NUI Galway

College of Engineering and Informatics

Automatic Chord Estimation on the Web

Luke Gibbons

Final Year Project 2020 - 2021

BSc. (Hons.) in Computer Science & IT

Supervised by James McDermott

Table of Contents

Table of Figures

Table of Tables

Chapter 1 – Introduction

Automatic Chord Estimation (ACE) refers to the task of estimating the chords that appear within a piece of musical composition. This task is visually presented in figure 1. It is a problem that has proven popular in the area of Musical Information Retrieval (MIR), attracting research from a number of domains including signal processing, and machine learning [1]. The reason for its attraction may come down to its versatility, as the research community has found ACE to work as both a stand-alone module in chord transcription engines, and as a sub-module for MIR tasks such as genre classification [2], and musical key detection [3].

ACE also holds a place in the annual Music Information Retrieval Evaluation eXchange (MIREX)[1] conference. Each year, entering ACE systems are tested on the accuracy to which they predict the chords on a set of unseen songs, for which the ground-truth is known. Since the tasks inception into the conference in 2008, the average ACE accuracy has seen a slow and steady improvement, with submissions in 2019 routinely surpassing and accuracy of 85%[2].

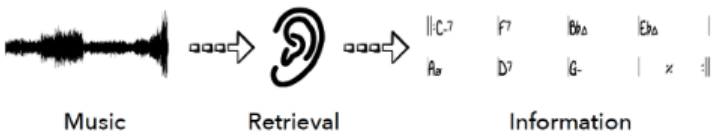

*Figure 1 - The task of an ACE. The ACE algorithm (**Retrieval**) will take the input audio (**Music**) and generate a set of chord predictions (**Information**). Source: [4]*

Outside of research, ACE is not seen as popular with very few applications making use of the task. Perhaps the most widespread application to use ACE is Capo[3], a transcription application exclusive to Apple devices and winner of the 2011 Apple Design Award. The Capo User Interface (UI) is found in figure 2.

The drawback of Capo is in its exclusivity to iOS and Macintosh devices which limits many users' access to such an application. This leads to the premise, and title of this project: ACE on the Web. From the research I conducted, I was unable to find any web application that utilized an ACE system. I therefore wanted to explore this concept as I believe ACE represents a great opportunity to assist in beginner musicians development, due to the requirement of an expertly trained ear to transcribe songs [1], a trait beginners are unlikely to own.

[1] MIREX home page: https://www.music-ir.org/mirex/wiki/MIREX_HOME
[2] MIREX 2019 Evaluation Results: https://www.music-ir.org/mirex/results/2019/MIREX2019Poster.pdf
[3] Capo: https://supermegaultragroovy.com/products/capo

This is not to say that applications don't exist that cater to this userbase, however using an ACE could potentially provide additional benefits. For example, Songsterr[4] is such an application that is in similar scope to the proposed project as its focused on allowing beginners to play-along with songs. The application, however, relies on community support and there is therefore a limited number of songs available. This is an issue that also exists in the website Ultimate Guitar[5] which offers musical transcription in the form of tabs (or tablatures) of which can be seen in Figure 3. Ultimate Guitar boasts a greater range of song choice over Songsterr, however tabs aren't ideal for allowing playing along with songs due their lack of chord timings and static format.

Figure 2 - The Capo UI. The predicted chords run along the bottom of the screen (above the grey bar) with the emphasis of the UI resting on the input audio wave.

```
[Verse]
G       C           D       G
How many roads must a man walk down,
          C         G
before you call him a man
G       C           D         G
How many seas must a white dove sail,
          C           D
before she sleeps in the sand
        G       C             D          G
Yes, and how many times must the cannonballs fly,
             C      G
before they're forever banned

[Chorus]
C           D       G               C
The answer my friend is blowin' in the wind
              D             G
The answer is blowin' in the wind
```

Figure 3 – A section of a guitar tab from Ultimate Guitar.

A web application using an underlying ACE would not have such limitations as there is no reliance on community submissions. Users have the potential to obtain the chords and play-along with any song

[4] Songsterr: https://www.songsterr.com
[5] Ultimate Guitar: https://www.ultimate-guitar.com

they wish while also having the flexibility to access the application through a range of devices due to the accessibility of the web. It is such an application that this project aims to research and build.

Before diving into this research and development for the application, the project brief is outlined in Chapter 2. This provides a set of goals, of which can be analysed to judge the project's success, and an outline of the project's scope, which keeps the focus on the key requirements.

Once the brief has been established, exploration on the workings of an ACE and the requirements of modern web applications are detailed in Chapter 3. This serves as the research element of the project and allowed for the making of informed decisions with regard to technical aspects of the project.

With the research complete, the attention turns towards the application development in Chapter 4. This chapter provides detail on each piece of the applications developmental process, beginning with the steps required to run the ACE, before detailing the creation of the client-side application, integration of the two components, and finally deploying the application to the web.

Once the research and development are covered, the project can be tested upon and this is demonstrated in Chapter 5. This involved both evaluation of the ACE, in terms of accuracy and performance, and testing of the application component which involved analyses on its device support and performance.

Finally, Chapter 6 serves as a conclusion to the thesis. This chapter offers closing remarks and discusses the success of the project through analysis of the goals and outcomes presented in Chapter 2. As a closing to both the chapter and thesis, a look at the further work which could be completed, in relation to this project, is outlined.

Chapter 2 – Project Specification

This chapter details a brief for the project, a short statement capturing the central focus of the problem. From the brief, goals and outcomes are extracted which provides targets in which to aim, while also providing a benchmark for which I can later judge the success of the project. Finally, the scope of the project is provided. Understanding what exactly is in and out of scope ensures the project is completed in the allotted time during the developmental process, while also allowing for a greater understanding of success when analyzing the project.

2.1 Brief

The transcription of songs has proven to be a difficult and time consuming task, with the annotation process for songs typically taking 2 or more musically trained experts and an average time of 8-18 minutes per annotator per song [1]. For this reason, the task of ACE has attracted a number of research from a wide array of disciplines.

While ACE has made it into a few software applications focused on the transcription of audio files, it has yet to be implemented as a web application. On top of this, the current selection of ACE software has little focus on allowing beginner and intermediate guitarists to play along with their input song.

Due to the absence of ACE in web applications, and the potential that exists for the task in allowing both beginner and intermediate guitarists easy access to play-along practice, the task is to build a modern web application that utilizes a form of ACE. The application should accept and process an audio file (mp3, wav) before displaying the chords in-sync with their occurrence in the input song. The application should be aimed at beginner and intermediate guitarists and there should, therefore, be focus on implementing a clean and easy to use UI.

2.2 Goals & Outcomes

Table 1 outlines the goals and outcomes for the project. In this instance, goals can be defined as specific action statements, while outcomes are the expected results upon completion of such actions.

Goal	Outcome
Utilize technologies used to build modern web applications	A web application compliant with today's standards. This means one that is: 1. Scalable 2. Responsive (accessible on different screen sizes) 3. Secure 4. Accessible to all potential users
Be aware of the Web Content Accessibility Guidelines (WCAG)[6] during development.	
Be aware of the Open Web Application Security Project's (OWASP) Top 10 Web Application Security Risks[7] during development.	
Use prototyping software during the design phase of the application	A thoughtfully designed application with a UI focused on ease-of-use.
Implement an ACE into the web application that can predict chords when provided with an audio file	An application providing the ability for users to enter an audio file (mp3, wav) and play along with the song through display of the predicted chords.
Display the ACE's predicted chords as they occur within the input audio file	

Table 1 - Goals and Outcomes for the project. The outcomes (right) are the expected achievement through completion of the adjacent goals (left).

2.3 Scope

Table 2 displays the tasks that are considered both in and out of scope for the project. Tasks deemed out of scope are not a reflection on their irrelevance to the project, but rather those likely to be infeasible to complete by the projects deadline. Many of these tasks are brought up again in Chapter 6, where further work on the project is discussed.

[6] WCAG 2.1: https://www.w3.org/TR/WCAG21
[7] OWASP Top Ten: https://owasp.org/www-project-top-ten

In	Out
Obtain a pre-trained ACE system and implement it on the web	Build and train an ACE from scratch
	Advancement of current ACE techniques
Use existing libraries that may help in the ACE pre-processing steps	Build pre-processing techniques from scratch where existing ones already exist
Detection of the 24 most common chord shapes: major and minor triads [5]	Detection of less common and advanced chord shapes such as extended and altered chords
Allow input of audio files (songs) in any HTML supported format[8]: mp3, wav, or ogg	Live detection of chords through microphone input
Building an application using modern JavaScript tools and techniques	Building of a Progressive Web Application (PWA), an extended, installable web application offering features such as offline support and notifications[9]

Table 2 - Tasks that are in (left) and out (right) of the projects scope.

[8] HTML accepted formats and browser support: https://www.w3schools.com/tags/tag_audio.asp
[9] Google's Getting Started with PWA's: https://developers.google.com/web/updates/2015/12/getting-started-pwa

Chapter 3: Technical Review

This chapter serves as a review into the technical aspects of the project, which can be divided into two core components: ACE, and Modern Web Applications. The first section in the chapter details the inner workings researchers have employed in ACE systems, and this is then followed with a section into Modern Web Applications where their characteristics will be discussed along with the tools used to build them. This chapter closes with an outline of the ACE and build tools chosen to be used in this project along with the reasoning behind these choices.

3.1 Automatic Chord Estimation (ACE)

The first piece of research on ACE dates back more than 20 years to the year 1999, where Fujishima wrote the paper: *Real Time Chord Recognition of Musical Sound: a System using Common List Music* [6]. This paper proceeded to light the spark and cause ACE research to be as popular and active as it is today [7].

As the reasoning for its popularity has already been discussed in both Chapter 1 and Chapter 2, this section will cut straight to the technical aspects of ACE. The first aspect of ACE to understand in more detail is that which is being estimated: chords. A chord is a harmonic unit with at least three different tones sounding simultaneously. The most common chords, and those that make up the majority of ACE research, are known as triads and these fall into one of two categories: major and minor. Both categories comprise of 3 notes: a root note (which gives name to the triad), a major/minor third (4/3 pitches above the root), and a perfect fifth (7 pitches above the root). Such notes come from the pitch class set: (C, C#/Db, D, D#/Eb, E, F, F#/Gb, G, G#/Ab, A, A#/Bb, B) [1]. Taking the C chord as an example: a C Major will have the root note C, the major third E, and the perfect fifth G. Alternatively, a C Minor chord will have the same root note and perfect fifth, but instead of a major third will have a minor third, which in this case is the note D#/Eb.

The commonly associated workflow in ACE research can be seen in figure 4.

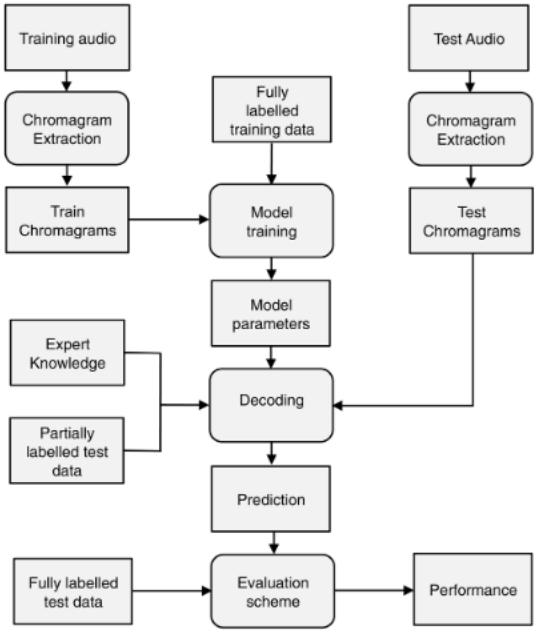

Figure 4 - The workflow commonly associated with ACE research per McVicar et al. [1]

3.1.1 Chromagram

The representation of audio used by most ACE systems is the chromagram. This is a real-valued matrix where each column represents a pitch class and each row represents a time frame. Each vector in the chromagram contains the prominence of the pitch classes at a specific time and is known as a chroma vector or chroma feature [1]. The visual representation of a chromagram for the start of *Let It Be* by *The Beatles* can be seen in figure 5.

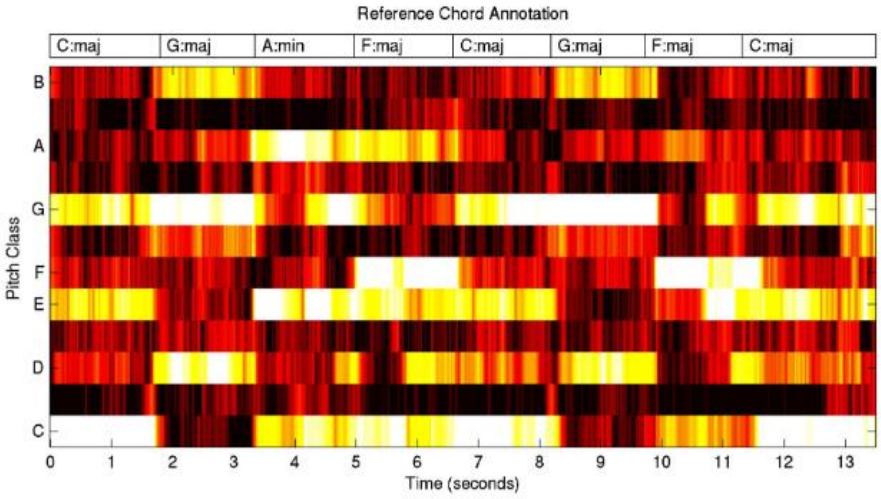

Figure 5 - A chromagram matrix for the opening of the song Let It Be (The Beatles). The ground-truth chords are also shown for comparison. Source [1]

At the beginning of ACE research, chromagrams started off relatively simple but have since incorporated a number of sophisticated techniques such as tuning, background-removal, and beat-synchronization, all of which can be found in figure 6. The latter of these was first put into place by Bello, who noticed that chords don't tend to change between the beats of a song. Its addition not only improved accuracy, but also required less computational power due to the reduced frequency in sampling [1].

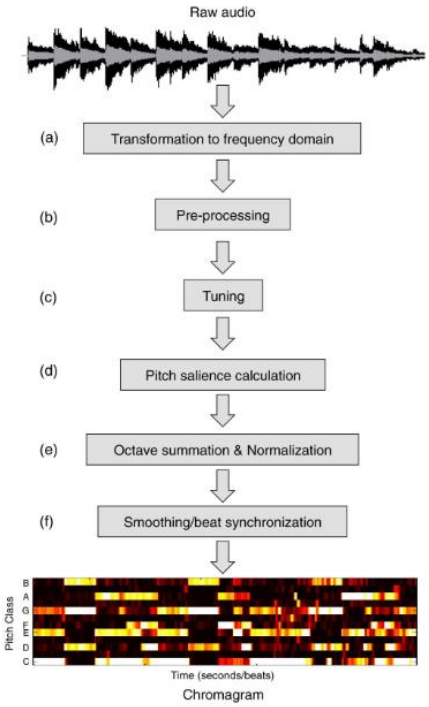

Figure 6 - Some common steps taken when converting audio to a chromagram. Source [1]

Before any of the aforementioned additional steps can take place, the audio has to be transformed to a frequency domain. This is most often done through a Short Time Fourier Transform (STFT). This technique has a considerable drawback however, in that it uses a fixed-length window. Setting this window length involves some experimentation as using too short a window will leave frequencies with long wavelengths indistinguishable, while using a large window size can cause a poor time resolution to be obtained [1].

