ELECTRONIC, ELECTRICAL & SYSTEMS ENGINEERING

UNIVERSITY OF BIRMINGHAM

BEng Final Year Project (EE3P)

Name: Daniel Heaton

Student ID: 1016389

Project Title: Detection of Leitmotivs in Wagner's Opera using Neural Networks

Supervisor: Peter Jancovic

Abstract

Neural networks are one of the most powerful and most utilized machine learning models in current use for finding deeply embedded patterns in many types of data, from audio and video to data on medical histories and shopping preferences. Much research has been done in the past 5 years as to the applications of these on intelligently processing audio data, including automatic speech recognition, genre classification for specific songs, and musical event detection in longer pieces of music. This report will be discussing one such example of this adaptation of neural networks, namely the implementation of a complete system for automatic Leitmotiv classification from a source audio file and subsequent detection of Leitmotivs in newly presented audio data. This will be done through creation and training of a convolutional neural network, implemented in the Python programming language and using the TensorFlow machine learning library developed by Google Brain, along with several other open-source audio processing libraries. This is then trained extensively on data extracted from the source .wav file provided to us, having had relevant audio features extracted from it, and later used to automatically detect Leitmotivs from new data using this neural network. The system is then continuously modified with different feature representations, network architectures, and hyperparameter settings to achieve an optimum setup for best achieving the project aims and requirements. By the end of the optimization of the system, we had achieved an overall accuracy of 96.21% over 5390 newly-presented 2-dimensional inputs representing Leitmotiv frame samples, which then help towards predicting 82.48% of Leitmotivs in the original file accurately. The system at this point can also be adapted fairly easily to detect other audio event class types such as musical instruments in a rock song or different voices in a piece of choir music.

Table of Contents

List of Tables, Graphs, and Figures

Glossary of Acronyms Used and Other Commonly Used Terms

Acronyms

- **ANN:** artificial neural network
- **CNN:** convolutional neural network
- **CQT:** constant-Q transform
- **DFT:** discrete Fourier transform
- **FC:** fully-connected (referring to fully-connected layers; synonymous here with hidden layers in feed forward ANNs)
- **LM:** leitmotiv (or 'leitmotif')
- **MFCC:** Mel-frequency cepstral coefficients
- **SVM:** support vector machine, a machine learning model that trysto minimize the distances between support vectors from a data-dividing line

Commonly Used Terms

- **2D matrix sample:** the 2-dimensional matrix of numbers that go into the CNN as an 'audio image', representing the 0.01s frame vector in the middle of the 2D matrix, which has a corresponding single encoded label (e.g. '0', '1', '2', etc.) representing an LM type (e.g. 'Audio', 'Nibelungen', 'Mime', etc.) that says to the CNN what LM the 2D matrix represents
- **Accuracy:** percentage of correctly classified test inputs (as 2D matrix samples) by the CNN
- **Area under ROC curve:** probability that a classifier will rank a randomly chosen positive instance higher than a randomly chosen negative one
- **Batch:** a group (typically 100-200) of input training samples that are passed through the neural network before its cumulative loss is calculated and backpropagation is used to update the weights
- **Chroma features:** features pertaining to the twelve pitch classes in music (where values in the 12D chroma vector of an audio frame indicate how much that specific pitch appears in the audio frame vector)
- **Dropout layer:** a layer placed before another layer that provides dropout functionality to that layer in the form of 'turning off' neurons of said layer during training to help in data regularization
- **Epoch:** a complete copy of all training data provided to a neural network during the training phase
- **F-measure:** measure of the ability of the system to classify frames as being an LM or not, though not necessarily as the correct specific LM

- **False negative:** an instance when the system falsely predicts a negative
- **False positive:** an instance when the system falsely predicts a positive
- **Input data shape:** the dimensions of the a 2D matrix sample input, measured as ('m' x 'n'), where 'm' is the number of audio frames in the sample and 'n' is how many features each frame contains
- **Kernel:** in the context of convolution, synonymous with 'filter'
- **Layer pair:** a convolution layer followed by a pooling layer (input of pooling layer being the output of the convolution layer)
- **Learning rate:** coefficient attached to the gradient descent equation that determines how fast or slow the model updates its weights, i.e. 'learns'
- **LM frame:** a 0.01s sample of audio containing a LM (or the 'Audio' LM type, i.e. no LM) as a vector of 'd' dimensions, where 'd' = number of features extracted; 'n' of these frames are then stacked on top of each other to create a '2D matrix sample'
- **LM label:** a specific 'name' of a LM encountered in our audio data (e.g. 'Nibelungen')
- **Loss:** a measure of the difference between what a neural network predicts to be an output vector and the real values for each node of the output vector
- **Model:** here used to indicate the built neural network model within 'CNNv3' or previous versions
- **Optimizer learning rate:** (see learning rate)
- **Overfitting:** refers to a neural network displaying excellent results on the training data as it has this memorized while having subpar results on the test data; it hasn't been able to ascertain trends and patterns in the training data, only having memorized them
- **Padding:** 0's introduced on a 2D input to a convolution layer, specifically on its edges and corners, to modify the size of the output of the layer (e.g. 'same' padding means 0's on the edges so the output size of the convolution layer is the same as the input size)
- **Precision:** ratio of times a positive guess was correct to the total number of positive guesses
- **Pseudocode:** a simplification of a piece of code or an algorithm in human-readable notation, which can then be adapted to be used in many programming languages
- **Recall:** ratio of times a positive guess was correct to the total number of actual positive labels
- **ReLU:** rectified linear unit, an activation function used in neural networks where the 'y' value (the output from a neuron) is linear if the 'x' value (the sum of the inputs to a neuron) is greater than 0, but 0 when 'x' is less than 0
- **System:** here used to mean the final version of each Python script used as part of the project to act as a complete solution to the project's tasks
- **System state:** a given setup of hyperparameter and network architecture values associated with a stage of testing (e.g. state 2 corresponds to when we are changing the feature representation while keeping all other variables static)
- **Training step:** a single iteration of passing in one batch and updating of the weights to train the model

- **True negative:** an instance when the system correctly predicts a negative
- **True positive:** an instance when the system correctly predicts a positive
- **Vanishing gradient problem:** the problem seen in deep neural networks where the front hidden layers are slow to train when gradients on the weight-loss curve used in back propagation are small, meaning the weights at the front of the layers train very slowly, if at all
- **Weight:** a value signifying the strength of a connection between two nodes of two parallel layers

1: Introduction

1.1: Motivation for Project

The amount of digital information that we create and accumulate on a daily basis is at an all-time high. Within the next 2 years, our cumulative created data will be expected to have increased from 4.4 zettabytes in 2017 to 44 zettabytes (44 trillion gigabytes) in 2020, with 1.7 megabytes being produced every second per person on the planet (Kumar, 2017). It has now become near-impossible for humans to analyse even small sections of it manually to find patterns, structure, outliers, and so on; hence, the need (and demand by cutting-edge research firms and technology companies) for tools that can do this automatically is currently in great demand. Machine learning, a subset of artificial intelligence, helps to partially solve this problem by giving us techniques and algorithms to have a system 'learn' the patterns and relationships in data, which can then make predictions on newly presented data. Deep learning and the use of neural networks are particularly good at finding patterns in data, which essentially model a human-brain in terms of computational graphs, matrices, sums, etc. Indeed, through the use of neural networks and reinforcement learning, there now exists an AI that is capable of mastering highly complex games (Go, chess, shogi) by simply knowing the rules and through self-play, with possible applications to more general AI problems in the future (Silver et al., 2017). These technologies and models can also be adapted to analyse audio data.

Audio analysis in the world of big data is a key focus presently for many companies. With the right tools, trends can be found in a person's music tastes (giving potential music vendors help to focus their advertising of new music releases) or a tool could be adapted to serve as an artificial intelligence, where the AI is trained to recognize certain audio events and act accordingly. One of the best tools to do this with are CNNs (convolutional neural networks), which are an adaptation of traditional feed-forward ANNs (artificial neural networks) that use convolutional and pooling layers between the input and hidden layers. Indeed, some of the world's leading technology-centric companies use these CNNs for audio data, including Google's 'WaveNet', which uses samples of real human speech sampled at tens of thousands times per second to generate raw audio waveforms to function as more human-sounding text-to-speech output (Oord et al., 2016). Additionally, Spotify have also looked into CNNs for use in music recommendation systems, where they help in recommending new music to users based on their preferences, which gets around the 'cold-start' problem where not enough people have listened to new music to get enough data on it (Dieleman, 2014).

It is hoped here that, in a manner similar to some of the work Google and Spotify have done, we can build a system that classifies variations of a certain type of audio event (Leitmotivs) and can thus detect them when presented with new data. This system could then be adapted to different types of audio event detection in other types of music, e.g. different instrument varieties in rock music, different voices in choir music, etc.

1.2: Background and Leitmotiv Exploration

Before we get to how we create our system to fulfil the main aims of the project, its worth exploring what we mean by a leitmotiv. Leitmotivs (short for “leading motive”), shortened here to 'LMs', can be broadly defined as a short, constantly recurring musical phrase associated with a particular person, place, or idea (Kennedy, 1987). However, it should also be able to be modified in terms of rhythm, harmony, or what instruments are used to play it, while retaining its original 'identity' (Warrack, 1995). This is a key part of why they aren't straightforward for a system to detect and predict: what constitutes one LM type might change as the musical piece progresses while still 'sounding like' the original to at least some degree. Leitmotivs also have several other properties:

- They can represent people, places and things, but also ideas and emotions like hope and fear
- They are subject to interpretation by the listener: what the listener perceives as an emotion or feeling implied by an LM may not be what the composer intended it to be, so we are reliant on one person's beliefs of what LMs are which when described to us (as done here in our source annotation file seen in Appendix III for this project, where the annotator has interpreted the LMs themselves)
- They are usually a short melody, but sometimes can also be tied to a particular chord sequence
- They enable the composer to tell a story without any words: therefore, it's theoretically possible for a system that detects LMs to make its own interpretation of the 'story' based on what LMs it's detected

LMs are most often associated with Richard Wagner, a 19th century German composer chiefly known for his operas (although other composers used them such as Richard Strauss and Claude Debussy in their respective works). In particular, they are most famously associated with his cycle of four operas, *Der Ring des Nibelungen*, which in total consist of over 15 hours of music and are presented as tragedies involving gods, heroes, and mythical creatures as they struggle over a powerful magical ring. Hundreds of LMs occur over throughout the cycle and many, such as those representing important characters like Siegfried and Wotan, may occur in several of the four operas (Grout and Williams, 2003). The opera itself we will be looking in the source .wav is Act 1 of the 'Siegfried' opera of Wagner's *Der Ring des Nibelungen* cycle.

What we therefore have is music that has somewhat mechanical-like processes within them that link to different parts of the opera. The idea with using a CNN is therefore to exploit these relationships in the musical piece by picking up patterns in the music and thus help classify LMs where no previously seen note structure or chord-sequence is present. Neural networks are phenomenally good at picking up patterns and relationships, however deep, and the idea is that, when presented with enough data, our system can learn this information and, when presented with a different piece of Wagner's music, detect LMs on which it has been trained to recognized and classify appropriately. Once this has been done, it's hoped that the system could easily be adapted to music event classification tasks, such as for different 'riffs' in rock music, instruments in other classical pieces, moods/emotions in choir music, etc.

1.3: Project Aims and Objectives

As with every large-scale project, it's worth having a list of aims and objectives to keep the project focused on achieving these while providing a useful metric later on to assess the success and failures of the project. Below, we can see a table of aims and objectives for this complete project:

Aim (what we hope to achieve)	**Objectives (actions taken to achieve aims)**
Experience in building preliminary machine learning models in preparation for the final system	• Learn about (from lectures) audio signal processing, Gaussian mixture models, EM-algorithm, parameter tuning, and artificial neural networks • Build programs that use different signal processing techniques, use GMMs/CNNs to classify male/female voices, and modify these to use difference neural network architectures and hyperparameter settings
Understand the basics of CNNs and how these are built and tuned in TensorFlow	• Learn from lecture and YouTube videos the basics of how CNNs are different to ANNs and how they can be applied to audio data • Read the introductory pages on the TensorFlow documentation website • Read several chapters of 'Python Machine Learning' textbook by Sebastien Raschka to understand how a TensorFlow CNN can be built
Have completely prepared audio data for use as input to a CNN	• Create graphs on the average length of LMs types, top 5 LMs types, etc. • Create smaller .wav files from the source .wav by splitting it using rules • Create .lab files that correspond to each new .wav file • Extract features from .wav files using different feature representations
Have a complete CNN that learns from prepared audio input	• Preprocess data as 2D 'X' data with corresponding 'Y' labels • Train network with all inputs and by deducing loss given predictions to determine how much weights should be updated by • Evaluate accuracy based on how accurately the system predicts LM type
Obtain optimized system hyperparameters, network architectures, and feature representations to find best network settings	• Experiment with different values for the CNN's settings, including numbers of layers, size of layers, amount of training steps, size of input data, learning rate, etc. • Test each setting and average over 10 runs of the system to obtain accuracy mean, accuracy variance, run time mean, run time variance, and F-measure
Accurate predictive ability so the tuned completed system can detect complete Leitmotivs in new data	• Build function that creates an output .csv containing predictions of what LM label individual LM frames (of time length 0.01s) could be, along with true classes and location of said LM frame in source file • Function should also create a .csv that shows the locations (start and finish time) of predicted complete LMs as well as the predicted labels

(Table 1 – Aims and Objectives)

2: System Design and Methodology

2.1: Choice of Language and Programming Environment

As with most software-related projects, one of the primary choices that must be made is what programming language to implement the components of the system in, along with what development environment it is to be built in. Both of these have a large impact in the time and ease it will take to develop the system, as well as how optimal it will be running in its final variation. For the choice of programming language, we chose to use **Python (3.6)** to build all scripts from for the following reasons:

- It takes very few lines to implement many things when compared with other languages like Java; for example, the code below shows how a neural network can be implemented in Python in only 9 lines:

```
num_extra_hidden_layers = 3
num_hidden_nodes = 30
ann = Sequential()

ann.add(Dense(units= num_hidden_nodes, activation= 'sigmoid', input_dim= num_dimensions))
for i in range(num_extra_hidden_layers):
    ann.add(Dense(units = num_hidden_nodes, activation= 'sigmoid'))
ann.add(Dense(units= 1, activation= 'sigmoid'))

ann.compile(optimizer= 'adam', loss = 'binary_crossentropy', metrics= ['accuracy'])
ann.fit(x= x_train, y= y_train, batch_size= 10, nb_epoch= 30)
```

(Figure 1 – Keras code example)

- The open-source nature of Python's community means there are very often modules that others have built that fit the profile of what we need, so we don't need to 'reinvent the wheel' by creating our own version of it; this can be seen below in the function to calculate MFCCs from an external library

```
data = np.asarray([float(datum) for datum in data.flatten()[0::2]])
features = chroma_cqt(y=data, sr=rate, n_chroma=reduced_dim).T
' audio_processing_choice == "mfcc":
from python_speech_features import mfcc
features = mfcc(signal=data, samplerate=rate, winlen=frame_time_len, winstep=frame_time_len,
                numcep=reduced_dim, nfilt=reduced_dim*2, nfft= frame_len)
```

(Figure 2 – python_speech_features implementation)

- Python has seen extensive use for building and testing machine learning and deep learning models by research and business communities; thus, it is easily the most well-developed with regards to open-source libraries such as TensorFlow, with TensorFlow's Python API being most complete of its various language implementations (TensorFlow.org, 2017)

With regards to where we shall develop the Python programs for this project, we have chosen to use the **PyCharm Community Edition 2016** integrated development environment for the following reasons:

- We've used it before for the previous summer's internship at Arm Holdings as part of a large project for working with JSON data, so prior experience with it makes for a more expedient process
- The layout of PyCharm makes for writing and immediate testing and modifying of code very simple, with debug options showing locations of compilation errors very easily and with clarity
- It's easy to add additional packages via 'Available Packages' in the 'Project Interpreter', which is useful as we need to add many additional libraries, from TensorFlow to simple packages like 'time'

2.2: Justification of Using CNNs

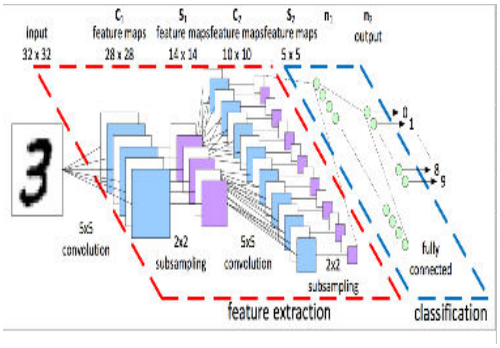
An important thing to clarify at this stage is how CNNs differ from 'standard' ANNs. Moreover, we also need to answer the question: why not just use ANNs instead of adding complexity with using additional layers associated with the convolution process? First, it's preferable to look at what is the same between the two types of neural network (note: an in-depth discussion of neural networks can be found in Appendix XII):

- Both involve training through calculating loss (i.e. the difference between actual values and predicted values) and using an optimization algorithm (e.g. standard gradient descent, RMSprop) to adjust the weights between neurons to help the network make more accurate predictions in the future
- Both have fully-connected hidden layers before the output layer with the numbers of neurons set by the network architect based on the requirements of the task
- The output layers for both types are the size of the number of classes we're interested in (i.e. if we want to classify 6 types of LM, we need a size of 6 neurons for the output layer)
- They can both be adapted to a variety of problems and are not limited to one type of data input

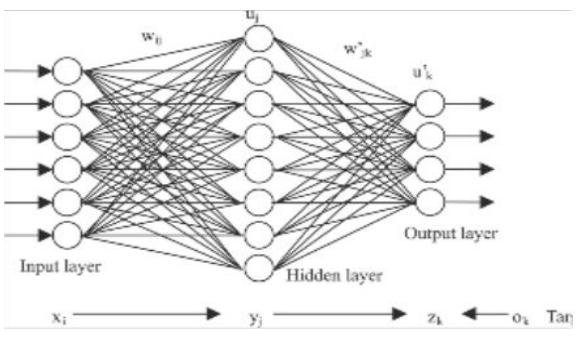
Next, we can see the differences between both model variations, the most notable of which are:

- In CNNs there are a set number of convolution layers before the fully-connected layers, which are very often succeeded by pooling layers, neither of which are in ANNs traditionally
- The input to CNNs are typically 2D matrices containing data of 1 sample (e.g. a 2D matrix of 8-bit RGB values to represent an image where the dimensions of the 2D is the pixel size of the image) that expands to a 3D matrix later on when additional channels are created, while ANNs typically have a vector of numbers as input (i.e. a 1D matrix)
- CNNs are typically adapted for use in image and video data, although use on audio data via transforms on the frequency domain is becoming more popular

We can see examples of the two types below, with the CNN on the left and the ANN on the right



(Figure 4 – CNN image example)



(Figure 5 – ANN image example)

Overview of Convolution and Pooling Processes

The primary function of the convolution layers is to take 'samples' of the 2D matrix by moving a 2D 'kernel' window (smaller than the matrix itself) with values set automatically by the system across the input 2D matrix and finding the average value within the overlap of the two matrices. This window is then shifted across the input 2D matrix until it reaches the end, when it moves down. This repeats until it has sampled over all parts of the input. The result of this is that we capture the most significant features of the input while removing noise, whilst also accounting for rotation, shifting, and transpositional variance (Rohrer, B., 2016).

The output of these layers is then input into pooling layers also as 2D inputs. We again have a set 2D pooling window with a given stride, where the window takes the maximum value found when placing the pooling window on the input. The result is a massive reduction in size while preserving the most important features, i.e. the maximum values preserved by the window in 'max-pooling' (Raschka and Mirjalili, 2017).

An essential part of CNNs, however, is the ability for these convolutional-pooling layer 'pairs' to be stacked on top of each other (output for one pool layer is the input of another convolution layer), and so on until the final pool layer is connected to the first FC hidden layer. This further reduces the dimensions of the input layer's input while further highlighting the most important features. This is particularly useful in high-dimensioned inputs; for example, an HD image of 1920x1080 pixels has over 2 million values, so reducing this through many convolution and pooling layers will make it manageable for the FC layers to process.

How do CNNs Relate to Audio Data?

So far, we have only looked at how CNNs are applied to images, which again is its most common use, but as we are detecting LMs in music, we are far more interested in how it works for audio data. The key here is visualizing the audio data as a frequency image. For example, if we have 30 frequencies for each 0.01s audio sample for an 0.40s time window, these frequency vectors can be stacked on top of each other to produce a 40x30 matrix of frequencies. But one of the essentials to this process works is an assumed small spacial invariance between values: for an image input, this means that neighbouring matrix values usually have similar values except for places of interest where the sudden change between values is preserved.

To a certain degree, neighbouring frequency values for a given audio frame don't change that much temporally with enough measured frequency dimensions unless there is a sudden shift to a particularly notable frequency, so the 'image' principle applies horizontally. Additionally, neighbouring audio frames don't have a sudden shift in frequency if sampled at a high-enough rate (0.01s is deemed to be sufficiently high), so the audio image principle also applies vertically. Thus, placing our audio data for a given time sample (0.40s in the above case) as a stacked series of frequency vectors for each audio frame means they can be interpreted as images by the CNN; the process is then identical as for images following this.

2.3: LeitmotivStats.py and LMStatsExtraGraphs.py

Motivation and Functional Purpose

Note that a complete system decomposition for all subsequent Python scripts described below can be found in Appendix IV. The first Python scripts we created as part of the individual portion of the project were several scripts showing graphs on basic statistics of LMs in the source .wav file, as given by the annotation file. In doing this, we hoped to determine which are the most common LMs within the source file and how much more common they are than others; from this, we then decided which LM types have enough examples to train a neural network on (i.e. only the top 5). Additionally, we also find out the frequencies of LMs for various time durations, which later will be impacting on the length of the input 2D matrices to the CNN along the temporal axis. So although these scripts aren't an actual part of the system in that they aren't required to transform data in any way, nor are they a dependency of any other scripts in the system, these do help us get valuable information about the LMs we will be training on and so will help with the system design later on.

Results and Interpretation

The full code for these two programs can be found in Appendix II, along with an explanation of how they run, and the graphs which are produced from these programs and that are referenced to below can also be found in Appendix V. From these graphs, we can make several observations about the LMs in the source file and which are described by the annotation file:

- The three most common LM types (Nibelungen, Mime, Jugendkraft) make up more than half (56.05%) of the total LMs, and adding the next top 2 most common (Grubel and Sword) gets us to over 67%; therefore, if we chose to only classify only these top 5 most common LMs in our CNN, we keep the majority of our raw data samples (over 2/3's) while not concerning ourselves with LMs that would be hard to classify due to lack of examples for the CNN to learn from
- We can see from the second graph that the majority of LMs are less than 0.5s in length, with the vast majority being less than 2s; hence a small temporal window as input into the CNN (e.g. 40 samples of 0.01s to make a 0.40s window) is preferable to capture the temporal characteristics of LMs
- Although we don't make direct use of salience rating later in the project, it's still useful to see that the 'quality' of LM increases as it gets longer, which is unsurprising as its easier to determine what event, person, feeling, etc. that Wagner was intending to portray with a longer musical sample
- The rest of the graphs show each of the top 5 most frequent LM labels and their distribution of durations, where we can indeed see that most, if not all, of these top 5 types durations are less than 2 seconds in length, though testing with different temporal window sizes needs to be done

2.4: LeitmotivExtractorV2.py

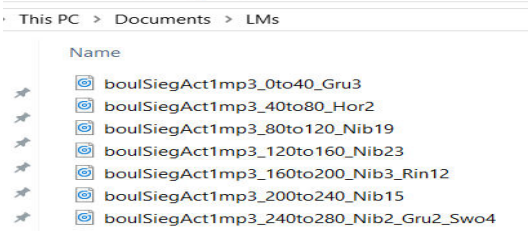
Motivation and Functional Purpose

The first script in the data flow visualization of our system is the LM extractor script. The purpose of this is to split up the source .wav file, which is approximately 1 hour 15 mins long, into segments of roughly 40 seconds long. Its worth noting that LMs only appear in the source file up to approximately 49 mins into the source, and thus we only take data samples up until this point. Hence, we start with 1 source .wav file of 1 hour 15 mins and end up with 72 source files each of approximately 40 seconds long.

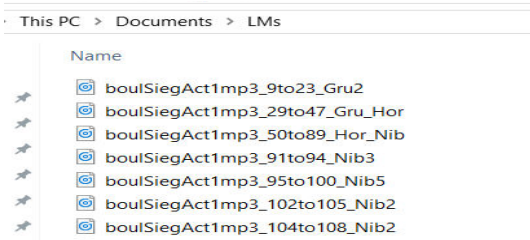
The reason we do this is to split up the raw input data into manageable 'chunks', so subsequent steps don't have to load in a single huge file (~3/4 GB) into memory to work with the audio data; instead, it can work with one small file at a time, which ends up being easier to debug in case anything goes wrong. Also, as we will be making label files at the next stage, having numerous smaller .wav files means we can create many smaller files that are easier to read from a human perspective, rather than a single label file with 1000s of entries. However, we don't want to divide these files along hard-and-fast 40s points; if this is done, LMs might be fragmented between files, or we might not have enough LMs in a particular file for it to be useful, and so on. Hence, we split the .wav up based on a set of rules that are covered in Appendix II of this report.

Results and Interpretation

The full code for how this program works can be found in Appendix II, along with an explanation of how it runs. Although we cannot show the visual effect of picking out the utterances containing the LMs, we can see below the smaller .wavs have been created, placed in the proper directory, and named in accordance to how we expect: 'boulSiegAct1mp3_', followed by the start and end times of the .wav in question in regards to the time period it represents from the source file, followed by the shortened names of the LM labels that occur in the .wav, along with the number of occurrences; this naming preserves the ordering of the LMs in said .wav. We can see below the files created by the new version that we currently use on the left ('LeitmotivExtractorV2.py') with the old version on the right ('LeitmotivExtractor.py'):



(Figure 6 – Files created in version 2)



(Figure 7 – Files created in version 1)

With the source .wav now in a more manageable format of being a sequence of smaller files, we are now ready to create label files for each of these .wavs that describe their LM content, which will be important when we come to the feature extraction phase using different feature representations.

2.5: LabelCreatorV2.py

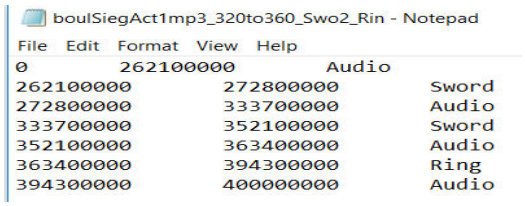
Motivation and Functional Purpose

For the next stage in the system's data pipeline, we must generate label files for each of the newly created .wav files. These must have the same name as each of the corresponding .wav files but with the .lab extension rather than the .wav extension. The idea is that these .lab files contain a 'description' of each of the .wav files; in other words, what LM times they contain and where they occur in each file.

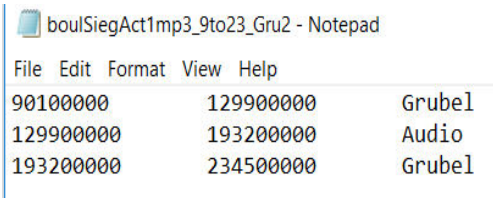
These will play a fundamental role at the next stage where we create .csv files of extracted feature vectors, which will also contain a cell at the end with the label type for that feature for a specific audio frame that's determined by the relevant .lab file. And within these .lab files, each row should equate to a LM within its corresponding .wav file and within the time parameters set by the row; hence, each row should contain the following: start time of the LM, followed by a tab, end time of LM, a tab, and the LM name.

Results and Interpretation

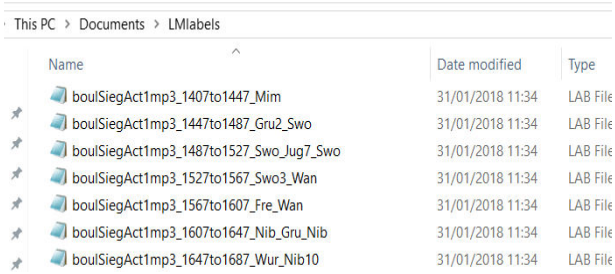
The full code for how this program works can be found in Appendix II, along with an explanation of how it runs. Below, we can see how the .lab files look when opened. On the left is the new version of the .lab files, including starting at 0 and with lines showing audio gaps at the beginning and end of the file, along with in between lines; on the right is the old version with neither of these added features. The next two images show how the '.lab' files have been created in the target location and how they were named, with the the files from 'LabelCreatorV2.py' on the left and those from 'LabelCreator.py' on the right.



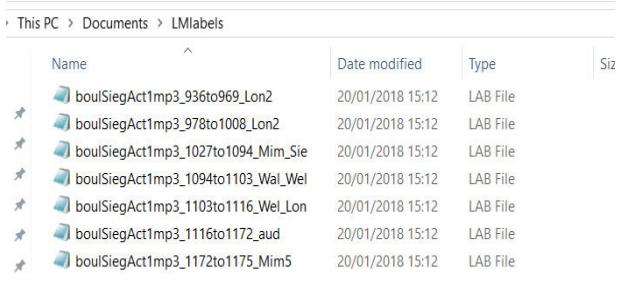
(Figure 8 – Contents of .lab file from version 2)



(Figure 9 – Contents of .lab file from version 1)



(Figure 10 – Files created in version 2)



(Figure 11 – Files created in version 1)

With all of the .wav and .lab files having been created, we now have the audio data in a form in which it can have its relevant audio features extracted from it via one of several feature extraction functions for different feature representations, which will be explored in the next section of this report.

2.6: FeatureExtractionV4.py

Motivation and Functional Purpose

Having created all .wav and .lab files needed in our system, it's now necessary to transform this data in some way so as to represent its most important features. This will be done by transforming the raw audio data onto the frequency domain, with each time sample producing a set number of dimensions (i.e. features) and that essentially represents the most important parts of each sample in frequency form. If we instead decided to proceed to the next section with raw audio files, not only would the data not be 'wide' enough in dimensions to best make use of the convolution and pooling layers, but features between audio frames would not be captured. Therefore, preparing the data in this way ideally sets it up for being presented into the CNN as a 'frequency-image' where principles of spacial and rotational invariance apply.

With a set number of dimensions (i.e. frequency bands) to extract and a set frame length, we produce a matrix of size 'c' x 'd', where c = (length of file in seconds / frame length) * 2 (to account for a 50% overlap between frequency band windows) and d = number of dimensions. For example, for a .wav file containing 40s of audio data, with setting dimensions to 30, frame length of 0.02 (so each row contains 0.01s audio sample), we create a matrix of size 4000 x 30, where each number is the magnitude of a given frequency for a specific 0.01s time window. This will then be concatenated horizontally with a label vector of size 4000 x 1 (determined by the .lab files) to produce the data needed for the CNN at the next phase of the project.

Results and Interpretation

The full code for how this program works can be found in Appendix II, along with an explanation of how it runs. The data below is a short section of the 4000 x 41 data for an utterance file of 40 second duration. The data is now seen as interpretable by a CNN.

	A	B	C	D	E
1	5.129497	1.851218	3.091377	3.653115	0.928806
2	5.075525	-0.07458	2.360801	3.71361	2.687514
3	5.009238	2.537766	1.262928	3.253702	0.388988
4	5.043457	2.00086	0.166171	2.49008	1.437141
5	5.068034	1.90023	2.903401	2.395836	2.339973
6	5.013141	2.53021	2.593396	3.627985	2.913983
7	4.977559	3.509801	2.870756	3.743689	3.131888
8	5.045119	3.752756	1.454099	3.618433	3.512024
9	5.0974	3.266144	1.455414	3.860267	1.797904
10	5.066564	1.872268	2.607811	3.59613	2.055662
11	5.10096	2.610603	2.456606	0.959635	2.147878
12	5.082864	3.222833	1.419275	2.271775	2.301837
13	5.067011	2.705977	1.306586	3.110937	2.195667
14	5.135257	3.339703	2.016017	3.618459	3.169548
15	5.106412	0.5751	3.54058	3.756604	2.75761
16	5.09388	1.635471	3.394182	3.582338	2.732288
17	5.067402	2.484609	2.486191	2.97846	2.669307
18	5.006145	-0.84256	1.712845	-1.58763	3.330659
19	5.054464	2.194653	2.002559	2.927322	2.45882
20	4.984284	2.240777	2.489159	3.224952	2.841816
21	4.947522	2.5149	2.271972	2.00149	2.320065
22	4.941988	2.292474	2.166901	1.385195	1.839939
23	4.905207	2.274797	2.001672	2.831428	0.62485
24	5.002572	2.427908	1.337086	3.283975	2.045176

(Figure 12 – Feature data in .csv, part 1)

	HN	HO	HP	HQ	HR
1	-1.01489	-0.95819	-1.45505	-0.55446	Audio
2	-1.16835	-0.86061	-1.08642	-1.02399	Audio
3	0.096177	-0.77205	-1.50602	-1.49896	Audio
4	-0.97846	-1.1958	-0.23677	-0.59794	Audio
5	-0.70809	-1.16528	-0.25296	-3.86734	Audio
6	-0.53453	-5.7425	-1.53766	-1.85642	Audio
7	-0.86619	-0.87182	-1.02108	-0.29095	Audio
8	-0.53875	-1.18232	-2.20937	-0.35027	Audio
9	-0.03922	0.267536	-2.11296	-1.61832	Audio
10	-0.66263	-0.36483	-0.55161	-1.02176	Audio
11	-1.0329	-1.52129	-3.49562	-0.587	Audio
12	-0.58679	-1.12494	-1.81456	-0.51021	Audio
13	-0.81931	-0.26635	-1.00606	-1.71119	Audio
14	-1.60353	-0.42642	-0.16824	-0.26305	Audio
15	-2.99267	-1.49382	-1.57606	-0.55899	Audio
16	-1.75755	-2.66093	-2.31837	-1.37241	Audio
17	-0.75414	-0.99524	-0.49115	-0.63622	Audio
18	-0.54689	-1.24786	-1.31883	-1.32424	Audio
19	-1.44636	-1.67793	-1.27602	-0.29973	Audio
20	-2.1244	-0.10885	-4.89725	-0.36507	Audio
21	0.02511	-0.38199	-0.98059	-2.57393	Hort
22	-1.00078	-1.1194	-0.99496	-0.65657	Hort
23	-0.5371	-1.17579	-1.6086	-1.38429	Hort
24	-1.83784	-1.34017	-1.81402	-2.25022	Hort

(Figure 13 – Feature data in .csv, part 2)

From here, we will be creating smaller matrices to be interpreted as pixel arrays (like an image) for a specified time period with an associated LM classification. This, however, will be done by the data preprocessing stage within the 'CNNv3.py' script, the mechanics of which shall be discussed later.

2.7: Use of TensorFlow to Implement Neural Networks

For programming in Python, there are numerous options for which library we can use to implement our CNN. For the semester 1 tasks of this project (shown in Appendix I), we used the Keras library, as it was easy to use and we could create, compile, train and test our neural network in less than 10 lines. However, for our CNN here we use TensorFlow. There are several reasons why we switched libraries:

- More extensive and highly detailed documentation and examples for TensorFlow
- Higher amount of direct control over the CNN with things such as weights and optimizers
- Better performance with TensorFlow through threading and queues to speed up the training process
- Availability of the TensorBoard visualization tool to help understand our CNN

Thus, using TensorFlow will hopefully lead to a more successful CNN that can better predict LMs. However, before implementing this CNN we needed to find out how to actually use the TensorFlow framework. This is primarily done by reading Chapter 14 of 'Python Machine Learning' where we learned:

- The benefits of using TensorFlow for neural network training performance in utilizing GPU cores, with using a high-end GPU resulting in ~15 times more floating-point calculations per second
- Concepts of graphs, sessions, ranks, tensors and operations, which gave clarity to the concepts of the computational graph structure used by TensorFlow
- The 'placeholder' concept of the requirement in TensorFlow of a variable to be an 'empty' variable that expects data input (in our case, these 'placeholders' will be implemented for 'X_train' / 'X_test')

With this obtained knowledge from the above textbook, along with examination of other examples found primarily on GitHub or the TensorFlow documentation website, we felt confident enough in our basis of the TensorFlow library that, coupled with prepared input data, our CNN could now be created.

2.8: Building of Requisite Knowledge for CNNv3

At this point, we were able to create an initial version of our CNN to classify audio frames as being part of a LM or not; this initial version is not ideal though, with low accuracies (<50%) and with a massive variance (12-58% accuracy) when being trained on only a few LM types (as the output of 'CNNv1.py'). Hence, there are still many improvements that we need to make before the system can be considered 'complete'. Here, we shall outline how we 'built' our knowledge about things related to CNNs and thus set up to be in a better state when we come to build the permanent model for our system. We thus need two things:

- A complete list of hyperparameters in our existing model to tune (i.e. modify with the hope of increasing accuracy or reducing time to train)
- Research papers' views on tuning CNNs for audio analysis, with a focus on parameter tuning

It's necessary to extensively tune and modify our CNN model by tuning its hyperparameters and modifying the network architecture to achieve better performance. This way, we are able to organize hyperparameters by groups and make the whole optimizing process much more organized. We thus have the list of the parameters in Appendix IX, obtained from examination of our programs.

Analysis of Research Papers Concerning CNNs for Audio

With 28 hyperparameters that we can tune, we have billions of parameter combinations that we can experiment with. Hence, it would most likely be a better approach to first narrow down the list of parameters we should be tuning by finding out set optimal values for others. Thus, we make use of previous research papers that show a completed CNN being used on audio data, which are discussed below:

'Audio spectrogram representations for processing with Convolutional Neural Networks' (Wyse, 2017)

- The paper focuses more on different audio feature representations for CNNs including as raw data, magnitude spectre and spectrograms; this all reinforced our belief of using the DFT to get spectrums
- The limitations of using an 'audio image' as input to a CNN is also highlighted here; possible solutions include the logging of frequencies and using numerous channels

'Sample-level Deep Convolutional Neural Networks for Music Auto-tagging Using Raw Waveforms' (Lee et al., 2017)

- This paper highlighted the use of max pooling as being the best type of sub-sampling for audio, along with a suggestion to have a very small temporal length to be inputted into the FC layers
- It suggests that >4 convolution/pooling layer pairs are optimal for accuracy, with large stride lengths

'End-to-end learning for music audio tagging at scale' (Pons et al., 2017)

- Settings for their model of music tagging include only several hidden layers, with a dropout of 50% before every FC layer and with stochastic gradient descent at an initial learning rate of 0.001
- Here, varying numbers of filters from each convolution layer (increasing per layer) provides best results

Conclusions from Research Papers

Having read through and extracted useful information from the aforementioned research papers, the following conclusions can be drawn from them regarding how we will be tuning and modifying our CNN:

- Different feature representations need to be experimented with that provide different features
- Necessity to include dropout for regularization purposes is stressed here
- Gradient descent should be the best way for the weights to be updated in the system
- Pooling window of size 2x2 with a stride of 2 reduces inputs to ~25% of their original size, which is big enough to capture important features but also reduces input enough to not need many layers
- Same padding for convolution layers is preferable to keep layer output size the same as its input

2.9: CNNv3.py

Motivation and Functional Purpose

As we now have the data in a form which is ready to be introduced into a neural network, we next create a functioning CNN that classifies input 2D matrix samples as belonging to a specific LM label or not (i.e. 'Audio' label). The data that we create from data preprocessing will be in the form ('number of frames' x 'number of features extracted per frame'+1), with the final column being an LM label (e.g. 'Sword', 'Mime', etc.) corresponding to what class each LM frame belongs. As each row/frame represents a 0.01s interval, we wish to take 'samples' of these rows for every 0.40s. Hence, each sample will have (when number of features = 30) a size of (40x30). As this corresponds to a column of size (40x1) of LM labels, we want to find the 'mid' value in this vector on which to classify the whole sample. This results in an 'X' input to the CNN as size (40x30) with a singular encoded label being from the 'mid' of the label vector.

