# Towards Intelligent Autonomous Audio Plugins: Implementing an Intelligent Dynamic Range Compressor Using Machine Learning Approaches

## *'AI Compressor'*

Music Informatics Final Report

Name: James Napier Stuart

Candidate Number: 19329

Supervisor: Dr Alice Eldridge

Music Informatics BA

School of Engineering and Informatics

University of Sussex

May 2014

Approx. 10,000 words

# Statement of Originality

This report is submitted as part requirement for the degree of Music Informatics BA at the University of Sussex. It is the product of my own labour except where indicated in the text. The report may be freely copied and distributed provided the source is acknowledged.

**Signed:**

James Stuart
12/05/2014

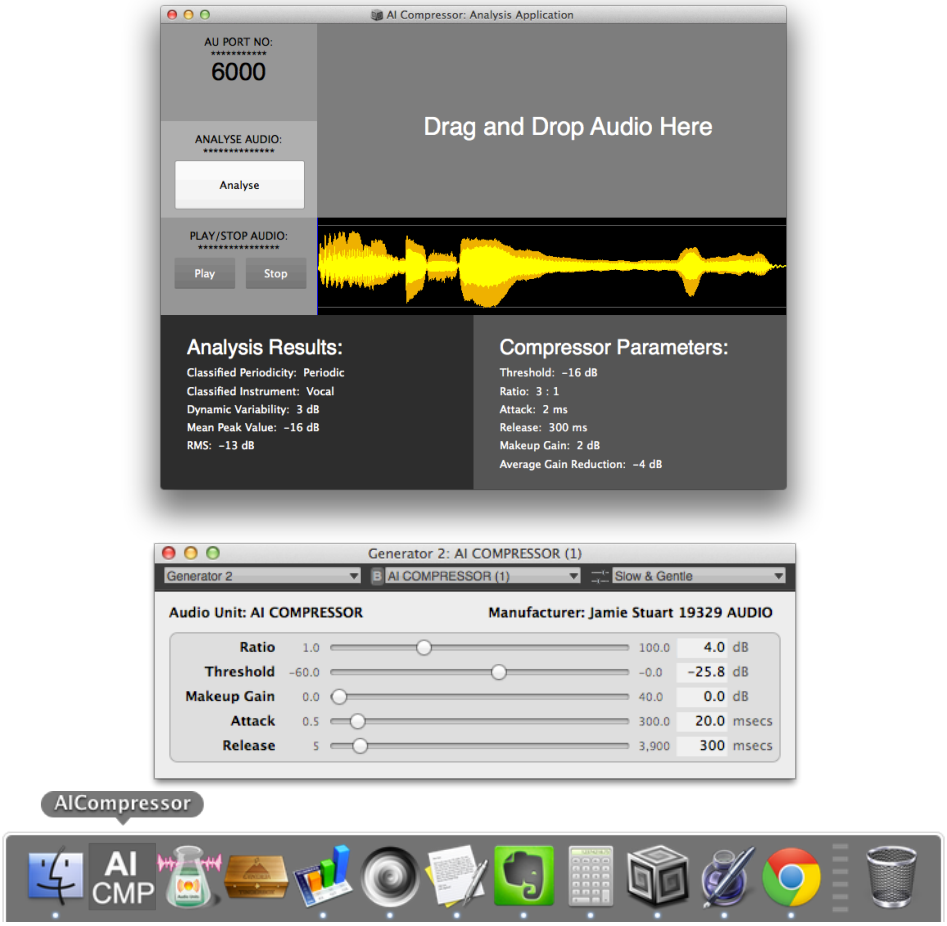
# Acknowledgements

A huge thank you to everybody who has helped me throughout my undergraduate degree. Thanks to the teachings of Nick Collins for my first two years at Sussex and to Alice Eldridge who inspired such enthusiasm and imaginative thinking in my studies. To my Godfather Julio who managed to feature in almost every essay I wrote. To the last remaining MI students, I wish you all the best and cannot wait to start building the weirdest creations the world has seen together. To my brother William Summers and my parents Amanda and Gareth for their unwavering love and support.

James N. Stuart
Candidate No. 19329

# Abstract

This report details the design, build and evaluation of the ‘AI Compressor’, an intelligent automated dynamic range compressor audio plugin and standalone analysis application. The audio analysis interface for music information retrieval and machine learning computations was written in the SuperCollider Language and the compressor plugin’s digital signal processing was built using the C++ AudioUnit Framework. Communication between these two interfaces was handled by Open Sound Control (OSC) protocols. The intelligent compressor is an AU plugin component that users add to their existing plugin libraries, to be used within their digital audio workstations. It functions in a similar manner to other compressor plugins by performing digital signal processing upon audio in real-time but differs by launching an external analysis application to perform machine listening and music information retrieval from the current audio stem being processed. From analysing the audio in non real-time the external application calculates the most suitable compression parameter values (threshold, ratio, attack & release and makeup gain) and sends these over OSC to the real-time audio unit plugin, which adjusts its parameters to the received settings automatically.



# Contents

# 1 Introduction

The ‘AI Compressor’ project aimed to create a dynamic range compressor plugin for the audio mixing domain that intelligently applied suitable processing upon audio stimuli by executing directions sent from an external analysis application. Implementing an intelligent autonomous design meant creating a system that analysed inputted audio from the user and based upon the content of the audio applied effects processing automatically. Digital signal processing was carried out through manipulation of compression parameters such as threshold, ratio, attack, release and makeup gain[1]. The manipulation of such parameters was handled by the external analysis system that used learnt functions, developed with machine learning techniques. The system aimed to bridge non real-time and real-time audio processing and extend research into autonomous audio plugins inspired by the investigations of Queen Mary University’s Centre for Digital Music Department [http://c4dm.eecs.qmul.ac.uk/index.html, last accessed 24/04/14].

Similar automatic mixing projects have been undertaken by academics at Queen Mary University London, such as ‘A Design of a Digital, Parameter-Automated Dynamic Range Compressor’ [Giannoulis, 2010]. However, the employed techniques focused purely on automating parameters within the real-time domain and didn't perform non real-time analysis of overall audio beforehand.

> "The idea of using machine learning techniques to train the compressor in discriminating between instruments so as to be able to recognise the input signal and adjust its automation to it might also be a great improvement although it would be quite challenging to design"
>
> [Gainnoulis, p85, 2010]

Concluding his masters project, Giannoulis states machine learning techniques could be employed for better improvement to parameter automation by classifying the instrument being compressed. This became one of the main justifications for undertaking this project. Within this introductory section I outline the current state of audio plugin processing, analyse attempted intelligent solutions to the problems they rise, and define the objectives needed to be achieved in order to accomplish this project.

---

[1] For an informative overview of the compression parameters implemented, please see Mike Senior’s *Compression Made Easy* article [http://www.soundonsound.com/sos/sep09/articles/compressionmadeeasy.htm, last accessed 24/04/14]

### 1.1 Current State of Audio Plugins: Defining the problem space

Audio Plugins are extensions within a Digital Audio Workstation (DAW) that can process audio in real time or non real-time, embody instruments for synthesis generation or be used as analysis tools. The most common type used by mixing engineers and musicians are real-time plugins, which are placed on channel strips of audio tracks and chained together to effect the sound. For the purposes of this report the term 'audio plugin' is assumed to be real-time specific unless explicitly stated otherwise.

Given the architectural constraints that real-time imposes, audio plugins can only analyse information at two points in time, the present (current amplitude) and the past (stored amplitude values in memory from real-time feeding of audio). Modern plugin models make use of both time paradigms but lack the ability of foreseeing what future amplitude values will be. This lack of knowledge means it is up to the mix engineer to scan through the audio learning the peaks and troughs of the signal to determine the best processing settings to use. This task can be challenging for beginner to amateur level engineers and also for musicians, who may not have received such analytical education that professional mix engineers have. Compression, is an example of a widely used processing tool that is commonly misused within mixing[2]. Users who may understand the functions of the parameters don't posses the correct analytical knowledge to know how to apply the most suitable parameter values given specific audio information.[3]

Plugin presets have been the primary solution to such problems. A preset allows the user to select compression settings for a specific sound or scenario e.g. 'Hard compression', 'Punchy Snare', 'Pop Guitar'[4] providing a solution to the uneducated. However presets recall parameter information stored in memory and therefore are not dependent on the current audio. Since presets don't adapt their settings based upon current audio, misuse

---

[2] Giannoulis also illustrates the 'overuse' of compressors within modern audio engineering, referencing the loudness war within his abstract. [http://c4dm.eecs.qmul.ac.uk/audioengineering/compressors/documents/Giannoulis.pdf, last accessed 24/04/14].

[3] Misuse of compression tools has been documented thoroughly by Mastering Engineer Ian Shepherd, who established the annual 'Dynamic Range Day' and writes thoroughly upon the topic of compression. [http://productionadvice.co.uk/blog/, last accessed 24/04/14].

[4] Arbitrary terms to describe the musical effect of the compressor. Can be tailored for genre specific scenarios such as 'Rock Vocal', instrument specific 'thumping kick' or use terms such as 'light', 'gentle'.

can still occur and the engineer will still have to analyse the content of the waveform to best adjust the preset settings.

### 1.2 Attempted Intelligent Solutions

Defining the problem space showed that audio plugins do not analyse and identify elements within the overall audio content to help set the most suitable parameter values. Attempts have been made by companies to resolve this.

**Real-Time Intelligent Solutions:**

Waves X-Noise [5]


Fig 1.2.1

Image Source: [Waves X-Noise, [http://www.waves.com/plugins/x-noise last accessed 24/04/14]

X-Noise is a plugin used to reduce noise from audio recordings. It 'learns' the information from the audio track in real-time by having the user play through a section of the audio. Noise content is learned and the plugin changes its parameter values automatically. The problem with this approach is that the user must play through the track in real-time for analysis to occur, which is an inefficient use of time, especially when dealing with minutes worth of audio.

[5] For more information on the functionality and use of X-Noise please consult the user guide [Waves, Algorithmix, http://www.waves.com/1lib/pdf/plugins/x-noise.pdf, last accessed 24/04/14]

Queen Mary University's Intelligent Dynamic Compression [Giannoulis, Massberg, Maddams, Finn, Reiss, 2010] (the implementation of the design proposed by Giannoulis discussed above) takes a different approach than the X-Noise plugin. Compression parameters adapt to the audio peaks passed through it immediately, rather than learning the audio data and then changing parameters. Waves Vocal and Bass Rider [http://www.waves.com/plugins/vocal-rider, http://www.waves.com/plugins/bass-rider, last accessed 24/04/14] also follow these principles. The main problems associated with this approach is that they do not allow user automation. A common technique employed by engineers is to automate the parameters of a plugin for a certain section of a track so having parameters that are being continuously automated by the computer restrict this ability. These types of design would be more suited for live performance domains rather than mixing environments[6].

**Non Real-Time Solutions:**

VAMP Plugins: [http://www.vamp-plugins.org/, last accessed 24/04/14]
A vast source of music information retrieval techniques for audio analysis. However, execution operates externally to digital audio workstations.

SCMIR Extensions: [Collins, http://www.sussex.ac.uk/Users/nc81/code.html, last accessed 24/04/14][7]. Music information retrieval extensions run within the SuperCollider synthesis language but share the same problem of the VAMP plugins, that they cannot be implemented within a DAW.

Logic Offline Audio Units and Pro Tools Audio Suite Plugins:
[Apple, Audio Units, https://developer.apple.com/library/mac/samplecode/sc2195/Introduction/Intro.html, last accessed 24/04/14], [Avid, Plugins, http://www.avid.com/plugins, last accessed 24/04/14]. Perform analysis on audio tracks as a whole, feedback to the user via text display of average amplitude values. Useful for determining how to set

[6] Examples of Live Auto Mixing Tools: Dugan Automixer [http://www.waves.com/plugins/dugan-automixer, last accessed 24/04/14]

[7] Research paper by Collins upon his SCMIR extensions detailing their use and capabilities [http://www.sussex.ac.uk/Users/nc81/research/scmir.pdf, last accessed 24/04/14]

plugin parameters. Can perform processing upon track too but this is a destructive process[8].

In summary, current real-time solutions are too time costly and can restrict user control over parameter automation. Non real-time solutions can inform the user with useful information but deeper music information retrieval (SCMIR and VAMP) are isolated outside the DAW and all lack the ability to control a real-time plugin with their computational analysis. There currently exists no communication between non real-time analysis tools and real-time plugins.