More recent approaches attempt to overcome the limitations associated with STFT by using a Constant-Q Transform (CQT) which enable the use of variable length windows, a technique first utilized in a musical context by Brown [1].

3.1.2 Predicting

Over the years, the techniques used to predict chord for ACE has advanced and increased in sophistication. There are three methods that have been utilized most frequently at this stage: template matching, Hidden Markov Models (HMM), and neural networks [1].

Template matching is the simplest method of the three, consisting of the comparisons between the real-valued chroma vectors and binary template chords. At a given time-step, a template matching algorithm will compare the chroma vector to each binary chord template it supports. The binary template that is closest in likeness to the chroma vector is then output as the predicted chord [1]. Taking the C Major chord mentioned earlier, containing the notes CEG, this will have the equivalent binary template: [1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0]. Notice that the 1's are in the same locations that the pitches fall in the pitch class set.

The next method, Hidden Markov Models (HMM), is one of the most common methods used in ACE research. HMM's attempt to overcome a drawback associated with template matching: failure to model the continuous nature of chord sequences [1]. They are probabilistic models for sequences of observed variables, and while the working details of a HMM is out of scope for this section, a description of one, in the context of an ACE, can be found by Ni et al. [8].

Recently, the go-to method for assigning labels to audio frames is through the use of neural networks, and in particular, deep learning. In this case, the network is trained on data (which will be looked at in the next section) and expected to learn the necessary weightings and transforms from this data. As with other ACE systems, they commonly use chromagrams as inputs (usually extracted via CQT) and output one-hot chord class vectors [7].

3.1.3 Training Data

There are a number of publicly available datasets for ACE research which allow the previously discussed prediction models to be trained and tested upon.

In terms of fully-labelled datasets, there are two commonly used in ACE research: the Isophonics dataset[10] which consists of over 180 songs from The Beatles and Queen, and the McGill Billboard annotations[11] consisting of over 700 songs taken from the *Billboard* charts[12]. Both datasets represent audio in the form {start_time end_time chord}, a syntax proposed by Chris Harte [5], shown in figure 7.

```
0.000000 2.612267 N
2.612267 11.459070 E
11.459070 12.921927 A
12.921927 17.443474 E
17.443474 20.410362 B
20.410362 21.908049 E
21.908049 23.370907 E:7/3
23.370907 24.856984 A
...
```

Figure 7 - Example of the chord annotation layout typically found in ACE datasets. The first column is the chords start time, followed by its end time and name.

There has, however, been some criticism on the available datasets, stating their lack in genre variety with the majority of their songs as many fall into either the pop or rock categories [7]. For this reason, along with the vast amount of data present on the internet, some researchers have turned to partially-labelled datasets. This usually comes from chord transcription websites such as the previously mentioned Ultimate Guitar. This data has its drawbacks however, as ACE systems can have a hard time interpreting the data due to noise and a lack of chord timings. Though, as noted by McVicar, the sheer volume of data makes it a valuable resource for ACE [1].

3.2 Modern Web Application Development

Since the first web page was published in 1991, the look and technologies used to build such pages has changed considerably. While the early technologies, HTML and CSS, are still fundamental aspects

[10] Isophonics dataset: http://www.isophonics.net/datasets
[11] McGill Billboard Project: https://ddmal.music.mcgill.ca/research/The_McGill_Billboard_Project_(Chord_Analysis_Dataset)/
[12] Billboard charts: https://www.billboard.com/charts

of web pages, they have continually been improved upon to allow for the rapid demands in the industry. Possibly the most important change to early web pages was the introduction of the term *web 2.0* which bolstered an increase in JavaScript functionality, allowing web pages the ability to act dynamically [9]. JavaScript, possibly most of all, has developed and improved with updates to the language occurring on a yearly basis [10]. This has enabled a transition from websites - a group of interlinked web pages - to web applications – advanced and complex software programs accessible on the browser.

This section will look at the characteristics that typically exist in modern web applications, followed by a closer look into their development where I will discuss the technologies frequently employed when building them.

3.2.1 Characteristics

There is no universal standard of characteristics for which a web application must exhibit to be classified as modern, however a number documents display some overlap when discussing modern web application.

Microsoft states that modern web applications, due to their high user expectation and demand, have to be available 24/7 from anywhere in the world and on any device screen while offering rich user experiences [11]. Google is in agreement to this statement, though uses different language stating reliability, engagement, and responsiveness as core concerns [12].

There is also agreement on modern web applications being both secure and fast, as well as scalable, which, as Zartis adds, can be achieved though automated testing and a modular approach to design [13].

3.2.2 Tools

To achieve the characteristics mentioned in the previous section, there have been tools developed for every aspect of the development process. These tools can then be bundled together and create what is known as a stack. The most common stack, known as a full-stack, used in web application development can be broken into three pieces: database, backend, and frontend [14]. This section will look at each of these elements individually.

A popular full-stack option known as the MEAN (MERN or MEVN) stack can be found in figure 8.

3.2.2.1 Database

For data intensive applications, the database is an essential piece in the development structure. The two types of database used most commonly is modern web development are relational, and document oriented. Relational databases, such as MySQL[13], keep the data in a structuralized form while document oriented databases have a looser structure. The latter of these has become an increasingly popular solution due to its format closely modelling that which it is dealt on both the server and client side of the application. MongoDB[14] is the typical solution when deciding to use a document oriented database [14].

3.2.2.2 Backend

There is an abundance of choice when it comes to tools existing on the backend of web applications and it typically comes down to the programming language in which the developers are most comfortable with. For PHP development there is the Laravel[15] framework, if Python is the language of choice Django[16] and Flask[17] are typically chosen, while JavaScript developers will often choose the Express[18] framework [14].

The latter of these is worth mentioning more as JavaScript is a client-side language, however, Express is built on NodeJS[19] which is not a language in its own right, but a means of executing JavaScript on the backend of an application. It is therefore an extremely common backend choice due to the convenience and practicality of having JavaScript on both the server and client side of the application [14].

3.2.2.3 Frontend

Unlike the backend, there is only one compatible programming language for the frontend: JavaScript. However, there are many frameworks in which to choose from with the most popular being React[20], Angular[21], and Vue[22] [14]. Outside of the syntax used, there is not much difference in the frameworks as they each utilize the idea of components. Components generally take an input and changes behaviour, which will manifest as a change in the User Interface (UI), based upon this input [15].

---

[13] MySQL: https://www.mysql.com/
[14] MongoDB: https://www.mongodb.com/
[15] Laravel: https://laravel.com/
[16] Django: https://www.djangoproject.com/
[17] Flask: https://flask.palletsprojects.com/en/2.0.x/
[18] Express: https://expressjs.com/
[19] NodeJS: https://nodejs.org/en/
[20] React: https://reactjs.org/
[21] Angular: https://angular.io/
[22] Vue: https://vuejs.org/

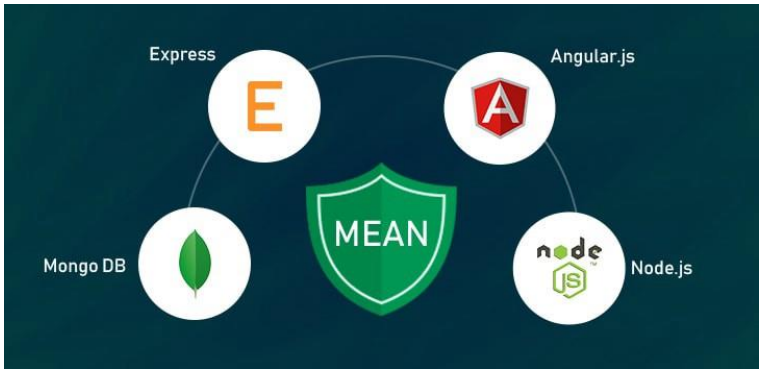

Figure 8 - The MEAN (MongoDB, Express, Angular, NodeJS) stack is one of the most used for web applications development. Angular can, alternatively, be replaced with React (MERN) or Vue (MEVN).

3.3 Project ACE and Build Tools

With a deeper understanding on ACE and modern web applications developed, a definition on the technologies used in the construction of the project may be discussed.

Before deciding on the specific methods to implement when building the project, I had a preference on making the application one that is entirely client-side (only consists of a frontend). The project is, by nature, not data intensive and therefore has no requirement for a database. The challenge would instead come from not utilizing a backend, this might not pose as a problem if a simple ACE predicting technique such as template matching was to be implemented, however, I had preference on utilizing a neural network from both a learning and performance perspective. The reason for this posing a challenge is that the backend typically carries out machine learning tasks due to the potential for increased power and speeds from servers [16].

With the latest developments of the TensorFlow.js[23] (TFJS) library, a machine learning library enabling neural networks to be run in the browser, I felt that it would be worth attempting a full client-side approach. Doing so would come with the benefit of requiring no server and database, thus no additional costs and issues that may arise in servers such as load balancing.

Figure 9 presents the three high-level features available in TFJS.

[23] TensorFlow.js: https://www.tensorflow.org/js

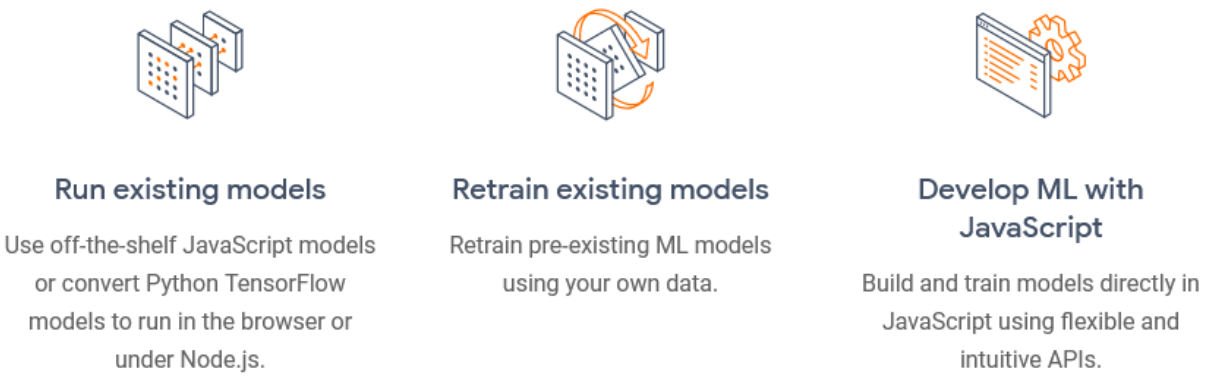

Figure 9 - The features available with TFJS

3.3.1 ACE

As previously mentioned in Chapter 2, building and training an ACE model from scratch was out of scope for the project. It was therefore a requirement to find an existing pre-trained model and implement it into the application.

This wasn't an easy task due to the limited availability of trained ACE models openly available on the web. Fortunately, TFJS has the ability to convert models trained in Python that used the TensorFlow[24] or Keras[25] libraries. I was able to find two such pre-trained models on GitHub, however only one of these had a usable[26] pre-trained model: pyace[27].

Pyace is the Python implementation of an ACE originally built in MATHLAB. Deng, the builder of both, states that the ACE is unlike other approaches as it separates the chord segmentation and chord classification into two distinct tasks. The former is done through the passing of chromagrams, obtained through CQT, into a HMM with Gaussian Emissions [4]. This is an extension to the HMM previously mentioned, and is typically done through a 12-dimensional Gaussian distribution where the probability of a chord emitting a chromagram frame is set using a 12–dimensional mean vector for each chord and a collection of covariance matrices for each chord [1]. These terms will be further explained in Chapter 4 where they are required to be initialized in the context of this project.

As was also stated in the Chapter 2 defined scope: implementing pre-processing steps, such as the chromagram extraction and HMM, from scratch was out of scope. While libraries for such tasks don't

[24] TensorFlow (Python version): https://www.tensorflow.org/learn
[25] Keras: https://keras.io/
[26] This was due to one of the models utilizing a compression technique rendering it unusable with TFJS.
[27] Pyace GitHub Repository: https://github.com/tangkk/pyace

exist in abundance, I was able to find the audio processing library Meyda[28] which utilizes STFT rather than CQT to obtain chromagrams, and a library allowing for the creation of HMM with Gaussian emissions based on the TFJS library[29].

The chromagrams are then passed into the neural network for classification. Deng offers a couple of neural network implementations, specifically a feed-forward model and a recurrent model. The recurrent neural network (RNN) is used in this project, chosen due to Deng statement of it being the most promising of the two [4].

A visual guide on the workflow of the pyace ACE is found in figure 10.

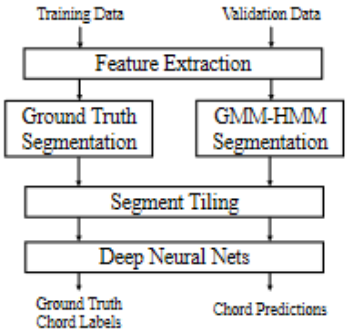

Figure 10 - The pyace ACE framework. Source [4]

3.3.2 Application Tools

Due to the their being no requirement for a database and backend there is less choices to be made on the projects build tools. As previously mentioned, there are three frameworks which are prominent in modern web application development. I chose to use the React framework for the application due to a preference in its syntax and its popularity in the community which has the benefit of offering an abundance in supported libraries.

Using React will allow the project to be built as a Single Page Application (SPA), making it more engaging for the user due it obtaining a native application look and feel [16]. It will allow it to be built in a modular fashion thorough the use of components which will keep the project clean and aid in its scalability. It also contains methods to compress the project which will keep the applications bundle small and assist in its speed.

[28] Meyda GitHub Repository: https://github.com/meyda/meyda
[29] Node HMM with Gaussian Emission GitHub Repository: https://github.com/nearform/node-hidden-markov-model-tf

Finally, GitHub[30] will be employed as version control for the application which will keep it secure in terms of accidents that may occur in development. It will also allow the project to be hosted via GitHub Pages[31] and to utilize Continuous Integration, a common methodology used in modern software engineering, through GitHub Actions[32].

