Tab My Pitch Up

Thomas Morley

May 6, 2022

Abstract

Musical notation is an essential tool for composers and musicians. With it, music can survive for centuries, notation providing the blueprint for symphonies to be reawakened, new styles to be learned and compositions to be analysed. Notation is the data structure we use to archive messages written in our most evocative language, and the best device for this remains to be the human brain. Despite the ongoing automation gold rush, transcribing music is an unsolved challenge, in some part because many sub-tasks exist within this challenge, each of which presenting a unique problem, and each of which associated with a diverse set of solutions that will be explicated in this paper. Also contained in the paper is the developmental journey of *Tab My Pitch Up*, a new software solution designed to produce guitar tablature from guitar performances.

Git: https://gitlab.doc.gold.ac.uk/tmorl002/finalproject

Contents

1 Keywords

- Automatic Music Transcription (AMT)
- Note Onset Detection (NOD)
- Neural Networks (NN)
- Pitch detection

2 Introduction

Music notation is relied upon for recording, storing and sharing music. For the music industry, the appearance of an automatic notation system would be an immense time saver. Automatic music transcription (AMT) is the field of study concerned with such things. The focus of this project is to explore AMT and subsequently produce an application - *Tab My Pitch Up* - that applies its methods in the context of producing guitar tablature from input audio files of guitar performances.

2.1 Motivation

So far, no approach to AMT has exceeded the competency of an expert musician, a fact that is unchanged by the fruits of this project; however, this paper will provide evidence that AMT methods can be applied in the commercial space in such a way that benefits the field of music. This evidence comes in the form of dialogues with real musicians, both professional and amateur, who were exposed to the system several times throughout the process, in addition to performance tests that yield cutting-edge results.

Further in this document, a brief argument will be put forward in favour of guitar tablature as the preferred form of notation for the contemporary guitarist. Needless to say, the guitar is one of the most popular instruments adopted by performers and creatives, and therefore we hypothesise high demand for a product that can alleviate the workload of working musicians and practising amateurs alike.

The ultimate intention is that this technology can compliment the workflow, education and communication of the artists it serves: workflow, in the sense that the process of recording music in notation becomes completely automated; education, in that the constituent notes of a performance can be easily revealed; and communication, in that digital tabs can be easily produced and then shared peer-to-peer.

2.2 Aims and scope

The primary focus of this project has been the curation and implementation of methods in the AMT literature, resulting in "Tab My Pitch Up", which has been designed to "listen" to guitar music and then produce the appropriate

music notation - guitar tablature, in our case. This paper will explore the AMT field, discuss its methods, and summarise its progress. Following this, we aim to provide satisfactory evidence that the Tab My Pitch Up platform combines both novel code and established frameworks in an effective way - "effective", in this context, referring to the project's demonstration of a potential commercial application for AMT, and a potential in-demand product for the music industry.

Some of the bigger challenges in AMT elude the scope of this project, such as *polyphony*; the presence of multiple notes played simultaneously. This means recordings may not include chords or double notes, only melodies comprising of a sequence of notes played individually - or, in short, *monophonic* recordings. The ability to separate and identify various instruments in an ensemble performance is also not within the scope of this project, nor is the ability to classify an instrument, for example as a "guitar" or "bass" - in fact, it is assumed that the inputs will be exclusively monophonic guitar recordings.

The concepts that are within the scope of this project include pitch detection and note onset detection (NOD). The former involves finding *f(0)* - or, the *fundamental frequency* - of a sound wave. With this information, we can determine *which notes are being played* - it therefore follows that NOD is concerned with *when* these notes are played and *in what order*. Thus, by combining the two concepts, we know enough of the musical content of the audio file to arrange the correct notes in the correct positions on the output *tab* (short for "tablature") - at this stage, it must be noted that the concept of guitar tablature is explained in the Background section of this document.

In order for non-technical users to navigate the functions described in the previous paragraph, the program must be equipped with an easy-to-use GUI system and helpful features such as saving/loading tablature files. Later on in this document, the architecture of the system, including these tertiary features, will be explained in depth. In order to bring about these features in the most effective way, users were consulted regularly during an iterative design process whereby the minimum viable product (MVP) was optimised in terms of its performance on a test data set and of its usability as measured by stakeholders; this way, the MVP is assessed as both a data processing model and as a software product.

3 Background

3.1 Artificial neural networks

An artificial neural network (NN) is a mathematical model whose parameters are adjusted automatically by machine learning. The functional design is intended to mimic that of an actual human brain [1]. The goal of NNs is to approximate mathematical functions that connect a subject's properties.

3.1.1 The Perceptron

A single neuron in a NN can be represented by a *perceptron*: a mathematical object consisting of a series of inputs that produce an output via some internal function. The original algorithm for a perceptron was put forward by Rosenblatt [2] and has since been adopted by the AI community as a fundamental component to contemporary machine learning.

3.1.2 Network architecture

Perceptrons are arranged in an NN either in parallel or in layers. We can intuit what are "layers" in the diagram below [fig.1]; these are the *collumns* of neurons. We also see the leftmost layer represented by a slightly different shape: these are the input *attributes* - the properties (as a numerical representation) of the object under examination. In a *dense* layer, every attribute is received by every neuron. Subsequently, every neuronal output from the current layer is received as input by the next set of neurons, in the same way as though they were attributes. Therein lies the difference between neurons of different layers - neurons in parallel share the same inputs, whereas neurons in subsequent layers process the output of their predecessors [3].

3.1.3 Activation and the Sigmoid

An activation function is the internal function of a perceptron [1]. Such a function is the Sigmoid function:

$$S(x) = \frac{1}{1+e^{-x}}$$

where *S(.)* represents the sigmoid function and x represents the weighted summation of the inputs of the neuron in question. The Sigmoid function maps its input to a value between 0 and 1.

3.1.4 Learning

The way that the NN learns from processing multiple data points is a two-step process; first, the *forward propagation*, during which data are fed through the network (left to right in [fig.1]) and the model makes its predictions; second, the *backward propagation*, whereby internal system parameters (weights) are updated depending on the success of the prediction, as measured by a *loss function* [4].

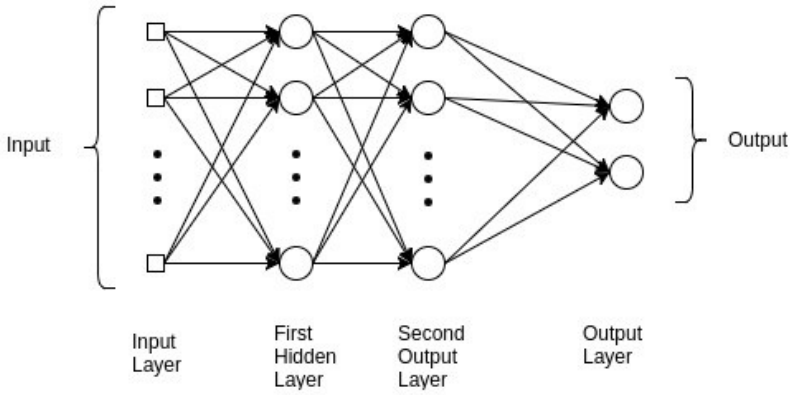

Figure 1: Multilayer perceptron architecture
Source: www.analyticsvidhya.com

3.1.5 Loss function and binary cross-entropy

For our purposes, it is only necessary to define the *binary cross-entropy* loss function. Cross-entropy is a measure of the absolute difference between the label that is predicted by the NN, and the *ground-truth annotation* - the correct answer. *Binary* cross-entropy is applied when these labels/annotations only take values of 0 or 1 [5]. The formula for binary cross-entropy is as follows:

$$L = -[y.log(p) + (1 - y).log(1 - p)]$$

where y is the ground-truth annotation and p is the NN's prediction.

3.1.6 Optimizers

Optimizers are responsible for updating the weights depending on the loss function. Generally, the optimizer's task is to minimise this loss function, and thus encourage more accurate predictions from the system [5]. The *learning rate* determines how drastically the weights are adjusted in accordance with minimising the loss function.

3.1.7 Convolution NNs

Convolution NNs (CNN) are a NN paradigm. The components of a CNN were introduced by K. Fukushima [6] as a means of achieving pattern recognition that is *translation invariant.*Via successive layers and downsampling, the CNN can begin to aggregate these learned patterns [7] into more complicated shapes.

3.2 Automatic music transcription

AMT is the process of converting an audio signal into abstract notation that represents its musical properties, via some computational method [8]. Since the year 2000, the International Symposium for Music Information Retrieval[1] has generated vast series of papers and discussions on the topic, in addition to other topics included in music information retrieval - this past year saw submissions such as attempting to translate image classification techniques to musical key estimation [9], to automatically identifying the perfect song to match the emotional tone of a given passage from a novel [10].

The key tasks of AMT are to retrieve frequency information from audio files, and then to translate it into human readable abstractions [8], in our case, guitar tablature. Both tasks carry their own sets of challenges. In the next section, methods for retrieving frequency information from audio signals will be discussed; namely, Fourier transform and Constant-Q transform (as these technologies appear consistently throughout the literature). Another concept that is vital to this project, is *note onset* - the point in time at which a note makes its appearance.

3.2.1 Fundamental frequency: fourier transform and constant-Q

Fourier transform and constant-Q transform are methods of estimating f(0). The role of this is to provide a reference for the musical content of the audio file; for example, the note C is attributed to the frequency of 32.7Hz[2]. However, C can also be attributed to the frequency of 65.41Hz. This is because doubling the frequency of a given signal lifts the pitch into the next octave, meaning as the frequency of the signal increases, the 12 notes are repeated. As such, notes are commonly denoted to indicate their octave register [11], ie C0, C1, C2, etc. Therefore, once the frequency is detected, musical intention can be deduced and described by notes and registers. Real world data is noisy, however; this means multiple frequencies will overlap each other simultaneously in an audio signal, resulting in waveforms that look less like consistent sine waves, and more like rollercoasters, making the fundamental frequency less clear.