### 1.3 Intelligent Autonomous Communication between Real-time Processing and Non Real-time Analysis as a Proposed Solution

From determining the existing pitfalls that plugins still suffer from, the proposed solution was to create a real-time compressor plugin that could communicate with a non real-time analysis application. The analysis system was to perform music information retrieval and machine learning procedures upon given audio and communicate to the real-time plugin how to adapt its parameters, all automatically.

This proposal improves upon the problem areas by analysing all overall audio content so the user doesn't have to. Using the information learnt it dictates the parameter values and therefore solves the problem that presets encounter. Allows user manipulation and automation of controls once its procedures have taken place, and also provides a means of communication between non real-time and real-time processes so deep sound extraction methods can be implemented and inform the real-time plugin.

[8] Destructive Audio processing, is the editing of content of the original sound file. Changing it's content permanently, an irreversible process.

### 1.4 Intended Users

'AI Compressor' is targeted at mix engineers, producers and artists who use compression plugins frequently. It aimed to benefit those who continue to spend long periods of time iterating over audio information every session before applying compression processing. The user base also included beginner to amateur level engineers and musicians who would benefit from an intelligent compression tool to avoid misuse and improve their mixing output. Since users would be most familiar with DAW environments such as Logic[9]/ Pro tools[10]/Ableton Live[11], the implementation had to conform to plugin standards and exist on a higher abstraction level with graphical user interfaces as opposed to having users execute code.

### 1.5 Report Structure

This report follows research carried out upon existing compressor models, current machine learning approaches for instrument classification and music information retrieval techniques. Research into these areas of study, led to establishing requirements for undertaking this project and the report continues to discuss the implementation and evaluation of the built system in terms of classification performance, digital signal processing analysis and eventually concludes the overall success of the project with implications and improvements for further research.

---

[9] Apple's Logic Pro [https://www.apple.com/uk/logic-pro/, accessed 24/04/14]

[10] Avid Pro Tools [http://www.avid.com/US/products/family/pro-tools, accessed 24/04/14]

[11] Ableton Live [https://www.ableton.com/, last accessed 24/04/14]

# 2 Professional Considerations

Considerations below align with the professional standards acceding to the BCS Code of Conduct [http://www.bcs.org/category/6030, last accessed 24/04/14].

**Public Interest:**

To realise this project, the use of existing programming libraries was necessary. For OSC implementation within the AU build, the minimalistic library OscPkt was used [http://grunttheepeon.free.fr/oscpkt/, last accessed 24/04/14]. It is licensed under the open source zlib license [http://opensource.org/licenses/zlib-license.php, last accessed 24/04/14] and uses UDP protocols to send and receive OSC packets, bridging communication between the Audio Unit plugin and SuperCollider Analysis APP [Open Sound Control, http://opensoundcontrol.org/, last accessed 24/04/14].

To build the audio unit, Apple's AudioUnit Framework [https://developer.apple.com/library/mac/samplecode/sc2195/Introduction/Intro.html, last accessed 24/04/14] and Core Audio Frameworks were necessary [https://developer.apple.com/library/ios/documentation/MusicAudio/Conceptual/CoreAudioOverview/CoreAudioFrameworks/CoreAudioFrameworks.html, last accessed 24/04/14]. Since this was a non-commercial project, the AudioUnit Logo License didn't need to be considered[12].

Additional help was required to learn the core audio frameworks, and begin plugin development. Resources used as a foundation tutorial for creating the audio unit are the TemplateAU from Lorenzo Stoakes [https://github.com/lorenzo-stoakes/TemplateAU, last accessed 24/04/14], a modified version for <= Xcode 4.6.3 and <= OSX 10.8, based upon Ghilini's Tutorial code [http://sample-hold.com/2011/11/23/getting-started-with-audio-units-on-os-x-lion-and-xcode-4-2-1/, last accessed 24/04/14]. Let it be noted that this Template AU Xcode project doesn't contain any digital signal processing, it is just a kick starter that links necessary core audio libraries together, that could be easily adaptable for the purposes of this project, but all the DSP was performed by me.

---

[12] Please let it be noted the Apple Links provided can only accessed once you have signed up as a developer. The link to the AudioUnit Logo License for the curious can be found here [https://developer.apple.com/softwarelicensing/agreements/audio.php, last accessed 24/04/14]

To perform feature extraction from audio, the SCMIR (SuperCollider Music Information Retrieval Extensions) were employed [Collins, http://www.sussex.ac.uk/Users/nc81/code.html, last accessed 24/04/14], used in conjunction with my own implementations of machine learning algorithms.

And building the SuperCollider standalone application was made possible by dathinaios' 3.6.6 standalone template [https://github.com/dathinaios/sc_osx_standalone, last accessed 24/04/14], who's shell script was compiled into a native OSX application using Platypus [http://sveinbjorn.org/platypus, last accessed 24/04/14] licensed under the GNU Public License [https://www.gnu.org/licenses/licenses.html#GPL, last accessed 24/04/14].

Finally external audio samples for machine learning purposes were downloaded to use as training data. All downloaded samples were legal, royalty free and provided by musicRadar [http://www.musicradar.com/news/tech/free-music-samples-download-loops-hits-and-multis-217833, last accessed 24/04/14]. Other audio samples also used as testing data were recorded by me when recording and mixing bands. All audio samples, either self recorded or downloaded are 24bit, 44.1kHz sampling rates.

**Duty to Relevant Authority:**

(Relevant authority being the University of Sussex)

Throughout the period of this project, all work carried out will always abide by the University of Sussex's laws and intellectual property rights [http://www.sussex.ac.uk/staff/research/projects/ip, last accessed 24/04/14].

# 3 Requirement Analysis

The development of AI Compressor, required the implementation of digital signal processing, following standards that are consistent with existing compressor models for users to immediately understand and use. Therefore analysis of digital compressors needed to be undertaken in order to implement the best design characteristics for my plugin and the appropriate parameters used. In addition as instrument classification needed to be carried out as stated in the introduction, machine learning and music information retrieval research was carried out.

## 3.1 Plugin Requirements

Firstly the plugin design needed to include standard compression functions. Senior [Compression Made Easy, 2009] outlines different compression forms:

1. Peak Reduction: The simplest is Peak Reduction and Gain controls found on the Teletronix LA2A. Increase in peak reduction results in more compression.

2. Threshold Level Control: When the input signal exceeds a set dB threshold level, compression is activated. The lower the threshold is set (meaning the more likely the signal will exceed it) the more compression occurs.

3. Locked Threshold Level: Used by the Urei 1176LN, a fixed threshold level is provided by the unit. When the signal exceeds compression takes place. It differs from the threshold level controller as the threshold level can't be changed but the gain of the input signal can. Increasing input-gain results in increase of compression.[13]

“I tend to steer newcomers to compression away from 1176LN-style processors, because the overall level increase that you get as you turn up the Input Gain control always tends to give the impression that your processing is improving the sound — even if the amount of compression is inappropriate” [Senior, 2009].

Since Senior advices new users against locked threshold level compressors, I immediately decided to remove this model as a possibility for my implementation.

[13] More Details on these types of compression are detailed here [Senior, Compression Made Easy, 2009, https://www.soundonsound.com/sos/sep09/articles/compressionmadeeasy.htm, last accessed 24/04/14]

### 3.1.1 Existing Compressor Models

**UAD LA2A:**


Fig 3.1.1

Image Source: [Universal Audio emulation LA2A, http://www.uaudio.com/hardware/compressors/la-2a.html, accessed 04/01/13]

The LA2A shown in figure 1.2, is one of the most popular and arguably the best compressor ever made. It uses a 'peak reduction' form as detailed in 3.1, and uses automatic attack and release functions under the hood, so users cannot alter them. The GUI is simple and easy to understand but I wanted user control over attack and release functions within the AI Compressor, and therefore chose a threshold level control type. Furthermore it is an emulation example of the Teletronix LA2A analog compressor[14]. Meaning great lengths were taken to emulate circuitry coloration. The amount of DSP calculations needed for analog emulation is vast, and not needed for the scope of this project.

**Logic Pro's Standard Compressor:**


Fig 3.1.2

Image Source: [Apple, Logic Pro Compressor, https://www.apple.com/uk/logic-pro/, accessed 04/01/13]

[14] Further information relating to the analog emulation implementation of the LA2A compressor can be found here: [Ask the Doctors, http://www.uaudio.com/blog/ask-doctors-ua-modeling-plug-ins/, last accessed 24/04/14]

Logic's standard compressor uses a threshold level control and new users would be familiar with it since it comes packaged with Logic, whereas the LA2A must be bought, downloaded and installed separately from Universal Audio. Familiarity is a positive, however the wide variety of control options is too overwhelming (circuit type, limiter on/off, peak/RMS, limiter threshold) and can be confusing to new users.

**Pro Tools Dyn3 Compressor:**

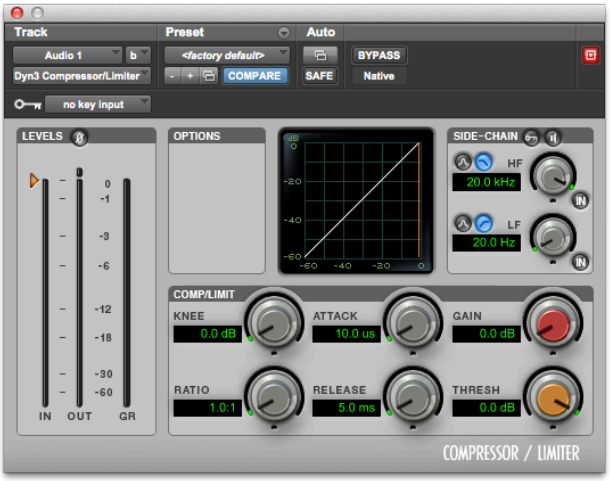



Fig 3.1.3

Image Source: [Avid Pro Tools Dyn3, http://www.avid.com/US/products/Dynamics-III, last accessed 24/04/14]

The Dyn3 comes installed with all Pro Tool versions and so is also a familiar sight to new and experienced engineers. It uses a threshold level control form and provides the most common compression parameters; knee, ratio, attack, release, gain and makeup gain. Since this allowed user manipulation over all these controls and didn't include a vast array of options found in Logic's design, it was chosen as the model for the AI Compressor.

### 3.1.2 Plugin Type Research

Music host applications can support different plugin development types such as Apple's Audio Unit (AU) [https://developer.apple.com/library/mac/documentation/MusicAudio/Conceptual/AudioUnitProgrammingGuide/Introduction/Introduction.html, last accessed 24/04/14], Steinbergs Virtual instrument (VST) [http://www.steinberg.net/en/products/vst.html, last accessed 24/04/14] and more[15]. For the AI Compressor project, I chose to use the Audio Unit plugin type since I had access to Xcode, core audio libraries and the AU Lab application, which was used as a test space for my compressor's digital signal processing implementation and evaluation.

## 3.2 SuperCollider Analysis Application Requirements

The introductory section established that there needed to be a form of communication between real-time plugins and non real-time analysis, therefore research into external music applications that communicate with DAWs was carried out.

### 3.2.1 External Communicative Music Application Research

1. **ReWire:** "ReWire is a system for transferring audio data between two computer applications, in real time. Basically, you could view ReWire as an "invisible cable" that streams audio from one computer program into another." [http://www.propellerheads.se/products/reason/?fuseaction=get_article&article=rewire, last accessed 24/04/14]. The ReWire protocol is used by many companies to communicate their software to and from DAWs. However it focuses on real-time audio handling rather than message communication which was required.

2. **Max for Live**: [https://www.ableton.com/en/live/max-for-live/, last accessed 24/04/14]. Allows Max/MSP implementation within Ableton Live. I could have created a patch that handled parameter automation using Max however analysis was to be performed within SuperCollider for machine listening and therefore this wasn't ideal.

---

[15] Other plugin types such as AAX, RTAS, TDM exist and most audio software companies will port their dsp code to run under all plugin branches for users to implement in any workspace.

3. **Open Sound Control (OSC):** "Open Sound Control (OSC) is a protocol for communication among computers, sound synthesizers, and other multimedia devices" [http://opensoundcontrol.org/introduction-osc, last accessed 24/04/14]. This became the chosen communication platform. Its networking capability meant we could send and receive information between two individual applications just as SuperCollider, Processing, OpenFrameworks and Cinder currently do. Since these applications are built using C++, I knew that there must be an OSC implementation for the C++ library, which proved useful to know because Audio Unit plugins are programmed in C++. I found the library called 'oscpack' [https://code.google.com/p/oscpack/, last accessed 24/04/14] was used as the OSC implementation within SuperCollider and planned to use that for my implementation, however this proved difficult as the implementation discusses.

