determining what seeds to use (as explained in Section 5.5) the following output can be seen.

```
   GENRES TO GENRE SEEDS
english indie rock  is usable in category  indie
english indie rock  is usable in category  rock
modern alternative rock  is usable in category  alternative
modern alternative rock  is usable in category  rock
brighton indie  is usable in category  indie
garage rock  is usable in category  garage
garage rock  is usable in category  rock
indie rock  is usable in category  indie
indie rock  is usable in category  rock
```

Tallying up the possible genres seeds you get: rock(4), indie(3), alternative(1), garage(1). Therefore, in this example, taking the most common choice would result in the seed genre being the most representative of the playlist. Much like the other solution this is not always the case though;

with the example of the K-Pop playlist, the totals are: pop(3), dance(1), k-pop(1), j-pop(1), indie(1).

Following the results discussed above, I chose to adapt the recommendations algorithm to use the logic of tallying up the possible genres and selecting the most frequent. Whilst it doesn't solve all of the problems mentioned, it does improve the algorithm to be more representative of the genre of the playlist in question.

Once the seeds to be used for recommendations have been determined, the recommendations themselves are produced via a call to the Spotify API. As a recap, the aim of the project was to produce a playlist which appeals to both user's music tastes. In the initial plan this was to be achieved by adding tracks which could not be classified to a mutually distinct owner to a new playlist. As the project progressed, slight alterations to the plan meant that the new goal was to generate a playlist which contained a good mix of tracks that appealed to each owner and tracks that appealed to both. The explanation for this follows.

When testing the finished system it immediately became clear that there were some noticeable patterns with the generated playlists. In summary, the higher the degree of difference there was between the two input playlists, the lower the number of tracks that appealed to both owners were in the generated playlist.

Table 11 shows the results of generating a range of playlists (all with a length of 30 tracks) with input playlists of varying degrees of similarity.

Even in the situation where the playlists are very dissimilar, the generated playlist will still have a good combination of tracks which appeal more the owner 1 and tracks which appeal more to owner 2. In a more ideal situation the system would be able to produce a playlist totally made up of tracks that appeal to both owners, but due to both time limitations and API restrictions I chose leave the implementation as is. Due to the fact it is very rare that a user-created playlist is strictly one genre, and is being paired with another playlist that is also strictly one (contrasting) genre, this is not a problem.

For example, when conducting similar tests on random playlists that both I and my connections on Spotify have created, the results are far more reliable; out of five tests, the average number of tracks that were predicted to

PL1 Genre	PL2 Genre	Similarity	No. of joint recommendations
Rock	Rock	Very High	29
Ska	Ska	Very High	26
Alternative	Emo	High	10
Emo	Rock	High	8
Pop	Dance	Medium	8
Country	Folk	Medium	7
Jazz	Classical	Medium	5
Classical	Sleep	Medium	2
Metal	Soul	Low	5
Dance	Focus	Low	4

Table 11: Results of testing the recommender system with playlists of varying similarity.

appeal to both users was 14 - almost half of the tracks in the playlist.

As discussed in Section 5.5, the recommendations endpoint of the API was not as flexible as I was initially hoping. I chose to stick with the Spotify API largely due to the fact it integrated so well with the existing system I had developed and allowed me to focus on building a complete system in the time frame I had to work with. If I had used another data set and built a standalone recommender system from the ground up I would have had to find a way to translate the Spotify track attributes/features into a form that could be understood by another music database, for what is most likely a minimal improvement in the recommended tracks.

With all things considered, I still believe that this requirement has been met due to the fact the system does recommend tracks which fit into the

music tastes of the playlist owners - however, the recommendations do have room for improvement.

6.1.7 Playlist Creation

The system must generate a playlist of recommended tracks (with the previously input options) and save it to the authenticated user's Spotify account.

Once all of the inputs have been provided, the classifier trained, and the recommender system has produced a list of tracks, the system then creates the final product - the playlist.