[30] GitHub: https://github.com/
[31] GitHub Pages: https://pages.github.com/
[32] GitHub Actions: https://github.com/features/actions

Chapter 4: Building the Project

This chapter outlines the development of the project and goes into detail on the various technical aspects involved. It is divided into three sections, each representing a fundamental piece of the developmental process. The first, Model Predictions, discusses the effort made on the ACE side of the project. This is followed with a section on the client-side application, titled Building the Application, and a final section, titled Finishing Touches, detailing the final work required to complete the project application.

4.1 Model Predictions

As mentioned in the previous chapter, the project can be divided into two elements: the ACE, and the client-side application. This section will explore the first of these, detailing the efforts required to convert the previously obtained Python model (pyace), the pre-processing steps required on the input audio, and the work in obtaining predictions from the model.

This part of the development was the first to be carried out due to the ACE being at the heart of the application. It was therefore important to ensure it was functional before building an application around the system.

4.1.1 Converting the Model

The first step of the process was to convert the pyace model into a format compatible with TFJS, which in turn allows it to be used in the browser. This requires the use of the Python version of tensorflowjs[33], which was installed using the Python package installer, pip[34]. Once installed, the trained model was loaded using *Keras*, and converted through the tensorflowjs method:

```
tfjs.converters.save_keras_model(model, target_directory)
```

This produces a folder, in the specified target directory, containing the model (dataflow graph) in JSON format and a group of binary weight files known as shards.

[33] TensorFlow.js Python package (for model conversion): https://pypi.org/project/tensorflowjs/
[34] Pip package: https://pypi.org/project/pip/

A demo environment was then created in NodeJS which allowed me to test the model conversion by attempting to load it using the TFJS library, downloaded by way of the node package manager (NPM)[35]:

```
Const model = await tf.loadLayersModel(model_path)
```

The above piece of code will asynchronously load the model from the path specified. It is required to be asynchronous due to its returning of a Promise, an object representing the requests completion or failure [18]. In this case, it returned a completed request for the converted model highlighting that the conversion process was successful.

4.1.2 Pre-Processing: Audio to Chromagram

The converted model expects an array of 6x12 matrices where each row in these matrices is a mean chroma vector over some time series [4]. The first step in accomplishing this is to obtain the chromagram from an input audio file. This was completed using the JavaScript Web Audio API[36], which allows for the manipulation of audio files, and the Meyda library, which allows for the extraction of chroma vectors.

Before I could begin using Meyda and extracting chromas, I had to first manipulate the audio, splitting it into chunks of equal length. The time-span for each chunk will effectively be the sampling frequency as each chunk will later have an equivalent chroma vector extracted. Combining these chroma vectors will then result in the audio's chromagram.

Splitting the audio requires a number of steps, the first of which requires the use of an Audio Context interface provided by the Web Audio API. This interface is defined as representing an audio-processing graph built from audio modules linked together [18] and contains many useful methods and variables. The first of which is the ability to set the sampling rate variable which had to be changed from 48,000Hz to 22,050Hz, as is done in the pyace code.

Once the Audio Context had been initialized, the input file could be processed. This involved retrieving the input audio file and converting it to an array buffer, an object representing a generic fixed-length raw binary data buffer (or byte array). This array buffer is then passed into a method provided by the Audio Context which will decode the audio data and return what may be known as an audio buffer.

This audio buffer can then be split into chunks and processed by Meyda. The chunk length is obtained through the buffer size, which is set to 512 in unison with the pyace code. A loop can then be run

[35] NPM: https://www.npmjs.com/
[36] Web Audio API Documentation: https://developer.mozilla.org/en-US/docs/Web/API/Web_Audio_API

which takes a chunk of the audio buffer and uses Meyda to extract the chroma. This chroma was then appended to a 2D array, which, at the end of the loop, is the audio files extracted chromagram.

4.1.3 Pre-Processing: HMM Segmentation

Once the chromagram has been extracted the HMM can be used for the chord segmentation. This means that the HMM will effectively be determining the positions in which a chord will begin and end. It will also make its own chord predictions at this stage, which will be investigated in Chapter 5 though are not required for this use-case.

Before using the HMM, there are a number of parameters that need to be set. These parameters are set using the NumPy[37] package in pyace, and while this package is unavailable on JavaScript I found that the TFJS library had many similar functionalities. The parameters for the HMM are as follows:

1. **Pi:** the starting probability of each chord, which in this case is set to 1/24 as there is 24 chords, each being equally likely to occur. Seen in figure 11.
2. **A**: This is a matrix representing the transitional probabilities between states. For example, the probability a C Major chord changes to a G Major chord. This is initialized as a 24x24 matrix of small numbers with 1's running along the diagonal, which is stating that the chord is most likely to remain the same rather than change. Seen in figure 12.
3. **Mu**: This is the mean parameters for each state, which in this case is the binary note representation of each chord. This representation has been seen earlier, in Chapter 3, in the case of template matching. The method used to generate 12 major and 12 minor chord is to start off with a base chord, for example C Major and C Minor, and to then shift each bit over by one. This then gives a C# Major and C# Minor chord and is repeated 12 times to produce the full set of 24 chords. Seen in figure 13
4. **Sigma**: the covariance parameters for each state. The shape of this particular parameter differs in shape to the HMM used in pyace. The pyace HMM has the representation as a 24x12 matrix where every value is 0.2. This is not a valid format in the JavaScript HMM and instead required a 24x12x12 matrix to be used, where each 12x12 matrix has 0.2 running along the diagonal. Seen in figure 14.

[37] NumPy: https://numpy.org/

```
Shape: 24

Tensor
    [0.0416667, 0.0416667, 0.0416667, ..., 0.0416667, 0.0416667, 0.0416667]
```

Figure 11 - The Pi shape and value

```
Shape: 24,24

Tensor
    [[1   , 1e-7, 1e-7, ..., 1e-7, 1e-7, 1e-7],
     [1e-7, 1   , 1e-7, ..., 1e-7, 1e-7, 1e-7],
     [1e-7, 1e-7, 1   , ..., 1e-7, 1e-7, 1e-7],
     ...,
     [1e-7, 1e-7, 1e-7, ..., 1   , 1e-7, 1e-7],
     [1e-7, 1e-7, 1e-7, ..., 1e-7, 1   , 1e-7],
     [1e-7, 1e-7, 1e-7, ..., 1e-7, 1e-7, 1   ]]
```

Figure 12 - The A shape and value

```
Shape: 24,12

Tensor
    [[1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0, 0],
     [0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0, 0],
     [0, 0, 1, 0, 0, 0, 1, 0, 0, 1, 0, 0],
     ...,
     [1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0],
     [0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0],
     [0, 0, 1, 0, 0, 0, 1, 0, 0, 0, 0, 1]]
```

Figure 13 - The Mu shape and value

```
Shape: 24,12,12

Tensor
    [[[0.2, 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  ],
      [0  , 0.2, 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  ],
      [0  , 0  , 0.2, 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  ],
      [0  , 0  , 0  , 0.2, 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  ],
      [0  , 0  , 0  , 0  , 0.2, 0  , 0  , 0  , 0  , 0  , 0  , 0  ],
      [0  , 0  , 0  , 0  , 0  , 0.2, 0  , 0  , 0  , 0  , 0  , 0  ],
      [0  , 0  , 0  , 0  , 0  , 0  , 0.2, 0  , 0  , 0  , 0  , 0  ],
      [0  , 0  , 0  , 0  , 0  , 0  , 0  , 0.2, 0  , 0  , 0  , 0  ],
      [0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0.2, 0  , 0  , 0  ],
      [0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0.2, 0  , 0  ],
      [0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0.2, 0  ],
      [0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0.2]],

     [[0.2, 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  ],
      [0  , 0.2, 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  ],
      [0  , 0  , 0.2, 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  ],
      [0  , 0  , 0  , 0.2, 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  ],
      [0  , 0  , 0  , 0  , 0.2, 0  , 0  , 0  , 0  , 0  , 0  , 0  ],
      [0  , 0  , 0  , 0  , 0  , 0.2, 0  , 0  , 0  , 0  , 0  , 0  ],
      [0  , 0  , 0  , 0  , 0  , 0  , 0.2, 0  , 0  , 0  , 0  , 0  ],
      [0  , 0  , 0  , 0  , 0  , 0  , 0  , 0.2, 0  , 0  , 0  , 0  ],
      [0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0.2, 0  , 0  , 0  ],
      [0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0.2, 0  , 0  ],
      [0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0.2, 0  ],
      [0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0.2]],

     [[0.2, 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  ],
      [0  , 0.2, 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  ],
      [0  , 0  , 0.2, 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  , 0  ],
```

Figure 14 - The Sigma shape and value

On top of these four parameters, there are a couple more which required initialization. These are named *states* and *dimensions* and are set as 24 (chords) and 12 (notes), respectively. Once set, the HMM was almost ready to predict requiring only for the input to be initialized.

The input into the HMM is a matrix with the shape: *observations* x *time* x *dimensions*. The first attempt at initializing these variables had the goal of replicating the pyace code whereby the observations is 1 (observing the entire song), while the time is the number of chroma vectors in the chromagram[38]. This, however, caused issues in the browser, frequently resulting in crashes and freezing.

In order to prevent this, the chromagram is broken into small equally sized matrices. These divisions can have a maximum of 1500 chroma vectors, which was found to be a cut-off point before browser slow-down was a recurring issue. The *observations* are set to the number of chromagram divisions and the *time* is the number of chromas in each. The potential drawbacks of this approach will be discussed in Chapter 5.

Another issue that arose when testing the HMM was in relation to TFJS compatibility. The model was converted using the latest TFJS version 3 and therefore requires this version to run, the HMM on the other hand utilizes TFJS version 1.7 and threw errors when version 3 was installed. To fix this issue, the HMM package was uninstalled from NPM and the GitHub files were directly downloaded. I then converted the syntax of these files to comply with the latest TFJS version and imported it into the project environment. Once the HMM dependencies were explicitly installed, it worked as intended.

Finally, once the input was entered into the HMM it produced the required output, which was simply a set of predictions at each time step. This was then passed into a function in order to find the chord start/end times (by analysing the starting and ending chromas occurring between each predicted chord) and thus segmenting the chromagram.

4.1.4 Pre-Processing: Segment Tiling

The final step of processing required tiling the segments. This effectively means taking the segments in which the HMM split the chromagram into and reducing each one into 6 chroma vectors which best describe it. As noted by Deng, the equalization of every input enables neural networks with fixed-length input [4].

To accomplish this, the chromagram, retrieved from Meyda, and the length of each segment, as identified by the HMM, are used as input into a function. This function loops through each segment

[38] For a typical 3-4 minute song, there is around 9000 chroma vectors in the chromagram.

of chromas and identifies if they are divisible by 6. If the number of chromas isn't divisible by 6, the final chroma vector is removed (if less than 3 away) or replicated (if greater of equal to 3 away) until it reaches divisibility.

Once the segment reaches divisibility by 6 (i.e is equal to 0 modulo 6), the segment is divided into 6 equal parts and the mean of each is calculated. These mean chroma vectors are then stacked to give a new 6x12 chromagram. This chromagram is then added to an array which holds the average chromagrams for each segment processed which serves as input to the model.

4.1.5 Predicting

Using the model to predict chords is a relatively straight forward process once the pre-processing steps are completed and the input is obtained.

Using the model variable obtained in section 3.1.1, we can simply call the model using:

```
const predictions = model.predict(input)
```

This returns an array for each input segment containing 25 real-valued integers. These integers represent the likelihood of each individual chord, where the first index of each array is the likelihood of a C Major chord and the final index represents the probability of it being an N chord (meaning no chord can be detected). The max value of each array can then be extracted and this gives the chord prediction for each segment detected by the HMM.

Once the model was outputting valid predictions I could then move onto the second part of the developmental process which involved building the client-side application.

4.2 Building the Application

The second aspect of the projects development relates to the client-side application. There were a number of stages present during this process in which this section covers in a logical fashion, beginning with the designing and planning of the application, before later moving into the implementation of the applications functionality.

4.2.1 Design

Before any code was written it was important to first outline a blueprint for the application. This allowed a solid visualization on how the application should look and feel to be built. To design the

application, I used a prototyping software, Figma[39]. Figma is a web application and allowed for the creation and iteration of multiple designs without having to write any lines of code.

In order to design the application, the pages (or screens) of the application had to be identified. They were determined to be:

1. **Landing** – the page in which the user is first greeted by when visiting the application. This page needed the applications logo and a short description on what the applications aim is, an input area in which a user can enter their song file, and a call-to-action button which will indicate the actions the user should take to move on.
2. **Loading** – the steps taken to generate predictions from the model are not immediate and there was therefore going to be a loading time.
3. **Main** – this is the page displaying the chord predictions and the one in which the user will likely spend most of their time on. It was required to have a clear display of the predicted chords, and a simple interface for which the user can interact with the songs playback.

A number of designs prototyped at this stage can be found in figure 15, while the final designs for the three pages can be seen in figure 16. The landing page design was chosen due to its flexibility. The application needed to be compatible on smaller screen sizes and a symmetric design makes it easier to scale down while keeping the same format across devices. The loading page design is clean and simple while also having a musical element to keep in theme with the application. The main page extends this clean design allowing the chords to be the main focus on the screen which is essential for beginner guitarists.

The name of the application, *Chordini*[40], was also chosen at this stage as can be seen from both figures 15 and 16.

[39] Figma: https://www.figma.com/
[40] A simple play on words between *chord* and the famous magicians name, *Houdini*.

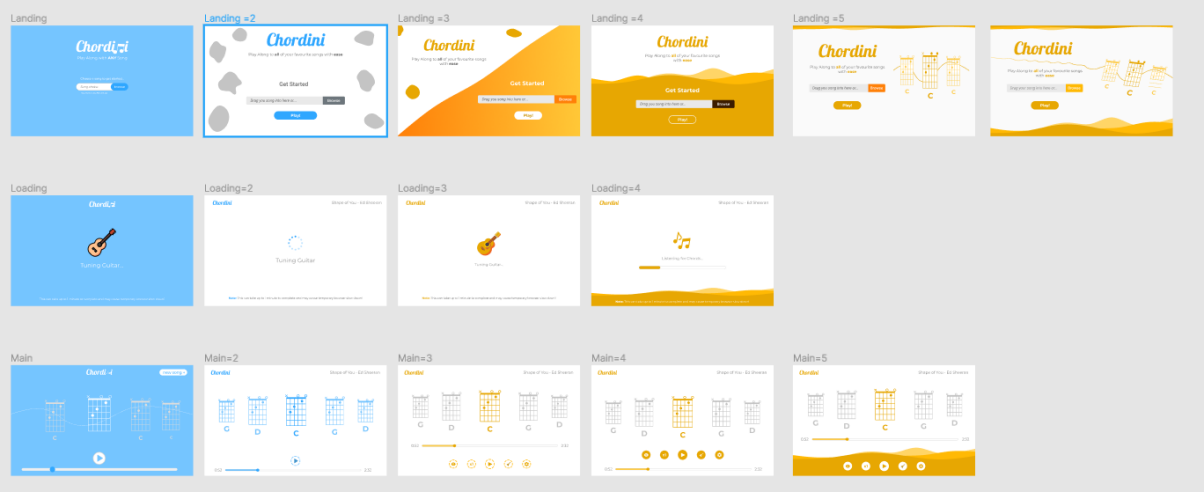

Figure 15 - A sample of the designs prototyped using Figma. The first row is the landing page, the second shows loading page variations and the final row displays the main pages prototyped designs.