# 4 Program Implementation

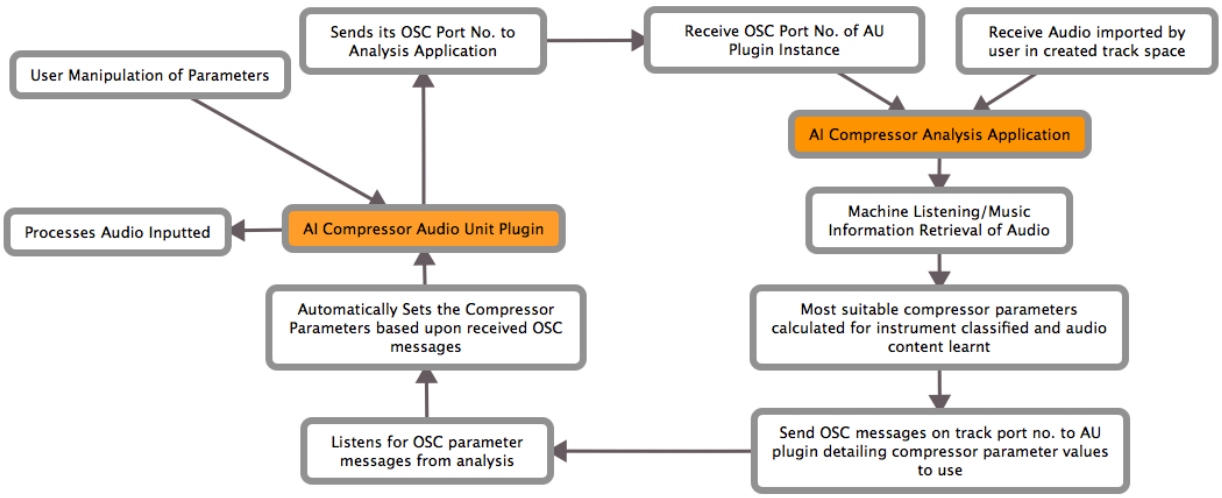



Fig 4.1 AI Compressor Architectural Overview

The ‘AI Compressor’ system, consisted of two main programs. First the Audio Unit Plugin (left) which handled digital signal processing within digital audio workstation environments, allowed user manipulation of parameters and listened for OSC information incoming from the Analysis Application. The second program built, was the analysis application (right) which takes audio information supplied by the user, performs analysis computations, calculates the best suited compression parameters and sends the information across OSC to the Audio Unit. The architectural overview figure reads in a clockwise loop starting from the AU plugin. However, I also decided to enable these two programs to function individually of one another so users can make the decision of the most appropriate time to use them. Giving the option of using both as a whole intelligent system, just the compressor plugin for projects or just the analysis application for insight into audio analytics. This flexibility also allows users the option to use their own compressor plugins, setting the parameter values to the ones the analysis application recommends.

## 4.1 Audio Unit Plugin Architecture

### 4.1.1 Compression Model Implementation

From the requirement analysis the decision was made to implement a simple dynamic range compressor design that used a threshold control level. Upon further research, it was apparent that there are two main types of compressor design of which the threshold control could be used reside under. Firstly ‘Feedforward Compressors’ and ‘Feedback Compressors’.


Fig 4.1.1.1 Feed-Forward and Feedback Compressor Designs
Image source [http://upload.wikimedia.org/wikipedia/en/d/d4/Compressors_Feed_Design.gif, last accessed 24/04/14]

A feedback compressor (right), takes the input into the side-chain after the gain stage, which resembles the way manual gain-riding works. A feed-forward type compressor (left) takes the input into the side-chain before the gain stage, most compressors are based on this design [Izhaki, Mixing Audio, 2008, p274].

With the feed-forward design, audio routes to the side-chain which carries out the processing of the audio signal. When passing through the side-chain the signal is first measured at Peak or RMS, it then detects if the signal is above/below the threshold, calculates the amount of gain reduction based upon the ratio and how far over the threshold it has exceeded, and then the gain reduction is re-calculated based upon attack and release times. Finally after the gain reduction has been applied it is fed into the amplifier or makeup gain stage where the the overall signal is increased by a set amount.

Since the feedforward compressor is the most common design as stated by Izhaki, this became the model for implementation. The Pro Tools Dyn3 Compressor/Limiter and Logic’s VCA compressor are both examples of feedforward designs which is another

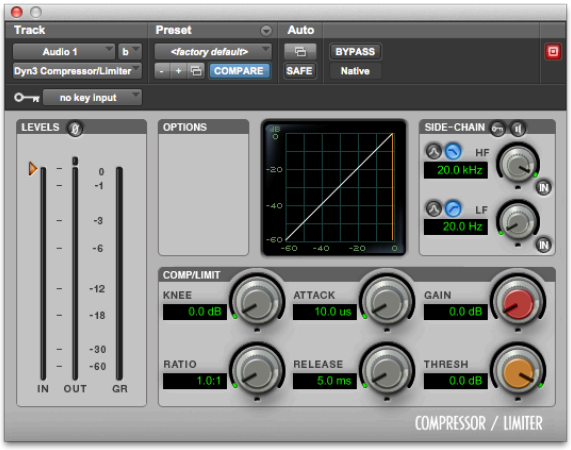
justification for basing the compressors design upon the Dyn3 as discussed in the requirements section and both proved suitable comparison subjects for evaluation.

The compressor will behave differently, depending on whether the side chain responds to average signal levels or to absolute signal peaks [White, Compression & Limiting, http://www.soundonsound.com/sos/1996_articles/apr96/compression.html, last accessed 24/04/14]. The decision to use absolute signal peaks was made for the audio plugin, based upon the inefficiencies that RMS level detection exhibits outlined by Dimitrios [Under the Hood of a Dynamic Range Compressor, http://www.eecs.qmul.ac.uk/~dimitrios/A%20Tutorial%20on%20dynamic%20range%20compression%20design.pdf, last accessed 24/04/14][16].

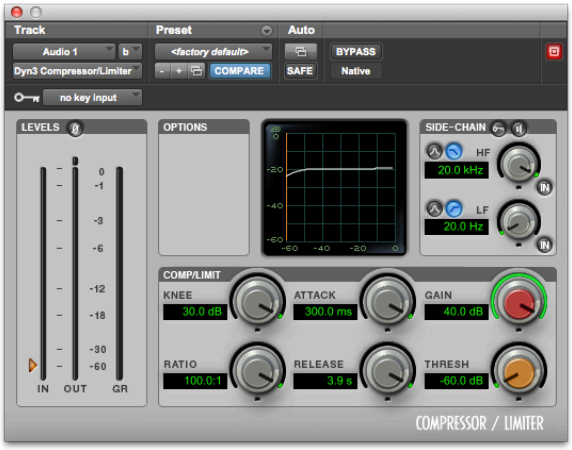
[16] Dimitrios' outlines RMS inefficiencies such as significant delay, the inclusion of an extra parameter and highlights studies that suggest peak and RMS behavior is generally very similar.

### 4.1.2 Digital Signal Processing

Using the Pro Tools Dyn3 Compressor/Limiter as the basis for the design, minimum and maximum values were determined for each parameter.




(Fig. 4.1.2.1 Dyn3 Min Values)




(Fig. 4.1.2.2 Dyn3 Max Values)

| Parameter | Min Value | Max Value |
|---|---|---|
| Threshold | -60.0dB | 0.0dB |
| Ratio | 1:1 | 100:1 |
| Makeup Gain | 0.0dB | 40.0dB |
| Attack | 0.01ms | 300.0ms |
| Release | 5.0ms | 3900ms |

(Fig. 4.1.2.3 Min/Max Table of Values)

| Parameter | Min Value | Max Value |
|---|---|---|
| Threshold | 0.001 peak-peak | 1 peak - peak |
| Ratio | 1:1 | 100:1 |
| Makeup Gain | 0.01 peak-peak | 1 peak-peak (risks distortion) |
| Attack | 0.441 samples | 13230 samples |
| Release | 220.5 samples | 171990 samples |

(Fig. 4.1.2.4 Min/Max Table of Values
DSP Implementation)

Fig 4.1.2.4 displays the minimum and maximum values abstracted from the Pro Tools Dyn3 and converted to peak to peak amplitude values and sample values. The amplitude values have been converted to absolute values (0-1) and time constants from attack and release in milliseconds have been converted to samples. Conversion was the most appropriate way to perform digital signal processing upon incoming audio since the Core Audio libraries function on a peak to peak, sample by sample level within the process() method.

### 4.1.3 Conversion Formulas

Decibel to amplitude value formula (where X = dB value):

$$10 \wedge (X/20)$$

Amplitude to decibel value formula (where X = amplitude value):

$$20 \cdot (\log 10(X/1))$$

MilliSeconds to samples (e.g Attack at 20ms):

1. Divide one second (1000ms) by the attack time set.
1000ms/20ms = 50ms
2. Divide one second by the result to achieve desired attack time.
1000ms/50ms = 20ms
3. Calculate milliseconds in samples
1000 ms = 1 second = 44100 samples
4. Divide amount of samples per second by the attack divide result (50ms).
44100/50 = 882
20ms = 882samples

Conversion formulas were needed within the program as the audio unit GUI operated in decibels and milliseconds whereas the DSP underneath operated in samples and peak to peak amplitudes. Furthermore decibel to amplitude conversion was required when receiving dB and ms information from the analysis application over OSC.

For the compression algorithm implemented please consult the AICompressor.cpp file included within the Xcode project folders.

### 4.1.4 Audio Unit Graphical User Interface

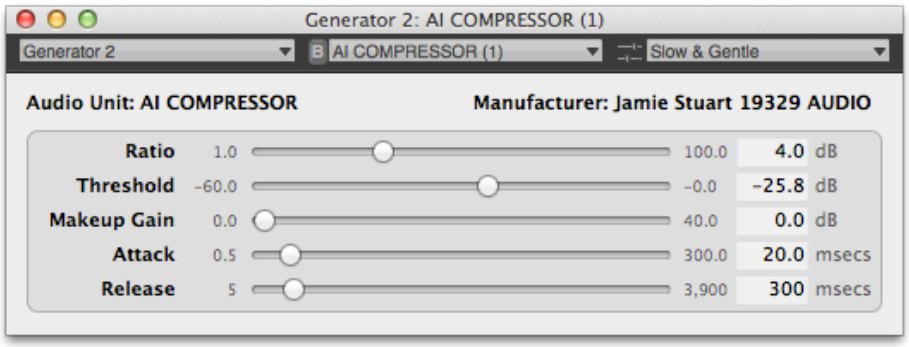



Fig 4.1.4.1 AU Plugin Generic View

The AudioUnit Framework provided by Apple, enables the developer to create either a 'Generic View' or 'Custom View'[17]. The generic view allowed automatic GUI creation from the parameters I set, however care had to be taken to specify the data types for certain parameters such as decibels and milliseconds otherwise the functionality failed. Using a GUI for the compression implementation fulfilled certain requirements, allowing users to manipulate parameters it also facilitated parameter automation within DAWs.

### 4.1.5 Open Sound Control (OSC) Implementation

Requirement analysis found that applications such as SuperCollider used oscpack a C++ OSC library to handle communication between applications. Initial attempts to implement this within the audio unit framework proved very difficult mostly due to the issues of connecting the external library within Xcode. Terminal commands were able to run examples provided but Xcode presented many frustrating problems. Therefore the AI Compressor uses oscpkt for its OSC communication which was much simpler to implement as it consisted of just two C++ header files [http://gruntthepeon.free.fr/oscpkt/, last accessed 24/04/14]. The UDP receiving sockets were used within the Process() method in the AICompressor.cpp code, listening on port 7000. The audio unit first checks if this port is active so as not to impact existing plugin instances and will increment its port

[17] Guide to the implementation of audio unit view's: [Apple, The Audio Unit View,https://developer.apple.com/library/mac/documentation/musicaudio/conceptual/audiounitprogrammingguide/TheAudioUnitView/TheAudioUnitView.html, last accessed 24/004/14]

number until a free one is found. On launch this port no. is sent to the analysis application which is listening on port 6880. The port number 6880 was use instead of SuperCollider's default 57120 to avoid errors that occurred when both were active. Once analysis had taken place and compression parameters calculated the analysis application sent the values across the received AU port number and the plugin changed its parameters immediately. Thus fulfilling the establish requirements of communication between real-time processing and non real-time analysis.