The playlist will have the same number of tracks as the user requested in the settings input section and be titled with the name they provided. As soon as the generation process is completed the playlist is able to be viewed on both the Smoothie website and a Spotify application, i.e. the Spotify desktop client.

An example of a generated playlist can be seen in Figure 22. This same playlist was also saved to my Spotify account right away, meaning that I (as the user) could view it outside of Smoothie - Smoothie is simply the tool to generate the playlist and the result, as per the requirement, is saved to the user's Spotify account and not the Smoothie system. Figure 23 shows the same playlist in the Spotify desktop client.

Taking the above into consideration, this requirement has been met.

6.2 Project Objectives

In Section 1.2.1 I set out the initial objectives of the project as a whole. In this section I will briefly cover whether these objectives were accomplished or not.

1 - *Conduct research on the subject area and compile findings into a literature review.* This objective was completed at the beginning of the project and can be found in Section 2.

2 - *Conduct further research into what related solutions are already available and compare how they differ from this project.* This objective was com-

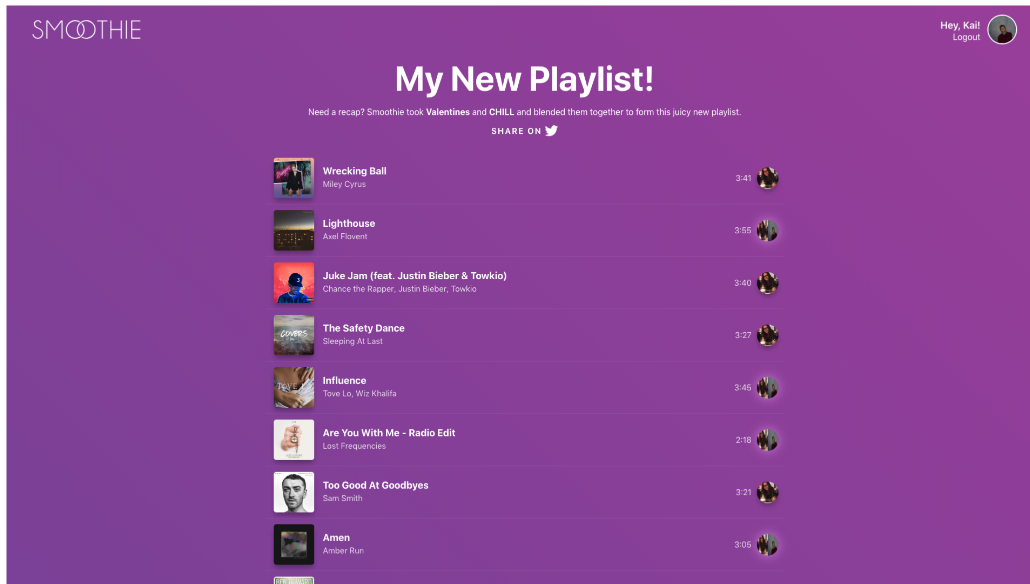

Figure 22: Smoothie generated playlist interface.

pleted as a subsection of the literature review, with numerous different solutions being researched and tested to determine their similarities to the system I was proposing. Section 2.3 covers this objective.

3 - *Design a web interface in which a user can log in, provide two playlists and view the playlist which is generated by the system.* Due to the agile and rapid development nature of the project I did not strictly design the interface as a standalone objective. As I knew the development would be an iterative process with many prototypes being built along the way, I chose to skip producing high-fidelity designs and focus on implementation as I believed this would be a better use of the limited time I had. I did however sketch out the initial designs on paper to ensure that the system would have all of the features required to be a success.

4 - *Develop the web interface defined in the previous objective.* Section 5 of this report covers how this objective was achieved. The web interface was developed to a high standard and is a major component of enabling Smoothie to be a successful project.

