
UNDERDISCOVER: A MOBILE APPLICATION FOR CONTENT-BASED MUSIC DISCOVERY

Abstract

Unparalleled as the principal benefit of the digitalisation of music consumption, effortless discovery has revolutionised the way in which music is found and listened to. Online platforms such as YouTube and Spotify provide incredible amounts of music to users whilst being significantly easier to navigate in comparison to older approaches, such as record shops. As returns of improvement in the quantity of discovery dissipate, research and development focus should shift onto the improvement of discovery quality. UnderDiscover is a mobile platform built around the concept of content-based recommendations, a methodology often unfavoured in industry when compared to a collaborative filtering approach. Developed with the Spotify API at its core, UnderDiscover harnesses the rich source of data Spotify stores on its catalogue of music – one of the largest in the world – to provide users with a tailorable discovery experience based upon various musical attributes ranging from factual to mathematically derived. Prominently, UnderDiscover twists the tradition of recommending popular music on its head, prioritizing less popular music in the collaborative part of its recommendation algorithm. Allowing users to take on as much or as little control over the algorithm as they wish, UnderDiscover facilitates attribute choice giving the user the ability to find music truly similar to what they enjoy already. This paper explores the development process from start to finish, including a significant literature review and project specification, design process and methodology review. It goes on to analyse the implementation and testing process, before critically discussing the achievements made.

Declaration

I certify that all material in this dissertation which is not my own work has been identified.

By Morgan Centini

Contents

1	Introduction.....	1
1.1	Motivation.....	1
1.2	Aims & Objectives.....	1
2	Literature Review	2
3	Project Specification	4
4	Design	6
4.1	Pathways & Functionality	6
4.2	Recommendation Engine.....	7
5	Development Lifecycle	8
6	Implementation	9
6.1	Pathways & Functionality	9
6.2	User Interface	12
6.3	Recommendation Engine.....	14
6.4	API Interaction	15
7	Testing & Evaluation	16
7.1	Performance Evaluation & Iterations	16
7.2	Acceptance Testing.....	17
8	Critical Discussion & Conclusion.....	19
8.1	Critical Discussion.....	19
8.2	Conclusion.....	20
9	Appendices.....	iii
	Appendix 1: Primary Research on Importance of Musical Attributes.....	iii
	Appendix 2: streamIntoString Function Iterations.....	iv
10	Bibliography	v

1 Introduction

1.1 Motivation

Searching for new music has evolved in the past 50 years away from record shops to digital formats such as CD's and now streaming services. A study (IFPI, 2018) found that global average music consumption per week sits at 2.5 hours a day. It also forms a huge economic industry; Spotify disclosed an annual revenue of almost \$6 Billion in 2018, with total users standing at over 200 million in the fourth quarter of that year (Spotify Technology, S.A, 2018). Principally, platforms such as Spotify have altered exposure methods and quantities. During the days of vinyl records, shopkeepers would recommend fresh music they have recently taken into stock; now, modern music platforms provide recommendations to users on new music to listen to in the form of a recommendation engine.

Platform-curated recommendations often form a significant chunk of consumption; (Economic Times, 2018) established a figure of 15% for the Gaana music platform. These recommendations are often based upon user feedback – known as collaborative filtering. Whilst this is likely to result in a majority of listeners receiving desired results, it often works against users wishing to discover less popular music. Originally, this project was motivated by a personal desire to discover music based upon similarities in the content, rather than music popular in the genre I was listening to or in my geographical location. As a DJ, I have found that typical collaborative recommendations show me music I am already aware of.

Music tracks have attributes which aren't created in artificial big-data situations; for example, signal frequencies can be analysed. The Spotify API is a fantastic case study of this; it stores data on danceability, valence and energy to give some examples, as well as storing more traditional, collaborative attributes such as popularity (Spotify AB (2), 2019). These content-based attributes can be combined with collaborative ones in order to provide more interesting and demand-specific suggestions. Specifically, the algorithmic balance between content-based and collaborative filtering can be shifted away from collaborative attributes and towards content-based ones in order to achieve the desired goal.

Therefore, the problem which this project aims to address is a use case in which users wish to discover music based upon its content; and in doing so, enhance the ability of such users to discover music which is both more unknown to the user and more specific to their demands. This use case can be applied to more typical circumstances, such as hosts of a party wishing to create a playlist matching a specific atmosphere, as well as less typical ones, such as my motivation in finding content-similar music to DJ with. In doing so, this project will contribute a unique application to the Spotify

1.2 Aims & Objectives

The aim of this project is to create a platform to provide refined music suggestions using a recommendation engine based on a content-based filtering technique. This aim can be broken down into several key objectives:

- **Deploy an easily accessible usable mobile application for music discovery**
The platform should be built for a mobile environment; as a stretch goal this would be across multiple major platforms, but a viable product would be functional on one.
- **Create a novel recommendation algorithm with attribute-based refinement**
The platform should utilise data-rich music sources only, in order to provide users with the specific content-based experience outlined.

- **Allow user control over the level of cognitive load they are exposed to**
The platform should be built with several pathways for different 'levels' of user, stretching from 'basic' users whose exposure to the algorithm should be minimal, to 'advanced' users who should be able to manipulate the algorithm themselves.
- **Provide interaction with an audio playback application**
The platform should give the user the ability to take the recommendations back to an audio playback application, utilising common functionality such as playlists.

2 Literature Review

Recommendations engines for music are inherently a big data problem; a term defined by Sagioglu & Sinanc as 'data sets large, more varied and complex structure with the difficulties of storing, analysing and visualizing for further processes' (Sagioglu & Sinanc, 2013). Figures from the aforementioned IFPI study and Gaana platform (IFPI, 2018) (Economic Times, 2018) suggest that users listen to just 22.5 minutes of recommended music a day, showing the challenge in creating good concise recommendations. Volume and Veracity are more relevant than Variety and Velocity to my project; (Spotify, 2019) has over 50 million individual tracks stored and its generated attributes are subjective.

Data characteristics in music can be split into 'content-based' and 'collaborative' (Lin, et al., 2014). Content-based attributes, such as 'danceability', refer to the content of music, whereas collaborative attributes, such as 'popularity', denote user preferences. Content-based attributes can be split into factual (musical key) and subjective (energy). Subjective attributes are the result of complex analysis to generate comparative values; the above examples are results from the Spotify API (Spotify AB (2), 2019).

Existing literature on designing music recommendation engines spans from complex mathematical designs to linear, iterative ones. Kodama et al outline a hybrid engine balanced towards content-based filtering, the system defining a 'mood' of a song by extracting values including tonality and signal levels; they then proceed to ask a user to describe how the song makes them feel (Kodama, et al., 2005). Shifting the balance further towards content, Kaji et al discuss a system utilising a three-stage system to generate and refine playlists through generation, transcoding and data analysis (Kaji, et al., 2005).

The 'cold start problem' is a common issue facing recommendation engines, described as the 'need to accumulate personal information in advance'; industry leaders such as Spotify & Apple Music face this problem (Wang, et al., 2014) (Jenkins & Yang, 2016). Existing literature shows the requirement of engines to use a combination of both techniques. One method would be to store users' listening history to prevent repeat recommendations – O'Bryant proposes use of such data in making better recommendations (O'Bryant, 2017). Proposing use of 'modulation analysis to extract timbral features' in creating a content-based solution to the cold-start problem, Soleymani et al argue that use of attributes more complicated than genre provide better results for users (Soleymani, et al., 2015).

Multi-recommender solutions are commonly advocated for in literature. Ekstrand et al. show that users will use functionality to switch between multiple recommendation systems; specifically, they recommend that "other systems may also want to consider allowing the user to have more control over the way in which their recommendations are computed" (Ekstrand, et al., 2015). Further to this, Jin et al. found that users overwhelmingly followed their experience level when it came to tweaking algorithmic properties, with users who could tolerate the increased cognitive load of a higher level of user control "more likely" to do so (Jin, et al., 2017).

Academic literature, such as Celma's study of Amazon's recommendations (Celma, 2008), explore the tendency of collaborative-biased engines to favour 'popular' music. Celma finds that collaborative based engines often lack adaptability and create obvious suggestions. This supports the proposition that minority use cases are disadvantages by collaborative filtering.

Ensuring the application can reach the maximum number of potential users is paramount to success as a commercial project. Android & iOS devices make up >99.9% of all smartphone sales internationally (Gartner, 2018). Further, Android units make up approximately 86% of this (IDC, 2019). (Majchrzak, et al., 2017)'s analysis of contemporary cross-development platforms concludes that React Native would be the most suitable development platform for an application targeting both Android & iOS; specifically, its development community is strong in comparison to Fuse & Ionic.

(Danielsson, 2016) highlights the strengths of the React Native platform in comparison to native Android development in efficiency; stating that, from an example 'less then half of the amount of code was used [in writing the React Native Application]'. Despite this, he notes that 'there are some faults in React Native', as it is a relatively newer platform, although Danielsson's paper is 4 years old at the time of writing, making React Native just 1 year old at the time (Papp, 2017).