At this point, the numbers from each row are now considered “temporally related” in each row (as they occur in the audio files within 0.01s of each other) so we can reduce them using convolution and pooling as we would do for an image. This is all done using TensorFlow to train our CNN model, as well as testing it on a smaller section of testing data to evaluate its performance. The data is firstly undersampled to reduce excessive amounts of 'Audio' frames in the data as a result of only classifying on 5 types of LMs; the reasons for doing this are further elaborated upon in Appendix VII. The data is then passed through several convolution and pooling layers for all training samples that are allotted from the total available data. Next, the accuracy and other metrics are found from testing the CNN on the testing partition, including the precision, recall, and F-measures (an explanation of these metrics are also included in Appendix VIII).

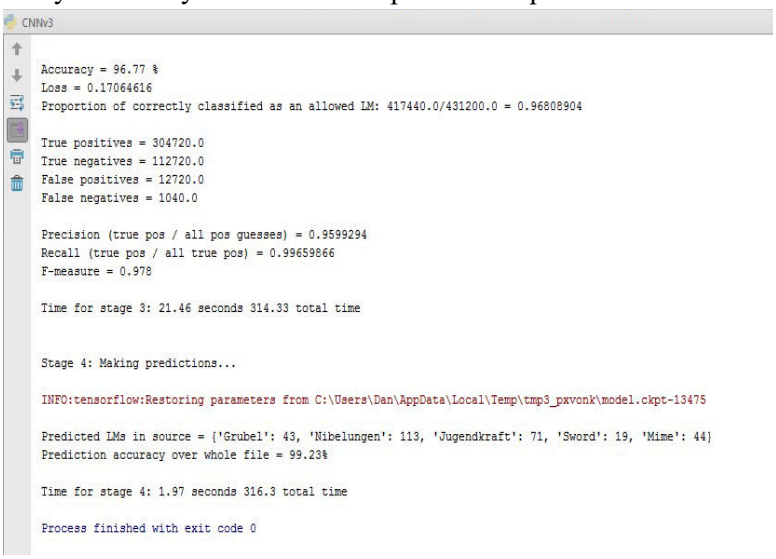
The final part of the system that we needed to create was the ability for it to make predictions based on the data that it is given, after it has been sufficiently trained on the training set. We therefore want this function to do three things (with a complete explanation of the function being found in Appendix II):

- Create a '.csv' file that contains complete LMs that are detected by the CNN Estimator object (the model) and their location in the source file
- Create another .csv file that contains every predicted, decoded audio frame label, start locations in the source .wav, the 'true' label, and whether the prediction is true or false
- Create a dictionary of total predicted complete LMs in the form 'Label':'count' and print to the user

Between these three outputs produced from this function, we then have sufficient information as to the predictive power of the model for it to be useful on newly-presented data, as well as being able to be adapted to work for other audio event types other than LMs. However, as when haven't chosen the best values for various hyperparameters, the network isn't classifying and predicting as well as it could be; this will be addressed in the next section ('Testing and System Optimization') of the report.

Results

The full code for how this program works can be found in Appendix II, along with an explanation of how it runs. The following is the result of running 'CNNv3.py', having run all other scripts beforehand to create the necessary .csv's containing the LM data (note: this image is after all hyperparameter tuning has been done, which shall be elaborated upon in the next section of the report; the accuracy before this tuning was much lower than what is displayed below). This output will be further examined in the 'Results' section of this report, where each output will be looked at in more detail, as it shows the final results of our system's ability to classify LM frames and predict complete LMs.



(Figure 14 – Accuracy output from CNNv3.py)

We can also see below a sample of the predictive ability of the system, with the following two images showing a short sample of the results placed into the respective .csv files, with the 'LM Detections.csv' on the left and the 'LM Frame Predictions.csv' on the right (more complete views of both of these outputs can be seen in Appendices XVI and XVII):

0:2:6:344	0:2:6:622	Nibelungen
0:2:6:684	0:2:6:922	Nibelungen
0:2:6:984	0:2:7:274	Nibelungen
0:2:7:324	0:2:7:554	Nibelungen
0:2:7:624	0:2:7:882	Nibelungen
0:2:40:284	0:2:41:162	Nibelungen
0:2:44:524	0:2:44:526	Sword
0:3:20:104	0:3:21:62	Nibelungen
0:3:21:104	0:3:21:382	Nibelungen
0:3:21:424	0:3:22:2	Nibelungen
0:3:22:44	0:3:23:562	Nibelungen
0:3:23:852	0:3:23:854	Nibelungen
0:3:26:704	0:3:26:982	Nibelungen
0:3:27:24	0:3:27:622	Nibelungen
0:3:27:644	0:3:27:914	Nibelungen
0:4:1:204	0:4:1:206	Sword
0:4:1:344	0:4:1:682	Nibelungen
0:4:1:804	0:4:2:142	Nibelungen
0:4:2:584	0:4:3:342	Grubel
0:4:3:664	0:4:4:602	Grubel
0:4:4:624	0:4:5:198	Sword
0:4:5:200	0:4:5:202	Grubel
0:4:5:204	0:4:5:206	Sword
0:4:5:208	0:4:5:210	Grubel
0:4:5:212	0:4:5:270	Sword
0:4:5:272	0:4:5:326	Grubel
0:4:5:328	0:4:5:330	Sword
0:4:5:332	0:4:5:342	Grubel
0:4:6:464	0:4:6:942	Sword
0:4:40:584	0:4:40:862	Nibelungen
0:4:41:172	0:4:41:238	Sword
0:4:41:244	0:4:41:582	Sword
0:4:41:744	0:4:42:2	Nibelungen
0:4:42:844	0:4:43:682	Grubel
0:4:43:984	0:4:44:2	Grubel
0:4:45:520	0:4:45:714	Grubel
0:5:25:224	0:5:25:442	Sword
0:5:26:664	0:5:27:22	Sword
0:6:0:824	0:6:2:502	Sword
0:6:4:160	0:6:4:162	Nibelungen
0:6:4:968	0:6:4:970	Nibelungen
0:6:40:296	0:6:40:298	Nibelungen
0:6:42:32	0:6:42:34	Nibelungen

(Figure 15 – LM Detections.csv example)

0:19:29:156	Mime	Mime	True
0:19:29:160	Mime	Mime	True
0:19:29:164	Mime	Mime	True
0:19:29:168	Mime	Mime	True
0:19:29:172	Mime	Mime	True
0:19:29:176	Mime	Mime	True
0:19:29:180	Mime	Mime	True
0:19:29:220	Audio	Audio	True
0:19:29:268	Audio	Audio	True
0:19:29:328	Audio	Audio	True
0:19:29:340	Audio	Audio	True
0:19:29:356	Audio	Audio	True
0:19:29:404	Audio	Audio	True
0:19:29:472	Audio	Audio	True
0:19:29:588	Audio	Audio	True
0:19:29:592	Audio	Audio	True
0:19:29:632	Audio	Audio	True
0:19:29:648	Audio	Audio	True
0:19:29:800	Audio	Audio	True
0:19:29:824	Audio	Audio	True
0:19:30:20	Audio	Audio	True
0:19:30:180	Audio	Audio	True
0:19:30:232	Audio	Audio	True
0:19:30:252	Audio	Audio	True
0:19:30:312	Audio	Audio	True
0:19:30:348	Audio	Audio	True
0:19:30:488	Audio	Audio	True
0:19:30:624	Audio	Audio	True
0:19:30:648	Audio	Audio	True
0:19:30:680	Audio	Audio	True
0:19:30:796	Audio	Audio	True
0:19:30:892	Audio	Audio	True
0:19:31:28	Audio	Audio	True
0:19:31:52	Mime	Audio	False
0:19:31:124	Audio	Audio	True

(Figure 16 – LM Frame Predictions.csv example)

3: Testing and System Optimization

3.1: Changing of Platform and Use of GPU

Up until now, we have been developing all Python code in a PyCharm environment run on a Dell laptop with limited memory, processing power, and integrated graphics, which has been an easier and more convenient approach. However, upon optimizing the hyperparameter and network architecture setups to give the network more complexity than its original values, the limits of the laptop became apparent, in particular:

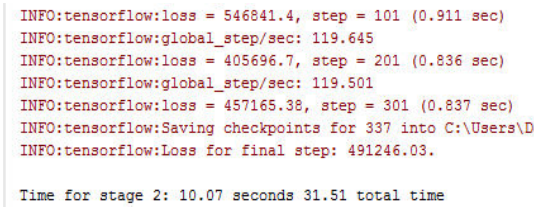
- Laptop would lock up with any more than 5GB of memory used by the CNN at any one point
- Relatively slow CPU meant every 100 steps of training the CNN would take more than 10 seconds
- No dedicated GPU meant we couldn't transfer the training process to a more suitable processing unit

As our CNN model grows larger (with more data samples as input and more training steps and epochs), we need a system that can handle this larger amount of data and takes a fraction of the time to train to enable us to do sufficient tests with different settings in a reasonable time frame. Hence, we decided to transfer the project onto an Alienware R4 desktop with much higher specs, as can be seen in the table below:

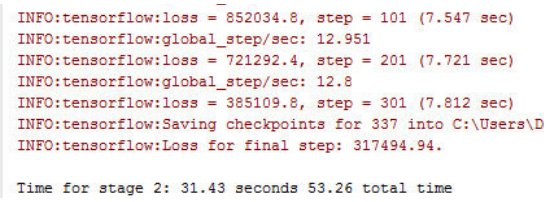
Component	**Laptop**	**Desktop**
CPU	Intel i5-7200U 2.5GHz 4-core	Intel i7-3820 3.6GHz 8-core
RAM	8GB DDR3	20GB DDR3
GPU	Intel HD 620 integrated graphics	GeForce GTX 1060 3GB
Storage	1TB HDD	256 SSD

(Table 2 – Comparison of laptop and desktop)

By transferring the project to the desktop, we can take advantage of the 1280 CUDA cores in the GPU when using TensorFlow (along with 12GB more RAM for holding large amounts of data in memory), thus reducing the time it takes to train the model as training can be done in parallel much more efficiently than if using the CPU. This is due to the training of a neural network requiring many small activities (e.g. computing the output of node from inputs and activation function, and updating of weights) to be done in parallel before moving onto the next step, which the numerous cores of a GPU are ideally suited to do. Once TensorFlow is setup to work with the GPU, we can see a dramatic reduction in time it takes every 100 steps to train in seconds, given a fixed batch size of 128 samples, with using GPU on the left and the GPU on the right (hence we shall use the GPU implementation from this point onwards).



(Figure 17 – Training time on GPU)



(Figure 18 – Training time on CPU)

3.2: Modifications to CNNv3: Feature Representations

Until now, we have primarily used our own implementation of a log filter bank on DFTed signals to extract feature representations. However, as we implemented later versions of the feature extraction script, we replaced our own implementation with a pre-made filter bank from an open-source library, including functions for MFCC and chroma features (which are more reliable than building our own functions). The necessity now is to determine which of them to use to enable the CNN to make the best possible predictions of what class a given audio input belongs to. This is done by two methods: analysis of research papers pertaining to feature representations commonly used in CNNs, and our own experimentation with different representations and how the CNN performs with other fixed hyperparameters.

Analysis of Research Papers Concerning Feature Representations

'Codebook based Audio Feature Representation for Music Information Retrieval' (Vaizman, 2013)

- This paper gives good descriptions of both the CQT and chroma features, and says MFCCs are good for capturing timbre qualities of sound and CQT/chroma are better for harmonic/melodic properties
- The decision in this paper is to use MFCCs as they are assumed to capture the timbre of the music better, and this system is more concerned with general sound similarity over melodic properties

'Classifying Music Audio with Timbral and Chroma Features' (Ellis, D., 2007)

- MFCCs reflect the instruments/arrangements of music; chroma is invariant to instrument changes
- Chroma used to distinguish “cover songs” where melodic content differs, not instrumentation
- Chroma vectors can be associated with a particular chord; use of these in a model helps capture if an artist was fond of a certain subset of chords (applicable to Wagner choosing chords for LMs)

'Unsupervised feature learning for audio classification using convolutional deep belief networks (Lee et al., 2009)

- 'Raw' spectrogram gives an accuracy similar to using MFCCs for music genre classification, with a ~3% increase by using MFCCs; this is enough for us to try a non-MFCC spectral representations
- Outlines the limitations of MFCC and spectral representations of features, as a deep belief network would almost always identify better feature representations within the network itself as 'beliefs'

Conclusions from Research Papers

With CQT and chroma's emphasis on melodic and harmonic feature capturing ability, they are looking to be ideal candidates for our feature representation. Conversely, limitations on the accuracy performance of MFCCs/spectrograms and their tendencies to better capture arrangements and instruments in music also imply that CQT or chroma features might be preferable for our system.

Results and Conclusions from Feature Representation Tuning

Having established that chroma or CQT should be preferable based on several research papers, we now display the results of our experimentation of different feature representations of the source audio data. The full results of these experiments can be seen in Appendix X. What we found, though, with the other representations is contrary to what we expected to see. We thought we'd see an improvement with accuracies when using CQT and/or chroma features as these emphasized harmonic and melodic differences, which we thought were key to the nature of LMs. However, these were actually on average 3.97% less accurate than when using log-filter and MFCC to extract features. This could be due to one or more of several reasons:

- LMs vary more by timbral/arrangement qualities than previously thought, less on melodic/harmonic
- Chroma and CQT, in our implementations, need their dimension size scaled up from their naturally low size to fit into our CNN, thus they don't contain as much 'raw' information as MFCC/log-filter

Hence, due to the higher accuracy than MFCC, we will use **log-filter bank** as the feature representation.

3.3: Modifications to CNNv3: Convolution and Pooling Layers

As the convolution process is essential to how CNNs work, we must devote time to finding the optimal setup for our system regarding the numbers of convolution and pooling layer pairs, the sizes of the filters / pooling windows, stride length, padding, and other tunable hyperparameters. This will be done in a manner similar to the previous section on finding the best feature representation for our system. We explore research papers to find information on the optimal convolution/pooling settings for audio analysis. This is then combined with the results of conv/pool experiments to decide on the final system settings.

Analysis of Research Papers Concerning Convolution and Pooling Layers

'ImageNet Classification with Deep Convolutional Neural Networks' (Krizhevsky et al., 2012)

- Uses 5 convolution and 3 FC layers; removing just one resulted in noticeably inferior performance
- Pooling layers only exist here between convolution layers 1 and 2, 2 and 3, and 5 and FC layer 1
- Emphasizes that network depth is important; removing a middle layer resulted in a 2% accuracy loss

'Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis' (Simard et al., 2003)

- States that kernel should have a size with sufficient overlap but not too much as to have redundant computation; e.g. if stride = 2, 3 is too small with 1 unit overlap, 7 is too large with 5 units overlap
- Indicates that, for optimal accuracy, we need to keep the overlap-to-kernel window ratio at ~40-50% with an increasing number of feature maps extracted from each successive convolution layer
- Padding didn't have a noticeable impact on performance

'Automatic Musical Pattern Feature Extraction Using Convolutional Neural Network' (Li et al, 2015)

- Uses 3 convolution layers with the number of feature maps produced from each being 3, 15, and 65
- Observed that a CNN with more than 3 convolution layers is more difficult to train as network convergence will easily get trapped in local minimas
- With regards to convolution and pooling windows sizes larger than 10 and 4, respectively, they are more difficult to train while smaller ones are subjected to capacity limitation

Conclusions from Research Papers

We will experiment with having between 3-6 convolution layers, with keeping a pooling layer after each convolution layer. Also, we will experiment with window sizes of 2-10 with an overlap-to-kernel window ratio of 40-60%, and features maps extracted per layer will increase by factor of 5 per layer.

Results and Conclusions from Convolution and Pooling Layers Tuning

The results of tuning the number of convolution and pooling layers, along with kernel sizes, can be seen in tables in Appendix X. We can see a noticeable increase in the accuracy of the model as we increase the number of layers: approximately 2.6% increase per layer (though were limited to 4 layer pairs due to the size of the 4th pool layer output being (3x1), with 625 channels, and couldn't go any smaller). Unlike with feature representations, this experimentation did perform as expected compared with the research papers we looked at. This is most likely due to the increase in convolution being beneficial to the preservation of important features of the input data, while removing any noise in the signal, and also being able to preserve the important qualities of the data even if the audio image is rotated, transposed, shifted, etc. Therefore, we will use **4 conv/pool layer pairs** for the remainder of the project.

Regarding kernel sizes, it was speculated that our increased number of conv/pool layer pairs (4) did much of the job of sampling along a wide temporal spectrum. In other words, a CNN with 4 layers and small filter sizes should perform similarly to one with 2 layers and increasing sizes. Hence, our theory was that we wouldn't experience any increase in accuracy with an increase in filter sizes as new potential information is already extracted by using 4 layers, and any potential gain captured by increasing the temporal domain size would be equally offset by the requirement to add more padding for widening kernels. This proves to be correct, as the table of kernel tuning results shows no discernible change in accuracy from our current setup even with several changes in kernel size. This reinforces the notion that increasing the filter size has no positive effect on our system (only longer to train) and thus we remain with **(3x3) kernels** for all layers.

3.4: Modifications to CNNv3: FC Layers

It is at this point when we must look to what we can tune within the FC layers; that is, what should change between the flattened data going into these layers and the output vector coming out (how this CNN

looks can be seen in Appendix XV). The principal two components that we wish to look at here are the FC hidden layers and the number of nodes (neurons) in each of them. The aim therefore is to find a layer number and layer sizes that maximise the accuracy. To this end and as before, we read through several research papers on the topic of neural networks and from these get opinions as to how they believe a neural network set up for audio analysis should set the hidden layers. This knowledge is then combined with the results of hidden layer modifications to choose a final setup for the appropriate hyperparameter values.

Analysis of Research Papers Concerning FC Layer Hyperparameters

'Improving neural networks by preventing co-adaptation of feature detectors' (Hinton et al., 2012)

- Dropout layers are vital to avoid overfitting; here, reduces the errors on the test set from 160 to 130
- Adding to the 50% dropout for FC layers a 20% dropout for the input reduces errors from 130 to 110 and applies to a variety of network architecture setups
- Dropout in all FC layers works better than dropout in only one FC layer and more extreme probabilities tend to be worse, hence 0.5 dropout rate in each layer

'Exploiting Image-trained CNN Architectures for Unconstrained Video Classification' (Zha et al, 2015)

- A deeper CNN architecture yields consistently better performance due to more stacked layers
- The ratio of hidden nodes to output nodes here is approximately 4:1

'Do Deep Nets Really Need to be Deep?' (Ba et al., 2014)

- Deep CNN with 3 hidden layers is shown to outperform a shallow neural network with up to 40x as many hidden units by up to 4.1%
- Shallow-layer network can only match CNN with 3 hidden layers by huge number (~400k) hidden units and by implementing as a 'mimic' neural network
- Given a parameter budget, deeper models are generally preferred over shallower models

Conclusions from Research Papers

For our system, we will implement dropout layers before all FC layers (an explanation of what dropout is can be seen in Appendix XI), starting at a 50% dropout rate and experimenting with this, along with a dropout layer for the input layer of between 20-50%. The number of hidden nodes per layer should be ~x4 the size of the output layer (will experiment with this too), while each FC layer will have the same number of nodes as each other (as per aforementioned research papers). Finally, the models in the papers had at least 2 FC layers (up to 4) so we will experiment with up to 4 as well.

Results and Conclusions from FC Layers Tuning

Referring to the Appendix X table on the results of dropout layer tuning, we see that the accuracy improves as we decrease the dropout rate from 0.5 to 0.1 (after which the variance rises too much to make

it ideal). This is probably due to the network not being at much risk of overtraining so a large dropout rate isn't really necessary. It also could be due to having a large number of neurons in each FC layer so even with a small dropout rate, it turns off enough neurons to force the CNN to learn in different ways; however, too small of a rate decreases accuracy and increases variance, implying that some regularization is needed. Hence, we set the **dropout rate to 0.1** (smaller than research papers recommend). For the input dropout, adding this features shows the accuracy getting progressively worse as this dropout rate increases, resulting in ~15% decrease with 50% input dropout. This could be due to removing temporally similar iterations having a negative consequence on neighbouring frame inputs, along with the general comparable lack of data compared to no input dropout. Hence, we set the system to have **no dropout rate on input layer**.

With regards to the numbers and size of the FC layers themselves, we know that too few nodes means its not able to learn complex enough data patterns, with too many not being able to update fast enough given set number of steps'; hence, a balance must be found in this. This was expected to be ~x4 the size of the output layer, though this isn't the case here, as larger number of nodes are preferable. This is probably due to being more features (both low- and high-level) that must be picked up by the CNN to detect general patterns in the data than previously thought. Hence, we set there to be **400 nodes per FC layer** to maximise the accuracy. Next, we experimented with different numbers of FC layers and, as can be seen in the table in Appendix X, this shows no improvement in accuracy from using 2 FC layers. This is most likely due to at least 1 FC layer not extracting high enough level audio features that the 2nd layer can detect, while more than 2 layers suffering from the vanishing gradient problem. Additionally, more than 2 layers to detect the deepest of features aren't necessary as the convolution/pooling process has already brought out several of the most prominent high level features so a deep FC structure isn't needed. Thus, we keep the CNN set at **2 FC layers**.

3.5: Modifications to CNNv3: Activation/Optimizer Functions

We next look at what activation functions we should be using for the convolution layers, FC layers, and final output layers (e.g. linear, ReLU, hyperbolic tan, etc.). These have a great impact on the values passed on from one node on one layer to another in the next, thus will have a significant impact on the values on the output layer (predictions) and hence the accuracy of the system. We also needed to work out the best optimizer function to be the algorithm we use to train the model. Before experimenting with different values, we found several research papers that gave examples of these hyperparameter settings used in an effort to help justify setting values. After this we experimented with viable options and decided on final values.

Analysis of Research Papers Concerning Activation and Optimizer Functions

'Convolutional Neural Network Architectures for Matching Natural Language Sentences' (Hu, 2014)

- Positive activation function on convolution layers keeps negative artefacts out of pooling layers
- ReLU is used as activation function for both convolution and FC layers here as it yields comparable

or better results compared to sigmoid-like functions, but converges faster

- Stochastic gradient descent was used for training with mini-batches of between 100-200 in size

'Fully Convolutional Networks for Semantic Segmentation' (Long et al., 2015)

- The paper experimented with learning rates for its system of 0.001, 0.0001, 0.00032; lowering this learning rate by factors of ~100 before final settings increased performance
- Non-linear activation function again used for convolution layers, while the loss function used as standard sum over spatial dimensions of final layer

'Network In Network' (Lin et al., 2014)

- Learning rate lowered by scale of 10 until best is found (being $1/100^{th}$ the size of the initial setting)
- Mini batch of size 128 is used here

Conclusions from Research Papers

From researching these papers, we conclude that our activation functions should be non-linear for both convolution and FC layers and also positive for convolution ones (ideally ReLU), though nothing was said of output activation functions so we are going in blind regarding this hyperparameter. Additionally, the optimizer should use stochastic gradient descent with mini batches of 100-200 samples, while learning rate should be experimented with between 1 and 0.0001, decreasing by factor of 10 between settings; however we won't be experimenting with this until the input data size, shape of input, and steps/epochs are set, as we can see the behaviour of using different learning rates when we have enough data and training iterations.

Results and Conclusions from Activation and Optimizer Functions Tuning

Observing the relevant tables in Appendix X showing the results of activation and optimizer function tuning, we can see that ReLU on convolution and FC layers remains the best activation function for getting the highest accuracy (as expected from research papers). For the FC and convolution layers, ReLU was >8% and >5%, respectively, more accurate than the runners up, which unsurprisingly were non-linear functions. This is most likely due to that the output of ReLU being '0' when sum of inputs is <0 helps filter out artefacts going into the pooling layers (reducing noise and improving accuracy), while linear outputs when sum is >0 allows all incoming weights to have maximum impact rather than being scaled back. However, for the output activation, we noticed a marginal increase in accuracy by using linear output, probably due to the inclusion of negative numbers for the output vector having a beneficial effect when it comes to computing the loss function (e.g. more marked difference between '0.5' and '-0.5', compared with '0.5' and '0.0'). Hence, we chose **ReLU activation for convolution and FC layers** and **linear activation for output layer**.

Finding the right optimizer for our system is essential for it to be training quickly while getting optimal results without overshooting and falling into a minima (see Appendix XII for explanation of how this

works). We can next see in the results for optimizer functions that Adam is, unsurprisingly, the best performing optimizer. This is most likely due to its ability to assign individual learning rates to each weight in the network, which continuously updates their learning rates based on a moving average and a decay rate while performing gradient descent, though RMS prop is still comparable due to also having learning rates for each weight. Meanwhile, standard gradient descent and AdaGrad are much worse, probably due to having to share a learning rate between all of their weights and it not being dynamically optimized in runtime. Hence, as no other optimizer is an improvement on our original optimizer we continue to use **Adam as optimizer**.

3.6: Modifications to CNNv3: Input Data Shape, Size, Number of Epochs and Steps, and Learning Rate

The final stage of the tuning of our system is the setting of ideal input data shapes (i.e. dimensions of each 2D data sample into the CNN), the size of the input data (i.e. how many of these samples we are able to generate in preprocessing from a fixed body of data), training steps / epochs, and the learning rate. Once this has been done, we will have reached the final 'state' that the system will remain in for the remainder of the project and for when we implement the predict functionality. The setting of these parameters are informed by several research papers that, to some degree, suggest a range of values for these hyperparameters. This, combined with results from experimentation, will determine the final settings for these hyperparameters.

Analysis of Research Papers Concerning Aforementioned Hyperparameters

'OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks' (Sermanet et al., 2014)

- Starts with learning rate of 0.05 and decreases by half roughly every 10 epochs
- Uses 80 epochs on training set

'Very Deep Convolutional Networks For Large-Scale Image Recognition' (Simonyan et al., 2015)

- Mini batch gradient descent used here with batch size of 256
- Learning rate initially set to 0.01 and decreased by a factor of 10 a total of 3 times
- 74 epochs were used (with corresponded to 370k steps, or iterations)
- Few epochs were needed to converge, most likely due to implicit regularization imposed by the larger network depth, smaller convolution filter sizes, and pre-initialisation of certain layers

'Stochastic Pooling for Regularization of Deep Convolutional Neural Networks' (Zeiler, M.D., 2013)

- Initial model trained on 280 epochs, with only increasing to 500 when number of feature maps produced by the final convolution layer is doubled
- With up to 500 epochs, new techniques were needed to make sure that the network doesn't overfit, even with significant amounts of data

Conclusions from Research Papers

We expected to need at least 70 epochs of our training data to achieve a plateaued loss (and thus the best possible accuracy), with several hundred probably leading to overtraining. We keep the batch size set at 128 samples per batch to work with mini-batch gradient descent, and the total number of training steps should be maximised to the point of a reduction in loss no longer appears with further steps.

Results and Conclusions from Input and Training Behaviour Tuning

As previously stated, the data goes into the CNN as a 2D input sample of size 'm'x'n', where 'm' equals the number of audio frames (each of 0.01s) and 'n' equals the number of features of each frame. Modifying the 'm' and 'n' values have an impact on the shape of the convoluted and pooled data that goes into the FC layers, and thus the accuracy. This can be seen in the table for the results of input data shape tuning in Appendix X, where the best results actually come from the smallest input data shape (40x30), which is somewhat counter-intuitive as this is less data into the system than larger sizes. However, this is probably due to increasing the dimension size ('n') only increases noise into the system and not supplying any more useful information, on top of the fact that lower dimensioned data input into FC layers are easier to train when limited on data steps and epochs. Thus, we decide to remain with feature dimensions of 30 with a window size of 40, which results in an input shape of **40x30**. Additionally, we can see from the table on the input data size (the number of training samples extracted from the source .wav, increased by lowering the window step) that more data into the CNN for training has a massive impact on network performance, as the more data it has the more the weights can tune more precisely and thus classify test data and predict LMs better. In decreasing the window step by 2 per setting, we get 20-33% increase in data samples produced per setting. We can see this causes the accuracy to peak at 68.89% with a 2 step size, though reducing this more to 1 shows a decrease in accuracy, probably due to input samples not being temporally different enough to provide more useful information without overtraining the model); thus, we determine a **step of 2** to be ideal.

As epochs and steps give more data to the network to train on and more time to train on this data, it's expected that an increase in these means the CNN can train more and thus update weights more precisely to achieve a better accuracy (see Appendix XIII for a breakdown of how epochs and steps differ), up to a point (~100 epochs) where the loss function plateaus (this can be seen in the TensorBoard graph of loss to steps later in the report). As expected and can be seen in the relevant table, the increase in epochs results in a higher accuracy, though it stops increasing beyond 10; however, once we increase the steps in tandem to the increase in epochs, the accuracy increases dramatically up to ~95% (we stop at 80 epochs and 16,000 steps as the dramatic increase in time to train does not offset any minor increases that we may get, along with the GPU being at risk of overheating if training for too long as it was running at 98% load and a temperature of 105 C). Thus we set the system to use **80 epochs and 16000 steps**. Finally, we experimented with different learning rates, as seen in the table in the appendix. We can see a slight increase in accuracy by increasing the rate by a factor of 10, with variance shrinking to a third of its former value (as with a higher learning rate, the system can explore more of the weight-loss graph to find the global minima quicker and update weights sooner and more precisely; see Appendix XII), with too high values leading to accuracy fluctuating wildly due to the CNN overshooting on the weight-loss graph, settling into local minimas often. Thus, we choose the value of **'0.01' for the learning rate** as it provides the highest accuracies without fluctuating wildly.

A complete table of final hyperparameter settings for our system can be found in Appendix XIV.

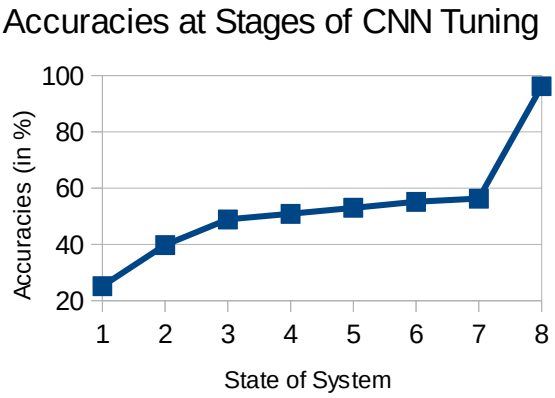
4: Results

4.1: Progress Table and Graph

We are now at a point where the system has been completely built and optimized to the best of our abilities given time, knowledge and hardware constraints, and it thus remains to examine the results of our project and of the system itself. Given below is a table and graph that both describe the general progress made by the system in terms of accuracy. Note that the accuracy given on the y-axis of the graph is an average accuracy of 10 test runs of each system state.

System State	Tuning Aspect	Accuracy Improvement
1	None, default settings	0% → 25.13%
2	Feature representation	25.13% → 39.77%
3	Conv/pool layers	39.77% → 48.85%
4	Shape of input	48.85% → 50.84%
5	Dropout layers	50.84% → 52.98%
6	FC layers	52.98% → 55.11%
7	Activ and optimization functions	55.11% → 56.23%
8	Size of input, epochs, step, learn rate	56.23% → 96.21%

(Table 3 – States, tuning aspects, and accuracies)

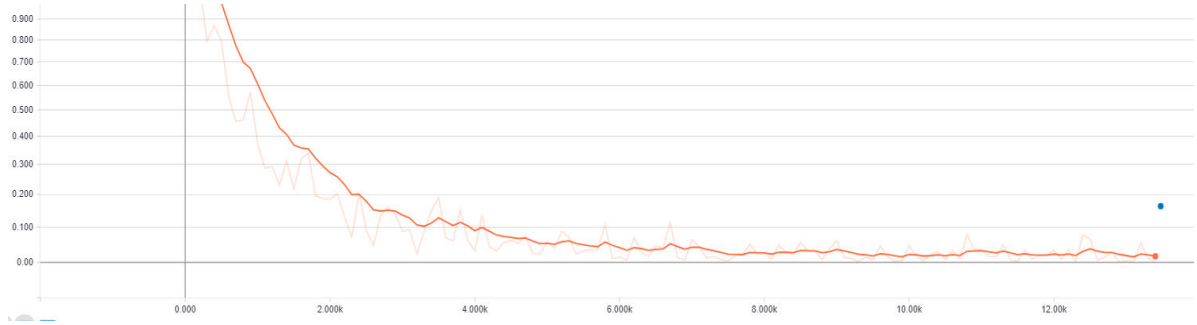


(Graph 1 – Accuracy of CNN at different states)

- Between States 3 and 7, we saw a total accuracy increase of less than 8%, which seems very minor; however, as we were restricted by lack of data and training iterations at this point, such an increase is still significant in the context of limited data samples
- We could have increased the amount of input (by reducing window step) and increased steps/epochs as one of the first tuning aspects, which would have resulted in a more noticeable increase between States 3 and 7 (this was instead done at the end to save on total testing time)

4.2: Implementation of TensorBoard

In an effort to help visualize our model, we also made use of the TensorBoard feature that comes with TensorFlow. This gives us a complete graph model that will aid in the visualization of the information flow in our network. It also helped in debugging and finding network dependencies that had, previously, informed us on when we'd constructed the network inefficiently. Along with the graph model it produces, it also gave access to monitoring tools for training performance and other parameter behaviour. In Appendix XV, we can see a printout from the graph that is a visualization from running our tuned 'CNNv3' model. We can also see below a printout of the loss function with respect to the number of training steps:

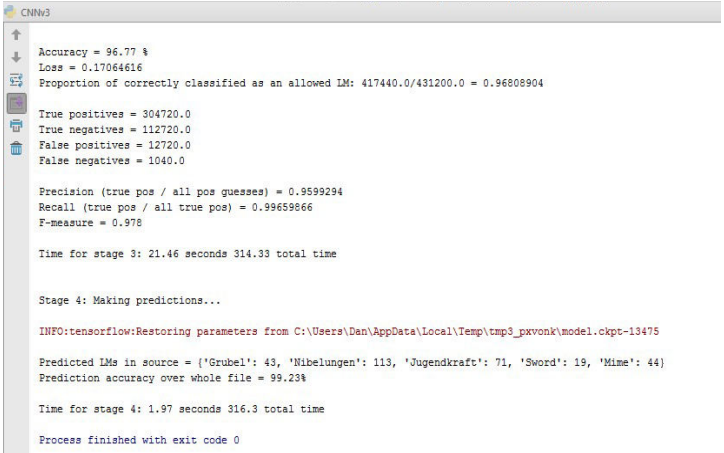


(Graph 2 – TensorBoard loss/steps)

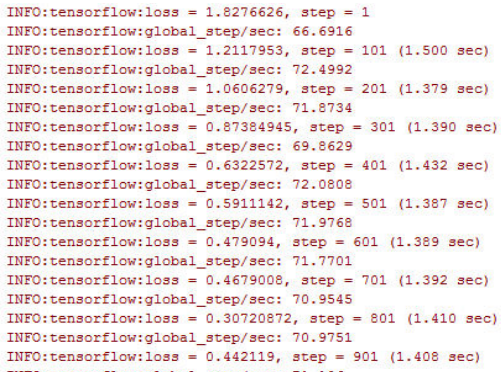
The x-axis shows the number of steps taken during the training process (from 0 to ~13.5k) and the y-axis is the loss for a given training step (ranging from >1 to 0.03). Note that the blue dot at the end (the finishing testing loss) is much higher than the orange dot (the finishing training loss) as it's representing testing data, which the model has never seen before and thus is more likely to have at least a somewhat higher loss on. Finally, the light line represents the true values of the loss function, while the bold line is a 'smoothed' version of this that shows a steadier descent with more training steps. From this, we can see that the majority of the learning happened during the first 4k, and the following 9k steps don't reduce it by that much. Additionally, we can see a noticeable gap between the loss of the training and testing data, which may point to overtraining on the training data (more regularization techniques other than dropout may fix this).

4.3: Output of IDE

What we can see in the next section is a screenshot of the various outputs produced on the console window of PyCharm (the IDE for Python we use throughout the project) when the 'CNNv3.py' is run. This is the final script in the complete system pipeline and thus is the point where we can assess the ability of the system to classify LM frames and predict complete LMs. We also could have shown the console printouts of other programs (e.g. 'LeitmotivExtractorV2.py' and 'LabelCreatorV2.py') but as these were mainly printouts of progress made during the programs' runtimes, it was felt that it wasn't really necessary to display here:



(Figure 19 – Example output from run of CNNv3.py)



(Figure 20 – Progression of loss over time)

- The accuracy of the model on the testing partition of the data is given as 96.77% (i.e. 96.77% of the time, the model correctly classifies a 2D audio image from the test set with the correct LM label)

- The final loss is printed out which is seen in the diagram above getting smaller during training
- The next line gives proportion of frames classified as a LM (any LM) or not (i.e. 'Audio')
- The 'supplementary' measures are printed on the next four lines, followed by the precision and recall calculations (equations for each given in the output) and the F-measure (explained in Appendix VIII)
- The total complete LMs that were predicted by the system (via analysing successive predicted LM frames) on the complete corpus of audio data is displayed as a dictionary
- The accuracy of frame predictions (as seen in 'LM Frame Predictions.csv' output file) is outputted as 99.23% accuracy (i.e. percentage of frames that are predicted the same as their true values)

4.4: Predicting Complete LMs

We can see in the table below the LMs that the system were trained on, how many it predicted were in the original source .wav file (shown by the above dictionary on the console output) and how many were actually in there (this can be seen in the graphs outputted from 'LMStatsExtraGraphs.py' earlier on):

LM Label	Number of Pred LMs in Source	Number of True LMs in Source	Analysis
Nibelungen	113	117	Underpredict: **3.4%** Nibelungens missed
Mime	44	92	Underpredict: **52.2%** Mimes missed
Jugendkraft	71	54	Overpredict: **31.5%** more Jugendkrafts
Grubel	43	29	Overpredict: **48.3%** more Grubels
Sword	19	22	Underpredict: **13.6%** Swords missed

(Table 4 – LM predictions and true labels)

As over 99% of the individual frames were classified accurately, there must be several other reasons why there is some variance in complete LM predictions versus the true number of LMs that include:

- Overestimating the number of Jugendkrafts and Grubels are most likely due to frames in the middle of the LM being incorrectly classified as 'Audio', as in our system it only takes 1 'Audio' frame between several other LM frame types to make the system classify both sides as separate LMs (possible fix could be to make 'predict_lm' function require several 'Audio' frames to divide up LMs)
- In the case of underpredicting the LMs (e.g. 'Mime), this could be not 'breaking up' the LMs in the predictions as much as they should be (e.g. single predicted LM of 5 seconds should 2x2.5s instead)
- This may be due to predicting true 'Audio' frames incorrectly as the same LM as the previous frame
- In total, 82.48% of the true LMs were picked up by our system, which predicted 92.36% the number of LMs as there were in the source (predicted LMs are seen in 'LM detections.csv' in the appendix)

4.5: Comparison with Other Systems

As well as evaluating the system on its own merits, it's conducive to a good analysis to do a brief comparison with other systems used for audio detection. This is good for helping to give our system a more

critical eye when compared with other systems that are more complicated and labour intensive. It also helps expose us to new potential directions that this system could move to if it were to be adapted to other tasks.