### 4.1.6 AU Validation

Following Apple's guidelines, all audio unit plugins must pass a validation test. This test builds and evaluates the created program and determines if there is anything within it that could crash the host application. The AI Compressor passes the validation test, and it was run within terminal using the following code:

```
auval -v aufx TEST TEST
 auval -64 -v aufx TEST TEST
```

## 4.2 SuperCollider Analysis Application Architecture

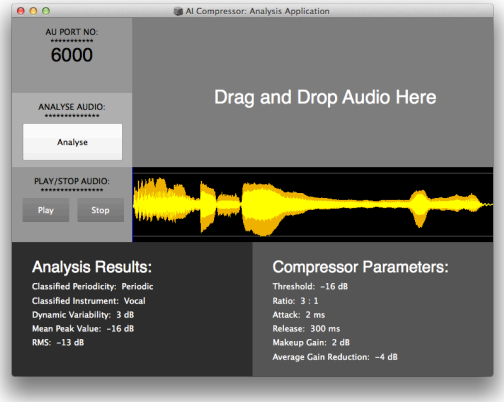



Fig 4.2.1 AI Compressor Analysis Application GUI

The AI Compressor Analysis Application was written inside SuperCollider for all its functionality. This meant music information extraction and machine learning code could be executed concisely and running the program under a single framework instead of branching out to external ones made it time and computationally efficient. The graphical user interface kept to clear, informative aesthetics and presented information that the target user would be familiar with.

### 4.2.1 Class Overview

| Class | Function |
|---|---|
| AICompressor | Creates an Instance of the AI Compressor Analysis Application (Responsible for GUI and high level functionality). |
| CompressorParameterCalculations | Calculates the most suitable compression values for the instrument and audio content provided and conducts offline compression. |
| EguitarVocalClassificationMFCC | Binary Classification via MFCC data with 13 coefficients. Using a Nearest Neighbor algorithm, for electric guitar & vocal. |
| InstrumentClassificationPeriodicNonPeriodic | Binary Classification via Onsets or Tartini. Using a KNN Algorithm for periodic/non periodic. |
| KickSnareClassificationOnsetTartini | Binary Classification via Onsets or Tartini. Using a KNN Algorithm, for kick and snare. |
| WriteTrainingDataMFCC13 | Writes Training Data to CSV files for MFCC data with 13 coefficients. |
| WriteTrainingDataOnsetTartini | Writes Training Data to CSV files for Onsets or Tartini SCMIR extensions. |

### 4.2.2 Machine Learning and Music Information Retrieval Processes

Since all programming was coded within the SuperCollider language, I decided to use the SCMIR extensions from Collins for music information extraction. These extensions were integrated within the machine learning algorithms as feature labels for musical instrument classes. At the time of programming this project, the machine learning extensions that existed within SuperCollider were not stable. I could have decided to use a different programming language to take care of the machine learning for me, but I wanted to implement my own algorithms to understand how to implement a model for instrument classification myself.

The SCMIR extensions currently support a number of SuperCollider analysis UGens for non real-time feature extraction[18]. The ones I found most useful for instrument classification were Onsets (Onset Detection), Tartini (Pitch Probability) and MFCC 13 (Mel-Frequency Cepstrum of thirteen coefficients). All data values extracted are normalized floating point numbers between 0-1. Training data was generated using these feature extractions upon four different instruments; Kick Drum, Snare Drum, Electric Guitar and Vocal. These were selected since they are some of the most common instruments engineers deal with when mixing.

Somerville and Uitdenbogerd found that using a K-nearest neighbour approach to instrument and timbral classification with MFCC feature extraction gained higher performance ratings than Naive Bayes or decision tree models[19]. Therefore the AI Compressor's machine learning adopted both nearest neighbour and K nearest neighbour models.

---

[18] SCMIR supported uGens: MFCC, Chromagram, Loudness, Tartini, SensoryDissonance, SpecCentroid, SpecPcile, SpecFlatness, FFTCrest, FFTSpread, FFTSlope, Onsets in raw detection function mode, Transient, SpectralEntropy, PolyPitch

[19] Multitimbral and Musical Instrument Classification [http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4654099&tag=1, last accessed 24/04/14]

As fig 4.2.2.1 illustrates, an MFCC with thirteen coefficients relates to thirteen frequency bins based upon the mel scale. “In technical terms, the coefficients are the result of matching cosine basis to frequency-warped spectrum. The MFCC UGen uses the Mel scale, a common psychoacoustic scale modeling auditory sensitivity on the basilar membrane” [Collins, Machine Listening, The SuperCollider Book, p446].

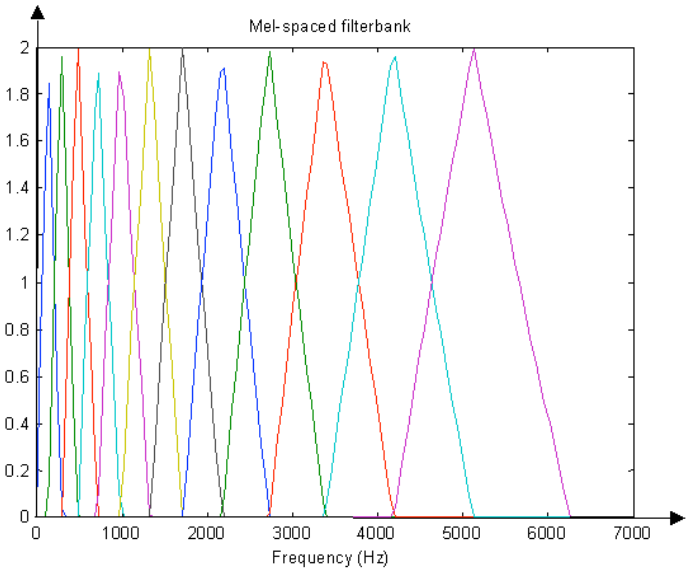



Fig 4.2.2.1 Mel Frequency Cepstral representing 13 coefficients
Image Source: [http://i.stack.imgur.com/YUH48.gif, last accessed 24/04/14]

Initial attempts at using MFCC data were very naive, writing CSV files of the coefficient numbers rather than the amplitude values associated with them. After fully understanding that the MFCC UGen takes amplitude values for the thirteen coefficients per frame of audio, I was able to take the mean, variance and standard deviation of the amplitude values for each coefficient. This resulted in a spectral plot overview per instrument, which unlabeled test samples could be compared to. The WriteTrainingDataMFCC13.sc class (see class overview) writes the standard deviations of each coefficient per audio sample as an array of values to CSV files for training data.

Median values were taken for Onset and Tartini feature extraction. This was because the mean seemed to include far more outliers in the audio information and caused classification problems whereas the median seemed to reduce these and improved overall reliability. Cross Examination of learning models and feature labels (see appendix 8.1), found that using a nested binary approach to classification yielded the best results in comparison to a single multi-classification model.

The nested approach upon imported audio is as follows:

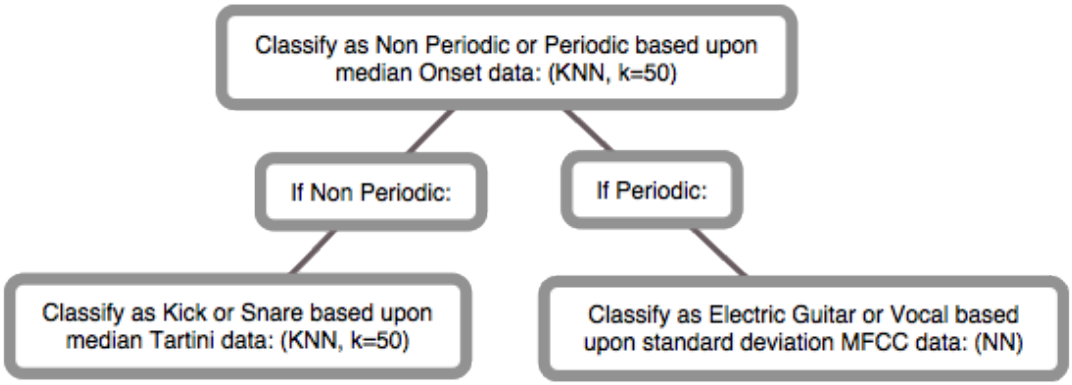



Fig 4.2.2.2 Nested Binary Classification Implementation

The classes that take care of the machine learning algorithms that I programmed from scratch are the InstrumentClassificationPeriodicNonPeriodic.sc, KickSnareClassificationOnsetTartini.sc, and EguitarVocalClassificatonMFCC.sc found in the class overview 4.2.1. For a deeper analysis please consult the source code. The various machine learning models outline here, are tested are evaluated in section 5.4.

### 4.2.3 Deciding Compressor Values based upon Music Information Retrieval

The second process the analysis application performed was the calculation of suitable compression settings based upon the instrument classified and further music information retrieval techniques. Firstly, the ratio, attack and release parameters were determined by the classified instrument using a constant set of values acquired and adapted from Sound on Sound's 'Compression and Limiting' useful compression settings [http://www.soundonsound.com/sos/1996_articles/apr96/compression.html, last accessed 24/04/14].

| | Ratio | Attack | Release | Hard/Soft | Gain Red. |
|---|---|---|---|---|---|
| Kick | 6:1 | 20ms | 0.2ms | Hard | 5-15dB |
| Snare | 5:1 | 20ms | 0.2ms | Hard | 5-15dB |
| Electric Guitar | 8:1 | 5ms | 500ms | Hard | 5-15dB |
| Vocal | 3:1 | 2ms | 300ms | Soft | 3-8dB |

Fig 4.2.3.1 Sound On Sound Table of Compression Values

Using just this methodology alone for calculating the compressor parameters would not have been an improvement over the current plugin architecture since with current designs a user can find a preset related to the instrument within the plugin menu which will set these values for them. The main pitfall of these 'preset' systems as highlighted within the introductory section, is that the threshold doesn't adapt to the audio and therefore will not provide a suitable amount of compression for the dynamic range being processed. Therefore it was necessary to set a threshold level based upon dynamic content retrieved from the audio. Referring to the table of values the gain reduction column was a useful foundation at trying to gage a successful threshold level.

Comparisons were made between RMS and Mean Peak values as values for threshold settings. I found that the mean peak value exhibited gain reduction results that better fitted the ranges suggested by Sound on Sound. Therefore the threshold level for the compressor was set at the absolute mean peak value of the audio signal inputted. Evaluation of using this approach can be read in section 5.3, which discusses the restrictive nature of using mean peak value as the threshold level.

The algorithm for gaging the makeup gain's value was based upon a DSP post by Tom Duffy of Tascam compressors [20]. Duffy explains "the functionality of Tascam's automatic makeup gain control is to calculate the amount of gain reduction that would be applied to a 0dBFS signal, halve it and add it as makeup gain" [computing compressor automatic makeup gain, 2009, http://music.columbia.edu/pipermail/music-dsp/2009-august/068025.html, last accessed 24/04/14].

MakeUpGain Illustration:

Threshold = -14dB
Ratio = 3:1
A 0dBFS signal is reduced -14/3dB
-14/3dB = -4.67dB
Gain Reduction = -4.67dB
Half = -2.3dB
Amount of Makeup Gain = +2.3dB

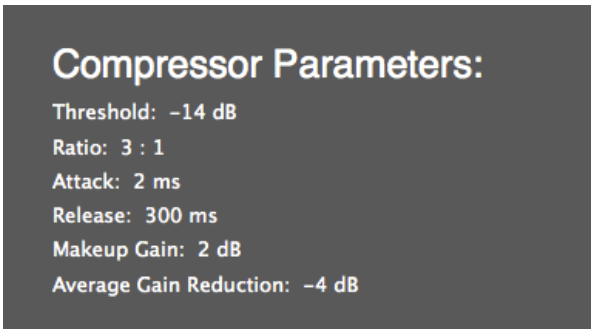



Fig 4.2.3.2 AI Compressor Parameters

Fig 4.2.1.2.1 demonstrates the successful implementation of the Tascam algorithm described by Duffy. The AI Compressor differs from the Tascam design using an average gain reduction calculation since the analysis is covering the whole track. To achieve these results 'offline compression' had to be carried out (found within the CompressorParameterCalculations.sc extension) that looped through the array of amplitude values and using the determined threshold and ratio values, found the average gain reduction. With the gain reduction defined it was able to provide a suitable makeup gain amount based on the Tascam algorithm.