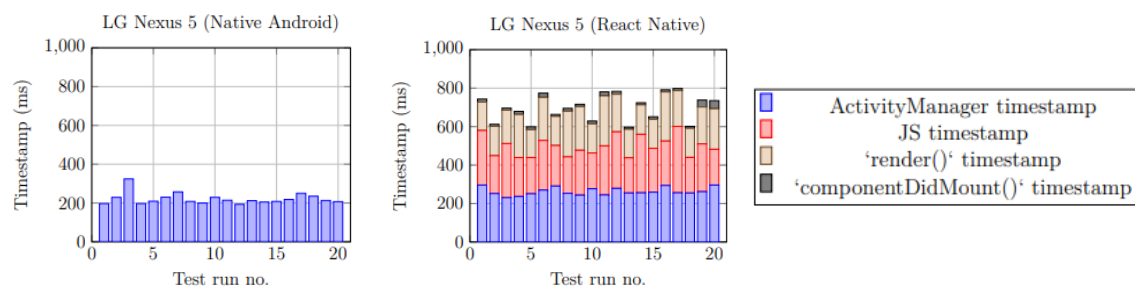


Figure 1 shows Native Android app start performance versus React Native (Eskola, 2018).

Performance differences between native Android and React Native development lends towards single platform development; (Eskola, 2018) highlights launch times as an area where React Native suffers, with applications 'load[ing] significantly slower than a native Android application' [1]. Eskola suggests the use of 'splash screens' in alleviating performance deficits, as well as stressing the increasing negligence of such effects as device performance increases.

3 Project Specification

I have broken my project requirements down into three distinctive categories: Minimum Viable Product (MVP), Desired Final Product (DFP) and Stretch Goal Aims (SGA). The MVP represents a prototype product which whilst not being ready for deployment would serve as a proof of concept of the project. The DFP represents the expected final product after the full development course. SGA's represent concepts to be subject to exploration during the implementation of the project; it is not immediately expected that they will be completed.

No.	Requirement	Product Category
Overarching System Requirements		
1	The system provides music suggestions to the user using recommendation engine biased towards content-based filtering.	MVP
2	The mobile application will function on the Android OS.	MVP
3	The mobile application will function on iOS.	SGA
User Access Control		
4	Users will be able to access the application's functionality when signed into a valid Spotify account.	MVP
5	The application will be function identically for users no matter whether their Spotify account is a free or premium account	DFP
6	Registered users will have their recommendation history stored by the system to prevent repeat suggestions.	SGA
Recommendation Engine		
7	The recommendation engine can take a song provided by the user and return a list of suggestions with no requirement for attributes to be specified by the user.	MVP
8	The recommendation engine can take a song provided by the user and return a list of suggestions using musical attributes specified by the user compared to the original songs.	DFP
9	The recommendation engine can take a song provided by the user and return a list of suggestions using musical attributes specified by the user and the percentage tightness with which these attributes match the initial songs.	DFP
10	The recommendation engine outputs suggestions to the user sorted by how closely they match the initial song.	DFP
11	The recommendation engine will weight the popularity of songs into their perceived match value, giving favourability to less popular music.	DFP
Pathways & Functionality		
12	The application will provide a single, clear pathway through the application which demonstrates it's basic functionality	MVP
13	The application will provide various pathways through the application, designed to suit various experience levels of user.	DFP
14	The 'search' function will, with use of just one or multiple pathways, provide for searching via tracks, albums and artists.	MVP
15	The 'request' function will provide at least one method for defining the algorithm with which suggestions will be made.	DFP
16	The 'result' function will allow the user to add suggested music into a new playlist linked to their Spotify account.	MVP
17	The 'result' function will allow the user to add suggested music into an existing playlist linked to their Spotify account.	DFP

User Interface (UI)		
18	The mobile application will provide a UI, in English, for the user to access the full functionality of the system.	MVP
19	The UI provided will be built to be flexible across numerous devices, maintaining structure on differing screen sizes.	DFP
20	The UI will allow users to view suggestions provided by the Music Recommendation Engine.	MVP
21	The UI will allow users to view results of search requests before any further action is taken on said requests.	MVP
22	The UI will provide the user with the values of all adjustable attributes in a clear and concise manner.	DFP
23	The UI will provide the user with optional informational popups describing each of the adjustable attributes.	DFP
24	The UI will provide users with a clear option to consume audio related to any tracks searched for by the user, or suggested by the system, before proceeding to take action on that track.	SGA
Performance		
25	The mobile application should present a 'splash' screen when launching.	DFP
26	The mobile application should take no longer than 1.5 seconds to transfer between pages not executing HTTP calls.	MVP
27	The mobile application should take no longer than 0.5 seconds to transfer between pages not executing HTTP calls.	DFP
28	The mobile application should take no longer than 5 seconds to transfer between pages executing HTTP calls (excluding generating recommendations).	MVP
29	The mobile application should take no longer than 3 seconds to transfer between pages executing HTTP calls (excluding generating recommendations).	DFP
30	The mobile application should take no longer than 20 seconds to transfer between pages executing HTTP calls to generate recommendations.	MVP
31	The mobile application should take no longer than 15 seconds to transfer between pages executing HTTP calls to generate recommendations.	DFP
32	The mobile application should take no longer than 10 seconds to perform an initial launch, including Spotify authentication.	DFP

4 Design

Designing the UnderDiscover application involved three key components; the recommendation engine and application pathways/functionality. The recommendation engine component focuses on both the contents of the novel algorithm and the presentation of its results. The design of application pathways outlines how the application will cater users of different experience levels and use cases, both typical and atypical.

4.1 Pathways & Functionality

The most important design decision in forming user pathways is separation of the request pathways into multiple sections based on user experience and/or ability. While a minimal viable product may only provide a single universal pathway through the request section, my desired final product will split the function pathway into two separate activities; 'basic' and 'advanced'. The 'basic' pathway will strip the majority of the determination from the user; users will not be able to specify the tightness to which recommendations must hug the original attributes, or be able to select multiple attributes at once. This pathway will fit a 'plug and play' user profile, where the focus may be on generating a large number of sufficient recommendations as opposed to a smaller set of very concise ones. Conversely, the 'advanced' pathway will hand the vast majority of control over determination to the user; attribute tightness will be user-determined and any number of attributes, or indeed none, will be selectable at once. This pathway will fit a user story more similar to the initial project motivation outlined in the introduction, where the user is a music professional.

Another important design decision when considering user pathways is the search functionality. Items stored in the Spotify API are broken down into types including album, artist and track (Spotify AB (3), 2020). One potential solution could be to have all items regardless of type returned if matched to the given text. This solution would cause two problems. Firstly, the returned results list would be filled with multiple objects, all of which require different user interaction before the user can proceed; this will cause issues with developing a consistent and reactive user interface for the results. Secondly, the merging of three sets of results into a single set will likely lead to very long lists of unconcise results.

An easy solution can be achieved by using an array of buttons below the search box as opposed to just one. Each button can specify a different item type in the query, making the results more concise whilst also making development of an appropriate UI easier. This will create a 'staggered' pathway, where the user can start at any of the stages (artist, album, track) of this segment of the application pathway but must complete all stages after the starting stage. For example, a user searching by track name will be able to immediately request metadata on that track. Similarly, a user searching by album name will be able to select an album from which a list of tracks will be presented to the user. However, a user searching by artist name will only be presented with a list of their albums; the API does not have an endpoint for getting an artist's tracks, just its albums (Spotify AB (4), 2020).

Considering the requirement for the user to be able to 'export' suggestions to an audio playback platform, or Spotify, playlist functionality is the most common and sensible method available. The Spotify API contains endpoints for both creating a new playlist (Spotify AB (7), 2020) and adding to an existing playlist (Spotify AB (8), 2020). Implementing this feature should not be technically challenging; the difficulties will lie in adding the functionality to a feature-heavy section of UI.

4.2 Recommendation Engine

Primarily, design of the recommendation engine involves selection of the attributes which are presented to users in the request pathway of the application. The Spotify API provides 14 attributes which can be modified for use in the 'Get Recommendations from Seeds' endpoint (Spotify AB (1), 2020). Modification of these attributes is the key component of the algorithm and thus they should be selected carefully and with regards to their usefulness.

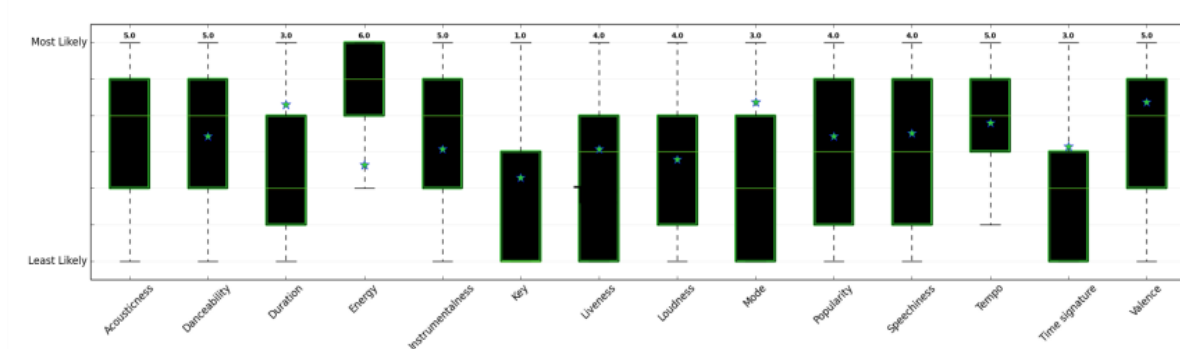


Figure 3.1 taken from (Millecamp, et al., 2018) showing attribute use likeliness.