'CNN Architectures for Large-Scale Audio Classification' (Hershey et al., 2017)

- System is used to classify the soundtracks of a dataset of 70 million training videos sourced from YouTube.com with 30k class labels, split into ~20 billion 960ms samples with up to 5 associated tags
- Used many different types of architectures to find best fit (while we only used 1, which was a CNN) including a standard fully-connected ANN, ResNet-50, Inception V3, and AlexNet (deep CNNs)
- Was able to train on up to 40 GPUs for several models to carry out the task, which we did not have access to in this project (only able to use 1 GPU)
- ResNet-50 found to be most optimal with a 0.926 area-under-curve of ROC curve

Thus, we can see that this project is far more advance than ours, involving more people (13 compared to 2), more GPUs (up to 40 compared to 1), more models tested with (5 models compared to 1), more classifications (30k compared to 6), and more training examples (~20 billion compared to ~25k). In other words, ours is a lot simpler, with a smaller data set, a simpler system and with fewer variations in network construction. However, ours predicts audio events at a rate similar to that of the paper's when using ResNet (with audio events here being YouTube video label tags for a given video the sample appears in)

'Classification of lung sounds using convolutional neural networks' (Aykanat et al., 2017)

- Respiratory sounds from 1630 subjects recorded for 17,930 samples (compared with ~25k in ours)
- Used both MFCC features with an SVM and spectrogram images with a CNN
- For the CNN model, created 8 data sets from the 17,930 (with a lot of overlap between datasets), ranging from 2 classes for one data set to 78 classes with another
- For one of the datasets created (w/ 15,328 samples and 3 classes), their CNN achieved an 80% test accuracy, while on another dataset, it achieved a 76% test accuracy (comparable to our system picking up 82.48% of LMs with 25k samples and 6 classes)
- Samples were around 10s (as suggested by chest physicians), while our samples were 0.4s long

This is a system that is more comparable to ours, as it involves similar amounts of data samples and, in some of the datasets, a more similar number of classes to our system. The accuracies are also comparable, (ours is within ~7% of both models and within 7 classes). However, their research used two model types to find the best solution, which was found to be either the SVM or CNN (both exhibited very similar test accuracies), while we only used 1, a CNN. Additionally, the audio samples they used were approximately 20 times longer in duration than our audio samples: this meant that there may have been a great deal more variation in their samples of respiratory data due to a longer timescale than our data.

5: Evaluation

5.1: Further Improvements to be Made

While the project is, for all intents and purposes, completed in most senses with a good LM frame classification and complete LM prediction ability, there are definitely improvements that can be made on the complete system which would give it greater extensibility to be adapted to do other tasks, perform better on the current task, or make the system generally easier to used. These improvements can be seen below:

Test more hyperparameter value combinations

- This would mean not permanently setting hyperparameter values (e.g. once we've established an optimal convolution kernel size and move onto testing activation functions, possibly trying with different convolutional kernel sizes again rather than keeping them permanently at one value)
- It could also mean experimenting with hyperparameters that we kept set through the whole testing process and didn't experiment with, such as batch size or pool window size and stride
- This may mean a lot more testing is needed, extending the duration of the project, but could expect to see a further increase in accuracy, decrease in variance, and possibly shorter runtime

Train system on a higher-end computer setup

- The most practical way of doing this would be training on a higher end desktop, such as one with a GeForce 1080ti graphics card, an extra 16GB of RAM, etc. (requiring a large financial investment)
- We would expect to see an improved training speed and amount of data the desktop can work with

Program using C++ implementation of TensorFlow

- As TensorFlow is build using a C++ backend, we can build our system using C++ with TensorFlow
- This would require learning advanced mechanics of C++ and setup a new programming environment
- However it would lead to a faster system as C++ works closer to low-level hardware features

Write script that calls all other scripts of the system

- This is more for convenience sake from the perspective of one wishing to use the system
- In its current form, from the beginning source audio file and an annotation file, each script needs to be run in turn, assuming the source files are in the expected location
- This may be inconvenient to a user, and thus creating a script that automatically calls each in turn (so the user only ever has to run the calling script) might make it a much more useable complete system

5.2: Gantt Chart and Project Timeline

(Figure 21 – Gantt chart)

In this section, we can see a Gantt chart showing the progress during this project's duration and when and roughly how long each task took to do relative to each other. Note that 'Week 1' started on 8th Jan 2018 and 'Week 12' ended on 1st April 2018. There are several observations that we can make from the above Gantt chart regarding the timeline that our project took:

- The creation of scripts involving the audio data preparation phase (not including final versions of the scripts created later that involved small tweaks and additions) took about a week longer than preferred, chiefly because new versions of the label creator and LM extraction scripts had to be made before proceeding onto the next section
- A week's gap in the project (Week 4) was taken due to needing to catchup on several other modules for the course, along with several days taken off due to illness
- An extra week was taken for stage 2, primarily due to excess time spent on setting up the programs and source files on the desktop, getting the programs to work on the GPU, and needing additional knowledge from a textbook and research papers before being able to build a better model
- All of the hyperparameter tunings and network architecture modifications were done in Weeks 8/9, which made for busy weeks and were not conducive to examining the 'bigger picture' of the project
- The final few weeks of the timeline is solely devoted to creating the presentation and the final report

Overall, the project took about as long as we expected it too, though taking longer in some areas (stages 1 and 2) than preferred, which meant that the later stages had to be rush a bit more.

5.3: Project Summary and Conclusions

Overall, it's deemed that the project was an overall success. This is on the basis that it satisfies the requirements of the system built and tasks given that include:

- Designing and developing a LM detection system based on CNNs
- Perform extensive experimental evaluations on given corpus of Wagner's music demonstrating the effect of different feature representations, network architectures, and network training parameters
- Thoroughly analysing the results and how the system could be improved upon

We have managed to build a system that, upon data being processed, shaped as input to a CNN, and provided as training to a model, predicts LM labels from 0.01s audio frames at 96.21% accuracy, with the model taking just over 5 minutes to train and predict. The model is then able to, from a position of being trained on a given corpus of data, predict complete LMs (of types it has been trained on) in given audio data and print the results of this to a file. However, the project was not without failures and/or things that could have been done better. The more notable successes and failures are summarized in the table below:

Successes	**Failures**
Overall strong finishing accuracy on test set of LM frames at 96.21%	Several attempts (and a week wasted) at getting the LM extractor script to work the way it should
Good predictive ability on complete LMs	Inability to get a confusion matrix returned from the 'build_cnn' function and unable to display as output
Hyperparameter tuning showed noticeable improvements on system and mostly conformed to what we expect would happen based on research	Requirement to 'undersample' data meant predicted complete LMs file has many gaps; failed to get the system to not predict all '0's without this

(Table 5 – Successes and failures table)

There are also several important takeaways from doing this complete project, not including the now-completed system, that include:

- Learning how CNNs work, how to implement them in Python code, and the effects of modifying their constituent components on general model accuracy and overall performance
- Experience with building, tuning, and debugging a model written using the TensorFlow library, as well as other addons including TensorBoard, that may transfer well into future projects requiring the machine learning library
- Understanding of the importance of controlled and thorough testing to optimize a system
- How to breakdown a project into tasks to ensure good time management and minimize 'roadblocks'

With all that said, the completed project is left in a good state, which should be easy to adapt to other audio tasks involving neural networks, and shows the potential to possibly build into more advanced systems of a similar nature in the future.

Bibliography

- Kumar, V. (2017) *Big Data Facts* Available at: https://analyticsweek.com/content/big-data-facts/ (Accessed: 20 March 2018)
- Silver, D., Schrittwieser, J., Simonyan, K., Antonoglou, I., Huang, A., Guez, A., Hubert, T., Baker, L., Lai, M., Bolton, A., Chen, Y., Lillicrap, T., Hui, F., Sifre, L., Driessche, G, Graepel, T., Hassabis, D. (2017) *Mastering the Game of Go without Human Knowledge*. Available at: https://deepmind.com/documents/119/agz_unformatted_nature.pdf (Accessed: 4 April 2018)
- Oord, A., Dieleman, S., Zen, H., Simonyan, K., Vinyals, O., Graves, A., Kalchbrenner, N., Senior, A. and Kavukcuoglu, K. (2016) *'WaveNet: A Generative Model for Raw Audio'*. Available at: https://arxiv.org/pdf/1609.03499.pdf (Accessed: 20 March 2018)
- Dieleman, S. (2014) *'Recommending music of Spotify with deep learning'* Available at: http://benanne.github.io/2014/08/05/spotify-cnns.html (Accessed: 20 March 2018)
- *API Documentation* (2017), <https://www.tensorflow.org/api_docs/> [accessed 20, March 2018]
- Kennedy, M., Kennedy, J.B. (1987) *The Concise Oxford Dictionary of Music*. Oxford, United Kingdom: Oxford University Press
- Warrack, J., Rosenthal, H. (1995) *Guide de l'opéra*. Paris, France: Fayard
- *Grout, D.J., Williams, H.W.* (2003) *A Short History of Opera*. New York, US: Columbia University Press
- Rohrer, B. (2016) *How Convolutional Neural Networks work*. Available at: https://www.youtube.com/watch?v=FmpDIaiMIeA&t (Accessed: 20 March 2018)
- Raschka, S. and Mirjalili, V. (2017) *Python Machine Learning*. 2nd edn. Birmingham, UK: Packt Publishing Ltd.
- Damien, A. (2017) *TensorFlow-Examples / examples / 3_NeuralNetworks / convolutional_network.py*. Available at: https://github.com/aymericdamien/TensorFlow-Examples/blob/master/examples/3_NeuralNetworks/convolutional_network.py (Accessed: 27 March 2018)
- Wyse, L. (2017) 'Audio spectrogram representations for processing with Convolutional Neural Networks', *Proceedings of the First International Workshop on Deep Learning and Music join with IJCNN*, 1(1), pp. 37-41
- Lee, J., Park, J., Kim, K.L., Nam, J. (2017) *Sample-level Deep Convolutional Neural Networks for Music Auto-tagging Using Raw Waveforms*, Korea Advanced Institute of Science and Technology. Available at: https://arxiv.org/pdf/1703.01789.pdf (Accessed: 26 March 2018)
- Pons, J., Nieto, O., Prockup, M., Schmidt, E.M., Ehmann, A.F., Serra, X. (2017) 'End-to-end learning for music audio tagging at scale', *Proceedings of the Workshop on Machine Learning for Audio Signal Processing (ML4Audio) at NIPS 2017*
- Buda, M., Maki, A., Mazurowski, M.A. (2017) *A systematic study of the class imbalance problem in convolutional neural networks*. Available at: https://arxiv.org/pdf/1710.05381.pdf (Accessed: 27 March 2018)
- Vaizman, Y., McFee, B., Lanckriet, G. (2013) 'Codebook based Audio Feature Representation for Music Information Retrieval', *Transactions on Audio Speech and Language Processing*, 22(10), pp. 1483-1493
- Ellis, D.P.W. (2007) *Classifying Music Audio with Timbral and Chroma Features*, Labrosa, Columbia University. Available at: https://labrosa.ee.columbia.edu/~dpwe/pubs/Ellis07-timbrechroma.pdf (Accessed: 22 March 2018)
- Lee, H., Largman, Y., Pham, P., Ng, A.Y. (2009) 'Unsupervised feature learning for audio classification using convolutional deep belief networks', *Advances in Neural Information Processing Systems*, 22
- Krizhevsky, A., Sutskever, I., Hinton, G. (2012) 'ImageNet Classification with Deep Convolutional Neural Networks', *Advances in Neural Information Processing Systems*, 25
- Simard, P.Y., Steinkraus, D., Platt, J.C. (2003) 'Best Practices for Convolutional Neural Networks Applied to Visual Document Analysis', *Seventh International Conference on Document Analysis and Recognition*, pp. 958-963

- Li, T., Chan, A.B., Chun. A. (2010) 'Automatic Musical Pattern Feature Extraction Using Convolutional Neural Network', *Proceedings of the International MultiConference of Engineers and Computer Scientists (IMECS 2010)*, 1, pp. 546-550
- Ruder, S. (2016) *An overview of gradient descent optimization algorithms*. Available at: http://ruder.io/optimizing-gradient-descent/index.html (Accessed: 26 March 2018)
- Hinton, G.E., Srivastava, N., Krizhevsky, A., Sutskever, I., Salakhutdinov, R.R. (2012) *Improving neural networks by preventing co-adaptation of feature detectors*, University of Toronto. Available at: https://arxiv.org/pdf/1207.0580.pdf (Accessed: 24 March 2018)
- Zha, S., Luisier, F., Andrews, W. (2015) *Exploiting Image-trained CNN Architectures for Unconstrained Video Classification*. Available at: https://arxiv.org/pdf/1503.04144.pdf (Accessed: 24 March 2018)
- Ba, L.J., Caruana, R. (2014) 'Do Deep Nets Really Need to be Deep?', *Advances in Neural Information Processing Systems*, 22
- Hu, B., Lu, Z., Li, H., Chen, Q. (2015) *Convolutional Neural Network Architectures for Matching Natural Language Sentences*. Available at: https://arxiv.org/pdf/1503.03244.pdf (Accessed: 25 March 2018)
- Shelhamer, E., Long, J., Darrell, T. (2016) *Fully Convolutional Networks for Semantic Segmentation*, University of California, Berkeley. Available at: https://arxiv.org/pdf/1605.06211.pdf (Accessed: 25 March 2018)
- Lin, M., Chen, Q., Yan, S. (2014) *Network in Network*, National University of Singapore. Available at: https://arxiv.org/pdf/1312.4400.pdf (Accessed: 25 March 2018)
- Sermanet, P., Eigen, D., Zhang, X., Mathieu, M., Fergus, R., LeCun, Y. (2014) *OverFeat: Integrated Recognition, Localization and Detection using Convolutional Networks*, New York University. Available at: *https://arxiv.org/pdf/1312.6229.pdf* (Accessed: 26 March 2018)
- Simonyan, K., Zisserman, A. (2015) *Very Deep Convolutional Networks For Large-Scale Image Recognition*, University of Oxford. Available at: *https://arxiv.org/pdf/1409.1556.pdf* (Accessed: 26 March 2018)
- Zeiler, M., Fergus, R. (2013) *Stochastic Pooling for Regularization of Deep Convolutional Neural Networks*, New York University. Available at: *https://arxiv.org/pdf/1301.3557.pdf* (Accessed: 26 March 2018)
- Hershey, S., Chaudhuri, Ellis, D.P.W., Gemmeke, J.F., Jansen, A., Moore, R.C., Plakal, M., Platt, D., Saurous, R.A., Seybold, B., Slaney, M., Weiss, R.J., Wilson, K. (2017) *CNN Architectures for Large-Scale Audio Classification*. Available at: *https://arxiv.org/pdf/1609.09430.pdf* (Accessed: 28 March 2018)
- Aykanat, M., Kılıç, Ö., Kurt, B., Saryal, S. (2017) 'Classification of lung sounds using convolutional neural networks', *Journal on Image and Video Processing*, 2017:65

Appendix I – Semester One Python Programs

TaskOne.py

```
import os
import numpy as np
import matplotlib.pyplot as plt
import scipy.io.wavfile as wvf
from scipy.fftpack import dct
from scipy import signal

file_path = "C:\Program Files\Python36\ThirdYearProject\Speech_TIMIT\\train\FCJF0"
file_list = os.listdir(file_path)

for file_name in file_list:
    try:
        file = file_path + "\\" + file_name
        rate, data = wvf.read(file)
        num_samples = len(data)
        framesize = int(0.064*rate)
        overlap = int(0.032 * rate)
        total_frames = int(num_samples/overlap)

        print("Samples of file " + file_name + " is: " + str(num_samples))
        print("Sampling rate of file: " + str(rate))
        print("Time length of file = " + str(len(data) / rate) + "s")
        print("Framesize: " + str(framesize) + " samples per frame")
        print("Overlap: " + str(overlap) + " samples per frame")
        print("Total frames: " + str(total_frames))
        print("\n")

        frames = np.ndarray((total_frames, framesize))
        dft_matrix = np.ndarray((total_frames, framesize))
        abs_dft_matrix = np.ndarray((total_frames, framesize))

        for i in range(total_frames):
            for j in range(framesize):
                if ((i*overlap + j) < num_samples):
                    frames[i][j] = data[i*overlap + j]
                else:
                    frames[i][j] = 0

        saw_filter_a = signal.waveforms.sawtooth(range(len(frames)), width = [0.5])
        saw_filter_b = signal.waveforms.sawtooth(range(len(frames)), width = [0.6])
        for i in range(total_frames):
            dft_matrix[i] = np.fft.fft(frames[i])
            dft_matrix[i] = signal.filtfilt(saw_filter_a, saw_filter_b, dft_matrix[i])
            dft_matrix[i] = dct(dft_matrix[i])
            abs_dft_matrix[i] = abs(dft_matrix[i]) * abs(dft_matrix[i]) / max(abs(dft_matrix[i]))
            abs_dft_matrix[i] = np.log10(abs_dft_matrix[i])


        f = open(file_name[:-4] + ".logFBE", "w+")
        f.writelines(str(abs_dft_matrix))

        t = range(len(abs_dft_matrix))
        plt.plot(t, abs_dft_matrix)
        plt.ylabel("Frequency")
        plt.xlabel("Frame number")

    except Exception as e:
        print("Exception thrown as: " + str(e))
        pass
```

TaskTwo.py

```
from math import sqrt, pi, exp

Y = [[0,0,0], [0.5,0,0], [0.5,-1,0], [0,0,-0.5],
        [1,0,0], [1,0,0.5], [0,0.5,0], [0,-0.5,0]]

c1_means = [[0,0,0], [1,0,0], [1,1,1]]
c1_vars  = [[[1,0,0], [0,2,0], [0,0,1]],
            [[1,0,0], [0,1,0], [0,0,1]],
            [[2,0,0], [0,2,0], [0,0,1]]]
c1_weights = [0.5, 0.3, 0.2]
c2_means = [[0,-1,0], [1,-1,0], [0,-1,-1]]
c2_vars = [[[1,0,0], [0,1,0], [0,0,1]],
           [[1,0,0], [0,1,0], [0,0,2]],
           [[2,0,0], [0,2,0], [0,0,1]],]
c2_weights = [0.4, 0.3, 0.3]

p_y_c1 = 1
p_y_c2 = 1
for i in range(8):              #for each yt feature vector
    weighted_sum_c1 = 0
    weighted_sum_c2 = 0
    for j in range(3):          #for each component
        temp_c1 = 0
        sigma_c1 = 1
        temp_c2 = 0
        sigma_c2 = 1
        for k in range(3):      #for each dimension
            temp_c1 += ((Y[i][k] - c1_means[j][k])**2) / c1_vars[j][k][k]
            sigma_c1 *= c1_vars[j][k][k]
            temp_c2 += ((Y[i][k] - c2_means[j][k])**2) / c2_vars[j][k][k]
            sigma_c2 *= c2_vars[j][k][k]
        p_yt_c1 = (1/sqrt(((2 * pi)**3)*sigma_c1))*exp(-0.5*temp_c1)
        weighted_sum_c1 += p_yt_c1 * c1_weights[j]
        p_yt_c2 = (1/sqrt(((2 * pi)**3)*sigma_c2))*exp(-0.5*temp_c2)
        weighted_sum_c2 += p_yt_c2 * c2_weights[j]
    p_y_c1 *= weighted_sum_c1
    p_y_c2 *= weighted_sum_c2

print("P(Y|C1) = " + str(p_y_c1))
print("P(Y|C2) = " + str(p_y_c2))
print()

p_c1 = 0.5
p_c2 = 0.5
p_y = p_c1*p_y_c1 + p_c2*p_y_c2

p_c1_y = p_c1*p_y_c1 / p_y
p_c2_y = p_c2*p_y_c2 / p_y

if p_c1_y > p_c2_y:
    print("Most likely class: C1")
    print("Estimated probability: " + str(p_c1_y))
else:
    print("Most likely class: C2")
    print("Estimated probability: " + str(p_c2_y))
```

TaskThree.py

```
import os
import numpy as np
import scipy.io.wavfile as wvf
from math import sqrt, pi, exp
from scipy import signal
from sklearn.mixture import GaussianMixture
from time import time
import warnings
warnings.filterwarnings('ignore')


print("\n\t//\t//\t//\tGaussian Multivariate Mixture Model for Audio Training and Prediction\t//\t//\t//")

num_components = 100
em_iterations = 100
reduced_num_dimensions = 50
tri_size = int(np.floor(512/reduced_num_dimensions) - 1)

def create_data_arrays(class_source, classes):
    stage_one_matrices = []
    for class_name in classes:
        class_matrices = []
        file_list = os.listdir(class_source + "\\" + class_name)
        if not os.path.exists(class_name):
            os.makedirs(class_name)
        for file_name in file_list:
            file = class_source + "\\" + class_name + "\\" + file_name
            rate, data = wvf.read(file)
            num_samples = len(data)
            framesize = int(0.064*rate)
            overlap = int(0.032 * rate)
            total_frames = int(num_samples/overlap)

            frames = np.ndarray((total_frames, framesize))
            dft_matrix = np.ndarray((total_frames, framesize))
            abs_dft_matrix = np.ndarray((total_frames, int(framesize/2)))

            for i in range(total_frames):
                for j in range(framesize):
                    if ((i*overlap + j) < num_samples):
                        frames[i][j] = data[i*overlap + j]
                    else:
                        frames[i][j] = 0

            saw_filter_a = signal.waveforms.sawtooth(range(len(frames)), width = [0.5])
            saw_filter_b = signal.waveforms.sawtooth(range(len(frames)), width = [0.5001])
            tri = signal.triang(M=tri_size)
            file_matrices = []
            for i in range(total_frames):
                dft_matrix[i] = np.fft.fft(frames[i])
                dft_matrix[i] = signal.filtfilt(saw_filter_a, saw_filter_b, dft_matrix[i])
                temp = abs(dft_matrix[i]) * abs(dft_matrix[i]) / max(abs(dft_matrix[i]))
                temp = np.log10(temp)
                abs_dft_matrix[i] = np.split(temp, 2)[0]
                final_dft_matrix = []
                for j in range(reduced_num_dimensions):
                    final_dft_matrix.append(np.dot(abs_dft_matrix[i][j*tri_size:(j+1)*tri_size], tri.T))
                file_matrices.append(final_dft_matrix)
            class_matrices.append(file_matrices)
        stage_one_matrices.append(class_matrices)
    return stage_one_matrices


#                       ***** STAGE 1 - FEATURE EXTRACTION FOR TRAINING DATA*****

print("\n\nStage 1...\n")

start_time = time()
class_source_train = "C:\Program Files\Python36\ThirdYearProject\Speech_TIMIT\\train\\"
train_classes = os.listdir(class_source_train)
stage_one_matrices = create_data_arrays(class_source_train, train_classes)
phase_one_time = time() - start_time
print(str(round(phase_one_time, 2)) + " seconds for stage 1; " + str(round(time() - start_time, 2)) + " seconds
total")


#                       ***** STAGE 2 - TRAINING THE GMMS *****

print("\n\nStage 2...\n")

weights, means, covariances = [], [], []

for i in range(2):
    temp_matrix = []
    for j in range(2):
        class_matrices = stage_one_matrices[2*i + j]
```

```
        for file_matrix in class_matrices:
            for frame in file_matrix:
                temp_matrix.append(frame)
    gmm = GaussianMixture(n_components = num_components, max_iter = em_iterations, covariance_type = 'full')
    gmm.fit(np.asarray(temp_matrix))
    print("GMM " + str(i+1) + " fitted")

    weights.append(gmm.weights_)
    means.append(gmm.means_)
    covariances.append(gmm.covariances_)

phase_two_time = time() - phase_one_time - start_time
print(str(round(phase_two_time, 2)) + " seconds for stage 2; " +
      str(round(time() - start_time, 2)) + " seconds total")


#                       ***** STAGE 3 - FEATURE EXTRACTION AND TESTING OF GMMS*****

print("\n\nStage 3...\n")

class_source_test = "C:\Program Files\Python36\ThirdYearProject\Speech_TIMIT\\test\\"
test_classes = os.listdir(class_source_test)
stage_three_matrices = create_data_arrays(class_source_test, test_classes)
conf_matrix = np.zeros((2,2))
equals_prob = 0


for class_num in range(len(stage_three_matrices)):                #for each class in 'test'
    print("Class " + str(class_num+1) + "...")
    for file_num in range(len(stage_three_matrices[class_num])):  #for each file within that class
        print("\tFile " + str(file_num+1) + "...")
        frame_matrix = stage_three_matrices[class_num][file_num]
        for frame_num in range(len(frame_matrix)):                #for each frame within the file

            frame = frame_matrix[frame_num]
            p_y_c_arr = []
            for i in range(2):                                    #for each GMM to compare it to

                p_y_c = 0
                for j in range(num_components):                   #for each component of the GMM
                    expon = 0
                    sigma = 1
                    for dim_num in range(reduced_num_dimensions):         #for each dimension of the frame
                        yt_n = frame[dim_num]           #FIX MEEEEEEEEE
                        mean = means[i][j][dim_num]
                        var = covariances[i][j][dim_num][dim_num]
                        expon += ((yt_n - mean) ** 2) / var
                        sigma *= abs(var)
                    sqrt_abs_sigma = sqrt(abs(sigma))
                    denominator = ((2 * pi) ** (reduced_num_dimensions/2)) * sqrt_abs_sigma
                    p_yt_c = np.float64((1 / denominator) * exp(-0.5 * expon))
                    p_y_c += np.float64(p_yt_c * weights[i][j])
                p_y_c_arr.append(p_y_c)
            p_c1, p_c2 = np.float64(0.5), np.float64(0.5)
            p_y = np.float64(p_c1*p_y_c_arr[0] + p_c2*p_y_c_arr[1])

            p_c1_y = np.float64(p_c1*p_y_c_arr[0] / p_y)
            p_c2_y = np.float64(p_c2*p_y_c_arr[1] / p_y)

            if p_c1_y > p_c2_y:
                conf_matrix[class_num][0] += 1
            elif p_c1_y < p_c2_y:
                conf_matrix[class_num][1] += 1
            else:
                equals_prob += 1

correct_nums = conf_matrix[0][0] + conf_matrix[1][1]
total_nums = conf_matrix[0][0] + conf_matrix[0][1] + conf_matrix[1][0] + conf_matrix[1][1] + equals_prob
accuracy = correct_nums / total_nums

print("\n")
print("Final confusion matrix")
print(conf_matrix[0])
print(conf_matrix[1])
print("Accuracy of: " + str(correct_nums) + " out of " + str(total_nums) +
      " = " + str(round((accuracy * 100), 2)) + "% accuracy\n")

phase_three_time = time() - phase_two_time - phase_one_time - start_time
print(str(round(phase_three_time, 2)) + " seconds for stage 3; " +
      str(round(time() - start_time, 2)) +" seconds total")
```

TaskFour.py

```
import os
from time import time
import numpy as np
from scipy import signal
import scipy.io.wavfile as wvf
from keras.models import Sequential
from keras.layers import Dense
from sklearn.metrics import confusion_matrix as cm
from sklearn.metrics import accuracy_score as acc_sco
import warnings
warnings.filterwarnings('ignore')


print("\n\t//\t//\t//\tAritifical Neural Network for Audio Training and Prediction\t//\t//\t//")

num_dimensions = 512

def create_data_arrays(class_source, classes):
    stage_one_matrices = []
    for class_name in classes:
        class_matrices = []
        file_list = os.listdir(class_source + "\\" + class_name)
        if not os.path.exists(class_name):
            os.makedirs(class_name)
        for file_name in file_list:
            file = class_source + "\\" + class_name + "\\" + file_name
            rate, data = wvf.read(file)
            num_samples = len(data)
            framesize = int(0.064*rate)
            overlap = int(0.032 * rate)
            total_frames = int(num_samples/overlap)

            frames = np.ndarray((total_frames, framesize))
            dft_matrix = np.ndarray((total_frames, framesize))
            abs_dft_matrix = np.ndarray((total_frames, int(framesize/2)))

            for i in range(total_frames):
                for j in range(framesize):
                    if ((i*overlap + j) < num_samples):
                        frames[i][j] = data[i*overlap + j]
                    else:
                        frames[i][j] = 0

            saw_filter_a = signal.waveforms.sawtooth(range(len(frames)), width = [0.5])
            saw_filter_b = signal.waveforms.sawtooth(range(len(frames)), width = [0.5001])
            for i in range(total_frames):
                dft_matrix[i] = np.fft.fft(frames[i])
                dft_matrix[i] = signal.filtfilt(saw_filter_a, saw_filter_b, dft_matrix[i])
                temp = abs(dft_matrix[i]) * abs(dft_matrix[i]) / max(abs(dft_matrix[i]))
                temp = np.log10(temp)
                abs_dft_matrix[i] = np.split(temp, 2)[0]
            class_matrices.append(abs_dft_matrix)
        stage_one_matrices.append(class_matrices)
    return stage_one_matrices


def data_preprocessing(data_matrices):
    y_length_zeroes = 0
    y_length_ones = 0
    stage_one_data = []
    for class_num in range(len(data_matrices)):
        for file_num in range(len(data_matrices[class_num])):
            for frame_num in range(len(data_matrices[class_num][file_num])):
                stage_one_data.append(data_matrices[class_num][file_num][frame_num])
                if class_num < (len(data_matrices) / 2):
                    y_length_zeroes += 1
                else:
                    y_length_ones += 1

    x = np.array(stage_one_data)
    zeroes = np.zeros(y_length_zeroes)
    ones = np.ones(y_length_ones)
    y = np.concatenate((zeroes, ones))

    return x,y


#                       ***** STAGE 1 *****

print("\n\nStage 1...\n")
start_time = time()
class_source_train = "C:\Program Files\Python36\ThirdYearProject\Speech_TIMIT\\train\\"
train_classes = os.listdir(class_source_train)
train_matrices = create_data_arrays(class_source_train, train_classes)
x_train, y_train = data_preprocessing(train_matrices)
```

```
class_source_test = "C:\Program Files\Python36\ThirdYearProject\Speech_TIMIT\\test\\"
test_classes = os.listdir(class_source_test)
test_matrices = create_data_arrays(class_source_test, test_classes)
x_test, y_test = data_preprocessing(test_matrices)

phase_one_time = time() - start_time
print(str(round(phase_one_time, 2)) + " seconds for stage 1; " +
      str(round(time() - start_time, 2)) + " seconds total")


#                       ***** STAGE 2 - TRAINING THE ANN *****

print("\n\nStage 2...\n")
num_extra_hidden_layers = 3
num_hidden_nodes = 30
ann = Sequential()

ann.add(Dense(units= num_hidden_nodes, activation= 'sigmoid', input_dim= num_dimensions))
for i in range(num_extra_hidden_layers):
    ann.add(Dense(units = num_hidden_nodes, activation= 'sigmoid'))
ann.add(Dense(units= 1, activation= 'sigmoid'))

ann.compile(optimizer= 'adam', loss = 'binary_crossentropy', metrics= ['accuracy'])
ann.fit(x= x_train, y= y_train, batch_size= 10, nb_epoch= 30)

phase_two_time = time() - phase_one_time - start_time
print(str(round(phase_two_time, 2)) + " seconds for stage 2; " +
      str(round(time() - start_time, 2)) + " seconds total")


#                       ***** STAGE 3 - TESTING THE ANN *****

print("\n\nStage 3...\n")

y_predict = ann.predict(x_test)
conf_mat = cm(y_true= y_test, y_pred= y_predict.round())
accuracy = acc_sco(y_true=y_test, y_pred= y_predict.round())

print("\n\nConfusion matrix")
print(conf_mat)
print("\n\nAccuracy")
print(str(round(accuracy*100, 2)) + "%")

phase_three_time = time() - phase_two_time - phase_one_time - start_time
print(str(round(phase_three_time, 2)) + " seconds for stage 3; " +
      str(round(time() - start_time, 2)) +" seconds total")
print("Hidden layers = " + str(num_extra_hidden_layers + 1) + " with " +
      str(num_hidden_nodes) + " nodes each")
```

Appendix II – Semester Two Python Programs

LeitmotivStats.py

```
import pandas as pd
import matplotlib.pyplot as plt
from collections import OrderedDict
import numpy as np

location = "C:\\Users\\Dan\\Dropbox\\Uni stuff\\EE3P\\"      #Location of annotation file
annotation_file = pd.read_csv(                              #Annotation file imported as object, only the
    location + "siegfried_annotations.csv").iloc[195:666]   #rows pertaining to our source .wav file

start_times = annotation_file.iloc[:, 6:7].values           #Extract start times, finish times, LM labels,
finish_times = annotation_file.iloc[:, 7:8].values          #and salience ratings from respective
lm_labels = annotation_file.iloc[:, 1:2].values             #columns in annotation file object
ratings = annotation_file.iloc[:, 5:6].values

lm_type_count = {}                                          #setup empty dictionary for LM names and counts
lm_names = []                                               #names of LMs in dictionary
counts = []                                                 #counts of LMs in dictionary

for lm in lm_labels:                                        #For each LM label in annotation file
    lm = str(lm)[2:-2]                                      #reduce lbl (e.g. "[[Mime]]" to "Mime")
    if lm in lm_type_count:                                 #If an LM is already in dictionary,
        lm_type_count[lm] += 1                              #increment its count
    else:                                                   #else, create new LM in dict and set
        lm_type_count[lm] = 1                               #count to 1

od = OrderedDict(sorted(lm_type_count.items(),              #Sort dictionary by ascending counts
                        key=lambda t: t[1]))
for lm_name in od:                                          #For each LM label in sorted dictionary
    lm_names.append(lm_name[:5])                            #append it to a list (first 5 letters)
    counts.append(od[lm_name])                              #and append its count to a list
lm_names = lm_names[::-1]                                   #Reverse both lists so counts are descending
counts = counts[::-1]                                       #with LM label list matches positions

plt.bar(range(len(lm_names)), counts, align='center')       #Plot bar graph as with names in LM label list
plt.xticks(range(len(lm_names)), lm_names, rotation=45)     #as x-axis, the bar's values as corresponding
plt.xlabel("Leitmotiv names")                               #positions in count list
plt.ylabel("Leitmotiv frequency")                           #Set the x labels as LM names at 45 degrees
plt.title("Frequency of specific leitmotivs")               #Set the x-axis, y-axis, and title labels
plt.show()                                                  #and display the graph

def time_converter(times):                                  #Function takes in times as list of strings,
    converted_times = []                                    #e.g. "[[01:13:45:384]]"
    for time in times:                                      #for each time in the list, convert to string,
        time = str(time)[2:-2]                              #remove ends of string to leave "01:13:45:384",
        split = str(time).split(':')                        #split each part on colons into list of parts,
        hrs_to_ms = int(split[0]) * 60 * 60 * 1000          #convert first part (hours) to milliseconds,
        mins_to_ms = int(split[1]) * 60 * 1000              #second part (minutes) to milliseconds,
        secs_to_ms = int(split[2]) * 1000                   #third part (seconds) to milliseconds,
        total_time = hrs_to_ms + mins_to_ms + \
                     secs_to_ms + (int(split[3]) * 10)      #and add all of them together to get time
        converted_times.append(total_time)                  #in milliseconds, and add to list
    return converted_times                                  #return converted times in milliseconds

durations = []                                              #Create empty durations list
for st, ft in zip(start_times, finish_times):               #For each start/finish time in annot file
    cts = time_converter([st, ft])                          #Convert the strings to milliseconds in integer
    duration = abs(cts[1] - cts[0]) / 1000                  #Find difference in start/finish times in seconds
    durations.append(duration)                              #and add to durations list

plt.hist(durations, 100)                                    #Plot all durations of LMs in durations list
plt.xlabel("Leitmotiv duration (seconds)")                  #as histogram
plt.ylabel("Frequency")                                     #Set the x-axis, y-axis, and title labels
plt.title("Frequency of leitmotiv durations")
plt.axis([0, 25, 0, 120])                                   #Set the scale of the axes
plt.show()                                                  #and display the graph

final_ratings = []                                          #Create empty ratings list
for rating in ratings:                                      #Extract ratings from each list
    for r in rating:                                        #from the ratings column in annotation file
        final_ratings.append(int(r))                        #and append to ratings list

plt.scatter(durations, final_ratings)                       #Plot scatter plot with durations on x-axis
plt.plot(np.unique(durations),                              #and extracted salience ratings on y-axis
         np.poly1d(np.polyfit(durations, final_ratings,     #with a line of best fit running through them
                              1))(np.unique(durations)))
plt.xlabel("Leitmotiv duration (seconds)")                  #Set the x-axis, y-axis, and title labels
plt.ylabel("Salience rating (1-5)")
plt.title("Ratings of leitmotivs against duration")
plt.show()                                                  #Display the graph
```

LMStatsExtraGraphs.py

```
import pandas as pd
import matplotlib.pyplot as plt
from collections import OrderedDict

location = "C:\\Users\\Dan\\Dropbox\\Uni stuff\\EE3P\\"      #Location of annotation file
annotation_file = pd.read_csv(                              #Annotation file imported as object, only the
    location + "siegfried_annotations.csv").iloc[195:666]   #rows pertaining to our source .wav file

start_times = annotation_file.iloc[:, 6:7].values           #Extract start times, finish times, LM labels,
finish_times = annotation_file.iloc[:, 7:8].values          #and salience ratings from respective
lm_labels = annotation_file.iloc[:, 1:2].values             #columns in annotation file object
ratings = annotation_file.iloc[:, 5:6].values

lm_type_count = {}                                          #setup empty dictionary for LM names and counts
lm_names = []                                               #names of LMs in dictionary
counts = []                                                 #counts of LMs in dictionary

for lm in lm_labels:                                        #For each LM label in annotation file
    lm = str(lm)[2:-2]                                      #reduce lbl (e.g. "[[Mime]]" to "Mime")
    if lm in lm_type_count:                                 #If an LM is already in dictionary,
        lm_type_count[lm] += 1                              #increment its count
    else:                                                   #else, create new LM in dict and set
        lm_type_count[lm] = 1                               #count to 1

od = OrderedDict(sorted(lm_type_count.items(),              #Sort dictionary by ascending counts
                        key=lambda t: t[1]))
for lm_name in od:                                          #For each LM label in sorted dictionary
    lm_names.append(lm_name)                                #append it to a list (full name)
    counts.append(od[lm_name])                              #and append its count to a list
lm_names = lm_names[::-1][:5]                               #Reverse both lists so counts are descending
counts = counts[::-1][:5]                                   #with LM label list matches positions
                                                            #and reducing lists to only first 5 (most common)


def time_converter(times):                                  #Function takes in times as list of strings,
    converted_times = []                                    #e.g. "[[01:13:45:384]]"
    for time in times:                                      #for each time in the list, convert to string,
        time = str(time)[2:-2]                              #remove ends of string to leave "01:13:45:384",
        split = str(time).split(':')                        #split each part on colons into list of parts,
        hrs_to_ms = int(split[0]) * 60 * 60 * 1000          #convert first part (hours) to milliseconds,
        mins_to_ms = int(split[1]) * 60 * 1000              #second part (minutes) to milliseconds,
        secs_to_ms = int(split[2]) * 1000                   #third part (seconds) to milliseconds,
        total_time = hrs_to_ms + mins_to_ms + \
                     secs_to_ms + (int(split[3]) * 10)      #and add all of them together to get time
        converted_times.append(total_time)                  #in milliseconds, and add to list
    return converted_times                                  #return converted times in milliseconds


for lm_name, count in zip(lm_names, counts):                #For each LM label and corresponding count
    durations = []                                          #create a new empty durations list
    for st, ft, lbl in zip(start_times, finish_times,       #For each start time, finish time, LM label
                           lm_labels):                      #in the annotation file,
        if lbl == lm_name:                                  #if the LM in the annot file matches the one
            cts = time_converter([st, ft])                  #we're working with currently, convert its
            duration = abs(cts[1] - cts[0]) / 1000          #start and end times to milliseconds as integer,
            durations.append(duration)                      #calculate duration from these in seconds and
                                                            #append to list of durations
    plt.hist(durations, 100)                                #Plot this histogram of durations
    plt.xlabel("'" + str(lm_name) +                         #Set x-axis based on name of LM we're looking at
               "' duration (seconds)")
    plt.ylabel("Frequency")                                 #Set y-axis to 'frequency'
    plt.title("Frequency of " + str(lm_name) +              #Set title based on name of LM we're looking at
              " durations (" + str(count) + " total)")      #and also its number of occurences in annot file
    plt.show()                                              #Display the graph
```