In addition to the standard compressor parameter values, analysis results were displayed on the left side of the GUI stating the periodicity, instrument classified, Mean Peak Value, RMS and the Dynamic Variability. The dynamic variability measured the level of dynamic variation within a track. This differed from the dynamic range (difference between lowest and highest peaks) as it measured the difference between Mean Peak Value and RMS. It

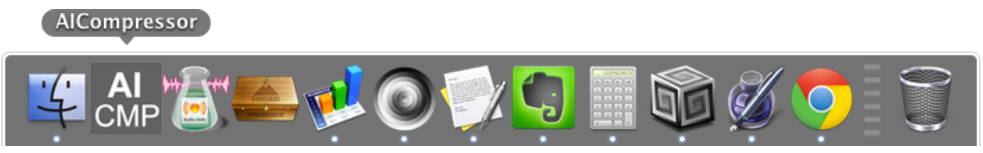
[20] Tascam Audio [http://tascam.com/, last accessed 24/04/14]

aimed to inform the user of the dynamic content of their audio and was modeled on the TT Offline Dynamic Range Meter [http://www.dynamicrange.de/, last accessed 24/04/14]. It should be noted though that measuring the dynamic variation of a track can be approached using different methods, and there is currently no definitive technique used to calculate it. This measure is used as an informative purposes and features largely within loudness war debates [Deruty, 2011].

### 4.3 Creating a SuperCollider Standalone (Compiling to Native Application)

The initial aim of implementing machine learning and music information retrieval processes, was to create a set of SuperCollider extensions to be executed within SuperCollider as a proof of concept. However the build and distribution of the analysis program as a standalone application managed to be realised in addition to the SuperCollider extensions meaning it could be executed as a native application on OSX, accomplishing a future project aim.

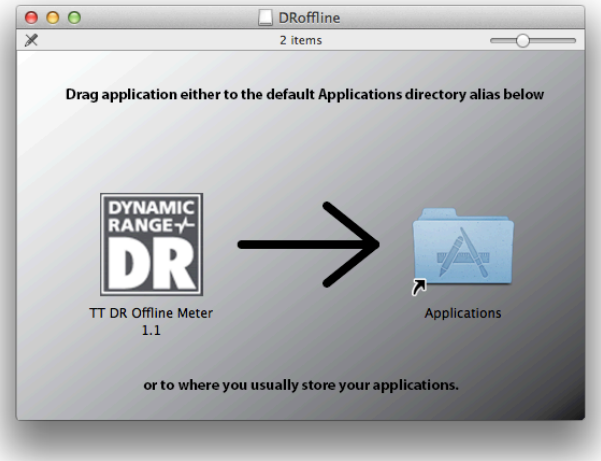
Currently there appears to be a wide spread issue of compiling and running a build of SuperCollider 3.6 and higher on Mac OS 10.8 machines. Many users have turned back to 3.4/3.5 for creating their own standalones however the functionality of my code uses updated classes such as OSCFunc to send and receive OSC messages outdating older classes such as OSCRepsonder and others. Fortunately a modified Linux 3.6.6 standalone template from dathinaios [https://github.com/dathinaios/sc_osx_standalone, last accessed 24/04/14] resolved this. After compiling and modifying the start up code from this standalone template the AI Compressor analysis application was able to run on mac computers without users having to install SuperCollider. However, mandatory source code editing of the common ‘Score’ class had to be undertaken to solve a problem that arises when the SCMIR extensions are moved from the appResource library. When moved, the classes no longer point at the original scsynth executable and so overrating the Score class was necessary [http://new-supercollider-mailing-lists-forums-use-these.2681727.n2.nabble.com/SCMIR-and-SC-3-5-rc3-td7342938.html, last accessed 24/04/14]. Furthermore the SCMIRFile class creates its own synth def, as does my code. Synthdefs would normally be stored in the user/library/applicationSupport/SuperCollider path and so errors occurred when building the synthdefs within the standalone. Therefore resetting the “SC_SYNTHDEF_PATH” and “SC_PLUGIN_PATH” was a necessary overwrite that was coded so the application could point within its own resources and find the required directories.



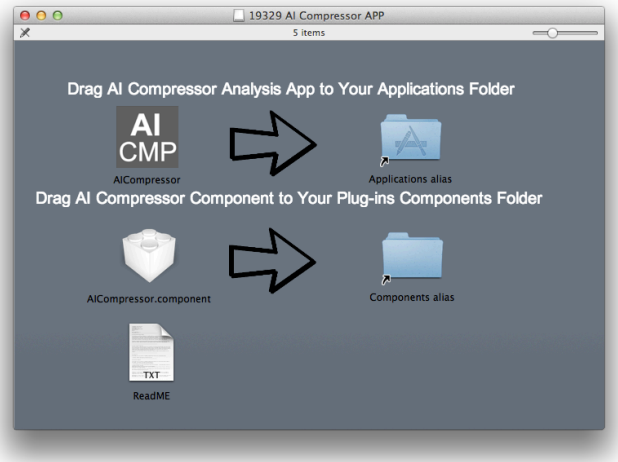

(Fig. 4.3.1 AI Compressor Dock Icon)

### 4.4 Packaging up the Audio Plugin and Analysis Application as Disk Image

Experience with purchasing music applications and audio plugins gave insight into the distribution of these products. The most common and clean way for users to get hold of audio plugins and apps is through downloading Disk Image (DMG) files from an audio company's website. These files provide the application/component content and alias folders for users to simply drag and drop the new files to the necessary destinations without having to see read me's or manuals beforehand. Disk Image conventions have a background image that has text explaining the action the user must perform and an arrow of where to drag their products to. This although surface level design, was still a step forward for this project, since the intended users are be mix engineers and musicians who are familiar with these paradigms and won't expect anything less than an easy install.




(Fig. 4.4.1 Dynamic Range DMG)
Image Source:
[http://www.dynamicrange.de/, last accessed 24/04/14]




(Fig. 4.4.2 AI Compressor DMG)

An exemplary DMG file (Fig 4.4.1) shows the app with details to drag to the users (alias) application folder. Fig 4.4.2 shows my implementation of a DMG file, who's design was based upon fig 4.4.1 using a background image to instruct users of interaction expected and arrows for emphasis. It is a clear design, easy to comprehend and therefore fulfills the purpose of being easy to install[21].

[21] The tutorial followed to create the disk image can be found here, [Walraven, Creating a mac Disk Image Installer, http://jtechdev.com/2013/04/05/creating-a-mac-disk-image-installer/, last accessed 24/04/14].

### 4.5 Version control

Github was used as a means of version control. Throughout this project I spent time committing and updating files to a private repository, where only I had access to viewing and editing the code. I was new to Github at the start of the project and although I still perform local back ups I found it very useful to compare the differences between program versions. Extending the use of Github, I created a public repository, which made it an excellent destination for distributing the final project. All the project data can be found here: https://github.com/jamesnapierstuart/19329-AI-Compressor

# 5 Evaluation

Since the goal of this project was to create proof of concept for communication between non real-time analysis and real time processing of audio, evaluation methodology focused upon the capabilities and functionality of system performance in terms of instrument classification accuracy, compression calculation suitability and DSP implementation. In addition, subjective analysis was carried out to assess whether the system improved the sound when automatic compression was applied, for future projects user testing would be a vital and great benefit to evaluate the system but at this stage it was not necessary.

## 5.1 Audio Unit Evaluation

### 5.1.1 Host Application Testing

Apple's AU Lab application was used as the primary test space for the audio plugin. AU Lab is a mock host application that can implement created audio units to process live input or audio files. I found that there were differences between Logic's and AU Lab's use of my audio unit when sending OSC messages. The UDP ports within the audio plugin, were programmed within its process() method and in AU lab this process is continuously called as soon as the plugin is launched. However within the Logic the process method is only called when the user is playing over audio in real-time. This presented automation problems when sending the OSC data from the analysis application. When Logic was not playing audio and OSC parameter values were sent to the plugin, it wouldn't immediately change its settings, yet as soon as I pressed play they would audibly move to the correct values. In a real world scenario this would be confusing to a user who would require immediate feedback. Possible solutions would be to create listeners within the audio unit that run on their own loop function constantly listening but this risks becoming a higher priority thread and as the Audio Unit development guide points out only the process method can have the highest priority.

### 5.1.2 DSP Evaluation

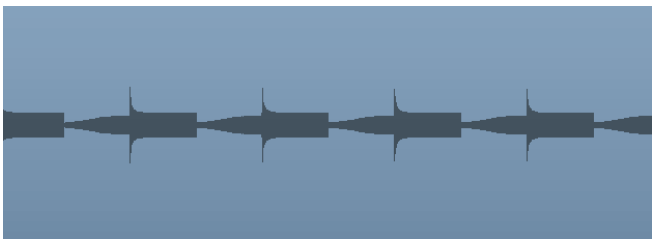


Fig 5.1.2.1 Dyn3 Compression Upon White Noise

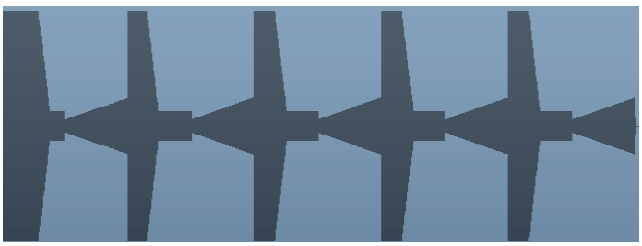


Fig 5.1.2.2 AI Compression Upon White Noise

**A/B Testing:**

Evaluation of the compressor plugin was carried out by comparing similar designs within a DAW, and analysing its effectiveness audibly and visually by comparing audio waveform graphics.

By analysing the graphical representation of the processed waveform, we are able to detect the differences between each plugins effect. Fig 5.1.2.2 shows an early attempt of attack and release implementation which although displays a ‘push and pulling’ manor it by no means compares to the Dyn3’s waveform in fig 5.1.2.1 Performing evaluative measures such as these was necessary as at times the changes to the waveform were not as obvious audibly as they were visually.

Final auditory A/B testing with Pro Tools Dyn3 and Logic’s standard compressor (VCA setting) showed that the attack settings performed similarly to the AI Compressor plugin. Exhibiting distorted artifacts with the fastest attack settings and smoother sounds with 5ms and greater attack settings. The release parameter on the other hand didn’t seem to effect the signal processing, on low levels the sound was similar to that of the Logic VCA compressor but when pushed to extremes the Logic plugin showed audible pulsating whereas the AI Compressor showed no audible change. In earlier iterations, I was able to create a release effect but it was accompanied by unwanted distortion and therefore was left out of the final realisation.

I could have settled for a SuperCollider implementation using the Compander class for compression, which would have definitely exhibited better sonically pleasing and accurate results compared to my DSP program. However, learning to create a basic plugin implementation within the audio unit framework was something I wanted to undertake. It was definitely and still remains one of the toughest programming challenges during my undergraduate degree because it uses no help from higher level abstracted libraries and focuses completely upon sample by sample DSP. Although it didn't yield the best sonic outcome I was proud of trying to reach beyond my current skill level in the attempt to increase my own education. Even though the final plugin implementation didn't reach the standards set by Logic or Pro Tools, the most important outcome gained from this aspect of the project, was how to evaluate plugin tools effectively.

### 5.1.3 Graphical User Interface

Apple's generic view (standard audio unit GUI), lacked graphical aesthetic and it didn't provide visual feedback upon the amount of gain reduction being applied to the signal. The graphics of a compressor can be important to audio engineers as they can imitate analog designs, to make engineers feel comfortable using the digital emulation. However, the look does not effect the signal processing and since this project was not focusing on making a top quality dynamic range compressor but rather an intelligent one that can communicate with external applications, the GUI was not the top priority. Future developments would see a Custom View implemented that can facilitate AudioUnitEventListeners which can make GUI sliders adapt to messages received, a limitation of the generic view and the reason why sending OSC messages shows no visual change.

## 5.2 Open Sound Control Evaluation

Overall the OSC implementation proved very successful, with low latency and immediate audible results. Areas of improvement include moving the UDP listener functionality within its own priority loop outside of the process method, so that automation doesn't require user playback as section 5.1.1 describes.

### 5.3 Compression Calculation Evaluation

Although the DSP implementation didn't achieve the best sonic results, tests were carried out to discover whether the compression values calculated by the analysis application would exhibit the same amount of gain reduction applied to an audio signal using third party compressors such as the Dyn3.