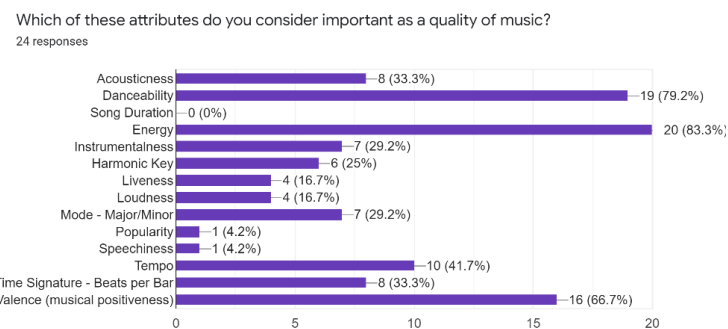
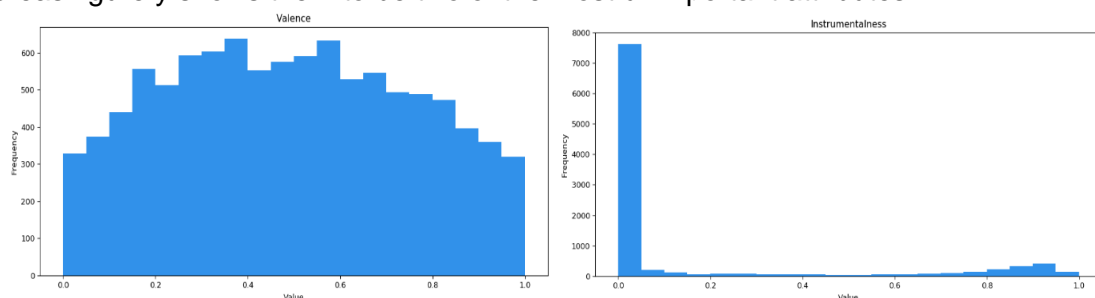


Figure 3.2 taken from primary research listed in Appendix 1; where respondents were from a ranged but heavily musical background and n=24.

The research shown in figures x and y show a consistent trend in regards to the most important attributes; energy, danceability, valence and tempo, as well as instrumentalness and acousticness to a lesser extent, are consistently favoured across Millecamp et al.'s research as well as my own. Findings are more sporadic in relation to the less important attributes; figure x shows that popularity and speechiness may also be important to users, whereas figure y shows them to be two of the most unimportant attributes.



Figures 3.3 and 3.4, showing the distribution of values for the 'Valence' and 'Instrumentalness' attributes respectively (Spotify AB (6), 2020).

There are technical considerations to make on top of consideration of user demand which may help make decisions on borderline attributes; for example, the distribution of values across the dataset for some attributes may not lend well to generating recommendations from them. We can see this in figures 3.3 and 3.4; valence is well distributed whereas instrumentalness is extremely condensed into the low end of the spectrum, making any recommendations based on valence many more time accurate than instrumentalness. This example can be applied to all available attributes and suggests that instrumentalness would be a poor choice to present to users, whereas acousticalness may provide meaningful results.

Whilst subject to testing and evaluation measures, my initial design will make use of the energy, valence, danceability, tempo, acousticalness, speechiness and loudness. Popularity will form a part of the algorithm; however, it will be used as a weighting agent within the algorithm as opposed to being one of the user-determinable attributes. Song duration, time signature, liveness and mode have been excluded as they fail to provide useful information on the content of the music. Key has been excluded as it is only useful to music professionals such as DJ's, and does not give any useful information to passive listeners.

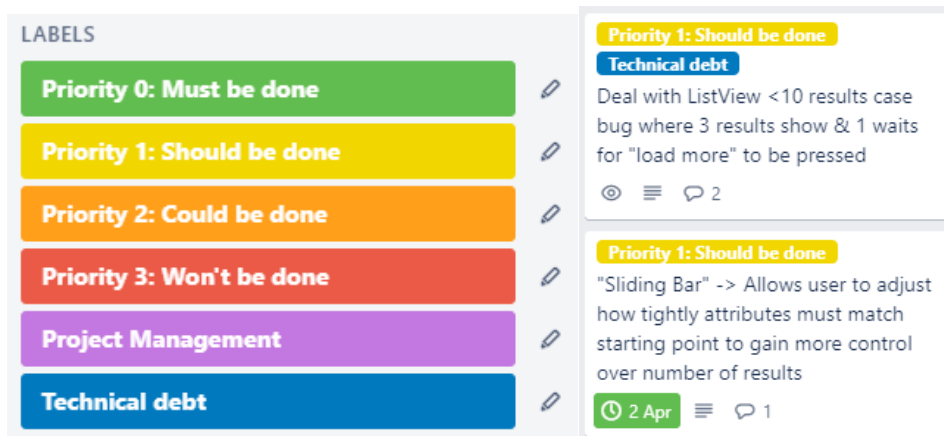
One key design feature is visualising the relationship between input tracks and recommended tracks; Millecamp et al's research on user interaction with musical attributes finds that "recommender must be able to [...] display recommendations together with a visualisation of the relationship between the input song and the recommended songs using the musical attributes" (Millecamp, et al., 2018). Some literature suggests more extreme visualisation solutions; (Bateira, et al., 2014) utilise "graph-like" maps visualising the relationships between artists in their RAMA system. Whilst informative, the benefits of such a feature stack would be small in comparison to the development investment it would require. My intended design for this feature is a collective percentage representing the algorithm calculation displayed with each of the recommended tracks; I considered showing individual attribute percentages but this would overwhelm less experienced users whilst more advanced users will be able to piece this data together themselves fairly easily.

5 Development Lifecycle

Delivering a software project efficiently, wholly and timely requires good development practices throughout. Two common lifecycle methods are used to achieve this: Waterfall and Agile. Waterfall is a traditional lifecycle method; linear in nature, it is a 'sequential [...] process [...] flowing increasingly downwards [...] through a list of phases that must be executed in order [...]', where 'requirements [should be] defined and analysed prior to any design or development' (Bassil, 2012) (Ruparelia, 2010). On the other hand, Agile is 'based on the idea of incremental and iterative development', dividing the lifecycle into 'smaller parts, called increments' (Leau, et al., 2012) (Beck, et al., 2001). This allows for changing requirements and creates deliverables more regularly than Waterfall. Often accused of being documentation-light, (Vijayasathy & Turk, 2008) claim code often acts as documentation.

Using project management tools helped to maintain order and effectiveness throughout the project. I heavily made use of Trello in order to keep track of ongoing and future tasks. My methodology for the Trello board consisted of having four key decks; Backlog, Started (Passive), Started (Active) and Completed. The backlog and completed piles contained tasks which had not yet been started but had been defined, and tasks that had been completed and manually tested respectively. The passive started pile contained tasks that had been started but were currently not being aimed to finish by the next scrum meeting. The active started pile contained tasks that were due to be finished by the next meeting.

Each card was written with a fully-descriptive title, an expected workload (referred to as EW) and labels defining its category of importance.



Figures 5.1 (left) and 5.2 (right), where 5.1 shows the labels used on the Trello board, and 5.2 shows typical cards used to maintain project structure (Centini (1), 2020).

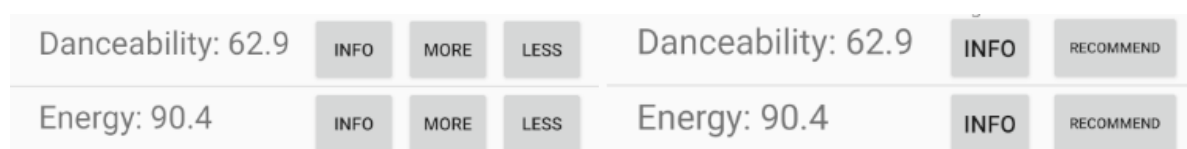
I also made use of GitHub in my project. The GitHub methodology followed fairly standard industry practises; commits were made to the master branch whenever a feature was completed or whenever a significant bug was fixed. Further, commits were made to the master when I moved between devices. Separate branches were used when development on features which required significant refactoring of existing code begun.

6 Implementation

Outlined in the design, the implementation of the UnderDiscover application involved four key components; the recommendation engine, application pathways, a user interface and interactions with the Spotify API. Each of the sections covers the aforementioned topics; the API interactions cover the justifications for HTTP calls to collect data to display within the application.