5 - *Develop a system using the Spotify API and machine learning which takes the playlists provided, gathers all the required data from the playlists, trains a neural network and creates a new playlist using classification and a*

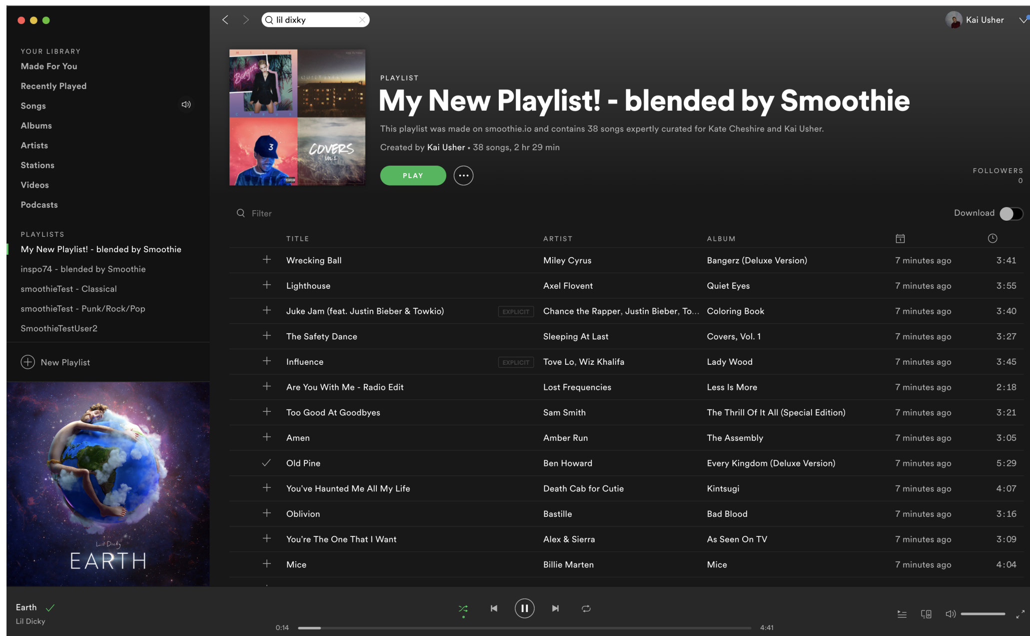

Figure 23: Spotify desktop client showing the generated playlist.

recommender system. Much like objective 4, the completion of this objective is explained in Section 5. The machine learning and recommender system of Smoothie was a resounding success, producing unique and representative playlists that are in line with the criteria defined throughout the planning stage of the project.

6 - *Comprehensibly test the system to ensure it is bug-free and fully functional.* Due to time limitations, this is the only objective that was not totally completed by the end of the project. As Agile software development is highly iterative, every time a new feature was added to the system a range of manual tests were completed to ensure that it functioned as expected before moving on to the the next stage of development. This meant that testing was being carried out at all time throughout the development cycle, however it could have been significantly more comprehensive and effective with a few minor changes - such as the implementation of automated testing.

7 - *Write a report on the result of the project and evaluate whether the aim has been met when judged against the criteria of the objectives.* This objective has been achieved as the report covers all aspects of the project

from the initial planning to the review of the final product in sufficient detail and to a high quality.

7 Discussion

7.1 Personal Development

Throughout the project I have gained new, and developed existing, skills across the entire software development and project management spectrum. The skills required included research for the literature review, planning a long-term project, designing a usable and engaging system and development throughout the whole technology stack.

Prior to starting the project I had little experience in almost all of these areas. Whilst I had good knowledge and experience with programming, I had no background with Python, React or interfacing with an API - the main development technologies used to build the project. Therefore, the development of the system drastically improved my skills in all areas spanned by the project and acted as a significant 'eye-opener' to what could be achieved if such skills and experience were to be applied to future projects.

Whilst there is truth in the fact that the project could have, and would have, been more successful had I chosen to utilise skills I was already equipped with (for example, use programming languages/frameworks I already know) due to the fact less time would have been spent learning and experimenting, I believe the right choice was still made. With the exception of automated testing, the project achieved all goals it set out to achieve and I have personally gained a lot more from the process than would have otherwise been possible.

7.2 Project Management

The use of Trello as a project management tool was highly effective in ensuring all component tasks of the project were set out and tracked in a meaningful way. Using the MoSCoW prioritisation system alongside the tasks meant that even from a quick glance it was clear what work needed to be completed next and how many tasks were yet to be started.