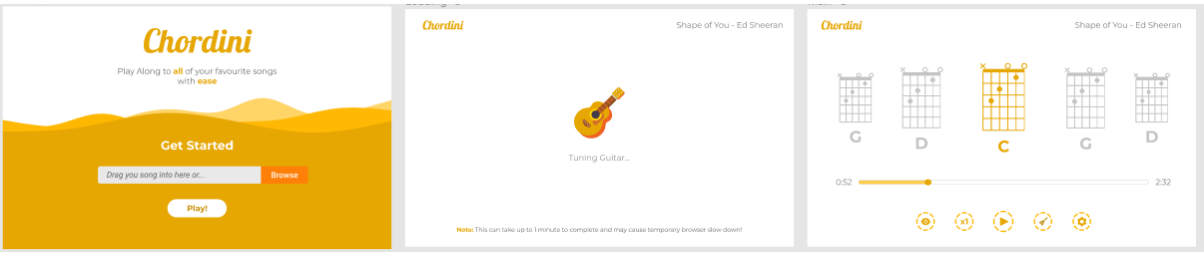

Figure 16 - The final designs chosen from those prototyped. From left to right these are the landing, loading, and main pages.

4.2.2 Environment Set-Up

With the designs in place, the applications build environment could begin setting up. This meant initialization of the code editor, application file hierarchy, and source control.

The first step in the environment set-up involved creating a new folder for the project and running the following in the command line:

```
npx create-react-app .
```

This is a tool developed by the React team and creates a React application in the folder location. It is noted by the React team as being the best way to get started building a single-page application, while also allowing for the scaling to many files and components, use of third-party libraries via npm, live-editing CSS and JavaScript in development, and optimization of the application for production [19].

Two additional tools installed by this React tool are Webpack and Babel. Webpack[41] is a bundler, which means it allows modular code to be written which is then bundled into a small package at run time in order to optimize load times. Babel[42] is a compiler, allowing for the use of modern JavaScript code to be used while remaining compatible with older browsers that may not support it [19].

Once the applications folder and file hierarchy was set up, attention could turn to the code and the style in which it will be written. When building an application that is built for both maintainability and scalability it is important to enforce a code style and use tools that can catch possible bugs as the code is being written. Two tools were used to achieve this: ESLint[43] and Prettier[44].

ESLint is a tool that statically analyses the code written and points out the problems that appear within it. It can be used in unison with a style guide, the popular Airbnb guide[45] one was used in this case, which keeps all of the code written for the application clean and consistent.

The second tool, Prettier, is a compliment to ESLint. It also enforces a consistent style, though more importantly, it formats the code to comply with the style guide upon saving of the file. This allows the focus to remain on the code and its functionality rather than having to additionally worry about its formatting.

4.2.3 Component Hierarchy

The final step before code can be written for the application is to identify the component hierarchy, as outlined in the *Thinking in React* documentation [20]. Components, as mentioned in Chapter 3, are a core mechanic of modern JavaScript frameworks.

In order to determine the components, the designs are used from the previous step along with the single responsibility principle which states that a component should ideally perform only function and if it grows, leading to a requirement of additional functionality, it should be decomposed into smaller components [20].

Figure 17 shows the components identified from the designs while figure 18 provides presentation of those identified and used in the project in a hierarchical format.

[41] Webpack: https://webpack.js.org/
[42] Babel: https://babeljs.io/
[43] ESLint: https://eslint.org/
[44] Prettier: https://prettier.io/
[45] Airbnb style guide GitHub Repository: https://github.com/airbnb/javascript

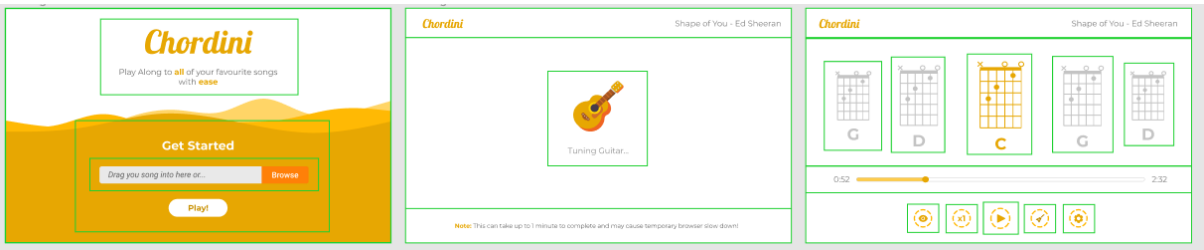

Figure 17 - The identified components from the Figma designs. The green boxes represent the possible components for the application.

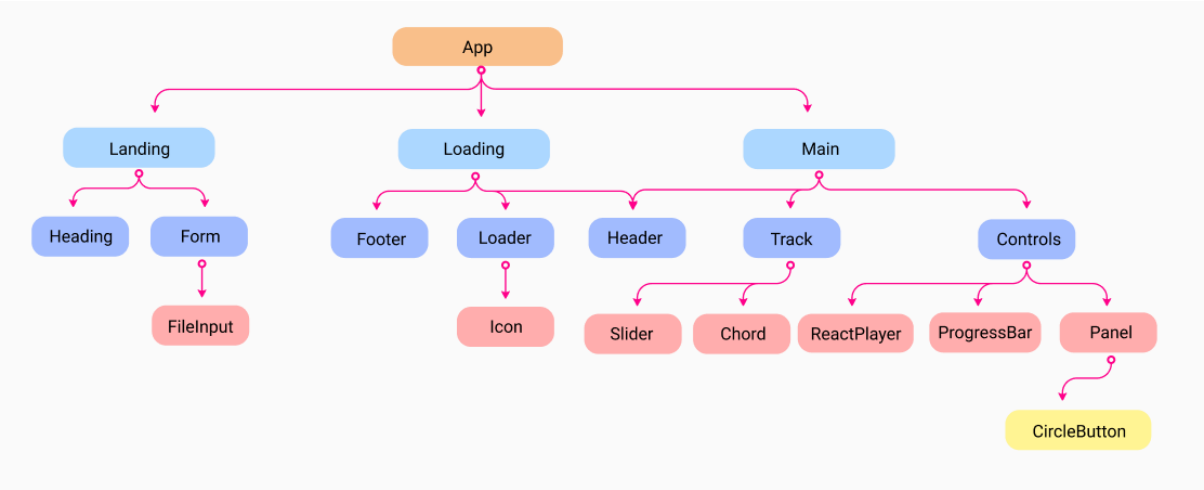

Figure 18 - The component hierarchy for the application.

4.2.4 Building a Static Version

As proposed in the *Thinking in React* documentation, once the component hierarchy has been established a static version of the application should be built. This required being concerned with the implementation of the UI over the functionality of the application. The React team expresses the reason for the decoupling of UI and interactivity, stating that the former generally requires a lot of typing and little thinking while the latter requires a lot of thinking and little typing [20].

Building a static version of the application was a relatively straight forward process, though there were some problems that arose in the designs which required changes to be made. The first of these was in relation to the song input present on the landing page, and the second was with regard to the media buttons found on the main page.

The first issue, with the song input, arose due to the difficult to style HTML file input tag. Producing a style similar to the design was a complicated task and I therefore opted for another solution: using a library. It was important, however, to choose these libraries wisely as many lack support, are difficult to use, and have large bundle sizes. A good metric, and one used in this project, was to choose libraries

based on their support and popularity in the community as this indicates a well-made library [21]. For the purpose of this task, the React-Dropzone library[46] was utilized to handle the task. The change this brought to the applications design can be seen in figure 19.

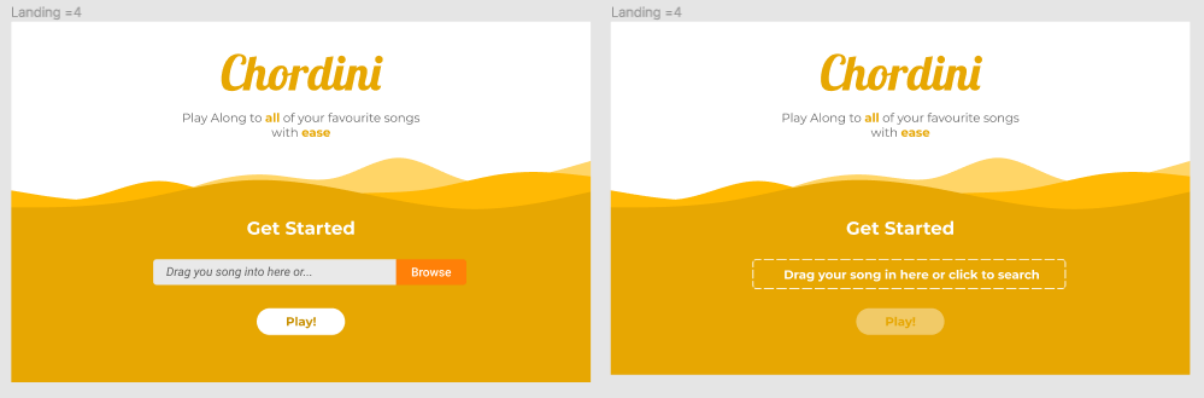

Figure 19 - The change in landing page design due to the use of React-Dropzone. On the left is the original design while the modified one is on the right. Notice the buttons also changes in design. This is for accessibility reasons as it is better at displaying a disabled button (which lights up when a valid song file is input).

The second issue that arose surrounded the media control buttons on the main page and, more specifically, their mouse effects. The designs in place didn't take into account changes that should occur when the mouse hovers over and clicks on a button. These are vital elements to consider for User Experience (UX) design as it lets the user know if an element is clickable. To accommodate for this, the buttons were redesigned to give a 3D effect through the use of background blur. This blur intensifies when the mouse hovers on a button, and the button inverts its colours when clicked. These mouse effects are visually presented in figure 20.

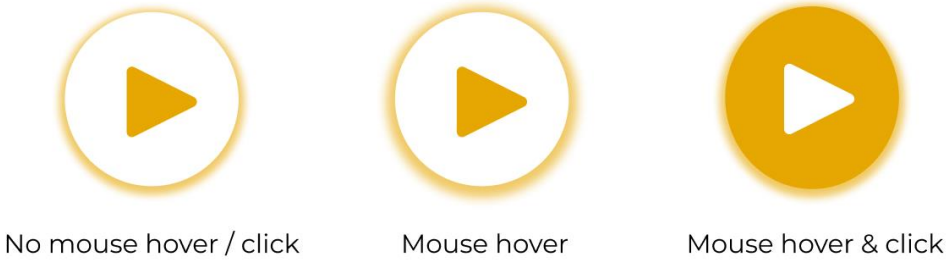

Figure 20 - The new button design with added support for mouse effects

When building the static version, one important concept is that of *props*. *Props* are a concept used in React that relate to the passing of data down the component hierarchy [20]. For example, in both the loading and main pages, the song name is displayed in the top right corner. When interactivity is

[46] React-Dropzone GitHub Repository (8.1k GitHub stars, +1 million weekly NPM downloads): https://github.com/react-dropzone/react-dropzone

implemented, the App component will retrieve this and pass it as a prop to the Loading and Main components. In the case of the static version, the *prop* value is simply hard-coded in.

4.2.5 Implementing Functionality

The final steps outlined in the *Thinking in React* documentation are concerned with the implementation of functionality in the static application [20]. In the previous steps, the React concept of *props* was installed into the application, however, functionality requires another core component of the framework: *state*.

State is the method in which React applications achieve interactivity. Whereas *props* are passed into components, the *state* is managed within the component and any change in the *state* will cause the component, and its sub-components, to re-render. To determine as to whether a piece of data should be classified as *state*, the React team outline three questions to ask:

1. Is it passed in, from a parent, via *props*? If yes, it is not likely *state*.
2. Does it remain unchanged over time? If yes, it is not likely *state*.
3. Can it be computed based on other *state* or *props* in the component? If yes, it's not likely *state*

If any of these questions yields a negative answer, then it is safe to assume that the data should act as *state* [20].

Using these questions, the minimal state and its location were identified for the application, as is found in figure 21.

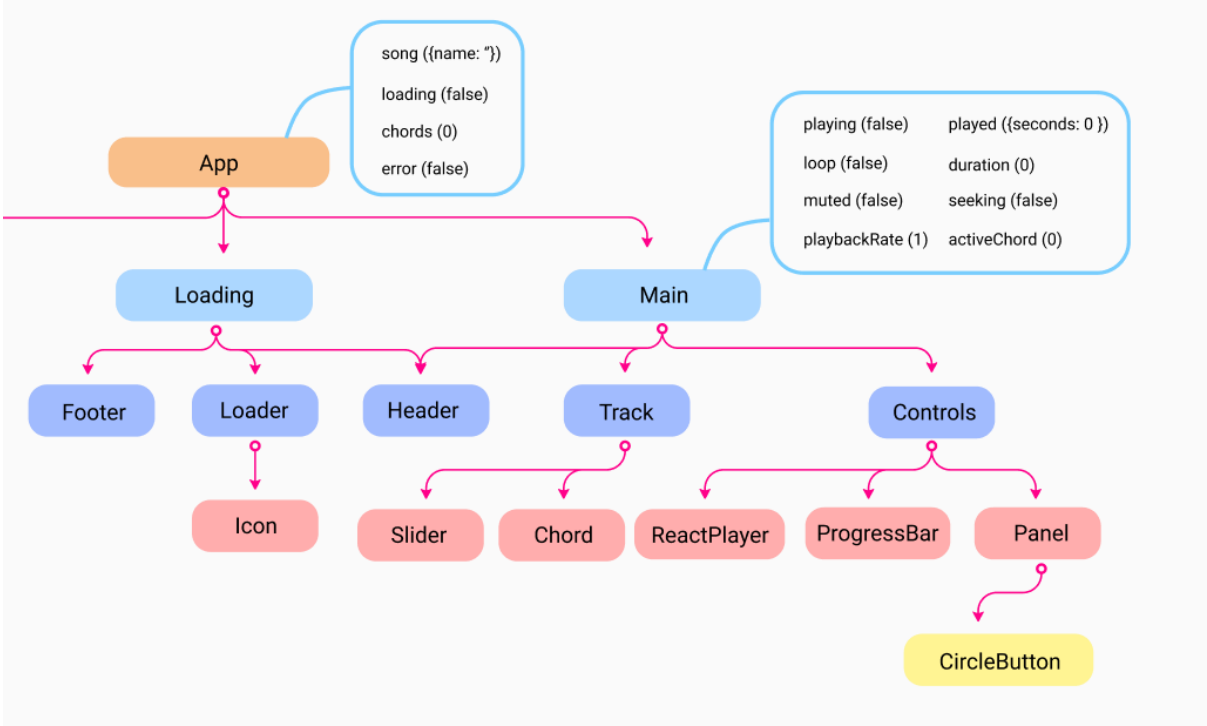

Figure 21 - The identified state required for the application. In brackets next to each state item is its initial state value.