Code Overview

The full explanation of how each of these codes run can be found below. 'LeitmotivStats.py' is first described as:

1.) Annotation file is read in and only rows that apply to our file are selected as 'annotation_file' variable
2.) Start times, finish times, LM labels, and salience ratings are extracted from appropriate columns
3.) Create a dictionary containing all LM labels as keys and their frequencies ('counts') as values
4.) Order this dictionary by descending frequencies, shorten LM labels to first 5 letters, add keys and values to separate lists, and reverse lists so LM labels and frequencies are now descending

5.) Plot these as a bar graph, with each label in LM labels being one bar and its height being frequency
6.) Each of the start and finish times of every LM is then converted from “hh:mm:ss:msmsms” into integers of total milliseconds
7.) The duration for each LM is found by calculating the difference between start and end times, dividing by 1000 (to get in seconds), which is appended to a list
8.) This 'durations' list is then plotted as a histogram to show frequencies of LM durations
9.) Each of the ratings of each LM is added to a list, which is then plotted against LM durations, which also includes a line of best fit through the plotted points

The first three steps of 'LMStatsExtraGraphs.py' are exactly the same as the first 3 from 'LeitmotivStats.py'; the steps that follow can then be described as the following:

4.) Order this dictionary by descending frequencies, **don't** shorten name to first 5 letters, add keys and values to separate lists, reverse lists so LM labels and frequencies are now descending, and take only the top 5 most common LM labels and their corresponding frequencies
5.) For each of these LM label and frequency pairs, look through the list of LMs with corresponding labels, start and finish times, and determine if each LM is the same label as the one in the label-frequency pair we are currently looking at: if it is, add the difference between its start and end time (duration) to a list of durations
6.) For each of these LM label and frequency pairs, plot the frequencies of LM duration lengths with the title of each graph saying which label it represents and the total number of LMs with that label

LeitmotivExtractor.py

```
from pydub import AudioSegment as AS
import pandas as pd
import os

location = "C:\\Users\\Dan\\Dropbox\\Uni stuff\\EE3P\\"      #Location of source .wav and annotation file
source_file = AS.from_wav(location +
                          "boulez_siegfried-act1.wav")      #Source file imported as object from directory
annotation_file = pd.read_csv(                              #Annotation file imported as object, only the
    location + "siegfried_annotations.csv").iloc[195:666]   #rows pertaining to our source .wav file
target_location = "C:\\Users\\Dan\\Documents\\LMs\\"        #Location to place newly created .wav files

for file in os.listdir(target_location):                    #Removes files already in LM directory
    os.remove(target_location + file)

start_times = annotation_file.iloc[:, 6:7].values           #Extract start times, finish times, and LM labels
finish_times = annotation_file.iloc[:, 7:8].values          #from respective columns in annotation file object
lm_labels = annotation_file.iloc[:, 1:2].values


def time_converter(times):                                  #Function takes in times as list of strings,
    converted_times = []                                    #e.g. "[[01:13:45:384]]"
    for time in times:                                      #for each time in the list, convert to string,
        time = str(time)[2:-2]                              #remove ends of string to leave "01:13:45:384",
        split = str(time).split(':')                        #split each part on colons into list of parts,
        hrs_to_ms = int(split[0]) * 60 * 60 * 1000          #convert first part (hours) to milliseconds,
        mins_to_ms = int(split[1]) * 60 * 1000              #second part (minutes) to milliseconds,
        secs_to_ms = int(split[2]) * 1000                   #third part (seconds) to milliseconds,
        total_time = hrs_to_ms + mins_to_ms + \
                     secs_to_ms + (int(split[3]) * 10)      #and add all of them together to get time
        converted_times.append(total_time)                  #in milliseconds, and add to list
    return converted_times                                  #return converted times in milliseconds


def reverse_time_converter(times):                          #Function takes in times as integers in a list,
    converted_times = []                                    #and converts each of them back to seconds,
    for time in times:                                      #rounding each down to closest second
        num_seconds = int(time / 1000)
        converted_times.append(num_seconds)
    return converted_times


min_length = 3000                                           #Minimum time length of .wav set here
file_counter = 0                                            #Counter to retrieve the correct label
lm_positions = []                                           #Empty lists to hold label, times
audio_gap_pos = []                                          #and second to hold info on audio gaps

for st, ft in zip (start_times, finish_times):              #For each start and finish time,
    file_counter += 1                                       #counter is incremented,
    cts = time_converter([st, ft])                          #start/finish times converted to ms,
    cts.append(lm_labels[file_counter-1])                   #corresponding label for times is found,
    lm_positions.append(cts)                                #and the [st, ft, lbl] listed added
                                                            #to 'lm_positions'

    if len(lm_positions) < 2:                               #this list then checked for 2 or more LM
        continue                                            #data is stored in list; if not, repeat
                                                            #previous 'for' loop steps
    else:
        finish_pos = lm_positions[-1][1]                    #else, total length of list of positions is
        start_pos = lm_positions[0][0]                      #checked to make sure LMs add up to >3000ms
        if (finish_pos - start_pos) < min_length:           #if not, repeat previous 'for' loop steps
            continue
        else:                                               #else, next iterate through each LM in list
            label_strings = []
            prev_label = "EMP"
            for i in range(len(lm_positions)):              #for each [st, ft, lbl] list in lm_positions
                label = str(lm_positions[i][2])[2:-2][:3]   #extract the lbl and shorten to 1st 3 letters
                if i != 0:                                  #if not first [st, ft, lbl] list in lm_pos.
                    prev_label = label_strings[-1][1:]      #label of prev list is examined
                if label[:3] == prev_label[:3]:             #if they are same (e.g. first 3 letters same),
                    if len(prev_label) == 3:                #if len(prev label) is 3, only 1 of LM type,
                        prev_label = "_" + prev_label + "2" #so prev label changed to 'lbl2'
                    else:                                   #else, last letter is incremented ('Jug4' to 'Jug5')
                        prev_label = "_" + prev_label[:-1] + \
                                     str(int(prev_label[-1])+1)
                    label_strings[-1] = prev_label          #list appended with change
                else:
                    label_strings.append("_" + label)       #if prev label not the same, append name on
                                                            #the end (e.g. 'Jug' becomes 'Jug_Nib')
            final_string = "".join(label_strings)           #all strings in list are stuck together
            lm = source_file[start_pos:finish_pos]          #the smaller .wav is extracted from start/finish pos
            audio_gap_pos.append([start_pos, finish_pos])   #the start/finish times appended to 'audio_gap' list
            rcts = reverse_time_converter(                  #the start/finish times converted to seconds
                [start_pos, finish_pos])
            lm.export(target_location + "boulSiegAct1mp3_"+ #smaller .wav written with created string to target
                      str(rcts[0]) + "to" + str(rcts[1]) +  #location, with times they represent also included
                      final_string + ".wav", format="wav")
```

```
            lm_positions.clear()                                #lm_positions list emptied for next iteration

for i in range(len(audio_gap_pos)):                             #for each number in 'audio_gap_pos'
    if i != 0:                                                  #if not the first entry
        start_time = audio_gap_pos[i-1][1]                      #checks final 'finish time' and first 'start time'
        end_time = audio_gap_pos[i][0]
        if (end_time - start_time) >= 40000:                    #if gap between them is more than 40000ms
            lm = source_file[start_time:end_time]               #create a smaller .wav with these times,
            rcts = reverse_time_converter(                      #convert times to seconds,
                [start_time, end_time])
            lm.export(target_location + "boulSiegAct1mp3_"      #and write to file with '_aud' describing file
                      + str(rcts[0]) + "to" +
                      str(rcts[1]) + "_aud.wav",
                      format="wav")
            lm_positions.clear()                                #lm_positions list emptied for next iteration
```

LeitmotivExtractorV2.py

```
from pydub import AudioSegment as AS
import pandas as pd
import os

location = "C:\\Users\\Dan\\Dropbox\\Uni stuff\\EE3P\\"      #Location of source .wav and annotation file
source_file = AS.from_wav(location +
                          "boulez_siegfried-act1.wav")      #Source file imported as object from directory
annotation_file = pd.read_csv(                              #Annotation file imported as object, only the
    location + "siegfried_annotations.csv").iloc[195:666]   #rows pertaining to our source .wav file
target_location = "C:\\Users\\Dan\\Documents\\LMs\\"        #Location to place newly created .wav files

for file in os.listdir(target_location):                    #Removes files already in LM directory
    os.remove(target_location + file)

start_times = annotation_file.iloc[:, 6:7].values           #Extract start times, finish times, LM labels
finish_times = annotation_file.iloc[:, 7:8].values          #from respective columns in annotation file object
lm_labels = annotation_file.iloc[:, 1:2].values


def time_converter(times):                                  #Function takes in times as list of strings,
    converted_times = []                                    #e.g. "[[01:13:45:384]]"
    for time in times:                                      #for each time in the list, convert to string,
        time = str(time)[2:-2]                              #remove ends of string to leave "01:13:45:384",
        split = str(time).split(':')                        #split each part on colons into list of parts,
        hrs_to_ms = int(split[0]) * 60 * 60 * 1000          #convert first part (hours) to milliseconds,
        mins_to_ms = int(split[1]) * 60 * 1000              #second part (minutes) to milliseconds,
        secs_to_ms = int(split[2]) * 1000                   #third part (seconds) to milliseconds,
        total_time = hrs_to_ms + mins_to_ms + \
                     secs_to_ms + (int(split[3]) * 10)      #and add all of them together to get time
        converted_times.append(total_time)                  #in milliseconds, and add to list
    return converted_times                                  #return converted times in milliseconds


def reverse_time_converter(times):                          #Function takes in times as integers in a list,
    converted_times = []                                    #and converts each of them back to seconds,
    for time in times:                                      #rounding each down to closest second
        num_seconds = int(time / 1000)
        converted_times.append(num_seconds)
    return converted_times


def create_lm_file(window, lms):                            #Function takes in window pos' and LMs within
    label_strings = []
    prev_label = "EMP"
    for i in range(len(lms)):                               #for however many LMs to be in this file
        short_label = lms[i][2][:3]                         #shorten lbl for each LM
        if i != 0:                                          #if not the first LM in list of lms
            prev_label = label_strings[-1][1:]              #get previous label appended in list
        if short_label[:3] == prev_label[:3]:               #if prev lbl is same as current lbl
            if len(prev_label) == 3:                        #if len(prev label) is 3, only 1 of LM type,
                prev_label = "_" + prev_label + "2"         #so prev label changed to 'lbl2'
            else:                                           #else, last letter is incremented ('Jug4' to 'Jug5')
                prev_label = "_" + prev_label[:3] + \
                             str(int(prev_label[3:]) + 1)
            label_strings[-1] = prev_label                  #list appended with change
        else:
            label_strings.append("_" + short_label)         #if prev label not the same, append name on
                                                            #the end (e.g. 'Jug' becomes 'Jug_Nib')
    final_string = "".join(label_strings)                   #all strings in list are stuck together
    lm = source_file[window[0]:window[1]]                   #the smaller .wav is extracted from window pos'
    rcts = reverse_time_converter([window[0], window[1]])   #the window times converted to seconds
    lm.export(target_location + "boulSiegAct1mp3_" +        #smaller .wav written to target location with
              str(rcts[0]) + "to" + str(rcts[1]) +          #window times as seconds and string of LMs inside
              final_string + ".wav", format="wav")


window = [0, 40000]                                         #defines window as list with a start and end time
end_file_time = time_converter([finish_times[-1]])[0]       #gets end time from last time in annotation file
while(window[1] < end_file_time):                           #while the end of window not exceeded end of file
    lms = []                                                #empty LMs list defined with '0' current LM durat
    lm_duration = 0
    for st, ft, lbl in zip(start_times,                     #for each start time, finish time,
                           finish_times, lm_labels):        #label in annotation file
        cts = time_converter([st, ft])                      #convert start and finish times to integers
        lbl = str(lbl)[2:-2]                                #reduce lbl (e.g. "[[Mime]]" to "Mime")
        if (int(cts[0]) >= int(window[0])):                 #if LM start time greater than window start time
            if (int(cts[1]) <= int(window[1])):             #if LM end time less than window end time
                lms.append([cts[0], cts[1], lbl])           #add [st, ft, lbl] of LM to list of LMs
                lm_duration += (cts[1] - cts[0])            #add duration of LM to total durations
        elif (cts[0] >= window[0]) and \
                (cts[0] <= window[1]):                      #else, if only start time of LM within window
            window[1] = ft                                  #extend window to edge of LM
            lms.append([cts[0], cts[1], lbl])               #add [st, ft, lbl] of LM to list of LMs
            lm_duration += (cts[1] - cts[0])                #add duration of LM to total durations
```

```
if len(lms) == 0:                                   #if no LMs extracted into list
    lm = source_file[window[0]:window[1]]           #create smaller .wav from window
    rcts = reverse_time_converter(                  #convert window times to milliseconds
        [window[0], window[1]])
    lm.export(target_location + "boulSiegAct1mp3_" +    #write .wav to file at target location
              str(rcts[0]) + "to" + str(rcts[1]) +      #with times in seconds and '_aud' ending
              "_aud.wav", format="wav")
elif len(lms) == 1:                                 #else if there's 1 LM in the list
    create_lm_file(window, lms)                     #create smaller .wav from window and LMs list
elif (lm_duration >= 3000) and (len(lms) >= 2):     #else if LM total duration exceeds 3 seconds and
                                                    #there are at least 2 LMs in the list
    create_lm_file(window, lms)                     #create smaller .wav from window and LMs list
else:
    for st, ft, lbl in zip(start_times,             #else, for each start time, finish time, lbl
                           finish_times, lm_labels):
        cts = time_converter([st, ft])              #convert times to milliseconds
        lbl = str(lbl)[2:-2]                        #reduce lbl (e.g. "[[Mime]]" to "Mime")
        if cts[0] > lms[-1][0]:                     #if LM start time greater than finish time of
                                                    #last in list (i.e. occurs after last in list)
            lms.append([cts[0], cts[1], lbl])       #append this to list of LMs
            lm_duration += (cts[1] - cts[0])        #add duration of LM to total durations
            window[1] = cts[1]                      #extend window to end of LM
        if lm_duration >= 3000:                     #if LM list now contains more than 3 secs of LMs
            break                                   #then exit the loop
    create_lm_file(window, lms)                     #create smaller .wav from window and LMs list

window[0] = window[1]                               #Move along window so new start time is old end
window[1] = window[0] + 40000                       #time and new end time is 4 seconds along
```

Code Overview and Changes Between Versions 1 and 2

The full explanation of how the 'LeitmotivExtractorV2.py' code runs can be found below:

1.) Read in the full source file using the AudioSegment.from_wav function from the 'pydub' library, read in the annotation file (only rows that pertain to the source file) via the read_csv function from the 'pandas' library, and delete all current files in the directory we wish to use
2.) Window is defined to be a list of 2 values that 'slides along' the file, with the first being the start and the second being the end point, and is initially set to 0 and 40000 respectively for 0s and 40s
3.) Determine where the 'end' file time is based on the final finish time seen in the annotation file and enter a while loop which will continue until the extraction window's 'end' value exceeds this end file time (i.e. until the extraction window goes beyond the end of the files)
4.) For each of the start and end times and the LM labels, convert the times into integers showing the duration in milliseconds, and check if the start and end times in question fall in between the extraction window: if it does, add it to the list of LMs to append and increase the value of 'lm_duration' by its duration; if only the start time falls in the window (i.e. the LM extends beyond the extraction window), extend the window to the end of the LM, add this LM to the list of LMs to append, and increase the 'lm_duration'
5.) One this is done for all LMs and the window size is set to extract complete LMs, we then check a series of conditions: if there are no LMs that fall in this window, write a .wav file with no LMs in it based on the start and end times shown in the window of 40s long; else if there is only one or if there are two LMs with cumulative time of 3 seconds, write a .wav with this given extraction window start and end times; otherwise, extend this extraction window until the total duration of LMs exceeds 3 seconds, at which point write this section of the source file to a new .wav file
6.) Move along the window via setting the new start value to the old end value and the new end value 40 seconds further along from the start value

The first version of this program (which is also included abovee as 'LeitmotivExtractor.py') was quite different in how it created new smaller .wav files: instead of starting with an extraction window and moving it across based on several conditions, we instead just checked for conditions starting from a length of 0 (rather than 40s in the case of the extraction window). And although we checked for many similar conditions (must have 2 LMs in each file, LMs must be at least 3 seconds in cumulative duration, and LMs must not be broken up between files), the result is that they are much smaller than what we would like (each file is on average 2-4 seconds long). Being such short files, its not as easy to be able to contrast LMs found in the files with other non-LM ('Audio') data in the same file in the same way that 40 second long files can be. Hence, it was deemed preferable to have fewer .wav files but of longer durations, while also retaining many of the rules from the initial version.

LabelCreator.py

```
import pandas as pd
import os

location = "C:\\Users\\Dan\\Dropbox\\Uni stuff\\EE3P\\"      #Location of source .wav and annotation file
lm_location = "C:\\Users\\Dan\\Documents\\LMs\\"            #Location of smaller .wavs from LM extractor
target_location = "C:\\Users\\Dan\\Documents\\LMlabels\\"   #Location to place new .lab files

for file in os.listdir(target_location):                   #Removes files already in LM directory
    os.remove(target_location + file)

annotation_file = pd.read_csv(                             #Annotation file imported as object, only the
    location + "siegfried_annotations.csv").iloc[195:666]  #rows pertaining to our source .wav file
start_times = annotation_file.iloc[:, 6:7].values          #Extract start times, finish times, LM labels
finish_times = annotation_file.iloc[:, 7:8].values         #from respective columns in annotation file object
lm_labels = annotation_file.iloc[:, 1:2].values


def time_converter(times):                                 #Function takes in times as list of strings,
    converted_times = []                                   #e.g. "[[01:13:45:384]]"
    for time in times:                                     #for each time in the list, convert to string,
        time = str(time)[2:-2]                             #remove ends of string to leave "01:13:45:384",
        split = str(time).split(':')                       #split each part on colons into list of parts,
        hrs_to_ms = int(split[0]) * 60 * 60 * 1000         #convert first part (hours) to milliseconds,
        mins_to_ms = int(split[1]) * 60 * 1000             #second part (minutes) to milliseconds,
        secs_to_ms = int(split[2]) * 1000                  #third part (seconds) to milliseconds,
        total_time = hrs_to_ms + mins_to_ms + \
                     secs_to_ms + (int(split[3]) * 10)     #and add all of them together to get time
        converted_times.append(total_time)                 #in milliseconds, and add to list
    return converted_times                                 #return converted times in milliseconds


for file_name in os.listdir(lm_location):                  #For each smaller .wav file
    f = open(target_location + file_name[:-4] +            #create a file with the same name but
             ".lab", "w+")                                 #as a .lab rather than a .wav
    start_time = int(file_name.split("_")[1].split("to")   #Get start time of small .wav in context
                     [0]) * 10000000                       #of overall source .wav time
    finish_time = int(file_name.split("_")[1].split("to")  #Get finish time of small .wav in context
                      [1]) * 10000000                      #of overall source .wav time
    for st, ft, lbl in zip(start_times,                    #For each start time, finish time, label
                           finish_times, lm_labels):       #from the source file for a specific LM
        cts = time_converter([st, ft])                     #convert the start/finish times to ms
        if cts[0] >= start_time \
                and int(cts[1]/10000000)*10000000 \
                        <= finish_time:                    #and check if LM falls completely within
            f.write(str(cts[0]) + "\t" +                   #time context of smaller .wav file
                    str(cts[1]) + "\t" +                   #if so, write a line as <LM start time> \t
                    str(lbl)[2:-2] + "\n")                 #<LM finish time> \t <LM label>
    f.close()                                              #and close the file

    f = open(target_location + file_name[:-4] +            #reopen the same file but only for reading
             ".lab", "r")
    lines = f.readlines()                                  #read in the file as a list of file lines
    f.close()                                              #close the file
    for i in range(len(lines)):                            #for each line in the list,
        if i != 0:                                         #if it's not the first line in the list
            prev_line = lines[i - 1]                       #get the line before the current one
            current_line = lines[i]                        #and the current line
            if current_line.split("\t")[0] > \
                    prev_line.split("\t")[1]:
                line_to_insert = \
                    str(prev_line.split("\t")[1]) + \
                    "\t" + \
                    str(current_line.split("\t")[0]) \
                    + "\t" + "Audio" + "\n"

                                                           #if start time of current line is greater
                                                           #than the finish time of previous line,
                                                           #add an audio line between them filling
                                                           #the gap in time
                lines.insert(i, line_to_insert)            #insert this 'Audio' line correctly into list
    f = open(target_location + file_name[:-4] +            #reopen the .labl file for writing
             ".lab", "w")
    lines = "".join(lines)                                 #join the list together as one long string and
    f.write(lines)                                         #write this string to the file
    f.close()
```

LabelCreatorV2.py

```
import pandas as pd
import os

location = "C:\\Users\\Dan\\Dropbox\\Uni stuff\\EE3P\\"      #Location of source .wav and annotation file
lm_location = "C:\\Users\\Dan\\Documents\\LMs\\"            #Location of smaller .wavs from LM extractor
target_location = "C:\\Users\\Dan\\Documents\\LMlabels\\"   #Location to place new .lab files

for file in os.listdir(target_location):                   #Removes files already in LM directory
    os.remove(target_location + file)

annotation_file = pd.read_csv(                             #Annotation file imported as object, only the
    location + "siegfried_annotations.csv").iloc[195:666]  #rows pertaining to our source .wav file
start_times = annotation_file.iloc[:, 6:7].values          #Extract start times, finish times, LM labels
finish_times = annotation_file.iloc[:, 7:8].values         #from respective columns in annotation file object
lm_labels = annotation_file.iloc[:, 1:2].values


def time_converter(times):                                 #Function takes in times as list of strings,
    converted_times = []                                   #e.g. "[[01:13:45:384]]"
    for time in times:                                     #for each time in the list, convert to string,
        time = str(time)[2:-2]                             #remove ends of string to leave "01:13:45:384",
        split = str(time).split(':')                       #split each part on colons into list of parts,
        hrs_to_ms = int(split[0]) * 60 * 60 * 1000         #convert first part (hours) to milliseconds,
        mins_to_ms = int(split[1]) * 60 * 1000             #second part (minutes) to milliseconds,
        secs_to_ms = int(split[2]) * 1000                  #third part (seconds) to milliseconds,
        total_time = hrs_to_ms + mins_to_ms + \
                     secs_to_ms + (int(split[3]) * 10)     #and add all of them together to get time
        converted_times.append(total_time)                 #in milliseconds, and add to list
    return converted_times                                 #return converted times in milliseconds


for file_name in os.listdir(lm_location):                  #For each smaller .wav file
    f = open(target_location + file_name[:-4] +            #create a file with the same name but
             ".lab", "w+")                                 #as a .lab rather than a .wav
    start_time = int(file_name.split("_")[1].split("to")   #Get start time of small .wav in context
                     [0]) * 10000000                       #of overall source .wav time
    finish_time = int(file_name.split("_")[1].split("to")  #Get finish time of small .wav in context
                      [1]) * 10000000                      #of overall source .wav time
    true_start_time = 0                                    #Set start time for .lab in relation to
    num_allowed_lms = 0                                    #source wav and the number of allowed LMs
                                                           #in .lab file both to 0
    sfn = file_name.split(".wav")[0].split("_")[2:]        #Info on LMs (e.g. 'Nib19' or 'Hor_Wor2'
                                                           #is extracted from file name, stored as
                                                           #list of these (e.g. ['Hor', 'Wor2']
    for lm in sfn:                                         #For each different LM in the file name
        if any(i.isdigit() for i in lm):                   #if any contain a digit (e.g. 'Wor2')
            digit = int(lm[-1])                            #extract the digit as a integer,
            num_allowed_lms += digit                       #add this digit to number of allowed LMs
            if any(i.isdigit() for i in lm[:-1]):          #in file (accounting for number in '10s'
                digit_ten = int(lm[-2:-1]) * 10            #as well as single digits
                num_allowed_lms += digit_ten
        else:                                              #if it doesn't have a digit in name (e.g.
            num_allowed_lms += 1                           #'Hor'), add 1 to number of allowed LMs

    i = 0                                                  #Set iterator variable to '0'
    for st, ft, lbl in zip(start_times,                    #For each start time, finish time, LM label
                           finish_times, lm_labels):       #in the annotation file for each LM
        cts = time_converter([st, ft])                     #convert start/finish times to ms (as int)
        if int(cts[0]/10000000)*10000000 >= \
                start_time:                                #If the start and finish times are
            if (int(cts[1]/10000000)*10000000) <= \
                    finish_time:                           #within the time the smaller .wav represents,
                if i == 0:                                 #if its the first of these allowed in the file,
                    if cts[0] > start_time:                #and if its greater than the start time
                        f.write("0 \t" +                   #of the .lab file, write it to file as '0 \t
                                str(cts[0]-start_time) +    #<end of LM local to the file> \t <LM label>
                                "\t" + "Audio" + "\n")
                    true_start_time = start_time           #Update the start time relative to source file
                if num_allowed_lms > 0:                    #If there are still more LMs allowed to be
                    cts[0] = cts[0] - true_start_time      #written to file, calculate start/finish times
                    cts[1] = cts[1] - true_start_time      #relative to current .lab file and write
                    f.write(str(cts[0]) + "\t" +           #to file as <start LM time> \t <end LM time>
                            str(cts[1]) + "\t" +           #\t <LM label.
                            str(lbl)[2:-2] + "\n")
                    num_allowed_lms -= 1                   #Subtract from remaining allowed LMs into
                i += 1                                     #lab file, and increment iterator
    f.close()                                              #Close the file

    f = open(target_location +                             #Reopen the same .lab file for writing
             file_name[:-4] + ".lab", "r")
    lines = f.readlines()                                  #Read in the lines of the file as a list
    f.close()                                              #close the .lab file
    lines_to_insert = {}                                   #create empty dictionary of lines to insert
    for i in range(len(lines)):                            #For each line in the list of lines,
        if i != 0:                                         #if its not the first line,
            pos_1 = int(lines[i - 1].split("\t")[1])       #get the finish time of the previous line
```

```
        pos_2 = int(lines[i].split("\t")[0])            #and the start time of the current line
        if pos_2 > pos_1:                               #If there is a gap between the two times,
            line_to_insert = str(pos_1) + "\t" \
                             + str(pos_2) + "\t" \
                             + "Audio" + "\n"           #create an 'Audio' line to go between them
            lines_to_insert[line_to_insert] = i         #and add to dictionary with line to insert
                                                        #as the key and the position to insert as value
offset = 0                                              #set list offset to 0
for line in lines_to_insert:                            #For each line in the lines to insert dict
    lines.insert(lines_to_insert[line]+offset, line)    #insert line into the list of lines at correct
    offset += 1                                         #position, and increment the offset
if lines:                                               #If the list of lines is not empty,
    if int(lines[-1].split("\t")[1]) < finish_time:     #if the last line's finish time is less than
                                                        #the file's finishing time,
        line_to_insert = str(lines[-1].split("\t")      #add an 'Audio' line that fills this gap
                             [1]) + "\t" + \
                         str(finish_time-start_time) \
                         + "\t" + "Audio" + "\n"        #as <end time of last line> \t <end time of file>
                                                        #\t 'Audio'
        lines.insert(len(lines), line_to_insert)        #and insert this into lines at correct position
f = open(target_location + file_name[:-4]               #Reopen .lab file for writing,
         + ".lab", "w")
lines = "".join(lines)                                  #join these lines in the list as one string,
f.write(lines)                                          #write these to the .lab file,
f.close()                                               #and close the .lab file
```

Code Overview and Changes Between Versions 1 and 2

The full explanation of how the 'LabelCreatorV2.py' code runs can be found below:

1.) The location where we will be placing the .lab files is cleared of any other files to prevent duplicates
2.) The annotation file is read in (only rows that pertain to the source file), and the start times, end times, and LM labels are extracted from this
3.) The program enters a 'for' loop that iterates over each .wav file created in the previous section that continues for the remainder of the program
4.) It then gets the start and finish times that the .wav file contains of the original source file (given in the name of the file), put them in units of 100ns (i.e. multiply values by 10000000) and gets the description of LMs and quantity (e.g. 'Nib2_Gru2_Swo4') from the end of the file name
5.) For each part of the file name LM description (e.g. 'Nib2' being one in the above case), if there is a number at the end of each part ('2' in the above case), convert this to an integer and add this to 'num_allowed_lms'; else, if there is no number after an LM name, add 1 to 'num_allowed_lms'
6.) For each start time, end time, and label for each LM in the annotation file, we convert the times to milliseconds via a 'time_converter' function; if the LM's start and end times fall within the .wav files start and finish times (as determined in step 4), we can write this LM description to our .lab file
7.) If it's the first LM to be written to the file, write it with the format '0', tab, 'length of LM', tab, 'name of LM'); otherwise, check to make sure we can still add more LMs to this file (if 'num_allowed_lms' > 0) and, if we can, add it to the .lab file with format 'start LM time in .wav file', tab, 'end LM time in .wav file', tab, 'name of LM', and subtract 1 from 'num_allowed_lms'
8.) For each pair of lines in each .wav file, determine if there is a 'gap' between two LMs (including 'Audio' here); in other words, if the start time of one LM in a .lab file is greater than the end time of the previous line
9.) If it is, insert an 'Audio' line that covers this in the form 'line 1 end time', tab, 'line 2 start time', tab, 'Audio', while adding an 'Audio' line at the beginning and end of the .lab file if the start time of the first LM line is greater than the file start time, or the end time of the last LM line is less than the file end time

The first version, as can be seen above, created .lab files having not centred the start and end times of each line of the file around their relative times (i.e. they showed the position of each LM's line as the position in the source file and not in the new .wav). This meant that it was harder to determine the locations within each .wav file of LMs given the .lab file if we wanted to manually check to see if they correctly labelled them. Additionally, the original label creator script did not write any 'Audio' gaps into the .lab files. This meant that a lot of the raw audio data could not be used for the feature extraction phase (even though its all 'Audio' data, i.e. non-LMs or LMs we aren't interested in, we still want to train our CNN on this).

FeatureExtraction.py

```
import os
import numpy as np
import matplotlib.pyplot as plt
import scipy.io.wavfile as wvf
from scipy.fftpack import dct
from scipy import signal

file_path = "C:\Program Files\Python36\
ThirdYearProject\Speech_TIMIT\\train\FCJF0"
file_list = os.listdir(file_path)                       #Specifies location of training files
                                                        #for semester 1 tasks and the list of files
                                                        #in this location


for file_name in file_list:                             #For each training file name in the list
    try:                                                #enter 'try' clause in case exception thrown
        file = file_path + "\\" + file_name             #during file reading, specify complete file
        rate, data = wvf.read(file)                     #name, read in file as .wav as data and its
        num_samples = len(data)                         #sampling rate, find the number of samples,
        framesize = int(0.064*rate)                     #compute frame size (# samples in 0.064s),
        overlap = int(0.032 * rate)                     #compute overlap as half the frame size
        total_frames = int(num_samples/overlap)         #and total frames in .wav is worked out

        print("Samples of file " + file_name +          #print number of samples in current file
              " is: " + str(num_samples))
        print("Sampling rate of file: " + str(rate))    #print sampling rate of current file (e.g. 44k)
        print("Time length of file = " +                #print the time duration of current file (in sec)
              str(len(data) / rate) + "s")
        print("Framesize: " + str(framesize) +          #print framesize (in number of samples)
              " samples per frame")
        print("Overlap: " + str(overlap) +              #print the overlap between successive frames
              " samples per frame")
        print("Total frames: " + str(total_frames))     #print total number of frames in current file
        print("\n")

        frames = np.ndarray((total_frames, framesize))  #convert frames to array of size 'frames x framesize'
        dft_matrix = np.ndarray((total_frames, framesize))  #create another 2 identical arrays to be used later
        abs_dft_matrix = np.ndarray((total_frames,
                                     framesize))

        for i in range(total_frames):                   #for each frame in the file
            for j in range(framesize):                  #for each sample in the frame
                if ((i*overlap + j) < num_samples):     #if the sample falls within file boundaries
                    frames[i][j] = data[i*overlap + j]  #add to frames matrix
                else:                                   #else, make spot in frames list a '0'
                    frames[i][j] = 0

        saw_filter_a = signal.waveforms.sawtooth(       #setup a sawtooth filter
            range(len(frames)), width = [0.5])          #(width=0.5 to make them triangular)
        saw_filter_b = signal.waveforms.sawtooth(       #setup a second sawtooth filter
            range(len(frames)), width = [0.6])
        for i in range(total_frames):                   #for each frame in file
            dft_matrix[i] = np.fft.fft(frames[i])       #apply fast fourier transform on frame
            dft_matrix[i] = signal.filtfilt(            #apply triangular filters on FFTed frames
                saw_filter_a, saw_filter_b, dft_matrix[i])
            dft_matrix[i] = dct(dft_matrix[i])          #apply discrete cosine transform on
                                                        #triang filtered frames

            abs_dft_matrix[i] = abs(dft_matrix[i]) * \
                                abs(dft_matrix[i]) / \
                                max(abs(dft_matrix[i]))  #Normalize the data around highest value
                                                         #in the frame
            abs_dft_matrix[i] = np.log10(abs_dft_matrix[i]) #log(base 10) the final frame



        f = open(file_name[:-4] + ".logFBE", "w+")      #create file with same name as training files
                                                        #but with different extension
        f.writelines(str(abs_dft_matrix))               #and write each transformed frame to the file

        t = range(len(abs_dft_matrix))                  #find the number of transformed frames
        plt.plot(t, abs_dft_matrix)                     #Plot the frame numbers against frequencies
        plt.ylabel("Frequency")                         #with y-axis label
        plt.xlabel("Frame number")                      #and x-axis label
        plt.show()                                      #and display the graph

    except Exception as e:                              #in case exception is thrown (e.g. when
        print("Exception thrown as: " + str(e))         #retrieving file, print this exception
        pass                                            #and continue with program without stopping
```

FeatureExtractionV2.py

```
import os
import numpy as np
import scipy.io.wavfile as wvf
from scipy import signal
import pandas as pd
import warnings
warnings.filterwarnings('ignore')


def create_features():                                  #Function extracts features for all files
                                                        #in directory
    tri_size = int(np.floor(512 / reduced_num_dimensions)   #define triangular filter size to be
                   - 1)                                 #size of 'reduced_num_dimensions'
    features = []                                       #create empty list for extracted features
                                                        #of all files
    for i, (file_name) in enumerate(file_names):        #for each file in list of files in directory
        print("File " + str(i+1) + "...")               #print the number of file program is working on
        full_file_name = source_location + "\\" + \
                         file_name                      #get full name of current file
        rate, data = wvf.read(full_file_name)           #read in file as .wav as data and its
        num_samples = len(data)                         #sampling rate, find the number of samples,
        framesize = int(0.064 * rate)                   #compute frame size (# samples in 0.064s),
        overlap = int(0.032 * rate)                     #compute overlap as half the frame size
        total_frames = int(num_samples / overlap)       #and total frames in .wav is worked out

        frames = np.ndarray((total_frames,              #convert frames to array of size 'frames x framesize'
                             framesize))
        dft_matrix = np.ndarray((total_frames,          #create another 2 identical arrays to be used later
                                 framesize))
        abs_dft_matrix = np.ndarray((total_frames,
                                     int(framesize/2)))

        for i in range(total_frames):                   #for each frame in the file
            for j in range(framesize):                  #for each sample in the frame
                if ((i * overlap + j) < num_samples):   #if the sample falls within file boundaries
                    frames[i][j] = \
                        abs(data[i*overlap + j][1]      #add difference in successive frames to frames matrix
                            - data[i*overlap + j][0])
                else:                                   #else, make spot in frames list a '0'
                    frames[i][j] = 0

        tri = signal.triang(M=tri_size)                 #create triangular filter based on specied size
        file_matrices = []                              #create empty list of file matrices
        for i in range(total_frames):                   #for each frame in the file
            dft_matrix[i] = np.fft.fft(frames[i])       #apply fast fourier transform on frame,
            temp = abs(dft_matrix[i]) * \
                   abs(dft_matrix[i]) / \
                   max(abs(dft_matrix[i]))              #normalize the data around highest value,
            temp = np.log10(temp)                       #apply log(base10) on frame,
            abs_dft_matrix[i] = np.split(temp, 2)[0]    #and only take every other value in the frame
            final_dft_matrix = []                       #create new empty list
            for j in range(reduced_num_dimensions):     #for each feature to be extracted
                final_dft_matrix.append(                #apply one of the triangular filters on frame
                    np.dot(abs_dft_matrix[i]
                           [j*tri_size:(j+1)*tri_size],
                           tri.T))
            file_matrices.append(final_dft_matrix)      #and append this to file_matrices
        features.append(file_matrices)                  #Append this set of features for a file to features
    return features                                     #and return extracted features of all files


def create_csv(features, target_location):              #Function takes in features of all .wav files and
                                                        #location in which to place .csvs of feature data
    for i, file in enumerate(features):                 #for each file's feature set
        label_file = open(label_location +              #open its corresponding .lab file
                          file_names[i][:-4] +          #and reads its lines to a list, 'label_file'
                          ".lab").readlines()
        label_lines = []                                #create new empty list of label lines
        for j in range(len(label_file)):                #for each line in the label file,
            start_num = int(label_file[j].split("\t")   #get the start time of the LM in seconds,
                            [0].strip(' ')) / 10000000
            end_num = int(label_file[j].split("\t")     #get the finish time of the LM in seconds,
                          [1].strip(' ')) / 10000000
            label = label_file[j].split("\t")[2].split  #get the label of the LM,
            ("\n")[0].strip(' ')
            label_lines.append([start_num,              #and append these as a list into another list
                                end_num, label])

        if (len(label_file) != 0):                      #if the .lab file isn't empty,
            end_time = int(label_file[-1].split("\t")   #get the local end time of the list in seconds
                           [1])/10000000
        else:                                           #else, its an 'Audio' .lab file (i.e. no LMs)
            end_time = 40                               #and thus 40s long
        num_rows = len(file)                            #Count number of rows (frames) in the .wav file,
        time_step = end_time / num_rows                 #calculate time difference between them,
        time_samples = np.arange(0.0, end_time, time_step)  #and create an array of numbers increasing from
```

```
                                                    #0.0 to finishing time of .wav with 'time_step'
                                                    #difference between successive numbers

        label_arr = []                              #create empty list
        for start_time in time_samples:             #For each time frame in list of time samples,
            start_time = round(start_time, 3)       #round the start time to 3 decimal places,
            finish_time = round(start_time + time_step, 3)  #get finishing time as next number in list,
                                                    #and round this to 3 decimal places
            for line in label_lines:                #For each line in the label file,
                if start_time >= round(line[0], 1): #if the time period of the LM file (between start
                    if finish_time <= round(line[1], 1):  #and finish times) falles between start and end
                        label_arr.append(line[2])   #times of a line of corresponding .lab file,
                        break                       #append that line's label to list of labels
                                                    #for frames, and move onto next time period
                                                    #in time_samples
            if len(label_lines) == 0:               #If there aren't any lines in .lab file,
                label_arr.append("Audio")           #set corresponding time sample label to 'Audio'

        df1 = pd.DataFrame(file)                    #Create DataFrame object from features of .wav file
        df2 = pd.DataFrame(label_arr)               #Create DataFrame object from labels for each feature
        csv_file_name = target_location + \
                        (file_names[i])[:-4] + \
                        ".csv"                      #Open a .csv file with same name as .wav but with
                                                    #.csv file extension instead
        df = pd.concat([df1, df2], axis=1)          #Concat the features and labels horizontally
        df.to_csv(csv_file_name, sep= ',',          #and write this concatenated object to .csv
                  header= False, index=False)       #with no header column or row


source_location = "C:\\Users\\Dan\\Documents\\LMs\\"         #define location of smaller .wav files
label_location = "C:\\Users\\Dan\\Documents\\LMlabels\\"     #define location of corresponding .lab files
file_names = os.listdir(source_location)                    #list of smaller .wav files in directory
target_location = "C:\\Users\\Dan\\Documents\\LMdata\\"      #define location to place new .csvs
reduced_num_dimensions = 225                                #define the number of features to extract from file
frame_time = 0.02                                           #define the time each row (frame) should equate to

features = create_features()                                #create features for every smaller .wav file
for file in os.listdir(target_location):                    #for each file already in location to place .csvs,
    os.remove(target_location + file)                       #remove them from directory
create_csv(features, target_location)                       #and create every .csv from every .wav and .lab file
```

