Deep Learning for Musical Form: Recognition and Analysis

by

Daniel James Szelogowski

A Thesis Submitted in
Partial Fulfillment of the
Requirements for the Degree of

Master of Science
in
Computer Science

At

The University of Wisconsin-Whitewater

April 2022

Graduate Studies

The members of the Committee approve the thesis of

Daniel James Szelogowski presented on April 29, 2022

_____

Dr. Lopamudra Mukherjee, Chair

_____

Dr. Hien Nguyen

_____

Dr. Zachary Oster

_____

Dr. Benjamin Whitcomb

Deep Learning for Musical Form:
Recognition and Analysis

by

Daniel James Szelogowski

The University of Wisconsin-Whitewater, 2022
Under the Supervision of Dr. Lopamudra Mukherjee

ABSTRACT

Musical form analysis is a rigorous task that frequently challenges the expertise of human analysts and signal processing algorithms alike. While numerous systems have been proposed to perform the tasks of musical segmentation, genre classification, and single-label segment classification in popular music, none have specifically focused on the analytical process used by classical musicians. Classical music form analysis facilitates a combination of these tasks, including form classification, structural segmentation, and multilabel large- and small-segment classification – tasks that lack feasible algorithms, machine learning models, and extensive research. Form analysis has many applications in the world of music, and a viable analytical system would greatly benefit performing musicians and academic researchers, both in musicology and signal processing. As well, current datasets used for related research tasks lack standardized analytical conventions, including form classification, and suffer from erroneous annotations and extensibility due to the data sources used for the music. In this thesis, we propose a new system to perform the task of automatic musical form analysis using deep learning models, as well as a new standardized dataset.

ACKNOWLEDGEMENTS

I would first like to thank my supervisor Dr. Lopamudra Mukherjee, who first introduced me to machine learning when I began the graduate program as a senior in my undergraduate degree. Your expertise and guidance reformed my skills as a mathematician and computer scientist, providing me with not only a new toolset as a researcher, but a new passion in the field of AI, and the drive to continue my path in academia.

I owe a debt of gratitude for the unending guidance from Dr. Jiazhen Zhou, who assisted me in developing my research skills in computer science from my senior year of high school throughout my undergraduate program, including acting as my advisor to help me eventually join the 4 + 1 program. Your help in guiding me into the graduate program reignited my passion for computer science, as the degree truly provided me with rewarding intellectual challenges and extremely useful skills as a scientist and independent researcher.

I would also like to thank Dr. Hien Nguyen, the other professor I had during my first semester as a graduate student. I could not have gained the technical reading and writing skills I have today without the help of your fantastic curriculum design. Learning how to decompose high-level research drastically changed my thought process on academic writing and helped me quickly integrate into the degree program.

I would like to thank Dr. Zach Oster, the first professor I had in the computer science department when I started as a freshman at the University of Wisconsin-Whitewater. Your classes especially provided me with useful information and tools that I constantly use to this day and learning assembly programming and computer organization in your class drastically improved my programming abilities and fundamentally changed my knowledge of computer science.

I would like to thank Dr. Benjamin Whitcomb, one of the only professors I had lectures with nearly every year of my college career, for greatly improving my musicianship, providing reward intellectual challenges, teaching 2 years of cello lessons, and sparking my interest in musical analysis. Your fantastic teaching inspired the subject matter of this very rewarding thesis.

I would also like to thank Mr. Robert Getka and Mr. Steven Mickelson, my high school computer science teachers, who not only provided me with the skills necessary to become a collegiate-level computer scientist but also continue to provide me guidance as an educator and greatly inspired my desire to become a teacher – my now biggest passion in life.

Most importantly, I would like to thank all of the faculty members in the Music Department for the fantastic education, skillset, and support they provided me as an undergraduate and the interest they have continued to provide me as an educator and researcher, as well as all of the faculty members in the Computer Science department for the knowledge and assistant both during my undergraduate and graduate careers, and lastly, my family – especially my parents David and Diane, friends, and teachers alike, who have supported me throughout my academic journey and continue to do so every day, as well as all of my students, who constantly remind me why I love teaching and inspire me to keep improving.

I dedicate this thesis to my family, friends, and educators, whose unending guidance and support have continued to lead me to a fulfilling life.

Thank you for all that you do.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

AI	Artificial Intelligence
ANN	Artificial Neural Network
CNN	Convolutional Neural Network
CTC	Connectionist Temporal Classification
FFT	Fast Fourier Transform
LSTM	Long Short-Term Memory
MFCC	Mel-Frequency Cepstral Coefficient
MIDI	Musical Instrument Digital Interface
MIR	Music Information Retrieval
MIREX	Music Information Retrieval Evaluation eXchange
ML	Machine Learning
MLS	Mel-scaled Log-magnitude Spectrogram
MPC	Music Perception and Cognition
MSCOM	Musical Structure Communities
NLP	Natural Language Processing
NMF	Non-negative Matrix Factorization
PCP	Pitch Class Profile
ReLU	Rectified Linear Unit
RNN	Recurrent Neural Network
SALAMI	Structural Analysis of Large Amounts of Music Information
STFT	Short-time Fourier Transform
SOM	Self-Organizing Map
SSLM	Self-Similarity Lag Matrix, or **SSM** (Self-Similarity Matrix)

CHAPTER 1
INTRODUCTION

1.1 Motivation

Classical music form analysis is a difficult task well-known to musicians and audio researchers alike, given the overall complexity of musical form design and the theoretical knowledge required to analyze pieces of music both at the phrase- and part-level, and large form classification. While there have been several various attempts utilizing novelty methods on audio features [6], community detection algorithms [1], and neural networks [7], none have proven to be sufficient for truly complex analysis tasks. This issue is especially true for more advanced musical forms such as sonata-allegro form where repetition of musical features is not nearly as clear as simpler two- or three-part forms, as opposed to the much clearer partitioning found in popular music written typically in verse-chorus form. In terms of model architecture as well, most current research utilizing deep learning methods like **Convolutional Neural Networks (CNNs)** lack the additional memory/recurrent cells found in **Long Short-Term Memory (LSTM)** Networks and **Recurrent Neural Networks (RNNs)** [7], resulting in an upper limit to the accuracy a model could achieve regardless of the training material [5].

One of the most prominent issues in the development of a more successful algorithm is the lack of an accurate dataset with standardized analytical conventions. The majority of all modern research on formal musical analysis algorithms has been performed using the same set of training data [3]: the **Structural Analysis of Large Amounts of Music Information (SALAMI)**

Annotation Dataset, a set of nearly 1,400 text files correlated to a particular "MP3" sound file for each song [10], containing tabulated timestamps within the piece of music and the corresponding analysis labels for each timestamp (e.g., a large form part letter/label, phrase letter/name, theme, instrument entrance, silence, etc.). While there are approximately 250 pieces of classical music in the database, the majority are analyzed either incorrectly by part/phrase – typically, neglecting repetition or incorrectly labeling repeated phrases, giving a unique part/phrase label to a repeated one, or incorrect naming of a phrase/theme based on the large form – or lack labels and/or timestamps for particular events within a piece.

While the annotation guidelines for the dataset are generally accurate for more contemporary music [9], they lack much of the regulations imposed by classical form analysis, including the classification of each piece into its large (compositional) form (i.e., sonata-allegro, rondo, binary, etc.) since the focus of the dataset is on musical 'partitions' [2]; given that the annotations also include instruments, the researcher must manually remove or exclude the labels, so as to avoid erroneous classification from instrument entrances. Also, while the dataset maintains records of the ID numbers of the MP3 files from their respective sources (for some, also the artist/performer names), the actual data files are difficult to obtain access to or are missing from their respective source/streaming service, or multiple performances can be found of the same piece and/or artist with differing durations or tempo changes, thus the timestamps become useless. Lastly, the use of MP3 files makes it difficult for a researcher to find the actual events within the sheet music/score of the piece since the timestamps are applied simply to waveforms, whereas another format such as MIDI can be used to analyze both the audio features of the waveform and the musical notation, avoiding copyright and licensing issues from the performing artists and allowing the researcher to verify the accuracy of an algorithm's label using the actual score against

any existing recording/performance of the piece. Since these issues arise from the design of the dataset, all models trained using the SALAMI annotations clearly suffer from these same discrepancies and lack ease of modification as a result.

This thesis proposes a new dataset (see *Chapter 3.1*) which also includes the classification of each piece into its respective large form (see *Appendix B*), and an improved model architecture from those found in prior research which implements the large form classification enabled by the new dataset. While these tasks are very specific to the analysis of classical music, these algorithms have great potential in their application for audio analysis in general (see *Chapter 7.2*), including:

- Performing structural analysis on any given audio or waveform, including spoken audio where patterns of repeated language may be used (i.e., poetry, forensic investigation, lecture, neuro-linguistic programming, **Natural Language Processing (NLP)** problems).
- **Optical Music Recognition (OMR)** – analyzing a piece using the sheet music, much like a human analyst, or correcting optical sheet music transcriptions using formal structures.
- Audio classification by content, with or without music (e.g., for sorting a web database, hearing-impaired accessibility tools, language classification from audio recordings [NLP]).
- Production of musical form/analysis-based anthologies, alongside other fields of musicology that lack significant research and technological advancement.
- Music practice and analysis tools, such as dividing a piece by themes for rehearsal, assignment generation using peak-picking, or a grading system for human-analyzed scores.
- Audio thumbnail/fingerprint generation, as applicable for a streaming service or web store.
- **Forensic Musicology**, where the analysis may be used to compare numerous pieces of music for similar or exact replications of musical phrases, motives, or other structures.
- Extension to video analysis to apply both visual and audio cues to the media's structure, whether formal in design (such as a music video, musical, etc.) or not (a movie/TV show).

The research presented in this thesis is intended for a reader with a strong computer science background and at least elementary knowledge of music theory. In the rest of this chapter, we will further discuss the aim of the thesis and outline its structure.

1.2 Goals and Objectives

This subsection will provide a discussion on the purpose and intentions of this thesis through research objectives.

1.2.1 Goals

The following list represents the goals of this thesis:

- Provide a new model to perform full form analysis (form classification, segmentation, part/phrase labeling), rather than simply peak-picking and segmentation (see *Chapter 2.2*).
- Develop a new, musically accurate dataset by common analytical conventions, including categorical divisions by large musical form.
- Present a more accurate deep learning model to perform both form-level and part/phrase-level analysis.
- Contribute to the literature by obtaining improved results from previous studies in the formal analysis of music using machine learning and peak-picking methods.

1.2.2 Objectives

The following list represents the objectives of this thesis:

- Examine the previous work done in the field through extensive background research.
- Choose suitable algorithms for the network architecture and signal processing through extensive and exhaustive research.
- Expand upon existing model architecture designs using recurrent memory cells to better recognize repeated audio patterns.
- Provide appropriate evaluation metrics and accurate model performance results.

1.3 Structure of the Report

- **Chapter 1: Introduction –** This chapter introduces the motivation, goals, objectives, and structure of this thesis.
- **Chapter 2: Background and Related Work –** This chapter discusses the background of this thesis and provides a definition of terms and a literature review of related research.
- **Chapter 3: Methodology –** This chapter illustrates the approach to developing the new dataset, system components and structure, model design, and related requirements.
- **Chapter 4: Implementation –** This chapter provides details on implementing the data preparation functions and neural network architecture, as well as the system constraints.
- **Chapter 5: Evaluation –** This chapter introduces the evaluation metrics of the system and reviews the results of the proposed model in both training and new testing data.
- **Chapter 6: Discussion –** This chapter provides a discussion on the final model design following the experiments and the areas which may be improved upon.
- **Chapter 7: Conclusion –** This chapter summarizes the thesis and discusses potential future research to expand on this topic.
- **Appendix A: Example Training Data –** This appendix illustrates examples of the training data used for the model, including intermediate data, novelty input, and annotation data.
- **Appendix B: Compositions Used for Training –** This appendix contains a categorical listing of each piece of music used in the new proposed dataset.
- **Appendix C: Related Definitions –** This appendix provides additional terminology related to music theory and machine/deep learning.
- **Appendix D: Implementation of Form-NN System –** This appendix provides the implementation of the proposed neural network model and system architecture.

CHAPTER 2
BACKGROUND AND RELATED WORK

In this chapter, we will give an overview of the relevant theoretical foundations and necessary technical background in machine learning/artificial intelligence and music theory (including methods of audio analysis), as well as provide a comprehensive literature review discussing the development of the field and the current state of the art.

2.1 Background

This section will provide a broad introduction to the fundamental concepts and definitions necessary to understand the content of this thesis, in both the fields of machine learning and music.

2.1.1 Machine Learning

Machine Learning is a branch of computer science and data (analysis) science, as well as a subfield of **Artificial Intelligence (AI)**, which employs algorithms and statistical models designed for teaching computers to learn and produce some output without explicit programming by analyzing and drawing inferences from patterns in the data [24]. These models are then typically trained on large quantities of data to produce a prediction or classification for new test data or to produce new, realistic data resembling the training set, among numerous other possible tasks [25]. Artificial Intelligence, on the other hand, is a broad term for any intelligence system/smart technology, whereas machine learning is a type of AI that focuses on data and information processing [26].

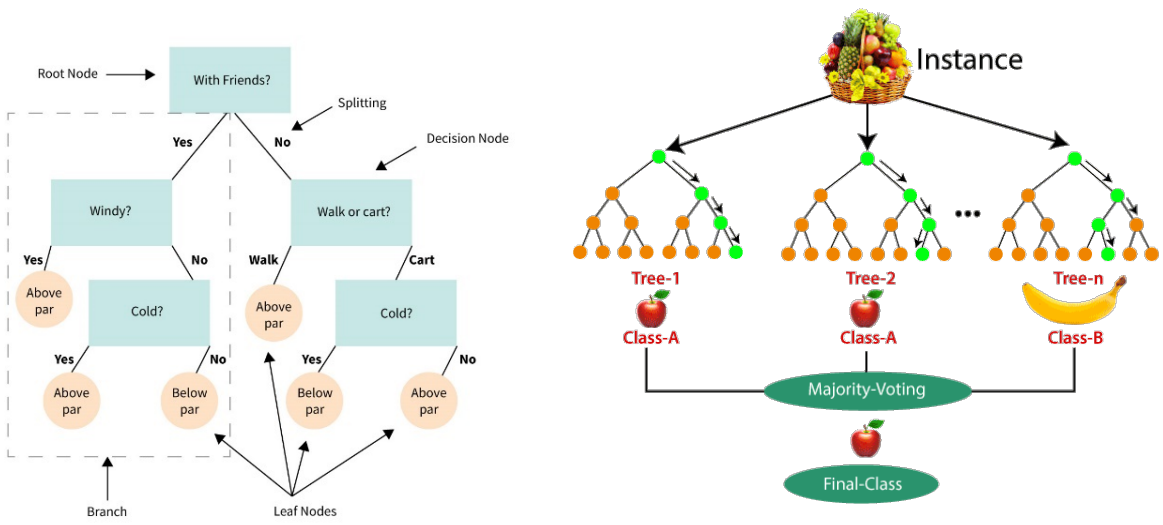
Machine learning (ML) tasks are typically classified by one of five different categories [8]:

- **Supervised Learning (Task-Driven) –** Classification, prediction (including decision trees), regression, similarity learning, Support-Vector Machines (SVMs).
- **Unsupervised Learning (Data-Driven) –** Clustering, dimensionality reduction.
- **Semi/Self-Supervised Learning (Self-Driven) –** Graph-based algorithms, generative adversarial networks (GANs), low-density separation, self-training.
- **Reinforcement Learning (Experience/Reward-Driven) –** Decision-making, dynamic programming, gaming, heuristic methods, Monte Carlo methods, navigation, Q-learning.
- **Deep Learning (Knowledge-Driven) –** Artificial neural networks (NN, CNN, RNN, …).

Common machine learning algorithms include Linear Regression, Logistic Regression, Naïve Bayes, **Decision Trees**, **Random Forests**, **k-Nearest Neighbors**, k-Means Clustering, Support-Vector Machines, Singular Value Decomposition (SVD), and Gradient Boosting [27].

2.1.1.1 Decision Trees and Forests

Decision Trees are an extremely useful tool for classification and regression tasks (i.e., supervised learning). The tree contains a **root node** that splits off into additional **decision nodes** based on the features of a dataset, representing the different "questions" the tree will attempt to answer, and **leaf nodes** which represent the **outcome** of the decision (see Figure 2.1 (a)). They train quickly and provide deterministic output (as opposed to the probabilistic output of a neural network) and work efficiently even with small datasets since they fit parameters explicitly to direct information flow [57]. They can be combined in parallel (**bagging**) or sequence (**boosting**) to enhance performance as an **ensemble** referred to as a **forest** [56]. A **random forest** uses bagging to fit the inner trees to randomly chosen subsets of the data and take the majority decision as output (see Figure 2.1 (b)).



(a) (b)

Figure 2.1: A simple decision tree model (a) [55], a random forest model (b) [56]

2.1.2 Deep Learning and Neural Networks

Deep Learning (sometimes Deep Structured/Hierarchical learning) is a machine learning technique that utilizes [artificial] **Neural Networks (NNs)** to mimic the function and structure of the human brain, allowing for improved performance for large quantities of data [28]. Neural networks are a type of deep learning algorithm comprised of layers of "neurons" that replicate the way that the human brain discovers patterns and recognizes relationships in data, allowing great modularity in what tasks the model is capable of [19]. Comparatively, machine learning algorithms require the user (e.g., data scientist) to perform the feature extraction manually before training the model, whereas deep learning performs both feature extraction and abstraction automatically as part of the neural network directly from the raw dataset with little need for human input [23].

Neural Networks are generally classified as one of two primary types (see Figure 2.2):

- **Artificial Neural Network (ANN) –** Often referred to simply as a Neural Network (NN), any network composed of an **input layer**, one/two **hidden layers**, and an **output layer**, mimicking the **dendrites**, **cell body**, and **axon** of a biological neuron (respectively) [54].
- **Deep (Learning) Neural Network (DNN) –** Any neural network composed of several hidden layers between the input and output layers (such as a Multilayer Perceptron) [15].

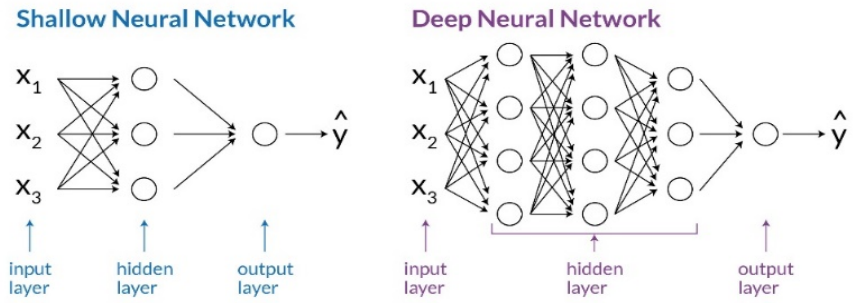


Figure 2.2: Artificial Neural Network graph vs Deep Neural Network [42]

Given the variability in architecture that neural networks allow, they are most often used for deep learning tasks such as Automatic Speech Recognition, Natural Language Processing (NLP), and Computer Vision [28]. Neural networks are effective for complex tasks because they typically do not need explicit rules to be defined for what to expect from their input and instead focus on learning from labeled data – referred to as "ground truth" data – which is used to provide an answer key during training [29]. The network processes this data in a series of **epochs**, a cycle where all the training data has been used exactly once (including a forward and backward pass, counted as one epoch) which is made up of some number of **batches** (a divided sample of the dataset) passed through in a series of **iterations** [30]. During this training cycle, data is passed through the input layer, assigned **weights** and **biases** as needed within each layer, and a new output is produced using a user-defined **activation function** (typically either the **Sigmoid** or **Unit Step functions)**. Each epoch updates the weights to produce more accurate results over time until a sufficient accuracy has been reached; typically, more data and variety will result in a higher accuracy [29].

Common forms of Neural Networks include the following (see *Appendix C.2*):

- **Perceptron –** The oldest and simplest form of neural network, containing only one neuron (in a single layer) which takes *n* inputs and (dot) multiplies them by their corresponding weights to produce a single summed output; used as a linear binary classifier (see Figure 2.3) [12].
- **Feedforward Neural Network (FNN) –** The first type of Artificial Neural Network created, and the most common form used today [14]; a type of multilayer neural network with the addition of layers (of neurons), in the form of an input layer, hidden layer(s), and output layer, where the data flow between each neuron only moves forward into the next layer (i.e., does not form a cycle) and never backward, used primarily for supervised learning problems such as basic pattern and image recognition [13].
- **Multilayer Perceptron (MLP) –** A feedforward network where each layer is fully connected (each neuron in layer *n* is connected to each neuron in layer *n* + 1); often used for computer vision and Natural Language Processing tasks [12].

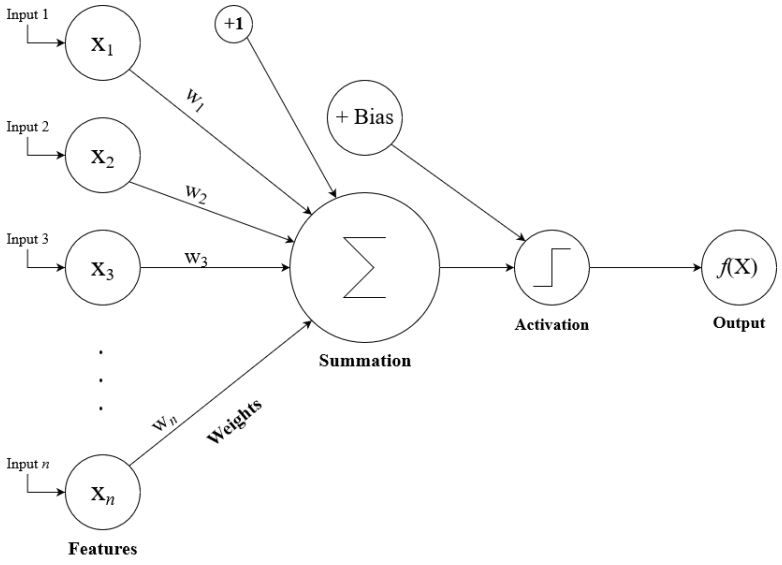


Figure 2.3: A (Single Layer) Perceptron

2.1.2.1 The Convolutional Neural Network

The **Convolutional Neural Network (CNN)** is a special feedforward network that introduces the **convolutional layer** (usually many), which applies filters (the **kernel**) to complex data (such as audio, imagery, or video) and **pools** characteristic features together to identify patterns and classifiable attributes within the data (see Figure 2.4) [12]. They are often used for computer vision tasks including image/video pattern recognition, image classification, object recognition, and self-driving vehicles, as well as NLP tasks and audio analysis [14]. The connectivity pattern between neurons in the CNN architecture is based on the organization of the visual cortex found in animals, allowing the computational model to bear a stronger resemblance to biological processes than other forms of neural networks [22]. The model trains on **spatial features** within the dataset media – in the case of an image, the relationship between the arrangement of pixels within the image [23]. Training these networks requires large, consistent datasets to create a model that is both reliable and accurate in prediction and classification, and due to the feedforward design of the architecture, highly accurate pattern recognition in temporal (time-based) data remains a difficult task [22].

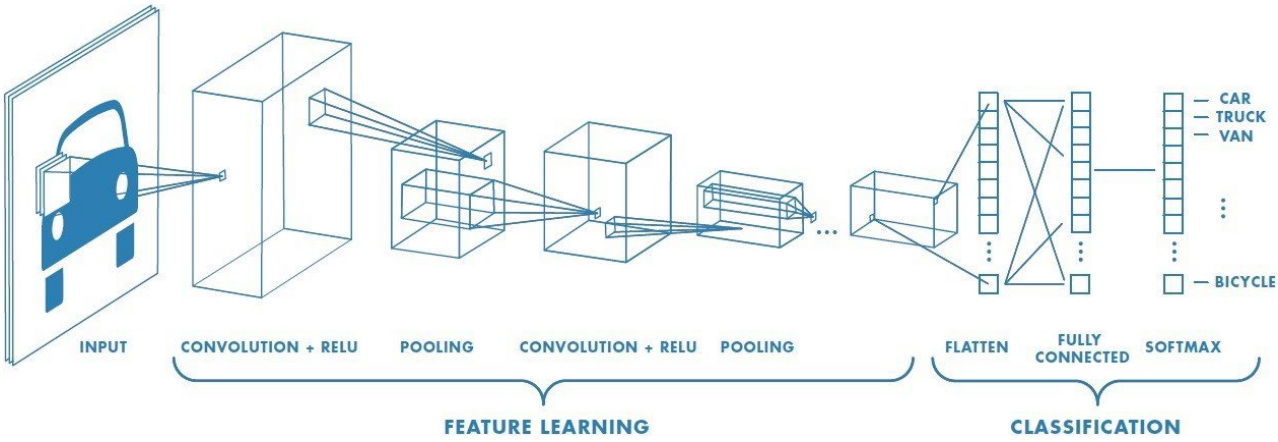


Figure 2.4: A typical Convolutional Neural Network architecture [34]

2.1.2.2 The Recurrent Neural Network and LSTM

The **Recurrent Neural Network (RNN)** is similar to a multilayer perceptron but with the addition of loops, wherein layer *n* can refer to previous information from previous layers through recurrence loops (see Figure 2.5), allowing the network to recall previous experiences (i.e., temporal/time-varying data, within either the current layer or prior layers) and influence future events [20]. These experiences are known as hidden states, which are stored in a form of short-term memory known as **state matrices** [12]. They are often used for time series forecasting and NLP problems, including natural language generation and text or speech recognition/prediction. However, aside from difficulty in training (a result of the Vanishing Gradient Problem), the lack of long-term memory limits the overall functionality of the network (see *Chapter 4.2.2*) [14].

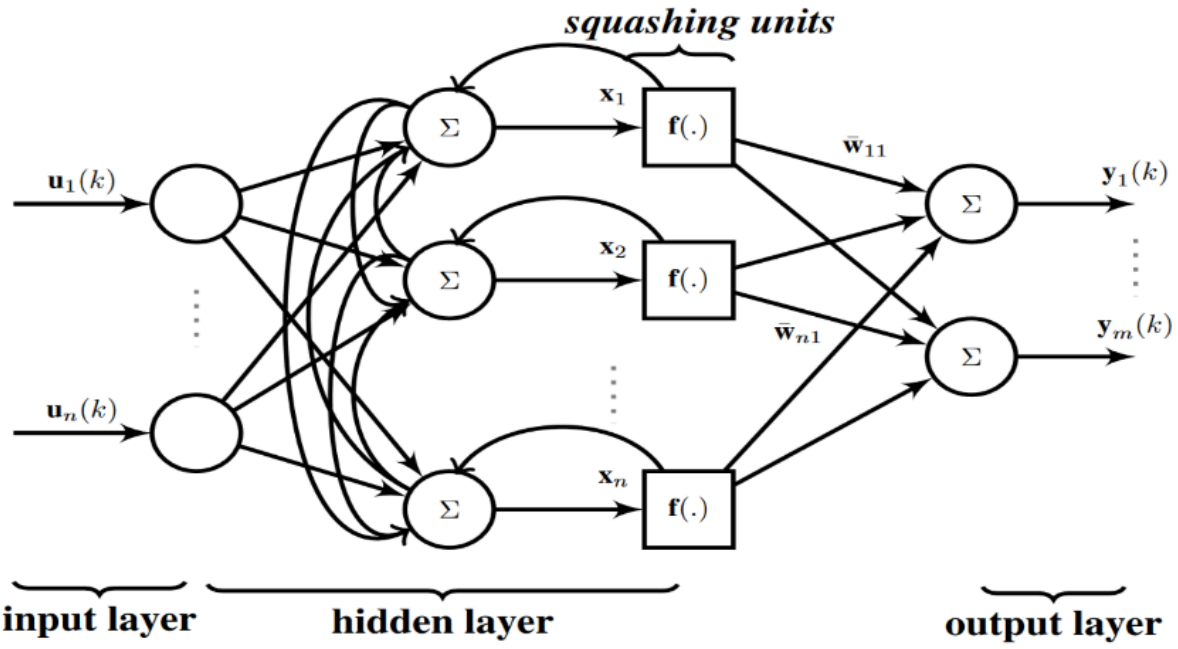


Figure 2.5: A typical Recurrent Neural Network architecture [43]

The solution to these difficulties is **Long Short-Term Memory (LSTM) Networks**, an expanded form of RNN that introduces an additional state matrix – one long-term state (cell states) and one short-term state (hidden states) – where more persistent states have a higher weight (stored in the long-term state matrix) and thus have a stronger consideration factor for new input (see Figure 2.6) [12]. The addition of the cell state allows the LSTM to regulate information in the cell through "**gates**" which add or remove information within the cell and provide a means of handling long-term dependencies [21]. As such, they are very efficient for pattern recognition in continuous data, such as NLP tasks and stock market or time-based data predictions. Recurrent neurons/memory cells are also often added to other network architectures to increase accuracy in pattern recognition or generation, including CNNs (usually as CRNNs or CNN-LSTMs) and GANs (typically C-RNN-GANs or LSTM-GANs) [14].