To aid in readability, this section is further divided into sub-sections, each providing details on the relevant functionality that was to be implemented.

4.2.5.1 Retrieving the Song File

The first interactive element encountered by a user is the song input on the landing page. As earlier discussed, the input uses the React-Dropzone component which makes the retrieval of files a simple procedure.

In order to allow the dropzone component to function as intended, there are two pieces of information that must be specified. The first is the file types accepted, which is this case is set to all audio file types. The second piece of information is the actions, or functions, that should be called upon an accepted dropped file. In this case, we call a function present in the App component which will handle the file object by setting it to the *song* state.

This *song* state can then be used to pass the name of the song to the Loading and Main components for display in their headings (the Header component). This state is later used for processing the song, the next discussed step, and playing of the song which requires the passing of the file in URL format to the Main component. This was achieved using the URL object and looks like the following:

```
<Main song={URL.createObjectURL(song)}/>
```

4.2.5.2 Processing the Song

Once the song has been retrieved, work had to be done on processing the song which involved integration of all of the steps found in section 3.1, Model Predictions, into the React application. This was a relatively straight forward process as it mainly consisted of dragging the processing files into the applications folder, however, three additional steps needed to take place: function clean-up, conversion to the latest JavaScript, and returning of the model predictions as an array of Objects.

The ACE implemented at the start of this chapter was done in a demo environment where concern was on its functionality rather than the code used to accomplish it. The first step was, therefore, to clean up the functions making them readable, usable, maintainable, and overall complying with the ESLint rules and style guidelines.

The demo environment used in the model implementation was also performed without Webpack or Babel which caused compatibility issues in the code. In order to solve this, the code had to be modified and use the latest JavaScript techniques (ES6+). Figure 22 shows some of the code changes required during this stage.

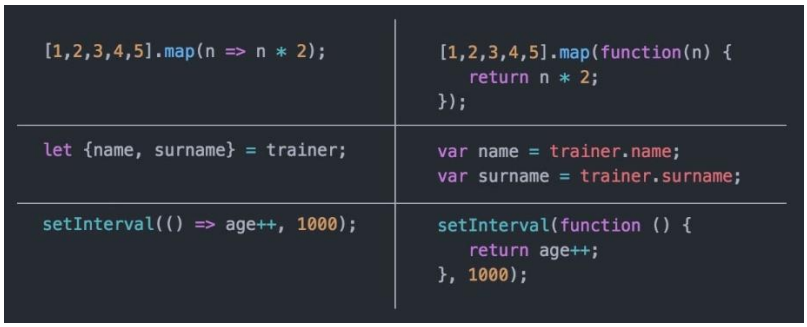

Figure 22 - Code examples showing the difference between ES6+ JavaScript (left) and the older ES5 version (right).

Finally, the model output had to be altered so that it produced a format that could be easily passed-to and read by components. This meant taking the output array of predictions and HMM timings and producing an Object for each prediction holding the name, end time, and duration of the chord. The array of these Objects could then be passed into components and presented an easier format for which their contents could be read.

Once these three steps were completed, the App component could import the functions and call them when the user hit the play button (found on the Landing page). The execution of the tasks can be found in figure 23. The end product of this function is the array of chord predictions for the song.

```
const getChords = async () => {
    setLoading(true);

    try {
        // load the model
        const model = await loadModel();

        // get audio buffer from song file
        const audioBuffer = await convertFile(song);

        // pass audio buffer into Meyda (will return chromagrams)
        const chromagrams = getChroma(audioBuffer);

        // pass chromagrams into HMM (will return predictions)
        const hmmPredictions = await getHmmPredictions(chromagrams);

        // pass predictions in for segmentation
        const modelInput = getModelInput(chromagrams, hmmPredictions);

        // get models predictions
        const predictions = getModelPredictions(model, modelInput);

        // set the chords state variable
        setChords(predictions);

        // set loading to false
        setLoading(false);
    } catch (err) {
        setError(true);
    }
};
```

Figure 23 - Function found in the App component to run all the tasks necessary to retrieve chord predictions for the input song. This function is run immediately upon pressing the play button in the landing page.

Another line to point out in the code seen in figure 23 is the setting of a loading state. When this is set to true, the App component will render the Loading component (page), and when the predictions are produced and the chords state is set, the App component will render the Main component (page).

4.2.5.3 Media Controls

As soon as the chords have been predicted the user is brought to the Main page which requires two core pieces of functionality, the first of which is the media controls. In order to implement the functionality of the controls produced in the static build, a React library had to be employed.

The package decided upon was React-Player[47], which offers a flexible component that allows the playing of URL's from Vimeo, YouTube and, most importantly, files. It contains *props* that provide direct access to the audio source which was useful for implementation of the applications play, mute and playback rate functionality. It also contains callback *props* which repeatedly update the current

[47] React Player GitHub Repository (5.2k GitHub stars, +300k weekly NPM downloads): https://github.com/cookpete/react-player

position in the song, used when implementing the track slider functionality. Finally, it has the ability to call a method upon the songs end which was used for the loop functionality in the application.

Once the state was applied in the Main component, it could then interact with the aforementioned *props* and thus the media control functionality was successfully implemented.

4.2.5.4 Moving the Chords

With the media controls working, the attention was turned to the second core element on the Main component: the chords. The Track component holds the chords and has the task of moving them in accordance to their predicted appearance in the song. The movement of components is an ideal use-case for a carousel component, which works in a similar sense to a slideshow, and the particular one chosen for the task was React-Slick[48].

The React-Slick library offers a Slider component of which contains options such as the number of slides to show (5), and the number of slides to scroll (1). The slider was then populated with the chords. This was done by mapping each object in the prediction array to a Chord component which displays the chords shape on a fretboard[49], and its name.

Once the slider had the chords in place, its auto-play option was utilized which allowed for testing of the sliders movement. This, of course, was only a temporary solution as the chords have to move dynamically and in accordance to the progress of the song.

4.2.5.4 Media and Chord Co-Operation

The final aspect of application functionality to implement was the co-operation between the media controls and the chord slider. This was completed through use of the *state* object, *ActiveChord* (initialized to 0, the index of the first predicted chord), in the Main component which allowed for the interaction of both the chords and progress of the song.

Previously, a function was called every 100 milliseconds that updated the current progress of the song. This function had to be extended to ensure the correct chord was shown in relation to the progress of the song. This was done by comparing the active chords end time with the current time-stamp of the playing audio file. If the active chords end time happened to be greater or equal to the current

[48] React Slick GitHub documentation (9.6k GitHub stars, +700k weekly NPM downloads): https://github.com/akiran/react-slick
[49] These were created using the chordpic website: https://chordpic.com/

progress (time) in the song, the Sliders current index was incremented along with the *ActiveChord state*. This allowed the Slider component to operate efficiently with the media playback, but only when no skipping and looping occurred.

In the case that the song was skipped or rewound, the closest possible chord with an end time less than the skipped/rewound to time was found. The Slider component is then called and told to go to the chord index identified and the *ActiveChord* state is set as such.

Finally, upon the tracks end, if the loop option is toggled a function calls the Slider to move to the next slide, which in this case will be a looping logo, and to move again. Now the slide index, due to an infinite loop *prop* being set in the Slider component, and the audio track is reset to 0 as well as the *ActiveChord* state.

Upon successful integration of the chords and media controls, the functionality and interactivity of the application was completed.

4.3 Finishing Touches

While the functionality of the application was completed, there was still work to be done to achieve today's standards for modern web applications. This section will detail each component that had to be worked on in order to achieve a modern web application.

4.3.1 Accessibility

The first step towards a modern web application was through its accessibility. An accessible application is one that usable by all potential users, regardless of auditory, cognitive, physical, and visual disabilities that may be present. One of the founders of the Internet, Tim Berners-Lee, has said: "The power of the Web is in its universality. Access by everyone regardless of disability is an essential aspect." [22]

There are a number of ways to implement accessibility into an application, all of which are noted in the WCAG by W3C. Some of these involve the use of alternative text for images, keyboard navigability, and the addition of labels which allows the web page to be read by a screen reader.

For this project, I utilized the A11Y Projects WCAG accessibility checklist[50], which places the WCAG in an easy to read and use format and made the implementation of accessible design and easier task to conquer. A section from the A11Y Projects checklist is seen in figure 24.

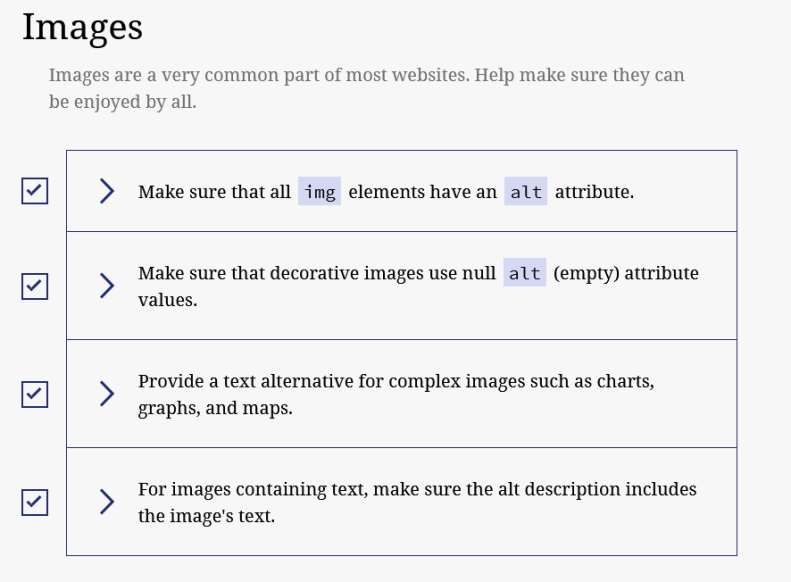

Figure 24 - The image accessibility section of the A11Y WCAG Checklist

4.3.2 Mobile Compatibility (Responsiveness)

The next step towards a modern web application was to make the application usable on all screen sizes. With the vast increase in both tablet and mobile devices using the web there is great importance on web applications scaling to meet these user demands.

In a similar manner taken when building the application, the first step of the process was through the prototyping tool, Figma. This allowed an idea to be built of how the application should look on smaller devices. While many changes only require a smaller font due to the symmetrical nature of the design, there were a couple of additional design considerations to take into account, particularly on mobile devices. For example, the name of the song file appearing in the top right of the screen would not work on smaller devices due to their limited horizontal space. This was also a problem with the displayed chords and media buttons on the Main page.

To accommodate for the limited space on mobile devices, the song name was removed in favour for a centred logo. The chords, typically displaying 5, were shortened to 3 on tablets, and 1 on mobile screens. Finally, the media buttons are altered on mobile devices, in that the speed controls are shifted

[50] A11Y Accessibility WCAG Checklist: https://www.a11yproject.com/checklist/

down to allow for the same functionality while keeping the same design elements. The three mobile designs can be found in figure 25.

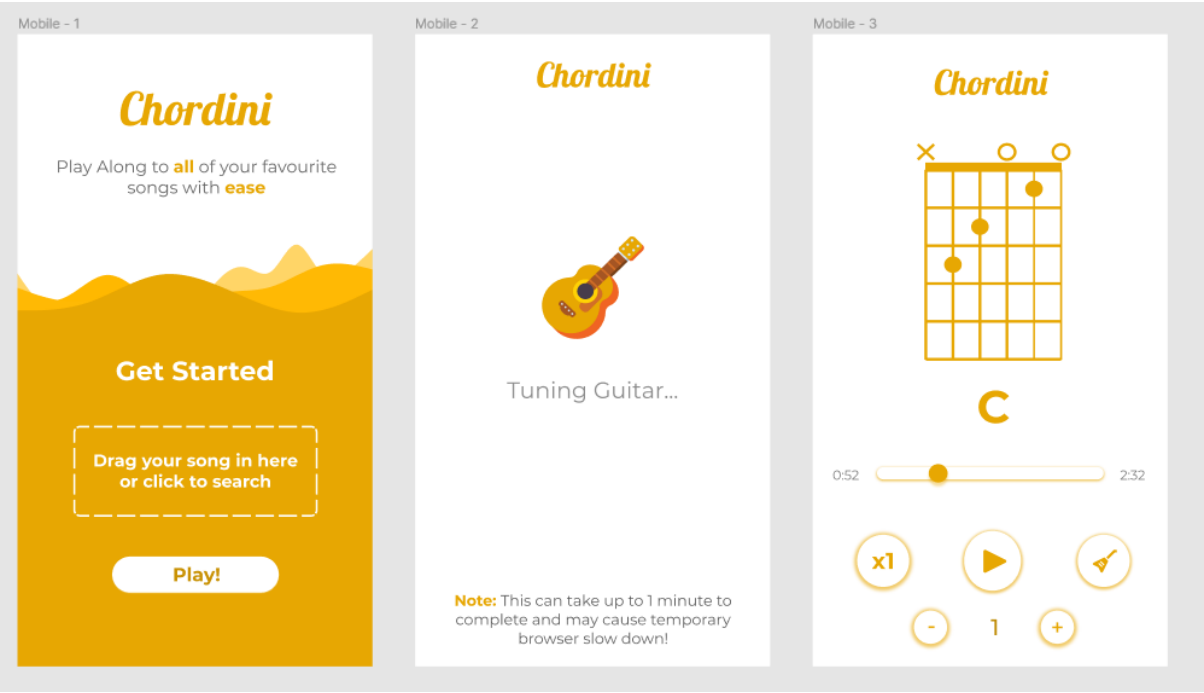

Figure 25 - The application designs scaled down for mobile devices. The landing (left), loading (middle), and main (right) pages can be seen.

Implementing these changes required the use of CSS media queries which allows style changes to occur when a screen width threshold is reached. This, however, only allows for static changes and is unable to accommodate for the dynamic pieces of the application such as the buttons and chords. To accomplish the dynamic changes required the React library, React-Responsive[51], was used. This allowed changes to occur in the applications dynamic code based on threshold values similar to the ones present in the CSS media queries.

4.3.3 Unit and Integration Tests

With the accessibility and application responsiveness in effect, the application could begin to be tested upon. Unit (testing individual components in isolation) and integration (testing the integration of multiple components) tests were utilized at this stage, though an additional testing technique – end-to-end testing – was carried out, and is detailed in Chapter 5.