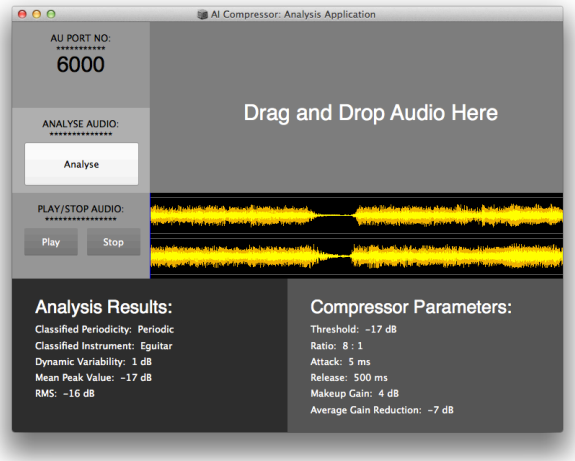



Fig 5.3.1 AI Compressor Analysis Application Compressor Calculations


Fig 5.3.2 Pro Tools Dyn3 Compressor

Figure 5.3.2. proves the settings calculated from the analysis application will give the same results for other compressors. E.g. the average gain reduction value -7dB is shown to be found with the Dyn3 Compressor using the settings suggested. Using the mean peak value as a threshold level for electric guitar and vocal I obtained favorable compression results whose gain reduction levels fell within the suggested Sound on Sound ranges (see section 4.2.2). With kick and snare instruments though, gain reduction levels were -15dB or lower which exceeded the suggested range, therefore the threshold measure should be adjusted depending on the instrument classified rather than using a generalised formula.

### 5.4 Evaluating Employed Machine Learning Approaches

As discussed within the implementation, nearest neighbour and K-nearest neighbour machine learning models were used for instrument classification. Numerous tests were performed on different features for classification comparing Binary and Multi-classification techniques and their performance was measured to determine which was the best to use. The application distinguishes between four instruments Kick, Snare, Electric Guitar and Vocal. Audio samples for each instrument were split into training data and testing data corpuses, with a 70% split for training data and 30% split for testing data. Taking a 70% split of audio samples instead of a higher value meant that we did not risk over-fitting the data and making the classifier too accustomed to these trained samples.

Below assesses the performance of the binary classifiers that were chosen for the analysis application, for all contingency tables collected and full classifier evaluation please consult appendix 8.1).

K-Nearest Neighbour, Periodic/NonPeriodic Binary Classification of Onsets Data: (Even Data, K = 50)

| | **Predicted Periodic** | **Predicted NonPeriodic** | |
|---|---|---|---|
| Actual Periodic | 233 | 12 | 245 |
| Actual NonPeriodic | 2 | 243 | 245 |
| | 235 | 255 | 490 |

Overall Classifier Accuracy = 97%

Error Rate = 3%

Precision Periodic = 99%

Recall Periodic (True Positive Rate) = 95%

Precision NonPeriodic = 95%

Recall NonPeriodic (True Negative Rate) = 99% [22]

[22] True Positive Rate (Sensitivity) = Correctly predicted Periodic/Amount of actual Periodic amount.
True Negative Rate (Specificity) = Correctly predicted NonPeriodic/Amount of actual NonPeriodic.
Precision = Proportion of Periodic predictions that were correct.
Recall = Proportion of Periodic instance that are predicted (also TPR).
(The Precision and Recall can be carried out for both class labels since we are interested in both Periodic and NonPeriodic classification results).

K-Nearest Neighbour, Kick/Snare Binary Classification of Tartini Data:
(Even Data, K = 50)

| | Predicted Kick | Predicted Snare | |
|---|---|---|---|
| Actual Kick | 83 | 7 | 90 |
| Actual Snare | 29 | 61 | 90 |
| | 112 | 68 | 180 |

Overall Classifier Accuracy = 80%

Error Rate = 20%

Precision Kick = 74%

Recall Kick = 92%

Precision Snare = 90%

Recall Snare = 68%

Nearest Neighbour, Electric Guitar/Vocal Binary Classification of MFCC 13 Data:
(Even Data)

| | Predicted Eguitar | Predicted Vocal | |
|---|---|---|---|
| Actual Eguitar | 90 | 0 | 90 |
| Actual Vocal | 8 | 82 | 90 |
| | 98 | 82 | 180 |

Overall Classifier Accuracy = 96%

Error Rate = 4%

Precision Electric Guitar = 92%

Recall Electric Guitar = 100%

Precision Vocal = 100%

Recall Vocal = 91%

Since binary classification of periodic and non periodic audio yielded the highest overall accuracy, this was chosen as the first classifier to separate the kick and snare from electric guitar and vocal. As contingency tables in appendix 8.1 show, multi-classification confused the classifiers more than when the instruments were divided into two groups. The above tables illustrate that precision and recall measures were both generally high, however we would benefit more from low recall and high precision percentages.

Implementing a nested binary classification path proved the most suitable model however the accuracy and performance measures shown above and displayed in the appendix only evaluate individual classifications rather than the nested binary classification as a whole. Also, multiple features such as FFT crest, Transient, and Spectral Centroids could have been used per class label to give a wider view of the vector space.

For the K-nearest neighbours classifiers, the amount of nearest neighbours was set to fifty (K=50). Initial tests carried out found K=50 acquired much higher accuracy levels than K=10. Setting K at a higher level reduced the amount of noise in the data but setting it very large meant we risked including more samples, which could have led to incorrect classification. To improve, a method for the calculation of K should have been implemented rather than a trial and error approach. As the implementation outlines, weights were applied to the neighbours to improve accuracy. Those closer to the test value have higher weight ratings and means are compared between the positive and negative class labels to determine which class the test instance is nearest to.

As discussed in the implementation the Onset and Tartini features were extracted at median values per audio file. Where this seemed to achieve reasonable classification results, on a larger corpus of data taking just a single value per audio track will become too restrictive. Therefore future implementations should see an array of median values taken over intervals of time within the audio to construct a finer defined picture to compare test data too. Based upon existing instrument classification research I chose to use nearest neighbour and K-nearest neighbour approaches, however to validate the classification models, comparisons could have been performed to evaluate which model would have performed best in this scenario.

For the purposes of evaluating which classification methods would be best to use, running the algorithms once per method was sufficient enough to compare classifiers against one another in terms of accuracy and performance to choose which to implement. However for larger data sets and to yield higher accuracy it would have been more favorable to run the classification algorithms multiple times and average out the accuracy results. Once we obtain a mean accuracy measure of each learning model, then we can conclude with higher validity which to implement.

### 5.4.1 Assessing Training and Testing Data

Part of the reason for high classification accuracies could be due to testing data being too similar to the training data and therefore would make the model less able to generalise to instruments inputted exhibiting different sound characteristics. The vocal samples are all sung by the same woman, and although consist of different styles her timbre can only vary so much. This would need to be addressed by having a much higher volume of audio samples consisting of vast variations of recordings and styles.Using an even amount of samples per instrument yielded higher overall accuracies and showed an increase in precisions but in realistic scenarios negative class labels (the instruments that we are not searching for) may outweigh the positive class labels so even data would over fit the model.

When organising the samples into training and testing data, the samples were not randomly selected but rather picked in order of top to bottom. This risked assigning testing data with a certain characteristic not found in the training data. For example the first 30% of the data used as testing, could have been organised from smallest to largest file size. Meaning the machine learning model has only been taught with a specific file size and when given a smaller or bigger than trained with could incorrectly classify the unlabeled audio file.

Audio samples used for training data were all from the same sample pack and exhibited very similar sound characteristics. This could have over fitted the classifiers and made them unable to generalise to a wider variety of test samples. Continuing the point of over fitting, when giving the application audio stems of instruments that contained lots of silence it seemed to perform poorly. This is because the audio samples used in training data contained barely any silence between transients. In real world scenarios mix engineers will be editing audio stems where the instrument may not play for the first 2 minutes of a track, therefore the model would need someway of still classifying the instrument without being skewed by silence. To combat this problem I tried implementing a strip silence algorithm that cut out samples in the audio file that were below a set threshold level. Stripping the audio worked well on vocal tracks and actually classified with higher accuracy, but when performing the same algorithm upon kick snare and electric guitar the system would tend to classify kick and snares as periodic. Therefore I removed it from the application but for

future implementations a more adaptable implementation of a strip silence could be a very useful tool for aiding the machine learning models.

The majority of audio samples used for testing and training were sub ten seconds long. I thought this would restrict the classifier's ability to perform well with longer audio samples, however I found that even longer samples (recordings of bands I had carried out) were still classified correctly for kick and snare yet showed more difficult for electric guitar and vocal usually confusing these last two instruments, so the validity of the model suffers from ill representations of real world training data. Whilst testing the system I discovered that a vocal with shorter breaks between notes and faster attack envelopes would be recognised as a kick or snare drum based upon high onset detection measured. We can easily imagine a musician creating unconventional sound characteristics with their instruments such as a beat boxer, or a guitarist using an eBow. We could train our classifier to generalise past stereotypical sound properties of an instrument but having a system that is 100% accurate at determining the inputted instrument would not actually be the best aid for automatic compression. Instead the compression should be tailored towards the envelope of the sound instead of its instrument label.

### 5.5 Discussion

Evaluating the different approaches employed highlighted certain implications. Firstly, using instrument classification should not be the primary focus of compression calculation since each instrument can exhibit sound characteristics that share similarities with others e.g. strumming a muted guitar can share percussive properties. Lengths could be undertaken to handle such varying characteristics for correct classification but in the example case it would be much better to treat the muted guitar as if it were a percussive instrument when compressing (slow attack and higher ratio).

Therefore the instrument should not be the focus of the compressor's intelligence but rather the full characteristics of the audio file. So transient envelopes should be measured alongside RMS and Mean peak values to identify the speed of a sound's attack, decay and release properties. By taking a more general approach to calculating compressor values based upon the sound properties rather than the instrument, more favorable compression will be realised.

# 6 Conclusion

The main purpose of this project was to build a proof of concept system that solved the problem areas associated with current generation audio plugins. I believe the AI Compressor system has been a successful implementation that achieved established requirements and with future development could be a system commonly used by engineers. Findings from the evaluation showed improved digital signal processing needs to be implemented and approaches to compression calculations should be based on deeper musical information extraction rather than focusing upon the instrument itself. Initial objectives were to create the system within the SuperCollider program, this was exceeded with the implementation of an AU plugin that can send and receive OSC messages and a standalone analysis application that can run as a native Mac application.

Many companies are now investing heavily into autonomous mixing tools, focusing upon real-time parameter adaption to incoming stimulus, which is addressing live musical performance scenarios very well. However, for the mixing domain further development for tools such as the AI Compressor need to be undertaken because richer musical information can be mined from analysing an audio track whole. Non real-time and real-time processing tools now need to be implemented within DAW environments so that engineers do not have to endure the hassle of using an external application. In order for this to be achieved host applications need to accommodate for such systems. This project can be used as a proof of concept of such functionality and with continued support and development could see this future aim realised. Furthermore developers would benefit from a platform that encompasses the development of real-time plugins and their non real-time sisters for accessibility into this new domain.

### 6.2 Proposed Future Developments

Where this project has extended the research by Gainnoulis using machine learning techniques for instrument classification, I would argue against taking an instrument orientated approach for future automated compression development. As outlined above, instruments that may for example conventionally exhibit slow attack characteristics have the potential to also create fast attack characteristics (and vice versa). Compression calculations cannot take this information into account based on the instrument label alone. Therefore a more in depth music information retrieval process needs to be carried out that identifies an instrument's sound envelope within a particular recording instance. Main developments should focus upon creating a system that retrieves means and standard deviations of peak values, RMS, dynamic variability, transient envelope measures and MFCC data. Different machine learning models should then be taught the sonic meanings of 'soft', 'hard', 'medium' compression types by having experienced mix engineers carry out compression processing upon audio samples. The realised system could then analyse any audio data and apply the most suited parameter settings based upon lower level feature extraction rather than higher instrument classification abstractions and thereby allowing the user to cycle through different compression types that will stay suited to the audio.

Implementing OSC listeners inside Audio Unit plugins has created a platform that could be positively exploited by programmers, DSP engineers and live performers. It opens the possibility of controlling plugin parameters from within SuperCollider which could be used as a great tool for composition, mixing or performance. A space that should definitely be explored in upcoming projects.