Fourier transform takes signals expressed in the time domain and transforms them into representations in the frequency domain. That is, given a waveform of arbitrary magnitudes over time, fourier transform allows us to represent this as magnitudes over the frequency spectrum. In this way, we can see which frequencies occupy the original signal based on their respective magnitudes in the transformation [12]. One of the limitations of Fourier transform is that time information is lost; in order to transcribe melodies it is essential that we understand the sequence that the constituent notes appeared in the recording - one possible way to tackle this would be to take Fourier transformations at regular time intervals along the input signal.

[1] https://www.ismir.net/conferences/

[2] https://pages.mtu.edu/ suits/notefreqs.html

The Constant-Q transform works similarly, but provides output against the log frequency, which better suits the exponential nature of octaves appearing at double the frequency value of the previous register [13], and more similarly mimics the behaviour of the human ear [14].

3.2.2 Note onset

As defined earlier, note onset detection (NOD) is concerned with deducing when a given note makes its appearance in the composition. This is less complex for the task of monophonic transcriptions, though it still remains challenging to have the system discern accurately between background frequencies and those emitted by the instrument, or to notice whether the same note has been played twice consecutively. The solution proposed by J. P. Bello, et al. in [15] involves downsampling the input signal in such a way that the transient [fig.2] becomes more prominent in the downsampled signal.

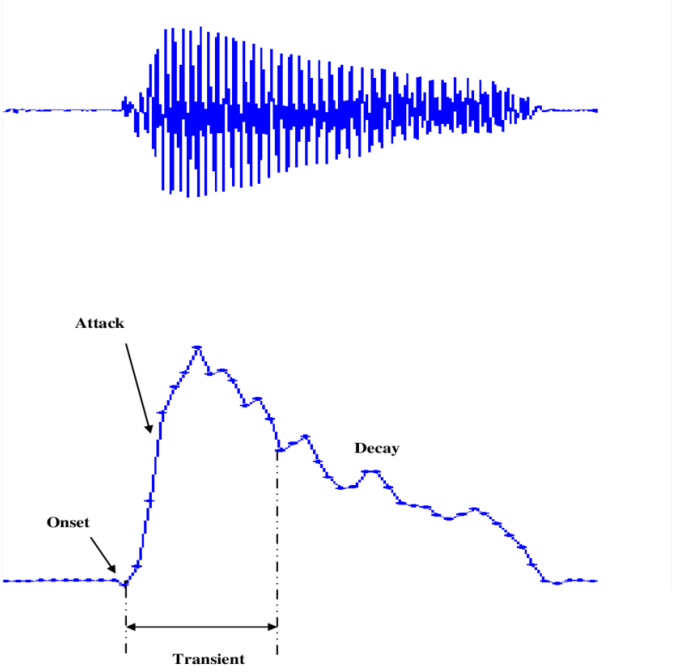

Figure 2: A note's transient
Source: fig.2 from [16]

This may include preprocessing the input in order to accentuate indicative features, ie sudden changes in frequency or amplitude. The final stage is then to identify peaks in the downsampled signal. Building upon this solution, N Degara, et al. [17] were able to incorporate knowledge of rhythmic structures

to successfully constrain predictions on the basis that temporal arrangements of music are not irregular nor random, suggesting that it can be advantageous to implement biases into our analytical solutions that contain information about music theory.

3.2.3 Sheet music vs guitar tablature

Many AMT systems attempt to analyse signals that contain performances by multiple instruments. This involves analysis of properties with structurally different abstract descriptions; timbre (characteristics distinguishing musical instruments), pitch (f(0), for intents and purposes), note onset/offset (the temporal domain of the note), rhythm (temporal structure of the piece) and so on [8]. Naturally, analyses have typically been sought to be represented by sheet music as this is the most general and commonly used notation device.

To the uninformed reader, sheet music would appear to be a combination of nonsensical, albeit quite pretty, symbols arranged along a ladder of horizontal lines. This notation is concerned with representing musical information; this means notes present, the correct sequence of these notes, rhythm, and other properties [18]. Traditionally, sheet music is the way in which melodies and compositions are archived for future reference.

This project aims to avoid the use of sheet music notation, substituting it instead for tablature notation. The relevance of sheet music notation for the modern guitarist appears to be dwindling - in fact, it has been reported that some of the most venerated contemporary guitarists, including Jimi Hendrix and Eric Clapton, were unable to read sheet music at all [19]. It has been argued that sheet music notation is better suited to instruments like the piano, wherein the functional design of the instrument consists of an arrangement of keys; for the guitar, a stringed instrument, tablature notation is better optimised [20], and appears to be both more popular and more accessible for modern guitar practitioners.

Shown below [fig.3], guitar tablature consists of six lines, each representing a string on the guitar. The numbers on these lines represent the fret to be played on that string - one needn't even know which musical notes constitute the melodies they are playing.

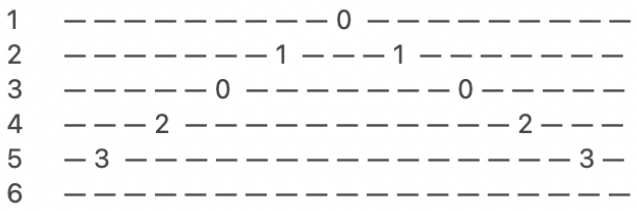

Figure 3: Arpeggiated C chord expressed in tablature

3.2.4 AMT for guitar

Investigations into guitar specific AMT mitigate some of the complications present in more generalised AMT solutions, such as transcribing multiple audio sources. What remains complicated in these investigations is polyphonic recordings; that is, chords or chord progressions played on the guitar. We have already said that this project will tackle monophonic recordings only, however the approaches and methods of the following investigations still bear relevance on the task at hand.

One such investigation attempts to deduce monophonic guitar parts from popular music [21]. The scope is limited to monophony for the same reasons as with this project. In [21], Fourier transform techniques such as Cepstrum are used to identify the pitch produced by the lead guitar amongst other instruments; this demands a more complex system, which in this case applies multiple mathematical principles and techniques for separating audio sources. The most effective technique applied in this paper consistently achieves above 90% in accuracy, and is the Harmonic Product Spectrum (HPS) technique - in short, Fourier transformations are taken from windowed portions of the input waveform and compared to find the most common peak in the resultant graphs. The spectral value of the most common peak is taken to be the estimation for f(0). It is mentioned in this paper that certain songs have been rerecorded - Sweet Child O' Mine (Guns N' Roses) has had its lead re-recorded without overdrive, instead with a clean guitar set-up, suggesting that this model struggles to generalise across various guitar tones. This would be expected in a system where one of the tasks is identifying which instrument is the guitar in the first place, as effects such as overdrive will significantly effect timbre.

Another investigation [13] applies CNNs to produce tablature from the GuitarSet[3] dataset, whereby predictions were made for each individual string's corresponding line in the tab. The average accuracy across each string was 84.23%. The data used to identify notes is acquired via Constant-Q transform. This result is roughly consistent with the methods used in [21], apart from the HPS technique which outperforms this result by a significant margin.

Comparisons between [13] and [21] should be taken with a pinch of salt for two reasons; [13] attempts to identify constituents of polyphonic guitar recordings, whilst [21] attempts to isolate and represent monophonic guitar participants of ensemble pieces; and, further still, both were evaluated on different datasets. The point is that guitar AMT (in fact, AMT generally, as we have already said) sets a very low bar for success, and while these results are impressive technically, they are far from being reliable in any serious practical capacity, and would not challenge the expertise of a human professional assigned to the same task.

[3] https://zenodo.org/record/1422265.YcIHgC2l1QK

3.2.5 The guitar fretboard

A problem for guitar transcriptions is the multiplicity of notes from the same register on the guitar fretboard. For example, a C note with a frequency of 130.81Hz appears on the 8th fret of the 6th string *and* on the 3rd fret of the 5th string. Given a series of audio data from guitars with varying set-ups, it would be impossible to tell from which exact point on the fretboard the detected note was coming from. It may be possible to achieve a successful classification algorithm for these duplicate frequencies if the same guitar was used with the same set-up every time, but in the general case, the various tones produced by different guitars with different set-ups would obfuscate any distinguishing data that might indicate which string the note was played on at the time of recording.

The question, then, is how to optimise the output tabulations in terms of convenience; in what way can we construct the melody on the fretboard so that the output is always reasonably navigable for the human hand.

Let's take, for example, Happy Birthday - G G A G C B. If we were to start this melody at, say, G4, the notes and registers would be: G4 G4 A4 G4 C5 B4. For the sake of demonstrating complexity, lets assume each of these notes appear exactly twice on the fretboard (in reality, notes usually appear more than twice). That gives 2^6 different potential tabs that describe the exact same melody. If we add a seventh note: 2^7, and so on. Potential tabulations increase exponentially as the length of the melody increases.

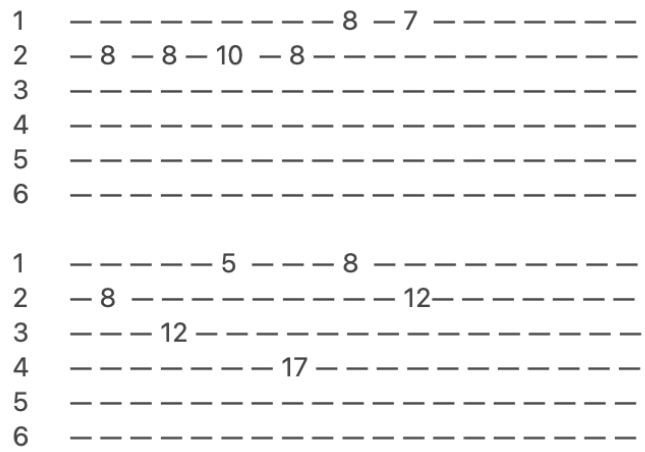

Figure 4: Two examples of 'Happy Birthday' played in the same octave register

Both of the above [fig.4] tabulations represent Happy Birthday as played from G4, but it is clear that the difficulty in playing these tabs varies wildly.

In approaching this issue from the angle of transcribing chords - that is, where the detected notes would all be played simultaneously - D. Tio [13] develops 'finger economy' values for potential notes. This includes finding all possible

solutions for a given chord, finding the distance (in terms of frets) each note is from the root note (the note played on the lowest string), and selecting the solution that minimises these distances.