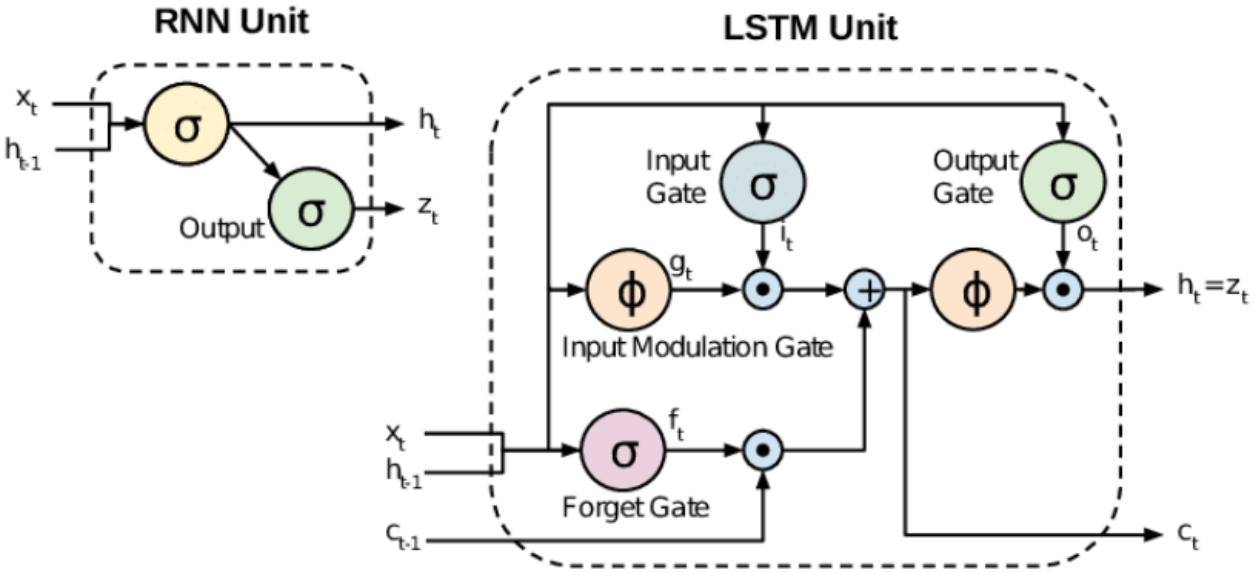


Figure 2.6: Recurrent Memory Cell versus Long Short-Term Memory Cell [44]

2.1.2.3 Music Information Retrieval

The field of **Music Information Retrieval (MIR)** is an interdisciplinary science regarding the retrieval of information from music which includes the extraction of a near-infinite number of possible **features**, such as instrumentation, genre, tempo, mood, time signature, key signature, form, harmonic progression, and more [31]. While MIR as a field is considerably small, it is applicable for numerous real-world applications, including in other academic fields such as computational intelligence, informatics, machine learning, musicology, optical music recognition, psychoacoustics, psychology, and signal processing, among many others [32]. Most often, MIR is used as a tool for academic and business projects to categorize, compose, and manipulate music, with artist, genre, and mood classification systems being the most popular applications of MIR [32].

2.1.2.4 Music Perception and Cognition

Similar to MIR, **Music Perception and Cognition (MPC)** is a subfield of cognitive psychology focused on the cognitive mechanisms involved in listening and responding to music [33]. MPC looks at numerous musical features – including the perception of pitch, tonality/harmony, time organization (tempo, rhythm, groupings, meter, events, form, etc.), and the cognition involved in musical ability (expertise and skill acquisition) and performance (such as interpretation, planning, and communicative structure) – among many other potential aspects of music performance and theory alike. Cognitive neuroscience is also an important branch of MPC, involving studies of electroencephalography, neuroimaging, and neuropsychology. An MPC researcher may also look at originating features including the development of music cognition, cross-cultural perception, and evolutionary music psychology (such as the development of relevant neural structures).

2.1.3 Formal Musical Analysis

Analysis of musical form refers to the analytical study of the **design**, **organization**, and **tonal structure** of a piece of music, including its **themes** and the relationships and/or correlations between each of them [2]. Green elaborates,

> The organization of [melodic] and rhythmic factors make up what is called the design of a composition. […] Design is the organization of those elements of music called melody, rhythm, cadences, timbre, texture, and tempo. The harmonic organization of a piece is referred to as its tonal structure. [… However, during the analytical process, aside from formal classification] we must also deal with a prior concern – the way in which music grows and shapes itself from tiny fragments into phrases and combinations of phrases [2, pp. 4-5].

As such, musical analysis should be treated as both an **aural and visual exercise** – the ear will often be deceived in recognizing **formal structures** compared to what the eye may see within the score of the piece (until the ear has developed proficiency from direct practice) [11].

2.1.3.1 Genre and Shape versus Form

It should be noted that **genre**, **shape**, and **form** are three distinct features of a piece of music. Classical (western art music) pieces will be sometimes named with titles reminiscent of a form – (such as “Rondo” or “Sonata”) yet feature multiple movements each with their own distinct forms – or are incorrectly classified by authors who frequently describe various genres of the musical eras, such as the motet, chanson, cantata, aria, suite, opera, etc., as being “forms of the time” [2]. While pieces of music may be in the same genre, they are often different in musical structure; as such, the analyst should be aware of what forms are often employed for a particular genre as well.

Musical shape may also sway the listener in distinguishing the overall structure of a piece. Shape as a musical quality refers to the overall surface contour of a piece and may change or vary by the events and interactions of building tension and relaxation within the musical "story arc" [2]. Green describes numerous factors which may influence the shape of a piece [2, p. 3]:

- Rise and fall of melodic lines, particularly in outer voices (like soprano and bass)
- Rhythmic activity
- Dynamics
- Texture
- Instrumentation
- Relative amount and degree of consonance and dissonance
- Harmonic rhythm (rate of chord change)

Hence, the musical shape within different sections of a piece may be very similar, but the overall form and/or harmony of the sections might differ drastically.

2.1.3.2 Form Analysis and Why Form is Difficult

The task of **form analysis** demonstrates a high degree of musical competency, often used as part of graduate entrance/exit exams [35] and/or theses, as well as for many undergraduate writing requirements [36] to show the student's proficiency in music theory. The explorative process involves the separation of a piece of music into its parts and decompiling the relationships between the parts and the relation to their sum [2]. As such, these parts must also be defined by the level of scope which they cover (see *Appendix C.1*):

- **Motive (or Motif)** – The smallest musical idea, typically between two to five notes long [2, p. 31]. A **Compound Motive** may be divided into two or more smaller motives [38].

- **Phrase** – The smallest complete musical statement, ending with a cadence; usually four measures long but may vary, including asymmetrical lengths [39].
- **Theme (Melody/Subject)** – A complete musical idea, often a complete phrase/period [40].
- **Section –** A subdivision of a part, often varying in length [38].
- **Part –** A large-scale portion of a piece or movement [38].

Composers often employ many additional ideas or attributes to phrases – including elisions, links/bridges, introductions/extensions, repetitions/imitations/variations, sequences, compounded parts, and so on – which further complicate and add great difficulty to the analytical process and are often the primary source(s) of disagreement between theorists [2]. Hence, the task can be viewed as a **Fuzzy Logic problem** (i.e., based on human reasoning/**degrees of truth**, rather than **definitive truth**), where providing exact and fuzzy rules for solving the problem is difficult or nearly impossible [37]. Green adds that "although the **phrase** is a basic unit of music, it almost eludes precise definition. Writers on music seem to agree only that the phrase (1) exhibits some degree of completeness, and (2) comes to a point of relative repose" [2, p. 6].

For analysts, cadences are the most validating musical feature in determining the division of phrases and parts. A **cadence** (comprised of a pair of chords) is the harmonic or point of rest within a musical passage or harmonic progression; much like the punctuation of a sentence, cadences vary in expression in their separation of musical "clauses" and provide a definitive divisor between musical events [39]. Numerous complex factors affect the strength of a cadence that a listener hears, as well as the degree to which a cadence sounds conclusive (see Table C.1 (a)) – similar to the inflection of a spoken phrase [38]. However, much like language, these alone are not enough to base the analysis on – given the fuzzy rules of form, there is never one spectrum on which a part may be based (melody, harmony, etc.) and there are numerous factors of difficulty.

Given these musical structures, Green defines a **Six-Step Method of Analysis** [2, pp. 93-94]:

1. Consider the successive phrases, note their cadences, and come to a conclusion regarding the tonal structure of the piece, deciding whether it exhibits a single, double, triple, or other harmonic movement. If single, is the harmonic movement complete, interrupted, or progressive? In a double harmonic movement, which type is each?
2. Consider the design from the cadential point of view, noting the presence or absence of any divisive conclusive cadence.
3. Consider the design from the motivic and melodic point of view, noting the presence or absence of contrasting phrases and phrases that are restatements of other phrases. (A Restatement differs from a repetition in that the former is a recurrence after contrast while the latter is an immediate recurrence. Restatements are important elements of form; repetitions are not.) Represent the design graphically by means of letter: A–BA, A–B–A, A || A′, and so on (see *Chapter 2.1.4*).
4. In the light of steps 1, 2, and 3, [conclude] the number of parts the piece will separate [into].
5. Consider steps 1 and 4, decide whether the form is continuous, sectional, or (if there are three or more parts) full sectional.
6. Decide whether or not the piece coincides in a general way with any of the commonly used standard forms [i.e., One Part, Continuous/Sectional Binary, Bar Form, Sectional/Full Sectional/Continuous Ternary, Sectional Four or Five Part] (see *Chapter 2.1.4*).

While these steps are useful for humans to analyze music, performing these same steps programmatically would be an extremely difficult task and would require higher-level processing and fuzzy logic skills currently impossible with current machine learning models. As well, the list of rules/conventions in music theory is far too vast to train a model on by **explicit programming**.

2.1.4 Discussion of Classical Music Forms

One of the most important factors in the classification of musical form is repetition, either at the phrase or part levels, or both – a prominent cause for error found within the **SALAMI** dataset (see *Chapter 1.1*) due to the lack of standardized analytical conventions. Repetition can most distinctly be classified into three variants: (exact) **repetition**, **varied repetition** (where some part of the musical feature is noticeably different from the original or has a different **harmonic goal**), and **contrast** (dissimilar but contrasting) [2, p. 50]. In formal analysis, **Parts** are most commonly labeled with capital letters (**A, B, C, …**), **Sections** are often labeled with numbers (usually in **Development** sections, i.e., **Sec. 1, Sec. 2, …**), and **Phrases** are typically labeled with lowercase letters (**a, b, c, …**) [2, Ch. 5-6]. Hence, to distinguish these parts by repetition class, exact repeats of a part/phrase are given the exact **same label** regardless of where they reoccur in the piece (**A, A, …** or **a, a, b, a**, and so on), though varied repeats may be given either the **same label** if it is similar enough to the original material, otherwise, they are given a **prime mark (′)** or **superscript** (^{n}) including varied repeats of varied repeats and contrasting varied repeats (e.g., **a′, a″, a³, a⁴,** etc.) [2, p. 68]. Lastly, contrasting material (whether independent or dependent on the original material) is given a new label – the **next alphabetical letter** not previously used (**a, b, a′, c, a′, …**). It should also be noted that a melody that is repeated exactly but in a different voice, instrument, or octave is still considered exact repetition and should be given the same label.

The following list defines the most common classical forms, used to classify the new proposed dataset (see *Appendix B*; <u>note that multiple additional rules also follow many of the definitions but are unnecessary for this thesis since they are unusable in training the model</u>):

- **Unary/Strophic** – A small form comprised of one part, either repeated (e.g., **AAA…**) or modified (**AA′A″…**) [45].

- **Through-Composed** – A continuous, non-repetitive, non-sectional small form in which each stanza/part is a different theme (**ABCD…**), in contrast to strophic form [46].
- **Binary** – A small form divided into two parts (including repetitions of either part) [39].
    - **Simple Binary** – A two-part form, either **AA′** or **AB** [2].
    - **Rounded Binary** – A common two-part form where part B is dependent on part A, and A returns at least partially (**ABA′**, see Figure 2.7) [38].
- **Bar** – A binary small form in the pattern **AAB** [2].
- **Ternary** – A three-part small form in which the third part is a repetition or varied repetition of part A, where part B is independent and all of A returns (**ABA** or **ABA′**) [38].
- **Three-Part** – A rare small form of three independent, contrasting parts (**ABC**) [38].
- **Arch (Chiastic) –** A palindromic form that is symmetrical (typically) around a central part, though themes may not always be repeated in entirety; usually **ABCBA** form [2].
- **Minuet & Trio** – A multi-movement **composite ternary** form comprised of a (usually) rounded binary or ternary form section (the **minuet** or **scherzo**), a second independent part of similar form (the **trio**), followed by a repetition of the minuet usually without repeats [2, p. 145]. Typically written as **A {** ||: A :||: $BA^{(\prime)}$:|| **}**, **B {** ||: C :||: DC :|| **}**, **A {** $ABA^{(\prime)}$ **}**.
- **Theme and Variation** – A strophic-like form consisting of a small **theme** that is repeated with a new **variation** on the theme (melody and accompaniment) for each repetition [2]. Types of variations include characteristic (of some other genre), contrapuntal, figural, melodic, ornamental, and simplifying [38]. Often, these pieces will contain numerous, sometimes tens of variations, and some composers may treat the last variation as a compound form **Coda** containing multiple consecutive variations before concluding [2].

- **Ritornello** – An episodic form that alternates between the **refrain** (part A) and contrasting sections called **episodes** (**ABACADA…**); it may be used in other compound forms like **sonata** [38].
- **Rondo** – A **ritornello-based** form (or expanded ternary form) – often employed as the last movement of a larger instrumental work – where the **refrain** (part A) is stated at least three times, typically comprised of either 5 or 7 parts but potentially up to 13 (usually ABACA or ABACABA) [38]. Rondos of 7 or more parts typically follow the **sonata principle** [2].
- **Sonata (Sonata-allegro)** – A **composite** (multi-layered with inner forms) **rounded binary** form comprised of three parts (see *Appendix C.1.1*): the **Exposition (Expos.)**, **Development (Dev.)**, and **Recapitulation (Recap.)** [38]. A **Concerto Sonata** may repeat the exposition, first by the orchestra and then by the soloist, and typically includes a **cadenza** (a virtuosic solo passage). A **Sonatina** simply excludes the Development section.

Additional descriptors to the various forms are also frequently used by analysts, such as **closed** versus **open** form (ending on the tonic chord or not) and **sectional** versus **continuous** form (part A ends on the tonic in the home key or not, respectively). Many pieces may also not fit into the list of common forms at all, instead being classified as a **Unique Form** (such as **ABCBDAC**) or an **Extended Form** (e.g., **AABABA** or **AAAAA**); some composers may also employ a **Hybrid Form** such as the Sonata Rondo, Rondo Sonata, or other composite forms [2]. Often composers will include a **Coda** as well, a section meant to bring a piece or movement to an end (irrelevant to the formal classification of the piece) – these may be of any length beyond four measures (possibly over 100) and should be labeled simply as "**CODA**" rather than a unique letter label [38]. Introductions, extensions, links/bridges, transitions, **Codettas** (a short musical idea that ends a part within a piece), and any other non-structural material (**fugue** subject/answer, stretto, and so on)

also should not be given a unique letter label, using their respective names instead (e.g., **intro, ext., bridge, trans.,** etc.) as they do not influence the form of the piece either. Theme and Variation pieces are typically divided by the composer as **Theme** and **Variation *n*** (**Var. I, Var. II, …**), so they are commonly labeled by the type of variation they utilize (the process or method that the melody is varied by), and any sub-forms found within the various divisions. Lastly, compounded forms are analyzed within each sub-form (a form hierarchically nested within a larger compound form), and episodic and thematic forms (**ritornello, sonata**) are sometimes divided by their own terminologies – such as **Refrain** (**Exposition/Middle Entry/Final Entry** for **Fugues**) and **Episode** for ritornello rather than **A** or **B/C/D…**, among the various sonata form parts (see *Appendix C.1.1*) – though analysts frequently use **letters** and **labels** in tandem depending on the complexity of the piece [2].

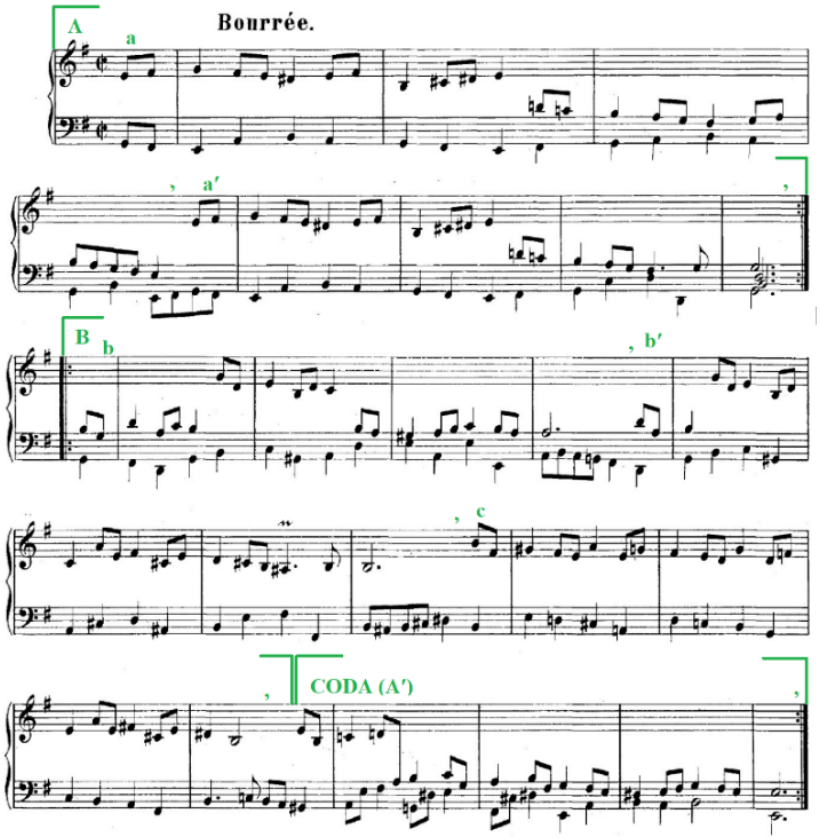


Figure 2.7: Example analysis of Bourrée from J.S. Bach's BWV 996 **(Rounded Binary)**

2.1.5 Audio Analysis Techniques

Audio analysis and **signal processing** techniques are very frequently used methods of preparing audio for use in machine learning problems, especially NLP and **Automatic Speech Recognition (ASR)** tasks [49]. One of the most popular methods of obtaining feature data from an audio signal is through a **Spectrogram** (see *Appendix A*) – a visual representation of the signal strength (i.e., volume/intensity or **energy distribution**, usually represented with a heat map) over time across various frequencies within a waveform. These graphs are calculated using the **Discrete Short-Time Fourier Transform (STFT)** to obtain the strength of small samples throughout the signal using the **Fast Fourier Transform (FFT)** and a **window function** (a function that is non-zero within a specified interval) $w : [0 : N - 1] \to \mathbb{R}$ of length $N \in \mathbb{N}$ (the duration of the sections) and **hop** (step) **size** $H \in \mathbb{N}$ [50]. The discrete STFT (a complex number) $\mathcal{X}$ of signal x is given by

$$\mathcal{X}(m,k) \coloneqq \sum_{n=0}^{N-1} x(n+mH)w(n)\, exp(-2\pi ikn/N) \tag{2.1}$$

such that $m \in [0 : M]$ and $k \in [0 : K]$, where $x : [0 : L - 1] := \{0, 1, \ldots L - 1\} \to \mathbb{R}$ is a real-valued discrete-time signal of length L obtained by equidistant sampling respective to the fixed **sampling rate** F_s specified in Hertz [51]. Here, $K = N/2$ represents the **Nyquist** frequency index (the highest frequency that can be sampled at a given rate to reconstruct a signal), $M := \left\lfloor \frac{L-N}{H} \right\rfloor$ is the maximal frame index where the window's time range is contained entirely within the signal's time range, and $\mathcal{X}(m,k)$ denotes the k^{th} Fourier coefficient for the m^{th} time frame. This function yields a **spectral vector** of size $K + 1$ for each fixed time frame m from the coefficients $\mathcal{X}(m,k)$ for $k \in [0 : K]$, the square magnitude of which is then used to calculate the (two-dimensional) spectrogram

$$\mathcal{Y}(m,k) \coloneqq |\mathcal{X}(m,k)|^2 \tag{2.2}$$

where the vertical axis represents frequency, and the horizontal axis represents time.

2.1.5.1 Audio Features and Feature Extraction

Audio features can be classified as one of three different types: **perceptual features** which are extracted using a perceptual representation of psychoacoustic data, **temporal features**, which are extracted from the waveform directly, and **spectral features** which are extracted from the frequency representation of a signal – one of the most often used features for ML tasks [52]. Hence, spectrograms can be used to extract a great number of audio (and music-specific) features by applying additional data transformations to the spectral information (see *Appendix A*) [22]:

- **Cepstrum** – The "spectrum of a spectrum" of the spectral vectors, found using the inverse FFT which can be used to analyze the vibration and sound of a signal. The **power cepstrum** is the distribution of the signal's power over its frequency and density; more commonly used is the **Mel-frequency cepstrum (MFC)**, a short-term representation of this feature. The **Mel Scale** is a psychoacoustic scale of the perceived frequency of a sound in relation to its actual frequency, intended to scale a sound to how the human ear would perceive it.
- **Mel-Frequency Cepstral Coefficients (MFCCs)** – The coefficients of which an MFC is comprised; used for representing compressed audio on the Mel scale, obtained using a **Mel-scaled Log-magnitude Spectrogram (MLS)** transformation (see *Chapter 2.2.3*).
- **Chromas** – Pitch class profiles that divide a signal into the 12 pitches of the octave [7].
- **Self-Similarity Matrices (SS[L]M)** – A visual representation of similar data sequences.

Acoustic features such as **low-level descriptors** (**LLDs**, features related to a signal's origin), **long-term average spectrums** (**LTAS**, used to view the average power cepstrum over time), **source-filter (LPC) coefficients** (features that model the estimated vocal tract which produced a signal, used in **Linear Predictive Coding** tasks), and **delta/difference** (also **delta-delta**) **coefficients** (the trajectories of MFCCs over time [53]) can also be found using the spectral data of a signal [22].

2.2 Related Work

This section will provide a chronological overview of significant literature related to the usage of machine learning algorithms and/or neural networks in recognizing and analyzing musical form and structure or boundary detection and segmentation.

2.2.1 Hörnel and Menzel, 1998

Hörnel and Menzel's research presents one of the first major projects in **musical structure analysis** using neural networks, seeking to prove the usefulness of **Artificial Neural Networks (ANNs)** in sensitively capturing structural and musical data such that the model is able to learn the characteristics of a composer's personal writing style [4]. The authors present methods of training models to recognize various harmonization styles to enable the prediction of relevant harmonic structure from [homorhythmic] compositions (i.e., baroque era chorales), also noting the difficulty in learning melodies compared to harmonizations. Because of this training discrepancy, the authors suggest the usage of several cooperating networks, each trained on a specific high-level musical structure (melody, harmony, phrases, motifs, etc.) rather than one model with multiple tasks.

While noting that the harmonic structure training and reproduction is easier than melodic, the authors present a series of models for both tasks. The "HARMONET" system uses Feed-forward Neural Networks to produce four-part harmonizations in a specified composer's style given a one-voice melody; the model was originally given textbook rules as constraints to produce "error-free" randomly generated harmonizations, but the result was found to be aesthetically unacceptable in replicating given composers. As such, the system then integrated symbolic algorithms alongside the neural networks, allowing the networks to focus on the creative parts of the generative tasks while conforming to the aesthetics of training examples from a specified

composer, and the conventional algorithms perform the error-correcting tasks such as adjusting pitch ranges and errors in counterpoint. The neural network's creation task is also divided into humanistic sub-tasks, first determining the harmonic structure of a melody, expanding the harmonies to chords, and finally inserting subdivisions (or "non-chord tones") – also using a cross-entropy error function with a softmax output to assist in data normalization during harmony generation, while allowing for "harmonic surprises" also.

Regarding the learning of melodies and melodic variation, the authors note that the recognition and attempted recreation of melodic structure – using a similar applied method to the harmonic structure approach – produces new sequences lacking coherence as a result of the models being unable to capture high-level musical structures occurring simultaneously and at multiple time scales. Thus, a multiscale learning approach is proposed for melodic structure learning, wherein cascaded sequential networks are used as a hierarchical network architecture; sequences of notes for training are divided into subsequence chunks, each associated with a different plan for one of the sequential networks specifically, then leaving a superordinate (or "master") network to learn the sequence of the plans. Again, however, this structure is also unsuitable for detecting higher-level musical structure, even with modifications to the network's memory unit system, as the resulting output of the model still lacks musical coherence.

The authors review a multitude of solutions, including chunking architectures that automatically decompose sequences, explicit structure representation based on formal music analysis, or representing the context of musical elements to capture relationships between parameters in various musical contexts – as such, the "MELONET" system, an architecture unbound by textbook rules, is presented as the solution in this research. MELONET uses the aforementioned multiscale network model to learn and generate structured high-level melodic

sequences and variations of melodies, using two independent interactive networks to combine both **unsupervised** (classifying and recognizing musical structure) and **supervised** (data prediction in time) learning approaches. The process of analyzing and learning phrases is divided into subtasks, first dividing melodies into motives/motifs, defined here as a sequence of four sixteenth notes – to create a simple variation of a four-note subdivision from each note in the original melody – allowing each quarter note to be a varied motif. These motives are then classified by similarity in features such as melodic contour and rhythm. The model then must decide, for a new note within a melodic sequence, what motif fits the best while also considering chronological and harmonic contexts, allowing a subnetwork to produce the new sequence which must then be reclassified before the process repeats.

Further sub-procedures are defined in recognizing melodic versus harmonic context of notes and intervals within a motive for classification as well. However, this approach to musical context analysis has great potential for higher-level scopes, such as defining the average musical phrase as four measures (or bars) long, then classifying phrases based on the motives that appear within each phrase to determine complex form information including repeating phrases appearing in different voices or variations.

2.2.2 Ponce de León and Iñesta, 2007

This research on shallow statistical descriptors seeks to present a framework for the use of pattern recognition algorithms in automatic musical style recognition of digital scores (**Musical Instrument Digital Interface**, or **MIDI** files) through the classification of harmonic, melodic, and rhythmic descriptors [6]. The system applies multiple classification methods, including Bayesian classification, **Self-Organizing Maps (SOMs)**, and **k-Nearest Neighbors (k-NN)**; the various

algorithms are benchmarked against different models and parameters in both jazz and classical music genres. Using this **Music Information Retrieval (MIR)** architecture, the authors discuss the issue of modeling musical style and their solutions of using the system to recognize musical genres over large music databases, stylistic features of composers, musical taste of individual users, as well as generative compositions which conform to a particular style.

One such solution, SOMs, may be especially applicable to formal analysis; an existing proposition utilizes this approach to organize digital music libraries by clustering features found in musical themes for classification purposes. MIDI also provides the potential for a variety of statistical features – including pitch histograms and fast characteristic classification through supervised learning methods such as decision trees, k-means clustering, and k-nearest-neighbor – as well as enabling easier methods of sequence-processing techniques such as hidden Markov models and universal compression algorithms for the classification of musical sequences. While the objective of this research aimed to classify musical styles based on lower-level feature extraction and selection (pitch, note and silence duration, intervals, and distribution normality), the feature extraction approaches demonstrated in the system may be potentially modified to recognize higher-level musical features as well.

2.2.3 Ullrich, Schlüter, and Grill, 2014

Ullrich et al. discuss the importance of boundary recognition in musical structure analysis and present a **Convolutional Neural Network (CNN)** architecture as a potential solution to the task [7]. The network model was trained on annotated **Mel-scaled log-magnitude spectrograms (MLSs)** derived from a subset of the SALAMI (Structural Analysis of Large Amounts of Music Information) annotation dataset, while also maintaining the flexibility of annotation guidelines for

extensions to the architecture. The authors also note that given the varying opinions on annotation placement between human annotators on any same given piece, a large corpus of annotated examples is used in training a neural network rather than designing an algorithm to solve the task. As such, SALAMI – a continuously expanding dataset containing over 1,400 annotated pieces of music in the form of text files with annotated timestamps – was chosen for training data.