One way Trello could have been made more effective was by including either, or both, a time estimate for how long it would take to complete a task or a set deadline for each task to be completed by. This would have meant that I had a clearer idea of how much time was required to implement

features and therefore could have potentially managed my time better to ensure that all planned features made it into the 'final' product.

In hindsight, GitHub was also not utilised to its full potential. I essentially used Git to act as a back-up medium with version control functionality. Issues should have been made to reflect the tasks set out on Trello, and feature branches could have been utilised to better segment periods of development. Whilst this wasn't impactful in the success of the project (and probably saved time by not doing so) it would have been good industry standard practice to aid further personal development.

7.3 Future Improvements

Whilst I would consider the execution of the project a success, there are a number of future improvements that could be made to result in a more polished and consumer-ready website/application.

7.3.1 Automated Testing

Due to time limitations, I was unable to implement the automated testing that was present in the original plan. Having completed manual tests throughout the development of the system and upon completion of the final product, I can confirm that the system works as intended and there are no feature-breaking bugs, however, automated testing would have saved a significant amount of time and effort which could have been used elsewhere in the development cycle.

Another advantage of adding some form of automated testing is that future additions to the system are less likely to introduce potentially feature-breaking bugs. For example, and in theory, should someone other than myself work on the Smoothie code base, if automated-testing in the form of code coverage governed whether or not alterations could be commit to the Git repository, someone with less knowledge of the inner-workings of the system would be restricted from causing significant problems.

7.3.2 Recommender System

As mentioned in Chapter 5, Section 5 (5.5 Recommender System), the recommender system could be improved by changing how the genres used as the

recommendation seeds for the API call are selected. The substring method currently implemented works in the large majority of cases, but should two niche playlists be input the list of recommended tracks is unlikely to accurately reflect the music tastes present in the original playlists.

To solve this, a significantly more complex directory of genres would have to be compiled which includes what genres relate to what other genres, and therefore a path from one genre could be traversed until a genre that is present in the list of possible genres for recommendations seeds is reached.

On the other hand, this issue is one that cannot truly be fixed or improved upon noticeably unless Spotify change the way recommendations are accessed through the API. With only 126 possible genre seeds, anything outside of those will never be truly represented by the recommendations. The recommendations endpoint of the API is relatively newer compared to the other functionality so could potentially see updates in the future, however it is unlikely Spotify would provide the same level recommendations seen in their platform through the API as to do so would be giving up a huge portion of the functionality that can be cited for their success.

Another improvement that would benefit the recommender system is the addition of more user control over the parameters that shape the recommended tracks. In its current state, the system uses a single genre from each playlist, and the top three artists from both combined. A future improvement would add more settings to the system that, for example, allow the user to choose the playlist to be built using only tracks recommended based on the genres, or exclude certain genres from the recommendations. These settings would result in an even more personalised playlist and therefore a better end result.

Finally, as shown in Section 6.1.6, the system results in a playlist that is a combination of tracks for owner 1, tracks for owner 2, and tracks for both owners. In an ideal situation, the system will generate a playlist composed of only songs that appeal to both owners. This could be achieved by storing the tracks that appeal to both users in a temporary variable after the initial call to the recommendations endpoint, making another call and appending the next set of tracks that appeal to both owners to the temporary variable, and repeating this process until 50 tracks that appeal to both owners are stored in the system. This variable can then be used to return a list of tracks to

be added to the playlist that completely satisfy the condition of generating a playlist to appeal to multiple user's music tastes.

One problem with this approach is particularly clear when considering two playlists which are highly dissimilar and therefore the time it will take to produce a playlist of perfect recommendations starts to become far too high. I decided to experiment with this idea to see how much of an impact a potential 'improvement' of this manner could have.

I took two distinctly different playlists - one containing all punk-rock tracks, and the other containing all classical tracks - and requested Smoothie to use them to generate a playlist containing 100 tracks (the maximum playlist length). The results can be seen in Table 12.