3.3 CREPE

CREPE [22] is a python module implemented using Keras[4] that uses deep learning to estimate the fundamental frequency of monophonic audio signals. CREPE forms the cornerstone of this project, acting as the vehicle for input signals to become pitch-trackings which are then described by our output tabs.

3.3.1 Pitch detection problem

The task of CREPE is to determine the f(0) of monophonic audio signals - this is referred to as *pitch tracking* or *pitch estimation*. F(0) is defined as the inverse of a signal's period - that is, in the case of perfectly periodic signals [23] - the challenge with real-life audio recordings is finding f(0) where the signal is not perfectly periodic. It is also worth noting that pitch is technically defined as a subjective experience [24], though it is so closely correlated with f(0) that the two terms are often used interchangeably in the field of AMT.

We have previously discussed methods for deducing pitch and made clear the importance of achieving accurate predictions for this project. CREPE takes an approach of applying a deep CNN that operates directly on the time-domain signal.

3.3.2 Implementation of the CREPE system

Below [fig.5] shows the CREPE architecture.

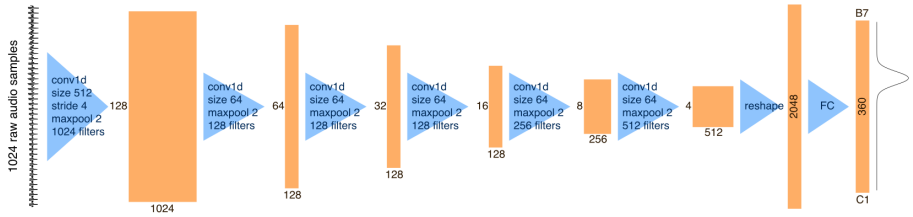

Figure 5: Crepe architecture
Source: fig.1 from [22]

Shown in the figure, the system takes a 1024-sample excerpt from the time-domain signal at once and passes the data through six convolutional layers, ultimately resulting in a latent 2048-dimensional representation. This culminates in a densely connected output layer that uses sigmoid activations and

[4]https://keras.io

outputs a 360-dimensional output vector, from which the pitch estimation is calculated deterministically.

At this stage, it is appropriate to define cents. The cent is a unit that represents musical intervals relative to a reference pitch, f_{ref}, given in Hz [22], where the cent, c, is a function of frequency, f, in Hz:

$$c(f) = 1200.log_2.\frac{f}{f_{\text{ref}}}$$

Each of the 360 nodes in the final layer of the above architecture represents a frequency bin covering 20 cents, where 100 cents is equidistant to 1 semitone (distance from one musical note to its immediate neighbour). The neural network outputs a Gaussian curve which highlights the frequency bins wherein lay the estimated fundamental frequency of the input. The values of these nodes have been selected so that the they can cover six octaves (registers 1 through 6). This covers the notes from C1 to B6 with frequencies of 32.70 Hz and 1975.5 Hz, respectively. The pitch prediction, c', is the weighted average of the pitches, c, associated with the approximated Gaussian curve output, y.

$$c' = \frac{\sum_{i=1}^{360} y_{\text{i}} c_{\text{i}}}{\sum_{i=1}^{360} y_{\text{i}}}$$

As such, predictions for frequency in Hz, f', can be acquired:

$$f' = f_{\text{ref}}.2^{c'/1200}$$

The targets used for training are 360-dimensional vectors whereby the correct frequency bin is given a magnitude of 1. The network is trained to minimise the binary cross-entropy between these targets and the network's predictions.

The loss function is optimised using the ADAM optimiser [25] with the learning rate set to 0.0002. The model has been trained for 32 epochs consisting of 500 batches, each containing 32 samples.

3.3.3 Other pitch detectors

Other means of achieving accurate pitch tracking were considered; for example, the pYIN [26] estimator uses the same auto-correlation function applied in [23], but garners improved results by producing pitch candidates appended with probabilities in order to reduce the loss of useful information. Further, the work in this paper employs a hidden Markov model to calculate smooth pitch tracking with improved precision and recall. In [22], performance of the CREPE system is compared directly [fig.6] with pYIN when evaluated on audio samples synthesised from the RWC Music Database and a re-synthesised subset of the MedleyDB set [27, 28]. CREPE consistently produces higher accuracies.

CREPE has also demonstrated improved performance over SWIPE [29], another popular pitch estimator that makes estimations by matching sawtooth waveform spectra with particular f(0) values, with the spectra of input signals.

Other machine learning techniques have been applied to pitch detection, such as SPICE [30] which uses self-supervised training in order to circumvent

	CREPE	pYIN	SWIPE
RWC synth	0.999±0.002	0.990±0.006	0.963±0.023
MDB stem-synth	0.967±0.097	0.919±129	0.925±0.116

Figure 6: Average raw pitch accuracies for CREPE, pYIN and SWIPE, from **table 1** in [22]

the issue of acquiring targets with completely accurate ground truth annotations for f(0) [8]; however, SPICE was unable to outperform CREPE in a significant or consistent way. In addition, CREPE has an abundance of 3rd party applications and educational resources available online that have aided in production of this project, which cannot be said to the same degree with SPICE.

3.4 Similar products to Tab My Pitch Up

There appear to be very few - if any - automatic transcription services for tablature. Nonetheless, the following applications have had to solve similar or the same problems as with this project, and serve as examples of AMT applied in the commercial space. Perhaps the key takeaway from this is how sparse the market is for transcription software specifically. The assumption made by this project is that this is an issue of supply rather than demand - this assumption is re-enforced by the richness of the research literature.

3.4.1 Melodyne

Melodyne[5] does not perform transcriptions, though it performs analysis of audio similar to what we hope to achieve. Input audio signals are re-imagined according to their musical properties to make information more relevant and intuitive for audio engineers.

3.4.2 AudioScore

AudioScore[6] provides transcriptions in the form of sheet music, and is designed to separate instruments - guitar and bass parts that overlap would be separated and transcribed separately, for instance. Reviews for this platform[7] have been poor, almost exclusively on the basis of poor transcription accuracy.

3.4.3 Transcribe!

Transcribe![8] operates as a general assistant for transcription - while the system does attempt to deduce rhythm, key, and notes, it does not produce sheet music.

[5] https://www.celemony.com/en/melodyne/

[6] https://www.avid.com/products/audioscore-ultimate/

[7] https://www.amazon.com/Sibelius-Audioscore-Ultimate-Recognition-Transcription/product-reviews/B003A10LQO

[8] https://www.seventhstring.com/xscribe/overview.html

It does allow users to manipulate and analyse audio tracks in ways that assist in transcription.

4 Methods

4.1 Scope

We have already loosely defined the scope; this section acts as a formal statement of the system's functionality.

Some of the requirements can be measured in a quantitative way; for instance, how accurate the system is will be measured using *precision* and *recall*. What is classed as "some degree of accuracy" [fig.7] has no numerical definition here; we simply aim to maximise these measures in the pursuit of it.

Criteria like "easily navigable" [fig.8] will be assessed by stakeholders during development.

4.1.1 Functional requirements

Requirement
The system can detect f(0) at regular time intervals throughout an input audio clip
The system can distinguish between the absence and presence of musical notes
The system accepts mono and stereo audio
The system can, with some degree of accuracy, produce guitar tablature that reflects a given input
The system can save tabs to storage in a file format that can be read by the system so it may reproduce the original tab on-screen without the need for the for the initial input audio clip
The user can save files in a location of their choosing
The system can, given a file input of the appropriate type, produce the same tab that was previously displayed from an audio input and subsequently saved to said file
The system can be launched from an executable file
The system can be safely closed

Figure 7: Functional requirements of the Tab My Pitch Up system

4.1.2 Non-functional requirements

Requirement
The system will accept wave (.wav) audio input only (because this is the exclusive file format accepted by CREPE)
Loading previously saved tabs must be faster than analysing the audio clip again
The system must be easily navigable by non-technical users

Figure 8: Non-Functional requirements of the Tab My Pitch Up system

4.2 System Design

4.2.1 Use Cases

Use cases [fig.9] for the system are relatively limited. The user may: (1) load a new audio file; (2) load a previously saved tab file; (3) close the system, or finally: (4) save current tab as a tab file.

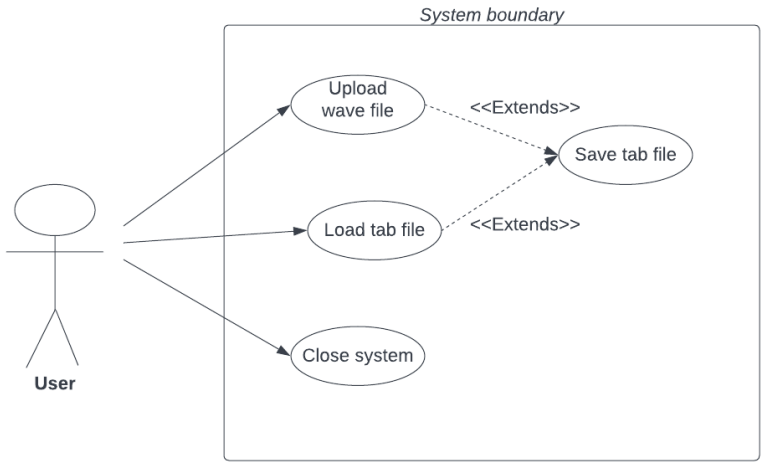

Figure 9: Use case diagram for Tab My Pitch Up

It may be counter-intuitive that 'save tab file' extends 'load tab file', but it is functionally possible within the system, and therefore represented in [fig.9]. Further, there may be some cases where the user would like to duplicate their files via the program (though it is no more useful or easy than a simple copy/paste of the file in the user's native file browser).

4.2.2 Class overview

In order to implement the process of uploading audio and producing a tab, two key stages must be performed by the system: first, input audio must be analysed in a way that extracts all information that is indicative of the musical intention; second, this information must be organised in a way that can be drawn as a tab within the GUI implementation of the system.