For feature extraction, magnitude spectrograms are computed before applying a Mel filter bank, then magnitudes are scaled logarithmically. To aid the network in performing predictions at both the beginning and end of an audio file, the audio spectrogram is padded with pink noise at -70dB (noting that "padding with silence is impossible with logarithmic magnitudes, and white noise is too different from the existing background noise in natural recordings"); subsequently, each frequency band must be normalized to zero mean and unit variance, and the spectrogram is broken down into multiple subsamples of adjacent time frames. The authors also compare multiple audio-training approaches, including Mel-frequency cepstral coefficients (MFCCs), chroma vectors, and derived fluctuation patterns and self-similarity matrices from each, but find that Mel spectrograms performed best and were the most suitable for the algorithm.

A method referred to as "target smearing" is used in accounting for disagreement in human-annotated boundary placement (given the lack of ground truth in formal music analysis), where all excerpts centered on a frame in a user-defined number of adjacent frames are presented as positive examples. The positive examples are then weighted during training by a **Gaussian kernel** (or Radial Basis Function kernel, a kernel function whose output value depends on the distance from some point or the origin) centered on the current boundary. After obtaining the boundary probability for each frame, each output value on the boundary activation curve is analyzed and passed through a threshold to obtain a list of boundary candidates with corresponding strength

values – a method referred to as “peak-picking” that aids in reducing the number of boundary candidates.

During evaluation, the median deviation (or median distance between annotated boundaries and their closest predicted boundaries) and hit rate (which predicted boundaries fall closest to unmatched annotated boundaries) are used in computing the model’s F-measure, precision, and recall. The final model presented in the research strongly outperformed prior algorithms, though the authors note that extending the model into a Recurrent Convolutional Neural Network would likely be more effective at accounting for repeating musical patterns. As well, the abrupt changes from background noise to the added pink noise in the SALAMI audio recordings may have added to the model errors, but an intelligent padding algorithm may be sufficient in reducing this issue.

2.2.4 Grill and Schlüter, 2015

This research utilizes a Convolutional Neural Network to identify boundaries and assign labels to resulting musical segments in digital audio through a cosine distance measure, **Mel-scaled log-magnitude (MLS)** spectrograms, and **Self-Similarity Lag Matrices (SSLMs)** [3]. The model was trained on human-annotated data and treated the objective of boundary prediction as a binary classification problem, where “given an excerpt of an audio signal, decide whether there is a structural boundary at its center or not.” Upon the successful training of a model, the research applies the model to sequences of excerpts through the **sliding-window** method (an algorithm for computing data over an underlying collection, such as the running average or set of all adjacent pairs, for example) to obtain the boundary probabilities as a curve – which is then analyzed for peaks to predict the boundaries of the evaluated piece of music.

Training data for the model was provided by the SALAMI structural annotation dataset using its corresponding annotation guidelines, which provide a basis for formal music analysis primarily through Schenkerian (tonal) analysis or analyzing phrases or form. The research aimed to retrieve the best possible model for submission to the **MIREX (Music Information Retrieval Evaluation eXchange)** evaluation campaign, using datasets disjoint from the data provided by the campaign; through the training on the SALAMI dataset (containing classical and popular music from both studio and live recordings), the model was able to outperform any prior conference submissions (from MIREX 2012 to 2014) and was tuned for a time tolerance in evaluation of ±0.5 seconds.

2.2.5 O'Brien, 2016

O'Brien's research seeks to review current implementations of musical form analyzing neural models and propose an extension to Ullrich's Convolutional Neural Network architecture [5]. The author discusses that "consumer applications such as recommendation systems benefit by taking into account song structure, [a salient aspect of appreciating music; one] could even employ music structure boundaries to automatically generate music "thumbnails" or summaries [as short audio snippets,] sections of the larger work" – this application would be especially useful for a classical music streaming service, not only for those who appreciate the genre but particularly for professional performers searching for new repertoire. One proposed method for performing the task of structure analysis is **Non-negative Matrix Factorization (NMF)**, where a piece of music is transformed into a feature matrix which is factored into multiple lower-dimensional matrices where the outer product equates to the original matrix, representing the activations of a song's basis features in time. Using the derived matrices, identifying musical boundaries becomes easier

using segment association: segments with (closely) identical basis features may reasonably be analyzed as coming from the same thematic material (i.e., phrase or verse).

The biggest issue noted with the CNN model is the lack of model memory – because the network has no memory cells, it cannot recognize repeated verses as being the same (as was possible with the NMF algorithm, though NMF is much slower than a CNN). As such, Ullrich's proposition for a Recurrent Convolutional Neural Network [7], or a Convolutional Sequence-to-Sequence model with both a CNN and RNN using Connectionist Temporal Classification (CTC, for sheet music-based segmentation), may be one of the very few solutions for the task. This is especially problematic for classical music analysis, where phrases and parts are frequently repeated or modified at various points in time within a piece, such as in sonata, ritornello, and rondo forms. The final model was trained on a small portion of the SALAMI dataset with evaluation metrics provided by the "mir_eval" Python package, a library containing metrics for Music Information Retrieval. Audio features are extracted using the Librosa library, used in extracting Mel-scale spectrograms from the audio files.

The author notes that, while the resulting model's performance was promising, the network architecture needs to be expanded to allow for a larger dataset. The current training data is also modified through a random number of insertions of positive examples into the training set – this would be problematic in training a much larger model for a streaming service, for example. As well, while the system was trained according to MIREX evaluation procedures, evaluation metrics are inherently imperfect due to the nature of objectively measuring a task that relies on subjective perception, so interpreting the results of a model must not solely influence the end training goal.

2.2.6 De Berardinis, Vamvakaris, Cangelosi, and Coutinho, 2020

This article presents a discussion on the challenge of automatic musical structure detection in audio recordings and the issue of most current algorithms being only able to produce flat segmentations which lack the ability to segment the music across multiple levels to reveal the piece's hierarchical structure [1]. The authors propose a new approach based on multi-resolution community detection and graph theory to create a new unsupervised learning task, which can perform both boundary detection and structural grouping without the need for specified constraints that limit the output segmentation. Given this approach, the algorithm allows for the ability of high-level (part, section, episode, etc.) and low-level (phrase, motive) analysis. In comparison to Convolution Neural Networks, which employ supervised learning methods to estimate fixed-depth segmentations based on the SALAMI dataset's annotation levels, the authors' Musical Structure Communities (MSCOM) algorithm is entirely unsupervised and can be categorized as a state-based or homogeneity-based procedure that identifies structural patterns derived from their respective acoustic features.

MSOM addresses the task of music segmentation as a community detection problem – that is, the algorithm attempts to find subnetworks with significantly more links between data in the same group than in different groups – a problem predicated on graph theory. A piece of music is first partitioned into numerous consecutive audio frames, which are then used to generate an indirect graph reflecting long-term recurrence and local temporal connectivity information from the frame data. This graph is subsequently processed by a "divisive community detection procedure based on modularity optimization at multiple resolution levels" to yield a structural hierarchy. Given this hierarchy, the first level will contain a single segment encompassing the entire piece, with the deepest level containing as many segments as the number of audio frames.

Additionally, the authors provide an extension to MSCOM (baseline) known as MSCOM dynamic (or DMSCOM) where “the self-similarity matrix [computed] on the beat-synchronized chroma features undergoes a dynamic filtration step before the construction of the recurrence graph,” allowing for finer tuning of the algorithm’s hyperparameters. The algorithm was evaluated on the SALAMI dataset and compared to the pre-existing state of the art methods (such as Laplacian Structural Decomposition and Ordinal Linear Discriminant Analysis) and was found to outperform these prior approaches through evaluation on the L-measure (a measurement of the hierarchical structure levels output from the algorithm). As well, the authors note that the method may also be used for structure visualization and finer-level musical structure analysis using tree representations to reflect additional structural relationships.

CHAPTER 3
METHODOLOGY

This chapter presents the design of the new proposed dataset and describes the structure and components of the feature extraction/data preparation methods, neural network architecture, and the requirements imposed by the system.

3.1 Proposed Dataset

The new dataset – proposed to solve the majority of problems found in the SALAMI annotation dataset (see *Chapter 1.1*) – contains 200 pieces of classical music divided categorically by large musical form classification (**theme and variation**, **ternary**, **through-composed**, etc.) with more weight given to the more complex forms, especially rondo and sonata-allegro (see *Appendix B*). The annotations are formatted in the same style as the SALAMI annotation guidelines: timestamps (in seconds, up to three decimal places) are labeled by section and/or phrase labels, beginning with the large form label, then "Silence" before the intermediate labels and ending with "End" to signify the last second of the piece's duration (see *Appendix A*). The audio data for each piece is stored in MIDI format in a folder corresponding to its large form. Nearly every piece in the dataset is classified by its form discussed in [2] and/or [11] with the intention of correctly analyzing each piece by standardized analytical conventions. The dataset utilizes analyzed sheet music for each piece of music to aid in timestamp analysis, especially if a different recording/MIDI file is desired for revised training data to allow for flexibility of the input audio file – including file format.

3.2 Requirements and Specifications

All components of the Form-NN system were implemented in the Python (3.9) programming language (see *Appendix D*). The project includes numerous popular open-source frameworks for machine learning and signal processing, including:

- **TensorFlow** – A library for symbolic math that supports machine learning operations, deep learning models, and numerical computations (especially for tensor data) [57].
- **Keras** – A subset of the TensorFlow library that provides a high-level deep learning API.
- **Scikit-Learn (Sklearn)** – A library (built on top of SciPy) containing many supervised and unsupervised learning algorithms, including decision trees and ensemble models like random forests [58].
- **Scikit-Image (Skimage)** – An extension of sklearn providing image processing operations.
- **NumPy** – A library for scientific computing, including linear algebra functions, the FFT, matrix operations, and extended array operations [59].
- **SciPy** – A scientific computation library that extends the functionality of NumPy.
- **Matplotlib** – A library supporting data visualization and graphical plotting [60].
- **Seaborn** – A data visualization library built on Matplotlib for statistical graphics plotting.
- **Pandas** – A data analysis and manipulation library which greatly supports tabular (relational) data, extending the functionality of Matplotlib.
- **Librosa** – A signal processing package for musical/audio analysis and MIR systems [61].
- **Pydub** – A library for audio manipulation and segmentation [62].
- **Pickle** – A module for serializing and de-serializing Python objects (such as machine learning models) [63].

3.3 System Design

The backend of the system will consist of three primary components: the **Form Analyzer model**, a **Peak-Picking algorithm**, and the **Phrase Analyzer model** (see Figure 3.2). The Form Analyzer will take an audio file as input and output a predicted **Form** label for the given piece (with the domain of the classes specified in *Appendix B*); a task known as **Multiclass Classification** [64] (see Figure 3.1). The Phrase Analyzer has a much more difficult task – again taking an audio file as input, the audio will first be passed to the Peak-Picking algorithm to detect the onset of musical events (i.e., finding where the audio should be segmented, typically at the point of a cadence – e.g., the end of a phrase). Then, the timestamp of each peak will be used to segment the audio into multiple parts (ideally into individual phrases). The original audio must also be passed to the Form Analyzer so the timestamps can be tagged with the large form to assist the Phrase Analyzer, which will finally output a Part and/or Phrase label (or multiple) for each timestamp. This task, called **Multilabel Classification**, is much more challenging since misclassifications cannot be said to be a hard right or wrong [64] – another reason why form analysis is difficult, especially since the results are **objective to the analyst**, rather than ground truth.

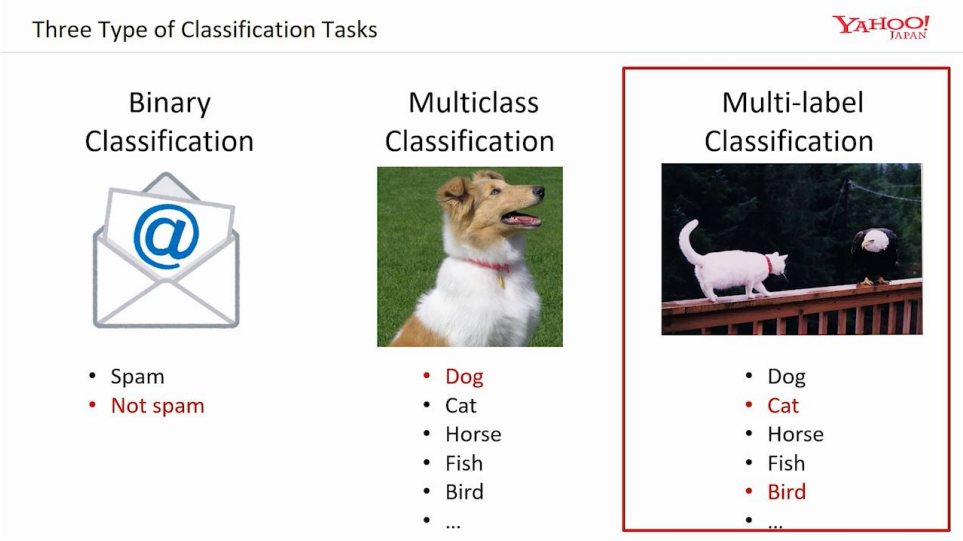


Figure 3.1: Comparison of Classification Tasks [65]

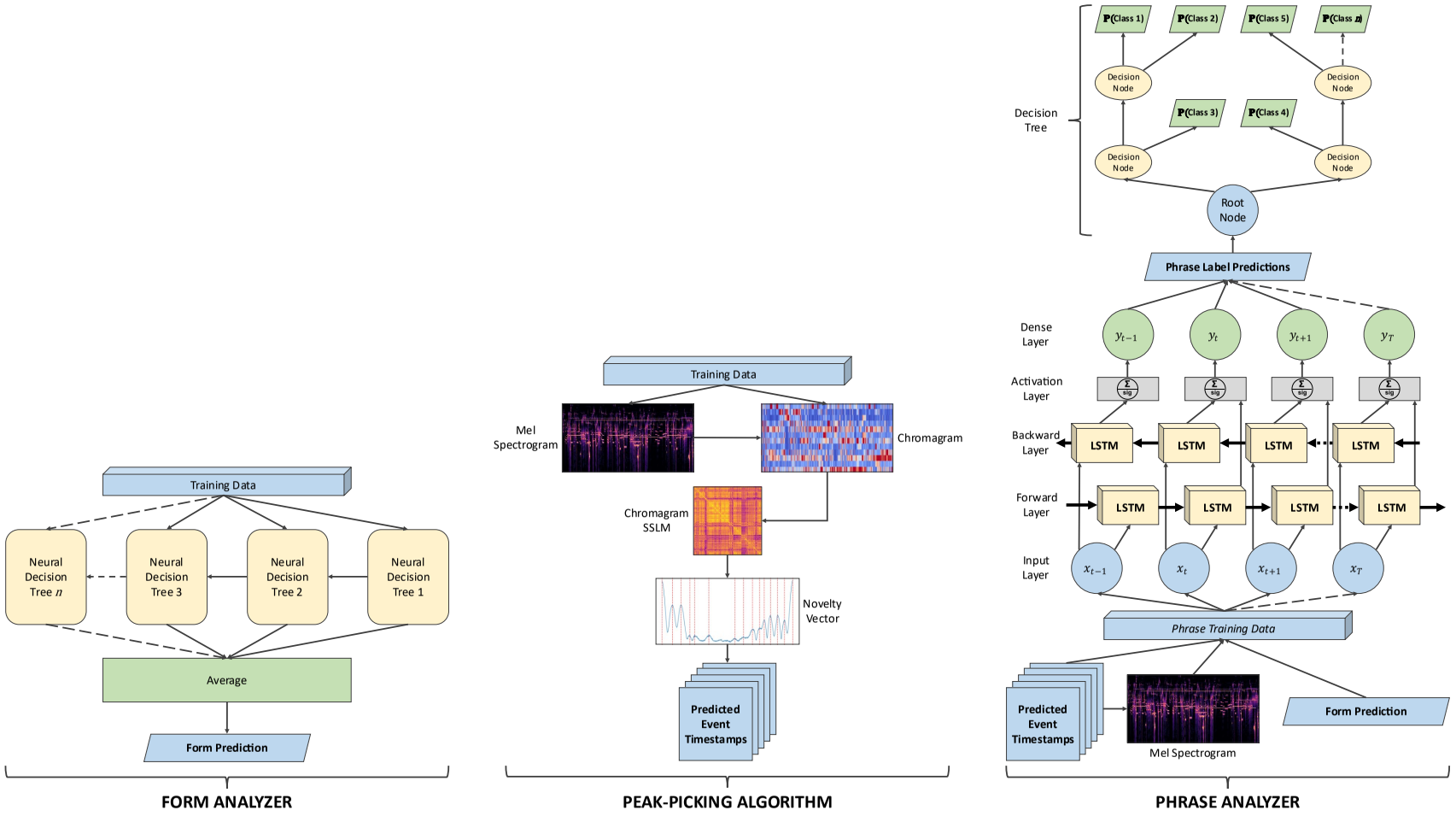


Figure 3.2: System Architecture Diagram

CHAPTER 4
IMPLEMENTATION

In this chapter, we will discuss the approach of preparing the dataset for the system, extracting relevant features from the raw audio data, and the architecture, implementation, and limitations of the classification models.

4.1 Data Preparation

Initially, we tried using the images of the Mel Spectrogram SSM, MFCC Spectrogram SSLMs (by Euclidian and Cosine distances), and Chromagram SSLMs to train a CNN using 2D convolutions. However, this was found to be inefficient for numerous reasons, including the storage space necessary for all 5 images per piece of music, the training time (on a GTX 1050 Ti GPU) was often extremely long – especially if dropout was added to the model to decrease validation error, and the model never achieved an accuracy above 13%. Instead, all audio features (see *Chapter 4.1.2*) were calculated from the audio files ahead of time, and the mean and variance arrays calculated from the returned spectrograms were stored in a tabulated data file – reducing the 2D spectral data into two 1D arrays. This is a technique known as **Dimensionality Reduction**, which helps reduce the complexity of a model (a problem known as the **Curse of Dimensionality**, see *Chapter 4.1.1*) and increase prediction accuracy [67].

4.1.1 Curse of Dimensionality

To improve the training time of the model and aid further research with the proposed dataset, the final tabulated dataset contains the mean and variance arrays for the following features: Mel Spectrogram SSM, MFCC Spectrogram SSLMs (Euclidian and Cosine distances), Chromagram SSLMs (Euclidian and Cosine distances), large form classification (see *Appendix B*), and 15 other feature sets (see *Appendix B.1*) that may be useful for training a model. However, expanding these arrays into individual feature columns yields over 16,000 features for a dataset of only 200 rows (datapoints, i.e., audio files) – an extremely complex training set (referred to as **high dimensional data**). Using all these features causes the **Curse of Dimensionality** – as the number of features increases, the dataset becomes sparser (i.e., the data becomes more separated, see Figure 4.1), and the number of data points needs to be increased to balance it [68] – or the dimensionality needs to be reduced to fit the number of datapoints more properly (see *Chapter 4.1.3*). This also leads to another issue known as **overfitting**, where the model performs well on the training dataset, but fails to generalize on new unseen data (e.g., the validation/test dataset(s)) because it learned both the training details and noise exactly (see *Chapter 5.2.1*).

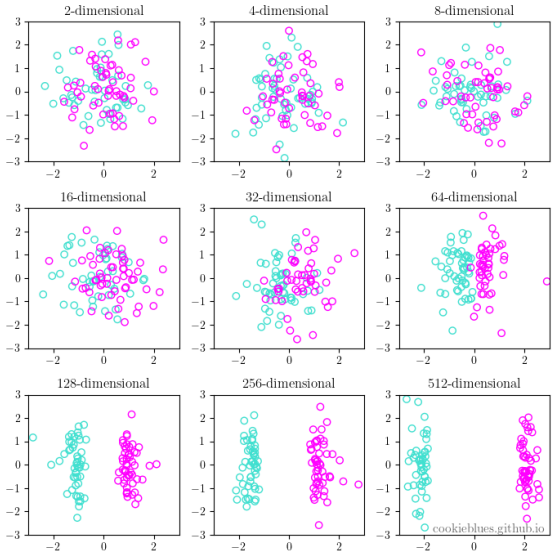


Figure 4.1: Illustration of the Curse of Dimensionality as features increase [68]

4.1.1.1 Data Augmentation

Given that the initial models were overfitting even with a small number of features selected (just the SSLMs), it became a priority to increase the size of the dataset. However, analyzing the form of another 200-1,000 pieces of music was an infeasible option due to the amount of hand analysis, data filtering, and file scraping (for MIDIs and the PDFs of the sheet music) necessary to collect more training data. Instead, the more realistic option was to **augment** the data – i.e., modify the data of the existing training set to create new data points. For audio, four augmentation methods were selected: **speed shifting**, **pitch shifting**, **time-shifting**, and **noise injection**. These methods were used in combination with various arguments to generate 5 new copies of the dataset (see Table 4.1). Since the musical form is based on the musical structure, none of these augmentations create improperly labeled data points and they can retain their original (un-augmented) label – thus, the training set was expanded to 1,200 total data points. This also helps to lessen the problem of **class imbalance**: there are uneven numbers of data points per class (form), so labels with more occurrences tend to have more bias in an improperly trained model [69].

	Aug_Set 1	**Aug_Set 2**	**Aug_Set 3**	**Aug_Set 4**	**Aug_Set 5**
Speed Shift Factor	0.7 (Slower)	1.4 (Faster)	0.5 (Slower)	2 (Faster)	1.1 (Faster)
Pitch Shift Factor (Half Steps)	-6 (Down 6 HS/a tritone)	4 (Up 4 HS/a major 3rd)	7 (Up 5 HS/a perfect 5th)	8 (Up 8 HS/a minor 6th)	1 (Up 1 HS/a minor 2nd)
Time Shift Factor/Direction	Random(0, 1), to the right				
Noise Injection Factor	0.005	0.01	0.03	0.02	0.007

Table 4.1: Modification arguments of each augmented dataset

Shifting the time and injecting noise are the two simplest operations – time-shifting simply involves modifying the starting (and/or ending) point of the audio by padding the signal (treated as an array of **audio frames**) with silence:

Algorithm 1 Shift Time

```
function SHIFT_TIME(audio_data , sampling_rate , shift_max , shift_direction):
    shift = RANDOMINT(0 , sampling_rate ∗ shift_max)
    if shift_direction == "right":
        shift ∗= −1
    # Move elements in array shift places to the right
    augmented_signal = ROLL(audio_data , shift)

    # Set empty audio data to 0 (silence)
    if shift > 0:
        # Set all audio from [0−shift] to 0
        augmented_signal[:shift] = 0
    else:
        # Set all audio from [shift−end] to 0
        augmented_signal[shift:] = 0
    return augmented_signal
end
```

Likewise, we can inject noise into the signal by adding a noise factor (a small decimal) to each audio frame, then multiply the entire signal by a random noise distribution using the **standard normal** (i.e., a bell curve):

Algorithm 2 Inject Noise

```
function INJECT_NOISE(audio_data , noise_factor):
    noise = RANDOMNORMAL(audio_data.LENGTH())
    augmented_signal = audio_data + noise_factor ∗ noise
    return augmented_signal
end
```

We can calculate the **normal distribution** $\mathcal{N}(\mu, \sigma^2)$ as

$$\varphi(x) = \frac{e^{-\frac{x^2}{2}}}{\sqrt{2\pi}} \tag{4.1}$$

and select *n* frames to randomly sample from the distribution, where *n* = the number of frames in the audio signal. This is defined as a special case for the **probability density function**

$$\varphi(x) = \frac{e^{-\frac{(x-\mu)^2}{(2\sigma^2)}}}{\sigma\sqrt{2\pi}} \tag{4.2}$$

where $\mu = 0$ (the mean) and $\sigma = 1$ (the standard deviation; hence, σ^2 in $\mathcal{N}$ is the variance).

Pitch and speed shifting is a much more difficult task, however. Speed shifting must be done in such a way that it does not influence the pitch unless desired – hence, we can use speed shifting to modify the pitch, but the speed should be independent:

Algorithm 3 Shift Pitch

```
function SHIFT_PITCH(audio_data , sampling_rate , num_steps , notes_per_octave=12):
    rate = 2^(−(num_steps)/notes_per_octave)
    augmented_signal = RESAMPLE(
        TIME_STRETCH(audio_data , rate) , # Shift speed
        sampling_rate / rate , sampling_rate
    )
    augmented_signal = FIX_LENGTH(audio_data , augmented_signal)
    return augmented_signal
end
```

Algorithm 4 Fix augmented signal length

```
function FIX_LENGTH(audio_data , augmented_signal):
    if augmented_signal.LENGTH() < audio_signal.LENGTH():
        # Pad signal with 0 to match old LENGTH
        augmented_signal = pad(augmented_signal , [0 ∗ audio_signal.LENGTH() −
augmented_signal.LENGTH()])
    else if augmented_signal.LENGTH() > audio_signal.LENGTH():
        augmented_signal = augmented_signal[:audio_signal.LENGTH()]
    return augmented_signal
end
```

Augmenting the pitch will require the audio to be resampled:

Algorithm 5 Resample in high quality using Kaiser window

```
function RESAMPLE(audio_data , old_sr , target_sr):
    if old_sr == target_sr:
        return audio−data
    ratio = target_sr / old_sr
    num_samples = ⌈ audio_data.SHAPE[−1] ∗ ratio ⌉
    augmented_signal = KAISER_RESAMPLE(audio_data , old_sr , target_sr)
    augmented_signal = FIX_LENGTH(augmented_signal , num_samples)
    return augmented_signal
end
```

The KAISER_RESAMPLE function refers to the **resampy** Python library's **resample** method, which computes the final resampling using a pre-computed Kaiser window filter –

$$w[n] = L \cdot \mathrm{w}_0\left(\frac{L}{N}(n - N/2)\right) = \frac{I_0\left[\beta\sqrt{1 - \left(\frac{2n}{N} - 1\right)^2}\right]}{I_0[\beta]}, \qquad 0 \le n \le N, \qquad \beta = \pi\alpha \tag{4.3}$$

where:

- I_0 is the 0^{th}-order modified Bessel function of the first kind (see Equation 4.4),
- L is the window duration,
- α is a positive real number used to determine the window shape, and
- the length of the window is $N + 1$ (even or odd),

with $\beta = 14.769656459379492$ – then returns the interpolated result [71]. More details of this implementation can be found in [70]. The Kaiser window is difficult to compute, so pre-computing this filter saves heavily on the runtime of the resampling algorithm. The modified Bessel function of the first kind is defined as

$$I_\alpha = i^{-\alpha} J_\alpha(ix) = \sum_{m=0}^{\infty} \frac{1}{m!\,\Gamma(m + \alpha + 1)}\left(\frac{x}{2}\right)^{2m+\alpha} \tag{4.4}$$

where Γ is the Gamma function [72].

Finally, we can implement the speed shift algorithm using the STFT (see Equation 2.1):

Algorithm 6 Shift Speed

```
function TIME_STRETCH(audio_data , speed_factor):
    stft_data = STFT(audio_data)
    stft_stretch = PHASE_VOCODER(stft_data , rate)
    stretch_len = int(⌊ audio_data.LENGTH() / speed_factor ⌉ )

    # Inverse STFT
    augmented_signal = ISTFT(stft_stretch , stretch_len)
    return augmented_signal
end
```

However, we also need two additional algorithms to perform the speed augmentation: the **Inverse STFT (ISTFT)** and a **phase vocoder**. Using the STFT coefficients $\mathcal{X}(n,k)$ obtained from the transposition of the initial discrete transformation, we can reverse the windowing process by obtaining the superposition over all shifted versions of windowed sections of the audio signal:

$$\sum_{n\in\mathbb{Z}} x_n(r-nH) = \sum_{n\in\mathbb{Z}} x_n(r-nH+nH)w(r-nH) = x(r)\sum_{n\in\mathbb{Z}} w\,(r-nH) \tag{4.5}$$

Then, the samples $x(r)$ can be recovered by

$$x(r) = \frac{\sum_{n\in\mathbb{Z}} x_n(r-nH)}{\sum_{n\in\mathbb{Z}} w\,(r-nH)} \tag{4.6}$$

under the constraint

$$\sum_{n\in\mathbb{Z}} w(r-nH) \neq 0, \qquad \forall r \in \mathbb{Z} \tag{4.7}$$

This is referred to as the **overlap-add (OLA)** method, where reconstructed windowed sections that overlap are overlaid and summed, then normalized to compensate for the windowing [74].

The phase vocoder is a variation on the STFT that uses phase information to improve frequency estimates, analyzing an input signal using the FFT to decompose a signal into frequency components [75] (more details of the phase vocoder implementation can be found in [73]).