Test No.	Generation Time (s)	No. of joint recommendations	Estimated total time (s)
1	47.5	6	791.7
2	47.6	6	793.3
3	47.17	14	336.93
4	49.46	12	412.17
5	48.12	12	401

Table 12: Testing the execution time to generate a 100-track length playlist with dissimilar inputs.

The estimated total time seen in Table 12 is calculated based on the proposed solution to keep making calls to the API until 100 joint recommendations have been retrieved. Whilst this is not a precise indication of how long the proposed solution may take it can still be used as a good indicator.

The upper-bound of the execution time was 13m 2s, and the lower bound was 5m 6s - the average estimated total time to generate a playlist of 100 perfect recommendations was 9m 1s. With the average execution time for a single call to the API (a single generation, as such) being 47.97s, this

equates to roughly the equivalent of 11.4 playlists being generated vs. 1 perfect playlist being generated.

Taking all of this into consideration, I believe the best option for a future improvement would be to find a medium between the current implementation and the implementation discussed above. For example, if repeat calls could be made until the proportion of the playlist which is comprised of joint recommendations is larger than the individual recommendations, this would result in a system that provides a better end result without compromising the user experience by introducing too much of a lengthy execution time.

7.3.3 Overall Website Functionality

Had time not been a concern for this project, I would have liked to add fully-functional user accounts. If a user was able to save playlists to a Smoothie account and therefore be able to return to the website to maybe edit them, share them, or perform some other functionality, it would most likely result in more return visitors. The ability to create an account creates a longer-lasting bond with a website that could have a positive impact on the user experience. Furthermore, the implementation shouldn't be too difficult as the user can continue to user their existing Spotify account, and from the development perspective some form of database would need to be added to record persistent data.

Smoothie in its current iteration is best described as a single-feature tool, but with more features and deeper functionality it could transition to an entire platform for music discovery and interaction.

Another feature that could aid this transition to a more fully-fledged platform would be the use of the Spotify Web Playback SDK [44]. This SDK, among other features, allows for the streaming of audio tracks in supported browsers. By implementing this functionality the user would essentially be able to preview the tracks in the generated playlist without having to navigate to the Spotify client itself, and therefore improving retention time on the Smoothie website.

By pairing the aforementioned playback functionality with features such as the ability to remove tracks from the generated playlist, reorder the generated playlist, and replace tracks on the generated playlist, for example, means the user can gleam significantly more value from Smoothie.

7.3.4 Mobile

Currently the system is only usable on desktop. Being a website, the functionality is still present when viewed from a web-browser on a mobile device, however the site is not visually optimised from a mobile perspective and no testing was carried out to assess the success of the system as a mobile site.

Had I have had more time towards the end of the project I would have like to put more focus on ensuring the system can be used successfully on mobile. This would involve designing a separate, albeit visually parallel, set of user interfaces and implementing them using media queries in a CSS file.

A further potential improvement would be to abstract the mobile version of the website into its own application, further aiding the transition into making Smoothie an entire music discovery platform.

7.3.5 User Research

At the core of any successful product is the satisfaction of the user, and that satisfaction is deeply rooted in the user experience. Throughout the project I've retained a good focus on designing and developing a system that provides a good user experience, and furthermore this is something I believe to have been successful. However, without input from real users of the system it is impossible to determine whether this is true or not.

Engaging potential users in some form of research workshop where they could get hands-on with the system, or producing multiple interfaces for the system to conduct A/B testing would greatly benefit the end-result of the Smoothie platform. Gathering data from consumers and turning it into executable adaptations would help to ensure the product is being developed in parallel with the expectation of the end users, and also ensure the final product meets the needs of those it is intended to be used by.

8 Conclusion

Prior to this project, there was no system available which could be used to allow two people to automatically generate a playlist that would fit both of their music tastes. The possibilities, most significantly creating a collaborative playlist manually, were time-consuming, assumptive and all-round not very usable.