The source code for Tab My Pitch Up is separated into Python modules, each addressing separate concerns. As the platform develops, it will be necessary to include more algorithms that address different facets of the problem, or remove algorithms as they become superseded or redundant; therefore, the process of adding new algorithms in conjunction with current algorithms must be performed with minimal chance for interference or disruption of the preexisting algorithms. Hence, Tab My Pitch Up facilitates a class ecosystem whereby algorithms analysing the input exist completely independent of one another, and algorithms constructing the output can receive objects from the analysing algorithms as parameters. The diagram below [fig.10] illustrates the proposed architecture.

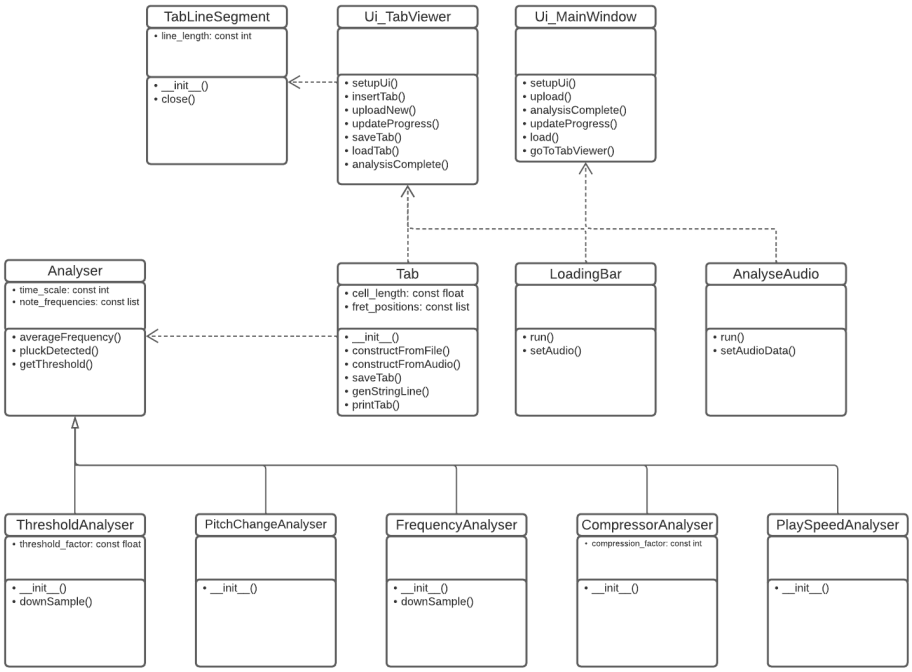

Figure 10: Class diagram for Tab My Pitch Up

4.2.3 Analyser

The analyser class is a super class that lends its class variables and functions to its five child classes illustrated above [fig.10]. This class also creates objects via a constructor function [fig.11] that combines outputs from its child classes, which each produce representations of the input that reflect relevant properties. The analyser class combines these representations and passes the result to the Tab class as a parameter to its constructor function. Given an audio input which is expressed as a one dimensional array of values, $\boldsymbol{x}$, the final representation, $f(\boldsymbol{x})$ is given by:

$$f(\mathbf{x}) = P(T(C(\mathbf{x})) * F(C(\mathbf{x}))) * T(C(\mathbf{x})) * F(C(\mathbf{x}))$$

where $P(.)$ represents the PitchChangeAnalyser output, $C(.)$ represents the CompressorAnalyser output, $T(.)$ represents the ThresholdAnalyser output, $F(.)$ represents the FrequencyAnalyser output, and $*$ represents a point-wise multiplication.

```
#Constructor function - produces representations of the data used in crafting tabs
def __init__(self, audioFile):


    CompressorAnalyser(audioFile)   #Apply dynamic compression to the audio

    self.threshold = ThresholdAnalyser(audioFile)   #Eliminate data below threshold from consideration

    self.frequency = FrequencyAnalyser(audioFile)   #Analyse frequencies

    merge = []      #Vector holding merged output of analyser outputs produced above

    #Change cell_length if play speed is fast - adjust length of cells proportionately
    idealCellLength = 1 / PlaySpeedAnalyser(audioFile, self.threshold).notesPerSec
    self.cell_length = idealCellLength if idealCellLength < self.cell_length else self.cell_length

    #Merge representations, and populate merge array
    for i in range(0, len(self.frequency.outputVector)):
        merge.append(self.threshold.outputVector[i] * self.frequency.outputVector[i])

    #Downsample merge array
    merge = self.downSample(merge, int(self.cell_length * self.time_scale))

    #Generate pitch change representation & merge with other representations
    self.pitchChanges = PitchChangeAnalyser(merge)
    for i, element in enumerate(merge):
        merge[i] = element * self.pitchChanges.outputVector[i]

    #Set analysis variable to contain final representation of audio
    self.analysis = merge
```

Figure 11: Constructor function for the Analyser class

ThresholdAnalyser

ThresholdAnalyser [fig.14] and FrequencyAnalyser [fig.16] both downsample the input, meaning that the length of their output arrays are only a fraction of the original length of the input array. Given a threshold magnitude value, ThresholdAnalyser will take a set of consecutive values from the input array, and represent this 'chunk' in its output by a 1 - if the average magnitude (audio volume) is above the threshold value - or by a 0 - if the average magnitude is below the threshold. The result is an array whereby chunks of 0s represent the absence of music and chunks of 1s represent notes intended by the performer. [fig.13] visualises the output from ThresholdAnalyser after processing [fig.12]. The threshold is calculated as a constant fraction, 0.25, of the highest magnitude present in the input data.

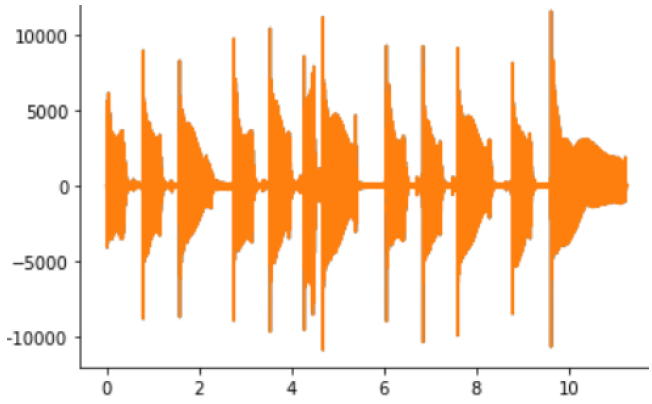

Figure 12: Original audio wave, given by magnitude over time

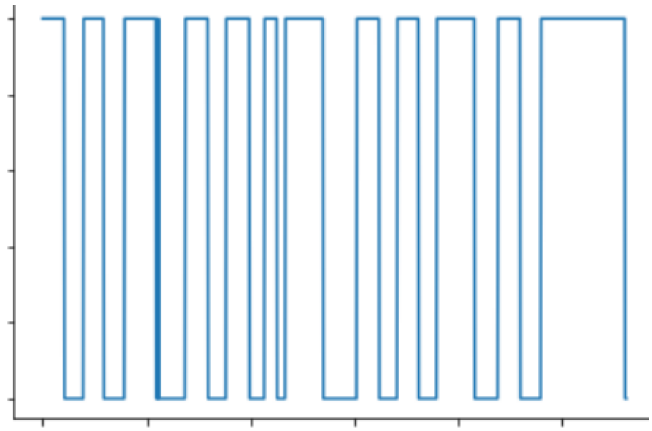

Figure 13: Graph representation of Threshold analysis. x = time, y = magnitudes above threshold (represented by a 1)

```
def __init__(self, audioFile):

    self.audioFile = audioFile
    maximumVolume = self.getMax()
    self.threshold = maximumVolume * self.threshold_factor
    print(maximumVolume)
    print(maximumVolume * self.threshold_factor)

    #Create output representation
    outputVector = np.zeros(int(self.audioFile.audio.shape[0]* self.time_scale / self.audioFile.sampleRate), dtype=(int))
    for i in range(0, len(outputVector)):
        outputVector[i] = self.pluckDetected( i * int(self.audioFile.sampleRate/self.time_scale), int(self.audioFile.sampleRate/self.time_scale) )

    self.outputVector = outputVector
```

Figure 14: Constructor function for the ThresholdAnalyser class

FrequencyAnalyser

FrequencyAnalyser produces an output array who's values represent the average frequency of the corresponding chunk from the input audio data. The frequencies are determined by CREPE. When the result is multiplied with the output from ThresholdAnalyser, the previously mentioned chunks of 1s are replaced with the estimated frequency. So now we can estimate *when* the notes are being played - as the note's presence is represented in ThresholdAnalyser by a series of 1s - and we can also estimate *which* notes are being played, by monitoring the frequency.

CREPE outputs the predicted f(0) of the audio clip for every 0.001 second interval - this downsamples the signal from 44.1kHz (default for wave files) to 1kHz. These sample rates describe the number of data points generated per second of audio. The FrequencyAnalyser class downsamples this data further (see [fig.15]) to 10Hz by means of average pooling. Furthermore, frequency values are now rounded to the nearest semitone. The total reduction in data resolution from 44.1KHz to 10Hz is drastic, but makes the output more useful because we do not expect users to play guitar notes at the rate of thousands per second. Furthermore, if we employed analytics at that level, the chances that false positives might appear in the output increases, and any tablature with the precision of 1000 units per second would be almost completely unreadable to humans because of its length.

```
def averageFrequency(self, frequency, index, chunkLength):   #Get average frequency over a given timefame

    #Get average frequency over given timeframe
    sumOfFreqs = 0.0
    count = 0

    for i in range (index, index + chunkLength):
        if frequency[i] > 76.00 and frequency[i] < 1065.00: #Filter out frequencies that are too low/high to be guitar notes
            sumOfFreqs += frequency[i]
            count += 1
    if count < 1:
        averageFreq = 0
    else:
        averageFreq = sumOfFreqs / float(count)

    #Round result to the nearest frequency attributed to a musical note
    roundedAverage = 0.0
    for i, freq in enumerate(self.note_frequencies):
        if abs(averageFreq - freq) < abs(averageFreq - roundedAverage):
            roundedAverage = freq

    return roundedAverage
```