[51] React Responsive GitHub Repository (5.5k GitHub stars, +300k weekly NPM downloads): https://github.com/contra/react-responsive

The React team recommends two testing libraries: Jest, and the React Testing Library. These libraries have different methodologies compared to classic unit and integration testing frameworks in that you only test what can be seen by the user [23]. This means that there is no ability to test functions that generate an output unseen to the user. There is, therefore, a blurred line between unit and integration tests when using the libraries as many tests can qualify as both.

A number of tests were carried out on the application such as ensuring the correct page was showing (by checking for core elements or words on the screen such as “Tuning Guitar”, etc.), and that the functionality of the audio buttons worked as intended (by checking if the button changes its logo, or if the speed label changes correctly according to the number of clicks).

4.3.4 Production and Deployment

When the tests were written and, more importantly, passing, the concern could be passed to the conversion of the application into a production build and subsequently deploying it on the web.

Before running the NPM command in which an optimized build of the application is created, the packages were first inspected and unused ones were removed. This step also allowed for a high-level look at the packages used in the project and their sizes. This was done using the webpack-bundle-analyser[52] tool of which the output can be seen in figure 26. One particular package that can be pointed out is React-Player, which on further review of the documentation could have its package size reduced by only importing the URL types required, which in this case is the File type. Running the package analyser after modifying this, reveals a decrease in the React-Player package size from 201.2kb to 80.32kb. While this may not cause any great performance impact on the application, such measures can be useful for large scale applications looking to achieve the best performance possible.

[52] Webpack-bundle-analyzer (+3.5 million weekly NPM downloads): https://www.npmjs.com/package/webpack-bundle-analyzer

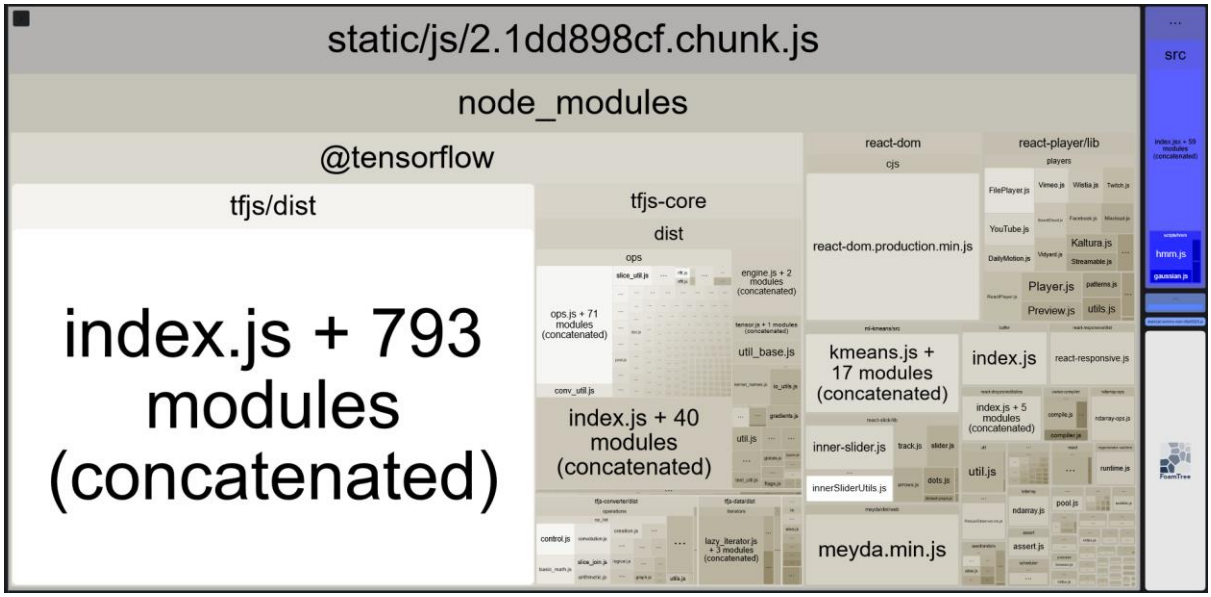

Figure 26 - The output produced by the webpack bundle analyser. The TensorFlow library is clearly the largest of the packages (all items under the @tensorflow heading), while the application code in relatively small (the purple box on the far right). The React Player bundle (seen on far right of beige box), was later reduced in size.

Once the packages had been inspected and their sizes lessened, the build of the application could be created using the command:

```
npm run build
```

Upon it successful completion, the build folder contains a single CSS and JavaScript file containing all of the applications CSS and JavaScript compressed into a small package. This has the effect of reducing the number of HTTP requests a user is required to make upon visiting the application, which has the knock on effect of speeding up the applications load times.

With the build folder created, the application was ready to be deployed using GitHub Pages. This required installing the GitHub Pages NPM package, gh-pages[53], and completing a number of additional steps, which were all detailed in the React Deployment guide [24]. Once the process was completed, a new branch was brought into existence on the GitHub repository[54] called *gh-pages* which contains the applications build folder. This is then the branch that GitHub Pages hosts the application.

4.3.5 Continuous Integration

The final part of project development involved the addition of continuous integration (CI). CI is a coding philosophy used in software development with the goal of establishing a consistent and

[53] GitHub Pages on NPM: https://www.npmjs.com/package/gh-pages
[54] This projects (Chordini) GitHub Repository: https://github.com/lukegib/chordini

automated way to build, package, and test applications. While CI is not completely necessary for solo development, including it in the application adds to the scalability of the project which is a core aspect of modern web applications.

The steps (pipeline) of CI are found in figure 27.

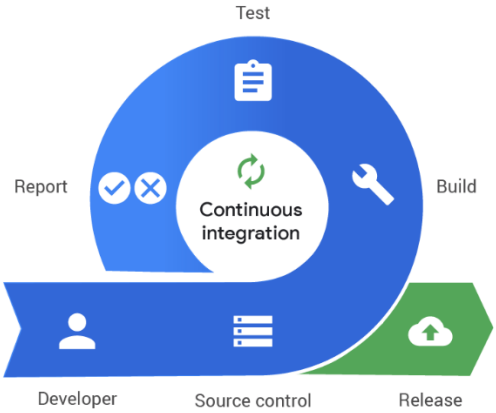

Figure 27 - The CI pipeline.

There are a number of tools that may be used to accomplish a CI pipeline, though the one used in this case is GitHub Actions due to the ease of its integration with the existing repository and deployment.

Upon specification of the application type, NodeJS, GitHub Actions provides a skeleton yaml file which defines the steps of the pipeline [25]. The altered yaml, seen in figure 28, displays the steps defined for this projects pipeline which are to:

1. Set up node environments running versions 12, 14, and 15 (line 16-18). These are the current supported versions.
2. Install the project dependencies (line 29-30). A failure here could indicate a corrupt, missing, or incompatible package.
3. Run the unit and integration tests, which were constructed in the earlier section 3.1.1 (line 32-33). A failed test will abort the process.
4. Build and deploy the application (line 35-48).

```yaml
jobs:
  build:
    runs-on: ubuntu-latest

    strategy:
      matrix:
        node-version: [12.x, 14.x, 15.x]

    steps:
      - name: Checkout Repository
        uses: actions/checkout@v2

      - name: Use Node.js ${{ matrix.node-version }}
        uses: actions/setup-node@v2
        with:
          node-version: ${{ matrix.node-version }}

      - name: Install Dependencies
        run: npm i

      - name: Run Tests
        run: npm test

      - name: Build
        run: npm run build

      - name: Deploy
        run: |
          git config --global user.name 'admin'
          git config --global user.email 'admin'
          git remote set-url origin https://${github_token}@github.com/${repository}
          npm run deploy
        env:
          user_name: 'github-actions[bot]'
          user_email: 'github-actions[bot]@users.noreply.github.com'
          github_token: ${{ secrets.ACTIONS_DEPLOY_ACCESS_TOKEN }}
          repository: ${{ github.repository }}
```

Figure 28 - A chunk of the yaml file showing the defined steps to take upon pushing a change to the repository.

Upon successful completion of all these steps, the pushed change which initiated the pipeline will be deployed to the live application.

Once this was successfully set up and worked as intended, of which the result for a commit can be seen in figure 29, the projects development was complete.

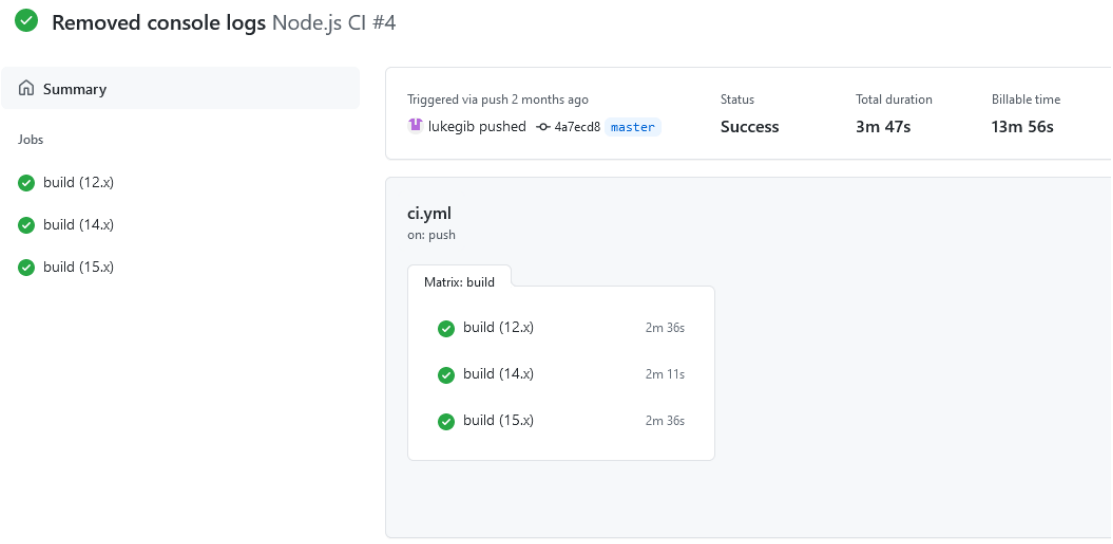

Figure 29 - A successful completion of the CI pipeline upon pushing the change "Removed console logs".

Chapter 5 – Evaluation

With the projects research and development outlined, this chapter carries out an evaluation into the projects components. The first of these components is the ACE, whereby experiments are run on both of the projects models (HMM and neural network), and those in which they are based off of (i.e. the pyace models). For evaluation of the second component, the application, tests are run to verify each step in the users path when using the application along with tests on its performance and device compatibility.

5.1 ACE

To measure the success of the ACE there are a number of details which can be studied including chord accuracy and timings. This section will look to carry out such an examination with the first section stating the experiments conducted followed by a second discussing the results obtained from said experiments.

For ease of reference, the project is referred to by the name *Chordini* in this section.

5.1.1 Experiments

In order to obtain results in which sufficient conclusions can be drawn, the implemented ACE was compared against the original one in which it is based: pyace. Comparing the two models against a ground truth set of chords will allow an understanding to be built on the effectiveness of the implementation.

To further this, the final ACE predictions, i.e. ones obtained through the neural network (referenced as NN for the remainder of section 5.1), were not the only ones considered for comparison. The HMM plays a vital part in the pre-processing of the song data and also generates its own set of predictions. Adding this into the experiment will allowed me to examine its performance in comparison to the NN and if there is a large or small leap in overall prediction performance.

Typically, researchers evaluating an ACE system will utilize automated testing where many songs can be quickly tested. However, due to the ACE in this project resting inside of a web application this form of testing is not possible and thus manual testing was executed.

A set of 5 songs were chosen from the McGill Billboard dataset, found in table 3, where emphasis was on chord and genre diversity, though as stated in Chapter 3, these are limited in ACE datasets. These songs were then input into both ACE systems, and the results were recorded for both of their HMM and NN predictions.

ACE Test Cases		
#	**Artist**	**Title**
1	Talking Heads	And She Was
2	Paul Simon	You Can Call Me Al
3	Eric Clapton	Promises
4	Lynyrd Skynyrd	Sweet Home Alabama
5	Aretha Franklin	Oh Me, Oh My (I'm a Fool for You Baby)

Table 3 - The songs used for the ACE experiments.

5.1.2 Results

The results are outlined in the three tables: 4, 5, and 6. The first of these, table 4. provides the overall accuracy achieved by each of the models on the 5 test songs, with the overall average also shown. The second table, table 5, displays the three most frequently occurring chords predicted by each model and those most frequently occurring in the each test song. Finally, table 6, summarizes the total chord changes made by the models compared to those in registered in the actual songs.

	Model Accuracy (as %)			
Song No.	**Chordini HMM**	**Chordini NN**	**Pyace HMM**	**Pyace NN**
1	26.31	41.82	47.09	47.09
2	18.43	14.31	22.12	28.85
3	16.89	9.58	51.14	44.29
4	14.28	18.13	29.12	42.49
5	10.3	14.48	17.82	18.66
Avg.	17.24	19.66	33.46	36.28

Table 4 – The model accuracy on each song along with the average accuracy across all 5 songs. This was recoded as the number of correctly predicted chord frames over the total chord frames.

	The Three Most Frequent Chords (No. Occurrences)				
Song No.	**Actual**	**Chordini HMM**	**Chordini NN**	**Pyace HMM**	**Pyace NN**
1	A (64) E (45) D (23)	E (31) A (22) C#m (7)	E (32) A (20) C#m (5)	E (27) A (26) A# (10)	E (27) A (26) C (13)
2	F (74) Bb (73) C (67)	C (41) F (35) Dm (15)	F (28) N (26) C (26)	F (29) C (21) Gm (19)	F (42) C (35) Gm (11)
3	G (21) C (19) D (17)	D (18) G (14) C (11)	G (21) D (10) F#m (7)	G (21) C (18) D (16)	G (28) C (18) D (13)
4	C (61) D (58) G (58)	D (40) G (30) C (24)	D (35) G (29) N (27)	Dm (47) C (44) G (33)	C (58) G (42) D (35)
5	Bb (24) F (17) Eb (14)	Bb (31) F (24) C (16)	F (24) Bb (22) N (19)	Bb (25) F (22) Gm (21)	Bb (29) F (26) Gm (17)

Table 5 - The most frequently predicted chords predicted by each model. The ground-truth for the songs (2nd column) is also displayed along with the frequency each chords is predicted.

	Total Chord Changes				
Song No.	**Actual**	**Chordini HMM**	**Chordini NN**	**Pyace HMM**	**Pyace NN**
1	180	81	80	98	95
2	217	125	97	112	106
3	74	74	51	70	67
4	183	182	160	174	164
5	89	138	128	129	127

*Table 6 - The total chord changes occurring in the ground-truth song (**Actual** column) and those predicted by each model.*

From these results in table 4, we can draw the conclusion that none of the models performed particularly well on the songs in the test dataset. The Chordini HMM was the poorest of the models, with the NN implemented in the application only averaging a slightly better performance. In terms of accuracy, the pyace models performed better overall with an increase of almost 89% compared to the

Chordini models. However, on a larger picture they didn't perform as well as I might have expected, with an average of only 36.28% from the pyace NN.