FeatureExtractionV3.py

```
import os
import numpy as np
import scipy.io.wavfile as wvf
from scipy import signal
import pandas as pd
import warnings
warnings.filterwarnings('ignore')

filter_bank_size = 225                                  #define the number of features to extract from file
frame_time_len = 0.02                                   #define the time each row (frame) should equate to
                                                        #(note: includes overlap, so true frame time is 0.01s)


def create_features():                                  #Function extracts features from all files
    tri_size = int(np.floor(512 / filter_bank_size) - 1)    #define triangular filter size to be
                                                        #the size of filter_bank size
    features = []                                       #create empty list for extracted features
                                                        #of all files
    for i, (file_name) in enumerate(file_names):        #for each file in list of files in directory
        print("File " + str(i + 1) + "...")             #print the number of file program is working on
        full_file_name = source_location + "\\" + \
                         file_name                      #get full name of current file
        rate, data = wvf.read(full_file_name)           #read in file as .wav as data and its
        num_samples = len(data)                         #sampling rate, find the number of samples,
        framesize = int(0.064 * rate)                   #compute frame size (# samples in 0.064s),
        overlap = int(0.032 * rate)                     #compute overlap as half the frame size
        total_frames = int(num_samples / overlap)       #and total frames in .wav is worked out

        frames = np.ndarray((total_frames,              #convert frames to array of size 'frames x framesize'
                             framesize))
        dft_matrix = np.ndarray((total_frames,          #create another 2 identical arrays to be used later
                                 framesize))
        abs_dft_matrix = np.ndarray((total_frames,
                                     int(framesize / 2)))

        for i in range(total_frames):                   #for each frame in the file
            for j in range(framesize):                  #for each sample in the frame
                if ((i * overlap + j) < num_samples):   #if the sample falls within file boundaries
                    frames[i][j] = \
                        abs(data[i*overlap + j][1]      #add difference in successive frames to frames matrix
                            - data[i*overlap + j][0])
                else:                                   #else, make spot in frames list a '0'
                    frames[i][j] = 0

        tri = signal.triang(M=tri_size)                 #create triangular filter based on specied size
        file_matrices = []                              #create empty list of file matrices
        for i in range(total_frames):                   #for each frame in the file
            dft_matrix[i] = np.fft.fft(frames[i])       #apply fast fourier transform on frame,
            temp = abs(dft_matrix[i]) * \
                   abs(dft_matrix[i]) / \
                   max(abs(dft_matrix[i]))              #normalize the data around highest value,
            temp = np.log10(temp)                       #apply log(base10) on frame,
            abs_dft_matrix[i] = np.split(temp, 2)[0]    #and only take every other value in the frame
            final_dft_matrix = []                       #create new empty list
            for j in range(filter_bank_size):           #for each feature to be extracted
                final_dft_matrix.append(                #apply one of the triangular filters on frame
                    np.dot(abs_dft_matrix[i]
                           [j * tri_size:(j + 1) * tri_size],
                           tri.T))
            file_matrices.append(final_dft_matrix)      #and append this to file_matrices
        features.append(file_matrices)                  #Append this set of features for a file to features
    return features                                     #and return extracted features of all files


def create_csv(features, target_location):              #Function takes in features of all .wav files and
                                                        #location in which to place .csvs of feature data
    for i, file in enumerate(features):                 #for each file's feature set
        label_file = open(label_location +              #open its corresponding .lab file
                          file_names[i][:-4] +          #and reads its lines to a list, 'label_file'
                          ".lab").readlines()
        label_lines = []                                #create new empty list of label lines
        for j in range(len(label_file)):                #for each line in the label file,
            start_num = int(label_file[j].split("\t")   #get the start time of the LM in seconds,
                            [0].strip(' ')) / 10000000
            end_num = int(label_file[j].split("\t")     #get the finish time of the LM in seconds,
                          [1].strip(' ')) / 10000000
            label = label_file[j].split("\t")[2].split  #get the label of the LM,
            ("\n")[0].strip(' ')
            label_lines.append([start_num,              #and append these as a list into another list
                                end_num, label])

        if (len(label_file) != 0):                      #if the .lab file isn't empty,
            end_time = int(label_file[-1].split("\t")   #get the local end time of the list in seconds
                           [1]) / 10000000
        else:                                           #else, its an 'Audio' .lab file (i.e. no LMs)
            end_time = 40                               #and thus 40s long
```

```
        num_rows = len(file)                                    #Count number of rows (frames) in the .wav file,
        time_step = end_time / num_rows                         #calculate time difference between them,
        time_samples = np.arange(0.0, end_time, time_step)      #and create an array of numbers increasing from
                                                                #0.0 to finishing time of .wav with 'time_step'
                                                                #difference between successive numbers


        label_arr = []                                          #create empty list
        for start_time in time_samples:                         #For each time frame in list of time samples,
            start_time = round(start_time, 3)                   #round the start time to 3 decimal places,
            finish_time = round(start_time + time_step, 3)      #get finishing time as next number in list,
                                                                #and round this to 3 decimal places
            for line in label_lines:                            #For each line in the label file,
                if start_time >= round(line[0], 1):             #if the time period of the LM file (between start
                    if finish_time <= round(line[1], 1):        #and finish times) falles between start and end
                        label_arr.append(line[2])               #times of a line of corresponding .lab file,
                        break                                   #append that line's label to list of labels
                                                                #for frames, and move onto next time period
                                                                #in time_samples
            if len(label_lines) == 0:                           #If there aren't any lines in .lab file,
                label_arr.append("Audio")                       #set corresponding time sample label to 'Audio'

        df1 = pd.DataFrame(file)                                #Create DataFrame object from features of .wav file
        df2 = pd.DataFrame(label_arr)                           #Create DataFrame object from labels for each feature
        csv_file_name = target_location + \
                        (file_names[i])[:-4] + \
                        ".csv"                                  #Open a .csv file with same name as .wav but with
                                                                #.csv file extension instead
        df = pd.concat([df1, df2], axis=1)                      #Concat the features and labels horizontally
        df.to_csv(csv_file_name, sep=',',                       #and write this concatenated object to .csv
                  header=False, index=False)                    #with no header column or row


source_location = "C:\\Users\\Dan\\Documents\\LMs\\"             #define location of smaller .wav files
label_location = "C:\\Users\\Dan\\Documents\\LMlabels\\"         #define location of corresponding .lab files
file_names = os.listdir(source_location)                        #list of smaller .wav files in directory
target_location = "C:\\Users\\Dan\\Documents\\LMdata\\"          #define location to place new .csvs

features = create_features()                                    #create features for every smaller .wav file
for file in os.listdir(target_location):                        #for each file already in location to place .csvs,
    os.remove(target_location + file)                           #remove them from directory
create_csv(features, target_location)                           #and create every .csv from every .wav and .lab file
```

FeatureExtractionV4.py

```
import os
import numpy as np
import scipy.io.wavfile as wvf
from librosa.feature import chroma_stft
from librosa.feature import chroma_cqt
from python_speech_features import mfcc, fbank, logfbank
import pandas as pd
import warnings
warnings.filterwarnings('ignore')

frame_time_len = 0.02                                   #define the time each row (frame) should equate
                                                        #(note: includes overlap, so true frame time is 0.01s)
reduced_dim = 30                                        #define the number of features to extract from file
frame_len = 882                                         #frame len parameter for 2 functions (882 is default)
audio_processing_choice = "logbank"                     #defines choice of feature extraction functions to use



def create_features():                                  #Function extracts features from all files
    print("\nConverting LM data to features...")       #prints to user when the function starts up
    features_list = []                                  #creates an empty list to hold features from all files
    for i, (file_name) in enumerate(file_names):        #for each smaller .wav file in specified directory
        print("File " + str(i+1) + "...")               #prints to use what file extracting features from
        full_file_name = source_location + "\\" + \
                         file_name                      #get full name of file, inc directory
        rate, data = wvf.read(full_file_name)           #read in file as .wav as data and its sampling rate

        if audio_processing_choice == "chroma":         #if feature extraction choice is 'chroma',
            data = np.asarray(                          #process data by flattening data to 1D, taking every
                [float(datum) for datum in              #other value of data, converting each to floats, and
                 data.flatten()[0::2]])                 #converting to a numpy array
            features = chroma_stft(y=data, sr=rate).T   #This modified data is passed to chroma function
                                                        #with sampling rate and result transposed
                                                        #to give ('# frames' x '12 features')
            features = np.repeat(features, 3, axis=1)   #Append several copies of this horizontally to give
                                                        #36 features (enough for CNNv3 to work with)

        elif audio_processing_choice == "cqt":          #else if feature extraction choice is 'cqt',
            data = np.asarray(                          #process data by flattening data to 1D, taking every
                [float(datum) for datum in              #other value of data, converting each to floats, and
                 data.flatten()[0::2]])                 #converting to a numpy array
            features = chroma_cqt(y=data, sr=rate,      #This modified data is passed to cqt function with
                                  n_chroma=reduced_dim).T   #sampling rate and result transposed to give
                                                        #('# frames' x 'reduced_dim features')

        elif audio_processing_choice == "mfcc":         #else if feature extraction choice is 'mfcc',
            features = mfcc(signal=data, samplerate=rate,   #pass .wav data directly with sampling rate
                            winlen=frame_time_len,      #to 'mfcc' function and result is feature vector as
                            winstep=frame_time_len,     #('# frames' x 'reduced_dim features')
                            numcep=reduced_dim,
                            nfilt=reduced_dim*2,
                            nfft= frame_len)

        elif audio_processing_choice == "fbank":
            features = fbank(signal=data, samplerate=rate,  #else if feature extraction choice is 'fbank',
                             winlen=frame_time_len,     #pass .wav data directly with sampling rate
                             winstep=frame_time_len,    #to 'fbank' function and result is feature vector as
                             nfilt=reduced_dim,         #('# frames' x 'reduced_dim features')
                             nfft=frame_len)[0]         #(with only first item from list as this is the numpy
                                                        #array we're interested in; the other being array
                                                        #of energies in each frame)

        else:                                           #else if feature extraction choice is anything else
            features = logfbank(signal=data,            #pass .wav data directly with sampling rate
                                samplerate=rate,        #to 'logfbank' function and result is feature vector
                                winlen=frame_time_len,  #as ('# frames' x 'reduced_dim features')
                                winstep=frame_time_len,
                                nfilt=reduced_dim,
                                nfft=frame_len)
        features_list.append(features)                  #Add the extracted features of current .wav file to
                                                        #list, and return this list after features
    return features_list                                #of all files have been extracted



def create_csv(features, target_location):              #Function takes in features of all .wav files and
                                                        #location in which to place .csvs of feature data
    print("\n\nCreating .csv files...")                 #prints to user when the function starts up
    for i, file in enumerate(features):                 #For each file's feature set
        print("File " + str(i + 1) + "...")             #print to user what .csv it's currently creating
        label_file = open(label_location +              #open its corresponding .lab file
                          file_names[i][:-4] +          #and reads its lines to a list, 'label_file'
                          ".lab").readlines()
        label_lines = []                                #create new empty list of label lines
        for j in range(len(label_file)):                #for each line in the label file,
            start_num = int(label_file[j].split("\t")   #get the start time of the LM in seconds,
```

```
                    [0].strip(' ')) / 10000000
        end_num = int(label_file[j].split("\t")          #get the finish time of the LM in seconds,
                      [1].strip(' ')) / 10000000
        label = label_file[j].split("\t")[2].split       #get the label of the LM,
        ("\n")[0].strip(' ')
        label_lines.append([start_num,                   #and append these as a list into another list
                            end_num, label])

    if (len(label_file) != 0):                           #if the .lab file isn't empty,
        end_time = int(label_file[-1].split("\t")        #get the local end time of the list in seconds
                       [1]) / 10000000
    else:                                                #else, its an 'Audio' .lab file (i.e. no LMs)
        end_time = 40                                    #and thus 40s long
    num_rows = len(file)                                 #Count number of rows (frames) in the .wav file,
    time_step = end_time / num_rows                      #calculate time difference between them,
    time_samples = np.arange(0.0, end_time, time_step)   #and create an array of numbers increasing from
                                                         #0.0 to finishing time of .wav with 'time_step'
                                                         #difference between successive numbers


    label_arr = []                                       #create empty list
    for start_time in time_samples:                      #For each time frame in list of time samples,
        start_time = round(start_time, 3)                #round the start time to 3 decimal places,
        finish_time = round(start_time + time_step, 3)   #get finishing time as next number in list,
                                                         #and round this to 3 decimal places
        for line in label_lines:                         #For each line in the label file,
            if start_time >= round(line[0], 1):          #if the time period of the LM file (between start
                if finish_time <= round(line[1], 1):     #and finish times) falles between start and end
                    label_arr.append(line[2])            #times of a line of corresponding .lab file,
                    break                                #append that line's label to list of labels
                                                         #for frames, and move onto next time period
                                                         #in time_samples
        if len(label_lines) == 0:                        #If there aren't any lines in .lab file,
            label_arr.append("Audio")                    #set corresponding time sample label to 'Audio'

    df1 = pd.DataFrame(file)                             #Create DataFrame object from features of .wav file
    df2 = pd.DataFrame(label_arr)                        #Create DataFrame object from labels for each feature
    csv_file_name = target_location + \
                    (file_names[i])[:-4] + \
                    ".csv"                               #Open a .csv file with same name as .wav but with
                                                         #.csv file extension instead
    df = pd.concat([df1, df2], axis=1)                   #Concat the features and labels horizontally
    df.to_csv(csv_file_name, sep=',',                    #and write this concatenated object to .csv
              header=False, index=False)                 #with no header column or row



source_location = "C:\\Users\\Dan\\Documents\\LMs\\"      #define location of smaller .wav files
label_location = "C:\\Users\\Dan\\Documents\\LMlabels\\"  #define location of corresponding .lab files
file_names = os.listdir(source_location)                 #list of smaller .wav files in directory
target_location = "C:\\Users\\Dan\\Documents\\LMdata\\"   #define location to place new .csvs

features = create_features()                             #create features for every smaller .wav file
for file in os.listdir(target_location):                 #for each file already in location to place .csvs,
    os.remove(target_location + file)                    #remove them from directory
create_csv(features, target_location)                    #and create every .csv from every .wav and .lab file
```

Code Overview and Changes Between Versions 1 and 4

The full explanation of how the 'FeatureExtractionV4.py' code runs can be found below:

1.) The frame time length is set, along with the number of dimensions (i.e. features) to be produced for each frame and a string representing the feature representation choice
2.) The 'create_features' function is called, which enters a 'for' loop of names of .lab files and, for each name, reads in their corresponding .wav files using the scipy.io.wavfile 'read' function to get the sampling rate (44.1kHz) and data from each file
3.) The appropriate feature transform function is called based on the setting of 'audio_processing_choice'; whichever the choice, the function is provided with the data, the sampling rate, and, in the of every function except the one for 'chroma', the frame time length and number of dimensions to be produced
4.) In the case of 'chroma' or 'cqt' feature transformations, the data produced from the wavfile's 'read' function is flattened, every other number taken, converted to floats, and place in a new numpy array before the data goes into its chosen function
5.) This process repeats for every .lab file name, the features from each feature extraction function return being appended to a list, which is then returned to the main program
6.) The 'create_csv' is then passed this list of files' features and enters a 'for' loop for each file's features within this list

7.) For each file's features, read its corresponding .lab file and extract each start time, end time, and label from every line in the file and add them to a list
8.) Create a 'time_samples' list containing time increments with a set step and end time (e.g. if end_time = 10 and step = 0.1, the list will contain [0, 0.1, 0.2, …. , 9.8, 9.9, 10.0])
9.) Each line in the .lab file is examined and, for every pair of adjacent numbers in the 'time_samples' list (e.g. 0.1 and 0.2, or 3.4 and 3.5), it sees if this pair is within the start and end times of a certain line of the .lab file; if it is, then the part of the features matrix corresponding to the time pair of 'time_samples' gets the LM label from the correct part of the '.lab' file, which is then appended to the 'label_arr' list
10.) Once this is done across all number pairs in 'time_samples' to generate a complete list of labels for a given file's features, this is then horizontally concatenated to the features matrix for a given file as a single column, which is then written as a '.csv' with the same name as its .wav and .lab counterpart

The most significant changes between old versions of the feature extraction script was the removal of using our own custom-built log-bank function from semester 1, as we had come across a library that already does this. In essence, we felt that there was no need to use a potentially not-perfect function when another had already been created and had a proven track record of reliability; additionally, as this library ('python_speech_features') that contained this log-bank function also had functions for normal filter bank and MFCC, we decided to implement these functions, while also using several more functions from 'librosa' for chroma features and constant-Q transforms. In other words, the most recent version of the feature extraction script uses prebuilt functions for feature extraction and enables us to chose between different types of feature extraction.

CNNv1.py

```
import pandas as pd
import tensorflow as tf
import os
import numpy as np
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder

data_file_location = "C:\\Users\\Dan\\Documents\\LMdata\\"  #Set location of .csv data files

X_concat, Y_concat = [], []                                #X and Y lists for ALL data
step = 50                                                  #Set 'step' to half-second size
num_output_classes = 5                                     #Set number of classes model is classifying
num_steps = 2000                                           #Number of iterations the model runs on

for file_name in os.listdir(data_file_location):           #For each data file
    data = pd.read_csv(data_file_location + file_name,     #read in the .csv with no header or index
                       header=None, index_col=None)
    for i in range(0, len(data), step):                    #from 0 to (end of file) by 'step'
        X = data.iloc[i:i+step, :-1].values                #read in the X and Y values as two lists
        X_concat.append(X)                                 #and append them to respective 'totals' lists,
        Y = data.iloc[i:i+step, -1:].mode().values         #with Y lists reduced to a single label via
        Y_concat.append(Y)                                 #the 'mode' function

X_concat = np.array(X_concat)                              #Reshape lists into numpy arrays
Y_concat = np.array(Y_concat)
print(np.shape(X_concat))
height = len(X_concat[0])                                  #Gets the height and width of the
width = len(X_concat[0][0])                                #samples in the X 'concat' list
X_concat = np.reshape(                                     #Reshape the X 'concat' from ('# samples' x 'matrix
    X_concat, (len(X_concat),                              #height' x 'matrix width') to ('# samples' x 'matrix
               len(X_concat[0])*len(X_concat[0][0])))      #height * matrix width') for entry into CNN
Y_concat = np.reshape(Y_concat, (len(Y_concat), 1))        #Reshape the Y 'concat' to a 2D of ('# samples' x 1)
Y_concat = Y_concat.flatten()                              #and flatten to a 1D array
Y_concat = LabelEncoder().fit_transform(Y_concat)          #Fit LabelEncoder object to Y 'concat' and transform
                                                           #Y 'concat' to encoded values
X_train, X_test, Y_train, Y_test = \
    train_test_split(X_concat, Y_concat,                   #Split the X and Y 'concat' lists into train and
                     test_size=0.2, random_state=45)       #test partitions based on value for 'test_size'


def build_cnn(features, labels, mode):
    x = features["x"]                                      #Get the 'x' data the CNN will work with
    x_4d = tf.reshape(x, shape=[-1, height, width, 1])     #Reshape from shape (a,b,c) to (a,b,c,1)
    x_4d = tf.cast(x_4d, tf.float32)                       #Change values in 'x' from ints to floats

    conv_1 = tf.layers.conv2d(x_4d, kernel_size=5,         #Creates 1st convolution layer with kernel,
                              filters=32, strides = 1,     #feature maps produced, padding, stride len,
                              padding="valid",             #and activation function as set by parameters
                              activation=None)
    pool_1 = tf.layers.max_pooling2d(conv_1, pool_size=3,  #Creates 1st pooling layer with window size,
                                     strides=3,            #strides, and padding as set by parameters
                                     padding="valid")
    conv_2 = tf.layers.conv2d(pool_1, kernel_size=5,       #Creates 2nd convolution layer with kernel,
                              filters=32, strides=1,       #feature maps produced, padding, stride len,
                              padding="valid",             #and activation function as set by parameters
                              activation=None)
    pool_2 = tf.layers.max_pooling2d(conv_2, pool_size=3,  #Creates 2nd pooling layer with window size,
                                     strides=3,            #strides, and padding as set by parameters
                                     padding="valid")
    flattened = tf.layers.flatten(pool_2)                  #Flatten pooling output onto a 1D tensor

    fc_3 = tf.layers.dense(flattened, 1024,                #Creates 1st FC layer with # of nodes and
                           activation=tf.nn.relu)          #activation function as set by parameters
    drop_3 = tf.layers.dropout(fc_3, rate=0.5,             #Creates 1st dropout layer with dropout
                               training=True)              #rate and when to activate as set by parameters
    fc_4 = tf.layers.dense(drop_3, 100,                    #Creates 2nd FC layer with # of nodes and
                           activation=tf.nn.relu)          #activation function as set by parameters
    drop_4 = tf.layers.dropout(fc_4, rate=0.5,             #Creates 2nd dropout layer with dropout
                               training=True)              #rate and when to activate as set by parameters
    logits = tf.layers.dense(drop_4,                       #Creates output layer with # nodes equal to
                             units=num_output_classes,     #number of classes to classify and with
                             activation=None)              #activation function as set by parameters

    pred_probs = tf.nn.softmax(logits, name="y_pred")      #Gets output probabilities for classes
    pred_cls = tf.argmax(pred_probs, dimension=1)          #Gets most probable class from probs
    if mode == tf.estimator.ModeKeys.PREDICT:              #If predicting, return here with
        return tf.estimator.EstimatorSpec(                 #predicted classes
            mode, predictions=pred_cls)

    loss_op = tf.losses.sparse_softmax_cross_entropy(      #Computed loss via loss function based
        labels=labels, logits=logits)                      #on labels and output layer
    optimizer = tf.train.AdamOptimizer(learning_rate=0.01) #Define optimizer for training
    train_op = optimizer.minimize(                         #Train CNN via using optimizer to
        loss_op, global_step=tf.train.get_global_step())   #minimize the loss
    acc_op = tf.metrics.accuracy(                          #Get accuracy from labels and pred classes
```

```
        labels=labels, predictions=pred_cls)

    estim_specs = tf.estimator.EstimatorSpec(                #Return predicted classes, loss funct,
        mode=mode, predictions=pred_cls, loss=loss_op,       #optimizer, and accuracy of the CNN as
        train_op=train_op,                                   #an EstimatorSpec object
        eval_metric_ops={'accuracy': acc_op})
    return estim_specs


cnn = tf.estimator.Estimator(build_cnn)                      #Setup the CNN object based on 'build_cnn' function

train_input = tf.estimator.inputs.numpy_input_fn(            #Define the training input to CNN
    x={"x": X_train}, y=Y_train, batch_size=len(X_train),    #from output of data preprocessing step
    num_epochs=10, shuffle=True)                             #with set batch size of all data and # train epochs
test_input = tf.estimator.inputs.numpy_input_fn(             #Define the testing input to CNN
    x={"x": X_test}, y=Y_test, batch_size=len(X_test),       #in same way as above
    num_epochs=10, shuffle=True)

cnn.train(input_fn=train_input, steps= num_steps)            #Train CNN on training input data with set # steps
accuracy = cnn.evaluate(input_fn=test_input)                 #Evaluates for accuracy on testing input

print("\n\nAccuracy = " +                                   #Prints accuracy on testing data to user
      str(accuracy['accuracy']*100) + "%")
```

CNNv2.py

```
import pandas as pd
import os
import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from scipy.stats import mode
import numpy as np
from collections import Counter
import random
from time import time

#Hyperparameters - preprocessing
window_size = 40
window_step = 2
lms_to_classify = ["Nibelungen", "Mime", "Jugendkraft",
                   "Grubel", "Sword"]
num_label_types = len(lms_to_classify) + 1
mode_or_mid = "mid"
one_hot_or_label = "label"
train_test_ratio = 0.2

#Hyperparameters - convolution/pooling layers
num_conv_pool_layers = 4
num_filters = [5, 25, 125, 625]
conv_size = [[3,3], [3,3], [3,3], [3,3]]
conv_stride = 1
conv_padding = "same"
conv_activation = tf.nn.relu
pool_size = [2,2]
pool_stride = 2
pool_padding = "valid"

#Hyperparameters - FC layers
num_fc_layers = 2
size_fc_layers = [400, 400, 400, 400]
fc_activation = tf.nn.relu
dropout_rate = 0.1
output_activation = None

#Hyperparameters - network training
optimizer_learning_rate = 0.001
batch_size = 128
epochs = 80
num_steps = 16000
device_used = "GPU"

data_file_location = "C:\\Users\\Dan\\Documents\\LMdata\\"  #Sets location of .csv data files
tf.logging.set_verbosity(tf.logging.INFO)                  #Sets to give progress output
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'                   #Filters out info and warning logs

if device_used == "CPU":                                   #Selects device to use between CPU and GPU
    os.environ["CUDA_VISIBLE_DEVICES"]="-1"



def preprocess_data(data_file_location):                   #Function takes in location of .csvs containing
                                                           #feature data from corresponding .wav files
    X_concat, Y_concat = [], []                            #X and Y lists for ALL data
    for file_name in os.listdir(data_file_location):       #For each data file
        data = pd.read_csv(data_file_location + file_name, #read in the .csv with no header or index
                           header=None, index_col=None)
        for i in range(0, len(data)-window_size,           #from 0 to (end file-window size) by 'window_step'
                       window_step):
            X = data.iloc[i:i+window_size, :-1].values     #read in the X and Y values as two lists
            Y = data.iloc[i:i+window_size, -1:].values
            for a in range(len(Y)):                        #for every position in the Y values
                for b in range(len(Y[a])):                 #for the one item in 'a'
                    if Y[a][b] not in lms_to_classify:     #if not in 'lms_to_classify', reassign to "Audio"
                        Y[a][b] = "Audio"
            if mode_or_mid == "mode":                      #If using mode for determining Y list's label
                Y = mode(Y)[0][0][0]                       #reassign Y values list to single Y mode value
            else:
                Y = Y[int((len(Y)-1)/2)][0]                #else assign to Y to mid value in Y values list
            X_concat.append(X)                             #Add the x and y values lists to overall data lists
            Y_concat.append(Y)
    X_concat = np.array(X_concat)                          #Reshape lists into numpy arrays and flatten the Y one
    Y_concat = np.array(Y_concat).flatten()
    X_concat, Y_concat = data_undersampling(               #Undersamples X and Y data to remove excess
        X_concat, Y_concat)                                #excess 'Audio' frames
    if one_hot_or_label == "label":                        #Use LabelEncoder or OneHotEncoder to transform Y vals
        Y_concat = LabelEncoder().fit_transform(Y_concat)  #from strings into encoded values
    elif one_hot_or_label == "one_hot":
        Y_concat = OneHotEncoder().fit_transform(Y_concat)
    X_train, X_test, Y_train, Y_test = \
        train_test_split(X_concat, Y_concat,               #Split the X and Y 'concat' lists into train
```

```
            test_size=train_test_ratio, random_state=45)    #and split data based on chosen
                                                            #ratio and return them
    return X_train, X_test, Y_train, Y_test


def data_undersampling(X, Y):                               #Function takes in X and Y lists of feature data
    lab_freq = Counter(Y).most_common()                     #Creates dictionary with LM labels and frequencies
    print("Labels and freq in Y: ", lab_freq)               #in Y in descending order and prints this to user
    Z = list(zip(X,Y))                                      #Zips together 2D frames with their single LM
    random.shuffle(Z)                                       #label, converts to a list, and shuffles the list

    X,Y = zip(*Z)                                           #The shuffled 'Z' is then returned to prev state
                                                            #with both X and Y lists now shuffled the same way
    tar_x_len = len(X) - (lab_freq[0][1]-lab_freq[1][1])    #This calculates the amount of samples to have in
                                                            #new 'X' array after excess 'Audio' samples removed
    new_x, new_y = [], []                                   #Creates empty lists for new reduced X and Y lists
    num_most_common_added = 0                               #Initializes variable at 0 for number of most
    i = 0                                                   #common LM added, along with an interator variable

    while(len(new_x) < tar_x_len):                          #While the new array still to add more frames,
        if Y[i] != lab_freq[0][0]:                          #if the frame is not an 'Audio' frame,
            new_x.append(X[i])                              #append X sample and Y sample to new X and Y arrays
            new_y.append(Y[i])
        else:                                               #else if it is an 'Audio' frame,
            if num_most_common_added < lab_freq[1][1]:      #and if havent already added more than most common LM
                new_x.append(X[i])                          #append X sample and Y sample to new X and Y arrays
                new_y.append(Y[i])
                num_most_common_added += 1                  #increment the number of 'Audio' samples included
        i += 1                                              #increment iterator variable
    new_x = np.asarray(new_x)                               #Convert both X and Y lists to numpy arrays
    new_y = np.asarray(new_y)
    new_lab_freq = Counter(new_y).most_common()             #Re-prints the dictionary with LM labels and
    print("New labels and freq in Y (after minimisation): ",#frequencies in Y in descending order (after
          new_lab_freq)                                     #excess 'Audio' samples are removed from X and Y)
    return new_x, new_y                                     #Returns the new X and Y arrays


def build_cnn(features, labels, mode):
    if mode == tf.estimator.ModeKeys.TRAIN:                 #Set variable to enable dropout layers
        is_training = True                                  #to activate if CNN is in training mode
    else:
        is_training = False

    layers = []                                             #Creates empty list to hold each layer of network
                                                            #so layers can be linked together via 'for' loops
    x = features["x"]                                       #Get the 'x' data the CNN will work with
    x_shape = x.get_shape().as_list()                       #Identify its matrix shape as a list
    x_4d = tf.reshape(x, shape=[-1, x_shape[1],
                                x_shape[2], 1])             #Reshape from shape (a,b,c) to (a,b,c,1)
    x_4d = tf.cast(x_4d, tf.float32)                        #Change values in 'x' from ints to floats
    layers.append(x_4d)                                     #Adds 'x' data to layers

    for i in range(num_conv_pool_layers):                   #For however many conv/pool layers are set,
        conv = tf.layers.conv2d(layers[-1],                 #Creates ith convolution layer with kernel,
                                kernel_size=conv_size[i],   #feature maps produced, padding, stride len,
                                filters=num_filters[i],     #and activation function as set by parameters,
                                strides=conv_stride,        #with each layer linked to last layer in 'layers'
                                padding=conv_padding,
                                activation=conv_activation)
        layers.append(conv)                                 #Appends this layer to end of 'layers'
        pool = tf.layers.max_pooling2d(layers[-1],          #Creates ith pooling layer with window size,
                                       pool_size=pool_size, #strides, and padding as set by parameters
                                       strides=pool_stride,
                                       padding=pool_padding)
        layers.append(pool)                                 #Appends this layer to end of 'layers'
    flat = tf.layers.flatten(layers[-1])                    #Flatten pooling output onto a 1D tensor
    layers.append(flat)                                     #Appends this layer to end of 'layers'

    for i in range(num_fc_layers):                          #For however many FC layers are set,
        drop = tf.layers.dropout(layers[-1],                #Creates ith dropout layer with dropout
                                 rate=dropout_rate,         #rate and when to activate as set by parameters
                                 training=is_training)
        layers.append(drop)                                 #Appends this layer to end of 'layers'
        fc = tf.layers.dense(layers[-1],                    #Creates ith FC layer with # of nodes and
                             size_fc_layers[i],             #activation function as set by parameters
                             activation=fc_activation)
        layers.append(fc)                                   #Appends this layer to end of 'layers'
    logits = tf.layers.dense(layers[-1],                    #Creates output layer with # nodes equal to
                             units=num_label_types,         #number of classes to classify and with
                             activation=output_activation)  #activation function as set by parameters

    pred_probs = tf.nn.softmax(logits, name="y_pred")       #Gets output probabilities for classes
    pred_cls = tf.argmax(pred_probs, axis=1)                #Gets most probable class from probs
    if mode == tf.estimator.ModeKeys.PREDICT:               #If predicting, return here with
        return tf.estimator.EstimatorSpec(                  #predicted classes
            mode=mode, predictions=pred_cls)

    loss_op = tf.losses.sparse_softmax_cross_entropy(       #Computed loss via loss function based
        labels=labels, logits=logits)                       #on labels and output layer
```

```
    optimizer = tf.train.AdamOptimizer(                     #Define optimizer for training
        learning_rate=optimizer_learning_rate)
    train_op = optimizer.minimize(                          #Train CNN via using optimizer to
        loss_op, global_step=tf.train.get_global_step())    #minimize the loss
    acc_op = tf.metrics.accuracy(                           #Get accuracy from labels and pred classes
        labels=labels, predictions=pred_cls)
    fns = tf.metrics.false_negatives(                       #Get false negatives, false positives,
        labels=labels, predictions=pred_cls)                #true negatives, true positives,
    fps = tf.metrics.false_positives(                       #precision, and recall from label types
        labels=labels, predictions=pred_cls)                #and predicted class for each input sample
    tns = tf.metrics.true_negatives(
        labels=labels, predictions=pred_cls)
    tps = tf.metrics.true_positives(
        labels=labels, predictions=pred_cls)
    prec = tf.metrics.precision(
        labels=labels, predictions=pred_cls)
    reca = tf.metrics.recall(
        labels=labels, predictions=pred_cls)
    estim_specs = tf.estimator.EstimatorSpec(               #Return predicted classes, loss funct,
        mode=mode, predictions=pred_cls, loss=loss_op,      #optimizer, and accuracy (plus other metrics)
        train_op=train_op,                                  #of the CNN as an EstimatorSpec object
        eval_metric_ops={'accuracy': acc_op, 'fns': fns,
                         'fps': fps, 'tns': tns,
                         'tps': tps, 'prec': prec,
                         'reca': reca})
    return estim_specs



def main():
    print("\n\\\t\\\t\\\tConvolution Neural Network for "   #Prints message on startup of the name
          "Leitmotiv Detection - Version 2\t\\\t\\\t\\\n")  #of the model

    print("\nStage 1: Extracting data from .csv's...\n")   #Prints message to user when Stage 1 starts
    total_time = 0                                          #Initializes variable for total time taken
    start_time = time()                                     #and starts the timer object
    X_train, X_test, Y_train, Y_test = \
        preprocess_data(data_file_location)                 #Preprocess data from all files into
                                                            #X and Y, training and testing partitions

    labels = {}                                             #Creates an empty dictionary
    for lbl in Y_test:                                      #For each label in 'Y_test',
        if lbl in labels:                                   #if it exists in the dictionary with frequency value,
            labels[lbl] += 1                                #add to this value
        else:                                               #else, add to dictionary with frequency = 1
            labels[lbl] = 1
    print("Labels and occurences in Y_test", labels)        #Prints to user the LM labels with their respective
    print("Total labels =", len(Y_test))                    #frequencies, along with total number of labels

    train_input = tf.estimator.inputs.numpy_input_fn(       #Define the training input to CNN
        x={"x": X_train}, y=Y_train,                        #from output of 'preprocess_data'
        batch_size=batch_size, num_epochs=epochs,           #with set batch size and # train epochs
        shuffle=True)
    test_input = tf.estimator.inputs.numpy_input_fn(        #Define the testing input to CNN
        x={"x": X_test}, y=Y_test,                          #in same way as above
        batch_size=batch_size, num_epochs=epochs,
        shuffle=True)

    cnn = tf.estimator.Estimator(build_cnn)                 #Setup the CNN object based on 'build_cnn' function
    stage_one_time = time() - start_time                    #Calculate time taken for Stage 1 to complete
    total_time += stage_one_time                            #and add to cumulative total time taken
    print("\nTime for stage 1:", round(stage_one_time, 2),  #Print both calculated above values to user
          "seconds,", round(total_time, 2), "total time\n")


    print("\nStage 2: Training CNNv2...\n")                 #Prints message to user when Stage 2 starts
    cnn.train(input_fn=train_input, steps= num_steps)       #Train CNN on training input data with set # steps
    stage_two_time = time() - stage_one_time - start_time   #Calculate time taken for Stage 2 to complete
    total_time += stage_two_time                            #and add to cumulative total time taken
    print("\nTime for stage 2:", round(stage_two_time, 2),  #Print both calculated above values to user
          "seconds", round(total_time, 2), "total time")


    print("\n\nStage 3: Determining accuracy...\n")        #Prints message to user when Stage 3 starts
    accuracy = cnn.evaluate(input_fn=test_input)            #Evaluates for accuracy on testing input
    print("\n\nAccuracy =",
          round(accuracy['accuracy']*100,2), "%")
    print("Loss =", accuracy['loss'])                       #Prints accuracy and loss to user

    total = accuracy['fns'] + accuracy['fps'] + \
            accuracy['tns'] + accuracy['tps']               #Calculates total number of predictions made
    correct = accuracy['tns'] + accuracy['tps']             #and number of those deemed 'correct'
    print("Proportion of correctly classified as an "
          "allowed LM: " + str(correct) + "/"               #Prints to user amount of frames classified
          + str(total) + " = " + str(correct/total))        #correctly as being a LM (but not specific type)

    print("\nTrue positives =", accuracy['tps'])            #Prints to user the number of true positives, true
    print("True negatives =", accuracy['tns'])              #negatives, false positives, and false negatives
    print("False positives =", accuracy['fps'])
    print("False negatives =", accuracy['fns'], "\n")
```

```
    f_measure = 2 * ((accuracy['prec']*accuracy['reca'])     #Calculates the F-measure from above values
                     /(accuracy['prec']+accuracy['reca']))
    print("Precision (true pos / all pos guesses) =",       #and prints this (rounded to 3 decimal places,
          accuracy['prec'])                                  #along with the precision and recall to the user
    print("Recall (true pos / all true pos) =",
          accuracy['reca'])
    print("F-measure =", round(f_measure,3))

    stage_three_time = time() - stage_two_time -\
                       stage_one_time - start_time           #Calculate time taken for Stage 3 to complete
    total_time += stage_three_time                           #and add to cumulative total time taken
    print("\nTime for stage 3:",                            #Print both calculated above values to user
          round(stage_three_time, 2),
          "seconds", round(total_time, 2), "total time")


if __name__ == "__main__":                                   #Only run script if called directly (i.e. not
    main()                                                   #as module in other program)
```