AI Compressor's scope and ambition was set very high. Although not every aspect implemented achieved all expectations, the education I gained from undertaking this project was invaluable.

# 7 Bibliography

Ableton. *Ableton Live 9.* Available: https://www.ableton.com/. Last accessed 24th April 2014.

Ableton. *Max for Live.* Available: https://www.ableton.com/en/live/max-for-live/. Last accessed 24th April 2014.

Anon. *MFCC 13.* Available: http://i.stack.imgur.com/YUH48.gif. Last accessed 24th April 2014.

Apple. *Audio Unit Programming Guide.* Available: https://developer.apple.com/library/mac/documentation/MusicAudio/Conceptual/AudioUnitProgrammingGuide/Introduction/Introduction.html. Last accessed 24th April 2014.

Apple. *Audio Units.* Available: https://developer.apple.com/softwarelicensing/agreements/audio.php. Last accessed 24th April 2014.

Apple. *Core Audio Frameworks.* Available: https://developer.apple.com/library/ios/documentation/MusicAudio/Conceptual/CoreAudioOverview/CoreAudioFrameworks/CoreAudioFrameworks.html. Last accessed 24th April 2014.

Apple. (2014). *Next Audio Unit Examples (AudioUnit Effect, Generator, Instrument and Offline).* Available: https://developer.apple.com/library/mac/samplecode/sc2195/Introduction/Intro.html. Last accessed 24th April 2014.

Apple. (2014). *Logic Pro.* Available: https://www.apple.com/uk/logic-pro/. Last accessed 24th April 2014.

Avid. *Dynamics III.* Available: http://www.avid.com/US/products/Dynamics-III. Last accessed 24th April 2014.

Avid. *Pro Tools Family.* Available: http://www.avid.com/US/products/family/pro-tools. Last accessed 24th April 2014.

BSC. *BSC Code of Conduct.* Available: http://www.bcs.org/category/6030. Last accessed 24th April 2014.

dathinaios. (2014). *sc_osx_standalone.* Available: https://github.com/dathinaios/sc_osx_standalone. Last accessed 24th April 2014.

Dimitrios Giannoulis, Michael Massberg and Josh Reiss. *UNDER THE HOOD OF A DYNAMIC RANGE COMPRESSOR.* Available: http://www.eecs.qmul.ac.uk/~dimitrios/A%20Tutorial%20on%20dynamic%20range%20compression%20design.pdf. Last accessed 24th April 2014.

Dimitrios Giannoulis . (2010). *A Design of a Digital, Parameter-automated, Dynamic Range Compressor .* Available: http://c4dm.eecs.qmul.ac.uk/audioengineering/compressors/documents/Giannoulis.pdf. Last accessed 24th April 2014.

Dynamic Range. *Dynamic Range.* Available: http://www.dynamicrange.de/. Last accessed 24th April 2014.

fredguile. (2011). *Getting Started With Audio Units on OS X Lion and XCode 4.2.1.* Available: http://sample-hold.com/2011/11/23/getting-started-with-audio-units-on-os-x-lion-and-xcode-4-2-1/. Last accessed 24th April 2014.

GNU. *The GNU General Public License.* Available: https://www.gnu.org/licenses/licenses.html#GPL. Last accessed 24th April 2014.

gruntthepeon. (2013). *Ultra minimalistic OSC library.* Available: http://gruntthepeon.free.fr/oscpkt/. Last accessed 24th April 2014.

Ian shepherd. *Production Advice.* Available: http://productionadvice.co.uk/blog/. Last accessed 24th April 2014.

Jeffrey Walraven . (2013). *Creating a Mac Disk Image Installer.*Available: http://jtechdev.com/2013/04/05/creating-a-mac-disk-image-installer/. Last accessed 24th April 2014.

Jamie Stuart. (2014). *19329-AI-Compressor.* Available: https://github.com/jamesnapierstuart/19329-AI-Compressor. Last accessed 24th April 2014.

lorenzo-stoakes. (2013). *TemplateAU.* Available: https://github.com/lorenzo-stoakes/TemplateAU. Last accessed 24th April 2014.

Mike Senior. (2009). *Compression Made Easy.* Available: http://www.soundonsound.com/sos/sep09/articles/compressionmadeeasy.htm. Last accessed 24th April 2014.

Music Radar. (2014). *Free music samples: download loops, hits and multis.* Available: http://www.musicradar.com/news/tech/free-music-samples-download-loops-hits-and-multis-217833/1. Last accessed 24th April 2014.

Nick Collins. *SCMIR: A SUPERCOLLIDER MUSIC INFORMATION RETRIEVAL LIBRARY.* Available: http://www.sussex.ac.uk/Users/nc81/research/scmir.pdf. Last accessed 24th April 2014.

Nick Collins. *Software.* Available: http://www.sussex.ac.uk/Users/nc81/code.html. Last accessed 24th April 2014.

Open Source Initiative. *The zlib/libpng License (Zlib).* Available: http://opensource.org/licenses/zlib-license.php. Last accessed 24th April 2014.

Open Sound Control. *Open Sound Control.* Available: http://opensoundcontrol.org/. Last accessed 24th April 2014.

padovani-2, Nick Collins-2, Josh-Parmenter. (2012). *SCMIR and SC-3.5-rc3.* Available: http://new-supercollider-mailing-lists-forums-use-these.2681727.n2.nabble.com/SCMIR-and-SC-3-5-rc3-td7342938.html. Last accessed 24th April 2014.

Paul White. (1996). *Compression & Limiting.* Available: http://www.soundonsound.com/sos/1996_articles/apr96/compression.html. Last accessed 24th April 2014.

Propellerhead. *ReWire.* Available: http://www.propellerheads.se/products/reason/?fuseaction=get_article&article=rewire. Last accessed 24th April 2014.

Queen Mary University London: School of Electronic Engineering and Computer Science. Available: http://c4dm.eecs.qmul.ac.uk/index.html. Last accessed 24th April 2014.

Queen Mary University London. *Vamp Audio Analysis Plugins.*Available: http://www.vamp-plugins.org/. Last accessed 24th April 2014.

Roey Izhaki (2008). *Mixing Audio: Concepts, Practices and Tools*. Great Britain: Elsevier. P272.

Ross Bencina. *A simple C++ Open Sound Control (OSC) packet manipulation library.* Available: https://code.google.com/p/oscpack/. Last accessed 24th April 2014.

Sveinbjorn Thordarson. *Platypus.* Available: http://sveinbjorn.org/platypus. Last accessed 24th April 2014.

Somerville, Uitdenbogerd. *Multitimbral Musical Instrument Classification.* Available: http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=4654099. Last accessed 24th April 2014.

Steinberg. *VST.* Available: http://www.steinberg.net/en/products/vst.html. Last accessed 24th April 2014.

Tascam. *Tascam.* Available: http://tascam.com/. Last accessed 24th April 2014.

Tom Duffy. (2009). *[music-dsp] computing compressor automatic makeup gain.* Available: http://music.columbia.edu/pipermail/music-dsp/2009-august/068025.html. Last accessed 24th April 2014.

Universal Audio. *Teletronix® LA-2A Classic Leveling Amplifier.*Available: http://www.uaudio.com/hardware/compressors/la-2a.html. Last accessed 24th April 2014.

Universal Audio. (2012). *UA's Art and Science of Modeling UAD Plug-Ins, Part 2.* Available: http://www.uaudio.com/blog/ask-doctors-ua-modeling-plug-ins/. Last accessed 24th April 2014.

University of Sussex. (2014). *Intellectual Property Rights.* Available: http://www.sussex.ac.uk/staff/research/projects/ip. Last accessed 24th April 2014.

Waves Audio. *Dugan Automixer.* Available: http://www.waves.com/plugins/dugan-automixer. Last accessed 24th April 2014.

Waves Audio. *X-Noise.* Available: http://www.waves.com/plugins/x-noise. Last accessed 24th April 2014.

Waves Audio. *Waves X-Noise Software Audio Processor Users Guide.* Available: http://www.waves.com/1lib/pdf/plugins/x-noise.pdf. Last accessed 24th April 2014.

Waves Audio. *Waves Vocal Rider.* Available: http://www.waves.com/plugins/vocal-rider. Last accessed 24th April 2014.

Waves Audio. *Waves Bass Rider.* Available: http://www.waves.com/plugins/bass-rider. Last accessed 24th April 2014.

Wikimedia. *Compressors Feed Design.* Available: http://upload.wikimedia.org/wikipedia/en/d/d4/Compressors_Feed_Design.gif. Last accessed 24th April 2014.

Wilson, Cottle, Collins (2011). *The SuperCollider Book*. United States of America: MIT Press. P446.

# 8 Appendices

## Appendix: 8.1 Machine Learning Classification Contingency Tables

Nearest Neighbour Binary Classification of Tartini Data:
(Even Data)

| | Predicted Periodic | Predicted NonPeriodic | |
|---|---|---|---|
| Actual Periodic | 209 | 36 | 245 |
| Actual NonPeriodic | 63 | 182 | 245 |
| | 272 | 218 | 490 |

Overall Classifier Accuracy = 80%
Error Rate = 20%
Precision Periodic = 77%
Recall Periodic = 454/517 = 85%
Precision NonPeriodic = 83%
Recall NonPeriodic = 74%

Nearest Neighbour Periodic/NonPeriodic Binary Classification of Onsets Data:
(Even Data)

| | Predicted Periodic | Predicted NonPeriodic | |
|---|---|---|---|
| Actual Periodic | 229 | 16 | 245 |
| Actual NonPeriodic | 28 | 217 | 245 |
| | 257 | 233 | 490 |

Overall Classifier Accuracy = 91%
Error Rate = 9%
Precision Periodic = 89%
Recall Periodic = 93%
Precision NonPeriodic = 93%
Recall NonPeriodic = 86%

K Nearest Neighbour Periodic/NonPeriodic Binary Classification of Onsets Data:
(Even Data, K = 50)

| | Predicted Periodic | Predicted NonPeriodic | |
|---|---|---|---|
| Actual Periodic | 233 | 12 | 245 |
| Actual NonPeriodic | 2 | 243 | 245 |
| | 235 | 255 | 490 |

Overall Classifier Accuracy = 97%
Error Rate = 3%
Precision Periodic = 99%
Recall Periodic = 95%
Precision NonPeriodic = 95%

Recall NonPeriodic = 99%

## K Nearest Neighbour Periodic/NonPeriodic Binary Classification of Tartini Data: (Even Data, K = 50)

| | Predicted Periodic | Predicted NonPeriodic | |
|---|---|---|---|
| Actual Periodic | 191 | 54 | 245 |
| Actual NonPeriodic | 65 | 180 | 245 |
| | 256 | 234 | 490 |

Overall Classifier Accuracy = 76%
Error Rate = 24%
Precision Periodic = 75%
Recall Periodic = 78%
Precision NonPeriodic = 77%
Recall NonPeriodic = 73%

## K Nearest Neighbour Multi Classification of Onset Data: (UnEven Data, K = 50)

| | Predicted Kick | Predicted Snare | Predicted Eguitar | Predicted Vocal | |
|---|---|---|---|---|---|
| Actual Kick | 97 | 7 | 32 | 19 | 155 |
| Actual Snare | 48 | 42 | 0 | 0 | 90 |
| Actual Eguitar | 12 | 1 | 18 | 87 | 118 |
| Actual Vocal | 5 | 1 | 29 | 364 | 399 |
| | 162 | 51 | 79 | 470 | 762 |

Overall Classifier Accuracy = 68%
Error Rate = 32%
Precision Kick = 60%
Recall Kick = 63%
Precision Snare = 82%
Recall Snare = 46%
Precision Electric Guitar = 23%
Recall Electric Guitar = 15%
Precision Vocal = 77%
Recall Vocal = 91%

K Nearest Neighbour Multi Classification of Tartini Data:
(UnEven Data, K = 50)

| | Predicted Kick | Predicted Snare | Predicted Eguitar | Predicted Vocal | |
|---|---|---|---|---|---|
| Actual Kick | 67 | 43 | 32 | 13 | 155 |
| Actual Snare | 6 | 26 | 24 | 34 | 90 |
| Actual Eguitar | 8 | 23 | 58 | 29 | 118 |
| Actual Vocal | 10 | 15 | 31 | 343 | 399 |
| | 91 | 107 | 145 | 419 | 762 |

Overall Classifier Accuracy = 65%
Error Rate = 35%
Precision Kick = 74%
Recall Kick = 43%
Precision Snare = 24%
Recall Snare = 49%
Precision Electric Guitar = 40%
Recall Electric Guitar = 49%
Precision Vocal = 82%
Recall Vocal = 86%