The reason for the believing the pyace model would perform better is due to the results from the ACE in which it is based. Deng's experiments show the original MATLAB model obtains a 72% average accuracy[4], and while a decrease could be expected in the python port which has a comparatively small chord library compared to its parent, a almost 36% decrease in average accuracy was not one I expected.

Moving to table 5, there are some positives that can be taken from the Chordini models. While in a different order of frequency, both the Chordini HMM and NN tend to predict the majority of the chord shapes found in the test cases. For example, in songs 3 and 4, the Chordini HMM correctly predicts the appearing chords though doesn't do well in terms of the frequencies to which they occur.

Another observation that can be made from Table 5 is the frequency in which the Chordini NN tends to predict the N chord. Over songs 2 and 4, the network predicts the chord a total of 53 times, though from the ground-truth chord frequencies it is evident that these frequent silences don't exist. The reasoning for this might sit in the chroma extraction method used in the project. As was noted in Chapter 3, there are two common extraction techniques: STFT and CQT. CQT is the most commonly used for ACE NN's as is the case for pyace, however, STFT was the only one available in JavaScript. The fact that the NN used in the project was trained on CQT chromas and the one it is being provided with are STFT chromas may be the answer to the recurrent N chord problem. The additional fact of the pyace NN never throwing this N chord is further evidence that this might be the case.

Figure 30 shows a chroma vector extracted using STFT (via Meyda) while figure 31 shows the chroma vector extracted for the same time stamp using CQT (via pyace). It can be noticed that the CQT chroma vector has few high values and many low values while the STFT has entirely high (over 0.5) values.

```
0: 0.8359751351322283
1: 0.8315128175409449
2: 0.946418060470192
3: 1
4: 0.9822215760470343
5: 0.9400195428680749
6: 0.9054321230241961
7: 0.7992906155507562
8: 0.7549523636749677
9: 0.6903101308598592
10: 0.8028770826814162
11: 0.8364809122826122
```

Figure 30 - A chroma vector extracted through STFT using Meyda.

```
[0.3022316  0.23111615 0.33710882 0.34962067 0.27806494 0.36555108
 0.5604815  0.88593096 1.         0.5603038  0.2833258  0.25393558]
```

Figure 31 - A chroma vector extracted through CQT. This was extracted at the same time as the STFT chroma found in figure 30.

The timings of the chord changes, a vital element is assuring songs can be played-along to, can also be examined. Table 6 can be used to attain a general idea on the chord timings in songs by looking at the total number of changes. Models that predicted much more changes than the actual song will result in the timings being too frequent and thus the song will be difficult to play along to. Models predicting too little chord changes often result in holding onto chords for too long which will also result in poor chord timings. In table 6, it can be seen that the results are varied for the songs though each of the models provide similar numbers. Songs 3 and 4 resulted in the best performance, though the models predicted much too little changes in songs 1 and 2, and far too many changes in song 5.

Figure 32 shows the chord changes for the Chordini NN across the first 30 seconds of song 1 accompanied by the ground-truth, in more detail. The NN rarely gets the exact chord change time, often resulting in a slow prediction or completely missing the change. This could have been predicted when analyzing its total chord changes for the song in Table 6.

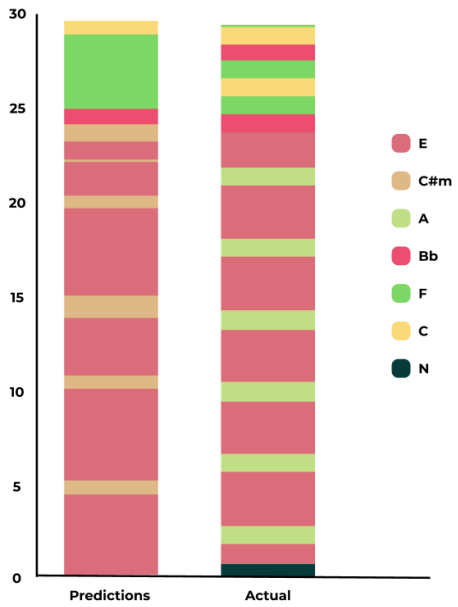

Figure 32 - The chord timings over the first 30 seconds of song #1 in table 3. The left bar is the Chordini neural network, the right shows the ground-truth chords and timings.

Upon final analysis, I believe the lower prediction accuracy of the Chordini NN can be explained by the lack of CQT extraction performed, with Meyda only providing STFT extraction. This potentially reveals

one of the limitations in working in JavaScript for such as task, while hugely popular there is limited libraries for dedicated machine learning tasks.

Another potential drawback can be witnessed through the occurrence of another question: the HMM's aren't pre-trained, so why does the Chordini HMM perform worse than that of the pyace HMM? I believe this comes down to the problem discussed in Chapter 4, whereby the HMM couldn't observe the entire song due to the browser crashing and freezing, therefore requiring to be split up into multiple parts. This would have meant a new chunk entering the HMM is viewed as a completely new song with all previous chunks unobservable. This likely resulted in a weaker performance and shows the limitations the browser encounters when dealing with computationally intensive tasks.

5.2 Application

The second component to evaluate is the application side of the project. For a thorough analysis of the application, the functionality, performance, and device compatibility are all required to be tested and experimented upon.

Similarly to the first section of this chapter, I will first out outline the tests to be carried out which will be followed by a discussion on the results obtained.

5.2.1 Tests

Previously, during the development process outlined in Chapter 4, unit and integration tests were utilized. While this form of testing is useful, it fails to test the applications functionality from a user's perspective. To accommodate for this drawback, end-to-end testing can be performed. This is a testing method, typically conducted manually, that involves testing an application workflow from beginning to end with the goal of replicating user scenarios. This allows the application to be validated for integration and data integrity [26].

To carry out this testing, a checklist was composed containing the key parts of the applications workflow in an order they would likely be encountered by a user. This checklist can be seen in table 7. The web application, BrowserStack[55], was also employed for this step which allowed testing of the application on systems and browsers I had no physical access to. Figure 33 and 34 shows BrowserStack running the application on devices I lacked physical access to.

[55] BrowserStack: https://www.browserstack.com/

#	Question
1	Do the pages (landing, loading and main) look consistent to the designs?
2	Can I enter a song on the landing page and click the play button?
3	Does the loading screen appear?
4	Does the main page appear displaying the song chords and controls?
5	Does the play/pause button work?
6	Does the mute button work?
7	Does the loop button work? (song loops to start if toggled)
8	Do the speed controls work?
9	Can I skip to a part in the song using the track slider?
10	Does clicking the logo take me back to the landing page?

Table 7 - A checklist used for the end-to-end testing of the application.

The list of browsers and operating systems (OS) in which the end-to-end testing was carried out can be found in table 8. Together, they cover 92.7% of web usage [27].

As mentioned in Chapter 3, one of the potential limitations of a client-side only approach to the project was the speed in which it can performs tasks, with machine learning typically taking place on the backend. The loading times for the application to perform the audio pre-processing and chord predicting were therefore recorded on each device and browser when carrying out the tests.

Operating System	Browsers				
Windows (10)	Firefox	Chrome	Opera	Edge	
Mac (Big Sur)*	Safari	Firefox	Chrome	Opera	Edge
Android (11)*	Firefox		Chrome		
iOS (14)	Safari		Chrome		

Table 8 - The operating systems (the particular version used is in brackets) and the browsers in which the end-to-end testing was carried out upon. The operating systems which I used BrowserStack for are symbolized with a ''.*

Finally, Lighthouse[56] was run on the application to obtain a deeper insight into its performance and build. Lighthouse is an automated tool offered through Google Chromes DevTools[57] that runs audits for performance, accessibility, best practices, and search engine optimization (SEO).

[56] Lighthouse: https://developers.google.com/web/tools/lighthouse
[57] These are a set of web developer tools built into the Google Chrome browser. https://developer.chrome.com/docs/devtools/

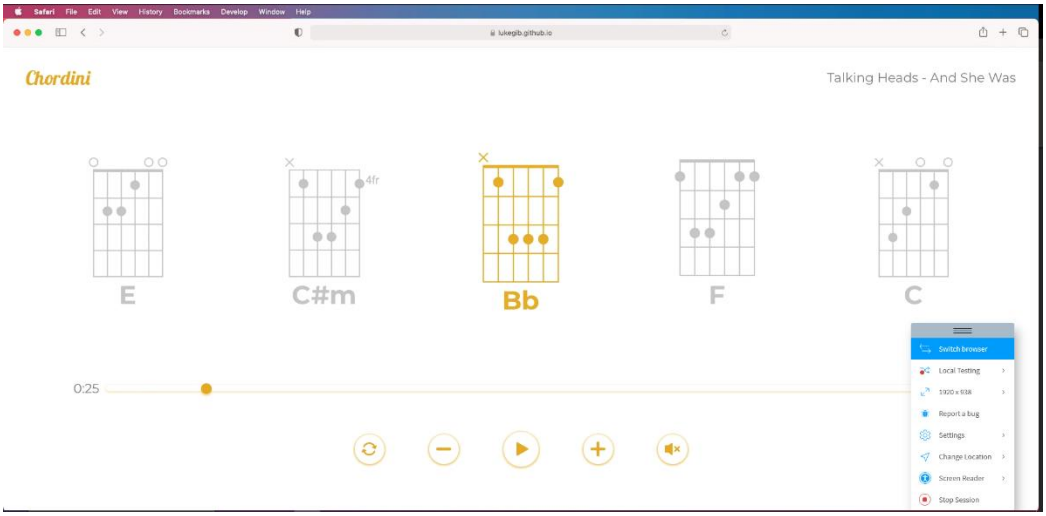

Figure 33 - Running the end-to end tests on a Mac (and Safari) through BrowserStack.

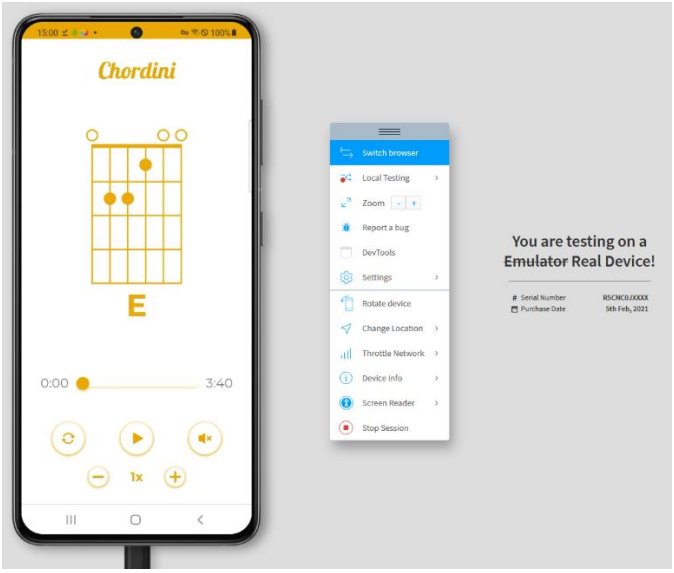

Figure 34 - Running the end-to end tests on an Android through BrowserStack.

5.2.2 Results

The first set of results to examine are those generated from the end-to-end tests. These can be found in table 9, where a tick represents the successful completion of each item in the checklist and a cross is caused by a failure to complete one or more questions in the checklist.

	SAFARI	FIREFOX	CHROME	OPERA	EDGE
WINDOWS	-	✔	✔	✔	✔
MAC	✔	✔	✔	✔	✔
ANDROID	-	✔	✔	-	-
IOS	✘	-	✘	-	-

Table 9 - Results for each OS and browser when performing the end-to-end testing checklist found in table 7.

From table 9, we can see that the application works on all operating systems and browsers except for those running on iOS devices such as iPhones and iPads. This failure occurs at step 2 of the workflow checklist, "Can I enter a song on the landing page and click the play button?", due to restrictions surrounding iOS devices in regard to browser file uploads.

Another barrier was encountered on the Safari browser running on the Mac. By default, the browser doesn't support WebGL version 2, the technology required by TFJS to run neural networks on the browser, and it is required to be toggled in the experimental feature settings. To avoid users running into this issue, a browser detection mechanism could be put in place that instructs those on Safari to toggle the feature.

The second set of results to be investigated are the ACE load times across the browsers and devices, of which can be found in table 10.

	SAFARI	FIREFOX	CHROME	OPERA	EDGE
WINDOWS	-	17.29	9.43	14.67	11.61
MAC	11.2	16	12.72	18.48	14.9
ANDROID	-	23.03	9.1	-	-
IOS	N/A	-	N/A	-	-

Table 10 - The average load times required by each OS and browser to complete the ACE element of the application. Song #1 was used from table 3 for each device and browser.

The results are quite surprising as they reveal that there is little difference in time between devices, and that the loading times are more resultant on the browsers. Possibly most surprising of all is that the mobile device running Android had, on average, the quickest loading time (when using the application on Google Chrome).

In terms of browsers, the reason for Chrome's increased speed may come down to possible optimization for TFJS as it is also developed by Google. This may also lead to the performances of both the Opera and Edge browser, which have only slightly slower times, as they are based on

Chromium[58] [29], the open-source codebase for web-browser which is also behind Chrome. Safari had the second best time overall, though was limited on device support. Safari is based on WebKit[59], in which Chromium was forked[60] from [29]. Firefox, while still performing quicker than expected, was the slowest browser overall. This could possibly be due to it having no relation to Chromium or WebKit.

The final set of result to examine are those generated by Lighthouse. The results, displayed in figure 32, show that the application performed very well is all the categories audited. In fact, the only non-perfect score was a 98 in performance whereby the tool has recommended dividing the JavaScript bundle generated through Webpack during the build and deploy stage discussed in Chapter 4. The tool also states that this task would reduce the initial application load time by 0.32 seconds.

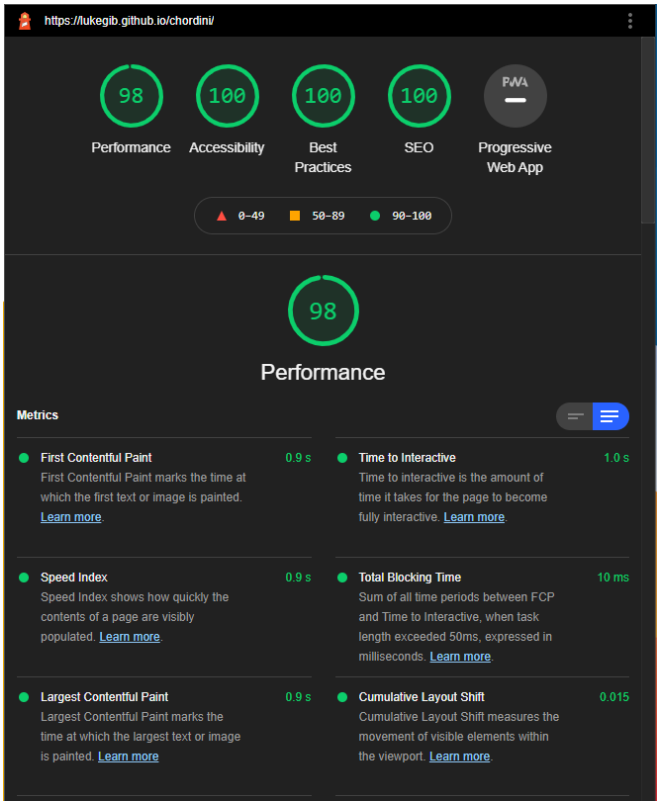

Figure 35 - The results from the Lighthouse audits. Additional information on the performance can also be seen.