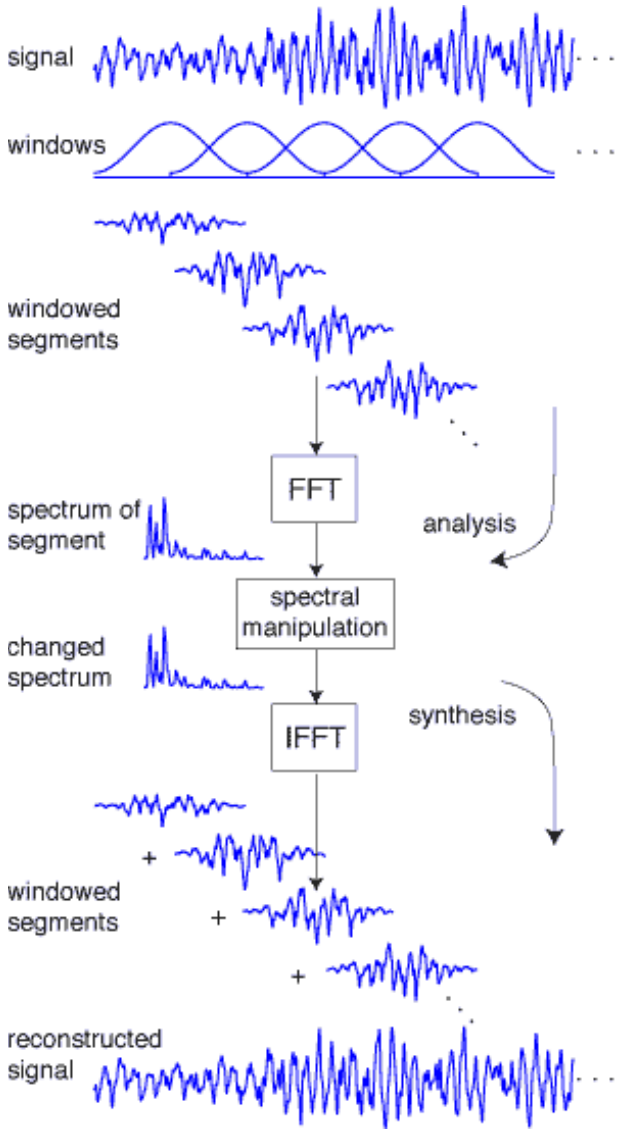


Figure 4.2: Illustration of phase vocoder system [76]

After synthesis, the returned frequency components can be passed back through the ISTFT and transformed to reconstruct the signal with the now shifted pitch or speed. Hence, we can safely (and quickly) modify both speed and pitch independently without losing audio quality or introducing a high degree of noise.

4.1.2 Feature Extraction

Through feature elimination (see *Chapter 4.1.3*), we found that the two most important audio features that influenced the form classification were the **duration** (the length of each piece) and the **(log scaled) Mel Spectrogram SSM** (see *Appendix A*). This greatly reduced the amount of computational overhead added when preparing new training and testing data, especially for the prediction system. To obtain the duration of the signal, we can simply divide the number of audio frames by the **framerate** (frames per second) to get the signal length in seconds. The Mel SSM, however, will require numerous steps.

We first obtain the Mel spectrogram by first resampling the audio with a sample rate of 44.1 kHz (see *Chapter 4.1.1.1*), then computing the STFT over the entire signal (see Equation 2.1) with a hop length of 6144 (0.139 seconds per frame) and 8192 samples per frame (i.e., a window size of 0.209 seconds per frame multiplied by the sampling rate). We can view the output spectrogram (see Figure 4.3) and convert the amplitude to decibels using

$$amp\_to\_db = 20 * log_{10}\left(amplitude^2 / \, max(amplitude^2)\right) \tag{4.8}$$

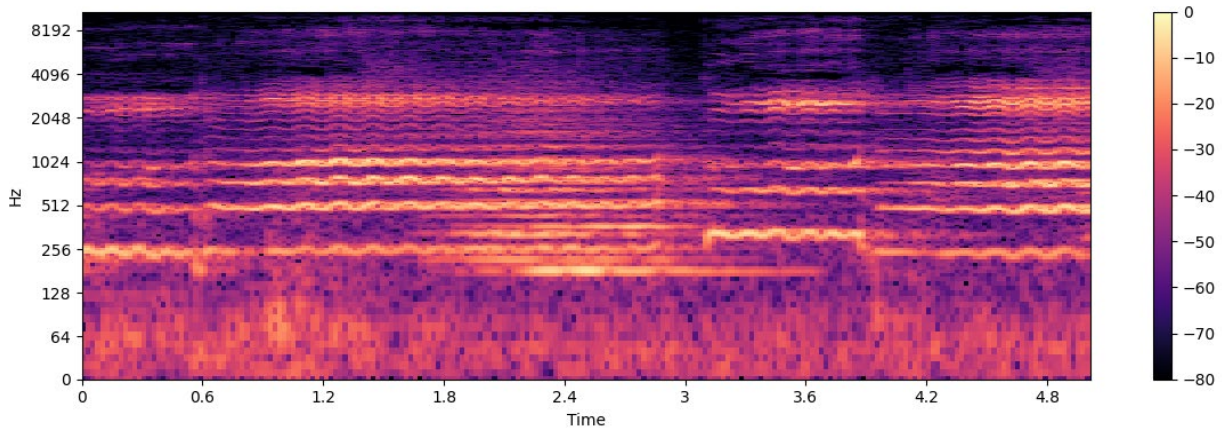


Figure 4.3: 5-second Log Scaled Mel Spectrogram sample [22]

Then, the Mel spectrogram needs to be converted into a recurrence matrix (SSM). This was done using the **librosa** Python library's **recurrence_matrix** function, which can be found in [81]. The function performs the following tasks (see *Chapter 4.1.2.1*):

- Swap the *x* and *y* axes, then flatten the 2D array into a 1D array
- Perform a **k-Nearest Neighbors** search for each frame (using *k*=13 for the MLS) to cluster frames by their *k* neighbors in a graph (i.e., the *k* most similar frames by distance) [82]
- Remove all diagonal connections in the graph (by setting each diagonal to 0)
- Omit extraneous neighbors by retaining only the top-*k* links per point
- Reset the diagonal connections to 1 and symmetrize the matrix using its transposition
- Transpose the graph to become a column-major matrix and return

The result is a 2D recurrence matrix mapping the Fourier coefficient into neighboring vectors (see Figure 4.4 (b)).

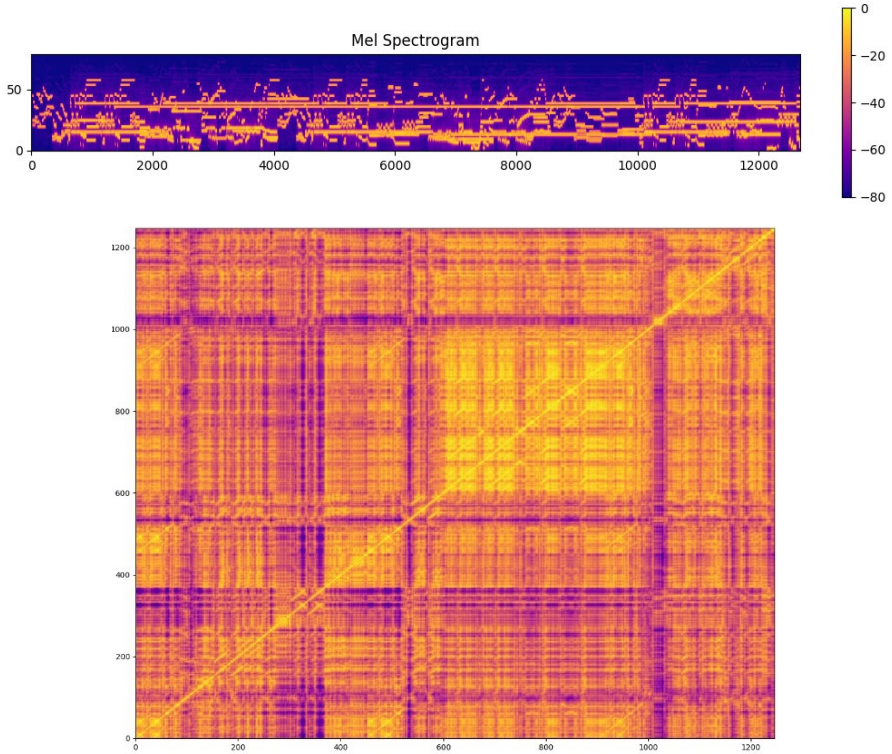


Figure 4.4: Mel Log-scaled Spectrogram (a) and Self-Similarity Matrix (b)

4.1.2.1 Novelty Onset Detection Algorithm

Peak-Picking (or **onset detection**) is the extraction of the frequencies of the peaks within a signal or region of the signal, which has numerous uses including structural segmentation [80]. Using a peak-picking algorithm, we can approximate the location of the musical segments and return them as an array of timestamps, much like the annotations for the SALAMI dataset. First, we need the Mel Spectrogram SSM (see *Chapter 4.1.2*) to provide the algorithm with the power matrix for the temporal data in the spectrogram. We also need to obtain the **Chromagram** of the audio signal – an octave-independent measure of the spectral frequencies mapped to all possible notes in the 12-note scale, allowing us to view the distribution of pitches over time grouped by pitch-class [77]. Given each Fourier coefficient $\mathcal{X}(n,k)$ with a sampling rate F_s (in Hz) and physical time position $T_{coef}(n) = nH/F_s$ (in seconds, see *Chapter 2.1.5*), we can find the physical frequency

$$F_{coef}(k) = \frac{k \cdot F_s}{N} \tag{4.9}$$

then map the frequency to the 12-note equal-tempered scale using

$$F_{pitch}(p) = 2^{\frac{(p-69)}{12}} \cdot 440 \tag{4.10}$$

where (p - 69) refers to the difference between the pitch and the MIDI value for the note A4, and 440 (Hz) is the actual frequency of A4 [79]. Then, each spectral coefficient should be assigned to the pitch with a center frequency closest to $F_{coef}(k)$ to derive a logarithmic frequency axis – hence, for each pitch $p \in [0:127]$ we define the set

$$P(p) \coloneqq k : F_{pitch}(p - 0.5) \leq F_{coef}(k) < F_{pitch}(p + 0.5) \tag{4.11}$$

Using the set of all $P(p)$, we obtain the log-frequency spectrogram $\mathcal{Y}_{LF} : \mathbb{Z} \times [0:127]$ using the pooling procedure

$$\mathcal{Y}_{LF}(n,p) \coloneqq \sum_{k \in P(p)} |\mathcal{X}(n,k)|^2 \tag{4.12}$$

Each pitch in $P(p)$ can be separated into two components – the **tone height** and **chroma**, where tone height refers to the **octave number** (the "4" in A4, typically in the range [0 – 8]) and chroma is the respective pitch within the set {C, C#, D, D#, E, F, …, B}. This set can be enumerated as the identity set [0 : 11], where 0 refers to chroma C, 1 to C#, etc. Thus, **pitch-class** is defined as the set of all pitches of the same chroma, such as the pitch class set for the chroma C : {…, C0, C1, C2, …} which consists of all pitches separated by the integer value of the number of octaves. These **chroma features**, also interchangeably referred to as **pitch class profile (PCP)** features, allow all spectral information related to a given pitch to be aggregated into a single coefficient. Given the pitch-based log-frequency spectrogram $\mathcal{Y}_{LF} : \mathbb{Z} \times [0:127] \to \mathbb{R}_{\geq 0}$, the chromagram (or **chroma representation**, see Figure 4.5) $\mathbb{Z} \times [0:11] \to \mathbb{R}_{\geq 0}$ can be derived from the sum of all pitch coefficients belonging to the same chroma

$$C(n,c) := \sum_{\{p \in [0:127] \,:\, p \bmod 12 = c\}} \mathcal{Y}_{LF}(n,p) \tag{4.13}$$

for $c \in [0:11]$. This provides us with a time series $chromas = X = [x_1, \ldots, x_{N'}]$ of length N' where each x_i is a column vector of the chroma matrix.

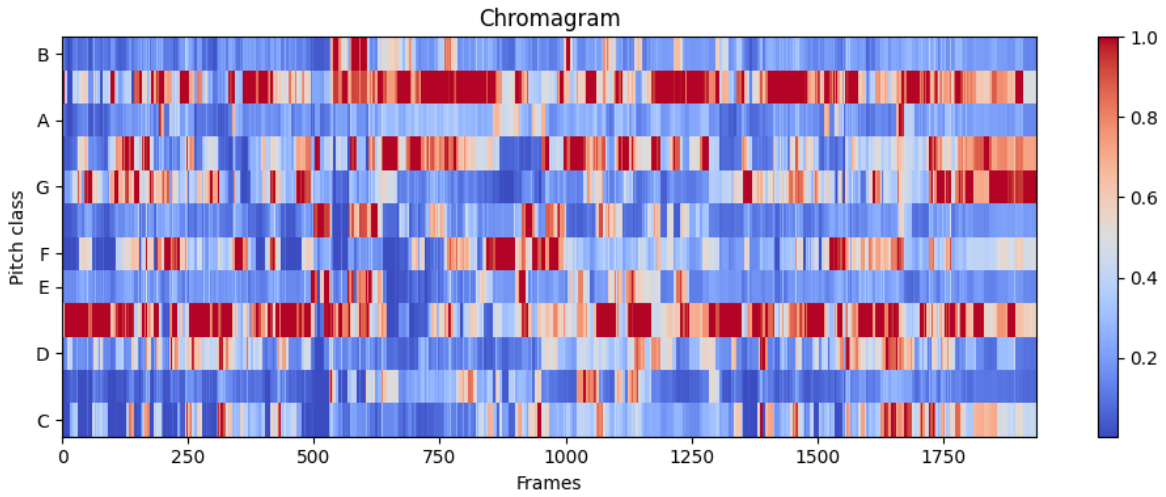
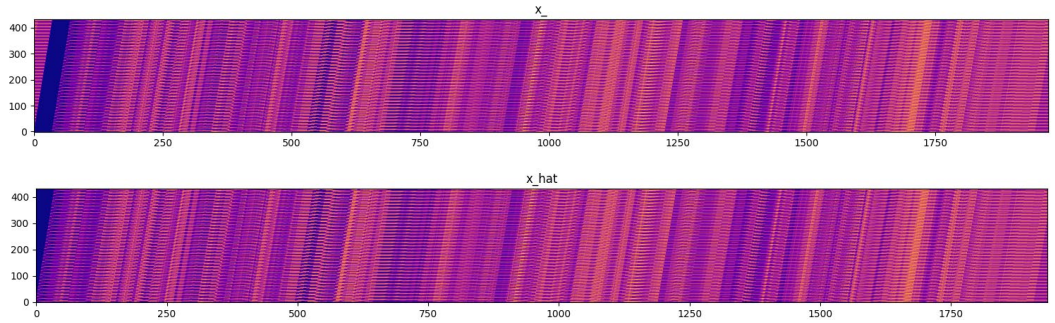


Figure 4.5: Sample Chromagram

To build the recurrence matrix, we need to construct new samples by concatenating the chromas (see Figure 4.6 (a)) with themselves after adding a time delay $\tau = 1$ (sample) to their transposition and embedding them:

$$\widehat{X}_\iota = \left[x_i^{\mathrm{T}}, x_{i-\tau}^{\mathrm{T}}, \ldots, x_{i-(m-1)\tau}^{\mathrm{T}}\right] \tag{4.14}$$

where m = 5 seconds (5 * 44,100 / 6144) = 36 samples (see Figure 4.6 (b)).



(a)

(b)

Figure 4.6: Chroma vector X (a) and transposed vector $\hat{X}$ (b)

We can then calculate the SSM R of $\hat{X}$ (see Figure 4.7 (b)) using the **recurrence_matrix** function from Librosa (see *Chapter 4.1.2*), then convert it to an SSLM L through the creation of a second matrix by circularly shifting the rows of R such that

$$L_{\mathrm{i,j}} = R_{\mathrm{k+1,j}} \tag{4.15}$$

for $i, j = 1, \ldots, N$, where $k = i + j - 2 \; mod \; N$ (see Figure 4.7 (c)) [80].

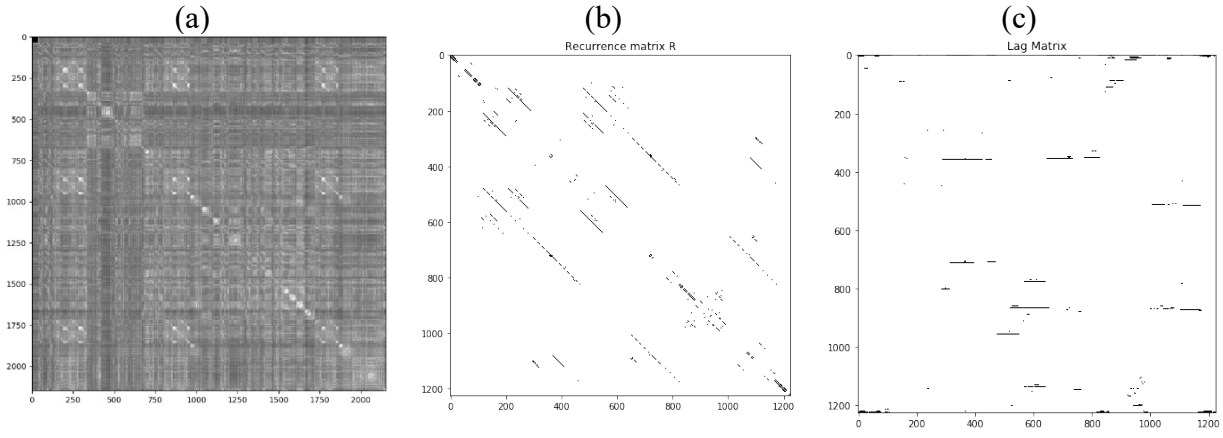


Figure 4.7: Recurrence Matrix R (a), R after k-NN (b), and Lag Matrix L (c)

After obtaining the lag matrix, we can construct a **Gaussian Kernel** G composed of two gaussian windows g_1 and g_t with sizes $s_1 = 0.3$ and $s_t = 30$ seconds, respectively:

$$\begin{aligned} \mathrm{s}_1 &\coloneqq \lfloor 0.3 * \mathrm{F\_s} / \mathrm{H} \rceil, & s_t &\coloneqq \lfloor 30 * F\_s / H \rceil \\ \sigma_1 &\coloneqq (\mathrm{s}_1 - 1)/(2.5 * 2), & \sigma_t &\coloneqq (s_t - 1)/(2.5 * 2) \\ g_1 &\coloneqq gaussian(s_1, \sigma_1).flatten, & g_t &\coloneqq gaussian(s_t, \sigma_t).flatten \\ & G = g_1 \times g_t^{\mathrm{T}} \end{aligned} \tag{4.16}$$

where *gaussian* refers to the normal distribution function (see Equation 4.1). Then, we can convolve matrix L with the new filter G to obtain the distribution matrix P

$$P = L * G \tag{4.17}$$

which can be used to construct the Novelty Curve as a vector $\vec{c} = [c_1, \dots, c_{N-1}]$ (see Figure 4.8):

$$c_i = norm(p_{i+1}, p_i) \tag{4.18}$$

where *norm* refers to the Euclidian (ℓ_2) norm $\left|\left|x\right|\right|_2 \coloneqq \sqrt{x_1^2 + \cdots + x_n^2}$, and p is a column of P.

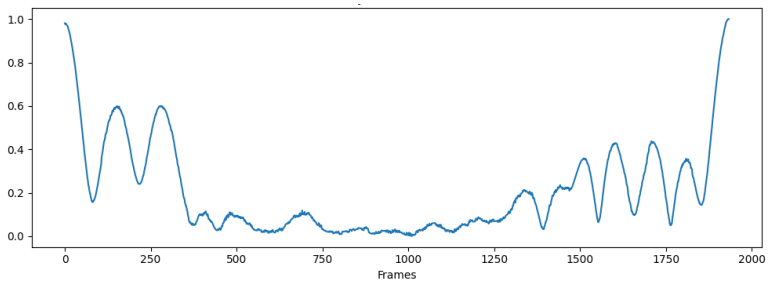


Figure 4.8: Novelty vector $\vec{c}$

Next, the peak selection step is performed by choosing the positions of the most prominent peaks in $\vec{c}$. A sample c_i is a peak if it is above a specified threshold δ (0.05) and corresponds to the global maximum of a window of length λ (6) centered at c_i. Because of the offset introduced by X̂, the exact boundary locations (timestamps) in the original time series X are set to the *selected peak locations* + *w* / 2:

$$\vec{b} = find\_peaks(\vec{c}, \delta, \lambda, width = \lfloor 0.5 * F_s/H \rceil), \qquad \lambda = \lfloor 6 * F_s/H \rceil \tag{4.19}$$

where *find_peaks* is a function that returns the maxima of the vector within the given constraints.

The resulting boundary vector can also be plotted against the novelty curve to display the locations of the peaks (see Figure 4.9). More details of the full peak-picking algorithm can be found in [80].

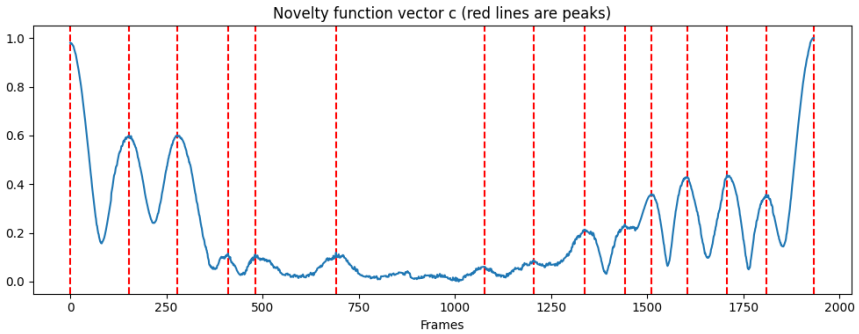


Figure 4.9: Novelty vector $\vec{c}$ with peak-picked boundaries

While further modifications to this algorithm are proposed by the original authors, we found that this is sufficient enough for the sake of program runtime (see *Appendix A*); additional computations include calculating a cumulative matrix for building a segment transitive binary similarity matrix to enforce transitivity and compute segment-segment similarities, but this greatly increases the number of matrix computations, and thus adds substantial overhead. We compared the output of the novelty function to numerous hand-labeled pieces from the dataset and found that the difference was negligible; hence, it was more feasible (and both faster and accurate) to use the algorithm than to train a CNN to perform the same task (see *Chapter 2.2*). It is worth noting that one piece in particular, Prelude, Op. 28, No. 7 by Chopin, is too short for the algorithm – the piece is under 40 seconds in length, and the resulting novelty curve has no peaks (see Figure 4.10). To compensate, we simply add a peak at the halfway timestamp (i.e., *duration* / 2).

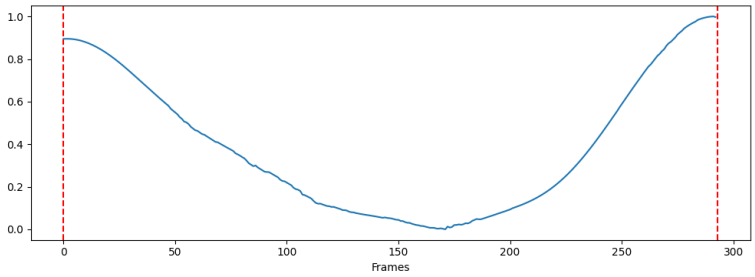


Figure 4.10: Novelty curve for Chopin's Prelude, Op. 28, No. 7

4.1.3 Preparing Training Data

With the newly increased (augmented) dataset, we used a decision tree as a base model for measuring the noisiness of the dataset (i.e., its potential for overfitting in a neural model) in preparation for the form classification model. Initially, we tried using all the features of the dataset (see *Appendix B.1*), but this of course yielded terrible accuracy. To mitigate this, we removed features by the right-most column one at a time until only the means and variances for the Mel SSM, MFCC SSLMs, and Chroma SSLMs remained, substantially boosting the accuracy of the test model nearly to 50%. However, this clearly meant that the dataset contained a lot of noise still, and attempting to train a CNN on the decreased feature set resulted in overfitting no matter what parameters were chosen.

We then used **Select k-Best (SKB)**, a **feature selection** algorithm provided by the **scikit-learn** Python library, to narrow down the number of features even further. This algorithm selects *k* features according to a specified feature selection method – in this case, we use the **Analysis of Variance (ANOVA) F-value** (the variation between sample means/variation within the samples) to remove all but the first *k* highest scoring features [83]. These F-values can also be plotted to display the importance of each feature (see Figure 4.11). Hence, this algorithm confirmed that the most important features in the dataset were the mean of the Mel SSM and the duration of the piece.

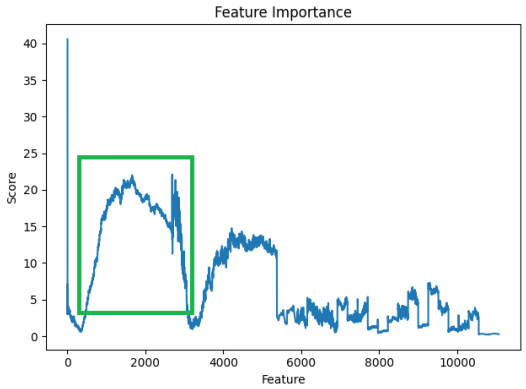


Figure 4.11: Feature importance using SKB (Mel SSM marked in green, index 0 is duration)

However, not every value in the Mel SSM array was equally important – most arrays had to be padded with numerous 0s to match the length of the longest piece, otherwise, the column contained null values. We initially began with the k=1000 best features but this still contained a lot of padding values (i.e., noise, leading to the test models still overfitting), so we eventually reduced the number of features to k=15 by trial and error – leading to the decision tree test model to its highest accuracy of 84%. We also tried using **Recursive Feature Elimination (RFE)** to reduce the number of features (detailed in [84]), but this was both slower and performed identical or worse compared to SKB. Finally, the dataset X was normalized using (X - mean(X)) / standard-deviation(X).

For the phrase classification model, we can use an approach similar to the form classifier without the need for calculating the SSM using the Peak-Picking algorithm to obtain approximate timestamps for the beginning/end of phrases (e.g., the cadence points; see *Chapter 4.1.2.1* – note that this is only necessary for unseen/testing data, as the training set includes these timestamps). Then, we can split the signal into multiple parts based on the set of timestamps, (i.e., the set of timestamps [0, 15, 40, 50, …] would yield segments of the original audio with the durations [15, 25, 10, …]) which will be stored alongside the new split signals, as splits of similar or equal length should assist the model in recognizing repeated phrases. We also store the form label with each audio split so the model can build a correlation between the form, length, and melodic/harmonic content of each phrase; for unseen data, we must rely on the form classifier to provide this label, meaning that it must be as accurate as possible. We can then calculate the Mel spectrogram (see Equation 2.1) for each audio split to measure the power of the signal over the frequency distribution to store the musical content of the segment [22] (see *Appendix A*), then return the mean (in this case, the temporal average) of the spectrogram to obtain the log scaled Mel spectrogram. The dataset X was normalized using Min-Max Scaling (X – min(X)) / (max(X) – min(X)).

4.2 System Components

The full system contains numerous components – primarily data preparation functions (see *Appendix D*) – though we will focus on the two most important ones, i.e., the deep learning models. The first is the **Form Analyzer**, which is comprised of one deep learning model extensively trained on the full augmented dataset. The second component, the **Phrase Analyzer**, features multiple models for the sake of output comparison as the dataset was much smaller – to use the full dataset, all 1,000 augmented files would have to be re-analyzed and timestamped by hand for every event, then compared to the original for the ground truth labels. However, the final model still performed well enough to provide usable results, and the output of two additional models is output alongside it for evaluation. To properly analyze the phrases of new (unseen) data, the model requires both the output from the Form Analyzer and the **Peak-Picking** algorithm, so these components in particular hold great importance in the accuracy of the final system (see *Chapter 3.3*).

4.2.1 TreeGrad – Form Classification

The Form Analyzer model was the most difficult architecture to develop. After trying numerous neural network architectures (CNNs, DNNs, N-SVM, etc. – see *Chapter 5.2.1*), it appeared that none would reach a validation accuracy above 50%. We tried applying numerous methods of regularization, including **kernel** and **bias regularization** (ℓ_1, ℓ_2, and both combined, used to constrain/regularize the coefficient estimates toward 0 [89]), **Dropout** (ignoring a percentage of random neurons to enforce generalization [90]), and adding **Gaussian noise** – however, none of these improved the overfitting more than a ~15% increase at most. Strangely, the NN models performed worse when more features were removed – especially on the SKB dataset (see *Chapter 4.1.2*) – but the decision tree test models were performing well. This correlation greatly inspired a new set of architecture changes: neural decision trees and decision tree neural networks.