Nearest Neighbour Multi Classification of MFCC 13 Data:
(Even Data)

| | Predicted Kick | Predicted Snare | Predicted Eguitar | Predicted Vocal | |
|---|---|---|---|---|---|
| Actual Kick | 27 | 63 | 0 | 0 | 90 |
| Actual Snare | 41 | 49 | 0 | 0 | 90 |
| Actual Eguitar | 0 | 0 | 0 | 0 | 90 |
| Actual Vocal | 0 | 0 | 84 | 6 | 90 |
| | 68 | 112 | 84 | 6 | 360 |

Overall Classifier Accuracy = 23%
Error Rate = 77%
Precision Kick = 40%
Recall Kick = 30%
Precision Snare = 44%
Recall Snare = 54%
Precision Electric Guitar = 100%
Recall Electric Guitar = 93%
Precision Vocal = 100%
Recall Vocal = 7%

Nearest Neighbour Electric Guitar/Vocal Binary Classification of MFCC 13 Data: (Even Data)

| | Predicted Eguitar | Predicted Vocal | |
|---|---|---|---|
| Actual Eguitar | 90 | 0 | 90 |
| Actual Vocal | 8 | 82 | 90 |
| | 98 | 82 | 180 |

Overall Classifier Accuracy = 96%
Error Rate = 4%
Precision Electric Guitar = 92%
Recall Electric Guitar = 100%
Precision Vocal = 100%
Recall Vocal = 91%

Nearest Neighbour Kick/Snare Binary Classification of MFCC 13 Data: (Even Data)

| | Predicted Kick | Predicted Snare | |
|---|---|---|---|
| Actual Kick | 72 | 18 | 90 |
| Actual Snare | 54 | 36 | 90 |
| | 126 | 54 | 180 |

Overall Classifier Accuracy = 60%
Error Rate = 40%
Precision Kick = 57%
Recall Kick = 80%
Precision Snare = 67%
Recall Snare = 40%

K Nearest Neighbour Kick/Snare Binary Classification of Tartini Data: (Even Data, K = 50)

| | Predicted Kick | Predicted Snare | |
|---|---|---|---|
| Actual Kick | 83 | 7 | 90 |
| Actual Snare | 29 | 61 | 90 |
| | 112 | 68 | 180 |

Overall Classifier Accuracy = 80%
Error Rate = 20%
Precision Kick = 74%
Recall Kick = 92%
Precision Snare = 90%
Recall Snare = 68%

## Appendix: 8.2 Project Log

Autumn Week 1:

Discussed project proposals with supervisor to see which avenues would be best to pursue. The top concepts in mind were as follows:

Intelligent Automated Mixing Software. Concerning the process of mixing multichannel audio in terms of an engineering environment. This process is normally carried out by mix engineers who are given recordings from composers/artists and who's job it is to turn their recordings into polished/completed sounding tracks ready for commercial release. This process includes compression, EQ, reverb and other applications and manipulations of effects processors. The intelligent system aims to automate the very basic/fundamental processes of the mixing stage that are very common and sometimes laborious. To aid the mix engineer by quickly applying the correct amount of processing according to the inputted audio, it will allow the mixing process to be put back in the hands of the composer without the need of a third party and hopefully can become used as an educational system for in experienced users to understand why and how certain processes are applied.
Not to be confused with DJ mixing software between two tracks but with mix engineering.

Autumn Week 2:

Discussed with supervisor the various challenges involved with undertaking this project. Realised early on that a full implementation of a multi-channel intelligent interface is something far beyond the scope of this project.
It would be much wiser to focus upon a single processor such as a compressor and by using machine learning algorithms learn the most suitable processing to apply to incoming audio.

Autumn Week 3:

Found a very intereting paper from one of the students of Reiss, ‘Dimitrios Giannoulis’. http://c4dm.eecs.qmul.ac.uk/audioengineering/compressors/documents/Giannoulis.pdf. This paper is for masters degree and is basically the exact same type of project that I have proposed for my third year project. As a student of Reiss, I need to investigate how he has implemented his proposed thesis to see what I can learn from it and how best to approach my own project based upon the findings I discover from reading his work.

Autumn Week 4:

Discussed the similar research paper by Ginnaoulis with supervisor to see whether this will impact how I tackle my own project. Feedback received was that the main gaol to achieve with this report (as I have outlined) is to learn about how we can use different Music Information Retrieval. Therefore I should not try and change my aims or project because someone else has already attempted a similar feet.

Informed that most ideas people have will sprout up in multiple places around the world at the same time anyway and that it doesn’t matter how novel an idea is as long as I am pushing my knowledge barrier.

Autumn Week 5:

This week I focused upon the design of my compressor GUI.
The GUI implementation is very far away in terms of work to be completed, however, it is necessary to visual how the user is going to use the system in order to suitably pick a machine learning and MIR algorithm. Research a number of compressor plugins especially those from Waves. And explored the compressors I own myself e.g. Pro Tools standard compressor and the Logic Pro standard compressor.

Autumn Week 6:

This week, was spent reading through literature in the topic of machine listening and Fourier transform techniques. “Introduction to Computer Music” by Nick Collins. Chapter 7.2 Music Information Retrieval. Chapter 3.2 Fourier Analysis (Short-time Fourier Transform). After a selective reading of Nick Collins’ pHD “Towards Autonomous Agents for Live Computer Music: Realtime Machine Listening and Interactive Music Systems”, I found the passage upon onset detection most valuable. Therefore Chapter 3.5 Feature extraction of Computer Music, which details onset detection is my next topic of research, to see if I can use it for determining which transients I should detect and analyse their amplitude values. Music Information Retreival researched tested out using SCMIR classes from Collins. Read through Nick’s PHD which has detailed machine learning algorithms.

Autumn Week 7:

Revised machine learning by watching lectures and videos from 2nd year. Focused learning the nearest neighbour and K nearest neighbour models that Bill Keller taught since these were going to be used for my application. Primarily focusing on the Interim report for hand in on the 5th November. This is a great way to combine all the knowledge gathered from the previous weeks works, detailed in the log entries above.

Autumn Week 8:

Implemented my own DFT by following the steps from the DSP Guide [Smith, Discrete Fourier Transform, Chapter 8, http://www.dspguide.com/ch8.htm]. Planning on using to analyse difference between instruments by storing the information from the DFT. However, very computationally heavy so FFT would be the next step to try. It helped me understand sound in a completely mathematical way which was eye-opening and very interesting. Not sure if I will use for the project as there are there Ugens that can take care of this for me.

Autumn Week 9:

Researched into music information retrieval processes. VAMP plugins seem very popular as does the EchoNest API. However these will have to be coded outside SuperCollider. The SCMIR classes from Collins may be more preferable. Researched into creating disk image files to package up and distribute files. Found a simple tutorial that shows how to create one from file and add background images, good to know since this is how most companies release their mac software.

Autumn Week 10:

I really want to implement an audio plugin to be run in a DAW. I tried to download and use an AU library (WDL and JUICE) but these were confusing since lots of files and libraries needed to run. A completely stripped back template to understand the core functionality is needed. So instead I plan to use SuperCollider's Compander class to act as the compressor.

Autumn Week 11:

Getting all the preset data from each of the presets in the pro tools compressor
Spending copious amounts of time obtaining the intended gain reduction upon each preset for the Pro Tools compressor with normalised audio peaking at 0dB. Later realised found that this wasn't needed at all and focused on 'hard and soft' compression types.

Spring Week 1:

First testing the compander class in SuperCollider and although tests worked I felt that a thing I would love to take away from the degree was much deeper knowledge of Digital Signal Processing and how commercial programming audio plugins work therefore I decided to try and tackle the audio unit framework. AU Library and Process: Started off making noise, then a very crude and basic form of a sine wave with plenty of digesting distortion along the way. Eventually deciphering how to loop through incoming samples and first off muting them for 2 seconds then un muting for 2 seconds. Used the AUTemplate tutorial here Guide I used to help get started http://sample-hold.com/2011/11/23/getting-started-with-audio-units-on-os-x-lion-and-xcode-4-2-1/.

Spring Week 2:

Conversion formulas for dB to amplitude and vice versa so people can keep in the same paradigms and I can perform DSP on the sample by sample amplitude -1 to 1 level.
Also understanding how milliseconds work for DSP. Understood how time is coded on sample by sample level, which was very hard to understand for attack and release parameters.

Spring Week 3:

Trying system commands from the audio plugin so I can open the external app from the plugin. However any commands within the process loop cause the computer to crash because it is too CPU heavy calling it to keep opening on a loop of 44100 times per second. Decided to leave out launching app from plugin all together as it can happen when Logic loads up which is not ideal.

Spring Week 4:

Industry Interviews:

Had a great informal skype chat with Tim Exile the renowned live electronic artist. I was starting to wonder if this project had any worth at all and he assured me that is certainly the direction the audio industry is moving in. He was particularly interested in the application intelligent plugins could have for live performance, which is something I would like to pursue after the final realisation of this project. Overall a very inspiring chat.

Also conducted an informal interview with Tim Hart the new music lecturer at Sussex University and experienced mix engineer. He didn't like the idea of presets, stating that they don't take into account the current audio the engineer is working with in a session. This gave more justification for the project. He also suggested some kind of program that could learn the mix engineers technique - rather than signature series plugins but an actual format that engineers take whilst approaching a mix.

Spring Week 5:

Tried OSC implementation with oscpack but had issues running in Xcode. oscpkt was a much better solution.

Brushed up more on compression knowledge from reading Mixing Audio. It goes into nice detail on the Pro Tools Dyn3 which I decided to base my compressor on so this should prove very useful. Learning how to read and write CSV files within supercollider so I could write array values for training data. Found a good site for royalty free sample packs ideal as corpus for ML algorithms and to use CSVs for.

Spring Week 6:

Trying to build a SuperCollider standalone application following the outline on the Github proved very difficult. Cmake was downloaded to try and build it but to no success, there seems to be large errors surrounding building a standalone on 10.8 for SuperCollider 3.6.6 and many people in the mailing lists have provided work arounds but none have worked so far. Possible to go back to 3.5 or 3.4 to use a template but my classes or QT might not work.

Spring Week 7:

Tested the Audio Unit plugin against the Logic standard compressor and the Pro Tool's Dyn3 I modeled my system on. Found the attack and release parameters don't quite match could be because I have used a linear approach and their's could be exponential, but the manuals do not go into this much detail.

Spring Week 8:

Fixed a big problem where the SCMIR classes cannot find the scsynth executable on the standalone, had to overwrite the score method and synthdef path to accomplish this. Finally understood how the MFCC works, and managed to go back through the analysis code and implement it, improving accuracy of the overall instrument classification. Fixed problem where standalone app wasn't reading the CSV file destinations correctly, changed from setting the paths to my local machine to be within the resource directory of the application contents.

Spring Week 9:

Tested the DMG file install on other machines and successfully runs on other 10.8 mountain lion and maverick operating systems. The alias' function as intended as long as the user's system preferences allow applications from unidentified developers. Fixed the GUI which was having size issues on smaller screens, generalized code as fix. Found the AU successfully loads up in Ableton 9 too.

Spring Week 10:

Finishing up the report with extra illustrations, checking through and editing small code comments and making sure all code and final report is accessible from the Github repository on other machines.

## Appendix: 8.3 Running AI Compressor

Download the AI Compressor Contents from the public Github Repository:
https://github.com/jamesnapierstuart/19329-AI-Compressor

Please consult the Read Me for the overview and installation instructions. Below also details the different ways to build and run the application and AU plugin created.

AI Compressor Installation:

1. AI Compressor DMG file: For Standalone Mac Application and Audio Unit Plugin Component. Double-Click the DMG file which will open a new finder window with instructions of dragging and dropping the application and component into the designated folders provided as aliases. This is the most simple and straight forward installation and includes a Read Me for help.

2. AI Compressor AU Component Xcode Project: For Xcode AI compressor Audio Unit Component Xcode files. To build and install the plugin component yourself. Please consult the Read Me that comes with this project of detailed instructions of how to run and place the executed component into the correct plug-in destination.

3. AI Compressor Analysis Application SuperCollider Code: The SuperCollider files, to run the AI Compressor Application within SuperCollider. Copy across the required extensions folder to your library/applicationSupport/SuperCollider/Extensions folder then run the .scd file provided. Again a Read Me file is provided for further instruction.