6.1 Pathways & Functionality

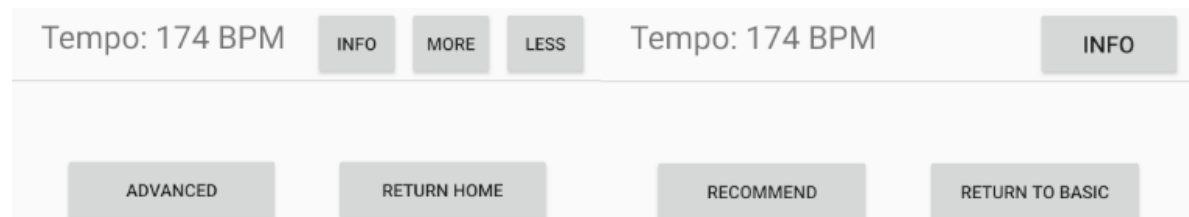
The first implementation of the 'basic' pathway in the separation of request functionality outlined in section 3.1 supplied the user with a simple 'recommend' button for each attribute. Whilst this was adequate, I felt that it would provide a confusing user experience for the desired 'plug and play' model; the recommendations would be relevant to the attribute selected, but when applied to the sorting algorithm would oscillate between lower and higher values of the selected attribute.



Figures 6.1 (left) and 6.2 (right) showing the first and final implementations of the 'basic' pathway for some attributes.

More appropriately, the final implementation offers the choice between recommendations consisting of tracks with attribute values higher or lower than that of the original track only. This implementation suits the 'plug and play' model more – for example, a user wishing to quickly obtain a playlist of tracks with a higher energy level than the current track can do so easily. Further, it does not disadvantage users who may have preferred the first solution; that functionality can be achieved by selecting a single attribute in the 'advanced' pathway.

The 'advanced' pathway of the request functionality has followed a largely unchanged development path in line with the design outlined in section 4.1. The user can switch between the two pathways using a footer button.



Figures 6.3 (left) and 6.4 (right) showing the footer buttons of the basic (6.3) and advanced (6.4) pathways respectively.

The 'advanced' pathway permits a far greater level of control over the algorithm; the user can select multiple, all, or even no attributes (an atypical case covered in section 6.3). The selection of attributes is handled by the user simply clicking on the attribute in the list which it wishes to include in its request; only highlighted attributes will be included.

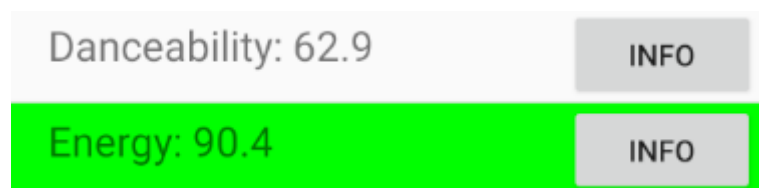


Figure 6.5 showing a highlighted attribute (energy) and a non-highlighted one (danceability).

The user also has control over the level of 'tightness' with which the matching songs' attribute values must adhere to in comparison with the original song. This is done using a sliding scale above the attribute listings; the scale defaults to 15%, and can be moved from 5% to 25%. There is little need for movement outside of these parameters; tightness values above 25% generally produce too many results to be accurate and values below 5%, although producing accurate results, generally produces too few results to be useful.

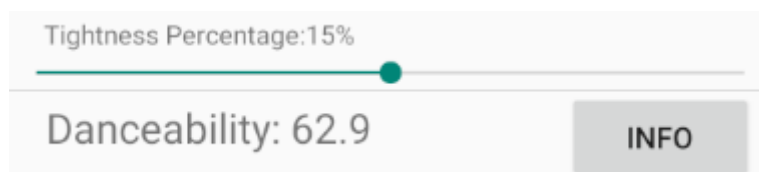
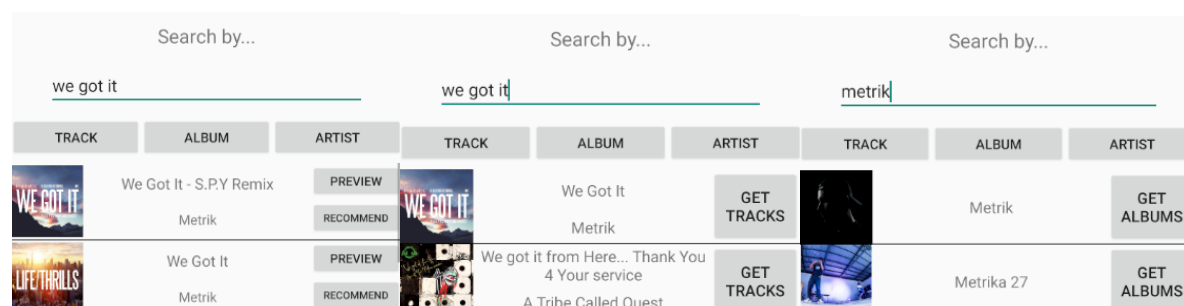


Figure 6.6 showing the tightness controller in relation to the attribute listings.

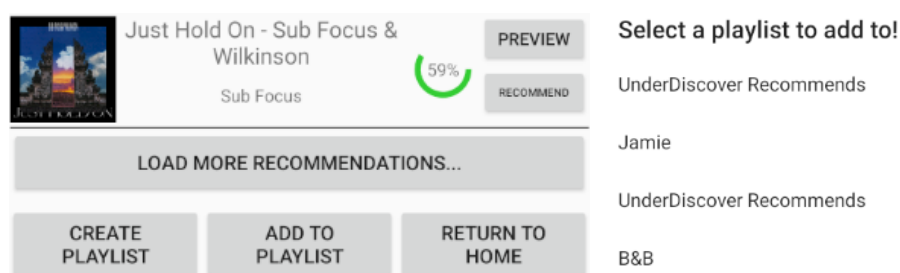
Outlined in the design is the ‘staggered pathway’ approach to handling the search function. This design has been followed in the implementation almost entirely. Users can search using three different buttons; track, album, and artist. The search box provided above is applicable to each of the three buttons. This system implements the ‘staggered’ nature discussed, where users may start at any point in the pathway but must then complete each following stage in the pathway.



Figures 6.7 (left), 6.8 (centre) and 6.9 (right) showing the results of the search function for track (6.7), album (6.8) and artist (6.9) searches respectively.

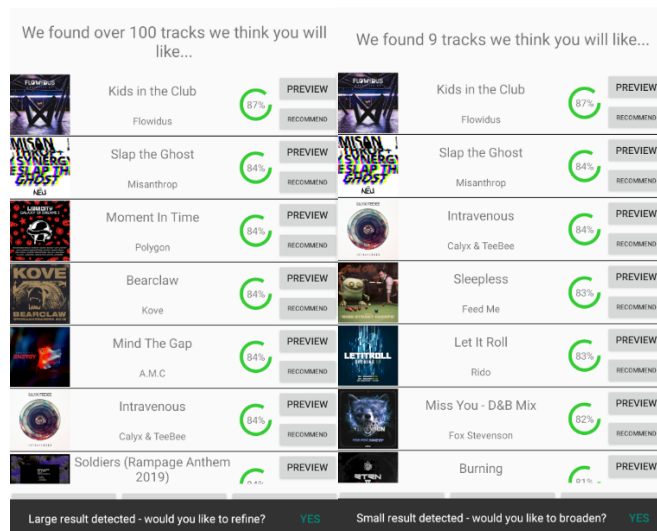
The difficulties in using a single button for this purpose can be visualised with the results given; the results would likely include ‘duplicate’ entries – for example, the ‘We Got It’ album shown in figure 6.8 only contains 3 tracks, all named ‘We Got It’ (including remixes). Including this album in the result stream when all 3 tracks would also be included would be inefficient and expensive for the applications performance. Despite this, there is a minor drawback in that each step you start behind the ‘track’ pathway takes an extra 2 clicks to reach the next stage of the application path. However, this is a minor drawback considering the performance gains.

Once recommendations have been generated, the user is able to either create a new playlist or add to an existing one should they wish to. This functionality is embedded into the results of the recommendation request at the bottom of the screen.



Figures 6.10 (left) and 6.11 (right) showing the footer button layout of the recommendation results screen (6.10) and the prompt when adding to a playlist (6.11) respectively.

Creation of a new playlist is a straightforward task for the user; a button click simply sends a list of track URIs from the above results list as a POST request to the Spotify API, creating a playlist named ‘UnderDiscover Recommends’. Adding to an existing playlist is slightly more complicated. The user will be presented with a list of playlists they have control over; this only includes singularly owned playlists, not collaborative playlists or non-owned playlists. The user then selects the playlist and the tracks are added in the same way via a POST request.