Figure 15: Class method that returns a downsampled version of the input by max pooling and rounding segments of the input

```
#Frequency analyser
class FrequencyAnalyser(Analyser):

    def __init__(self, audioFile):
        self.audioFile = audioFile

        #Create representation
        outputVector = np.zeros(int(self.audioFile.audio.shape[0]* self.time_scale / self.audioFile.sampleRate), dtype=(float))
        for i in range(0, len(outputVector)):
            outputVector[i] = self.averageFrequency(self.audioFile.frequency, i * int((1/self.audioFile.time[1]) / self.time_scale), int((1/self.audioFile.time[1]) / self.time_scale))

        self.outputVector = outputVector
```

Figure 16: Constructor function for the FrequencyAnalyser class

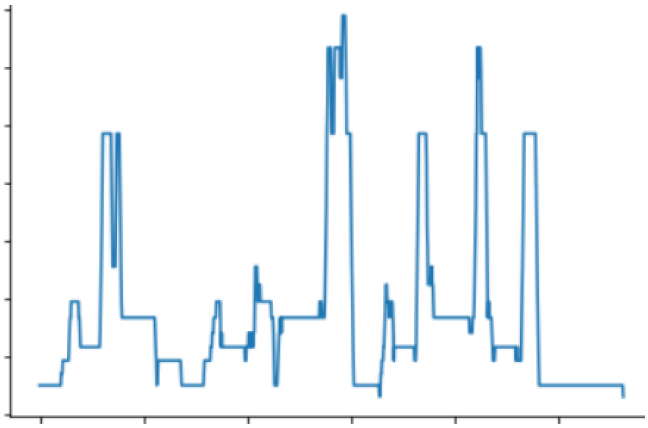

Figure 17: Graph representation of FrequencyAnalyser taking [fig.12] as input. x = time, y = frequency

PitchChangeAnalyser

It may sometimes - in fact, due to the successive downsampling of the input audio data, quite often - be the case that there is not a sufficiently long period of quiet between two notes. Considering the methods described above in pertinence to ThresholdAnalyser and FrequencyAnalyser, this would mean that the latter note would be ignored because it would not be distinguishable from the former. In practical terms, the two notes would homogenise into a single chunk. To address this, PitchChangeAnalyser [fig.19] looks for changes in frequency from one whole semitone to another, manually inserting a 0 value *immediately before* the appearance of the new note, artificially producing a new chunk. It also mulitplies all values in a chunk by 0 apart from the first. This means that the resultant representation [fig.20] is a sparse array where the predicted notes - represented as a frequency - appear only once, at the predicted onset.

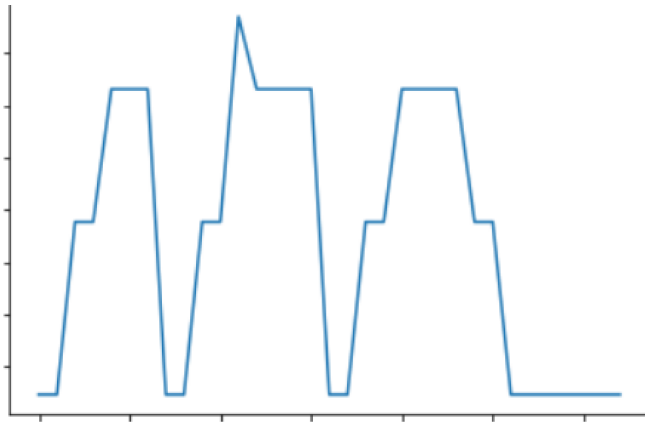

Figure 18: Graph representation of product of FrequencyAnalyser and ThresholdAnalyser produced from [fig.12]. x = time, y = frequency

```
def __init__(self, inputFrequencies):

    self.inputFrequencies = inputFrequencies

    #Create array of 0s for every 0.01 interval in the audio clip
    outputVector = [1]

    #If a change in pitch is detected, set the current element in outputVector to 1
    currFreq = self.inputFrequencies[0]
    for i in range(1, len(inputFrequencies)):
        currFreq = self.inputFrequencies[i]
        if currFreq != self.inputFrequencies[i-1]:
            outputVector.append(1)
        else:
            outputVector.append(0)

    self.outputVector = outputVector
```

Figure 19: Constructor function for the PitchChangeAnalyser class

It is worth mentioning also that PitchChangeAnalyser does not take the input audio as an input parameter, rather, it takes the product of the outputs of FrequencyAnalyser and ThresholdAnalyser [fig.18]. PitchChangeAnalyser is applied in this way because note changes detected at, say, 1kHz are not useful - that is to say, changes in semitone between 0.001 second intervals are less informative than changes between 0.1 second intervals for our purposes, neither are changes detected when the guitar is not being played.

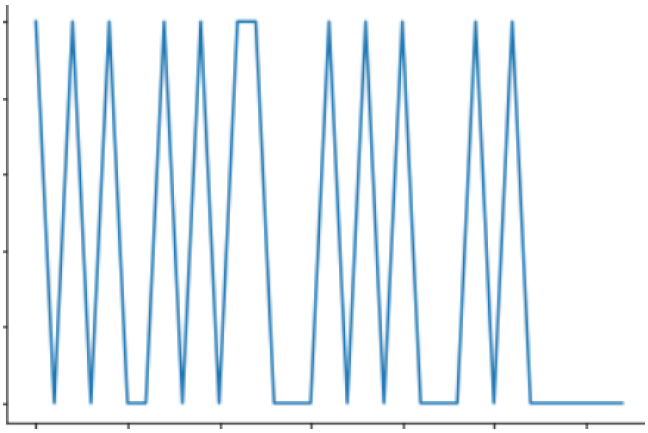

Figure 20: Graph representation of pitch change analysis. x = time, y = pitch changes (represented by a 1)

CompressorAnalyser

CompressorAnalyser [fig.21] behaves like a dynamic compressor - in music, a dynamic compressor will reduce the difference between the highest magnitudes and the smallest magnitudes by lowering the higher magnitudes to lower values (our implementation does the reverse), thereby *compressing* the signal [fig.22]. With a higher threshold for ThresholdAnalyser, we were able to eliminate some of the noise from the output, but in turn lost some of the correct notes that were previously detected. Implementing CompressorAnalyser recovered some of these lost notes without incurring the noise again. It is unclear why this has been more effective than just lowering the threshold, but one might assume that it is because the *shape* of the signal has changed rather than just the threshold magnitude.

```
def __init__(self, audioFile):
    self.audioFile = audioFile
    maximumVolume = self.getMax()

    if self.audioFile.audio[0].shape[0] > 1:  #Stereo compression
        for i in range(len(audioFile.audio)):
            audioFile.audio[i][0] = (audioFile.audio[i][0] + abs( audioFile.audio[i][0] - maximumVolume ) / self.compression_factor)
    else:                                     #Mono compression
        for i in range(len(audioFile.audio)):
            audioFile.audio[i] = (audioFile.audio[i] + abs( audioFile.audio[i] - maximumVolume ) / self.compression_factor)
```

Figure 21: Constructor function for the CompressorAnalyser class

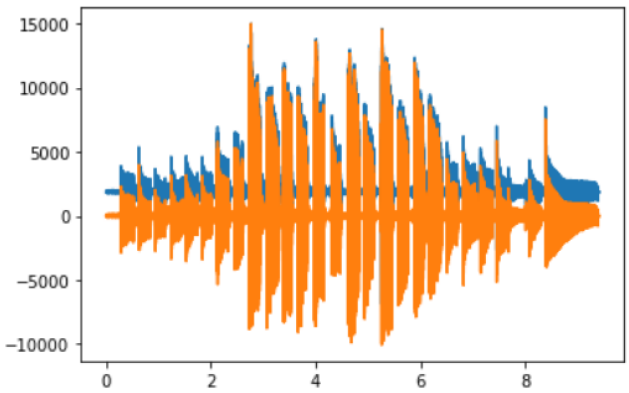

Figure 22: Compressed signal (blue) compared with original (orange), demonstrating how lower frequencies will now have been raised above the threshold

PlaySpeedAnalyser

PlaySpeedAnalyser [fig.23] is not present in the output merging equation defined above because its output is not a representation of the input. Instead, the output is a float value that defines the estimated number of notes played per second. If the value is above a certain limit, the length of the columns in the output tabs will begin to decrease proportional to the amount that the estimated notes per second exceeds said limit. By "length", we are referring to how much time in seconds is represented by each unit - each space that holds a value - in the output tab. For example, if the length of these units is 1 second each, but the performer is playing 3 notes every second, 2/3 of the information gets lost because only 1 of these notes of the 3 can fit into its corresponding cell. In this situation, the system will attempt to extend the overall length of the tab so that the output can be more precise.

```
#Determine playing speed based on the regularity of audio exceeding threshold
def __init__(self, audioFile, thresholdAnalyser):
    self.notesPerSec = sum(thresholdAnalyser.outputVector) / audioFile.audio.shape[0] / audioFile.sampleRate
```

Figure 23: Constructor function for the PlaySpeedAnalyser class

4.2.4 Tab

The constructor function for the Tab class [fig.24] receives two input parameters, one of which being a string that reads "audio" or "file". This string describes the other parameter, which is either an output representation from the Analyser class or the data from a previously saved tab file.

```
#Constructor for new tab
def __init__(self, *args):


    #Instance is created by constructFromFile or constructFromAudio function depending on input type
    if len(args) == 1:
        #If input is .wav file (default setting)
        self.constructFromAudio(args[0])
    elif len(args) > 1:
        if (args[1] == "file"):
            #If input is .tab file
            self.constructFromFile(args[0])
        elif (args[1] == "audio"):
            #If input is .wav file
            self.constructFromAudio(args[0])
        else:
            #Raise exception if input is invalid
            raise Exception("Format not recognised: ", args[1])
```

Figure 24: Constructor function for the Tab class

If "audio" is specified (or if nothing is specified, as "audio" is the default setting), the constructor calls the constructFromAudio() method [fig.25] which translates the input parameter, an Analyser object, into an array that informs the TabLineSegment class (responsible for drawing the tablature on-screen) where to allocate the notes from the performance.

If "file" is specified, the constructor calls the constructFromFile() method [fig.26] which loads a document specified by the user and generates a Tab instance from the data therein. The compatible files have ".tab" extensions and are formatted as a string of text that represents the information from the Tab instance that created it. File handling is implemented using PyQt, and SciPy[9] for wave audio files.