Smoothie was developed with the aim of building a system that took the manual work out of collaborative playlists, and following its completion has met all requirements set out in the original plan. The system is presented in a simplistic web-application with a focus on stand-out user experience and ease-of-use, which resulted in a tool that can be accessed by almost anybody with zero tutorialisation needed. The use of a machine learning model trained on two individual music tastes results in playlists that would be difficult to produce based on manual assumption of what would be liked by both user's. Whilst there is still room for improvement in the recommender system, the results have been more than satisfactory with every generated playlist providing a good insight into how a playlist represents a user's musical preferences.

Therefore, the result of the project is a system that challenges the current approach to collaboration in regards to music consumption and proposes a novel way to recommend new music to user's of a platform such as Spotify.

The term 'information overload' is commonly used to describe the situation many people find themselves in when making their way through the hyper-connected world as we know it. Much like information overload, the situation in where we are presented with too many choices - overchoice or 'the paradox of choice' - can have a negative impact on our ability to perform optimally. With over 40 million songs in the Spotify platform [4], Smoothie provides a unique and effective way to explore new music that takes the anxiety of overchoice away from the user and returns results that would have otherwise been unobtainable.

References

[1] Spotify, "Spotify Financial Report Q3 2018," p. 1, Nov. 2018.

[2] C. Purcell, "With 50 Million Subscribers, Apple Music Still Lags Behind Spotify – For Now," accessed 2018-11-21. [Online]. Available: https://www.forbes.com/sites/careypurcell/2018/05/15/with-50-million-subscribers-apple-music-still-lags-behind-spotify-for-now/#e08c226557fd

[3] Spotify For Developers, "Web API," accessed 2018-11-21. [Online]. Available: https://developer.spotify.com/documentation/web-api/

[4] Spotify, "Company Info," accessed 2018-11-21. [Online]. Available: https://newsroom.spotify.com/companyinfo/

[5] Spotify For Developers, "Get Audio Features for a Track," accessed 2018-11-24. [Online]. Available: https://developer.spotify.com/documentation/web-api/reference/tracks/get-audio-features/

[6] J. F. McCarthy and T. D. Anagnost, "Musicfx: an arbiter of group preferences for computer supported collaborative workouts," in *Proceedings of the 1998 ACM conference on Computer supported cooperative work.* ACM, 1998, pp. 363–372.

[7] F. Ricci, L. Rokach, and B. Shapira, "Recommender systems: introduction and challenges," in *Recommender systems handbook.* Springer, 2015, pp. 1–34.

[8] Netflix, "Netflix Prize," accessed 2019-01-08. [Online]. Available: https://www.netflixprize.com/index.html

[9] R. Burke, "Hybrid recommender systems: Survey and experiments," *User modeling and user-adapted interaction*, vol. 12, no. 4, pp. 331–370, 2002.

[10] F. Ricci, L. Rokack, and B. Shapira, *Recommender Systems Handbook.* Springer US, 2015.

[11] Michael Tauberg, “Music is Getting Shorter,” accessed 2018-12-13. [Online]. Available: https://medium.com/@michaeltauberg/music-and-our-attention-spans-are-getting-shorter-8be37b5c2d67

[12] J. Masthoff, “Group recommender systems: Combining individual models,” in *Recommender systems handbook*. Springer, 2011, pp. 677–702.

[13] P. C. Fishburn, *The theory of social choice*. Princeton University Press, 2015.

[14] J. Masthoff and A. Gatt, “In pursuit of satisfaction and the prevention of embarrassment: affective state in group recommender systems,” *User Modeling and User-Adapted Interaction*, vol. 16, no. 3-4, pp. 281–319, 2006.

[15] A. Van den Oord, S. Dieleman, and B. Schrauwen, “Deep content-based music recommendation,” in *Advances in neural information processing systems*, 2013, pp. 2643–2651.