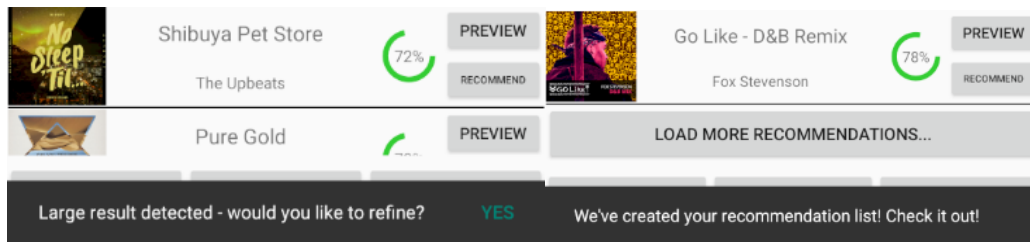
Figures 6.12 (left) and 6.13 (right) showing refinement prompts for large and small results.

When users are recommended large or small amounts of music, the system will detect this and suggest an automatic refinement of the search in order to generate a more acceptable amount of music. This suggestion is presented via a Snackbar and the user must click 'Yes' in order for it to occur. Large results, defined as over 50 tracks, will upon confirmation rerun the algorithm with the tightness value at 80% of it's previous value; for example, a previous run with a tightness value of 15 would be rerun with a tightness value of 12. Small results, defined as under 10 tracks, will similarly rerun the algorithm with a tightness value at 120% of it's previous value; where it previously was 15 it would become 18.

6.2 User Interface

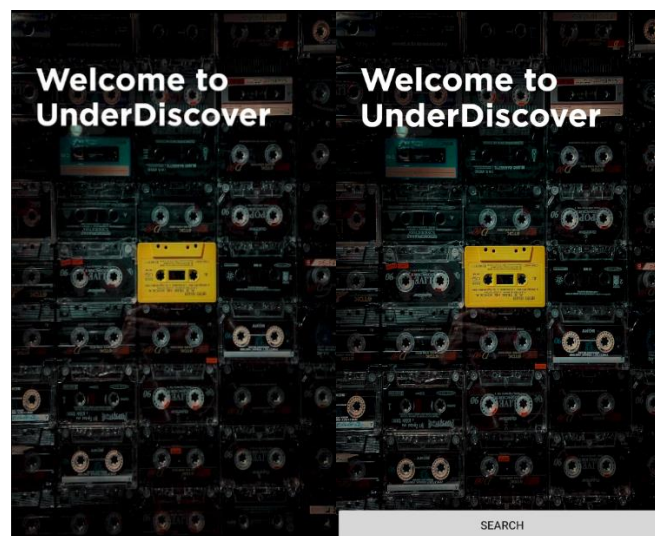
Data lists form an integral part of the user interface of my application. When developing the first implementations of the key functions, I experimented with both the ListView and RecyclerView libraries in order to build a consistent foundation. I decided early on to stick with ListView; although RecyclerView provides increased flexibility in solving some complicated issues, for example where lists are populated with several different views, it did not make the development process of ListView significantly easier to the extent where switching would be worth it. Further, as I have previous programming experience with ListView, it made sense to stick to what I could confidently and efficiently develop in if there were no gains to be made from switching. One inconsistency between function lists is the lack of a 'Load More' button on any list other than the recommendation results list (see fig 6.10). This decision was largely made from an ease of use perspective; by loading in only 10 tracks at a time in the recommendation result list, users should be encouraged to explore each of their suggestions in further depth. Conversely, showing all search results at once encourages users to make a selection and move to the primary functionality of the app.

Providing users with notifications when action may need to be taken is an important part of the user interface. I chose to implement these notifications using Snackbars for various reasons; primarily, they appear at the footer of the application. Their location on screen means that their presence covers the footer button array on screen. This is beneficial in two ways; firstly, it prevents the notification from taking up the entire screen and flustering the user. Notifications from Snackbars do not need to be acted on, they only aim to improve the user experience. As such, presenting them in an obvious but not overwhelming way is appropriate. Secondly, the covering of the footer button array prevents any further actions being taken before the user has acted upon information they may previously have not had.



Figures 6.14 (left) and 6.15 (right) showing various uses of the SnackBar library.

Expanding on 'footer button arrays', the implementation of my user interface follows a largely universal approach. Each unique screen of the application is separated out into three sections; a header area used for data collection or holding information, a main body usually containing a list of either static or dynamic data, and a footer area containing buttons for navigating further upwards, or back down, the application pathway. Even the opening splash screen follows this design pattern, although it is less strict in screen division than the other screens. This is important as there is a observed link between cognitive load and interface consistency (Mendel & Pak, 2009) (Punchoojit & Hongwarittorn, 2017).















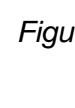
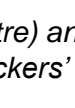
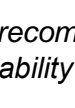
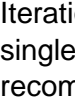
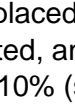
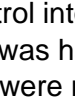
Figures 6.16 (left) and 6.17 (right) showing the splash screen before and after Spotify authentication respectively.

The use of a splash screen upon opening the application has both performance and visual reasonings at its core. Splash screens, as suggested by (Eskola, 2018), prevent any performance issues from concerning users or causing bugs upon opening the application. They also provide an opportunity to create a visually appealing first screen for the user when they enter the application. This image fits the core purpose of the application; finding a rare or unlikely tape amongst many. It also provides an interesting juxtaposition where a now outdated piece of technology is taking centre stage on its replacement.

6.3 Recommendation Engine

Developing the recommendation engine was an iterative process, each step increasing the complexity and variety of the algorithm. This process was broken down into three key steps:

1. Filtering algorithm based on a single attribute without sorting or weighting component.
2. Filtering & sorting algorithm based on single/multiple attributes with partial weighting.
3. Filtering & sorting algorithm based on single/multiple attributes with full weighting.

We found 48 tracks we think you will like...			We found 80 tracks we think you will like...			We found 80 tracks we think you will like...		
	Go Like - D&B Remix Fox Stevenson	PLAY METADATA		Go Like - D&B Remix Fox Stevenson	100% PLAY METADATA		Kids in the Club Flowidus	64% PREVIEW RECOMMEND
	Point Em Up Kanine	PLAY METADATA		Start A Rampage (Rampage Anthem 2020) Murdock	99% PLAY METADATA		Bearclaw Kove	61% PREVIEW RECOMMEND
	Start A Rampage (Rampage Anthem 2020) Murdock	PLAY METADATA		Point Em Up Kanine	99% PLAY METADATA		Shibuya Pet Store The Upbeats	61% PREVIEW RECOMMEND
	Kids in the Club Flowidus	PLAY METADATA		Intravenous Calyx & TeeBee	99% PLAY METADATA		non-responsive Phace	59% PREVIEW RECOMMEND
	Intravenous Calyx & TeeBee	PLAY METADATA		Kids in the Club Flowidus	99% PLAY METADATA		Slap the Ghost Misanthrop	59% PREVIEW RECOMMEND
	Just Hold On - Sub Focus & Wilkinson Sub Focus	PLAY METADATA		Bearclaw Kove	98% PLAY METADATA		Where is the Sun Joe Ford	58% PREVIEW RECOMMEND

Figures 6.18 (left), 6.19 (centre) and 6.20 (right) showing a recommendation request using the track 'Hackers' by Metrik and the Danceability attribute.

Iteration 1 (Centini (2), 2020) placed the least amount of control into the user's hands; only a single attribute could be selected, and the tightness variable was hard-coded into the recommendation algorithm at 10% (see fig 6.18). The tracks were returned to the user in the order returned by the API, and no explicit justification is provided as to how the tracks were selected. However, as a prototype iteration it satisfied the system requirement to recommend tracks to the user based upon a content-based filtering approach – although as opposed to the algorithm being biased towards this filtering method it is wholly reliant on it.

Iteration 2 (Centini (3), 2020) maintained this wholly content-based approach, however it provided significant improvements in variable user control (see fig 6.19). Firstly, users were able to select from multiple attributes at a time. Secondly, users were able to adjust the tightness variable which was previously hard-coded into the system (see section 6.1 for both). This iteration also brought in the first complex steps of the algorithm; in weighting and sorting. The engine would take the attribute values of the original song and compare them to each suggested tracks' attribute values. To calculate the value shown, the engine ran an averaging formula to generate a similarity rating, where x represents each selected attribute's value difference between the new track and original track, and n represents the number of selected attributes:

$$100 - \left(\frac{x_1 + x_2 + \dots x_n}{n} \right)$$










One problem that this formula brought, as can be seen visually in figure 6.19, is the tightly packed nature of results; it leaves little room for differential in more specific requests. For example, the algorithm would give a match value of 100% if the original and recommended

tracks had the same value whether 1 attribute or 6 were used. Another problem is that the algorithm is still entirely based on content factors, with no collaborative factors weighed in.

Iteration 3 addresses both of the problems above (see fig 6.20). The following equation, after several iterations of testing, is the final algorithm for the recommendation engine, where x and n represent the same values as in iteration 2, and y represents the popularity of the new track:

$$100 - \left\{ \left(\frac{x_1 + x_2 + \dots x_n}{n} \right) + \left(\frac{y}{1.5n} \right) \right\}$$

This algorithm solves the tightly packed nature of iteration 2 by factoring in the popularity variable in relation to the number of attributes selected; smaller numbers of attributes selected will cause the algorithm to favour popularity more without compromising the integrity of content-based sorting. This is because the popularity variable is spread from 0 to 100 but not limited to the tightness variable as other attributes are. Inherently, it also solves the problem of the algorithm having no collaborative weighting at all. Figure 6.20 shows lower match values for the same search than figure 6.19, addressing the aforementioned issue. Figure 6.21 (below) verifies this, showing a situation where the same track and tightness percentage are used with 3 attributes selected.