The final model was thus implemented using the **TreeGrad** architecture, an extension of another architecture known as **Deep Neural Decision Forests** [91]. A decision tree is normally defined as a tree-structured classifier that consists of **decision** (split/internal) nodes, indexed by $\mathcal{N}$, and **prediction** (leaf/terminal) nodes indexed by $\mathcal{L}$ (see *Chapter 2.1.1.1*) [92]. The classification problem has **input space** $\mathcal{X}$ and (finite) **output space** $\mathcal{Y}$. All prediction nodes hold a probability distribution $\frac{\pi_\ell}{y}, \forall \ell \in \mathcal{L}$, and all decision nodes are assigned a decision function $d_n(\cdot;\Theta) : \mathcal{X} \to [0,1], \forall n \in \mathcal{N}$ parameterized by Θ which controls the routing samples along the tree. Then, when a sample $x \in \mathcal{X}$ reaches a decision node *n*, it will be split to the left or right subtree based on the output of the decision $d_n(x;\Theta)$. However, as opposed to a normal decision tree where d_n is binary and has deterministic routing, a neural decision tree utilizes probabilistic routing; hence, the final prediction from tree *T* for sample *x* with decision nodes parameterized by Θ is given by

$$\mathbb{P}_T[y|x,\Theta,\pi] = \sum_{\ell \in \mathcal{L}} \pi_{\ell_y}\, \mu_\ell(x|\Theta) \tag{4.20}$$

where $\pi = (\pi_\ell)_{\ell \in \mathcal{L}}$ and π_{ℓ_y} denotes the probability of a sample reaching leaf ℓ and being labeled as class y, with $\mu_\ell(x|\Theta)$ representing the routing function which provides the probability of sample *x* reaching leaf ℓ, and $\sum_\ell \mu_\ell(x|\Theta) = 1, \forall x \in \mathcal{X}$ [92, pp. 2-3]. Lastly, we consider each decision node *n* to have stochastic routing using the decision function

$$d_n(x;\Theta) = \sigma\big(f_n(x;\Theta)\big), \qquad \sigma(x) = (1 + e^{-x})^{-1} \tag{4.21}$$

where $\sigma(x)$ is the **sigmoid function** (which defines the probability of routing *x* to binary nodes under node *n* – i.e., either to the left or right subtree of node *n*) and $f_n(\cdot;\Theta) : \mathcal{X} \to \mathbb{R}$ is a real-valued function that depends on the sample and the parameterization Θ. More details can be found in [92].

The neural decision tree clearly follows the same framing as a neural network: a **decision (input) layer**, a **routing (hidden) layer**, and a **prediction (output) layer**. Using the sigmoid decision function, we can express the final neural decision tree model as

$$\mu_\ell(x|\Theta) = \prod_{n \in \mathcal{N}_\ell} d_{n_+}(x;\Theta) d_{n_-}(x;\Theta) \tag{4.22}$$

where $\mathcal{N}_\ell$ represents the set of nodes that are part of the route for leaf ℓ and d_{n_+}, d_{n_-} indicate the probability of moving to the positive and negatives routes of node *n*, respectively [91]. Hence, the final layer acts as a dense (**fully connected**) layer from the respective routing layer, leading to the prediction of the neural decision tree. This model can then be extended to create a **forest** (i.e., an **ensemble** of decision trees) $\mathcal{F} = \{T_1, \dots, T_k\}$ which delivers a prediction for sample *x* by averaging the output of each tree [92]:

$$\mathbb{P}_{\mathcal{F}}[y|x] = \frac{1}{k} \sum_{h=1}^{k} \mathbb{P}_{T_h}[y|x] \tag{4.23}$$

TreeGrad further improves on this by using **stacking** to form the ensemble as $\hat{y} = \sum_k v_k M_k$, where v_k is the set of real weights $\forall k \in M$, where M is the universe of candidate models [91].

The new architecture also conforms to the **AdaNet Generalization Bounds**, which requires the weights of each layer to be bounded by the $\ell_p - norm$ with $p \geq 1$, and all activation functions between each layer to be **1-Lipschitz activation functions** bound coordinate-wise. The size of each layer is based on the number of nodes *n* with a corresponding number of leaves $\ell = n + 1$. Let $h_0 \in \mathbb{R}^k$ denote the corresponding feature vector $\forall x \in \mathcal{X}$. The decision node layer is defined by **trainable** parameters $\Theta = \{W, b\}$ with weights $W \in \mathbb{R}^{k \times n}$ and biases $b \in \mathbb{R}^n$. Let $\widetilde{W} = [W \oplus -W]$ and $\tilde{b} = [b \oplus -b]$, representing the positive and negative routes of each node (where $\oplus$ is the matrix concatenation operator). Hence, the output of the first layer is $H_1(x) = \widetilde{W}^{\mathrm{T}} x + \tilde{b}$, the linear decision boundary dictating the routing of each node. We also define the activation

function $\phi_2(x) = (log \circ softmax)(x)$ for the decision layer. The (**untrainable**) routing layer is a predetermined binary matrix $Q \in \mathbb{R}^{2n \times (n+1)}$ – hence, the output of the second layer is $H_2(x) = Q^{\mathrm{T}}(\phi_2 \circ H_1)(x),\ H_2(x) \in (-\infty, 0)$. The final (**trainable,** dense) layer is defined by $\pi \in \mathbb{R}^{n+1}$, representing the number of leaves, with an activation function defined as $\phi_3(x) = exp(x)$. Thus, the output of the last layer (the prediction) is $H_3(x) = \pi^{\mathrm{T}}\big(\phi_3 \circ H_2(x)\big)$, and the complete model has the corresponding generalization error bounds related to the AdaNet architecture (see Figure 4.12). More details can be found in [91]. Given the success of this architecture, all other models (see *Chapter 5.2.1*) were deprecated.

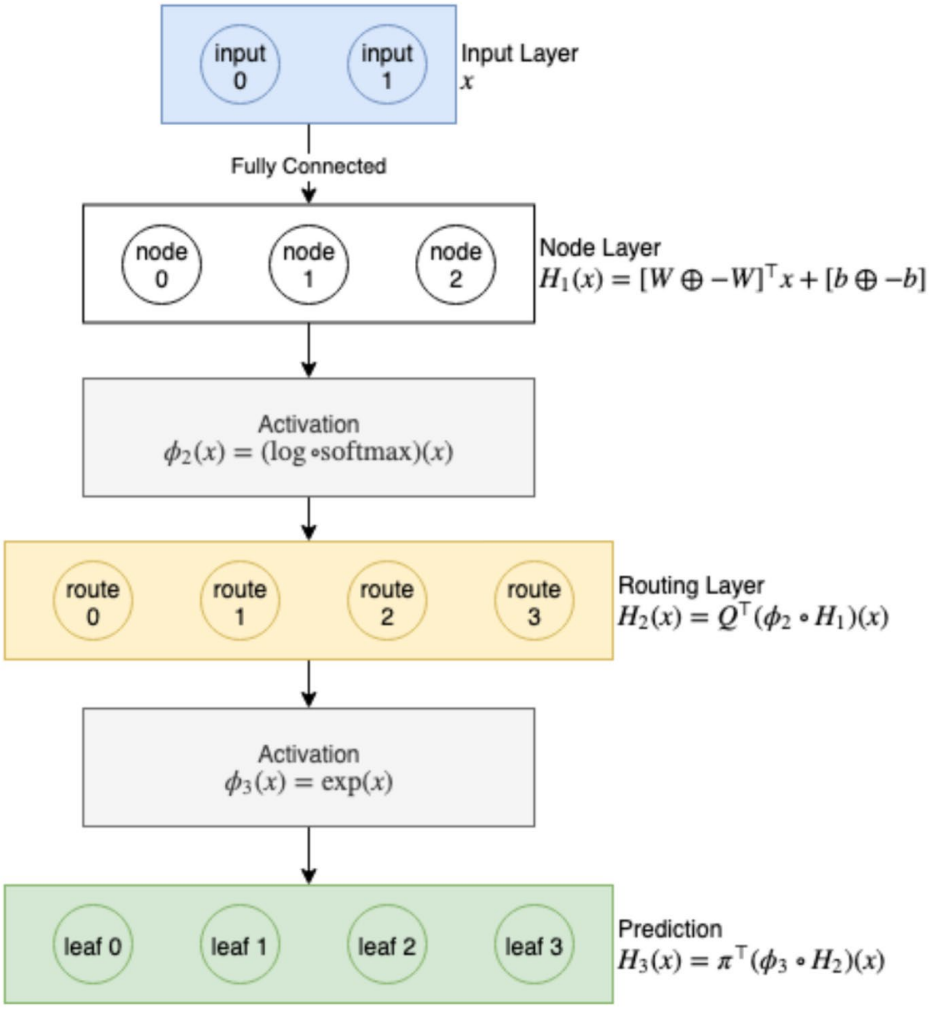


Figure 4.12: TreeGrad architecture modeled as a three-layer Neural network [91]

4.2.2 LSTM-Tree – Phrase Classification

The Phrase Analyzer model was limited to a small number of potential architectures due to the difficulty of performing **Multilabel Classification** – few machine learning algorithms provide good support for this task besides decision trees and k-NN [93]. We initially attempted to use a Dense NN, but the model quickly overfits regardless of the chosen parameters due to the smaller size of the dataset (see *Chapter 5.2.2*). The more feasible choice was to use some form of **RNN** since each audio slice (split by timestamp, see *Chapter 4.1.3*) should have information preserved in the model's memory states to allow recollection of similar phrases – hence, the **Bidirectional LSTM** architecture was chosen. A Bi-LSTM consists of two independent LSTMs (see *Chapter 2.1.2.2*), where one takes input in a forward direction (past to future) and the other in a backward direction (future to past) [94], allowing the networks to have bidirectional information about the sequence at every time step; these are often used in Natural Language Processing (NLP) tasks.

In a standard RNN, the current state is calculated as

$$h_t = f(h_{t-1}, x_t) \tag{4.24}$$

where h_t is the current state, h_{t-1} is the previous state, and x_t is the input state. We can then apply the tanh activation function

$$h_t = tanh(W_{hh}h_{t-1} + W_{xh}x_t) \tag{4.25}$$

where W_{hh} is the weight at the recurrent neuron and W_{xh} is the weight at the input neuron. Thus, we can calculate the output as

$$y_t = W_{hy}h_t \tag{4.26}$$

where y_t is the output and W_{hy} is the weight at the output layer [95]. To modify this into an LSTM, we add a set of activation rules, and each cell now provides two outputs – a new activation and a

new candidate value (which holds possible values to add to the cell state [96]. The new candidate is calculated using

$$c^{N<t>} = tanh(W_c[a^{<t-1>}, x^{<t>}] + b_c) \tag{4.27}$$

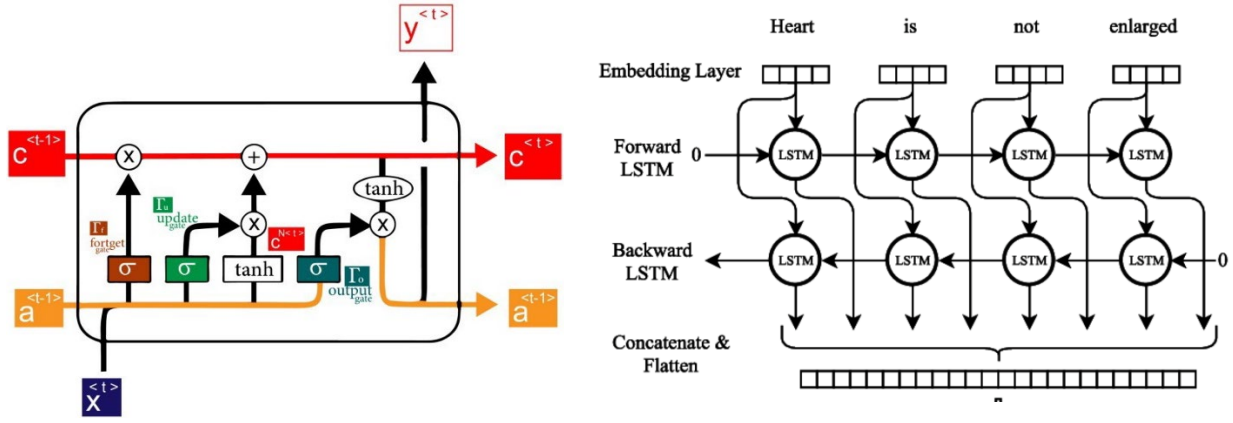
and the memory cells are controlled using three different gates – **Update**, **Forget**, and **Output**:

$$\begin{aligned} \Gamma_u &= \sigma(W_u[a^{<t-1>}, x^{<t>}] + b_u) \\ \Gamma_f &= \sigma\big(W_f[a^{<t-1>}, x^{<t>}] + b_f\big) \\ \Gamma_o &= \sigma(W_o[a^{<t-1>}, x^{<t>}] + b_o) \end{aligned} \tag{4.28}$$

The new candidate and activation can thus be calculated using the gates as

$$(c^{<t>}, a^{<t>}) = \begin{cases} c^{<t>} = \Gamma_u * c^{N<t>} + \Gamma_f * c^{<t-1>} \\ a^{<t>} = \Gamma_o * c^{<t>} \end{cases} \tag{4.29}$$

to create the full **LSTM unit** (see Figure 4.13 (a)).



(a) (b)

Figure 4.13: Complete LSTM unit (a) [95] and a Bi-LSTM system (b) [94]

Hence, we can pair LSTM units in both forward and backward directions (see Figure 4.13 (b)) to calculate the output $\hat{y}$ at time t

$$\hat{y}^{<t>} = g\big(W_y[\vec{a}^{<t>}, \overleftarrow{a}^{<t>}] + b_y\big) \tag{4.30}$$

using **forward activation** $\vec{a}$ and **backward activation** $\overleftarrow{a}$. More details can be found in [95].

The Phrase Analyzer model follows the opposite of the TreeGrad architecture – the Bi-LSTM is trained on the dataset, with the final Dense layer being wrapped in a **Time Distributed** layer – allowing the Dense layer to be activated for every timestamp [97]. The output of this Time Distributed layer is then used to fit a decision tree to the dataset using the Dense output as decision weights, forming a **Dense Bi-LSTM Decision Tree** architecture. For comparison, the prediction system also provides the output of a TreeGrad model and the test decision tree model, though our LSTM Tree architecture outperforms both. Other models (Random Forest, MLP, k-NN, etc.) failed to provide any usable output and were thus deprecated (see *Chapter 5.2.2*). The final model uses **Binary Cross-entropy** (or **log loss**, used to evaluate the performance of the model [103]) as its loss function, calculated as

$$H_p(q) = -\frac{1}{N}\sum_{i=1}^{N} y_i \cdot log\big(\mathbb{P}(y_i)\big) + (1-y_i)\cdot log\big(1-\mathbb{P}(y_i)\big) \tag{4.31}$$

and utilizes the **Adam** optimizer to update weights, a combination of gradient descent algorithms:

$$m_t = \beta_1 m_{t-1} + (1-\beta_1)\left[\frac{\partial L}{\partial w_t}\right], \qquad v_t = \beta_2 v_{t-1} + (1-\beta_2)\left[\frac{\partial L}{\partial w_t}\right]^2$$
$$\widehat{m}_t = \frac{m_t}{1-\beta_1^t}, \qquad \hat{v}_t = \frac{v_t}{1-\beta_2^t} \tag{4.32}$$
$$w_{t+1} = w_t - \widehat{m}_t\left(\frac{\alpha}{\sqrt{\hat{v}_t}+\varepsilon}\right)$$

where:

- w_t = weights at time t
- w_{t+1} = weights at time $t+1$
- α = learning rate/step size (0.001)
- ∂L = derivative of Loss Function
- ∂w_t = derivative of weights at time t
- m_t = aggregate of gradients at time t
- v_t = sum of the square of past gradients $(\sum \nabla L(w_{t-1}))$
- β_1, β_2 = decay rates of average gradients $(\beta_1 = 0.9, \beta_2 = 0.999)$
- ε = small positive constant (10^{-8}) used to prevent division by 0 (as $v_t \to 0$)

and time t can be interpreted as iteration i. More details can be found in [104].

4.3 Constraints

The final prediction system takes any audio file and can be used to predict the form and/or phrase labels of any given piece. While the system may work with any musical genre, these analytical classes are specific to classical music and may not be useful for popular music beyond the Peak-Picking algorithm (see *Chapter 7.2*). The **LSTM-Tree** is also constrained to a smaller set of possible labels; more complex labels such as those used for Sonata and Ritornello forms (and varied repetition labels where a **prime** or **superscript** would be attached) are reduced to the combination of the following sets:

Large Parts = { A, B, C, D, Theme, Variation, CODA, End, Silence }

Phrases = { a, b, c, d, e, f, transition, codetta, section (abbr. "sec") }

Theme Variations = { characteristic, figural, melodic, ornamental, simplifying }

With a larger dataset, the Phrase Analyzer could be modified to predict labels specific to the predicted form, such as those commonly used for Sonatas (Exposition, Primary Theme, …, see Appendix *C.1.1*), but we found these to be viable enough for analysis while preventing overfitting.

CHAPTER 5
EVALUATION

This chapter contains the approaches to evaluating the classification models and the relevant metrics for the system, as well as a comparison of the final models compared to other draft models (including the previous state-of-the-art and machine learning models).

5.1 Evaluation

For the Form Analyzer, the full (augmented) dataset is split into **83.1% training** and **16.9% testing** (or validation) – a method known as **Cross-Validation** [99]. This ratio was found by trial and error using the decision tree test model and the deprecated CNNs. The labels are encoded as both integers (where each unique label has a unique integer mapping from [0 - *n*]) for TreeGrad and **One-Hot Encoding (OHE)** for the DT and CNN test models, and **LSTM-Tree**. OHE removes the potential for ordinal relationships between class labels by converting the integer representation to a binary variable based on if an item belongs to a particular class (see Table 5.1) [98].

Integer Encoding				
Red = 1, Green = 2, Blue = 3				
Item/OHE	**Red**	**Green**	**Blue**	
1	[1,	0,	0	]
2	[0,	1,	0	]
3	[0,	0,	1	]
Purple (Multilabel)	[1,	0,	1	]

Table 5.1: Integer and One-Hot Encoding Sample [98]

After fitting the TreeGrad to the training set, the model performs a prediction on the testing set and returns predicted label set $\hat{y}_{Test}$. This is first evaluated for accuracy using the **Jaccard** score

$$J(A,B) = \frac{|A \cap B|}{|A \cup B|} \tag{5.1}$$

where $A = y_{Test}$, the ground truth labels, and $B = \hat{y}_{Test}$. We also measured the validity of this score using a **Confusion Matrix** (a map of the predicted labels vs. the actual labels by the number of predictions) and the **Precision**, **Recall**, and **F1** scores. These are calculated using the **True Positives (TP)**, **False Positives (FP)**, and **False Negatives (FN)** from the prediction [100]:

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$F1 = 2 * \frac{Precision * Recall}{Precision + Recall} \tag{5.2}$$

5.1.1 Metrics for Phrase Classifier

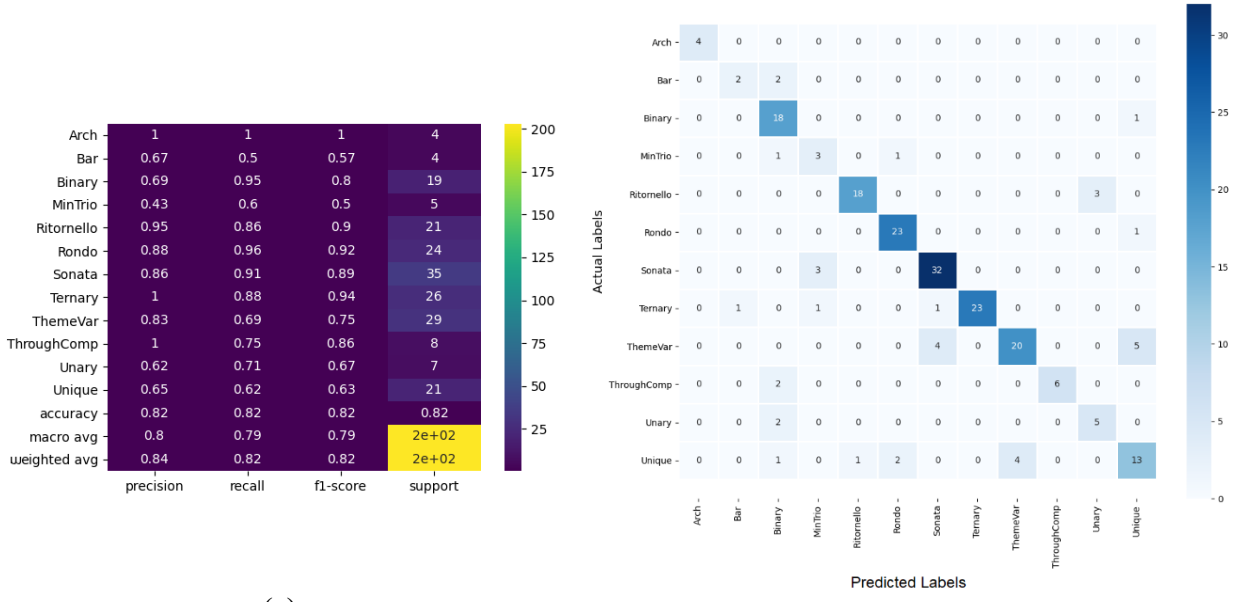
For the Phrase Analyzer, since the model is working with individual timestamps rather than the entire piece of music, as well as the issue of **Multilabel Classification**, Jaccard score is infeasible because the model may predict the correct set of labels but in the correct order (among other issues such as partial correctness). Instead, we used the **Hamming** score, which measures the accuracy for each instance by the proportion of the correctly predicted labels over the total number of actual and predicted labels for that instance, and the average represents the overall accuracy [101]:

$$H(y,\hat{y}) = \frac{1}{n}\sum_{i=1}^{n}\frac{|y_i \cap \hat{y}_i|}{|y_i \cup \hat{y}_i|} \tag{5.3}$$

This is much more similar to how a human analyst could be graded compared for correctness given the objective nature of form analysis, as Jaccard accuracy may also be skewed to the training set.

5.2 Results

The final Form Analyzer model (TreeGrad) achieved a maximum accuracy of 83% (see Figure 5.1). While we also measured the precision and recall, these scores were almost exactly correlated to the validation accuracy (since the task was Multiclass Classification instead of Multilabel), which was found to be true for the other draft models as well. The **hyper-parameters** (a parameter used to control the model's learning process [85]) for TreeGrad can be seen in Table 5.2 which were selected by trial and error. As well, the model does not always guess the same class if it is unable to accurately predict the form of a particular piece, which was an issue in the CNN models – hence, this model could potentially be put into an ensemble to achieve even higher accuracies.



(a) (b)

Figure 5.1: TreeGrad Classification Report (a) and Confusion Matrix (b)

Form Analyzer Parameters						
	# Of Leaves (Per Tree)	**Max Depth**	**Learning Rate**	**# Of Estimators (Trees)**	**Batch Size**	**Refit Splits?**
Value	31	-1 (∞)	0.1	100	32	True

Table 5.2: TreeGrad Hyper-parameters

The Phrase Analyzer system achieved an accuracy of 100% and a hamming score of 100% both on the training data. However, this model is much more difficult to score programmatically, as numerous factors affect the final system (see *Chapter 6.1*):

- The dataset is too small to split well into training and testing proportions,
- For predictions, the model gets the timestamps from the Peak-Picking algorithm (which is also difficult to compare to a human annotated ground truth due to **integrative disagreement**) – if the data was split into a test set, the results would likely be less truthful of the model's performance due to poor generalizing, and,
- The labels are often highly subjective, and some labels are implicit (part A continues until timestamp *n* but is normally only labeled at the first occurrence).

As such, we found that the LSTM-Tree was sufficient to avoid harsh overfitting compared to the other models, likely the result of the boosting from the decision tree (see *Chapter 5.2.2*). This model also greatly outperformed the decision tree test model, which was fit using the same approach. The Peak-Picking algorithm was also clearly comparable to other machine learning approaches, as even pre-labeled data points were nearly identical to those marked by a human analyst (see *Appendix A*). A sample output from both prediction systems can be seen in Figure 5.2.

brahms_opus117_1

Ternary

Guesser:	**LSTMTree**	DcnTree	TreeGrad
0.0	**Silence**	Silence	Silence
0.1	**[A, a]**	[A]	B
21.177	**[A]**	[B]	B
38.87	**[A, sec]**	[CODA, f]	B
57.121	**[A, sec]**	[CODA, f]	B
67.013	**[A]**	[A, sec]	B
96.27	**[B]**	[A, sec]	B
150.187	**[b]**	[A]	B
167.741	**[a]**	[A, sec]	B
186.41	**[B, sec]**	[B]	B
200.899	**[B, c]**	[A]	B
210.512	**[CODA]**	[A]	B
223.608	**[CODA]**	[A, sec]	B
237.958	**[A]**	[A, sec]	B
252.029	**[A]**	[A]	B
269.444	**End**	End	End

(a)

Performing predictions on **anna-magdalena_book_14**
Predicted form: **Unary**
Performing predictions on **brahms_opus117_1**
Predicted form: **Ternary**
Performing predictions on **faure_nocturne_99_no10**
Predicted form: **Sonata**
Performing predictions on **bthvn_pno_concerto_2_19_3**
Predicted form: **Rondo**
Performing predictions on **schubert_D935_2**
Predicted form: **Ternary**
Performing predictions on **schumann_evening_song**
Predicted form: **Rondo**
Performing predictions on **schbrt_strquartet_13-mvt3**
Predicted form: **MinTrio**
Performing predictions on **tchaik_nocturne_19_4**
Predicted form: **Binary**

(b)

Figure 5.2: Full [Form & Phrase] prediction output (a) and Form output for multiple files (b)

5.2.1 Comparison of Form Analyzing Models

We initially tried using numerous variations of CNN models, initially using 2D convolutions with a multi-channel network that would take the Mel SSM in one channel and the 4 SSLMs (MFCCs and Chromas) in the other, as well as adding a third channel for the set of Spectral features (contrast, bandwidth, etc. – see *Appendix B.1*) – however, none of these models ever achieved a validation accuracy higher than 13%, or ~20% when boosted as an ensemble of 5 CNNs, due to the intense **overfitting**. Eventually, we switched to a single-channel CNN with 1D convolutions using the means and variances of each feature (see *Chapter 4.1.1*). This model greatly outperformed the previous 2D CNNs, yielding a test accuracy of 50% even on the augmented data, but this was still insufficient for the final system.

We attempted numerous methods of tuning the CNN hyper-parameters including both randomized and grid searching, manual tuning, and using the **hyperas** Python library to provide a set of fixed hyper-parameter options. However, none of these, including modifying the activation functions and loss function, were sufficient to boost the model to a higher accuracy. We eventually switched to a standard DNN with fully connected (Dense) layers (sometimes called a **Dense Neural Network**), which performed the same as the CNN. Another solution was to use the **Autokeras** Python library, which performs a full neural architecture search using multiple combinations of different layers and hyperparameters to find the best model – nonetheless, the absolute best model from this system only achieved 34% validation accuracy at best. We then attempted combining deep learning algorithms with machine learning, such as **SVM** (**Support Vector Machines**, a supervised learning algorithm that classifies data with the widest possible margin between each class [19], of which the Neural SVM only achieved 23% test accuracy – see *Appendix D*) and finally neural decision tree hybrids.

The first hybrid tree network used was **XBNet**, a NN architecture that uses a method known as **eXtreme Gradient Boosting** [86] (or **XGBoost** – taken from decision trees, where weak learners, i.e., trees with only one split, are sequentially combined so each tree corrects the previous one [87]). While this architecture seemed promising, it only achieved a test accuracy of 24% with its best learning rate being 0.01. This was followed by another similar architecture known as **Deep Jointly Informed Neural Networks (DJINN)**, which maps a collection of decision trees trained on the dataset into a collection of initialized neural networks whose structures are determined by the trees [88]. This model was tuned to the same parameters as the test decision tree (the same number of layers and trees – in this case, 18 and 1, respectively), and obtained a test accuracy of 45% at best. The last attempt was to combine a Dense Neural Network with a decision tree, where the NN would be fit to the dataset, then the output of the last hidden layer would be fit to a decision tree for final classification. While the NN itself only achieved ~26-35% accuracy, the combined model reached a maximum of 53% on the validation set – the best of all other previous models.

Finally, the **TreeGrad** model was introduced, which immediately achieved a test accuracy of 83%, nearly identical to the best decision tree model. With a very small number of trials to tune hyperparameters, this architecture was solidified as the final model, deprecating all prior systems. A comparison of these models can be seen in Table 5.3, and the accuracy, loss, and classifier report from the best 1D CNN model can be seen in Figure 5.3 (compared to TreeGrad, see *Chapter 5.2*).