[9]https://scipy.org

```
#Create instance using Analyser instance
def constructFromAudio(self, analyser):

    #Initialise cell length and tab array
    self.cell_length = analyser.cell_length
    self.tabulation = [self.cell_length, []]

    #Variable to store the fret of the first note detected
    prevIndex = -1

    #Enumerate through analyser output
    for i, note in enumerate(analyser.analysis):

        noteLocations = []  #Array storing all fretboard locations of the current note
        currentFret = [0, 0, 0.0]

        #Search fretboard positions for appearances of the note
        for j, string in enumerate(self.fret_positions):
            for k, fret in enumerate(self.fret_positions[j][1]):
                if note == fret:
                    currentFret = [string[0], k, i * self.cell_length]
                    noteLocations.append(currentFret)


        #Append next element to tab array
        #If no note is present
        if len(noteLocations) < 1:
            self.tabulation[1].append([0, 0, 0.0])

        #If the note appears only once on the fretboard
        elif len(noteLocations) == 1:
            self.tabulation[1].append(noteLocations[0])
            if prevIndex < 0:
                prevIndex = noteLocations[0][1]


        #If the note appears mulitple times on the fretboard
        else:
            #Select the note that minimises the distance between frets of the chosen notes
            minDistanceNote = noteLocations[0]
            for i in range(1, len(noteLocations)):
                if abs(noteLocations[i][1] - prevIndex) < abs(minDistanceNote[1] - prevIndex):
                    minDistanceNote = noteLocations[i]

            #Append note to tabulation array
            if prevIndex < 0:
                prevIndex = (minDistanceNote[1])
                self.tabulation[1].append(minDistanceNote)
            else:
                self.tabulation[1].append(minDistanceNote)
```

Figure 25: constructFromAudio() method

Instances of Tab contain arrays of 3-tuple elements in the form [string, fret, time]. These elements are initially produced from the merged representation produced by Analyser [fig.27]. Each element represents a time length equal to the *cell_length* constant. For example, where *cell_length* = 0.25, elements represent 0.25 seconds of audio. To change the length of the units in output tabs, this value can be adjusted without any need to modify analysers - the necessary conversions will take place within Tab. This value will also be changed automatically depending on the output from PlaySpeedAnalyser.

```
#Create instance using .tab file
def constructFromFile(self, tab):

    self.tabulation = [0.00, []];  #Initialise tab data structure

    #Iterate .tab file and create tab array
    cellLength = ""
    lengthRead = False
    currentNote = []
    currentValue = ""
    count = 1
    for value in str(tab):
        if (not lengthRead):
            if value == 'x':
                lengthRead = True
            else:
                cellLength = cellLength + value
        else:
            self.cell_length = float(cellLength)
            self.tabulation[0] = self.cell_length
            if value == '(' or value == '\'' or value == ',' or value == ')':
                pass
            else:
                if value != '-':
                    currentValue = currentValue + value
                if value == '-' and count == 3:
                    currentNote.append(float(currentValue))
                    self.tabulation[1].append(currentNote)
                    currentValue = ""
                    currentNote = []
                    count = 1
                elif value == '-' and count != 3:
                    currentNote.append(int(currentValue))
                    currentValue = ""
                    count += 1
```

Figure 26: constructFromFile() method

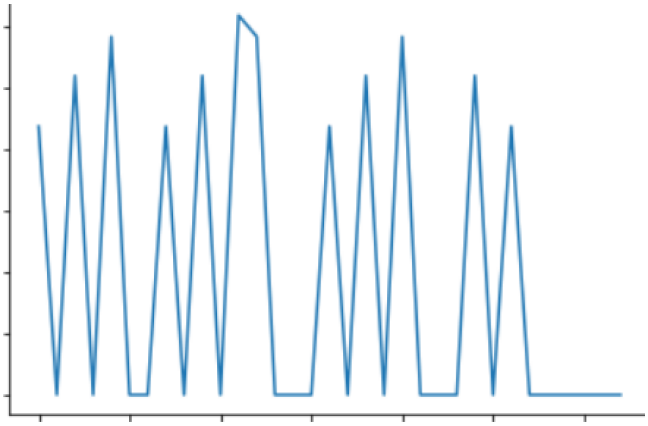

Figure 27: Graph representation of product of PitchChangeAnalyser, ThresholdAnalyser and FrequencyAnalyser. x = time, y = frequency

4.2.5 TabLineSegment

TabLineSegment is responsible for building the PyQt widgets that contain the tabs, and passing those widgets to Ui_TabViewer.

```
def __init__ (self, host, currentLine):

    #Create six lines (for each string)
    self.label = QtWidgets.QLabel()
    self.label_2 = QtWidgets.QLabel()
    self.label_3 = QtWidgets.QLabel()
    self.label_4 = QtWidgets.QLabel()
    self.label_5 = QtWidgets.QLabel()
    self.label_6 = QtWidgets.QLabel()
    tabLineFont = QFont( "Courier" )
    self.label.setFont(tabLineFont)
    self.label_2.setFont(tabLineFont)
    self.label_3.setFont(tabLineFont)
    self.label_4.setFont(tabLineFont)
    self.label_5.setFont(tabLineFont)
    self.label_6.setFont(tabLineFont)

    #Fetch data from Tab object
    self.label.setText(host.tabBuilder.genStringLine(1, self.line_length * currentLine, self.line_length))
    self.label_2.setText(host.tabBuilder.genStringLine(2, self.line_length * currentLine, self.line_length))
    self.label_3.setText(host.tabBuilder.genStringLine(3, self.line_length * currentLine, self.line_length))
    self.label_4.setText(host.tabBuilder.genStringLine(4, self.line_length * currentLine, self.line_length))
    self.label_5.setText(host.tabBuilder.genStringLine(5, self.line_length * currentLine, self.line_length))
    self.label_6.setText(host.tabBuilder.genStringLine(6, self.line_length * currentLine, self.line_length))

    #Add each string to the current line
    host.lineSegments.append(self.label)
    host.lineSegments.append(self.label_2)
    host.lineSegments.append(self.label_3)
    host.lineSegments.append(self.label_4)
    host.lineSegments.append(self.label_5)
    host.lineSegments.append(self.label_6)

    #Add spaces below for formatting
    host.lineSegments.append(QtWidgets.QLabel(""))
    host.lineSegments.append(QtWidgets.QLabel(""))
```

Figure 28: Constructor function for TabLineSegment class

.

4.2.6 Ui_MainWindow

Ui_MainWindow is a PyQt[10] class that executes the initial window that is presented to the user [fig.29]. PyQt is a python framework for implementing GUI systems. The widgets (GUI objects) are relatively straightforward; these are the loading bar, title and logo, and the buttons that implement the use cases described above (apart from the "close program" case which is implemented implicitly).

[10]https://riverbankcomputing.com/software/pyqt/

Figure 29: Main window

4.2.7 Ui_TabViewer

Ui_TabViewer produces a separate window [fig.30] from the main window with widgets for displaying tablature. In addition, a new button is present which implments the "save tab" use case.

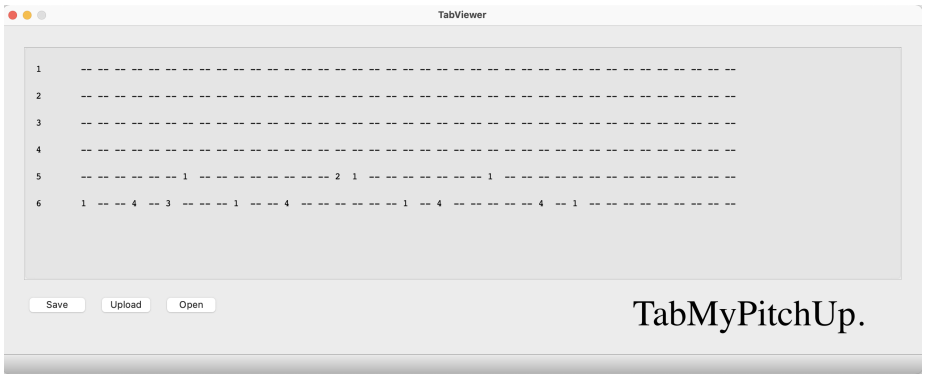

Figure 30: Tab display window

4.2.8 LoadingBar and AnalyseAudio

AnalyseAudio [fig.31] is responsible for commencing the CREPE analysis, and passing the results back to the class that instantiated it.

```
#Analyse audio class
class AnalyseAudio(QThread):
    analysis = pyqtSignal(AudioFile)
    def run(self):
        currentFile = AudioFile(self.sampleRate, self.audioData) #AudioFile class from analysers module
        self.analysis.emit(currentFile)
    def setAudioData(self, sampleRate, audioData):
        self.sampleRate = sampleRate
        self.audioData = audioData
```

Figure 31: AnalyseAudio class

The LoadingBar class [fig.32] is responsible for updating the progress bar [fig.33] at a fixed time interval by an amount that is inversely proportional to length of the input audio clip. In this way, the progress bar will update at regular time intervals no matter the length of the clip, but the amount that it increases by will be smaller for larger inputs, meaning overall it will take longer.

```
#Analyse audio class
class AnalyseAudio(QThread):
    analysis = pyqtSignal(AudioFile)
    def run(self):
        currentFile = AudioFile(self.sampleRate, self.audioData) #AudioFile class from analysers module
        self.analysis.emit(currentFile)
    def setAudioData(self, sampleRate, audioData):
        self.sampleRate = sampleRate
        self.audioData = audioData
```

Figure 32: LoadingBar class

The reason that these tasks have been implemented in separate classes is that the instructions therein can be executed on separate threads. This is necessary so that updating the progress bar does not have to wait for completion of the CREPE analysis, or vise versa, which would obviously be unworkable because then the progress bar would not convey information in real time to the user, which it needs to do. The philosophy behind keeping the user aware of the systems internal state was inspired by Jakob Nielsen's Ten Usability Heuristics[11] and has been implemented as a solution to criticisms received by users during the evaluation process.