We found 17 tracks we think you will like...			We found over 100 tracks we think you will like...		
	Kids in the Club Flowidus	87% PREVIEW RECOMMEND		Jungle The Qemists	86% PREVIEW RECOMMEND
	Start A Rampage (Rampage Anthem 2020) Murdock	85% PREVIEW RECOMMEND		The Message Prolix	85% PREVIEW RECOMMEND
	Lost It Mode Friction	82% PREVIEW RECOMMEND		Mud Black Sun Empire	84% PREVIEW RECOMMEND
	Breaking Point (feat. Robin Adams) Joe Ford	80% PREVIEW RECOMMEND		Requiem The Qemists	84% PREVIEW RECOMMEND
	Point Em Up Kanine	80% PREVIEW RECOMMEND			

Figures 6.21 (left) and 6.22 (right) showing a showing a recommendation request using the track 'Hackers' by Metrik, where 6.21 shows 3 selected variables and fig 6.22 shows none.

As well as this, iteration 3 facilitates a use case where the user selects no attributes in the advanced recommendation mode. The algorithm runs a basic collaborative equation:

$$100 - \left(\frac{y}{2} \right)$$

Due to the unique nature of this use case, I did not feel obliged to worry about percentage matches in the same light as when attribute selection is at play; the use case is inside the advanced section of the algorithm dictation activities and provides a warning to the user via a pop-up that their search will be conducted in this way before they proceed.

6.4 API Interaction

API Interaction was handled by the HttpURLConnection library in Android. HTTP requests are expensive operations; although the basic premise of making these calls to obtain relevant data from the Spotify API was outlined in the design phase, individual decisions had to be reflected on during the development of the application in response to cost-benefit analysis.

When obtaining the details of a track in either the search or recommendation functions, the key details are usually provided in the initial HTTP call return; that is, the singular call to return all search results for either function. These details include the tracks name, the artists name and the track URI. These details are effectively obtained 'for free' as no additional HTTP calls had to be made in order to acquire them.

Other details require expensive additional calls to achieve. The single most expensive additional HTTP call to make is acquiring album art for tracks. This process requires taking the image url provided in the main return call alongside other key details and running a HTTP call for each track; further, this requires the drawing of this image onto the UI via a Drawable object. The cost of this is acceptable in my eyes; album artwork has been demonstrated to have a significant relationship with the music it is designed for (Ward, 2018). Further, demand for artwork to be incorporated more heavily into music platforms has been shown in academic literature; (Cook, 2013) suggests that 76% of surveyed users "wanted to see more cover art [...] incorporated online".

Another expensive call is acquiring previews of tracks. Obtaining a 30 second audio clip is bound to be expensive for the system. However, it has the large benefit against album art of not being mandatory; The url of the track preview is saved at the same time as the other key details, and that url is passed to the Audio Preview Manager in the event that the user requests a preview of a specific track. Thanks to this, the load on the system is spread out. This makes the call far more justifiable than the need for album art.

Generating recommendations from a request from the user is particularly expensive in comparison to search or attribute request functionality. This is partially due to the large number of additional HTTP calls needed to be made. When generating recommendations for the user, the system will make an initial call to the Spotify API using the attribute definitions as well as the seed of the original track to generate the suggestions required. However, in order to present a visualisation of the relationship between each recommended track and the original, as well as sort the tracks into the order with which the application recommends them, each track must have an additional call to the API made to acquire its attribute values. Although expensive, these additional calls are vital in order to satisfy requirements 10 and 11 of the project specification.

7 Testing & Evaluation

7.1 Performance Evaluation & Iterations

One of the first performance bottlenecks I noticed was the time taken to handle HTTP calls to the Spotify API. Breaking down the request process, there are 3 common sections; output of HTTP request, input of received data and processing of data into the GUI (usually via a list of items). Network performance is largely out of my hands and processing is independent to each function in performance terms. Thus, the aforementioned bottleneck was likely to be found in input processing. Input processing involves processing an InputStream into a String; Appendix 1 shows the original function as well as the proposed improved function.

Time in Milliseconds, AVG of 3 runs	StringBuilder	String Append	Performance Improv %
SearchActivity "Hackers" AVG	1392.978933	5321.5466	73.82379526
SearchActivity "Culture Shock" AVG	589.1631333	4106.7412	85.6537555
RecommendedResultActivity (Mackey Gee Tour as basis) Basic - Energy (Less) AVG	10308.58973	20077.5098	48.65603436
RecommendedResultActivity (Mackey Gee Tour as basis) Basic - Danceability (More) AVG	5486.1461	7857.102033	30.17595957
RecommendedResultActivity (CamelPhat Breathe as basis) Basic - Energy (More) AVG	6963.0806	12320.75147	43.48493581
RecommendedResultActivity (CamelPhat Breathe as basis) Basic - Valence (Less) AVG	258.0313	281.8684667	8.456840507

Figure 7.1 shows the performance difference between the functions shown in Appendix 2 in various examples. Time is measured from moment of HTTP call until final function result.

As figure 7.1 shows, the performance improvement between using += appending and StringBuilder is substantial; the search function is over 70% faster in both iterations and the recommendation function varies from just under 10% faster to almost 50% faster. Larger performance improvements can be found when dealing with more complicated results; the recommendation function handled 2 matching songs for entry 6, compared to over 100 in the case of entry 3. Reduced improvements between the search and recommendation functions can be attributed to the percentage of the operation taken up by the input handling; the recommendation algorithm takes up a substantive part of the operation in that function, whereas the input handling is the most expensive part of the search function.

Evaluating the effectiveness of the attributes provided to the user for customization, it is clear that loudness is a far less effective attribute than the other six provided. Firstly, the loudness of the track does not signify anything particular about the musical qualities of the tracks; rather, it is a measure how a track was mastered by the producer. Further, it proved far more difficult to adapt into the algorithm than other attributes due to its non-uniform scale; loudness is measured typically between -60 and 0. However, I found during manual use testing that the attribute did not actively hinder usage of the app. Further, with a distribution similar to that of speechiness and acousticness, meaningful suggestions could be made based upon the value – those suggestions will not relate to musical content but that provides users with an alternative attribute with which to experiment.

One small issue which arose during manual use testing was a niche issue in which a combination of selecting only 1 attribute for the recommendation engine to use and results containing extremely popular music, where the algorithm would occasionally end up returning a negative percentage result. This was due to the iteration 3 algorithm outlined in section 5.1 not altering the popularity value – previously using $\frac{y}{n}$ instead of $\frac{y}{1.5n}$ songs with a popularity identifier of over 85 could feasibly return negative values. Mathematically, it is now impossible for this issue to occur as the most extreme values (100 for popularity, 25 for percentage difference) would now return a minimum percentage match of 8%.

7.2 Acceptance Testing

The most important judgement of the success or failure of this project must be to compare it against the project specification. Accepting a requirement as completed should come with a justification for doing so; similarly, rejecting a requirement as completed should be justified.

Considering the overarching system requirements, numbers 1 and 2 should be accepted; these can be noted as complete from the wider picture of this document and the submission of relevant code. Number 3 should be rejected as no development on the iOS platform has taken place.

Numbers 4 and 5, apart of the user access control requirements, should be accepted. The application will not allow access to any further functionality beyond the search screen without adequate Spotify authentication. Users will also experience no differences in functionality if their accounts are free or premium; an early prototype branch worked on implementing full-song previews as opposed to 30 second previews but this avenue was not viable due to API limitations (Centini (4), 2020). Number 6 should be rejected as no local user data is stored.

All 5 requirements under the recommendation engine requirements should be accepted. Number 7 represents an overridden requirement in place to fall back upon in the event of development issues. Numbers 8 is represented by the 'basic' request pathway and numbers 9 and 11 are represented by the 'advanced' request pathway respectively. Number 10 is present in the recommendation results for both pathways.

Within the pathways and functionality section, number 12 should be accepted although it is an overridden requirement. Number 13 should also be accepted as is justified in section 6.1. Numbers 14-17 can also all be accepted as they are represented by completed functionality in the implementation section.

Numbers 18-24 of the user interface section should all be accepted as all functionality is available to interact with and visualise. Number 19 should be accepted specifically in the context of the demonstration, where the application will be shown on both a Nexus 5X (via the Android Emulator) and a physical Samsung S7 Edge. Number 25 should be accepted due to the justification given in section 6.2 regarding the opening screen.