CNNv3.py

```
import pandas as pd
import os
import tensorflow as tf
from sklearn.model_selection import train_test_split
from sklearn.preprocessing import LabelEncoder
from sklearn.preprocessing import OneHotEncoder
from scipy.stats import mode
import numpy as np
from collections import Counter
import random
from time import time

#Hyperparameters - preprocessing
window_size = 40
window_step = 2
lms_to_classify = ["Nibelungen", "Mime", "Jugendkraft",
                   "Grubel", "Sword"]
num_label_types = len(lms_to_classify) + 1
mode_or_mid = "mid"
one_hot_or_label = "label"
train_test_ratio = 0.2

#Hyperparameters - convolution/pooling layers
num_conv_pool_layers = 4
num_filters = [5, 25, 125, 625]
conv_size = [[3,3], [3,3], [3,3], [3,3]]
conv_stride = 1
conv_padding = "same"
conv_activation = tf.nn.relu
pool_size = [2,2]
pool_stride = 2
pool_padding = "valid"

#Hyperparameters - FC layers
num_fc_layers = 2
size_fc_layers = [400, 400]
fc_activation = tf.nn.relu
dropout_rate = 0.1
output_activation = None

#Hyperparameters - network training
optimizer_learning_rate = 0.001
batch_size = 128
epochs = 80
num_steps = 16000
device_used = "GPU"

data_file_location = "C:\\Users\\Dan\\Documents\\LMdata\\"  #Sets location of .csv data files
tf.logging.set_verbosity(tf.logging.INFO)                  #Sets to give progress output
os.environ['TF_CPP_MIN_LOG_LEVEL'] = '2'                   #Filters out info and warning logs

if device_used == "CPU":                                   #Selects device to use between CPU and GPU
    os.environ["CUDA_VISIBLE_DEVICES"]="-1"



def preprocess_data(data_file_location):                   #Function takes in location of .csvs containing
                                                           #feature data from corresponding .wav files
    X_concat, Y_concat = [], []                            #X and Y lists for ALL data
    frame_times = []                                       #Create empty frame times list
    for file_name in os.listdir(data_file_location):       #For each data file
        data = pd.read_csv(data_file_location + file_name, #read in the .csv with no header or index
                           header=None, index_col=None)
        for i in range(0, len(data)-window_size,           #from 0 to (end of file-window size) by 'window_step'
                       window_step):
            X = data.iloc[i:i+window_size, :-1].values     #read in the X and Y values as two lists
            Y = data.iloc[i:i+window_size, -1:].values
            frame_times.append(get_frame_time              #append the start time of the current 2D
                               (file_name, i))             #'X' sample system currently working with
            for a in range(len(Y)):                        #for every position in the Y values
                for b in range(len(Y[a])):                 #for the one item in 'a'
                    if Y[a][b] not in lms_to_classify:     #if not in 'lms_to_classify', reassign as "Audio"
                        Y[a][b] = "Audio"
            if mode_or_mid == "mode":                      #If using mode for determining the Y list's label
                Y = mode(Y)[0][0][0]                       #reassign Y values list to a single Y mode value
            else:
                Y = Y[int((len(Y)-1)/2)][0]                #else assign to Y to middle value in Y values list
            X_concat.append(X)                             #Add X values list and Y value to overall data lists
            Y_concat.append(Y)
    X_concat = np.array(X_concat)                          #Reshape lists into numpy arrays and flatten Y one
    Y_concat = np.array(Y_concat).flatten()
    X_concat, Y_concat, fts = data_undersampling(          #Undersamples X and Y data to remove excess
        X_concat, Y_concat, frame_times)                   #excess 'Audio' frames
    le_map = {}                                            #Create empty label encoder dictionary
    if one_hot_or_label == "label":                        #If choice is to use label encoder,
        le = LabelEncoder()                                #create label encoder object,
```

```
        le.fit(Y_concat)                                    #fit it to the Y data,
        le_map = dict(zip(le.classes_,                      #and add mapping of LM labels (keys)
                          le.transform(le.classes_)))       #and number encodings (values)
        Y_concat = le.transform(Y_concat)                   #Transform Y data into encoded values
    elif one_hot_or_label == "one_hot":                     #Else if choice is one hot,
        Y_concat = OneHotEncoder().\
            fit_transform(Y_concat)                         #create OneHotEncoder object, fit to Y
                                                            #and encode Y to one hot values
    X_train, X_test, Y_train, Y_test = \
        train_test_split(X_concat, Y_concat,                #Split the X and Y 'concat' lists into train
            test_size=train_test_ratio, random_state=45)    #and split data based on chosen
                                                            #ratio and return them
    return X_train, X_test, Y_train, Y_test, X_concat, Y_concat, fts, le_map



def get_frame_time(file_name, i):
    file_start_time = int(file_name.split("_")[1].split     #Gets the global start time of .wav file in ms
                          ("to")[0])*1000
    time_offset = (window_size / 2)                         #Computes time offset from half of window size
    frame_time = file_start_time + (i*window_step) + \
                 time_offset                                #Computes exact time of frame in ms
                                                            #with respect to the source .wav
    return frame_time                                       #and returns this



def data_undersampling(X, Y, fts):                          #Function takes in X and Y lists of feature data
                                                            #and the start time of the 2D X data
    lab_freq = Counter(Y).most_common()                     #Creates dictionary with LM labels and frequencies
    print("Labels and freq in Y: ", lab_freq)               #in Y in descending order and prints this to user
    Z = list(zip(X,Y, fts))                                 #Zips together 2D frames with single LM label
    random.shuffle(Z)                                       #and frame time, converts to list, and shuffles it

    X,Y,fts = zip(*Z)                                       #The shuffled 'Z' is then returned to prev state
                                                            #with X/Y lists + frame time now shuffled same way
    tar_x_len = len(X) - (lab_freq[0][1] - lab_freq[1][1])  # his calculates the amount of samples to have in
                                                            #new 'X' array after excess 'Audio' samples removed
    new_x, new_y, new_fts = [], [], []                      #Creates empty lists for new reduced X and Y lists
                                                            #and new frame times
    num_most_common_added = 0                               #Initializes variable at 0 for number of most
    i = 0                                                   #common LM added, along with an interator variable

    while(len(new_x) < tar_x_len):                          #While the new array still to add more frames,
        if Y[i] != lab_freq[0][0]:                          #if the frame is not an 'Audio' frame,
            new_x.append(X[i])                              #append X sample and Y sample to new X and Y arrays
            new_y.append(Y[i])                              #and the sample time to new time samples array
            new_fts.append(fts[i])
        else:                                               #else if it is an 'Audio' frame,
            if num_most_common_added < lab_freq[1][1]:      #and if havent already added more than most common LM
                new_x.append(X[i])                          #append X sample and Y sample to new X and Y arrays
                new_y.append(Y[i])                          #and the sample time to new time samples array
                new_fts.append(fts[i])
                num_most_common_added += 1                  #increment the number of 'Audio' samples included
        i += 1                                              #increment iterator variable
    new_x = np.asarray(new_x)                               #Convert both X and Y lists to numpy arrays
    new_y = np.asarray(new_y)
    new_lab_freq = Counter(new_y).most_common()             #Re-prints the dictionary with LM labels and
    print("New labels and freq in Y (after minimisation): ",#frequencies in Y in descending order (after
          new_lab_freq)                                     #excess 'Audio' samples are removed from X and Y)
    return new_x, new_y, new_fts                            #Returns the new X/Y arrays and frame times



def build_cnn(features, labels, mode):
    if mode == tf.estimator.ModeKeys.TRAIN:                 #Set variable to enable dropout layers
        is_training = True                                  #to activate if CNN is training
    else:
        is_training = False

    layers = []                                             #Creates empty list to hold each layer of network
                                                            #so layers can be linked together via 'for' loops
    x = features["x"]                                       #Get the 'x' data the CNN will work with
    x_shape = x.get_shape().as_list()                       #Identify its matrix shape as a list
    x_4d = tf.reshape(x, shape=[-1, x_shape[1],
                                x_shape[2], 1])             #Reshape from shape (a,b,c) to (a,b,c,1)
    x_4d = tf.cast(x_4d, tf.float32)                        #Change values in 'x' from ints to floats
    layers.append(x_4d)                                     #Adds 'x' data to layers

    for i in range(num_conv_pool_layers):                   #For however many conv/pool layers are set,
        conv = tf.layers.conv2d(layers[-1],                 #Creates ith convolution layer with kernel,
                                kernel_size=conv_size[i],   #feature maps produced, padding, stride len,
                                filters=num_filters[i],     #and activation function as set by parameters,
                                strides=conv_stride,        #with each layer linked to last layer in 'layers'
                                padding=conv_padding,
                                activation=conv_activation)
        layers.append(conv)                                 #Appends this layer to end of 'layers'
        pool = tf.layers.max_pooling2d(layers[-1],          #Creates ith pooling layer with window size,
                                       pool_size=pool_size, #strides, and padding as set by parameters
                                       strides=pool_stride,
                                       padding=pool_padding)
```

```
        layers.append(pool)                                  #Appends this layer to end of 'layers'
    flat = tf.layers.flatten(layers[-1])                     #Flatten pooling output onto a 1D tensor
    layers.append(flat)                                      #Appends this layer to end of 'layers'

    for i in range(num_fc_layers):                           #For however many FC layers are set,
        drop = tf.layers.dropout(layers[-1],                 #Creates ith dropout layer with dropout
                                 rate=dropout_rate,          #rate and when to activate as set by parameters
                                 training=is_training)
        layers.append(drop)                                  #Appends this layer to end of 'layers'
        fc = tf.layers.dense(layers[-1],                     #Creates ith FC layer with # of nodes and
                             size_fc_layers[i],              #activation function as set by parameters
                             activation=fc_activation)
        layers.append(fc)                                    #Appends this layer to end of 'layers'
    logits = tf.layers.dense(layers[-1],                     #Creates output layer with # nodes equal to
                             units=num_label_types,          #number of classes to classify and with
                             activation=output_activation)   #activation function as set by parameters

    pred_probs = tf.nn.softmax(logits, name="y_pred")        #Gets output probabilities for classes
    pred_cls = tf.argmax(pred_probs, axis=1)                 #Gets most probable class from probs
    if mode == tf.estimator.ModeKeys.PREDICT:                #If predicting, return here with
        predictions = {'pred_probs': pred_probs,             #predicted classes and probabilities
                       'pred_cls': pred_cls}
        return tf.estimator.EstimatorSpec(
            mode=mode, predictions=predictions)

    loss_op = tf.losses.sparse_softmax_cross_entropy(        #Computed loss via loss function based
        labels=labels, logits=logits)                        #on labels and output layer
    optimizer = tf.train.AdamOptimizer(                      #Define optimizer for training
        learning_rate=optimizer_learning_rate)
    train_op = optimizer.minimize(                           #Train CNN via using optimizer to
        loss_op, global_step=tf.train.get_global_step())     #minimize the loss
    acc_op = tf.metrics.accuracy(                            #Get accuracy from labels and pred classes
        labels=labels, predictions=pred_cls)
    fns = tf.metrics.false_negatives(                        #Get false negatives, false positives,
        labels=labels, predictions=pred_cls)                 #true negatives, true positives,
    fps = tf.metrics.false_positives(                        #precision, and recall from label types
        labels=labels, predictions=pred_cls)                 #and predicted class for each input sample
    tns = tf.metrics.true_negatives(
        labels=labels, predictions=pred_cls)
    tps = tf.metrics.true_positives(
        labels=labels, predictions=pred_cls)
    prec = tf.metrics.precision(
        labels=labels, predictions=pred_cls)
    reca = tf.metrics.recall(
        labels=labels, predictions=pred_cls)
    estim_specs = tf.estimator.EstimatorSpec(                #Return predicted classes, loss funct,
        mode=mode, predictions=pred_cls, loss=loss_op,       #optimizer, and accuracy (plus other metrics)
        train_op=train_op,                                   #of the CNN as an EstimatorSpec object
        eval_metric_ops={'accuracy': acc_op, 'fns': fns,
                         'fps': fps, 'tns': tns,
                         'tps': tps, 'prec': prec,
                         'reca': reca})
    return estim_specs



def predict_lms(predictions, expected, fts, le_map):
    preds, expects = [], []                                  #Create empty predicts and expects lists
    for pred, expect in zip(predictions, expected):          #For each prediction and expected value,
        preds.append(pred['pred_cls'])                       #append predicted and expected values
        expects.append(expect)                               #to respective lists
    fts, preds, expects = (                                  #Sort each list by ascending frame start
        list(t) for t in zip(                                #time in the same manner
        *sorted(zip(fts, preds, expects))))

    sts, ets, lbls = [], [], []                              #Create empty start/end time and label lists
    lm_predicts, lm_trues = {}, {}                           #Creates empty LM predictions and expected
                                                             #value dictionaries
    window = [0,0]                                           #Initialize window as a list with 2 numbers
    i = 0                                                    #Initialize iterator variable

    while window[1] < (len(preds)-1):                        #While the window hasn't reached end of preds
        if preds[i] == preds[i+1]:                           #if there are two consecutive predictions,
            window[1] += 1                                   #extend the window and
            i += 1                                           #increment the iterator variable
        else:                                                #otherwise, if next prediction is different
            if preds[i] != 0:                                #and is not an 'Audio' prediction,
                sts.append(                                  #append its start time to start times list
                    mill_to_str_time(fts[window[0]]))        #as a string of "hh:mm:ss:msmsms"
                ets.append(                                  #append its end time to end times list
                    mill_to_str_time(fts[window[1]]+2))      #as a string of "hh:mm:ss:msmsms"
                for lm_name in le_map:                       #For each LM label in mapping,
                    if preds[i] == le_map[lm_name]:          #if the prediction equals the label encoding
                        lbls.append(lm_name)                 #append its corresponding label to labels list
                        if lm_name in lm_predicts:           #If the decoded LM label is in dictionary,
                            lm_predicts[lm_name] += 1        #increase its corresponding value (count)
                        else:                                #else, add it to dictionary with count of 1
                            lm_predicts[lm_name] = 1
            window[0] = window[1]+1                          #Set start of window to previous end of window
            window[1] = window[0]                            #and end of window to same value as start
            i = window[0]                                    #Set iterator to start of window
```

```
    df = pd.concat([pd.DataFrame(sts), pd.DataFrame(ets),    #Concatenate lists as DataFrame
                    pd.DataFrame(lbls)], axis=1)             #objects horizontally as DataFrame object
    df.to_csv("C:\\Users\\Dan\\Dropbox\\Uni stuff\\"
              "EE3P\\LM Detections.csv", sep=',',            #Write this DataFrame object as a .csv file
              header=["LM Start Time", "LM End Time",        #as 'LM Detections.csv' with headers
                      "Predicted LM"], index=False)

    new_fts, new_preds, new_expects, new_equals = \
        [], [], [], []                                       #Create empty frame times, predictions, expected
                                                             #values, and equals lists
    for ft in fts:                                           #For each frame time in list,
        new_fts.append(mill_to_str_time(ft))                 #append this as a string of "hh:mm:ss:msmsms"
                                                             #to new list of frame times
    for pred, expect in zip(preds, expects):                 #For each prediction and expected value,
        for lm_name in le_map:                               #for each LM label in mapping,
            if pred == le_map[lm_name]:                      #if the prediction equals the LM name
                new_preds.append(lm_name)                    #append the LM label to prediction list
            if expect == le_map[lm_name]:                    #if the expected value equals the LM name
                new_expects.append(lm_name)                  #append the LM label to expected value list
        if pred == expect:                                   #If the predicted label for frame equals
            new_equals.append("True")                        #the expected label for frame, append 'True'
        else:                                                #to list, else append 'False'
            new_equals.append("False")
    df = pd.concat([pd.DataFrame(new_fts),                   #Concatenate the frame start times,
                    pd.DataFrame(new_preds),                 #predicted LM labels, expected LM labels,
                    pd.DataFrame(new_expects),               #and whether each are the same ('True'/'False')
                    pd.DataFrame(new_equals)], axis=1)       #horizontally as DataFrame objects as
                                                             #one DataFrame object
    df.to_csv("C:\\Users\\Dan\\Dropbox\\Uni stuff\\"         #Write this DataFrame object as a .csv file
              "EE3P\\LM Frame Predictions.csv",sep=',',      #as 'LM Frame Predictions.csv' with headers
              header=["Times (0.2s LM duration)",
                      "Predicted LM", "Expected LM",
                      "Predict == Expected?"], index=False)

    corrects, incorrects = 0, 0                              #Initializes corrects and incorrects variabls
    for i, pred_cls in enumerate(preds):                     #For each predicted frame,
        true_cls = expects[i]                                #get the true frame at the same frame time
        if pred_cls == true_cls:                             #If they are the same, increment corrects,
            corrects += 1                                    #else increment incorrects
        else:
            incorrects += 1
    accur = round((corrects / (corrects + incorrects))       #Calculated the percentage of total frames
                  * 100, 2)                                  #are correctly predicted, rounded to
                                                             #2 decimal places
    print("\nPredicted LMs in source =", lm_predicts)        #Print to the user to dictionary of complete
    print("Prediction accuracy over whole file = " +        #LMs predictions, along with the accuracy of
          str(accur) + "%")                                  #frame predictions on the whole file


def mill_to_str_time(ft):                                    #Function takes in a frame time as ms
    hrs = max(int(ft / 3600000), 0)                          #Extracts hours in frame time,
    ft -= hrs * 3600000                                      #subtracts hours from frame time,
    mins = max(int(ft / 60000), 0)                           #extracts minutes in frame time,
    ft -= mins * 60000                                       #subtracts minutes from frame time
    secs = max(int(ft / 1000), 0)                            #extracts seconds in frame time,
    ft -= secs * 1000                                        #subtracts seconds from frame time,
    return (str(hrs) + ":" + str(mins) + ":"                 #and appends these together with
            + str(secs) + ":" + str(int(ft)))                #remainder of frame time in ms
                                                             #as string with ":" between numbers
                                                             #and returns the string


def main():
    print("\n\\\t\\\t\\\tConvolution Neural Network for "     #Prints message on startup of the name
          "Leitmotiv Detection - Version 3\t\\\t\\\t\\\n")   #of the model

    print("\nStage 1: Extracting data from .csv's...\n")     #Prints message to user when Stage 1 starts
    total_time = 0                                           #Initializes variable for total time taken
    start_time = time()                                      #and starts the timer object
    X_train, X_test, Y_train, Y_test, \
    X, Y, fts, le_map = preprocess_data(data_file_location)  #Preprocess data from all files into
                                                             #X and Y, training and testing partitions

    labels = {}                                              #Creates an empty dictionary
    for lbl in Y_test:                                       #For each label in 'Y_test',
        if lbl in labels:                                    #if it exists in the dictionary with frequency value,
            labels[lbl] += 1                                 #add to this value
        else:                                                #else, add to dictionary with frequency = 1
            labels[lbl] = 1
    print("Labels and occurences in Y_test", labels)         #Prints to user the LM labels with their respective
    print("Total labels =", len(Y_test))                     #frequencies, along with total number of labels

    train_input = tf.estimator.inputs.numpy_input_fn(        #Define the training input to CNN
        x={"x": X_train}, y=Y_train,                         #from output of 'preprocess_data'
        batch_size=batch_size, num_epochs=epochs,            #with set batch size and # train epochs
        shuffle=True)
    test_input = tf.estimator.inputs.numpy_input_fn(         #Define the testing input to CNN
        x={"x": X_test}, y=Y_test,                           #in same way as above
        batch_size=batch_size, num_epochs=epochs,
```

```
        shuffle=True)

    cnn = tf.estimator.Estimator(build_cnn)                 #Setup the CNN object based on 'build_cnn' function
    stage_one_time = time() - start_time                    #Calculate time taken for Stage 1 to complete
    total_time += stage_one_time                            #and add to cumulative total time taken
    print("\nTime for stage 1:", round(stage_one_time, 2),   #Print both calculated above values to user
          "seconds,", round(total_time, 2), "total time\n")


    print("\nStage 2: Training CNNv2...\n")                 #Prints message to user when Stage 2 starts
    cnn.train(input_fn=train_input, steps=num_steps)        #Train CNN on training input data with set # steps
    stage_two_time = time() - stage_one_time - start_time   #Calculate time taken for Stage 2 to complete
    total_time += stage_two_time                            #and add to cumulative total time taken
    print("\nTime for stage 2:", round(stage_two_time, 2),  #Print both calculated above values to user
          "seconds", round(total_time, 2), "total time")


    print("\n\nStage 3: Determining accuracy...\n")        #Prints message to user when Stage 3 starts
    accuracy = cnn.evaluate(input_fn=test_input)            #Evaluates for accuracy on testing input
    print("\n\nAccuracy =",
          round(accuracy['accuracy'] * 100, 2), "%")
    print("Loss =", accuracy['loss'])                       #Prints accuracy and loss to user

    total = accuracy['fns'] + accuracy['fps'] + \
            accuracy['tns'] + accuracy['tps']               #Calculates total number of predictions made
    correct = accuracy['tns'] + accuracy['tps']             #and number of those deemed 'correct'
    print("Proportion of correctly classified as an "
          "allowed LM: " + str(correct) + "/"               #Prints to user amount of frames classified
          + str(total) + " = " + str(correct / total))      #correctly as being a LM (but not specific type)

    print("\nTrue positives =", accuracy['tps'])           #Prints to user the number of true positives, true
    print("True negatives =", accuracy['tns'])              #negatives, false positives, and false negatives
    print("False positives =", accuracy['fps'])
    print("False negatives =", accuracy['fns'], "\n")

    f_measure = 2 * ((accuracy['prec'] * accuracy['reca'])  #Calculates the F-measure from above values
                     /(accuracy['prec'] + accuracy['reca']))
    print("Precision (true pos / all pos guesses) =",      #and prints this (rounded to 3 decimal places,
          accuracy['prec'])                                 #along with the precision and recall to the user
    print("Recall (true pos / all true pos) =",
          accuracy['reca'])
    print("F-measure =", round(f_measure, 3))

    stage_three_time = time() - stage_two_time - \
                       stage_one_time - start_time          #Calculate time taken for Stage 3 to complete
    total_time += stage_three_time                          #and add to cumulative total time taken
    print("\nTime for stage 3:",                           #Print both calculated above values to user
          round(stage_three_time, 2),
          "seconds", round(total_time, 2), "total time")


    print("\n\nStage 4: Making predictions...\n")          #Prints message to user when Stage 4 starts
    total_input = tf.estimator.inputs.numpy_input_fn(       #Define the total input to CNN
        x={"x": X}, y=Y, batch_size=batch_size,             #from output of 'preprocess_data'
        num_epochs=1, shuffle=False)                        #with set batch size and # train epochs

    predictions = cnn.predict(input_fn=total_input)         #Get predicted classifications for each frame
                                                            #from the total data input (train and test)
    predict_lms(predictions, Y, fts, le_map)                #Create prediction files and console output
                                                            #from predictions, true values, frame times,
                                                            #and the label encoder map

    stage_four_time = time() - stage_three_time - \
                      stage_two_time - stage_one_time - \
                      start_time                            #Calculate time taken for Stage 4 to complete
    total_time += stage_four_time                           #and add to cumulative total time taken
    print("\nTime for stage 4:",                           #Print both calculated above values to user
          round(stage_four_time, 2),
          "seconds", round(total_time, 2), "total time")


if __name__ == "__main__":                                  #Only run script if called directly (i.e. not
    main()                                                  #as module in other program)
```

Code Overview

The full explanation of how the 'CNNv1.py' code runs can be found below: Note that much of the 'build_cnn' is influence by an example on GitHub (Damien, 2017):

1.) All hyperparameter values used for this model are set as global values at the beginning of the program, which means they are easy to keep track of regarding their values and also being easier to avoid accidentally modifying the wrong hyperparameter during testing
2.) The 'preprocess_data' function is first called, being described below by steps 3 – 10

3.) For each file in the .csv file location, read it in via the 'panda' library's 'read_csv' function
4.) For each row number of the file from 0 to the end of the file, in steps set as the length of the window produce (so there are is no overlap of rows), the 'X' and 'Y' columns are selected and assigned to variables, with the 'Y' being the last column and 'X' being all other columns
5.) Samples with non-allowed LM labels have their labels changed to 'Audio'
6.) Depending on how a variable is set, the 'Y' is reduced to either the mode or mid of the column vector
7.) The 'X' and 'Y' samples are appended to their respective lists, which are converted to numpy arrays
8.) The data is then undersampled (see Appendix VII for details of why we do this), with the newly undersampled data being returned from the separate 'data_undersampling' function
9.) The 'Y' list is fitted and then encoded via a label encoder object
10.) Both lists are then split into their respective train and test partitions to form: 'X_train', 'X_test', 'Y_train', 'Y_test' with a split ratio of 20% to the test sets and 80% to the train sets, and returned to the 'main' function
11.) The CNN object is then built: this involves reshaping the 'X' input it is to receive to be 4D, casting all numbers in 'X' input to floats, creating convolution and pooling layers with set padding, activation functions, and number of feature maps to produce
12.) Within the build function and after the last pooling layer, the output of this is then flattened, and passed through several FC and dropout layer pairs until it is output as 'N' nodes, where N = number of output classes
13.) The probabilities for each class given a particular input is then calculated, which calculates the 'loss' (difference between true probabilities and predicted probabilities), which trains the network via the Adam optimizer and attempting to minimize the loss function
14.) Accuracy is then specified to be produced, and this information is to be passed back when the evaluation method is called on the CNN object
15.) Once the CNN object is created, the train and test partitions are used to create 'numpy_input_fn' objects that our CNN object can train and test on; these objects refer to 'X_train', 'Y_train' and 'X_test', 'Y_test' for train and test objects, respectively, along with specifying epochs to use
16.) The CNN object is trained on the train 'numpy_input_fn' object via the 'train' method
17.) The accuracy of the model is determined via the CNN object's 'test' object using the test 'numpy_input_fn' object
18.) This accuracy is then printed to the user
19.) The CNN object then calls it's 'predict' method on the total sum of data (both train and test) and returns with predicted values for each frame on every input sample
20.) This is then passed to the 'predict_lms' function, the overview of which is given below

'Predict_lms' function Overview

The full explanation of how the 'predict_lms' function within 'CNNv3.py' works can be found below:

1.) The predictions that the model has made on the complete set (both train and test) via the Estimator object's 'predict' function is passed to 'predict_lms', along with the expected (true) values for the train and test sets, the time occurrences of each of these predictions, and a mapping of different labels to their encoded values (e.g. {'Audio': 0, 'Nibelungen':1,...} states that the 'Audio' LM is encoded as a '0', 'Nibelungen' with a '1', etc.)
2.) The class from each prediction (i.e. what LM the prediction represents) is extracted and placed into a new list
3.) The predictions, true values, and times of occurrence are then all sorted in parallel in order of ascending times of occurrence: this means that each position in one list corresponds to information in the other lists at the same location
4.) A sliding window initially of size 0 then slides along the predictions (which is a list of strings of numbers); if the next number in the list is the same as the previous, extend the window until it encounters a label that is not either itself or '0' ('Audio')
5.) At this point, the complete LM is said to be detected (i.e. as a list of successive, unbroken LM frames of one particular type); the specific label, start and end times of the detected LM is added to lists, while the detected LM type is added to the 'lm_predicts' dictionary (used in a later part)

6.) The window is then repositioned to start 1 position after the end of this detected LM with size 0
7.) Once this window has moved to the end of the predictions and the start, end, and labels of all detected LMs have been added to their respective lists, these are then horizontally concatenated together as DataFrame objects, with a header row added on top for readability, and is then written to the “LM Detections.csv” file
8.) For the next output, we loop through each predicted and expected value in parallel
9.) The mapping for label encodings is then used to convert each label (for both predictions and true values) back to their true LM label type (e.g. 'Mime')
10.) The true and predicted values at each time position are compared; if they are the same, add 'True' to an 'equals' list, else add 'False'
11.) The frame start times, predicted values, expected values, and equals list are all horizontally concatenated again as DataFrame objects with a header row and written to the “LM Frame Predictions.csv” file
12.) Each prediction and true value is then once again checked to see if they are equal to each other at specific time value and, if so, add 1 to 'corrects', else add to 'incorrects'
13.) Accuracy of predictions on the whole dataset is then determined as the number of corrects divided by the number of corrects plus the number of incorrects
14.) The 'lm_predicts' dictionary created earlier is then printed along with the 'accuracy'

It should be noted here that we are predicting on not just the test set but the whole set of data (i.e. both train and test set combined). This is generally speaking not a good idea, as we are asking the system to predict on values that it has already been trained on, which it is much more likely to do well on than data it has never seen before until it has already been trained (i.e. the test set).

However, in this case, we already have an idea of how well it has done by evaluating the accuracy beforehand on the test set exclusively and determined that the system does indeed classify well on unseen data. Therefore, we are able to look at both sets of data in tandem as a way of seeing how the predictions look without fearing that it's predicting much worse on the test set portion than the training set. It also allows us to look at a much broader portion of the data and its predictions rather than just 20% of the original set, which is better for comparing the predicted values and the source annotation file side by side.

Finally, we also should note that there are noticeable gaps in both the 'LM Frame Predictions.csv' file where frames are just missing; this is as a result of the undersampling function taking out many of the 'Audio' frames for the reasons previously discussed, which means that the frame predictions file is only a partially complete predictions file compared with its source data.

Appendix III – Annotation File

Shown below is a series of sections of our annotation file that corresponds to the locations, salience qualities, and names of LMs within our source .wav file of Act 1 of Wagner's *Siegfried* opera, as part of the *Der Rings des Nibelungen* operatic cycle. The names of the LMs are given in the second column, with the salience ratings in the sixth column, and the start and end times of the LMs within the source .wav given in the seventh and eighth columns, respectively:

DW	Ring	Boulez	boulez_siegfried-act1.mp3	Head	1	00:03:08:71	00:03:10:16
DW	Sword	Boulez	boulez_siegfried-act1.mp3	Full	5	00:03:14:80	00:03:20:02
DW	Nibelungen	Boulez	boulez_siegfried-act1.mp3	Full	4	00:03:20:60	00:03:22:22
DW	Nibelungen	Boulez	boulez_siegfried-act1.mp3	Full	5	00:03:22:43	00:03:23:94
DW	Nibelungen	Boulez	boulez_siegfried-act1.mp3	Full	5	00:03:24:05	00:03:25:44
DW	Nibelungen	Boulez	boulez_siegfried-act1.mp3	Full	5	00:03:25:64	00:03:27:03
DW	Nibelungen	Boulez	boulez_siegfried-act1.mp3	Full	5	00:03:27:20	00:03:28:62
DW	Nibelungen	Boulez	boulez_siegfried-act1.mp3	Full	5	00:03:28:74	00:03:30:13
DW	Nibelungen	Boulez	boulez_siegfried-act1.mp3	Full	5	00:03:30:31	00:03:31:90
DW	Nibelungen	Boulez	boulez_siegfried-act1.mp3	Full	4	00:03:31:94	00:03:33:32
DW	Nibelungen	Boulez	boulez_siegfried-act1.mp3	Full	5	00:03:33:40	00:03:34:87
DW	Nibelungen	Boulez	boulez_siegfried-act1.mp3	Full	5	00:03:34:93	00:03:36:42
DW	Nibelungen	Boulez	boulez_siegfried-act1.mp3	Full	5	00:03:36:48	00:03:37:95
DW	Nibelungen	Boulez	boulez_siegfried-act1.mp3	Full	3	00:03:53:63	00:03:54:98
DW	Nibelungen	Boulez	boulez_siegfried-act1.mp3	Full	2	00:03:55:19	00:03:56:62
DW	Nibelungen	Boulez	boulez_siegfried-act1.mp3	Full	2	00:03:56:77	00:03:58:20
DW	Nibelungen	Boulez	boulez_siegfried-act1.mp3	Full	2	00:03:58:32	00:03:59:67
DW	Nibelungen	Boulez	boulez_siegfried-act1.mp3	Full	2	00:03:59:83	00:04:01:19
DW	Nibelungen	Boulez	boulez_siegfried-act1.mp3	Full	5	00:04:06:83	00:04:08:47
DW	Nibelungen	Boulez	boulez_siegfried-act1.mp3	Full	5	00:04:09:12	00:04:10:81
DW	Grubel	Boulez	boulez_siegfried-act1.mp3	Full	5	00:04:12:98	00:04:16:78
DW	Grubel	Boulez	boulez_siegfried-act1.mp3	Full	5	00:04:18:36	00:04:23:14
DW	Sword	Boulez	boulez_siegfried-act1.mp3	Head	5	00:04:23:25	00:04:24:93
DW	Sword	Boulez	boulez_siegfried-act1.mp3	Head	4	00:04:25:00	00:04:26:75
DW	Sword	Boulez	boulez_siegfried-act1.mp3	Head	5	00:04:32:37	00:04:33:58
DW	Sword	Boulez	boulez_siegfried-act1.mp3	Head	4	00:04:33:67	00:04:34:76
DW	Nibelungen	Boulez	boulez_siegfried-act1.mp3	Full	4	00:04:39:69	00:04:41:11
DW	Nibelungen	Boulez	boulez_siegfried-act1.mp3	Full	4	00:04:42:99	00:04:44:42
DW	Sword	Boulez	boulez_siegfried-act1.mp3	Head	2	00:04:46:05	00:04:48:00
DW	Nibelungen	Boulez	boulez_siegfried-act1.mp3	Full	1	00:04:48:81	00:04:50:09
DW	Grubel	Boulez	boulez_siegfried-act1.mp3	Full	5	00:04:54:33	00:04:58:46
DW	Wurm	Boulez	boulez_siegfried-act1.mp3	Full	5	00:05:01:90	00:05:08:56
DW	Wurm	Boulez	boulez_siegfried-act1.mp3	Full	4	00:05:14:89	00:05:21:10
DW	Sword	Boulez	boulez_siegfried-act1.mp3	Head	4	00:05:46:21	00:05:47:28
DW	Sword	Boulez	boulez_siegfried-act1.mp3	Head	3	00:05:53:37	00:05:55:21
DW	Ring	Boulez	boulez_siegfried-act1.mp3	Full	3	00:05:56:34	00:05:59:43
DW	Ring	Boulez	boulez_siegfried-act1.mp3	Full	3	00:05:59:56	00:06:02:59
DW	Sword	Boulez	boulez_siegfried-act1.mp3	Full	4	00:06:04:18	00:06:06:82
DW	Sword	Boulez	boulez_siegfried-act1.mp3	Full	4	00:06:07:03	00:06:09:58
DW	Sword	Boulez	boulez_siegfried-act1.mp3	Full	4	00:06:09:79	00:06:12:62

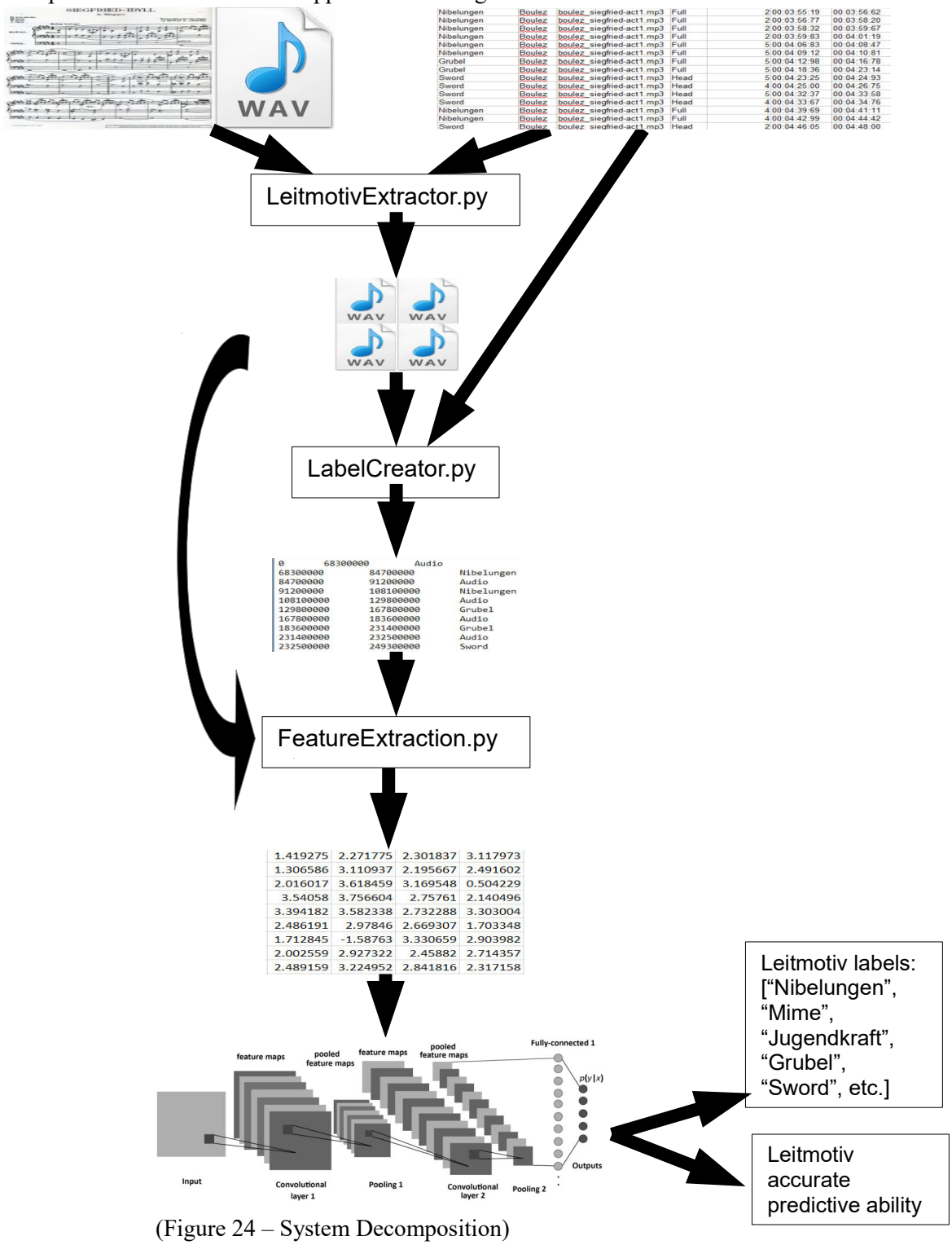
(Figure 22 – Annotation file, part 1)

DW	Horn	Boulez	boulez_siegfried-act1.mp3	Head	3	00:11:18:02	00:11:19:90
DW	Mime	Boulez	boulez_siegfried-act1.mp3	Full	4	00:11:55:97	00:11:56:58
DW	Mime	Boulez	boulez_siegfried-act1.mp3	Full	4	00:11:57:99	00:11:58:55
DW	Mime	Boulez	boulez_siegfried-act1.mp3	Full	4	00:11:59:87	00:12:00:45
DW	Mime	Boulez	boulez_siegfried-act1.mp3	Full	4	00:12:01:89	00:12:02:45
DW	Mime	Boulez	boulez_siegfried-act1.mp3	Full	4	00:12:04:00	00:12:04:62
DW	Mime	Boulez	boulez_siegfried-act1.mp3	Full	4	00:12:06:15	00:12:07:08
DW	Jugendkraft	Boulez	boulez_siegfried-act1.mp3	Full	5	00:12:28:63	00:12:29:86
DW	Jugendkraft	Boulez	boulez_siegfried-act1.mp3	Full	3	00:12:35:18	00:12:36:74
DW	Jugendkraft	Boulez	boulez_siegfried-act1.mp3	Full	3	00:12:41:71	00:12:42:99
DW	Jugendkraft	Boulez	boulez_siegfried-act1.mp3	Full	2	00:12:48:33	00:12:49:56
DW	Mime	Boulez	boulez_siegfried-act1.mp3	Full	5	00:12:57:18	00:12:57:79
DW	Mime	Boulez	boulez_siegfried-act1.mp3	Full	5	00:12:58:48	00:12:59:12
DW	Mime	Boulez	boulez_siegfried-act1.mp3	Full	5	00:12:59:74	00:13:00:37
DW	Mime	Boulez	boulez_siegfried-act1.mp3	Full	5	00:13:00:49	00:13:01:12
DW	Mime	Boulez	boulez_siegfried-act1.mp3	Full	4	00:13:01:18	00:13:01:79
DW	Mime	Boulez	boulez_siegfried-act1.mp3	Full	4	00:13:01:86	00:13:02:47
DW	Mime	Boulez	boulez_siegfried-act1.mp3	Full	4	00:13:02:62	00:13:03:18
DW	Mime	Boulez	boulez_siegfried-act1.mp3	Full	4	00:13:03:24	00:13:03:99
DW	Jugendkraft	Boulez	boulez_siegfried-act1.mp3	Full	4	00:13:12:82	00:13:14:16
DW	Jugendkraft	Boulez	boulez_siegfried-act1.mp3	Full	3	00:13:19:46	00:13:20:83
DW	Longing	Boulez	boulez_siegfried-act1.mp3	Full	4	00:13:56:98	00:14:03:31
DW	Mime	Boulez	boulez_siegfried-act1.mp3	Full	2	00:14:11:87	00:14:12:46
DW	Mime	Boulez	boulez_siegfried-act1.mp3	Full	2	00:14:12:50	00:14:13:15
DW	Mime	Boulez	boulez_siegfried-act1.mp3	Full	1	00:14:13:27	00:14:13:94
DW	Mime	Boulez	boulez_siegfried-act1.mp3	Full	1	00:14:14:01	00:14:14:59
DW	Longing	Boulez	boulez_siegfried-act1.mp3	Full	5	00:14:21:83	00:14:28:60
DW	Longing	Boulez	boulez_siegfried-act1.mp3	Full	4	00:14:38:34	00:14:45:13
DW	Longing	Boulez	boulez_siegfried-act1.mp3	Full	4	00:15:20:58	00:15:26:69
DW	Longing	Boulez	boulez_siegfried-act1.mp3	Full	4	00:15:36:28	00:15:43:49
DW	Longing	Boulez	boulez_siegfried-act1.mp3	Full	4	00:16:03:43	00:16:09:87
DW	Longing	Boulez	boulez_siegfried-act1.mp3	Full	4	00:16:18:27	00:16:24:10
DW	Longing	Boulez	boulez_siegfried-act1.mp3	Full	4	00:16:39:85	00:16:48:33
DW	Mime's Complaint	Boulez	boulez_siegfried-act1.mp3	Full	4	00:17:07:68	00:17:15:72
DW	Siegfried	Boulez	boulez_siegfried-act1.mp3	Full	4	00:18:05:53	00:18:14:80
DW	Walsungs	Boulez	boulez_siegfried-act1.mp3	Full	4	00:18:14:98	00:18:17:78
DW	Wellen	Boulez	boulez_siegfried-act1.mp3	Full	4	00:18:22:28	00:18:23:37
DW	Wellen	Boulez	boulez_siegfried-act1.mp3	Full	4	00:18:23:42	00:18:24:71
DW	Longing	Boulez	boulez_siegfried-act1.mp3	Full	3	00:18:31:69	00:18:36:56
DW	Mime	Boulez	boulez_siegfried-act1.mp3	Full	4	00:19:32:67	00:19:33:28
DW	Mime	Boulez	boulez_siegfried-act1.mp3	Full	4	00:19:33:35	00:19:33:93

(Figure 23 – Annotation file, part 2)

Appendix IV – System Decomposition

Below, we can see a diagrammatic breakdown of the complete system that we have created for this project. It includes all Python scripts used as part of the complete system (not including scripts to create graphs on LM properties), along with showing how data flows through the system, the dependencies between various stages, and what we expect to see as an output. The above system can be summarized as a sequence of steps, whereupon each Python script is run in turn to transform data in some way on its way to training the CNN. A complete breakdown of what happens at each stage can be found below.