[16] Spotify, “Collaborative playlists,” accessed 2018-12-02. [Online]. Available: https://support.spotify.com/uk/using_spotify/playlists/create-playlists-with-your-friends/

[17] Spotify For Developers, “Developer Showcase,” accessed 2018-11-30. [Online]. Available: https://developer.spotify.com/community/showcase/

[18] Noon Pacific, “Mixtapes,” accessed 2018-12-02. [Online]. Available: https://noonpacific.com/

[19] Zach Hammer, “Playlist Souffle,” accessed 2018-12-02. [Online]. Available: https://playlistsouffle.com

[20] Rutger Ruizendaal, “Musical Data,” accessed 2018-12-02. [Online]. Available: https://musicaldata.com

[21] Michael Schwarz, “Klangspektrum,” accessed 2018-12-02. [Online]. Available: http://www.klangspektrum.digital

[22] Darrell Hanley, “Dubolt,” accessed 2018-12-02. [Online]. Available: https://dubolt.com

[23] Joel Lovera, “Magic Playlist,” accessed 2018-12-02. [Online]. Available: https://magicplaylist.co/

[24] José Manuel Pérez, “C-Listening Room,” accessed 2018-12-02. [Online]. Available: https://c-spotify.herokuapp.com

[25] Facebook Inc., “React - A JavaScript library for building user interfaces,” accessed 2018-12-09. [Online]. Available: https://reactjs.org

[26] Facebook, “React,” accessed 2018-12-09. [Online]. Available: https://github.com/facebook/react

[27] Node.js Foundation, “About Node.js,” accessed 2018-12-09. [Online]. Available: https://nodejs.org/en/about/

[28] Spotify for Developers, “Web API Tutorial,” accessed 2018-12-09. [Online]. Available: https://developer.spotify.com/documentation/web-api/quick-start/

[29] Armin Ronacher, “Flask (A Python Microframework),” accessed 2019-03-26. [Online]. Available: http://flask.pocoo.org

[30] Burke, Kevin and Conroy, Kyle and Horn, Ryan and Stratton, Frank and Binet, Guillaume, “Flask (A Python Microframework),” accessed 2019-03-26. [Online]. Available: https://flask-restful.readthedocs.io/en/latest/

[31] JMPerez, “spotify-web-api-js GitHub Repository,” accessed 2019-01-05. [Online]. Available: https://github.com/jmperez/spotify-web-api-js

[32] thelinmichael, “spotify-web-api-node GitHub Repository,” accessed 2019-03-25. [Online]. Available: https://github.com/thelinmichael/spotify-web-api-node

[33] Paul Lamere, “Spotipy,” accessed 2019-01-05. [Online]. Available: https://spotipy.readthedocs.io/en/latest/#

[34] pandas, "pandas - Python Data Analysis Library," accessed 2019-01-05. [Online]. Available: https://pandas.pydata.org

[35] David Cournapeau, "scikit-learn," accessed 2019-01-06. [Online]. Available: https://scikit-learn.org/stable/

[36] TensorFlow, "TensorFlow," accessed 2019-01-06. [Online]. Available: https://www.tensorflow.org

[37] Trello, "Trello," accessed 2019-01-07. [Online]. Available: https://trello.com

[38] Atlassian, "Kanban," accessed 2019-01-07. [Online]. Available: https://www.atlassian.com/agile/kanban

[39] GitHub, "Student Developer Pack - GitHub Education," accessed 2019-01-07. [Online]. Available: https://education.github.com/pack

[40] Spotify for Developers, "Authorisation Guide," accessed 2019-03-28. [Online]. Available: https://developer.spotify.com/documentation/general/guides/authorization-guide/

[41] Mozilla, "Cross-Origin Resource Sharing (CORS)," accessed 2019-04-21. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/HTTP/CORS

[42] scikit-learn, "Choosing the right estimator," accessed 2019-03-05. [Online]. Available: https://scikit-learn.org/stable/tutorial/machine_learning_map/

[43] Spotify for Developers, "Get Recommendations Based on Seeds," accessed 2019-04-06. [Online]. Available: https://developer.spotify.com/documentation/web-api/reference/browse/get-recommendations/

[44] ——, "Web Playback SDK," accessed 2019-04-23. [Online]. Available: https://developer.spotify.com/documentation/web-playback-sdk/