	2D CNN	**2D CNN Ensemble**	**1D CNN**	**Auto-keras**	**Neural SVM**	**XBNet**	**DJINN**	**Neural DTree**	**TreeGrad**	**Decision Tree**
Best Accuracy	13%	21%	50%	34%	23%	24%	45%	53%	**83%**	84%

Table 5.3: Performance comparison of Form Analyzer models

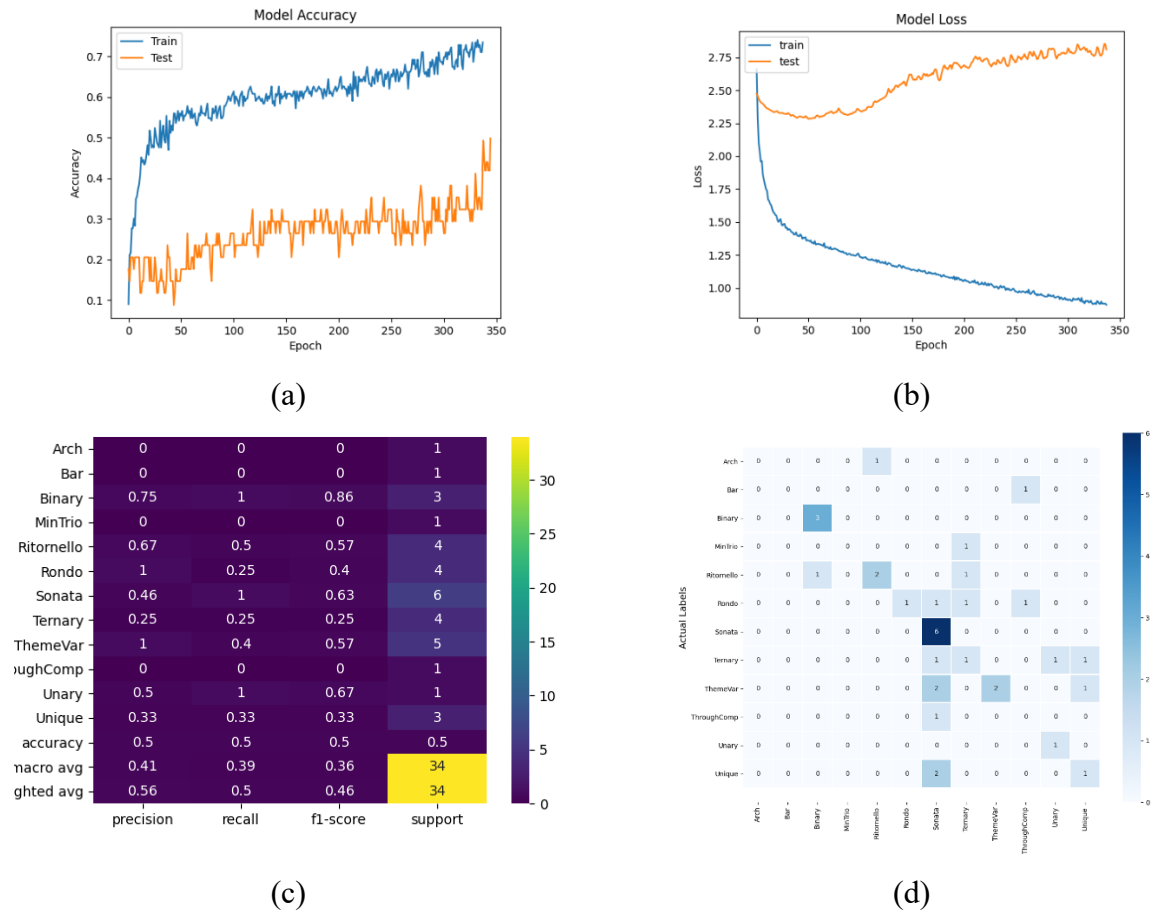


Figure 5.3: Best CNN Accuracy (a), Loss (b), Classification Report (c), and Conf. Matrix (d)

A sample neural decision tree from the TreeGrad model can be seen in Figure 5.4 (i.e., one of the 100 estimators).

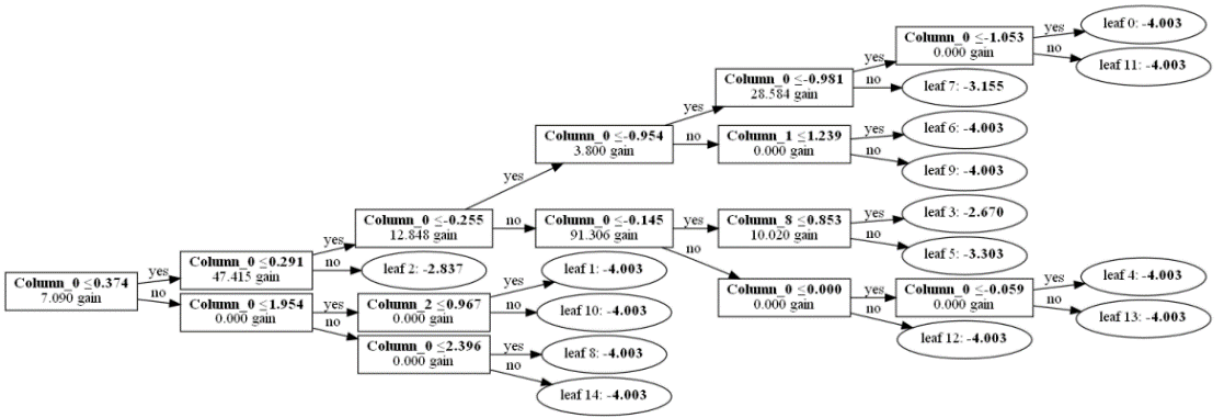


Figure 5.4: Neural Decision Tree at index 0 from the TreeGrad forest

5.2.2 Comparison of Phrase Analyzing Models

Given the complexity of the Phrase classification task, there were very few options for architecture that would provide feasible output. A decision tree was again used as a control model (acting as a proof-of-concept), which is left in to provide comparable output to the **LSTM-Tree** for prediction (see *Chapter 5.2*). The first architecture attempted was a standard Dense NN, which performed extremely poorly due to overfitting (with a top training accuracy of 13%) and trained slowly – thus, the need for an RNN architecture became immediately apparent. We also attempted to use another TreeGrad model (using the same hyper-parameters as the Form Analyzer), but this model both overfits and can only be used for single-label classification; however, this model was also left in for comparison against the LSTM-Tree. We then tried numerous other **Machine Learning** algorithms, including Random Forest, k-NN, Extra Trees, Radius Neighbors, and Multilayer Perceptrons, but all of these failed to provide usable predictions on unseen data (i.e., the models failed to generalize). In particular, the forest models failed due to class imbalance – since phrases are not labeled with an even distribution for each label, the forests tend to overfit – another problem that neural networks are prone to [102].

Through trial and error, the final LSTM-Tree was constructed with 4 LSTM units – 1 unit, equivalent to a Bidirectional RNN, performed decently, but all models with 8 or more (sometimes as few as 5 or more) LSTM units would overfit on the training set. The hyper-parameters of the model can be seen in Table 5.4. A diagram of the model's architecture can be seen in Figure 5.5.

Phrase Analyzer Parameters								
	LSTM Units	**LSTM Dropout**	**Merge Mode**	**Loss**	**Activation**	**Optimizer**	**Batch Size**	**Epochs**
Value	4	0.2	Concat	Binary Cross-entropy	Sigmoid	Adam	1	5

Table 5.4: LSTM-Tree Hyper-parameters

bidirectional_input	InputLayer	input:	[(None, None, 1)]
		output:	[(None, None, 1)]

↓

bidirectional(lstm)	Bidirectional(LSTM)	input:	(None, None, 1)
		output:	(None, None, 8)

↓

time_distributed(dense)	TimeDistributed(Dense)	input:	(None, None, 8)
		output:	(None, None, 22)

↓

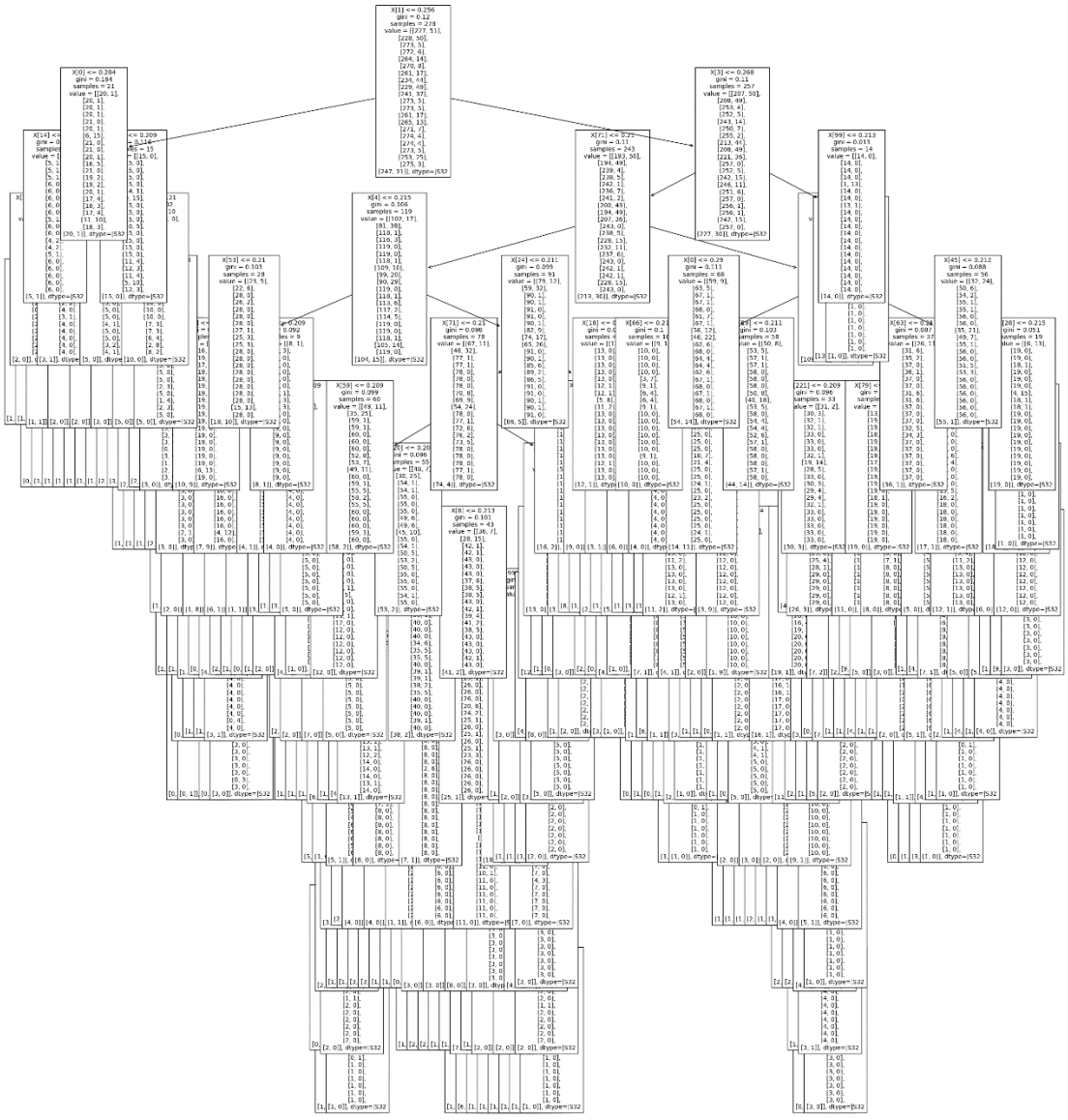


Figure 5.5: LSTM-Tree architecture

CHAPTER 6
DISCUSSION

In this chapter, we review the results of the system evaluation and discuss methods of further improving the models and dataset for increased performance and accuracy and the implications of the resulting system.

6.1 Potential Improvement

Both the Form and Phrase Analyzers present room primarily for indirect improvement through the dataset. However, this would require a massive increase in the size of the dataset (not including augmentation methods), especially for the LSTM-Tree. The LSTM-Tree model may likely be improved using **Curriculum Learning** (a strategy of training a model from easier data to harder data [104]), much like that of a traditional **Form and Analysis** class. An increased dataset would also require new hyper-parameters to match the new data size, and the features may likely need to be decreased using feature elimination methods to prevent overfitting (or underfitting). However, manually expanding the dataset would require many analysts all trained on the same methodology analyzing each piece individually before reaching an agreement for the ground truth labels (especially since two analysts may label varying numbers of events). An **Autoencoder** architecture or **Sequence-to-Sequence (seq2seq)** framework may also be useful in creating a more accurate and/or faster performing system. As well, the peak-picking algorithm could be used to train a more accurate music segmentation network, allowing the entire system to be treated as one large Deep Learning system (see *Chapter 2.2*). The LSTM-Tree also appears to prioritize the large form labels, and often tends to leave out the phrase label or generalizes it as a “section” without a unique letter.

6.1.1 Improving the Dataset

One of the biggest flaws in the dataset currently is the class imbalance – more popular forms (such as Sonata, Rondo, and Ternary) dominate over other smaller forms (Bar, Arch, etc.) and thus can be a primary reason for both machine learning and deep learning models overfitting. We hope that further research expands upon this dataset by contributing to the anthological list provided in *Appendix B*, as very few textbooks and online resources contain any such list of classical music classified by their form much like [2] – though this issue plagues the realm of musicology also. As well, many pieces have yet to be analyzed for phrase and part labeling (including timestamping each event), which would greatly boost the accuracy of the LSTM-Tree model and ideally prevent overfitting in other common classification algorithms. A **human-in-the-loop strategy** may be useful in further developing both the dataset and deep learning models using the current system.

6.2 Implications

In its current state, the Form-NN system is accurate enough to be implemented into the backend of a higher-level system (see *Chapter 7.2*), such as a grading tool for human-analyzed scores or a musical practice tool. The peak-picking algorithm could be used in creating analysis assignments for classical music, using the form classification and phrase labels as a baseline with room for simple corrections as needed. For rehearsal, the Phrase Analyzer could be used to narrow down the set of unique phrases in a piece of solo repertoire, allowing the musician to practice non-linearly – this greatly cuts down the amount of practice time that would be expended from reading the music linearly, as the repeated themes have already been learned. This technique is frequently applied by conductors during score study to speed up the rehearsal process in preparation for an upcoming concert but is also applicable for solo musicians as well. For a musicologist, this system could be used to discover when and why a composer may choose a particular form over another.

CHAPTER 7
CONCLUSION

This chapter discusses the achievements of the thesis relative to its goals and objectives and provides additional research questions for future work to expand upon.

7.1 Conclusion

In this thesis, we have devised a system for the task of automatic musical form recognition and analysis using deep learning methods. Using hybrid **Neural Network-Decision Tree** models, we were able to design a framework that significantly outperformed standard neural architectures and greatly reduced the potential for overfitting. This system completely analyses a piece of classical music, including locating the points of musical events, labeling them according to their formal part and/or phrase classification, and classifying the piece by its large form structure. We also presented a new dataset that seeks to correct the errors presented by previous commonly used databases (see *Chapter 1.1*) which lack standardized analytical conventions and measurement of annotation set accuracy, while also providing both pre-computed spectral data to speed up the training process of future models and the form classification of each piece of music. Hence, we were able to achieve the initial goals and objectives hypothesized in *Chapter 1.2*, and the final system is in a usable state for individual use or implementation into a more complex piece of software.

7.2 Future Work

The research presented in this thesis can be extended to several different areas, such as Schenkerian and/or contemporary music analysis, audio classification, video segmentation analysis, NLP, ethnomusicology, anthological data mining, and audio thumbnail generation (e.g., for a web-based music store). Although the Form-NN system is specific to classical music analysis, it could be extended to allow for the classification of additional forms including those found in popular music and classical form derivatives such as Rondo-Sonata/Sonata-Rondo, Da Capo Aria, Concerto Sonata, etc. **Optical Music Recognition (OMR)** is another challenging task lacking substantial research – the methods we proposed could be potentially extended to perform visual musical analysis and perform the segmentation and classification of the sheet music itself, much like that of a human analyst (see Figure 2.7). The system could also be extended for use in the field of **Forensic Musicology**, as the analysis provided by the system may be useful in comparing multiple pieces of music for potentially similar or exact replications of musical phrases.

REFERENCES

[1] Jacopo de Berardinis, Michalis Vamvakaris, Angelo Cangelosi, and Eduardo Coutinho. 2020. Unveiling the hierarchical structure of music by multi-resolution community detection. *Transactions of the International Society for Music Information Retrieval* 3, 1: 82–97. https://doi.org/10.5334/tismir.41

[2] Douglass Marshall Green. 1979. *Form in tonal music: An introduction to analysis*. Cengage Learning.

[3] Thomas Grill and Jan Schlüter. 2015. *Structural segmentation with convolutional neural networks MIREX submission*.

[4] Dominik Hörnel and Wolfram Menzel. 1998. Learning musical structure and style with neural networks. *Computer Music Journal* 22, 4: 44–62. https://doi.org/10.2307/3680893

[5] Tim O'Brien. 2016. Musical Structure Segmentation with Convolutional Neural Networks. *17th International Society for Music Information Retrieval Conference*.

[6] Pedro Ponce de León and José Iñesta. 2007. Pattern recognition approach for music style identification using shallow statistical descriptors. *IEEE Transactions on Systems, Man and Cybernetics, Part C (Applications and Reviews)* 37, 2: 248–257. https://doi.org/10.1109/tsmcc.2006.876045

[7] Karen Ullrich, Jan Schlüter, and Thomas Grill. 2014. Boundary Detection in Music Structure Analysis using Convolutional Neural Networks. *15th International Society for Music Information Retrieval Conference*.

[8] Ali Bou Nassif, Ismail Shahin, Imtinan Attili, Mohammad Azzeh, and Khaled Shaalan. 2019. Speech Recognition Using Deep Neural Networks: A Systematic Review. *IEEE Access* 7: 19143–19165. http://dx.doi.org/10.1109/ACCESS.2019.2896880

[9] Jordan Smith. 2019. "SALAMI Annotator's Guide." *GitHub*. Retrieved December 21, 2021 from https://github.com/DDMAL/salami-data-public/blob/master/SALAMI%20Annotator%20Guide.pdf.

[10] McGill DDMAL. 2009. A Structural Analysis of Large Amounts of Music Information (SALAMI). *DDMAL*. Retrieved from https://ddmal.music.mcgill.ca/research/SALAMI/.

[11] Leon Stein. 1979. *Structure & Style: The Study and Analysis of Musical Forms, Expanded Edition*. Summy-Birchard Music.

[12] Vansh Sethi. 2019. Types of Neural Networks (and what each one does!) Explained. *Towards Data Science*. Retrieved December 22, 2021 from https://towardsdatascience.com/types-of-neural-network-and-what-each-one-does-explained-d9b4c0ed63a1

[13] DeepAI. 2019. Feed Forward Neural Network. *DeepAI*. Retrieved December 22, 2021 from https://deepai.org/machine-learning-glossary-and-terms/feed-forward-neural-network

[14] Naveen Joshi. 2019. 3 types of neural networks that AI uses. *Allerin*. Retrieved from https://www.allerin.com/blog/3-types-of-neural-networks-that-ai-uses

[15] Bossy Mostafa, Noha El-Attar, Samy Abd-Elhafeez, and Wael Awad. 2020. Machine and Deep Learning Approaches in Genome: Review Article. Alfarama Journal of Basic & Applied Sciences 0, 0 (August 2020), 0–0. DOI:https://doi.org/10.21608/ajbas.2020.34160.1023

[16] Quantum Neural Network — PennyLane. *PennyLane*. Retrieved December 23, 2021 from https://pennylane.ai/qml/glossary/quantum_neural_network.html

[17] Mike Wang. 2021. Deep Q-Learning Tutorial: minDQN. *Towards Data Science*. Retrieved December 23, 2021 from https://towardsdatascience.com/deep-q-learning-tutorial-mindqn-2a4c855abffc

[18] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. 2013. Playing Atari with Deep Reinforcement Learning. arXiv:312.5602v1

[19] Daniel Szelogowski. 2020. Generative Deep Learning for Virtuosic Classical Music: Generative Adversarial Networks as Renowned Composers. arXiv:2101.00169

[20] DeepAI. 2019. Recurrent Neural Network. *DeepAI*. Retrieved December 23, 2021 from https://deepai.org/machine-learning-glossary-and-terms/recurrent-neural-network

[21] Christopher Olah. 2015. Understanding LSTM Networks. *Colah's Blog*. Retrieved from https://colah.github.io/posts/2015-08-Understanding-LSTMs/

[22] Daniel Szelogowski. 2021. Emotion Recognition of the Singing Voice: Toward a Real-Time Analysis Tool for Singers. arXiv:2105.00173

[23] Aravindpai Pai. 2020. ANN vs CNN vs RNN. *Analytics Vidhya*. Retrieved from https://www.analyticsvidhya.com/blog/2020/02/cnn-vs-rnn-vs-mlp-analyzing-3-types-of-neural-networks-in-deep-learning/

[24] DeepAI. 2019. Machine Learning. *DeepAI*. Retrieved December 23, 2021 from https://deepai.org/machine-learning-glossary-and-terms/machine-learning

[25] IBM Cloud Education. 2020. What is Machine Learning? *IBM*. Retrieved from https://www.ibm.com/cloud/learn/machine-learning

[26] Kristin Burnham. 2020. Artificial Intelligence vs. Machine Learning: What's the Difference? *Northeastern University Graduate Programs*. Retrieved December 23, 2021 from https://www.northeastern.edu/graduate/blog/artificial-intelligence-vs-machine-learning-whats-the-difference/

[27] Sunil Ray. 2017. Commonly Used Machine Learning Algorithms. *Analytics Vidhya*. Retrieved December 23, 2021 from https://www.analyticsvidhya.com/blog/2017/09/common-machine-learning-algorithms/

[28] DeepAI. 2019. Deep Learning. *DeepAI*. Retrieved December 23, 2021 from https://deepai.org/machine-learning-glossary-and-terms/deep-learning

[29] DeepAI. 2019. Neural Network. *DeepAI*. Retrieved December 23, 2021 from https://deepai.org/machine-learning-glossary-and-terms/neural-network

[30] Baeldung. 2020. Epoch in Neural Networks. Baeldung on Computer Science. Retrieved from https://www.baeldung.com/cs/epoch-neural-networks

[31] MIR. Why_mir. Music Information Retrieval. Retrieved from https://musicinformationretrieval.com/why_mir.html

[32] M. Sai Chaitanya and Soubhik Chakraborty. 2021. *Musical information retrieval. Signal Analysis and Feature Extraction using Python*. GRIN Verlag.

[33] Timothy C. Justus and Jamshed J. Bharucha. 2002. Music Perception and Cognition. In *Stevens' Handbook of Experimental Psychology, Third Edition, Volume 1: Sensation and Perception*. John Wiley & Sons, Inc., Hoboken, NJ, USA, 453-492. DOI:http://dx.doi.org/10.1002/0471214426.pas0111

[34] Sumit Saha. 2018. A Comprehensive Guide to Convolutional Neural Networks — the ELI5 way. *Towards Data Science*. Retrieved December 24, 2021 from https://towardsdatascience.com/a-comprehensive-guide-to-convolutional-neural-networks-the-eli5-way-3bd2b1164a53

[35] The University of Kansas. 2014. Form Analysis. *KU School of Music*. Retrieved from https://music.ku.edu/form-analysis

[36] Stony Brook. MUS Degrees & Requirements. *Stony Brook Undergraduate Bulletin*. Retrieved from https://www.stonybrook.edu/sb/bulletin/current/academicprograms/mus/degreesandrequirements.php

[37] Sayantini. 2019. What is Fuzzy Logic in AI and What are its Applications? *Edureka*. Retrieved December 25, 2021 from https://www.edureka.co/blog/fuzzy-logic-ai/

[38] Benjamin Whitcomb. 2021. *Form Terms*.

[39] Jeremy Burns. 2018. 53-Form and Analysis Pt.2: Small Forms. *Music Student 101*. Retrieved December 26, 2021 from https://musicstudent101.com/53-form-and-analysis-pt.2-small-forms.html

[40] Bryan Townsend. 2013. Phrase, Motif and Theme. *The Music Salon*. Retrieved December 26, 2021 from https://themusicsalon.blogspot.com/2013/01/phrase-motif-and-theme.html

[41] Stefan Kostka, Dorothy Payne, and Byron Almén. 2017. Tonal Harmony (8th ed.). McGraw-Hill Education.

[42] Ritesh Singh. 2020. Understanding neural networks through visualization. *Druva*. Retrieved December 26, 2021 from https://www.druva.com/blog/understanding-neural-networks-through-visualization/

[43] Olalekan Ogunmolu, Xuejun Gu, Steve Jiang, and Nicholas Gans. 2016. Nonlinear Systems Identification Using Deep Dynamic Neural Networks. (October 2016).

[44] Ashutosh Tripathi. 2021. What is the main difference between RNN and LSTM. *Data Science Duniya*. Retrieved December 27, 2021 from https://ashutoshtripathi.com/2021/07/02/what-is-the-main-difference-between-rnn-and-lstm-nlp-rnn-vs-lstm/

[45] Michael Tilmouth. 2001. *Strophic*. Oxford University Press. Retrieved December 27, 2021 from http://dx.doi.org/10.1093/gmo/9781561592630.article.26981

[46] Samuel Chase. 2021. What Is Through Composed Form in Music? *Hello Music Theory: Learn Music Theory Online*. Retrieved December 28, 2021 from https://hellomusictheory.com/learn/through-composed-form/

[47] Robert Hutchinson. 2021. Sonata Form. *Music Theory for the 21st-Century Classroom*. Retrieved December 29, 2021 from https://musictheory.pugetsound.edu/mt21c/SonataIntroduction.html

[48] Steven Ashby. Sonata-Allegro Form. *Music Appreciation (MHIS 243)*. Retrieved December 29, 2021 from https://rampages.us/mhis243/lectures/lesson-8/sonata-allegro-form/

[49] Anuj Sable. 2021. Introduction to Audio Analysis and Processing. *Paperspace Blog*. Retrieved from https://blog.paperspace.com/introduction-to-audio-analysis-and-synthesis/

[50] Meinard Müller. 2021. *Fundamentals of Music Processing: Using Python and Jupyter Notebooks*. Springer Nature.