(Figure 24 – System Decomposition)

1.) The source file here is a ~1 hour 15 mins .wav file of Act 1 of the 'Siegfried' opera from Wagner's *Der Ring des Nibelungen* operatic cycle

2.) The other file is the annotation file, which here is an extensive list of all LMs that occur within the opera we are concerned with, along with other musical pieces (that we disregard); it's worth noting that these labels are all the doing of the annotator given as 'DW' and not the names given by Wagner himself; the annotation file itself contains the LM names, the start time in the .wav source file, the end time, and its salience rating (used only for graphs), among other information that we don't use

3.) Both of these are then passed through the LeitmotivExtractor.py script to extract LMs from the source .wav (~50 mins worth containing LMs) into smaller .wav files (~40 secs) by referencing the annotation file to determine where are LMs; the size and cut-off points of new .wav files are based on rules we define (more on these later)

4.) The smaller .wav files and the annotation file are then used to create 'label files' with the '.lab' extension via the LabelCreator.py script; each .lab file corresponds to a .wav file with the same name but different extension, and contains the start time, finish time, and names of all LMs in the corresponding .wav file, while being centred around the 'local' time of the file (i.e. first label in each .lab starts at 0) and with gaps being annotated as 'Audio' (i.e. no LM present)

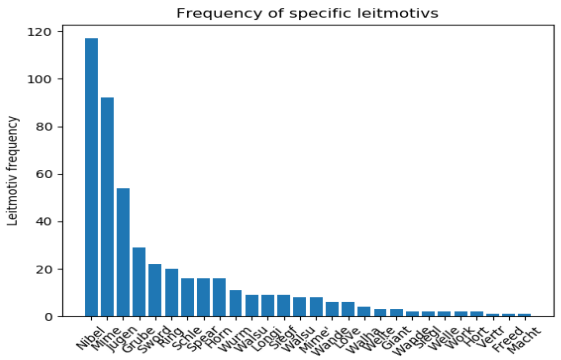
5.) The aforementioned smaller .wav files are then ready to be transformed by a function defined in FeatureExtraction.py for a specific feature representation, usually onto the frequency domain; once this has been done using the data from the .wav's, we then look for the corresponding .lab file and find what label each transformed audio frame should be based on where it falls under on the rows of the label file; the determined label is appended onto the feature vector and added to a .csv data file, again having the same name as .wav and .lab files, containing feature vectors from a .wav file

6.) These .csv data files are now in a form that can be preprocessed by our CNN; 2D matrices are extracted from these .csv's (excluding the label column) and added to the 'X' array, while the labels vector for each 2D matrix had their non-allowed LMs labels changed to 'Audio', with the mid or mode of the vector being found and hence reducing the vector to 1 number, which is added to 'Y'; this is flattened, and both 'X' and 'Y' are split into train and test partitions

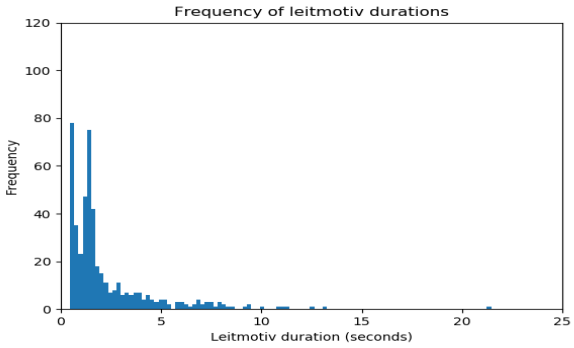
7.) The neural network is setup with a number of convolution, pooling, and FC layers, along with their respective dimensions, kernel/pool window sizes, learning rate, optimizer to use, and so on; the network is then trained on the training partition until an accurate LM predictive ability is able to be found on the test partition

8.) The trained CNN is able to then produce predictions on the locations and lengths of LMs on new or old data, as well as their most likely label

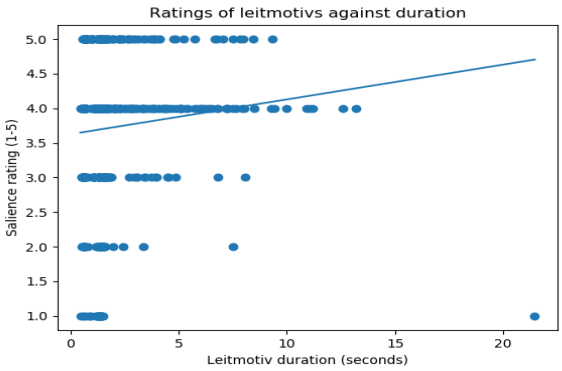
Appendix V – Graphs Produced from LeitmotivStats.py and LMStatsExtraGraphs.py



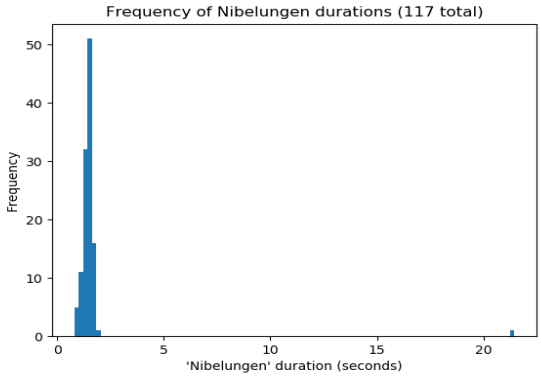
(Graph 3 – Frequency of LMs)



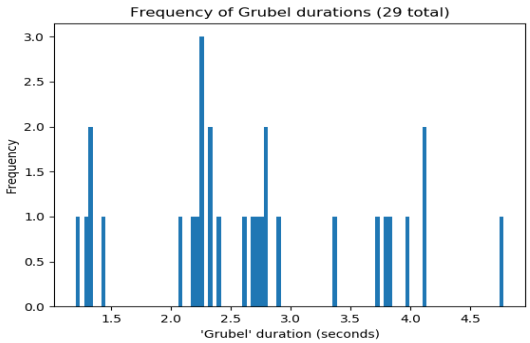
(Graph 4 – Frequency of LM durations)



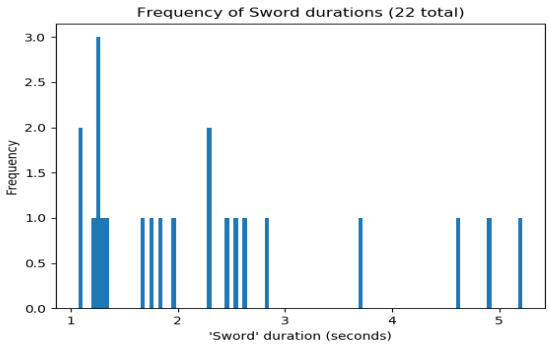
(Graph 5 – Ratings of LMs)



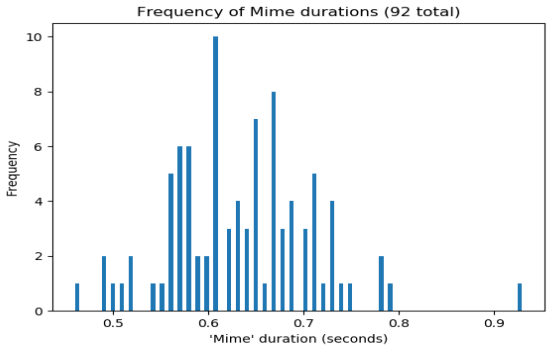
(Graph 6 – Nibelungen durations)



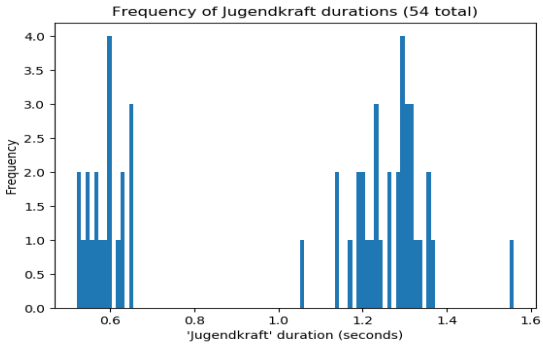
(Graph 7 – Grubel durations)



(Graph 8 – Sword durations)



(Graph 9 – Mime durations)



(Graph 10 – Jugendkraft durations)

Appendix VI – Pseudocode Descriptions

'LeitmotivExtractorV2.py' pseudocode

```
time_converter/rever_time_converter(time):
        time in 'hh:mm:ss' to 'ms' / vice versa
        return conv time

create_lm_file(window, lms):
        for each lm:
                add to name of create file
        extract from source by window times
        write lm file w/ extract data + file name

while not at end of source file:
        for each start, end time in annot file:
                conv lm times w/ time_converter
                count # lms in current window
        if...else statements (as in requires):
                create_lm_file(window, lm in win)
        shift along window ~40 secs
```

(Figure 25 – 'LeitmotivExtractorV2.py' pseudocode)

'LabelCreatorV2.py' pseudocode

```
time_converter(time):
        time in 'hh:mm:ss' to 'ms'
        return conv time

for each LM file:
        get start/end times from LM file name
        count # LMs into lbl from LM file name
        for each start,finish time of LM in file:
                conv lm time w/ time_conversion
                        If lm within start/end times
                        and within # allowed lms
                                centre times around '0'
                                write line in .lbl (see lft)
        for each line in lbl file:
                if time gap between two lines:
                        write 'Audio' line in between
        add 'Audio' lines at start/end of file
```

(Figure 26 – 'LabelCreatorV2.py' pseudocode)

'FeatureExtractionV4.py' pseudocode

```
define params for feature extraction

create_features():
        for each LM file:
                If given param setting == audio
                processing option:
                        process .wav file w/ option
                        add to features array
        return features array

creat_csv(features, target_location):
        for each feature set in features array:
                open .lbl file
                for each line in .lbl file:
                        get start/end/labels in lbl file
                write labels to vector for given
                .lbl file + set frame time (~0.1s)
                transpose and concat vector with
                'features' horizontally
                write concat to .csv
```

(Figure 27 – 'FeatureExtractionV4.py' pseudocode)

Appendix VII – Undersampling Function

One of the problems we had with our data preprocessing methodology was that it created a disproportionate number of 'Audio' labels for inputs (i.e. labels representing no LM or an LM we aren't interested in because it isn't in the top-5 most frequently occurring LMs). This is because we transform many of the labels that correspond to a 2D 'X' input of a non-allowed LM to being an 'Audio' frame; this results in approximately 85% of all labels being 'Audio' inputs (which are later encoded by the label encoder function as a '0') and thus our model was heavily overtraining on these compared to the other labels. Thus, the model would usually give results as seen below:

```
Accuracy = 87.18 %
Loss = 0.57694113
Proportion of correct: 51220.0/58750.0

True positives = 0.0
True negatives = 51220.0

False positives = 0.0
False negatives = 7530.0

Time for stage 3: 4.11 seconds 42.76 total time

Process finished with exit code 0
```

(Figure 28 – CNNv2 output before undersampling)

As we can see above, the model only predicted negatives, which meant that it predicted that every input would be an 'Audio' frame which, although being completely useless as a differentiating tool, leads to a high accuracy due to the high proportion of 'Audio' inputs.

To fix this, we refer to an academic paper addressing this problem: 'A systematic study of the class imbalance problem in convolutional neural networks' (Buda et al., 2017). This paper looks at the pros and cons of data level class balancing methods (among other things) are summarized below:

- Oversampling
  - random minority sampling
  - replicates randomly selected samples from minority classes
  - effective but can lead to overfitting
  - ensures uniform class distribution of each mini-batch and controls the selection of examples from each classification

- Undersampling
  - also results in having the same number of examples in each classification
  - examples are removed randomly from majority classes until all classes have the same number

of examples

- obvious disadvantage is that it disregards a portion of available data
- modifications can be introduced to more carefully select examples to be removed

We thus decide on using undersampling, as creating thousands more of each label (as per oversampling) would most likely lead to heavy overfitting. However, we don't bring all classes level in quantity with each other as we only want to discard some data, but rather only reduce the number of 'Audio' frames until they are equal to the number of the most common LM. This avoids the model training on too many 'Audio' data inputs, but also preserves other potentially useful LM data.

Below, we can see the new output of our CNN model, which shows the model is now predicting 'positives' (i.e. non-'Audio') a lot of the time, both correctly and incorrectly, which shows that our function is working as we intended it to.

```
Accuracy = 20.78 %
Loss = 118315.33
Proportion of correct: 6180.0/10780.0

True positives = 5090.0
True negatives = 1090.0

False positives = 2010.0
False negatives = 2590.0

Time for stage 3: 2.82 seconds 34.32 total time

Process finished with exit code 0
```

(Figure 29 – CNNv2 output after undersampling)

We can also see the results of implementing the undersampling function on the distribution of LM labels on the input data, as shown below:

```
Labels and freq in Y:  [('Audio', 127713), ('Nibelungen', 7789), ('Grubel', 3785), ('Mime', 2877), ('Jugendkraft', 2740), ('Sword', 1970)]
New labels and freq in Y (after minimisation):  [('Audio', 7789), ('Nibelungen', 7789), ('Grubel', 3785), ('Mime', 2877), ('Jugendkraft', 2740), ('Sword', 1970)]
INFO:tensorflow:Using default config.
WARNING:tensorflow:Using temporary folder as model directory: C:\Users\Dan\AppData\Local\Temp\tmp0bv3ohvf
Labels and occurences in Y_test {0: 1558, 4: 1551, 3: 589, 1: 760, 5: 389, 2: 543}
INFO:tensorflow:Using config: {'_model_dir': 'C:\\Users\\Dan\\AppData\\Local\\Temp\\tmp0bv3ohvf', '_tf_random_seed': None, '_save_summary_steps': 100, '_save_checkpoi
Total labels = 5390
```

(Figure 30 – Distribution of LM labels from console)

From the top line, we can see the distribution of LM labels ('Y') for corresponding 2D data inputs ('X') prior to the undersampling function, with the following line showing the complete data (training and testing partitions) after being undersampled. Note that all LM labels were not reduced in quantity, though the 'Audio' labels and their 'X' counterparts were reduced to the size of the next most common LM ('Nibelungen') via randomly removing 'Audio' samples. The next two output lines show the distribution of encoded labels in the testing partition, along with the total number of labels.

Appendix VIII – Implementation of F-measure

One of the downsides with using the accuracy output of our system to determine progress made with system tuning is that it assesses all of the frames that are passed to the system in the evaluation stage. In the case for some systems, this is fine as the 'negatives' (i.e. test data with labels encoded as '0's) are just as important to predict as the positives. In the case of our system, however, this is not the case. We mainly train on 'Audio' data to give the system an idea of what makes a LM, which is better reinforced by also giving it data of what is not a LM. But in the evaluation and prediction stage, we sometimes just want to look at LM frames, and so in calculating the F-measure we find out how capable our system is in this regard.

The F-measure is computed using two measures: precision and recall. These are, respectively, computed by:

- Precision = true positives (number of positives correctly guess as such)

  ---

  true positives + false positives (number of times a frame was guessed to be a positive)

- Recall = true positives (see above)

  ---

  positives (number of times a frame was actually a positive)

Precision can essentially be thought of as the ratio of times a positive guess (i.e. in our case where the system predicts a '1', '2', '3', etc.) was correct to the total number of these positive guesses (i.e. the result of completely removing all negative predictions from consideration). Recall, meanwhile, can be thought of as the ratio of times a positive guess was correct to the total number of actual positive labels (i.e. how many of the positives in the test set the system picked up on).

The F-measure (alternatively called the F1-score) is then computed to be:

F-measure = 2 * ((precision*recall) / (precision+recall))

One of the key problems that the F-measure helps assess is the class imbalance problem. After our data has been undersampled via our ad hoc function, we still have very skewed representations of certain classes: the 'Audio' and the most frequently occuring LM are much more represented than the least common LM, for example, and we don't wish to make them all occur the same amount as this would involve discarding a lot of the data or generating new data, neither of which is desirable. This has the effect of skewing the system to predict negatives (i.e. 'Audio' labels) more often than it should. For instance, in the

version of the system at the time of implementing the F-measure feature we have an accuracy of 50.84% but an F-measure of 0.669.

Here we can see a much higher F-measure than accuracy, which seems to indicate a much higher ability to predict LMs if 'Audio' isn't considered. This is quite important, as at a later point where we ask the system to correctly predict if a frame or set of frames is a specific non-'Audio', the F-measure will be giving a more reliable measure of predictive power than accuracy. However, as at this point we are more concerned with general performance, we will be using the F-measure jointly with accuracy, with accuracy being the primary focus of our efforts to increase as we tune the model. Also note that a very similar accuracy and F-measure indicates a very similar precision and recall, which indicates negatives not having too much influence, though at this point precision is much greater than the recall.

Appendix IX – Complete List of Hyperparameters to Tune

- Feature extraction / audio filtering hyperparameters
  - log size (and position of 'log') after DFT function
  - filter bank size (i.e. number of columns in the data .csv files)
  - time length of the sample (i.e. number of rows in the data .csv files)

- Data preprocessing hyperparameters
  - train/test split ratio of feature vectors and label column
  - window step size for 2D matrix extraction from .csv files
  - number of label types (i.e. number of LM types to classify)
  - method for extraction of label type for given 2D sample (i.e. mid or mode of label vector of a single 2D extracted sample)
  - label encoding type ('one hot', normal label encoding, etc.)

- Convolution and pooling hyperparameters
  - number of convolution and pooling layers (e.g. a '1' for this equals 1 of each convolution and pooling layer
  - number of filters to use for each convolution layer, where number of feature maps produced is the same as the number of filters (equals the number of output channels provided to FC layers)
  - convolution filter kernel size
  - convolution filter stride (i.e. how much the window moves along by at each convolution step before moving down by the same amount)
  - convolution padding type ('full', 'same', or 'valid')
  - convolution activation function
  - pooling window size
  - pooling stride (i.e. how much the pooling window moves along by)
  - pooling padding type ('full', 'same', or 'valid')

- FC layer hyperparameters
  - number of FC layers (i.e. a '2' equals 2 hidden FC layers, not including input or output layers)
  - size of FC layers (i.e. how many nodes in each hidden layer)
  - FC layers activation function (not including output layer)
  - dropout rate used for each dropout layer
  - activation function of the output layer

- Neural network training parameters
  - loss function used to calculate error weight of each input
  - optimizer used to update the weights of the CNN
  - optimizer learning rate
  - train and test batch sizes for training and testing the model
  - number of epochs to train the data on
  - number of steps for which to train the model (with 'None' meaning train forever or until an exception is thrown)

Appendix X – Tables of Hyperparameter Tuning Results

Note that all other settings of the network were kept unchanged at the point of testing a hyperparameter of the CNN, and the performance of each value of the hyperparameter being modified is judged by the accuracy it produced. The tests for each setting was also done 10 times to account for variances in results produced.

Feature Representation	Accuracy (mean)	Accuracy (variance)	Total time in secs (mean)	Total time in secs (variance)
Log-filter bank	39.77%	14.33	34.31s	3.94
Filter bank	25.13%	6.60	33.94s	0.07
MFCC	38.79%	30.81	34.35s	0.40
CQT	36.28%	8.55	31.37s	0.19
Chroma	34.34%	2.63	30.70s	0.30

(Table 6 – Results of feature representation tuning)

# conv/pool layers	Conv # filters	Accuracy (mean)	Accuracy (variance)	Total time in secs (mean)	Total time in secs (variance)
2	5, 25	43.63%	19.75	35.71s	3.89
3	5, 25, 125	45.31%	11.23	34.25s	0.04
4	5, 25, 125, 625	48.89%	16.38	93.47s	0.45

(Table 7 – Results of convolution/pooling layers tuning)

Kernel Size	F-measure	Accuracy (mean)	Accuracy (variance)	Total time in secs (mean)	Total time in secs (variance)
(3,3), (3,3), (3,3), (3,3)	0.669	50.84%	4.78	37.40s	0.11
(9,3), (7,3), (5,3), (3,3)	0.665	50.11%	2.55	39.74s	0.69
(3,3), (5,3), (7,3), (9,3)	0.651	49.53%	4.79	43.28s	0.37
(3,3), (5,3), (7,3), (3,3)	0.631	49.13%	6.58	38.72s	0.09
(4,3), (4,3), (4,3), (4,3)	0.649	50.58%	4.63	39.53s	0.14
(5,3), (5,3), (5,3), (5,3)	0.668	50.12%	2.22	53.47s	2.28

(Table 8 – Results of kernel size tuning)

Dropout rate	F-measure (mean)	Accuracy (mean)	Accuracy (variance)	Total time in secs (mean)	Total time in secs (variance)
0.5	0.669	50.84%	4.78	37.40s	0.11
0.4	0.653	50.64%	1.58	43.22s	15.81
0.3	0.655	50.58%	2.59	42.40s	8.49
0.2	0.667	51.89%	4.95	40.39s	0.15
0.1	0.719	52.98%	10.50	40.38s	0.14
0.05	0.711	49.67%	33.35	40.98s	4.75
0	0.490	44.46%	86.01	42.78s	14.78

(Table 9 – Results of dropout rate tuning)

Dropout rate	F-measure (mean)	Accuracy (mean)	Accuracy (variance)	Total time in secs (mean)	Total time in secs (variance)
0.1, no inp dropout	0.719	52.98%	10.50	40.38s	0.14
0.1, input rate = 0.5	0.783	38.09%	62.56	40.37s	0.40
0.1, input rate = 0.3	0.771	40.78%	18.39	40.18s	0.10
0.1, input rate = 0.1	0.746	46.16%	15.23	40.23s	0.06

(Table 10 – Results of input dropout rate tuning)

Number of FC layers	Size of FC layers	F-measure (mean)	Accuracy (mean)	Accuracy (variance)	Total time in secs (mean)	Total time in secs (variance)
2	100, 50	0.719	52.98%	10.50	40.38s	0.14
2	300, 300	0.717	52.76%	16.75	40.24s	0.06
2	400, 400	0.729	55.11%	4.45	39.94s	0.48
2	500, 500	0.719	54.27%	6.82	40.40s	0.06
2	750, 750	0.678	50.04%	31.45	40.22s	0.25

(Table 11 – Results of FC layer size tuning)

Number of FC layers	Size of FC layers	F-measure (mean)	Accuracy (mean)	Accuracy (variance)	Total time in secs (mean)	Total time in secs (variance)
1	400	0.663	49.15%	22.93	38.19s	0.12
2	400, 400	0.729	55.11%	4.45	39.94s	0.48
3	100, 50, 50	0.617	49.07%	24.50	41.38s	0.05
3	400, 400, 400	0.731	54.38%	1.63	41.12s	0.24
4	400, 400, 400, 400	0.718	52.67%	5.28	42.22s	0.41

(Table 12 – Results of FC number tuning)

Conv activation functions	F-measure (mean)	Accuracy (mean)	Accuracy (variance)	Total time in secs (mean)	Total time in secs (variance)
ReLU	0.729	55.11%	4.45	39.94s	0.48
Sigmoid	0.583	28.59%	1.28	39.21s	0.29
Tanh	0.631	47.94%	45.56	39.32s	0.44
Linear	0.517	47.36%	111.62	39.09s	0.06
SeLU	0.588	49.77%	122.63	39.24s	0.15

(Table 13 – Results of convolution activation function tuning)

FC activation functions	F-measure (mean)	Accuracy (mean)	Accuracy (variance)	Total time in secs (mean)	Total time in secs (variance)
ReLU	0.729	55.11%	4.45	39.94s	0.48
Sigmoid	0.525	43.70%	46.20	39.44s	0.19
Tanh	0.584	46.79%	69.04	39.18s	0.08
Linear	0.419	43.84%	129.10	39.15s	0.19
Selu	0.468	42.24%	85.69	39.13s	0.22

(Table 14 – Results of FC activation function tuning)

Output activation functions	F-measure (mean)	Accuracy (mean)	Accuracy (variance)	Total time in secs (mean)	Total time in secs (variance)
ReLU	0.729	55.11%	4.45	39.94s	0.48
Sigmoid	0.649	48.38%	15.01	39.52s	0.16
Tanh	0.648	52.27%	3.43	39.26s	0.09
Linear	0.772	56.23%	9.99	39.44s	0.09
Selu	0.773	54.71%	6.30	39.29s	0.20

(Table 15 – Results of output activation function tuning)

Optimizer	F-measure (mean)	Accuracy (mean)	Accuracy (variance)	Total time in secs (mean)	Total time in secs (variance)
Adam	0.772	56.23%	9.99	39.44s	0.09
Adagrad	0.540	34.35%	17.95	38.58s	1.33
Gradient Descent	0.395	32.79%	15.91	36.08s	0.20
RMS Prop	0.696	50.12%	3.75	38.89s	0.30

(Table 16 – Results of optimizer function tuning)

Filter-bank size	Window size	Accuracy (mean)	Accuracy (variance)	Total time in secs (mean)	Total time in secs (variance)
30	40	50.84%	4.78	37.40s	0.11
50	50	49.15%	10.43	44.53s	1.66
70	70	50.20%	4.78	54.21s	1.83

(Table 17 – Results of input data shape tuning)

Window Step	F-measure	Accuracy (mean)	Accuracy (variance)	Total time in secs (mean)	Total time in secs (variance)
10	0.772	56.23%	9.99	39.44s	0.09
8	0.798	57.68%	3.82	44.74s	1.32
6	0.814	60.67%	5.62	54.11s	0.24
4	0.819	65.63%	5.28	80.45s	16.11
2	0.829	68.69%	5.29	123.45s	4.04
1	0.830	67.19%	4.52	216.45s	1.68

(Table 18 – Results of window step tuning)

Number of epochs	F-measure	Accuracy (mean)	Accuracy (variance)	Total time in secs (mean)	Total time in secs (variance)
2	0.759	56.00%	11.24	111.02s	1.01
5	0.828	64.95%	6.09	118.37s	2.14
10	0.829	68.69%	5.29	123.45s	4.04
20	0.832	67.84%	4.81	124.30s	0.30
40	0.831	67.55%	4.76	128.68s	2.21

(Table 19 – Results of epochs tuning)

Number of epochs	Number of steps	F-measure	Accuracy (mean)	Accuracy (variance)	Total time in secs (mean)	Total time in secs (variance)
10	1000	0.829	68.69%	5.29	123.45s	4.04
10	2000	0.858	74.26%	12.90	132.98s	1.62
20	4000	0.911	85.06%	5.80	157.20s	6.22
40	8000	0.955	93.11%	0.82	221.54s	91.88
80	16000	0.966	94.83%	1.56	312.06s	7.37

(Table 20 – Results of epochs/steps tuning)

Optimizer learning rate	F-measure	Accuracy (mean)	Accuracy (variance)	Total time in secs (mean)	Total time in secs (variance)
0.0001	0.966	94.83%	1.56	312.06s	7.37
0.001	0.975	96.21%	0.54	310.56s	4.76
0.01	0.834	54.62%	554.99	307.26s	6.63

(Table 21 – Results of optimizer learning rate tuning)

Appendix XI – Dropout Explanation

One of the dangers with training a neural network as complicated and as large as ours is that it is prone to overfitting on data when presented with somewhat limited data (as in our case where we only have just over an hour's worth of audio data, which is hardly enough). This manifests as excellent results on the training data, as it has this memorized, while having subpar results on the test data. In other words, the network hasn't been able to ascertain trends and patterns in the training data, it only memorizes them: this is what we mean by overfitting. To prevent this, we can apply one of several regularization techniques, and the one we chose to use here (primarily because of ease of implementation) is the addition of dropout layers.

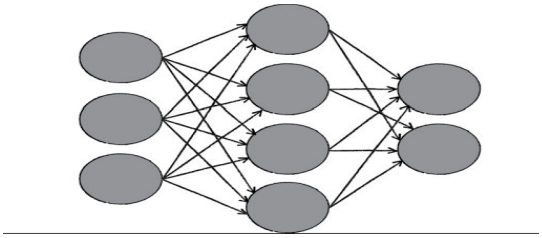
With a dropout layer before an FC layer in TensorFlow, it determines, given a probability 'p', the chance that each node of the next layer is 'turned off' (dropped) at each training iteration, i.e. each node has a 'p' probability of not being switched on for a given training iteration of 'batch size' samples. With certain neurons turned off, the weights associated with the remaining neurons are rescaled to account for these turned off neurons. This forces the network to learn redundant representations of data and thus can't rely exclusively on a certain subset of them for all incoming data as any or all of them may be turned off at any point; hence, the network must learn more general redundant patterns from the data.

Note that these are only activated during training as we wish to use the full power of the network's nodes during evaluation and prediction. The parameter 'p' is set as a hyperparameter (i.e. set by the user) and thus we must find the optimal value. Although our research seems to indicate a value of 0.5 to be optimal, our results from experimentation say otherwise, which are discussed in the main body of the report.

Additionally, a significant improvement was made on the system used by Geoffrey Hinton et al. in their paper 'Improving neural networks by preventing co-adaptation of feature detectors' (2012) by adding a dropout layer prior to the first layer of their model (i.e. just after the input). Hence, if p=0.2 in this case, then each input sample has only an 80% chance on training the network on any given epoch. This seems to have the effect of preventing the model from training too heavily on one class, although the exact reasons were not elaborated upon. Nonetheless, we thought it would be worthwhile in observing the effects of adding this dropout layer with varying rates; the results of which are again discussed in the main body of the report.

Appendix XII – Neural Networks, Gradient Descent and Network Training

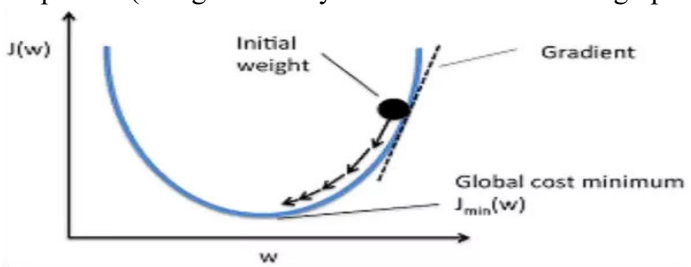
An important concept to understand about neural networks is how they actually 'learn' during the training process. In the case of most implementations of a neural networks (this system included), they make use of variations of gradient descent algorithms. Each of the nodes (or 'neurons') of each layer of a fully-connected feedforward neural network are connected to the nodes/neurons of the next layer, be it another hidden layer or the output layer. Each connection between each node has a 'weight' that corresponds to the magnitude of this connection (i.e. how much of one node's value gets transferred to the next layer's node, with the weight being between 0 and 1). To calculate an output on any neuron, the values of its incoming connections from the previous layer's nodes are summed up and produced based on an activation function. This output from a given nodes acts as an input for nodes in the next layer, and so on.



(Figure 31 – Neural network example)

The final layer is an output layer, which will have values for the output of each of its neurons. These values are interpretted as a single 'output vector'. This is where it is compared with the 'real' values: if the true label for whatever input produced this output vector is 'Nibelungen', for example, which is encoded as a '4' in the network, the resultant 'true' vector for this input is given as [0, 0, 0, 1, 0, 0], with each value corresponding to what the output of each neuron on the output layer should be. As the network is never 100% sure of any predictions, there will be some difference between the predicted values of the output nodes and its true values. This is what we mean by 'loss': a way of interpreting the difference between the true vector and the predicted vector; here, we use the sparse softmax cross entropy function to compute loss.

With this loss computed for a given batch of inputs (i.e. after the above process has been done 'd' times, where 'd' is the batch size, or number of input samples in a batch), these losses are summed up. It is here where the loss is used to update the system. Based on the values of the weights, a graph of the loss and weights values can be plotted (though in reality it's a multidimensional graph rather than just 2D as below):



(Figure 32 – Weight-loss graph example)

By plotting several of these loss values given certain weights, the rate of change between points can be found. The higher the rate of change, the faster the loss is changing between given weights. In the case where this is a larger positive rate of change ('gradient'), it means that the loss is rapidly increasing at the point given the values of the weights of neurons, and vice versa. Through these gradient values, the network can determine which direction to 'move' the weights in: if the network has increased the values of the weights and the gradient is now positive, that means that increasing the weights more would increase the loss further (which is unwanted); hence it should instead reduce the values of the weights.

This gradient is useful in that not only its direction is utilized (i.e. either positive or negative gradient line), but the how steep the line is. If it is particularly steeply increasing at a given point, that means that increasing the weights leads to a large increase in the loss, whereas a shallower gradient means only a small increase in loss given a weights increase. This then informs how much to modify the weights by: in the above case, if the gradient was large, it should decrease the weights by a lot, as the global minima (the minimum value for the loss, i.e. the ideal loss value) is further away and needs a large weights modification to reach it. And using this method of gradient descent to calculate to what values we need to modify the values of the weights, the information is passed to the optimizer algorithm which updates the weights in a way given by the algorithm specifics (a complete discussion of the Adam optimizer algorithm used here is beyond the scope of this explanation however), though as a generalized equation can be seen as:

$$\theta_j := \theta_j - \alpha \frac{\partial}{\partial \theta_j} J(\theta)$$

(Figure 33 – Gradient descent equation)

Where θj = weights, $J(\theta)$ is the objective function (i.e. the 'loss'), and α = learning rate.

Upon doing this, the system has now has weights between neurons that are more likely to give an output prediction vector that is closer to the true prediction vector than before.

Appendix XIII – Epochs and Training Steps

With regards to neural network training, an 'epoch' is a complete training of a neural network on all input training data, which is divided into epochs (i.e. 10 epochs would mean there are 10 copies of all training data that is passed through the network). The more epochs a network has to train on, the more opportunities it has to tune the weights and thus achieves a better accuracy. Hence, we would expect that with an increased number of epochs on a fixed input training size (via finding an optimum window step size), we achieve a higher accuracy on the testing data until ~100 epochs (according to relevant research papers), where it begins to overfit on the data unless we provide more regularization tools to the system beyond dropout layers.

As we would expect from our system, if we refer to the relevant table in Appendix X, we can see an initial increase in epochs causes the weights to be tuned better and results in a higher accuracy (note that in all previous experiments, epochs were set at 10). However, this completely stops increasing with any increases in the number of epochs beyond 10. This is due to how we are holding constant the number of steps, which also needs to increase along with epochs to see any actual difference.The number of steps, epochs, batch size, and input size are all linked by the equation:

samples * epochs = steps * batch size

'Samples' is defined as the number of input 'X' and 'Y' pairs in the input training set (i.e. the length of 'X_train' / 'Y_train'). This is defined for us (with a window step of 2) as 21560 2D training samples. We also have a set batch size of 128, as this is the largest batch size we can use before we run out of available GPU memory in the system's current form. Thus, the equation becomes:

21560 * epochs = steps * 128, steps/epochs = 21560/128 = 168.44

A 'step' (or 'iteration') is defined as one training iteration over a single batch (in this case, a training input of 128 2D matrix samples) where the weights are then updated once. In this case, an 'epoch' can be seen as a single copy of all the training samples: thus, if epochs = 2, we have 21560 * 2 = 43120 actual training data samples flowing through the network in total. This, by definition, equals the total size and quantity of steps taken in training the network (assuming all produced samples are trained upon), hence the above equation.