[11] https://www.nngroup.com/articles/ten-usability-heuristics/

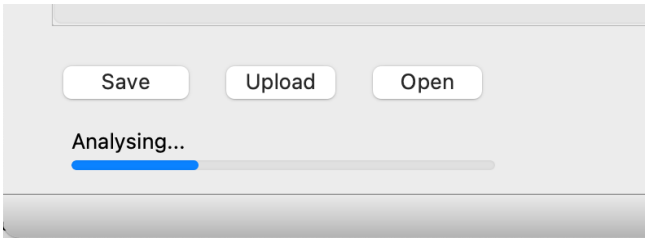

Figure 33: Progress bar widget rendered in a Ui_TabViewer instance

The loading bar widget is actually contained within the active UI class - the LoadingBar class is responsible solely for updating it.

4.3 Overview of evaluation process

The evaluation results will be discussed in the next section, but the following provides an overview of the evaluation process.

The first thing to say, which has been mentioned briefly before, is that the evaluation is divided into two concerns which must be measured differently.

Firstly, the system's efficacy in transcribing music. This must be measured quantitatively so that we can precisely compare between experiments within this project, and so that we can make a claim about the system's position in the broader literature. Given the total notes intended by the musician during the performance, N_{intended}, the notes detected by the system, N_{detected}, and the false positives, whereby the system has detected notes that were not actually intended, $N_{\text{manufactured}}$, we can determine the *precision* and *recall* of the system thus:

$$Precision = \frac{N_{\text{detected}}}{N_{\text{detected}} + N_{\text{manufactured}}}$$

$$Recall = \frac{N_{\text{detected}}}{N_{\text{intended}}}$$

Precision gives the percentage of predictions that are actually correct, while recall gives the percentage of intended notes that were detected by the platform. We can combine these measure into an *F1-score*, which represents both precision and recall in a single value:

$$F1 = 2.\frac{precision.recall}{precision + recall}$$

We will measure these metrics against an evaluation dataset. This dataset has been designed to test the system in different ways to ensure that the platform is effective against a variety of styles and performances. This involved designating a series of categories to which belong five different recordings of the same melody/scale but at different tempos and with different guitar settings. This totalled 45 audio clips in the dataset. The categories are as follows:

Long

The long audio clips consist of the A minor pentatonic scale played back to back five times. These clips last a minimum of one minute each to test the platform's ability to process larger inputs, and display long tabs on screen.

Effects

The effects category contains four sub categories: clean, reverb, delay and distortion. These are twenty recordings of the same melody, but played with different effects pedals active on the guitar, providing a gist of how well the platform generalises across different guitar tones.

A minor and D minor extended pentatonic scales

These categories, containing five clips each, test the systems ability to map detected notes across the neck horizontally; because they are extended scales, they cover a larger range on the neck of the guitar.

C and A arpeggiated chords

These categories also contain five performances each, and test how well the system maps notes vertically; because they are chords, the notes should be playable without the user moving their hand. Because the notes are played on different strings, there is a large frequency range between them, and the system may be tempted to scatter the notes drastically about the fretboard. Furthermore, these notes will be left to ring out, meaning that the system will also be tested on its ability to distinguish background noise from the guitar.

In addition to this, there are qualitative measures that we need to account for as well. These are issues that pertain to how easily users can navigate the system, how readable the tabs are, or how well the platform's features address the full needs of stakeholders.

The approach to measuring these aspects of the system consisted of getting feedback from three guitar musicians. These musicians were shown the output tabs from the above evaluation clips and asked to play them. Their feedback was recorded as an indication of how well the note mapping process had been implemented. The musicians were also asked to input their own recordings to the system and provide feedback on that process, giving us an insight as to what the customer experience would be like. Finally, these assessors were asked to evaluate the following criteria [fig.34] on a pass/fail basis.

Requirement	Context	Expected result
Saving	Tab viewer window	Clicking 'save' allows the user to save a .tab file in their desired location
Load	Main window	The user is able to select a .tab file of their choice, and the data should be preserved
Cancel	File browser opened from main window (loading)	File-browser closes, user is returned to the screen
Load	Tab Viewer window	The user is able to select a .tab file of their choice, and the data should be preserved
Cancel	File browser opened from tab viewer window (loading)	File-browser closes, user is returned to the screen
Upload	Main window	The user can select a .wav file of their choice which is then processed by the platform
Cancel	File browser opened from main window (uploading audio)	File-browser closes, user is returned to the screen
Upload	Tab viewer window	The user can select a wav file of their choice which is then processed by the platform
Cancel	File browser opened from tab viewer window (uploading audio)	File-browser closes, user is returned to the screen
Progress bar	Main window	Progress bar indicates progress of analysis on screen
Progress bar	Tab viewer window	Progress bar indicates progress of analysis on screen

Figure 34: Requirements that are assessed by users

The process ran for six weeks; each week involved implementing a novel algorithm or AMT technique to the platform and measuring the results against

the evaluation test set, and then showing the new platform to stakeholders in weekly user evaluations. The user evaluations were not monitored or guided - they were conducted remotely.

4.4 Minimum viable product (MVP)

Once the data was available for each of the methods in the above described experiments, the architecture of our optimal version came about. A final round of evaluations on the optimal version was conducted on this optimal system to produce the definitive results of the project.

5 Evaluation results

5.1 Statement of week-by-week changes to the system

Week 1

- Week 1 consisted only of initial prototype feedback

Week 2

- Add frets 12-20 to platform
- Implement line breaks to output tabs
- Implement "NoiseAnalyser" to make threshold factor dynamic
- Add filter that removes frequencies between 80Hz and 1050Hz from consideration

Week 3

- Set note mapping to minimise distance from previous note in the tab
- Remove "NoiseAnalyser"
- Increase threshold factor from 0.1 to 0.25

Week 4

- Implement progress bar
- Set note mapping to minimise distance from first note in the tab
- Make cell length dynamic according to amount of pitch changes detected

Week 5

- Add compatibility for mono audio
- Add CompressionAnalyser class
- Remove "maximise" option from window buttons
- Remove dynamic cell length

Week 6

- Applied monospacing to resolve misalignment of strings in output tabs
- Make cell length dynamic according to number of threshold breaches
- Remove CompressorAnalyser

5.2 Qualitative feedback of usability and design

5.2.1 User comments on system

Week 1

- There are no line breaks so longer tabs run off the screen
- Platform tries to attribute notes to muted strums
- Platform is less accurate when notes are allowed to ring out
- Platform struggles with guitar distortion

Week 2

- It takes a long time to analyse, especially longer audio clips
- The notes appear very high on the neck, especially when top strings are played

Week 3

- System crashes when selecting "cancel" option on file browser window
- Note mappings are better, but sometimes result in awkward finger positioning
- High rate of undetected notes on faster performances

Week 4

- Does not work with mono audio
- Spacing between notes is inconsistent - some tabs are stretched and some are squashed
- Higher detection rate for faster performances

Week 5

- Notes are being omitted again with faster performances

Week 6

- Overall length of the tab is not always consistent with the length of the input clip

5.2.2 User comments on output tabs

Week 1

- Some notes will appear much further away than the rest of the notes, making the tab awkward to play
- Notes in scales do not appear in the traditional places
- D minor scale is missing the higher notes

Week 2

- D minor higher notes are now present
- Notes are all mapped way too high on the neck
- Arpeggio tabs are now harder to play

Week 3

- Arpeggios are easier to play because they require less hand movement
- Scales are very difficult to play, and notes on the way down the scale are in different locations than they appear on the way up

Week 4

- Generally, spacing of notes is more appealing
- In some cases, tabs are excessively squashed
- In some cases, there are now an absurd number of false positives

Week 5

- Tabs are more consistently easy to play
- Some tabs have anomalous notes that appear high on the neck, requiring the hand to move up quickly and immediately back down again
- Arpeggios are much easier to play

Week 6

- Tabs are now much more presentable and the strings are no longer becoming misaligned

5.3 Quantitative results of technical performance

Week	Recall	Precision	F1-score
1	93.59	90.68	92.11
2	90.38	92.52	91.43
3	92.31	94.74	93.51
4	92.31	79.82	85.61
5	94.23	90.32	92.23
6	94.01	86.11	89.89

Figure 35: Technical results

5.4 Performance of the MVP

Given the previous results, it was hypothesised that reinstating the CompressorAnalyser, used in conjunction with the dynamic cell length implemented in week 6 and with a threshold factor of 0.25, would be the most effective setup. Although the larger threshold factor had a negative impact on recall, the CompressorAnalyser appears to make up for it without significantly affecting precision. The dynamic cell length based on threshold will be implemented, because it too yielded a high recall, and also, it may resolve the issue some players have of faster performances not being fully transcribed. Finally, the note mappings will be based on minimising distance from the first note in the tab only, because this is what users responded most positively to.

Recall	96.03
Precision	92.81
F1-score	94.39

Figure 36: MVP results

6 Discussion of results

For context, the recall metric gives an account of how many notes were detected out of how many the musician intended. If there are a large amount of notes manufactured in error by the system, which completely obfuscate the useful output in the tabs, this would be reflected only in the precision metric.

In the first week, high recall demonstrates a large number of successful note-detections, but the (relatively) low precision tells us there is a large number of false notes. Users were unhappy at this stage that notes above the 12th fret were not taken into consideration as it made tabs harder to play, and in fact,

Requirement	Context	Pass/fail
Saving	Tab viewer window	Pass
Load	Main window	Pass
Cancel	File browser opened from main window (loading)	Pass
Load	Tab Viewer window	Pass
Cancel	File browser opened from tab viewer window (loading)	Pass
Upload	Main window	Pass
Cancel	File browser opened from main window (uploading audio)	Pass
Upload	Tab viewer window	Pass
Cancel	File browser opened from tab viewer window (uploading audio)	Pass
Progress bar	Main window	Pass
Progress bar	Tab viewer window	Pass

Figure 37: User evaluation of MVP functionality

many unique frequencies exist above this fret, such as higher notes in the D minor extended scale.