[51] Meinard Müller. 2015. C2_STFT-Basic. *International Audio Laboratories Erlangen*. Retrieved December 29, 2021 from https://www.audiolabs-erlangen.de/resources/MIR/FMP/C2/C2_STFT-Basic.html

[52] Slim Essid. 2018. Audio Data Analysis. *Telecom ParisTech*. Retrieved from https://perso.telecom-paristech.fr/essid/ces_ds/audio-analysis-lecture_2018.pdf

[53] Desh Raj. 2019. A note on MFCCs and delta features. *Desh2608 Github*. Retrieved December 30, 2021 from https://desh2608.github.io/2019-07-26-delta-feats/

[54] Dishashree Gupta. 2020. Activation Functions. *Analytics Vidhya*. Retrieved January 22, 2022 from https://www.analyticsvidhya.com/blog/2020/01/fundamentals-deep-learning-activation-functions-when-to-use-them/

[55] 2021. What Is a Decision Tree? *Master's in Data Science*. Retrieved April 1, 2022 from https://www.mastersindatascience.org/learning/introduction-to-machine-learning-algorithms/decision-tree/

[56] Sruthi E R. 2021. Random Forest. *Analytics Vidhya*. Retrieved April 1, 2022 from https://www.analyticsvidhya.com/blog/2021/06/understanding-random-forest/

[57] 2018. What is TensorFlow? *Databricks*. Retrieved April 2, 2022 from https://databricks.com/glossary/what-is-tensorflow

[58] Codecademy. What is Scikit-Learn? *Codecademy*. Retrieved April 2, 2022 from https://www.codecademy.com/article/scikit-learn

[59] Introduction to NumPy. *W3Schools*. Retrieved April 2, 2022 from https://www.w3schools.com/python/numpy/numpy_intro.asp

[60] Mode Resources. 2016. Pandas. *Mode Resources*. Retrieved April 2, 2022 from https://mode.com/python-tutorial/libraries/pandas/

[61] Librosa. *Librosa — Librosa 0.9.1 Documentation*. Librosa. Retrieved April 2, 2022 from https://librosa.org/doc/latest/index.html

[62] James Robert. jiaaro/pydub @ GitHub. *Pydub*. Retrieved April 2, 2022 from http://pydub.com/

[63] 2017. Understanding Python Pickling with example. *GeeksforGeeks*. Retrieved April 2, 2022 from https://www.geeksforgeeks.org/understanding-python-pickling-example/

[64] Scikit-Learn. 1.10. Multiclass and multilabel algorithms — scikit-learn 0.15-git documentation. *Scikit-Learn*. Retrieved April 2, 2022 from https://scikit-learn.org/0.15/modules/multiclass.html

[65] Saugata Paul. 2019. A detailed case study on Multi-Label Classification with Machine Learning algorithms and predicting movie tags based on plot summaries! *Medium*. Retrieved April 2, 2022 from https://medium.com/@saugata.paul1010/a-detailed-case-study-on-multi-label-classification-with-machine-learning-algorithms-and-72031742c9aa

[66] Daniel Szelogowski. 2021. GitHub - DanielAtHome19/Form-NN: Master thesis project. *GitHub*. Retrieved April 2, 2022 from https://github.com/danielathome19/Form-NN

[67] Lorraine Li. 2019. Principal Component Analysis for Dimensionality Reduction. *Towards Data Science*. Retrieved April 2, 2022 from https://towardsdatascience.com/principal-component-analysis-for-dimensionality-reduction-115a3d157bad

[68] Stefan Hrouda-Rasmussen. 2021. The Curse of Dimensionality. *Towards Data Science*. Retrieved April 2, 2022 from https://towardsdatascience.com/the-curse-of-dimensionality-5673118fe6d2

[69] Jason Brownlee. 2019. A Gentle Introduction to Imbalanced Classification. *Machine Learning Mastery*. Retrieved April 2, 2022 from https://machinelearningmastery.com/what-is-imbalanced-classification/

[70] Resampy. 2016. Core functionality — resampy 0.1.5 documentation. *Resampy Documentation*. Retrieved April 2, 2022 from https://resampy.readthedocs.io/en/0.1.5/api.html

[71] Brian McFee. 2018. resampy/interpn.py at master · bmcfee/resampy. *GitHub*. Retrieved April 2, 2022 from https://github.com/bmcfee/resampy/blob/master/resampy/interpn.py

[72] Weisstein. Gamma Function -- from Wolfram MathWorld. *Wolfram MathWorld*. Retrieved April 2, 2022 from https://mathworld.wolfram.com/GammaFunction.html

[73] Théo Royer. 2019. Pitch-shifting algorithm design and applications in music. Thesis. KTH Royal Institute of Technology - School of Electrical Engineering and Computer Science. Retrieved from http://kth.diva-portal.org/smash/get/diva2:1381398/FULLTEXT01.pdf

[74] Meinard Müller. 2015. C2_STFT-Inverse. *International Audio Laboratories Erlangen*. Retrieved April 2, 2022 from https://www.audiolabs-erlangen.de/resources/MIR/FMP/C2/C2_STFT-Inverse.html

[75] MATLAB. Inverse short-time Fourier transform. MATLAB istft. Retrieved April 2, 2022 from https://www.mathworks.com/help/signal/ref/istft.html

[76] William Sethares. 2007. phase vocoder in Matlab. *University of Wisconsin*. Retrieved April 2, 2022 from https://sethares.engr.wisc.edu/vocoders/phasevocoder.html

[77] Chris Tralie. 2014. Musical Pitches and Chroma Features. *ECE 381 Data Expeditions Lab 1*. Retrieved April 3, 2022 from https://www.ctralie.com/Teaching/ECE381_DataExpeditions_Lab1/

[78] Carlos Hernandez-Olivan, Jose R. Beltran, and David Diaz-Guerra. 2020. Music Boundary Detection using Convolutional Neural Networks: A comparative analysis of combined input features. arXiv:2008.07527

[79] Meinard Müller. 2015. C2_STFT-Inverse. *International Audio Laboratories Erlangen*. Retrieved April 2, 2022 from https://www.audiolabs-erlangen.de/resources/MIR/FMP/C2/C2_STFT-Inverse.html

[80] Joan Serrà, Meinard Müller, Peter Grosche, and Josep Arcos. 2014. Unsupervised Music Structure Annotation by Time Series Structure Features and Segment Similarity. IEEE Transactions on Multimedia 16, 5 (August 2014), 1229–1240. DOI:https://doi.org/10.1109/TMM.2014.2310701

[81] Librosa. 2021. librosa.segment — librosa 0.8.1 documentation. *Librosa Documentation*. Retrieved April 3, 2022 from https://librosa.org/doc/0.8.1/_modules/librosa/segment.html

[82] Christopher De Sa. 2018. Lecture 2: k-nearest neighbors / Curse of Dimensionality. *CS 4/5780 Introduction to Machine Learning*. Retrieved April 3, 2022 from https://www.cs.cornell.edu/courses/cs4780/2018fa/lectures/lecturenote02_kNN.html

[83] Scikit-Learn. sklearn.feature_selection.SelectKBest. *scikit-learn*. Retrieved April 3, 2022 from https://scikit-learn.org/stable/modules/generated/sklearn.feature_selection.SelectKBest.html#sklearn.feature_selection.SelectKBest

[84] Scikit-Learn. 1.13. Feature selection. scikit-learn. Retrieved April 3, 2022 from https://scikit-learn.org/stable/modules/feature_selection.html

[85] Jason Brownlee. 2017. What is the Difference Between a Parameter and a Hyperparameter? Machine Learning Mastery. Retrieved April 4, 2022 from https://machinelearningmastery.com/difference-between-a-parameter-and-a-hyperparameter/

[86] Tushar Sarkar. 2021. XBNet : An Extremely Boosted Neural Network. arXiv.org. arXiv:2106.05239

[87] Cheshta Dhingra. 2020. A Visual Guide to Gradient Boosted Trees (XGBoost). Towards Data Science. Retrieved April 4, 2022 from https://towardsdatascience.com/a-visual-guide-to-gradient-boosted-trees-8d9ed578b33

[88] K. D. Humbird, J. L. Peterson, and R. G. McClarren. 2017. Deep neural network initialization with decision trees. arXiv.org. arXiv:1707.00784

[89] Prashant Gupta. 2017. Regularization in Machine Learning. *Towards Data Science*. Retrieved April 4, 2022 from https://towardsdatascience.com/regularization-in-machine-learning-76441ddcf99a

[90] Amar Budhiraja. 2018. Dropout in (Deep) Machine learning - Amar Budhiraja. *Medium*. Retrieved April 4, 2022 from https://medium.com/@amarbudhiraja/https-medium-com-amarbudhiraja-learning-less-to-learn-better-dropout-in-deep-machine-learning-74334da4bfc5

[91] Chapman Siu. 2019. Transferring Tree Ensembles to Neural Networks. In *Neural Information Processing*. Springer International Publishing, Cham, 471–480. Retrieved April 5, 2022 from http://dx.doi.org/10.1007/978-3-030-36711-4_39

[92] Peter Kontschieder, Madalina Fiterau, Antonio Criminisi, and Samuel Rota Bulo. 2015. Deep Neural Decision Forests. In *2015 IEEE International Conference on Computer Vision (ICCV),* IEEE. Retrieved April 5, 2022 from http://dx.doi.org/10.1109/iccv.2015.172

[93] Scikit-Learn. 1.12. Multiclass and multioutput algorithms. *scikit-learn*. Retrieved April 5, 2022 from https://scikit-learn.org/stable/modules/multiclass.html

[94] Papers With Code. Papers with Code - BiLSTM Explained. *Papers With Code*. Retrieved April 5, 2022 from https://paperswithcode.com/method/bilstm

[95] Raghav Aggarwal. 2019. Bi-LSTM - Raghav Aggarwal. *Medium*. Retrieved April 5, 2022 from https://medium.com/@raghavaggarwal0089/bi-lstm-bc3d68da8bd0

[96] Michael Phi. 2020. Illustrated Guide to LSTM's and GRU's: A step by step explanation. *Towards Data Science*. Retrieved April 5, 2022 from https://towardsdatascience.com/illustrated-guide-to-lstms-and-gru-s-a-step-by-step-explanation-44e9eb85bf21

[97] Keras Team. Keras documentation: TimeDistributed layer. *Keras*. Retrieved April 5, 2022 from https://keras.io/api/layers/recurrent_layers/time_distributed/

[98] Jason Brownlee. 2017. Why One-Hot Encode Data in Machine Learning? *Machine Learning Mastery*. Retrieved April 5, 2022 from https://machinelearningmastery.com/why-one-hot-encode-data-in-machine-learning/

[99] Payam Refaeilzadeh, Lei Tang, and Huan Liu. Cross-Validation. SpringerLink. Retrieved April 5, 2022 from https://link.springer.com/referenceworkentry/10.1007/978-0-387-39940-9_565

[100] Koo Ping Shung. 2020. Accuracy, Precision, Recall or F1? *Towards Data Science*. Retrieved April 5, 2022 from https://towardsdatascience.com/accuracy-precision-recall-or-f1-331fb37c5cb9

[101] MMA. 2020. Metrics for Multilabel Classification. *Mustafa Murat ARAT*. Retrieved April 5, 2022 from https://mmuratarat.github.io/2020-01-25/multilabel_classification_metrics

[102] Johnson and Khoshgoftaar. 2019. Survey on deep learning with class imbalance. *Journal of Big Data* 6, 1 (March 2019), 1–54. DOI:https://doi.org/10.1186/s40537-019-0192-5

[103] Daniel Godoy. 2019. Understanding binary cross-entropy / log loss: a visual explanation. *Towards Data Science*. Retrieved April 5, from https://towardsdatascience.com/understanding-binary-cross-entropy-log-loss-a-visual-explanation-a3ac6025181a

[104] GeeksforGeeks. 2020. Intuition of Adam Optimizer. GeeksforGeeks. Retrieved April 6, 2022 from https://www.geeksforgeeks.org/intuition-of-adam-optimizer/

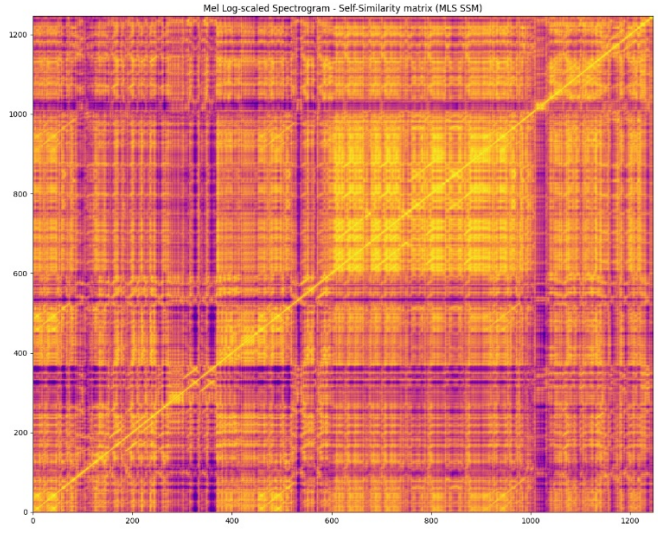
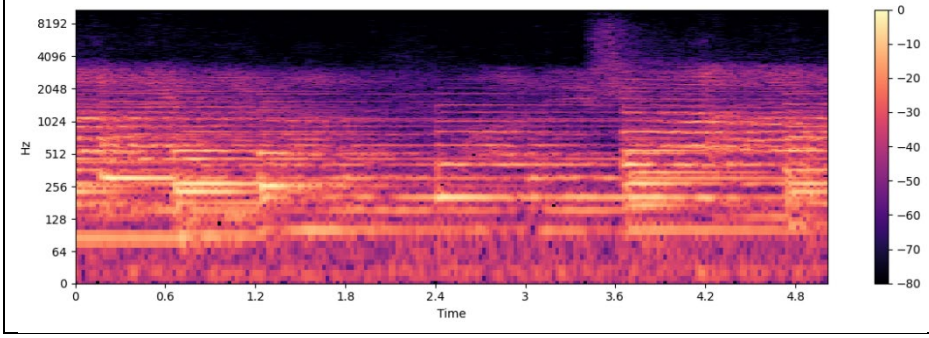
[105] Xin Wang, Yudong Chen, and Wenwu Zhu. 2020. A Survey on Curriculum Learning. arXiv:2010.13166

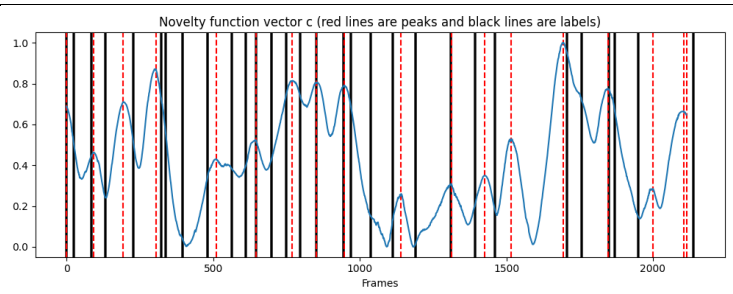
APPENDICES

APPENDIX A
EXAMPLE TRAINING DATA

This appendix contains examples of the data used in training the model architecture (as well as novelty functions and intermediate data used for calculating model input) plotted as a graph, including visual, audio, and text input. All data samples were retrieved from the **Rondo** "Sonata No. 14, K. 457 – Mvt. 3" by Wolfgang Amadeus Mozart (see *Appendix B*).

Model Input Data:

	SSM of Mel Log-scaled Spectrogram
	Log Mel Spectrogram



Novelty Function Prediction

Red lines are peaks

Black lines are ground truth labels

```
Rondo
0.000           Silence
3.370           A, a
11.670          a'
18.350          b
31.570          b
44.850          bridge
47.050          B, c
55.230          c'
66.890          d
78.390          d
85.120          codetta
90.000          retransition
97.400          A', a
104.320         a'
111.040         b
118.550         transition
131.670         bridge
135.000         C, e
144.410         e'
155.080         B', c''
165.660         f
182.600         d'
194.230         transition2
203.560         A'', a''
237.750         b
244.940         transition
257.610         bridge
260.550         C', e''
271.670         CODA
298.000         End
```

Ground truth labels from the new proposed dataset in replicated SALAMI annotation format

```
Event: 0:00:00          Ground Truth: 0:00:00          Difference:  0.000000
Event: 0:00:13.235374   Ground Truth: 0:00:01.150000   Difference: -12.085374
Event: 0:00:28.560544   Ground Truth: 0:00:23.995000   Difference: -4.565544
Event: 0:00:40.124082   Ground Truth: 0:00:29.945000   Difference: -10.179082
Event: 0:01:00.186122   Ground Truth: 0:00:37.545000   Difference: -22.641122
Event: 0:01:18.715646   Ground Truth: 0:00:48.645000   Difference: -30.070646
Event: 0:01:42.678639   Ground Truth: 0:00:56.495000   Difference: -46.183639
Event: 0:01:54.102857   Ground Truth: 0:01:40.195000   Difference: -13.907857
Event: 0:02:14.443537   Ground Truth: 0:01:59.161000   Difference: -15.282537
Event: 0:02:27.678912   Ground Truth: 0:02:22.011000   Difference: -5.667912
Event: 0:02:44.397279   Ground Truth: 0:02:52.811000   Difference:  8.413721
Event: 0:02:56.100136   Ground Truth: 0:03:19.611000   Difference:  23.510864
Event: 0:03:09.196190   Ground Truth: 0:03:32.411000   Difference:  23.214810
Event: 0:03:22.292245   Ground Truth: 0:03:51.761000   Difference:  29.468755
Event: 0:03:36.642177   Ground Truth: 0:04:10.761000   Difference:  34.118823
Event: 0:03:59.072653   Ground Truth: 0:04:24.211000   Difference:  25.138347
Event: 0:04:15.233741   Ground Truth: 0:04:30.761000   Difference:  15.527259
Event: 0:04:31.255510   Ground Truth: 0:04:38.761000   Difference:  7.505490
Event: 0:04:41.983129   Ground Truth: 0:04:47.761000   Difference:  5.777871
Event: 0:05:01.209252   Ground Truth: 0:04:59.911000   Difference: -1.298252
Event: 0:05:19.878095   Ground Truth: 0:05:06.511000   Difference: -13.367095
Event: 0:05:34.228027   Ground Truth: 0:05:31.761000   Difference: -2.467027
Event: 0:05:48.438639   Ground Truth: 0:05:43.861000   Difference: -4.577639
Event: 0:06:08.918639   Ground Truth: 0:06:08.001000   Difference: -0.917639
Average (absolute) time difference: ±14.828637755102045
```

Demonstration of peak-picking algorithm compared to ground truth annotation

APPENDIX B
COMPOSITIONS USED FOR TRAINING

This appendix contains an alphabetical listing of each piece of music used as training data for the model architecture, grouped by their respective musical form, and the list of extracted signal features.

Unary/Strophic:

Bach, Johann Sebastian:
- Well-Tempered Clavier, Vol. 1, Prelude No. 2

Chopin, Frédéric:
- Nocturne in D Flat Major, Op. 27, No. 2
- Prelude, Op. 28, No. 1
- Prelude, Op. 28, No. 2
- Prelude, Op. 28, No. 4
- Prelude, Op. 28, No. 9
- Prelude, Op. 28, No. 23

Through-Composed:

Bach, Johann Sebastian:
- Chromatische Fantasie und Fuge, BWV 903

Brahms, Johannes:
- Ein Deutches Requiem – Mvt. 6

Fauré, Gabriel:
- Requiem, Op. 48 – Mvt. 1 (“Introit and Kyrie”)
- Requiem, Op. 48 – Mvt. 3 (“Sanctus”)
- Requiem, Op. 48 – Mvt. 5 (“Agnus Dei”)
- Requiem, Op. 48 – Mvt. 6 (“In Paradisum”)

Schubert, Franz:
- Die Schöne Müllerin, Op. 25, No. 3 ("Halt!"), D. 795
- Erlkönig

Binary:

Bach, Johann Sebastian:
- Minuet in G Major, BWV Anh. 114
- Partita No. 1 in B Minor, BWV 1002 – Mvt. 5
- Suite in A Minor, BWV 818 – Mvt. 2
- Well-Tempered Clavier, Vol. 1, Prelude No. 23
- Well-Tempered Clavier, Vol. 1, Prelude No. 24

Chopin, Frédéric:
- Prelude, Op. 28, No. 7

Handel, George Frideric:
- Passepied in A Major, HWV 560
- Recorder Sonata in A Minor, Op. 1, No. 4 – Mvt. 2

Haydn, Franz Joseph:
- Piano Sonata No. 50, Hob.XVI:37 - Mvt. 2
- Quartet, Op. 76, No. 5 – Mvt. 3
- Symphony No. 101 (“Clock”) – Mvt. 3 (“Minuet”)

Mozart, Wolfgang Amadeus:
- Magic Flute, No. 10
- Magic Flute, No. 11
- Magic Flute, No. 17

Scarlatti, Domenico:
- Sonata, K. 158
- Sonata, K. 159
- Sonata in A Major, K. 322

Schubert, Franz:
- Piano Sonata in E Major, D. 157 – Mvt. 2

Tchaikovsky, Pyotr Ilyich:
- Nocturne, Op. 19, No. 4

Ternary:

Bach, Johann Sebastian:
- Cantata No. 78 (Jesu der du meine Seele) – Mvt. 2

Beethoven, Ludwig van:
- Quartet, Op. 135 – Mvt. 3
- Sonata No. 4, Op. 7 – Mvt. 2
- Sonata No. 12, Op. 26 – Mvt. 3
- Sonata No. 15, Op. 28 – Mvt. 3
- Symphony No. 3 (Eroica) – Mvt. 2

Brahms, Johannes:
- Ein Deutches Requiem – Mvt. 1
- Ein Deutches Requiem – Mvt. 2
- Ein Deutches Requiem – Mvt. 3
- Ein Deutches Requiem – Mvt. 5
- Ein Deutches Requiem – Mvt. 7
- Intermezzo, Op. 118, No. 2

Intermezzo, Op. 119, No. 2
Chopin, Frédéric:
Mazurka, Op. 7, No. 1 (No. 5)
Mazurka, Op. 63, No. 3
Polonaise in A Major, Op. 40, No. 1
Prelude, Op. 28, No. 12
Prelude, Op. 28, No. 13
Waltz in A Minor, Op. 34, No. 2
Fauré, Gabriel:
Requiem, Op. 48 – Mvt. 2 ("Offertory")
Handel, George Frideric:
The Trumpet Shall Sound (from Messiah)
Waft Her, Angels, to the Skies (from Jephtha)
Lang, Josephine:
Two Lieder, Op. 28, No. 1 – Traumbild
Mendelssohn, Felix:
Song without Words, Op. 53, No. 5
Schumann, Robert:
Nachtstücke, Op. 23, No. 3

Arch:

Bach, Johann Sebastian:
Well-Tempered Clavier, Vol. 1, Fugue No. 16 in G Minor
Barber, Samuel:
String Quartet Op. 11 – Mvt. 2 ("Adagio for Strings")
Brahms, Johannes:
Rhapsody in E Flat Major, Op. 119, No. 4
Chopin, Frédéric:
Waltz, Op. 34, No. 1

Theme and Variation:

Bach, Johann Sebastian:
Chaconne from Partita No. 2 in D Minor, BWV 1004
Crucifixus from Mass in B Minor, BWV 232
Magnificat in D Major, No. 5, BWV 243
Passacaglia in C Minor for organ, BWV 582
Violin Concerto in E Major, BWV 1042 – Mvt. 2
Beethoven, Ludwig van:
Diabelli Variations, Op. 120
Piano Sonata No. 12 in Ab Major, Op. 26 – Mvt. 1
Quartet No. 15 in A Minor, Op. 132 – Mvt. 3
Quartet No. 16, Op. 135 – Mvt. 2 (mm. 143-190)
Sonata No. 10 in G Major, Op. 14, No. 2 – Mvt. 2
Symphony No. 5 – Mvt. 2

Symphony No. 7 – Mvt. 2
Symphony No. 9 – Mvt. 3
Thirty-Two Variations for piano in C Minor, WoO 80
Brahms, Johannes:
Symphony No. 4 – Mvt. 4
Variations on a Theme by Haydn
Debussy, Claude:
String Quartet in G Minor, Op. 10 – Mvt. 2
Dvořák, Antonín:
Tema con Variazioni, Op. 36
Handel, George Frideric:
Suite No. 7 in G Minor – Mvt. 6 (“Passacaille”)
Haydn, Franz Joseph:
Piano Variations in F Minor, Hob.XVII:6
Piano Trio in G Major, Hob.XV:25 – Mvt. 1
Mozart, Wolfgang Amadeus:
Concert Rondo, K. 382
Piano Concerto in C Minor, K. 491 – Mvt. 3
Theme with Variations from Sonata, K. 284
Twelve Variations "Ah, vous dirai-je, maman", K. 265
Purcell, Henry:
When I Am Laid in Earth (from Dido and Aeneas)
Schubert, Franz:
Der Doppelgänger (Schwanengesang), D. 957
Vivaldi, Antonio:
Concerto in A Minor (2 Violins), Op. 3, No. 8 – Mvt. 1

Minuet & Trio:

Bach, Johann Sebastian:
French Suite No. 3 (“Menuet and Trio”) – Mvt. 5 & 6
Beethoven, Ludwig van:
Minuet in G, WoO 10, No. 2
Brahms, Johannes:
Cello Sonata No. 1, Op. 38 – Mvt. 2
Mozart, Wolfgang Amadeus:
Minuet and Trio in G, No. 1, K. 1
String Quartet No. 14, K. 387 – Mvt. 2

Bar:

Bach, Johann Sebastian:
Jesu, Meine Freude, BWV 227 – Mvt. 1
Jesu, Meine Freude, BWV 227 – Mvt. 5
Jesu, Meine Freude, BWV 227 – Mvt. 9
Wachet auf, ruft uns die Stimme, BWV 140 – Mvt. 7

Ritornello:

Bach, Johann Sebastian:
- Art of Fugue, Contrapunctus No. 9
- Invention No. 1
- Invention No. 2
- Invention No. 3
- Invention No. 4
- Invention No. 5
- Invention No. 6
- Invention No. 7
- Invention No. 8
- Invention No. 9
- Invention No. 10
- Invention No. 11
- Invention No. 12
- Invention No. 13
- Invention No. 14
- Invention No. 15
- Well-Tempered Clavier, Vol. 2, Fugue No. 4
- Well-Tempered Clavier, Vol. 2, Fugue No. 5
- Well-Tempered Clavier, Vol. 2, Fugue No. 6
- Well-Tempered Clavier, Vol. 2, Fugue No. 14
- Well-Tempered Clavier, Vol. 2, Fugue No. 20

Rondo:

Bach, Johann Sebastian:
- Violin Concerto in E Major, BWV 1042 – Mvt. 3

Beethoven, Ludwig van:
- Rage Over a Lost Penny, Op. 129
- Sonata No. 2 in A Major, Op. 2, No. 2 – Mvt. 4
- Sonata No. 3 in C Major, Op. 2, No. 3 – Mvt. 4
- Sonata No. 8, Op. 13 ("Pathetique") – Mvt. 2
- Sonata No. 8, Op. 13 ("Pathetique") – Mvt. 3
- Violin Sonata No. 4, Op. 23 – Mvt. 3

Brahms, Johannes:
- Ein Deutches Requiem – Mvt. 4
- Quartet for Piano and Strings, Op. 25 – Mvt. 4

Chopin, Frédéric:
- Piano Concerto No. 1 in E Minor, Op. 11 – Mvt. 3
- Rondo in C Minor, Op. 1

Dvořák, Antonín:
- Cello Concerto in B Minor, Op. 104, B. 191 – Mvt. 3
- Rondo for Cello and Orchestra in G Minor, Op. 94

Haydn, Franz Joseph:
- Piano Sonata No. 50 in D Major, Hob.XVI:37 – Mvt. 3

Symphony No. 101 ("Clock") – Mvt. 4
Trio No. 39 in G Major, Hob.XV:25 – Mvt. 3
Mahler, Gustav:
Symphony No. 5 – Mvt. 5
Mozart, Wolfgang Amadeus:
Piano Sonata K. 331 – Mvt. 3
Piano Sonata in C Major, K. 467 – Mvt. 3
Sonata No. 3, K. 281 – Mvt. 3
Sonata No. 8, K. 310 – Mvt. 3
Sonata No. 14, K. 457 – Mvt. 3
Sonata No. 15, K. 494 (Rondo)
Stravinsky, Igor:
Firebird Suite – Mvt. 9 ("Rondeau des Princesses")

Sonata:

Beethoven, Ludwig van:
Leonora Overture No. 2, Op. 72
Quartet No. 14, Op. 131 – Mvt. 2
Quartet No. 15, Op. 132 – Mvt. 1
Quartet No. 16, Op. 135 – Mvt. 4
Sonata No. 5, Op. 10, No. 1 – Mvt. 2
Sonata No. 8, Op. 13 ("Pathetique") – Mvt. 1
Sonata No. 14 ("Moonlight"), Op. 27, No. 2 – Mvt. 1
Sonata No. 21, Op. 53 – Mvt. 1
Sonata No. 29, Op. 106 – Mvt. 1
Sonata No. 30, Op. 109 – Mvt. 2
Symphony No. 9 – Mvt. 1
Violin Sonata No. 1, Op. 12, No. 1 – Mvt. 1
Brahms, Johannes:
Cello Sonata No. 1, Op. 38 – Mvt. 3
Clarinet Sonata, Op. 120, No. 2 – Mvt. 1
Symphony No. 1 – Mvt. 4
Symphony No. 3 – Mvt. 4
Tragic Overture, Op. 81
Haydn, Franz Joseph:
Quartet, Op. 77, No. 1 – Mvt. 1
Sonata No. 48 in C Major, Hob.XVI:35 – Mvt. 1
Symphony No. 94 ("Surprise") – Mvt. 1
Symphony No. 100 – Mvt. 1
Symphony No. 102 – Mvt. 1
Symphony No. 103 ("Drumroll") – Mvt. 1
Symphony No. 104 – Mvt. 4
Mendelssohn, Felix:
Symphony No. 4 ("Italian") – Mvt. 1
Symphony No. 4 ("Italian") – Mvt. 4

Mozart, Wolfgang Amadeus:
- Overture (from The Magic Flute)
- Piano Sonata K. 545 – Mvt. 1
- Piano Sonata No. 8, K. 310 – Mvt. 1
- Sonata in C Major K. 279 – Mvt. 1
- Sonata in G Major K. 283 – Mvt. 1
- Sonata K. 333 – Mvt. 3
- Symphony No. 36 ("Linz"), K. 425 – Mvt. 1
- Symphony No. 39, K. 543 – Mvt. 1

Unique:

Bach, Johann Sebastian:
- Well-Tempered Clavier, Vol. 1, Prelude No. 9
- Well-Tempered Clavier, Vol. 1, Prelude No. 21

Berlioz, Hector:
- Symphonie Fantastique – Mvt. 1
- Symphonie Fantastique – Mvt. 3
- Symphonie Fantastique – Mvt. 4
- Symphonie Fantastique – Mvt. 5

Brahms, Johannes:
- Intermezzo, Op. 119, No. 3
- Alto Rhapsody, Op. 53

Chopin, Frédéric:
- Ballade in G Minor, Op. 23
- Prelude, Op. 28, No. 24

Debussy, Claude:
- L'après-midi d'un faune
- Prelude No. 10 (“The Engulfed Cathedral”)

Fauré, Gabriel:
- Élégie, Op. 24

Mahler, Gustav:
- Symphony No. 4 – Mvt. 1

Mozart, Wolfgang Amadeus:
- Fantasia, K. 475
- Magic Flute, No. 3

Schubert, Franz:
- Winterreise, No. 5
- Winterreise, No. 11
- Winterreise, No. 16

Strauss, Richard:
- Don Juan, Op. 20

Wagner, Richard:
- Prelude to Tristan und Isolde

B.1 Dataset Features

The tabulated dataset contains the mean and variance of the following 2D features:

- Mel Spectrogram SSM
- MFCC Spectrogram SSLMs (Euclidian and Cosine distances),
- Chromagram SSLMs (Euclidian and Cosine distances)
- STFT Chromagram
- Constant-Q Transform Chromagram
- Energy Normalized Chromagram (CENS)
- Mel Spectrogram
- MFCC Spectrogram
- Spectral Bandwidth
- Spectral Centroid
- Spectral Contrast
- Spectral Flatness
- Spectral Rolloff
- Polynomial Features
- Tonal Centroid Features (Tonnetz)
- Zero Crossing Rate
- Tempogram
- Fourier Tempogram

APPENDIX C
RELATED DEFINITIONS

This appendix presents additional terminology that may aid in familiarizing the reader with related subject vocabulary.