What we end up with, as shown with the above calculations, is a ratio of 168.44 steps needed for running 1 epoch if we wish to train upon all produced data. If we refer to the relevant table in Appendix X, this is why 20 epochs didn't yield a higher accuracy: we would need 20*168.44 = 3369 steps to train

completely on the data (note that we had steps = 1000 as a constant for all experiments). Indeed, even with 10 epochs, not all of this data is used by the networks, as this would require 1684 steps (but still uses more than 5 epochs, hence the increase in accuracy). The maximum we can set epochs to be (without producing surplus epochs) is 1000/168.44 = 5.94 = 6 epochs. This is also why there isn't as big of a jump in accuracy from 5 to 10 epochs as from 2 to 5.

Appendix XIV – Tables of Final Hyperparameter Settings

In total, we manually set 24 hyperparameter settings for our system and found the best possible settings for each with regards to achieving the best accuracies. In total, we took ~240 individual testing iterations to find the best results. The results of this extensive testing can be seen in the tables below showing the results of tuning, and also the values for variables in the 'FeatureExtractionV4.py' and 'CNNv3.py' Python files found in Appendix II. This is followed by a table of untuned hyperparameters that were not modified during testing, either due to external sources advising a specific value for most models (e.g. a 2x2 pooling window with a stride of 2) or if it was deemed to not have a noticeable impact on the system (e.g. how LMs are encoded from strings):

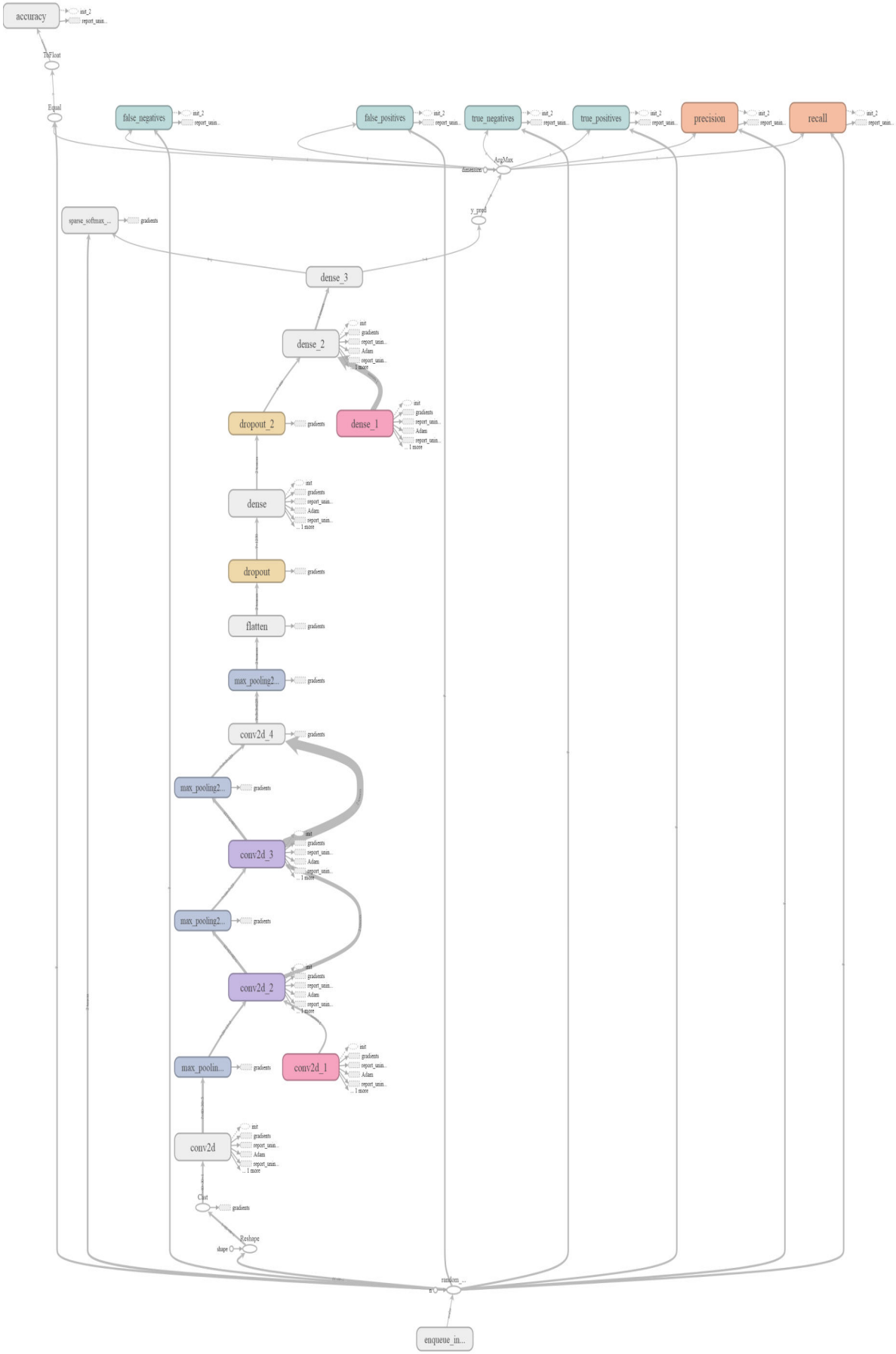
Hyperparameter	**Optimal Setting**
Feature representation	Log-filter bank
Number of convolution and pooling layers	4
Convolution kernel size	(3, 3) for every layer
Dropout rate for FC layers	0.1
Dropout rate on input	None
Size of FC layers	400 neurons per layer
Number of FC layers	2
Convolution activation functions	ReLU
FC activation functions	ReLU
Output activation functions	Linear
Optimizer	Adam
Input data shape	40x30 ('# of audio frames' x '# of features')
Window step	2
Number of epochs	80
Number of steps	16000
Optimizer learning rate	0.001

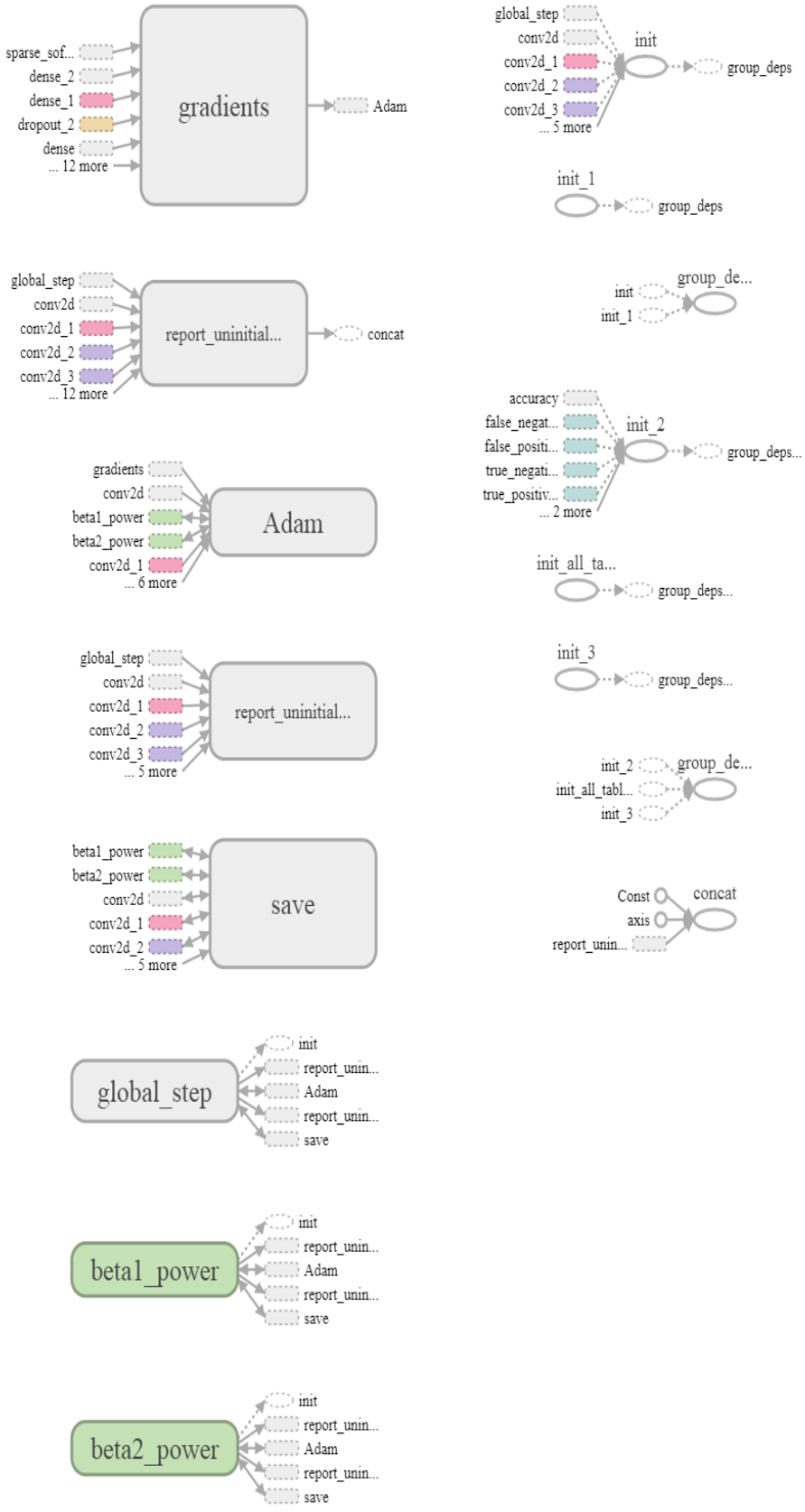
(Table 22 – Final settings of tuned hyperparameters)

Hyperparameter	**Permanent Setting**
Frame time length	0.01 seconds (0.02s including a 0.01s overlap)
Label vector reduction technique	Mid
LM encoder type	LabelEncoder
Train/test split ratio	0.2
Number of feature maps per layer	[5, 25, 125, 625]
Pool window size and stride	(2, 2) with 2 stride
Batch size	128

(Table 23 – Final settings of untuned hyperparameters)

Appendix XV – TensorBoard Graph





(Figure 34 – TensorBoard graph produced from 'CNNv3.py')

Appendix XVI – LM Detections.csv

The LM detections.csv created from the 'CNNv3.py' script can be seen below as the system's predictions of where complete LMs occur and what label they are predicted to be, with the first third of the .csv being the leftmost image, the second third of the .csv being the middle, and the final third being the right. Also, the header for the .csv can be seen on the top of the left image.

LM Start Time	LM End Time	Predicted LM
0:0:1:784	0:0:2:582	Grubel
0:0:3:844	0:0:4:718	Grubel
0:0:5:944	0:0:5:970	Grubel
0:0:5:984	0:0:6:722	Grubel
0:0:47:596	0:0:47:598	Nibelungen
0:1:20:164	0:1:20:166	Grubel
0:1:21:444	0:1:21:854	Nibelungen
0:1:22:12	0:1:22:14	Nibelungen
0:1:22:232	0:1:23:462	Nibelungen
0:1:23:764	0:1:24:102	Nibelungen
0:1:24:404	0:1:25:242	Nibelungen
0:1:25:264	0:1:25:682	Nibelungen
0:1:27:44	0:2:0:146	Nibelungen
0:2:0:244	0:2:0:562	Nibelungen
0:2:0:604	0:2:1:842	Nibelungen
0:2:1:884	0:2:2:802	Nibelungen
0:2:2:844	0:2:5:42	Nibelungen
0:2:5:84	0:2:5:494	Nibelungen
0:2:5:724	0:2:6:322	Nibelungen
0:2:6:344	0:2:6:622	Nibelungen
0:2:6:684	0:2:7:262	Nibelungen
0:2:7:316	0:2:7:318	Nibelungen
0:2:7:324	0:2:7:882	Nibelungen
0:2:40:284	0:2:40:522	Nibelungen
0:2:40:572	0:2:41:162	Nibelungen
0:3:20:104	0:3:21:702	Nibelungen
0:3:21:724	0:3:23:562	Nibelungen
0:3:26:380	0:3:26:382	Nibelungen
0:3:26:704	0:3:26:982	Nibelungen
0:3:27:24	0:3:27:922	Nibelungen
0:4:0:44	0:4:0:46	Nibelungen
0:4:0:336	0:4:0:338	Nibelungen
0:4:1:344	0:4:1:682	Nibelungen
0:4:1:804	0:4:2:154	Nibelungen
0:4:2:584	0:4:3:342	Grubel
0:4:3:664	0:4:4:602	Grubel
0:4:4:612	0:4:5:346	Sword
0:4:6:464	0:4:6:942	Sword
0:4:40:584	0:4:40:862	Nibelungen
0:4:41:184	0:4:41:590	Sword
0:4:41:744	0:4:41:878	Nibelungen
0:4:41:888	0:4:42:2	Nibelungen
0:4:42:844	0:4:43:682	Grubel
0:4:44:248	0:4:44:250	Sword
0:4:47:796	0:4:47:798	Grubel
0:5:25:224	0:5:25:442	Sword
0:5:25:728	0:5:25:730	Nibelungen
0:5:26:664	0:5:27:22	Sword
0:6:0:824	0:6:2:502	Sword
0:6:3:364	0:6:3:366	Mime
0:6:4:136	0:6:4:138	Nibelungen
0:6:5:476	0:6:5:478	Nibelungen
0:6:40:24	0:6:40:26	Mime
0:6:41:816	0:6:41:818	Nibelungen
0:7:20:596	0:7:20:598	Nibelungen
0:7:22:532	0:7:22:534	Jugendkraft
0:7:24:256	0:7:24:258	Nibelungen
0:7:24:288	0:7:24:290	Nibelungen
0:8:1:0	0:8:1:2	Nibelungen
0:8:4:704	0:8:4:706	Nibelungen
0:8:5:384	0:8:5:386	Nibelungen
0:8:7:784	0:8:7:786	Mime
0:8:40:664	0:8:40:902	Jugendkraft
0:8:40:944	0:8:41:702	Jugendkraft
0:8:42:124	0:8:42:126	Jugendkraft
0:8:42:244	0:8:42:502	Jugendkraft
0:8:42:764	0:8:43:42	Jugendkraft
0:8:43:304	0:8:43:422	Jugendkraft
0:8:43:444	0:8:43:942	Jugendkraft
0:8:43:964	0:8:44:202	Jugendkraft
0:8:44:224	0:8:45:122	Jugendkraft
0:8:45:364	0:8:45:378	Nibelungen
0:8:45:384	0:8:46:162	Jugendkraft
0:9:20:624	0:9:20:842	Jugendkraft
0:9:21:124	0:9:21:382	Jugendkraft
0:9:21:404	0:9:22:182	Jugendkraft
0:9:22:744	0:9:23:2	Jugendkraft
0:9:23:24	0:9:24:902	Jugendkraft
0:9:24:904	0:9:25:234	Mime
0:9:25:444	0:9:26:702	Mime
0:9:26:724	0:9:27:2	Mime
0:9:27:24	0:9:27:162	Mime
0:9:27:184	0:9:27:302	Mime
0:10:1:484	0:10:1:742	Jugendkraft
0:10:1:828	0:10:1:830	Nibelungen
0:10:2:40	0:10:2:42	Nibelungen
0:10:2:344	0:10:2:442	Mime
0:10:2:644	0:10:2:802	Mime
0:10:2:936	0:10:4:142	Mime
0:10:4:164	0:10:4:574	Mime
0:10:4:604	0:10:5:322	Mime
0:10:5:644	0:10:5:942	Mime
0:10:5:964	0:10:6:242	Mime
0:10:6:244	0:10:6:482	Jugendkraft
0:10:7:104	0:10:7:462	Mime
0:10:7:484	0:10:7:862	Mime
0:10:40:124	0:10:40:742	Mime
0:10:40:764	0:10:41:22	Mime
0:10:41:144	0:10:41:282	Mime
0:10:41:444	0:10:41:542	Mime
0:10:41:704	0:10:41:822	Mime
0:10:41:984	0:10:42:302	Mime
0:10:45:756	0:10:45:758	Nibelungen
0:10:46:692	0:10:46:694	Mime
0:11:22:772	0:11:22:774	Mime
0:11:26:648	0:11:26:650	Nibelungen
0:11:27:196	0:11:27:310	Mime
0:11:27:592	0:11:27:686	Mime
0:11:27:968	0:11:28:82	Mime
0:11:28:356	0:11:28:482	Mime
0:11:28:780	0:11:28:898	Mime
0:11:29:196	0:11:29:354	Mime
0:12:11:304	0:12:11:562	Jugendkraft
0:12:11:756	0:12:11:758	Mime
0:12:12:624	0:12:12:922	Jugendkraft
0:12:13:924	0:12:14:182	Jugendkraft
0:12:47:244	0:12:47:502	Jugendkraft
0:12:48:652	0:12:48:654	Mime
0:12:49:24	0:12:49:142	Mime
0:12:49:284	0:12:49:398	Mime
0:12:49:524	0:12:50:382	Mime
0:12:51:948	0:12:51:950	Jugendkraft
0:12:52:144	0:12:52:422	Jugendkraft
0:12:52:592	0:12:52:594	Jugendkraft
0:12:52:700	0:12:52:702	Jugendkraft
0:12:53:484	0:12:53:634	Jugendkraft
0:12:53:648	0:12:53:742	Jugendkraft
0:13:34:56	0:13:34:58	Nibelungen
0:14:7:964	0:14:8:502	Mime
0:14:8:968	0:14:8:970	Mime
0:14:10:812	0:14:10:814	Nibelungen
0:16:49:908	0:16:49:910	Mime
0:16:53:488	0:16:53:490	Mime
0:18:11:648	0:18:11:650	Mime
0:18:11:952	0:18:11:954	Nibelungen
0:18:48:984	0:18:48:986	Jugendkraft
0:18:49:704	0:18:49:706	Jugendkraft
0:18:51:848	0:18:51:850	Nibelungen
0:18:52:976	0:18:52:978	Jugendkraft
0:18:53:256	0:18:53:258	Jugendkraft
0:18:53:640	0:18:53:642	Nibelungen
0:19:28:124	0:19:28:362	Mime
0:19:28:384	0:19:28:762	Mime
0:19:28:784	0:19:29:182	Mime
0:20:7:200	0:20:7:202	Grubel
0:20:7:288	0:20:7:290	Grubel
0:20:53:836	0:20:53:838	Grubel
0:21:29:872	0:21:29:874	Grubel
0:22:14:136	0:22:14:138	Nibelungen
0:22:47:460	0:22:47:462	Nibelungen
0:22:49:444	0:22:49:446	Jugendkraft
0:23:27:360	0:23:27:362	Nibelungen
0:23:28:764	0:23:28:786	Mime
0:23:32:336	0:23:32:338	Nibelungen
0:23:34:496	0:23:34:498	Grubel
0:24:11:864	0:24:12:542	Grubel
0:24:12:844	0:24:13:582	Grubel
0:24:13:624	0:24:14:542	Sword
0:24:50:764	0:24:51:522	Sword
0:24:52:524	0:24:53:282	Jugendkraft
0:24:53:312	0:24:54:282	Jugendkraft
0:24:54:664	0:24:54:922	Sword
0:25:27:204	0:25:28:2	Sword
0:25:28:360	0:25:28:362	Jugendkraft
0:25:32:28	0:25:32:30	Mime
0:25:32:912	0:25:32:914	Nibelungen
0:25:33:776	0:25:33:778	Sword
0:25:34:820	0:25:34:822	Jugendkraft
0:26:7:808	0:26:7:826	Jugendkraft
0:26:53:844	0:26:54:122	Nibelungen
0:26:54:144	0:26:54:602	Grubel
0:26:54:644	0:26:54:882	Nibelungen
0:27:28:924	0:27:29:902	Nibelungen
0:27:29:944	0:27:30:262	Nibelungen
0:27:30:584	0:27:30:962	Nibelungen
0:27:31:224	0:27:31:542	Nibelungen
0:27:31:808	0:27:31:810	Mime
0:27:31:864	0:27:33:162	Nibelungen
0:27:34:720	0:27:34:722	Nibelungen
0:28:12:472	0:28:12:490	Nibelungen
0:28:48:576	0:28:48:578	Nibelungen
0:30:11:772	0:30:11:814	Jugendkraft
0:30:54:500	0:30:54:502	Mime
0:32:8:700	0:32:8:722	Nibelungen
0:32:10:296	0:32:10:370	Grubel
0:32:10:456	0:32:10:458	Grubel
0:32:12:204	0:32:12:750	Grubel
0:32:13:144	0:32:13:602	Grubel
0:32:14:324	0:32:14:326	Grubel
0:32:51:604	0:32:51:750	Nibelungen
0:32:51:756	0:32:51:802	Nibelungen
0:32:51:824	0:32:52:42	Nibelungen
0:32:52:184	0:32:52:742	Grubel
0:32:53:84	0:32:53:562	Grubel
0:33:27:504	0:33:27:802	Nibelungen
0:33:28:24	0:33:28:602	Grubel
0:33:29:24	0:33:29:582	Grubel
0:33:33:604	0:33:33:606	Nibelungen
0:33:34:40	0:33:34:442	Nibelungen
0:33:34:624	0:34:7:442	Nibelungen
0:34:7:480	0:34:7:482	Nibelungen
0:34:7:644	0:34:7:902	Nibelungen
0:34:8:124	0:34:9:438	Nibelungen
0:34:9:728	0:34:9:730	Nibelungen
0:34:10:964	0:34:11:274	Nibelungen
0:34:11:384	0:34:11:702	Nibelungen
0:34:11:808	0:34:12:562	Nibelungen
0:34:12:724	0:34:13:42	Nibelungen
0:34:13:164	0:34:13:902	Nibelungen
0:34:47:188	0:34:47:190	Nibelungen
0:34:47:536	0:34:47:538	Nibelungen
0:34:48:88	0:34:48:90	Nibelungen
0:34:48:464	0:34:48:662	Nibelungen
0:34:48:960	0:34:48:962	Grubel
0:34:50:484	0:34:51:22	Grubel
0:34:51:384	0:34:51:862	Grubel
0:34:54:324	0:34:54:862	Grubel
0:35:27:304	0:35:27:722	Grubel
0:35:33:464	0:35:33:466	Nibelungen
0:36:11:912	0:36:11:914	Nibelungen
0:36:14:864	0:36:14:878	Nibelungen
0:36:47:764	0:36:47:766	Nibelungen
0:36:48:680	0:36:48:682	Nibelungen
0:36:50:204	0:36:50:442	Nibelungen
0:36:50:484	0:36:50:962	Grubel
0:36:51:148	0:36:51:182	Nibelungen
0:36:51:372	0:36:51:822	Grubel
0:36:54:280	0:36:54:282	Nibelungen
0:36:54:304	0:36:54:862	Grubel
0:37:27:304	0:37:27:762	Grubel
0:37:30:684	0:37:30:686	Nibelungen
0:37:31:748	0:37:31:750	Nibelungen
0:38:7:848	0:38:7:850	Sword
0:38:13:304	0:38:13:306	Nibelungen
0:38:52:96	0:38:52:98	Nibelungen
0:40:8:604	0:40:8:842	Nibelungen
0:40:49:144	0:40:49:158	Nibelungen
0:40:51:348	0:40:51:350	Nibelungen
0:40:54:632	0:40:54:634	Sword
0:41:27:872	0:41:27:874	Nibelungen
0:41:32:964	0:41:33:322	Nibelungen
0:42:7:252	0:42:7:274	Nibelungen
0:42:11:756	0:42:11:758	Nibelungen
0:42:11:952	0:42:11:954	Nibelungen
0:42:13:44	0:42:13:370	Nibelungen
0:42:13:464	0:42:14:122	Nibelungen
0:42:47:116	0:42:47:118	Nibelungen
0:42:47:240	0:42:47:302	Nibelungen
0:43:30:144	0:43:30:502	Nibelungen
0:43:31:104	0:43:31:422	Nibelungen
0:43:31:796	0:43:31:798	Jugendkraft
0:43:34:928	0:43:34:930	Nibelungen
0:44:8:484	0:44:8:486	Nibelungen
0:44:10:20	0:44:10:22	Nibelungen
0:44:11:196	0:44:11:222	Nibelungen
0:44:13:836	0:44:13:838	Nibelungen
0:44:49:624	0:44:49:862	Grubel
0:44:50:64	0:44:50:342	Grubel
0:44:50:508	0:44:50:782	Grubel
0:44:50:988	0:44:51:222	Grubel
0:44:51:444	0:44:51:722	Grubel
0:45:29:764	0:45:30:82	Nibelungen
0:45:30:704	0:45:31:162	Sword
0:45:32:884	0:45:33:158	Nibelungen
0:45:33:804	0:45:34:262	Sword
0:46:7:204	0:46:7:562	Nibelungen
0:46:7:664	0:46:7:942	Nibelungen
0:46:8:108	0:46:8:402	Nibelungen
0:46:14:584	0:46:14:882	Nibelungen
0:46:47:884	0:46:47:978	Nibelungen
0:46:47:984	0:46:47:986	Nibelungen
0:46:47:996	0:46:48:182	Nibelungen
0:46:50:184	0:46:50:186	Sword
0:46:53:624	0:46:53:722	Mime
0:46:53:744	0:46:54:742	Mime
0:46:54:764	0:46:54:882	Mime
0:47:28:68	0:47:28:70	Nibelungen
0:48:9:184	0:48:9:682	Sword
0:48:10:396	0:48:10:398	Sword
0:48:10:984	0:48:11:262	Nibelungen
0:48:11:404	0:48:11:662	Nibelungen
0:48:11:804	0:48:12:82	Nibelungen
0:48:14:408	0:48:14:410	Nibelungen
0:48:50:996	0:48:50:998	Nibelungen
0:48:52:260	0:48:52:262	Jugendkraft

(Figure 35 – LM Detections, part 1) (Figure 36 – LM Detections, part 2)(Figure 37 – LM Detections, part 3)

Appendix XVII – LM Frame Predictions.csv

Several sections of the LM Frame Predictions.csv created from the 'CNNv3.py' script can be seen below as the system's predictions of what each frame in the source data is predicted to be, followed by their true values and whether these values match or not. As this file is over 26k lines long, we are only able to show several parts of this below, with each of the three images below showing several different sections of the .csv file. Also, the header for the .csv can be seen on the top of the left image.

Times (0.2s LM duration)	Predicted LM	Expected LM	Predict == Expected?
0:0:0:64	Audio	Audio	True
0:0:0:96	Audio	Audio	True
0:0:0:176	Audio	Audio	True
0:0:0:208	Audio	Audio	True
0:0:0:236	Audio	Audio	True
0:0:0:252	Audio	Audio	True
0:0:0:260	Audio	Audio	True
0:0:0:324	Audio	Audio	True
0:0:0:408	Audio	Audio	True
0:0:0:528	Audio	Audio	True
0:0:0:576	Audio	Audio	True
0:0:0:776	Audio	Audio	True
0:0:0:812	Audio	Audio	True
0:0:0:968	Audio	Audio	True
0:0:1:64	Audio	Audio	True
0:0:1:108	Audio	Audio	True
0:0:1:184	Audio	Audio	True
0:0:1:308	Audio	Audio	True
0:0:1:324	Audio	Audio	True
0:0:1:340	Audio	Audio	True
0:0:1:384	Audio	Audio	True
0:0:1:540	Audio	Audio	True
0:0:1:688	Audio	Audio	True
0:0:1:784	Grubel	Grubel	True
0:0:1:788	Grubel	Grubel	True
0:0:1:792	Grubel	Grubel	True
0:0:1:796	Grubel	Grubel	True
0:0:1:800	Grubel	Grubel	True
0:0:1:804	Grubel	Grubel	True
0:0:1:808	Grubel	Grubel	True
0:0:1:812	Grubel	Grubel	True
0:0:1:816	Grubel	Grubel	True
0:0:1:820	Grubel	Grubel	True
0:0:1:824	Grubel	Grubel	True
0:0:1:828	Grubel	Grubel	True
0:0:1:832	Grubel	Grubel	True
0:0:1:836	Grubel	Grubel	True
0:0:1:840	Grubel	Grubel	True
0:0:1:844	Grubel	Grubel	True
0:0:1:848	Grubel	Grubel	True
0:0:1:852	Grubel	Grubel	True
0:0:1:856	Grubel	Grubel	True
0:0:1:860	Grubel	Grubel	True
0:0:1:864	Grubel	Grubel	True
0:0:1:868	Grubel	Grubel	True
0:0:1:872	Grubel	Grubel	True
0:0:1:876	Grubel	Grubel	True
0:0:1:880	Grubel	Grubel	True
0:0:1:884	Grubel	Grubel	True
0:0:1:888	Grubel	Grubel	True
0:0:1:892	Grubel	Grubel	True
0:0:1:896	Grubel	Grubel	True
0:0:1:900	Grubel	Grubel	True
0:0:1:904	Grubel	Grubel	True
0:0:1:908	Grubel	Grubel	True
0:0:1:912	Grubel	Grubel	True
0:0:1:916	Grubel	Grubel	True
0:0:1:920	Grubel	Grubel	True
0:0:1:924	Grubel	Grubel	True
0:0:1:928	Grubel	Grubel	True
0:0:1:932	Grubel	Grubel	True
0:0:1:936	Grubel	Grubel	True
0:0:1:940	Grubel	Grubel	True
0:0:1:944	Grubel	Grubel	True
0:0:1:948	Grubel	Grubel	True
0:0:1:952	Grubel	Grubel	True
0:0:1:956	Grubel	Grubel	True
0:0:1:960	Grubel	Grubel	True
0:0:1:964	Grubel	Grubel	True
0:0:1:968	Grubel	Grubel	True
0:0:1:972	Grubel	Grubel	True
0:0:1:976	Grubel	Grubel	True
0:0:1:980	Grubel	Grubel	True
0:0:1:984	Grubel	Grubel	True
0:0:1:988	Grubel	Grubel	True
0:0:1:992	Grubel	Grubel	True
0:0:1:996	Grubel	Grubel	True
0:0:2:0	Grubel	Grubel	True
0:0:2:4	Grubel	Grubel	True
0:0:46:716	Audio	Audio	True
0:0:46:780	Audio	Audio	True
0:0:46:856	Audio	Audio	True
0:0:46:932	Audio	Audio	True
0:0:46:964	Audio	Audio	True
0:0:47:144	Audio	Audio	True
0:0:47:152	Audio	Audio	True
0:0:47:216	Audio	Audio	True
0:0:47:232	Audio	Audio	True
0:0:47:240	Audio	Audio	True
0:0:47:288	Audio	Audio	True
0:0:47:316	Audio	Audio	True
0:0:47:412	Audio	Audio	True
0:0:47:436	Audio	Audio	True
0:0:47:472	Audio	Audio	True
0:0:47:480	Audio	Audio	True
0:0:47:568	Audio	Audio	True
0:0:47:596	Nibelungen	Audio	False
0:0:47:608	Audio	Audio	True
0:0:47:620	Audio	Audio	True
0:0:47:628	Audio	Audio	True
0:0:47:680	Audio	Audio	True
0:0:47:828	Audio	Audio	True
0:0:47:844	Audio	Audio	True
0:0:47:888	Audio	Audio	True
0:1:20:24	Audio	Audio	True
0:1:20:64	Audio	Audio	True
0:1:20:164	Grubel	Audio	False
0:1:20:228	Audio	Audio	True
0:1:20:264	Audio	Audio	True
0:1:20:320	Audio	Audio	True
0:1:20:332	Audio	Audio	True
0:1:20:340	Audio	Audio	True
0:1:20:532	Audio	Audio	True
0:1:20:564	Audio	Audio	True
0:1:20:728	Audio	Audio	True
0:1:20:852	Audio	Audio	True
0:1:20:948	Audio	Audio	True
0:1:21:48	Audio	Audio	True
0:1:21:124	Audio	Audio	True
0:1:21:160	Audio	Audio	True
0:1:21:224	Audio	Audio	True
0:1:21:424	Audio	Audio	True
0:1:21:444	Nibelungen	Nibelungen	True
0:1:21:448	Nibelungen	Nibelungen	True
0:1:21:452	Nibelungen	Nibelungen	True
0:1:21:456	Nibelungen	Nibelungen	True
0:1:21:460	Nibelungen	Nibelungen	True
0:1:21:464	Nibelungen	Nibelungen	True
0:1:21:468	Nibelungen	Nibelungen	True
0:1:21:472	Nibelungen	Nibelungen	True
0:1:21:476	Nibelungen	Nibelungen	True
0:1:21:480	Nibelungen	Nibelungen	True
0:1:21:484	Nibelungen	Nibelungen	True
0:1:21:488	Nibelungen	Nibelungen	True
0:1:21:492	Nibelungen	Nibelungen	True
0:1:21:496	Nibelungen	Nibelungen	True
0:1:21:500	Nibelungen	Nibelungen	True
0:1:21:504	Nibelungen	Nibelungen	True
0:1:21:508	Nibelungen	Nibelungen	True
0:1:21:512	Nibelungen	Nibelungen	True
0:1:21:516	Nibelungen	Nibelungen	True
0:1:21:520	Nibelungen	Nibelungen	True
0:1:21:524	Nibelungen	Nibelungen	True
0:1:21:528	Nibelungen	Nibelungen	True
0:1:21:532	Nibelungen	Nibelungen	True
0:1:21:536	Nibelungen	Nibelungen	True
0:1:21:540	Nibelungen	Nibelungen	True
0:1:21:544	Nibelungen	Nibelungen	True
0:1:21:548	Nibelungen	Nibelungen	True
0:1:21:552	Nibelungen	Nibelungen	True
0:1:21:556	Nibelungen	Nibelungen	True
0:1:21:560	Nibelungen	Nibelungen	True
0:1:21:564	Nibelungen	Nibelungen	True
0:1:21:568	Nibelungen	Nibelungen	True
0:1:21:572	Nibelungen	Nibelungen	True
0:1:21:576	Nibelungen	Nibelungen	True
0:1:21:580	Nibelungen	Nibelungen	True
0:1:21:584	Nibelungen	Nibelungen	True
0:1:21:588	Nibelungen	Nibelungen	True
0:1:21:592	Nibelungen	Nibelungen	True
0:9:24:812	Jugendkraft	Jugendkraft	True
0:9:24:816	Jugendkraft	Jugendkraft	True
0:9:24:820	Jugendkraft	Jugendkraft	True
0:9:24:824	Jugendkraft	Jugendkraft	True
0:9:24:828	Jugendkraft	Jugendkraft	True
0:9:24:832	Jugendkraft	Jugendkraft	True
0:9:24:836	Jugendkraft	Jugendkraft	True
0:9:24:840	Jugendkraft	Jugendkraft	True
0:9:24:844	Jugendkraft	Jugendkraft	True
0:9:24:848	Jugendkraft	Jugendkraft	True
0:9:24:852	Jugendkraft	Jugendkraft	True
0:9:24:856	Jugendkraft	Jugendkraft	True
0:9:24:860	Jugendkraft	Jugendkraft	True
0:9:24:864	Jugendkraft	Jugendkraft	True
0:9:24:868	Jugendkraft	Jugendkraft	True
0:9:24:872	Jugendkraft	Jugendkraft	True
0:9:24:876	Jugendkraft	Jugendkraft	True
0:9:24:880	Jugendkraft	Jugendkraft	True
0:9:24:884	Jugendkraft	Jugendkraft	True
0:9:24:888	Jugendkraft	Jugendkraft	True
0:9:24:892	Jugendkraft	Jugendkraft	True
0:9:24:896	Jugendkraft	Jugendkraft	True
0:9:24:900	Jugendkraft	Jugendkraft	True
0:9:24:904	Mime	Mime	True
0:9:24:908	Mime	Mime	True
0:9:24:912	Mime	Mime	True
0:9:24:916	Mime	Mime	True
0:9:24:920	Mime	Mime	True
0:9:24:924	Mime	Mime	True
0:9:24:928	Mime	Mime	True
0:9:24:932	Mime	Mime	True
0:9:24:936	Mime	Mime	True
0:9:24:940	Mime	Mime	True
0:9:24:944	Mime	Mime	True
0:9:24:948	Mime	Mime	True
0:9:24:952	Mime	Mime	True
0:9:24:956	Mime	Mime	True
0:9:24:960	Mime	Mime	True
0:9:24:964	Mime	Mime	True
0:9:24:968	Mime	Mime	True
0:9:24:972	Mime	Mime	True
0:9:24:976	Mime	Mime	True
0:9:24:980	Mime	Mime	True
0:9:24:984	Mime	Mime	True
0:9:24:988	Mime	Mime	True
0:9:24:992	Mime	Mime	True
0:9:24:996	Mime	Mime	True
0:9:25:0	Mime	Mime	True
0:9:25:4	Mime	Mime	True
0:9:25:8	Mime	Mime	True
0:9:25:12	Mime	Mime	True
0:9:25:16	Mime	Mime	True
0:9:25:20	Mime	Mime	True
0:9:25:24	Mime	Mime	True
0:9:25:28	Mime	Mime	True
0:9:25:32	Mime	Mime	True
0:9:25:36	Mime	Mime	True
0:9:25:40	Mime	Mime	True
0:9:25:44	Mime	Mime	True
0:9:25:48	Mime	Mime	True
0:9:25:52	Mime	Mime	True
0:9:25:56	Mime	Mime	True
0:9:25:60	Mime	Mime	True
0:9:25:64	Mime	Mime	True
0:9:25:68	Mime	Mime	True
0:9:25:72	Mime	Mime	True
0:9:25:76	Mime	Mime	True
0:9:25:80	Mime	Mime	True
0:9:25:84	Mime	Mime	True
0:9:25:88	Mime	Mime	True
0:9:25:92	Mime	Mime	True
0:9:25:96	Mime	Mime	True
0:9:25:100	Mime	Mime	True
0:9:25:104	Mime	Mime	True
0:9:25:108	Mime	Mime	True
0:9:25:112	Mime	Mime	True
0:9:25:116	Mime	Mime	True
0:9:25:120	Mime	Mime	True
0:9:25:124	Mime	Mime	True
0:9:25:128	Mime	Mime	True

(Figure 38–LM frame preds, part 1) (Figure 39–LM frame preds, part 2) (Figure 40–LM frame preds, part 3)

Appendix XVIII – General Ethics Questionnaire

Department of Electronic, Electrical and Systems Engineering

UNIVERSITY OF BIRMINGHAM

GENERAL ETHICAL QUESTIONNAIRE FOR ALL STUDENTS

Name of student	Daniel Heaton
Email address of student	DJH589@student.bham.ac.uk
Name of supervisor	PJ

Title of Research Project:	Detection of Leitmotivs in Wagner's Opera using Neural Networks

Will the research project involve humans as participants of the research (with or without their knowledge or consent at the time)? This will include any survey, interview or questionnaire that human participants may be asked to complete at any stage as the main part of the project or in the evaluation of the results/deliverables of the project. It will also include any testing of devices, software and other deliverables, which may arise as a result of a project and involves human participants other than the student undertaking the project. Analysis of images or other recordings of human participants or their property and personal possessions is also included.	No
Are the results of the research project likely to expose any person to physical or psychological harm? (Note, before starting the project you will need to complete a risk assessment in all cases)	No
Will you have access to personal information that allows you to identify individuals, or to corporate or company confidential information (that is not covered by confidentiality terms within an agreement or by a separate confidentiality agreement)?	No
Does the research project present a significant risk to the environment or society?	No
Are there any ethical issues raised by this research project that in the opinion of your supervisor require further ethical review?	No

You have answered NO to all of the above questions. Further ethical review is not necessary. You should have this form available at the bench inspections and include it in your final report.