In the second week, users were struggling with the fact that more of the notes were being plotted on the bottom strings than before, spreading the notes out along the neck. This is because frets are searched sequentially from the 0th fret of the bottom string; now that frets above the 12th fret were available, the platform was plotting notes exclusively on the bottom string that could also have been in the bottom-middle area of the neck on the middle strings. Using NoiseAnalyser reduced the detection rate (demonstrated by lower recall) but it did clear up some of the manufactured notes in the distorted recordings, having a clear impact on precision. Furthermore, results for the arpeggio recordings were improved, which may perhaps be because the frequency filter was able to clear up some of the background noise interfering with analyses.

In the third week, the problem of all the notes being mapped on the bottom string had been resolved by implementing the new note mapping algorithm, and furthermore, there were less instances of notes being scattered across the neck so drastically. Increasing the threshold had a significant effect on precision. Some of the recall was recovered by removing NoiseAnalyser, but the increased threshold appears to have now reduced the detection rate.

In the fourth week, tabs were more symmetrical as a result of the new note

mapping algorithm; by this, we mean that when a tab walks up a scale, it walks back down on the same frets; previously, because distance was measured from the previous note, the reference point for mapping the notes was moving up the neck with each new note, pushing the rest of the tab up the neck with it. The new algorithm also requires less hand movement. Finally, the dynamic cell length appeared to have no impact on detection rate (see recall) but the precision suffered massively from the increased opportunity for appearances of false positives.

In the fifth week, removing the dynamic cell length resolved the decline in precision somewhat, however precision is still relatively low. This is because the CompressorAnalyser has again pushed more false positives above the threshold, although it has had a positive impact on recall when used in conjunction with a fixed cell length.

In the sixth week, the CompressorAnalyser was removed, and this resulted in a lower precision. This demonstrated the importance of using it. Furthermore, the dynamic cell length based on threshold breaches instead of pitch changes, has had a significant improving effect on recall.

Throughout the experiments, there were no instances where the platform failed the functional tests outlined in [fig.34].

The MVP shows very strong results. These results do suggest that there is potential for a product that can automate music transcription which is fit for purpose. To be truly fit for purpose, the product would need to transcribe polyphonic audio clips, because chords and double notes appear so often in guitar music. The product also demonstrates the efficacy of CREPE as a pitch estimator, and gives an example of how these highly accurate pitch detectors could be used in a practical way. This project can also suggest a format for the design of this type of product, backed up by the positive user feedback.

7 Limitations and future work

The most obvious limitation for this project, already discussed at length, is that of CREPE's inability to process polyphonic audio accurately. This is a huge limiting factor as to the usefulness of the program. In fact, the involvement of CREPE altogether was a significant diversion from the original focus of the project, which was to train a novel NN to understand guitar music. This task would have been much more involved, and would have perhaps meant foregoing the commercial aspects of the current project, such as a GUI system that is navigable by customers. The reason for this diversion was that a functioning NN may not have been feasible by the November prototype deadline, and by the time the deadline passed, too many algorithms had been implemented around CREPE that it did not make sense to return to square one. This did bring the bigger picture into clearer view, however; the current project has demanded a more extensive overview of AMT, and thus more consideration as to how this work fits into it, and also more consideration for how this project may be received by the market - details that may otherwise have been replaced by the

technical minutiae of NN design.

Another issue is the lack of time and data. CREPE likes to take around 2-3 seconds for every 1 second of audio and so evaluation is a long process, causing limitations on the amount of data processed each week. Furthermore, the clips were all recorded by a single musician on the same guitar with the same recording equipment; therefore, any changes made to improve the technical performance of the app were tailored specific to a single style and setup. Although other users provided feedback on their experince with their own performances, this is not factored into the technical evaluation, and if a highly experienced player were to use this system, they may not be expecting results that are as robust as with a more basic performer. This should have a limited effect, since CREPE ultimately is just looking for f(0), but is still important to note because the algorithm responsible for accommodating faster performances is not totally reliable. This is because a performer's speed (as in notes per second) changes wildly from second to second in a performance; although we have used average notes per second as a reference for the cell length - as appose to maximum notes per second, which we feared may make the output tabs far too long for this very reason - this may not be a sophisticated enough measure to comprehend the musical intention of the performer fully enough. Notes will inevitably be missed sometimes, or other times, tabs will be stretched out too long.

Only three users were engaged for the entire duration of the evaluation. Obviously, this is a narrow sampling of the market (although, somewhat fortuitously, the volunteers did fall into distinct categories: amateur, singer-songwriter, and lead guitarist in a touring band) but there is also an issue of familiarity with the system: there was a lot more to be said in the early showcasings of the system, and toward the later weeks our musicians already had expectations of the system and an understanding of its format. It may have been more helpful to replace the focus group each week to observe the true initial reaction of a potential customer for each version.

Before the system would be fully fit for purpose, it would be important to re address the issue of cell length to ensure that faster performers can be accommodated dynamically - it was considered that the cell length of the tab could be manually adjusted in settings, and while this may have been effective, it undermines the intention of the product being responsive to users and automating the tedious aspects of their work. Polyphony must be also incorporated, as well as more robust tools for handling distortion. It was often the case that performances involving distortion effects were often unreadable due to the amount of electrical noise registered by the platform as false positives. If polyphony and distortion could be raised to the accuracy levels seen in the evaluation process, the product would perhaps be ready for a beta testing phase. It may also be prudent to update the GUI wrappers to something less grey.

8 Conclusion

The results from user feedback and technical evaluation are extremely promising, though as we have discussed at length, there are considerable limitations as to how reliable these results can be, as well as how extensively the MVP addresses the task of guitar transcription. It can be said that this paper provides an empirically measured comparison between some various AMT approaches, and a glimpse into what kind of product this market might respond to.

References

[1] S. Haykin, *Neural Networks: A Comprehensive Foundation*. Prentice-Hall, 1998.

[2] F. Rosenblatt, "The perceptron: A probabilistic model for information storage and organization in the brain," *Psychological Review*, vol. 65, no. 6, pp. 386–408, 1958.

[3] L. Noriega, "Multilayer perceptron tutorial," *School of Computing. Staffordshire University*, 2005.

[4] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning internal representations by error propagation," tech. rep., California Univ San Diego La Jolla Inst for Cognitive Science, 1985.

[5] F. Chollet, *Deep Learning with Python*. Manning, 2017.

[6] K. Fukushima, "Neocognitron," *Scholarpedia*, vol. 2, no. 1, p. 1717, 2007.

[7] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[8] E. Benetos, S. Dixon, Z. Duan, and S. Ewert, "Automatic music transcription: An overview," *IEEE Signal Processing Magazine*, vol. 36, no. 1, pp. 20–30, 2018.

[9] S. A. Baumann, "Deeper convolutional neural networks and broad augmentation policies improve performance in musical key estimation,"

[10] M. Won, J. Salamon, N. J. Bryan, G. J. Mysore, and X. Serra, "Emotion embedding spaces for matching music to stories," *arXiv preprint arXiv:2111.13468*, 2021.

[11] R. Hutchinson, "Music theory for the 21st-century classroom,"

[12] Mathworks.com, "Fourier transforms,"

[13] D. Tio, "Automated guitar transcription with deep learning," tech. rep., towardsdatascience.com, 2019.

[14] C. Schörkhuber and A. Klapuri, "Constant-q transform toolbox for music processing," in *7th sound and music computing conference, Barcelona, Spain*, pp. 3–64, 2010.

[15] J. P. Bello, L. Daudet, S. Abdallah, C. Duxbury, M. Davies, and M. B. Sandler, "A tutorial on onset detection in music signals," *IEEE Transactions on speech and audio processing*, vol. 13, no. 5, pp. 1035–1047, 2005.

[16] M.-Y. Kao, C.-B. Yang, and S.-H. Shiau, "Tempo and beat tracking for audio signals with music genre classification," *International Journal of Intelligent Information and Database Systems*, vol. 3, no. 3, pp. 275–290, 2009.

[17] N. Degara, A. Pena, M. E. Davies, and M. D. Plumbley, "Note onset detection using rhythmic structure," in *2010 IEEE International Conference on Acoustics, Speech and Signal Processing*, pp. 5526–5529, IEEE, 2010.

[18] www.intractables.com, "How to read sheet music for beginners,"

[19] T. Barnes, "Why not being able to read music means nothing about your musical ability," *www.mic.com*, 2014.

[20] H. Reid, "Why aren't guitar players better at reading music?," *www.woodpecker.com*, 2016.

[21] C.-F. Arndt and L. Li, "Automated transcription of guitar music," 2012.

[22] J. W. Kim, J. Salamon, P. Li, and J. P. Bello, "Crepe: A convolutional representation for pitch estimation," in *2018 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 161–165, IEEE, 2018.

[23] A. De Cheveigné and H. Kawahara, "Yin, a fundamental frequency estimator for speech and music," *The Journal of the Acoustical Society of America*, vol. 111, no. 4, pp. 1917–1930, 2002.

[24] W. M. Hartmann, *Signal, Sound, and Sensation*. Springer, 1997.

[25] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[26] M. Mauch and S. Dixon, "pyin: A fundamental frequency estimator using probabilistic threshold distributions," in *2014 ieee international conference on acoustics, speech and signal processing (icassp)*, pp. 659–663, IEEE, 2014.

[27] M. Goto, H. Hashiguchi, T. Nishimura, and R. Oka, "Rwc music database: Popular, classical and jazz music databases.," in *Ismir*, vol. 2, pp. 287–288, 2002.

[28] R. M. Bittner, J. Salamon, M. Tierney, M. Mauch, C. Cannam, and J. P. Bello, "Medleydb: A multitrack dataset for annotation-intensive mir research.," in *ISMIR*, vol. 14, pp. 155–160, 2014.

[29] A. Camacho and J. G. Harris, "A sawtooth waveform inspired pitch estimator for speech and music," *The Journal of the Acoustical Society of America*, vol. 124, no. 3, pp. 1638–1652, 2008.

[30] B. Gfeller, C. Frank, D. Roblek, M. Sharifi, M. Tagliasacchi, and M. Velimirović, "Spice: Self-supervised pitch estimation," *IEEE/ACM Transactions on Audio, Speech, and Language Processing*, vol. 28, pp. 1118–1128, 2020.