In terms of accessibility, the application passed all 9 audits which included checks for accessible button names, and colour contrast ratios. It also passed the 17 audits on best practices, which includes the use of HTTPS, avoidance of JavaScript libraries containing vulnerabilities, and not logging any browser errors. Finally, it passed all 10 SEO audits which required having crawlable links, viewport tags, etc.

[58] Chromium: https://www.chromium.org/
[59] WebKit: https://webkit.org/
[60] Forking refers to the act of copying an open-source repository. It may then be experimented upon without affecting the original project.

Overall, I believe the results obtained from the application tests have been successful. With the exception of iOS devices, the application is accessible on all devices and has reasonably quick load times when performing the predictions, which was previously deemed as a potential drawback to the client-side approach. It can also be concluded through the Lighthouse audits that the application has good performance and is built in accordance to modern standards.

Chapter 6 – Conclusion

This chapter serves as a conclusion to the thesis, providing closing remarks on the project's success through the analysis of the goals presented in Chapter 2 along with a section on further work that could be completed on the project.

6.1 Goal Review

Chapter 2 outlined the goals and the outcomes that hoped to be achieved through their completion. To grasp an overall conclusion of the project's success the goals and outcomes are now returned to, and the success of each goal, as found in Table 11, is briefly discussed.

<table>
<tr><th>Goal</th><th>Success</th><th>Overcome</th><th>Success</th></tr>
<tr><td>Utilize technologies used to build modern web applications</td><td>✔</td><td rowspan="3">A web application compliant with today's standards. This means one that is:
1. Scalable
2. Responsive (accessible on different screen sizes)
3. Secure
4. Accessible to all potential users</td><td rowspan="3">✔</td></tr>
<tr><td>Be aware of WCAG during development.</td><td>✔</td></tr>
<tr><td>Be aware of the OWASP Top 10 Web Application Security Risks during development.</td><td>✔</td></tr>
<tr><td>Use prototyping software during the design phase of the application</td><td>✔</td><td>A thoughtfully designed application with a UI focused on ease-of-use.</td><td>✔</td></tr>
<tr><td>Implement an ACE into the web application that can predict chords when provided with an audio file</td><td>✔</td><td rowspan="2">An application providing the ability for users to enter an audio file (mp3, wav) and play along with the song through display of the predicted chords.</td><td rowspan="2">✔
&
✘</td></tr>
<tr><td>Display the ACE's predicted chords as they occur within the input audio file</td><td>✔</td></tr>
</table>

Table 11 - The goals and outcomes presented in Chapter 2 with additional measures of success (green tick shows success, red cross shows failure).

From analysis of the goals and outcomes, it can be concluded that the project was successful overall. The project application complied with the standards imposed on modern web applications, as was evidently demonstrated by the result found in Chapter 5.

Using Figma allowed a focus to be brought upon the UI of the application, which was identified as a clinical aspect of the application in order for it to be easy-to-use for beginners.

The only failing of the application comes from the final outcome. While it can't be deemed a complete failure due to the application successfully allowing the user to input an audio file, I believe it fails at allowing the user to play-along with the song. The ACE is successful at providing predictions, and so too is the application at displaying these predictions, however, the failing is a result of the predictions themselves as was seen in the ACE experiments in Chapter 5. The model wasn't accurate enough in predicting the songs chords and timings and it would therefore be difficult for beginners to play along with their input.

This having been said, I'm not sure there was more that could have been done to remedy the ACE. Even if it achieved the results of the pyace model seen in Chapter 5, I believe it would also be deemed as failing this outcome.

Overall, I am happy with the resulting project. Through its completion I learnt more on both neural networks and application development which will prove highly beneficial to my future endeavours. I believe the power that exists in browsers to complete machine learning tasks was also highlighted in the project which I would be interested in further exploring. The next section will further this statement by examining the further work that could be undertaken in relation to this thesis' topic and project.

6.2 Further Work

This section discusses two areas for which additional work on the thesis topic and project could be explored. The first of these relates to the ACE, and the second has focus on the application and possible feature additions.

6.2.1 ACE

The ACE implemented in the project was based off of Deng's pyace, which achieved an average accuracy of 36.28% in the testing carried out in Chapter 5. This means that even if the projects ACE

problems were worked on, in particular adding support for CQT[61] and implementing a new HMM, the peak performance for which it can achieve is capped at 36%.

The solution to this problem would therefore be on building and training an ACE neural network from scratch, a task that was out-of-scope for this project due to the efforts required. This path would also allow room for experimentation as many interesting solutions have been proposed in ACE research. One such example is the use of genre specific models, as it was noted by Lee that single models for a range of genres lead to poor generalisation with the use of genre information increasing performance by almost 10% [1]. This approach could also work out due to the ACE existing in an application as the user could be prompted to input the genre. This would reduce the need for a genre-identifying model, or the probabilistic method of testing all genre-specific models implemented by Lee [1].

Another path that could be taken is through the relationship between chords and the musical key. Prior knowledge of a musical key makes certain chords more likely than others and this also holds true for the other direction [1]. Perhaps a key-detection model could be experimented upon, where it is retrieved from the audio before being passed into the ACE model. Another possible approach might be in estimating the key from the ACE models predicted chords. This has been successfully applied by Shenoyand Wang, where the extraction of the key was performed with high accuracy (28/30) and the ACE accuracy increased by over 15% [1].

Other paths that may lead to a high performing ACE also exist, such as the use of bass notes, downbeat positioning, and common chord sequences. Each of these are discussed in more detail by Pauwels et al. [7] and could be worth attempting.

It might also be noted that Deng's approach of using a HMM and neural network was unlike others in ACE research [4]. Through analysis of the HMM results presented in Chapter 5, I believe it is fair to say that they did not provide notable performance and it may therefore be wise to return to allowing the neural network to predict both the chords and their timings, as is typically done [7]. This would also have the potential benefit of reducing the applications ACE prediction loading time by 55.4% as this is the average length, in percentage, the HMM took to segment the input audio[62].

[61] This is an open issue on Meyda, though it is unknown when/if it will be implemented as it has not been active since 2018: https://github.com/meyda/meyda/issues/77

[62] The chromagram extraction took 6.7%, and the neural network predictions took 37.9% of the total load time.

6.2.2 Application

While I believe work on the ACE for the project is of higher importance, there are some improvements that could be made on the application side of the project.

Research conducted in 2018 by the Royal Philharmonic Orchestra found that 9 in 10 children wanted to learn an instrument, with the most popular choices being guitar (45%) and piano (36%) [28]. The application currently has sole-support for guitar, so it would make sense to extend this support for piano which would cover 81% of beginner musicians. The chords for these instruments are essentially the same with the only difference being finger technique and position [29], and might therefore prove a relatively straight-forward task.

For other additional features that could be added, Capo, the Apple-exclusive ACE application mentioned in Chapter 1 can be looked to. While not an application aimed at beginners, Capo has features that could be helpful to this target group. Two such features are beat-tracking, and key changing. The first might offer an easier way for the user to know when the current chord is ending (like acting as a countdown), though adding this feature would also require work on the ACE side of the project as it currently doesn't support knowledge on the songs beats per minute. The second feature, key changing, would allow songs presenting difficult chords (in particular, barre chords [30]) to be transposed into those that are deemed easier to play for beginners.

Finally, it may be worth implementing features in the application for it to comply with Progressive Web Application (PWA) standards, an out-of-scope task mentioned in Chapter 2. Potentially, the greatest benefit the application could obtain from being a PWA is offline support. Due to it requiring no back-end, it is already a relatively light application and requires no calls to external services. It might therefore transition well into offline support, as if the neural network is stored on the users device the application is fully operational offline. Thus moving to a PWA would provide all of the benefits a native application offers while still allowing the same codebase to be utilized.

References

[1] M. McVicar, R. Santos-Rodríguez, Y. Ni and T. De Bie, "Automatic Chord Estimation from Audio: A Review of the State of the Art," *IEEE/ACM Transactions on Audio, Speech and Language Processing,* vol. 22, no. 2, pp. 556-575, 2014.

[2] C. Pérez-Sancho, D. Rizo and J. M. Iñesta, "Genre classification using chords and stochastic language models," *Connection Science,* vol. 21, no. 3, pp. 145-159, 2009.

[3] V. Zenz and A. Rauber, "Automaticchord detection incorporating beat and key detection," *Proc. IEEE Int. Conf. Signal Process. Commun,* pp. 1175-1178, 2007.

[4] D. Junqi, "Large Vocabulary Automatic Chord Estimation from Audio Using Deep Learning Approaches," 2016.

[5] C. Harte, Towards automatic extraction of harmony information from music signals, Ph.D. dissertation, Univ. of London, London, U.K, 2010.

[6] T. Fujishima, Real Tim Chord Recognition of Musical Sound: a System using Common List Music, ICMC, 1999.

[7] J. Pauwels, K. O'Hanlon, E. Gómez and M. B. Sandler, "20 years of automatic chord recognition from audio," in *20th International Society for Music Information Retrieval Conference*, Delft, 2019.

[8] Y. Ni, M. McVicar, R. Santos-Rodriguez and T. De Bie, "An end-to-end machine learning system for harmonic analysis of music," *IEEE Trans. Audio, Speech, Lang. Process,* vol. 20, no. 6, pp. 1171-1783, 2012.

[9] S. Grinakar, "Websites: past and present," enonic, 16 January 2019. [Online]. Available: https://enonic.com/blog/websites-past-and-present. [Accessed 22 4 2020].

[10] C. Kopecky, "JavaScript Versions: How JavaScript has changed over the years," educative, 18 December 2020. [Online]. Available: https://www.educative.io/blog/javascript-versions-history. [Accessed 22 April 2021].

[11] Microsoft, "Characteristics of Modern Web Applications," Microsoft, 12 Janauary 2020. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/modern-web-applications-characteristics. [Accessed 22 April 2021].

[12] Google, "Web Fundamentals," Google, 20 May 2021. [Online]. Available: https://developers.google.com/web/fundamentals. [Accessed 2 June 2021].

[13] Zartis, "3 Undeniable Characteristics of Modern Web Apps," Zartis, 23 September 2020. [Online]. Available: https://www.zartis.com/3-undeniable-characteristics-of-modern-web-apps/. [Accessed 23 April 2021].

[14] A. Senecki, "Web development stacks – what stacks (should) we use in 2021?," The Software House, 9 July 2020. [Online]. Available: https://tsh.io/blog/web-development-stacks/. [Accessed 23 April 2021].

[15] Vue, "Comparison with Other Frameworks," Vue, [Online]. Available: https://vuejs.org/v2/guide/comparison.html. [Accessed 30 April 2021].

[16] G. Sharabok, "Why Deep Learning Uses GPUs?," Towards Data Science, 26 July 2020. [Online]. Available: https://towardsdatascience.com/why-deep-learning-uses-gpus-c61b399e93a0. [Accessed 23 April 2021].

[17] Microsoft, "Choose Between Traditional Web Apps and Single Page Apps (SPAs)," Microsoft, 12 January 2020. [Online]. Available: https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/choose-between-traditional-web-and-single-page-apps. [Accessed 23 April 2021].

[18] Mozilla, "Promise," 26 May 2021. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Promise. [Accessed 2 June 2021].

[19] Mozilla, "AudioContext," 29 May 2021. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/AudioContext. [Accessed 2 June 2021].

[20] React (Facebook), "Create a New React App," 2021. [Online]. Available: https://reactjs.org/docs/create-a-new-react-app.html. [Accessed 3 March 2021].

[21] React (Facebook), "Thinking in React," 2021. [Online]. Available: https://reactjs.org/docs/thinking-in-react.html. [Accessed 23 April 2021].

[22] N. Aharoni, "How to Choose the Right NPM Package for Your Project," Better Programming, 19 August 2020. [Online]. Available: https://betterprogramming.pub/how-to-choose-the-right-npm-package-for-your-project-c3d1cc25285e. [Accessed 28 April 2021].

[23] W3C, "Accessibility," 2018. [Online]. Available: https://www.w3.org/standards/webdesign/accessibility. [Accessed 23 April 2021].

[24] React (Facebook), "Testing Overview," 2021. [Online]. Available: https://reactjs.org/docs/testing.html. [Accessed 2 June 2021].

[25] React (Facebook), "Deployment," 2021. [Online]. Available: https://create-react-app.dev/docs/deployment/. [Accessed 2 June 2021].

[26] GitHub, "Workflow syntax for GitHub Actions," 2021. [Online]. Available: https://docs.github.com/en/actions/reference/workflow-syntax-for-github-actions. [Accessed 2 June 2021].

[27] S. Bose, "End To End Testing: A Detailed Guide," BrowserStack, 19 May 2021. [Online]. Available: https://www.browserstack.com/guide/end-to-end-testing. [Accessed 02 June 2021].

[28] Wikipedia, "Usage share of web browsers," Wikipedia, 2 March 2021. [Online]. Available: https://en.wikipedia.org/wiki/Usage_share_of_web_browsers. [Accessed 23 April 2021].

[29] Microsoft, "GitHub," 6 December 2018. [Online]. Available: https://github.com/MicrosoftEdge/MSEdge/blob/7d69268e85e198cee1c2b452d888ac5b9e5995ca/README.md. [Accessed 29 April 2021].

[30] ARS, "Google going its own way, forking WebKit rendering engine," ARS Technica, 4 March 2013. [Online]. Available: https://arstechnica.com/information-technology/2013/04/google-going-its-own-way-forking-webkit-rendering-engine/. [Accessed 29 April 2021].

[31] Royal Philharmonic Orchestra, "A new era for orchestral music," Royal Philharmonic Orchestra, 2018.

[32] S. Felix, "Are Piano and Guitar Chords the Same?," Rusty Guitar, 28 January 2019. [Online]. Available: https://www.rustyguitar.com/are-piano-and-guitar-chords-the-same/. [Accessed 2 June 2021].

[33] J. Sandercoe, "The Dreaded F Chord," JustinGuitar, 2021. [Online]. Available: https://www.justinguitar.com/guitar-lessons/the-dreaded-f-chord-bc-161. [Accessed 3 June 2021].