Test + Iteration	Raw Nano Time	Time in Seconds
Page Switch No HTTP - Splash to Search (IT1)	7859300	0.0078593
Page Switch No HTTP - Splash to Search (IT2)	5095700	0.005097
Page Switch No HTTP - Splash to Search (IT3)	8783800	0.0087838
Page Switch No HTTP - Basic to Advanced (IT1)	6383200	0.0063832
Page Switch No HTTP - Basic to Advanced (IT2)	5763300	0.0057633
Page Switch No HTTP - Basic to Advanced (IT3)	7445700	0.0074457
Requirement 26 + 27 Test Average	6888500	0.006888517

Figure 7.2 showing performance testing results for page switches not involving HTTP calls.

Numbers 26 and 27 can be accepted; 26 is an overridden requirement by number 27, and 27 can be shown to be true by the performance figures in figure 7.2.

Numbers 28-31 can all be accepted based on the performance testing shown in figure 7.1 (it is important to note that results in 7.1 are an average of 3 runs); numbers 28 and 29 show page switch times under 2 seconds in both cases. Numbers 30 and 31 show a highest result of just over 10 seconds, with the rest of the runs coming in far lower.

Test + Iteration	Raw Nano Time	Time in Seconds
Full Start - Up until progression button shown - IT1	1277786500	1.2777865
Full Start - Up until progression button shown - IT2	1292237900	1.2922379
Full Start - Up until progression button shown - IT3	1268473400	1.2684734

Figure 7.3 showing performance testing results for a full application start.

Number 32 can also be accepted on the basis of the performance testing shown in figure 7.3; despite this, I am cautious to note that first time users will experience longer start up times due to the authentication procedure.

Overall, the vast majority of requirements set out for this project have been met; further, one of the stretch goal aims in providing an audio interface for users has also been met.

8 Critical Discussion & Conclusion

8.1 Critical Discussion

UnderDiscover represents a unique contribution to Spotify ecosystem. The closest application publicly available to this application is Nelson, a web application which facilitates attribute specification to generate attributes but does not delve to a track, album, or artist specific starting point, only allowing genre to be set (Vaniderstine, 2018). The Spotify developer showcase highlights several applications which partially match my projects functionality such as 'Discover Quickly' and 'Musical Data', but none which fully match the goals and objectives I set out (Spotify AB (5), 2020). Indeed, the academic paper by Millecamp et al. provides a system more similar to mine than any applications publicly available; despite this, it only facilitates the 'artist' starting point missing from Nelson and suffers from the invariably problematic issue of not being publicly available (Millecamp, et al., 2018).

One important aspect of the project critique is determining whether the recommended music is 'good'; unfortunately, any attempt to definitively answer that question will end in failure as ultimately music taste is subjective. In spite of this, quantifying the 'relevance' of the music presented may be more unbiased. The recommender will always, at a minimum, return music in the same genre as the initially selected track thanks to the 'seeding' used in the API; 'seeding' being where the original track is "matched against similar artists and tracks" (Spotify AB (1), 2020). The recommender algorithm is novel, which inherently comes with uncertainties; despite this, those uncertainties can only be positive as the algorithm will always provide 'relevant' music, even if they are not always 'good' recommendations.

During my time working on this project, the first criticism I have gotten from friends and colleagues is 'What about Discover Weekly'. Discover Weekly is a platform-generated playlist updated weekly for users which suggests 30 songs which match your listening habits. The algorithm behind the playlist is incredibly sophisticated, far more so than what is being presented by UnderDiscover. However, it fails to produce flexibility to the user. Discover Weekly utilises a snapshot of your entire listening history to generate recommendations; it doesn't provide the flexibility of developing very specific recommendations on the go.

Considering my initial motivation, I personally DJ genres across the dance music scene and it is reflected in my listening habits on Spotify. When I receive my Discover Weekly playlist, it often has 3-4 tracks from each of the dance music genres I listen to. I often play genre specific sets which require me to 'dig deeper' and find more new music in one specific genre. UnderDiscover facilitates that better than Discover Weekly.

8.2 Conclusion

Software development should never be seen as finished; a “notion of computer systems as finished products [...] leads to viewing system enhancement as an error correcting activity” (Bjerknes, et al., 1991). There are several further avenues of development that this project would benefit from. Most importantly, porting the system to iOS would open up a non-insignificant section of the mobile application market to the system. React Native, addressed in the literature review, could be used to achieve this. Secondly, improvements to the user interface would benefit the application from a commercial standpoint; the current interface is effective and easy to use. Despite this, it would be more visually appealing with more colour or a theme based on the title screen background. Finally, expansion of user pathways would allow the application to cover more unique user stories. For example, displaying individual attribute calculations to the user on a separate screen is a function I justified not including in section 3.1; only the most experienced users are likely to make use of this advanced feature. Further development could cater to such users by including these types of functions; similarly, very inexperienced users could be provided with even simpler recommender request lines such as ‘party’ or ‘study’ which would remove even the attributes from the user’s line of sight.

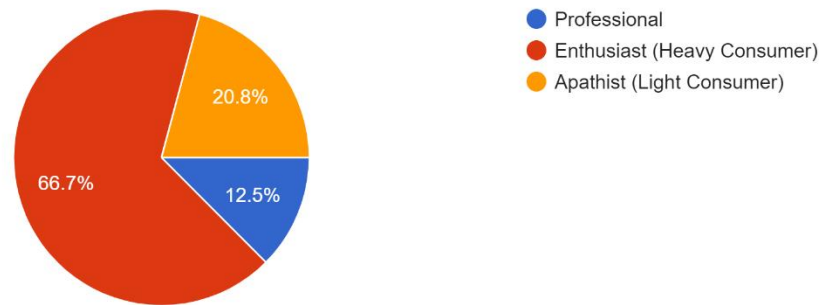
Not only having a future development plan, UnderDiscover has triggered personal discussion with researchers including Paul Goodge, a PhD student who is interested in how my application could be used in relationship with fans of progressive rock and their listening habits. I am also investigating avenues with which to commercialise the project after it has served its academic purpose.

9 Appendices

Appendix 1: Primary Research on Importance of Musical Attributes

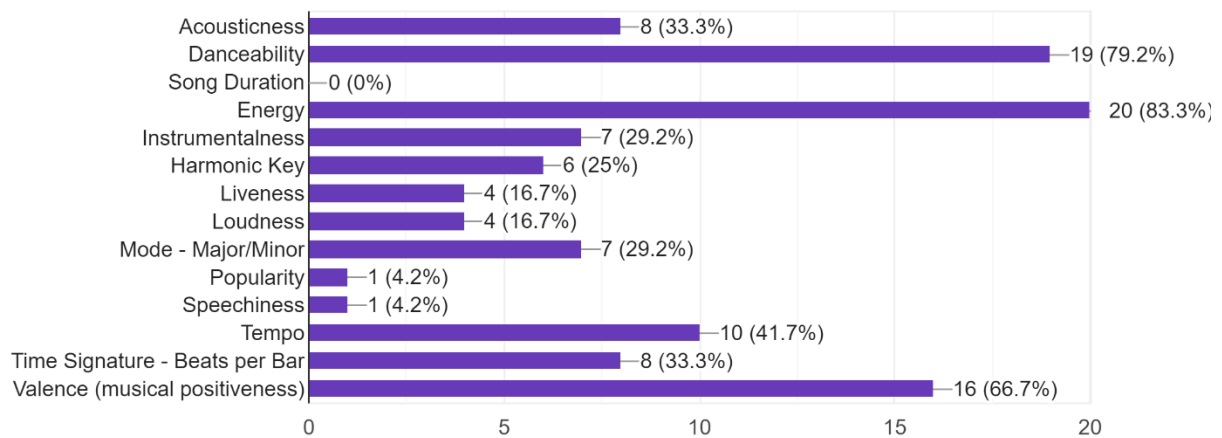
Do you consider yourself a music...

24 responses



Which of these attributes do you consider important as a quality of music?

24 responses



Appendix 2: streamIntoString Function Iterations

Iteration 1: Appending to String object using +=

```
protected static String streamIntoString(InputStream stream) {  
    //Method to process and correctly separate input streams  
    try {  
        BufferedReader reader = new BufferedReader(new InputStreamReader(stream));  
        String data;  
        String result = "";  
  
        while ((data = reader.readLine()) != null) {  
            result += data;  
        }  
        if (null != stream) {  
            stream.close();  
        }  
        return result;  
    }  
    catch (IOException eIO) {  
        eIO.printStackTrace();  
        System.exit( status: 3);  
    }  
    return null;  
}
```

Iteration 2: Appending to StringBuilder object

```
protected static String streamIntoString(InputStream stream) {  
    //Method to process and correctly separate input streams  
    try {  
        BufferedReader reader = new BufferedReader(new InputStreamReader(stream));  
        String data;  
        StringBuilder result = new StringBuilder();  
  
        while ((data = reader.readLine()) != null) {  
            result.append(data);  
        }  
        if (null != stream) {  
            stream.close();  
        }  
        return result.toString();  
    }  
    catch (IOException eIO) {  
        eIO.printStackTrace();  
        System.exit( status: 3);  
    }  
    return null;  
}
```


10 Bibliography

Bassil, Y., 2012. A Simulation Model for the Waterfall Software Development Life Cycle. *International Journal of Engineering and Technology*, 2(5), pp. 742-749.

Bateira, J., Gouyon, F., Davies, M. & Caetano, M., 2014. Music Discovery in Spotify with RAMA. *15th International Society for Music Information Retrieval Conference*.

Beck, K. et al., 2001. *Manifesto for Agile Software Development*. s.l.:s.n.

Bjerknes, G., Bratteteig, T. & Espeseth, T., 1991. Evolution of Finished Computer Systems. *Scandinavian Journal of Information Systems*, 3(1), pp. 25-45.

Celma, O., 2008. *Music recommendation and discovery in the long tail*, Barcelona: UPF.

Centini (1), M., 2020. *UnderDiscover: Trello Board*. [Online]
Available at: <https://trello.com/b/YIJgcx8M/underdiscover>
[Accessed 19 May 2020].

Centini (2), M., 2020. *GitHub Commit for "Iteration 1" in Figure 1*. [Online]
Available at:
<https://github.com/MorganC46/UnderDiscover/commit/14eb3175ccc0bc75e67a67fdc707b98be27235b4>
[Accessed 1 April 2020].

Centini (3), M., 2020. *GitHub Commit for "Iteration 2" in Figure 2*. [Online]
Available at:
<https://github.com/MorganC46/UnderDiscover/commit/5f104498b87df98f5d702cc4a73721c437d1241a>
[Accessed 4 April 2020].

Centini (4), M., 2020. *UnderDiscover: GitHub Branch "Premium Distinction"*. [Online]
Available at: <https://github.com/MorganC46/UnderDiscover/tree/FEATURE-PREMIUM-DISTINCTION>
[Accessed 17 May 2020].

Cook, K., 2013. *The Effect of Cover Artwork on the Music Industry*, California: California Polytechnic State University.

Danielsson, W., 2016. *React Native application development - A comparison between native Android and React Native*, Linköping: Linköpings universitet.

Economic Times, 2018. *Music app Gaana introduces new features to improve user experience*. [Online]
Available at: <https://economictimes.indiatimes.com/industry/media/entertainment/music-app-gaana-introduces-new-features-to-improve-user-experience/articleshow/66813088.cms?from=mdr>
[Accessed 13 November 2019].

Ekstrand, M. D., Kluver, D., Harper, F. M. & Konstan, J. A., 2015. Letting Users Choose Recommender Algorithms. *Proceedings of the 9th ACM Conference on Recommender Systems (RecSys '15)*, p. Vienna : ACM Press.

Eskola, R., 2018. *React Native Performance*, Aalto: Aalto University School of Science.

Gartner, 2018. *Gartner Says Worldwide Sales of Smartphones Returned to Growth in First Quarter of 2018*. [Online]
Available at: <https://www.gartner.com/en/newsroom/press-releases/2018-05-29-gartner-says-worldwide-sales-of-smartphones-returned-to-growth-in-first-quarter-of-2018>
[Accessed 11 November 2019].

IDC, 2019. *Smartphone Market Share*. [Online]
Available at: <https://www.idc.com/promo/smartphone-market-share/os>
[Accessed 11 November 2019].

- IFPI, 2018. *Music Consumer Industry Report 2018*. [Online]
Available at: <https://www.ifpi.org/downloads/music-consumer-insight-report-2018.pdf>
[Accessed 7 November 2019].
- Irilli, R., 2018. *Spotworm GitHub Repository*. [Online]
Available at: <https://github.com/ryanirilli/spotworm>
[Accessed 29 April 2020].
- Jenkins, E. & Yang, Y., 2016. *Creating a Music Recommendation and Streaming*. Porto, Springer.
- Jin, Y., Cardoso, B. & Verbert, K., 2017. How Do Different Levels of User Control Affect Cognitive Load and Acceptance of Recommendations?. *Proceedings of the 4th Joint Workshop on Interfaces and Human Decision Making for Recommender Systems co-located with ACM Conference on Recommender Systems (RecSys 2017)*, pp. 35-42 : Como.
- Kaji, K., Hirata, K. & Nagao, K., 2005. *A music recommendation system based on annotations about listeners' preferences and situations*. Florence, IEEE.
- Kodama, Y. et al., 2005. *A music recommendation system*. Las Vegas, NV, IEEE.
- Leau, B. Y., Loo, W. K., Yip, T. W. & Tan, S. F., 2012. *Software Development Life Cycle AGILE vs Traditional Approaches*. Singapore, IACSIT Press.
- Lin, N., Tsai, P.-C., Chen, Y.-A. & Chen, H. H., 2014. *Music recommendation based on artist novelty and similarity*. Jakarta, IEEE.
- Majchrzak, T. A., Biørn-Hansen, A. & Grønli, T.-M., 2017. *Comprehensive Analysis of Innovative Cross-Platform App Development Frameworks*. Hawaii, 50th Hawaii International Conference on System Sciences.
- Mendel, J. & Pak, R., 2009. The Effect of Interface Consistency and Cognitive Load on User Performance in an Information Search Task. *Proceedings of the Human Factors and Ergonomics Society Annual Meeting*, 53(22), pp. 1684-1688.
- Millecamp, M., Htun, N. N., Jin, Y. & Verbert, K., 2018. Controlling Spotify Recommendations: Effects of Personal Characteristics on Music Recommender User Interfaces. *Proceedings of the 26th Conference on User Modeling, Adaptation and Personalization*, pp. 101-109.
- O'Bryant, J., 2017. *A survey of music recommendation and possible*. [Online]
Available at: <https://jacobobryant.com/about/mrs.pdf>
[Accessed 10 November 2019].
- Papp, A., 2017. *The History of React.js on a Timeline*. [Online]
Available at: <https://blog.risingstack.com/the-history-of-react-js-on-a-timeline/>
[Accessed 16 November 2019].
- Punchoojit, L. & Hongwarittorn, N., 2017. Usability Studies on Mobile User Interface Design Patterns: A Systematic Literature Review. *Advances in Human-Computer Interaction*, Volume 2017, pp. 1-22.
- Ruparelia, N. B., 2010. Software Development Lifecycle Models. *Software Engineering Notes*, 35(3), pp. 8-13.
- Sagiroglu, S. & Sinanc, D., 2013. *Big Data: A review*. San Diego, IEEE.
- Soleymani, M., Aljanaki, A., Wiering, F. & Veltkamp, R. C., 2015. *Content-based music recommendation using underlying music preference structure*. Turin, IEEE.
- Spotify AB (1), 2020. *Get Recommendations from Seeds*. [Online]
Available at: <https://developer.spotify.com/documentation/web-api/reference/browse/get-recommendations/>
[Accessed 19 March 2020].

- Spotify AB (2), 2019. *Get Audio Features for Several Tracks*. [Online]
Available at: <https://developer.spotify.com/documentation/web-api/reference/tracks/get-several-audio-features/>
[Accessed 10 October 2019].
- Spotify AB (3), 2020. *Search for an Item*. [Online]
Available at: <https://developer.spotify.com/documentation/web-api/reference/search/search/>
[Accessed 12 April 2020].
- Spotify AB (4), 2020. *Get an Artist's Albums*. [Online]
Available at: <https://developer.spotify.com/documentation/web-api/reference/artists/get-artists-albums/>
[Accessed 17 April 2020].
- Spotify AB (5), 2020. *Developer Showcase*. [Online]
Available at: <https://developer.spotify.com/community/showcase/>
[Accessed 11 May 2020].
- Spotify AB (6), 2020. *Get Audio Features for a Track*. [Online]
Available at: <https://developer.spotify.com/documentation/web-api/reference/tracks/get-audio-features/>
[Accessed 18 March 2020].
- Spotify AB (7), 2020. *Create a Playlist*. [Online]
Available at: <https://developer.spotify.com/documentation/web-api/reference/playlists/create-playlist/>
[Accessed 31 March 2020].
- Spotify AB (8), 2020. *Add Tracks to Playlist*. [Online]
Available at: <https://developer.spotify.com/documentation/web-api/reference/playlists/add-tracks-to-playlist/>
[Accessed 31 March 2020].
- Spotify Technology, S.A, 2018. *Shareholder Letter Q4 2018*. [Online]
Available at: https://s22.q4cdn.com/540910603/files/doc_financials/quarterly/2018/q4/Shareholder-Letter-Q4-2018.pdf
[Accessed 9 October 2019].
- Spotify, 2019. [Online]
Available at: <https://newsroom.spotify.com/company-info/>
[Accessed 9 November 2019].
- Vaniderstine, A., 2018. *Nelson*. [Online]
Available at: <https://nelson.glitch.me/>
[Accessed 28 April 2020].
- Vijayasathy, L. R. & Turk, D., 2008. Agile Software Development; A Survey of Early Adopters. *Journal of Information Technology Management*, 19(2), pp. 1-8.
- Wang, M. et al., 2014. Context-Aware Music Recommendation with Serendipity Using Semantic Relations. *Lecture Notes in Computer Science*, 21 May, pp. 17-32.
- Ward, T., 2018. *Aesthetics of Sound: The Relationship Between Music and Its Artwork*, Mississippi: ProQuest LLC.