C.1 Music Theory Terms

Musical Structures:

- **Phraselet –** A subdivision of a phrase that may or may not end with a cadence, typically two measures (bars) long [38].
- **Period** – A structure of two (or more) consecutive phrases, related typically by tonal structure or harmonic organization [2, p. 55]. Both phrases may begin with the same melodic material (**Parallel Period**) or not (**Contrasting Period**) [38].
    - **Antecedent (Question)** – The first of two phrases in a period [39].
    - **Consequent (Answer)** – The second of two phrases in a period [39].
    - **Double Period** – A period in which the Antecedent and Consequent are both comprised of two phrases [38].
- **Link/Bridge** – A musical passage too short to be classified as a phrase and is independent of neighboring phrases [38].
- **Elision** – A musical event where the cadence of one phrase aligns exactly with the start of the succeeding phrase.
- **Sequence** – A musical idea repeated at a different pitch level or series of pitch levels [38].

C.1.1 Sonata Form

Sonata (or **sonata-allegro**) form is a composite form comprised of numerous distinct parts with a variety of compositional rules/conventions, the primary being the **sonata principle** (often used in other forms such as rondo), where material first stated in a key other than the tonic key (the original key of the piece) must return later in the tonic key [47]. Sonatas are comprised of a series of **Themes** (excluding an optional **Introduction** before the Expos. or **Coda** after the Recap.), which are categorized by their function rather than a series of capital letter labels (see Figure C.1) [38]:

- **Primary Theme (PT) –** The theme(s) stated in the tonic key until the Transition begins.
- **Transition (Trans.)** – A section that seeks to modulate to a new key in the Exposition.
- **Secondary Theme (ST)** – The theme(s) stated in the new key after the Transition.
- **Closing Theme (CT)** – The theme(s) which sound functionally closing in the Exposition.
- **Retransition (Retrans.)** – A (Dev.) section that seeks to modulate back to the tonic key.

Hence, the form itself is divided into three parts, each with distinct responsibilities for the themes:

- **Exposition (Part A) –** Introduces the PT and modulates through a Trans. before introducing the ST. Sometimes contains a smaller form.
- **Development (B) –** An unstable part of the piece which "develops" previous themes (manipulates them in various manners) and frequently modulates to different keys.
- **Recapitulation (A/A′) –** The return of part A but without the modulation during the Trans.

It is worth noting that these definitions are an oversimplification of the full list of conventions applied to sonatas, simply for the sake of describing each part as necessary for the neural network.

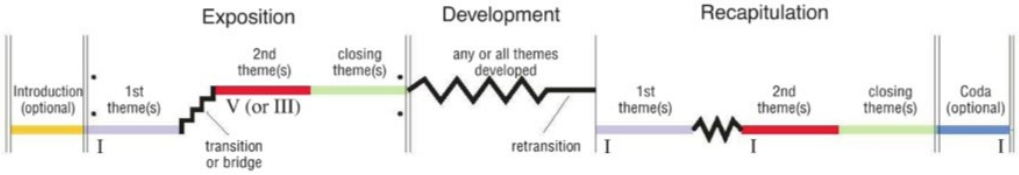


Figure C.1: Generalized sonata form diagram [48]

C.1.2 Cadence Variants

Cadences are generally divided into four distinct variants (see Table C.1 (b)) [41, Ch. 10]:

- **Authentic Cadence (AC) –** The strongest form of cadence, the movement of the dominant or leading tone chord to the tonic chord (V or vii^{o}–I); acts as the conclusion to a phrase. It may be either **Perfect** (**PAC,** V or V^{7}–I) where both chords are in root position and $\hat{1}$ (the tonic note) is in the melody voice of the I chord, or an **Imperfect (IAC)** otherwise.
- **Deceptive Cadence (DC) –** A progression that appears to be moving to an authentic cadence but ends on a different chord excluding the tonic – typically V–VI or vi, or V–something else – subverting the listener's expectations of hearing a conclusive cadence.
- **Half Cadence (HC) –** The most common inconclusive/unstable cadence; a progression from any chord to the dominant (x–V), acting as a pause awaiting resolution between phrases.
- **Plagal Cadence (PC) –** A final sounding cadence, typically a IV–I progression which often follows a PAC but is less important and conclusive than an authentic cadence.

Cadential Strength	
Conclusive Cadences:	PC, IAC, PAC
Progressive Cadences:	DC, HC

(a)

		Ends on a tonic chord?	
		Yes	No
Previous chord contains leading tone?	Yes	AC (ex: III–I)	DC
	No	PC (ex: ii–I)	HC

(b)

Table C.1: Cadential strength by degree of conclusion (a) [2], tonal classification (b) [41]

C.2 Machine/Deep Learning Terms

Additional deep learning model architectures:

- **Deep Q-Network (DQN) –** A special neural network designed for reinforcement learning tasks in which input states are mapped to (action, *Q*-value) pairs (where *Q*-value refers to the estimated optimal future value using the *Q* function/Bellman Equation) to approximate some state-value function or perform an experience replay [17]; used for Deep *Q*-learning tasks such as simulating intelligent video gameplay and self-driving vehicles [18].
- **Quantum Neural Network (QNN) –** A neural network that combines concepts from ANNs with Quantum Computing, including quantum circuits, states, and activation functions; used for quantum machine learning tasks [16].
- **Generative Adversarial Network (GAN) –** A special neural network comprised of a Generative model (for supervised tasks) and a Discriminative model (for unsupervised tasks, judges the accuracy of the Generative model output) that uses unlabeled datasets to select samples from a distribution of the data developed from competition between the two models to generate new, realistic data; used for generating media such as imagery, video, language, and audio [19].

APPENDIX D
IMPLEMENTATION OF FORM-NN SYSTEM

This appendix presents an overview of the code used to implement the final system, including the Form and Phrase Analyzers, Peak-Picking algorithm, and assisting functions for data preparation. Some code has been deprecated due to poor performance or lack of present purpose for the final models and has such been removed from this thesis but can be found on GitHub [66].

D.1 Main.py

```
1.  import itertools
2.  import time
3.  import glob as gb
4.  import librosa
5.  import matplotlib.pyplot as plt
6.  import librosa.display
7.  import pickle
8.  import pandas as pd
9.  from sklearn.metrics import confusion_matrix, accuracy_score
10. import os
11. import soundfile as sf
12. import sys
13. import warnings
14. from keras.utils.vis_utils import plot_model
15. from sklearn.model_selection import train_test_split
16. from sklearn.preprocessing import LabelEncoder
17. import tensorflow.keras as keras
18. from sklearn.svm import LinearSVC
19. from tensorflow.keras.layers import Input
20. from tensorflow.keras.regularizers import l2, l1_l2
21. import seaborn as sns
22. from tensorflow.keras.callbacks import EarlyStopping, ModelCheckpoint
23. from sklearn.metrics import classification_report
24. import tensorflow as tf
25. from tensorflow import keras
26. from tensorflow.keras import layers
27. import statistics
28. from sklearn import tree
29. from sklearn.dummy import DummyClassifier
30. from tensorflow.keras.utils import to_categorical
31. from tensorflow.keras.models import Sequential
32. import random
33. from numpy import inf
34. import audioread
35. import librosa.segment
36. import numpy as np
37. import data_utils as du
38. import data_utils_input as dus
39. from data_utils_input import normalize_image, padding_MLS, padding_SSLM, borders
40. from keras import backend as k
41. from shutil import copyfile
42. import fnmatch
```

```
43. from sklearn import preprocessing
44. from sklearn.feature_selection import SelectKBest
45. from sklearn.feature_selection import f_classif
46. from ast import literal_eval
47. from sklearn.feature_selection import RFE
48. from skimage.transform import resize
49. from tensorflow.python.ops.init_ops_v2 import glorot_uniform
50. import lightgbm as lgb
51. from treegrad import TGDClassifier
52. from sklearn.preprocessing import MultiLabelBinarizer
53. import logging
54. 
55. tf.get_logger().setLevel(logging.ERROR)
56. k.set_image_data_format('channels_last')
57. os.environ['TF_CPP_MIN_LOG_LEVEL'] = '3'
58. # os.environ["CUDA_VISIBLE_DEVICES"] = "-1"
59. gpus = tf.config.experimental.list_physical_devices('GPU')
60. for gpu in gpus:
61.     tf.config.experimental.set_memory_growth(gpu, True)
62. 
63. if not sys.warnoptions:
64.     warnings.simplefilter("ignore")  # ignore warnings
65. warnings.filterwarnings("ignore", category=DeprecationWarning)
66. 
67. 
68. # region Directories
69. MASTER_DIR = 'D:/Google Drive/Resources/Dev/Python/Machine Learning/Master Thesis/'
70. MASTER_INPUT_DIR = 'F:/Master Thesis Input/'
71. MASTER_LABELPATH = os.path.join(MASTER_INPUT_DIR, 'Labels/')
72. WEIGHT_DIR = os.path.join(MASTER_DIR, 'Weights/')
73. 
74. MIDI_Data_Dir = np.array(gb.glob(os.path.join(MASTER_DIR, 'Data/MIDIs/*')))
75. FULL_DIR = os.path.join(MASTER_INPUT_DIR, 'Full/')
76. FULL_MIDI_DIR = os.path.join(FULL_DIR, 'MIDI/')
77. FULL_LABELPATH = os.path.join(MASTER_LABELPATH, 'Full/')
78. # endregion
79. 
80. 
81. # region DataTools
82. def generate_label_files():
83.     """
84.     Generate label '.txt' file for each MIDI in its respective Form-folder.
85.     Pre-timestamps each file with silence and end times
86.     """
87.     cnt = 1
88.     for folder in MIDI_Data_Dir:
89.         for file in os.listdir(folder):
90.             foldername = folder.split('\\')[-1]
91.             filename, name = file, file.split('/')[-1].split('.')[0]
92.             print(f"\nWorking on {os.path.basename(name)}, file #" + str(cnt))
93. 
94.             path = os.path.join(os.path.join(MASTER_DIR, 'Labels/'), foldername) + '/' +
    os.path.basename(name) + '.txt'
95.             if not os.path.exists(path):
96.                 with audioread.audio_open(folder + '/' + filename) as f:
97.                     print("Reading duration of " + os.path.basename(name))
98.                     totalsec = f.duration
99.                 fwrite = open(path, "w+")
100.                fwrite.write("0.000\tSilence\n" + str(totalsec) + '00\tEnd')
101.                fwrite.close()
102.            cnt += 1
103. 
104. 
105.def get_total_duration():
106.    """
107.    Return the sum of all audio file durations together
108.    """
109.    dur_sum = 0
110.    for folder in MIDI_Data_Dir:
```

```
        for file in os.listdir(folder):
            filename, name = file, file.split('/')[-1].split('.')[0]
            with audioread.audio_open(folder + '/' + filename) as f:
                dur_sum += f.duration
    print("Total duration: " + str(dur_sum) + " seconds")
    # Total duration: 72869.0 seconds
    # = 1214.4833 minutes = 20.241389 hours = 20 hours, 14 minutes, 29 seconds
    return dur_sum


def prepare_model_training_input():
    print("Preparing MLS inputs")
    dus.util_main(feature="mls")

    print("\nPreparing SSLM-MFCC-COS inputs")
    dus.util_main(feature="mfcc", mode="cos")
    print("\nPreparing SSLM-MFCC-EUC inputs")
    dus.util_main(feature="mfcc", mode="euc")

    print("\nPreparing SSLM-CRM-COS inputs")
    dus.util_main(feature="chroma", mode="cos")
    print("\nPreparing SSLM-CRM-EUC inputs")
    dus.util_main(feature="chroma", mode="euc")


def multi_input_generator_helper(gen1, gen2, gen3, gen4, concat=True):
    while True:
        sslm1 = next(gen1)[0]
        sslm2 = next(gen2)[0]
        sslm3 = next(gen3)[0]
        sslm4 = next(gen4)[0]
        if not concat:
            yield [sslm1, sslm2, sslm3, sslm4], sslm1.shape
            continue

        if sslm2.shape != sslm1.shape:
            sslm2 = resize(sslm2, sslm1.shape)
        if sslm3.shape != sslm1.shape:
            sslm3 = resize(sslm3, sslm1.shape)
        if sslm4.shape != sslm1.shape:
            sslm4 = resize(sslm4, sslm1.shape)
        yield tf.expand_dims(
            np.concatenate((sslm1,
                            np.concatenate((sslm2,
                                            np.concatenate((sslm3, sslm4),
                                                           axis=-1)), axis=-1)), axis=-1), axis=-1),
    sslm1.shape


def multi_input_generator(gen1, gen2, gen3, gen4, gen5, gen6, feature=2, concat=True,
    expand_dim_6=True, augment=False):
    while True:
        mlsgen = next(gen1)
        mlsimg = mlsgen[0]
        if augment:
            yield [mlsimg, [[0, 0], [0, 0], [0, 0], [0, 0]],
                   next(gen6)[0]], mlsgen[1][feature]  # tf.expand_dims(next(gen6)[0], axis=0)],
    mlsgen[1][feature]
        else:
            sslmimgs, sslmshape = next(multi_input_generator_helper(gen2, gen3, gen4, gen5, concat))
            if not expand_dim_6:
                yield [mlsimg, sslmimgs, next(gen6)[0]], mlsgen[1][feature]
                continue
            if mlsimg.shape != sslmshape:
                mlsimg = resize(mlsimg, sslmshape)
            yield [mlsimg, sslmimgs, tf.expand_dims(next(gen6)[0], axis=0)], mlsgen[1][feature]


def get_column_dataframe():
```

```
    df = pd.DataFrame(columns=['piece_name', 'composer', 'filename', 'duration',
                               'ssm_log_mel_mean', 'ssm_log_mel_var',
                               'sslm_chroma_cos_mean', 'sslm_chroma_cos_var',
                               'sslm_chroma_euc_mean', 'sslm_chroma_euc_var',
                               'sslm_mfcc_cos_mean', 'sslm_mfcc_cos_var',
                               'sslm_mfcc_euc_mean', 'sslm_mfcc_euc_var',  # ---{
                               'chroma_cens_mean', 'chroma_cens_var',
                               'chroma_cqt_mean', 'chroma_cqt_var',
                               'chroma_stft_mean', 'chroma_stft_var',
                               'mel_mean', 'mel_var',
                               'mfcc_mean', 'mfcc_var',
                               'spectral_bandwidth_mean', 'spectral_bandwidth_var',
                               'spectral_centroid_mean', 'spectral_centroid_var',
                               'spectral_contrast_mean', 'spectral_contrast_var',
                               'spectral_flatness_mean', 'spectral_flatness_var',
                               'spectral_rolloff_mean', 'spectral_rolloff_var',
                               'poly_features_mean', 'poly_features_var',
                               'tonnetz_mean', 'tonnetz_var',
                               'zero_crossing_mean', 'zero_crossing_var',
                               'tempogram_mean', 'tempogram_var',
                               'fourier_tempo_mean', 'fourier_tempo_var',  # }---
                               'formtype'])
    return df


def create_form_dataset(filedir=FULL_DIR, labeldir=FULL_LABELPATH, outfile='full_dataset.xlsx',
    augment=False):
    # if augment then ignore sslms and replace with [0, 0]

    mls_full = dus.BuildDataloader(os.path.join(filedir, 'MLS/'),
                                   label_path=labeldir, batch_size=1, reshape=False)
    midi_full = dus.BuildMIDIloader(os.path.join(filedir, 'MIDI/'),
                                    label_path=labeldir, batch_size=1, reshape=False,
    building_df=True)
    if not augment:
        sslm_cmcos_full = dus.BuildDataloader(os.path.join(filedir, 'SSLM_CRM_COS/'),
                                              label_path=labeldir, batch_size=1, reshape=False)
        sslm_cmeuc_full = dus.BuildDataloader(os.path.join(filedir, 'SSLM_CRM_EUC/'),
                                              label_path=labeldir, batch_size=1, reshape=False)
        sslm_mfcos_full = dus.BuildDataloader(os.path.join(filedir, 'SSLM_MFCC_COS/'),
                                              label_path=labeldir, batch_size=1, reshape=False)
        sslm_mfeuc_full = dus.BuildDataloader(os.path.join(filedir, 'SSLM_MFCC_EUC/'),
                                              label_path=labeldir, batch_size=1, reshape=False)

        print("Done building dataloaders, merging...")
        full_datagen = multi_input_generator(mls_full, sslm_cmcos_full, sslm_cmeuc_full,
    sslm_mfcos_full,
                                             sslm_mfeuc_full, midi_full, concat=False,
    expand_dim_6=False)
        print("Merging complete. Printing...")
    else:
        print("Done building dataloaders, merging...")
        full_datagen = multi_input_generator(mls_full, None, None, None, None,
                                             midi_full, concat=False, expand_dim_6=False,
    augment=True)
        print("Merging complete. Printing...")

    np.set_string_function(
        lambda x: repr(x).replace('(', '').replace(')', '').replace('array', '').replace("       ", '
    '), repr=False)
    np.set_printoptions(threshold=inf)

    df = get_column_dataframe()
    label_encoder = LabelEncoder()
    label_encoder.classes_ = np.load(os.path.join(WEIGHT_DIR, 'form_classes.npy'))
    for indx, cur_data in enumerate(full_datagen):
        if indx == len(mls_full):
            break
```

```
239.         c_flname = mls_full.getSong(indx).replace(".wav.npy", "").replace(".wav", "").replace(".npy",
    "")
240.         c_sngdur = mls_full.getDuration(indx)
241.         c_slmmls = cur_data[0][0]
242.         c_scmcos = cur_data[0][1][0]
243.         c_scmeuc = cur_data[0][1][1]
244.         c_smfcos = cur_data[0][1][2]
245.         c_smfeuc = cur_data[0][1][3]
246.         c_midinf = cur_data[0][2]
247.         c_flabel = cur_data[1]
248.         c_flabel = label_encoder.inverse_transform(np.where(c_flabel == 1)[0])[0]
249.
250.         df.loc[indx] = ["", "", c_flname, c_sngdur, c_slmmls[0], c_slmmls[1], c_scmcos[0],
    c_scmcos[1],
251.                         c_scmeuc[0], c_scmeuc[1], c_smfcos[0], c_smfcos[1], c_smfeuc[0], c_smfeuc[1],
252.                         c_midinf[2], c_midinf[3], c_midinf[4], c_midinf[5], c_midinf[6], c_midinf[7],
253.                         c_midinf[8], c_midinf[9], c_midinf[10], c_midinf[11], c_midinf[12],
    c_midinf[13],
254.                         c_midinf[14], c_midinf[15], c_midinf[0], c_midinf[1], c_midinf[16],
    c_midinf[17],
255.                         c_midinf[18], c_midinf[19], c_midinf[20], c_midinf[21], c_midinf[22],
    c_midinf[23],
256.                         c_midinf[24], c_midinf[25], c_midinf[26], c_midinf[27], c_midinf[28],
    c_midinf[29], c_flabel]
257.     for col in df.columns:
258.         df[col] = df[col].apply(lambda x: str(x)
259.                                 .replace(", dtype=float32", "").replace("],", "]")
260.                                 .replace("dtype=float32", "").replace("...,", ""))
261.     # df.to_csv(os.path.join(MASTER_DIR, 'full_dataset.csv'), index=False)
262.     df.to_excel(os.path.join(MASTER_DIR, outfile), index=False)
263.
264.
265.def prepare_augmented_audio(inpath=FULL_DIR, savepath='', augmentation=1):
266.     if not os.path.exists(savepath):
267.         os.makedirs(savepath)
268.         print("New directory created:", savepath)
269.
270.     def inject_noise(adata, noise_factor):
271.         noise = np.random.randn(len(adata))
272.         augmented_data = adata + noise_factor * noise
273.         augmented_data = augmented_data.astype(type(adata[0]))
274.         return augmented_data
275.
276.     def shift_time(adata, sampling_rate, shift_max, shift_direction):
277.         shift = np.random.randint(sampling_rate * shift_max)
278.         if shift_direction == 'right':
279.             shift = -shift
280.         elif shift_direction == 'both':
281.             direction = np.random.randint(0, 2)
282.             if direction == 1:
283.                 shift = -shift
284.         augmented_data = np.roll(adata, shift)
285.         # Set to silence for heading/ tailing
286.         if shift > 0:
287.             augmented_data[:shift] = 0
288.         else:
289.             augmented_data[shift:] = 0
290.         return augmented_data
291.
292.     def shift_pitch(adata, sampling_rate, pitch_factor):
293.         return librosa.effects.pitch_shift(adata, sampling_rate, n_steps=pitch_factor)
294.
295.     def shift_speed(adata, speed_factor):
296.         return librosa.effects.time_stretch(adata, speed_factor)
297.
298.     start_time = time.time()
299.     for (dir_path, dnames, fnames) in os.walk(inpath):
300.         for f in fnames:
301.             augdatapath = savepath + f.split('.')[0] + '_aug' + str(augmentation) + '.wav'
```

```
302.            if os.path.exists(augdatapath):
303.                continue
304.            start_time_song = time.time()
305.            fdatapath = dir_path + '/' + f
306.            y, sr = librosa.load(fdatapath, sr=None)
307.            sr = 44100
308.            if augmentation == 1:
309.                y = shift_speed(y, 0.7)  # Slower
310.                y = shift_pitch(y, sr, -6)  # Shift down 6 half-steps (tritone)
311.                y = shift_time(y, sr, random.random(), 'right')
312.                y = inject_noise(y, 0.005)
313.            elif augmentation == 2:
314.                y = shift_speed(y, 1.4)  # Faster
315.                y = shift_pitch(y, sr, 4)  # Shift up 4 half-steps (major 3rd)
316.                y = shift_time(y, sr, random.random(), 'right')
317.                y = inject_noise(y, 0.01)
318.            elif augmentation == 3:
319.                y = shift_speed(y, 0.5)
320.                y = shift_pitch(y, sr, 7)  # Shift up perfect 5th
321.                y = shift_time(y, sr, random.random(), 'right')
322.                y = inject_noise(y, 0.003)
323.            elif augmentation == 4:
324.                y = shift_speed(y, 2)
325.                y = shift_pitch(y, sr, 8)  # Shift up minor 6th
326.                y = shift_time(y, sr, random.random(), 'right')
327.                y = inject_noise(y, 0.02)
328.            elif augmentation == 5:
329.                y = shift_speed(y, 1.1)
330.                y = shift_pitch(y, sr, 1)  # Shift up minor 2nd
331.                y = shift_time(y, sr, random.random(), 'right')
332.                y = inject_noise(y, 0.007)
333.            sf.write(augdatapath, y, sr)
334.            print("Successfully saved file:", augdatapath, "\tDuration: {:.2f}s".format(time.time() -
    start_time_song))
335.    print("All files have been converted. Duration: {:.2f}s".format(time.time() - start_time))
336.    pass
337.
338.
339.def generate_augmented_datasets():
340.    for i in range(1, 6):
341.        prepare_augmented_audio(savepath=os.path.join(MASTER_INPUT_DIR, 'Aug' + str(i) + '/MIDI/'),
    augmentation=i)
342.        dus.util_main(feature="mls", inpath=os.path.join(MASTER_INPUT_DIR, 'Aug' + str(i) + '/'),
343.                      midpath=os.path.join(MASTER_INPUT_DIR, 'Aug' + str(i) + '/'))
344.        create_form_dataset(filedir=os.path.join(MASTER_INPUT_DIR, 'Aug' + str(i) + '/'),
    augment=True,
345.                            outfile='full_dataset_aug' + str(i) + '.xlsx')
346.    df = pd.read_excel(os.path.join(MASTER_DIR, 'full_dataset.xlsx'))
347.    df1 = pd.read_excel(os.path.join(MASTER_DIR, 'full_dataset_aug1.xlsx'))
348.    df2 = pd.read_excel(os.path.join(MASTER_DIR, 'full_dataset_aug2.xlsx'))
349.    df3 = pd.read_excel(os.path.join(MASTER_DIR, 'full_dataset_aug3.xlsx'))
350.    df4 = pd.read_excel(os.path.join(MASTER_DIR, 'full_dataset_aug4.xlsx'))
351.    df5 = pd.read_excel(os.path.join(MASTER_DIR, 'full_dataset_aug5.xlsx'))
352.    df = pd.concat([df, df1, df2, df3, df4, df5], ignore_index=True).reset_index()
353.    df.to_excel(os.path.join(MASTER_DIR, 'Data/full_augmented_dataset.xlsx'), index=False)
354.
355.
356.def prepare_lstm_peaks():
357.    MIDI_FILES = os.path.join(MASTER_INPUT_DIR, 'Full/MIDI/')
358.    PEAK_DIR = os.path.join(MASTER_INPUT_DIR, 'Full/PEAKS/')
359.    cnt = len(os.listdir(PEAK_DIR)) + 1
360.    for file in os.listdir(MIDI_FILES):
361.        foldername = MIDI_FILES.split('\\')[-1]
362.        filename, name = file, file.split('/')[-1].split('.')[0]
363.        if str(os.path.basename(name)) + ".npy" in os.listdir(PEAK_DIR):
364.            continue
365.        print(f"\nWorking on {os.path.basename(name)}, file #" + str(cnt))
366.        fullfilename = MIDI_FILES + '/' + filename
367.        peaks = du.peak_picking(fullfilename, name, foldername, returnpeaks=True)
```

```
368.        print(peaks)
369.        np.save(os.path.join(PEAK_DIR, os.path.basename(name)), peaks)
370.        cnt += 1
371.# endregion
372.
373.
374.# region FormModel
375.def trainFormModel():
376.    # region DataPreProcessing
377.    df = pd.read_excel(os.path.join(MASTER_DIR, 'Data/full_augmented_dataset.xlsx'))
378.    # df = pd.read_excel(os.path.join(MASTER_DIR, 'full_dataset.xlsx'))
379.    names = df[['piece_name', 'composer', 'filename']]
380.    y = df['formtype']
381.    # """
382.    df = df.drop(columns=['sslm_chroma_cos_mean', 'sslm_chroma_cos_var', 'sslm_chroma_euc_mean',
    'sslm_chroma_euc_var',
383.                          'sslm_mfcc_cos_mean', 'sslm_mfcc_cos_var', 'sslm_mfcc_euc_mean',
    'sslm_mfcc_euc_var'])
384.    # """
385.    df.drop(columns=['spectral_bandwidth_var', 'spectral_centroid_var', 'spectral_flatness_var',
    'spectral_rolloff_var',
386.                     'zero_crossing_var', 'fourier_tempo_mean', 'fourier_tempo_var'], inplace=True)
    # Remove useless
387.    # nonlist = df[['duration', 'spectral_contrast_var']]
388.    nonlist = df[['duration']]
389.    df.drop(columns=['piece_name', 'composer', 'filename', 'duration', 'spectral_contrast_var',
    'formtype'],
390.            inplace=True)
391.    # df = df[['ssm_log_mel_mean', 'ssm_log_mel_var', 'mel_mean', 'mel_var', 'chroma_stft_mean',
    'chroma_stft_var']]
392.    # df = df[['ssm_log_mel_mean', 'ssm_log_mel_var']]
393.    df = df[['ssm_log_mel_mean']]  # best decision tree accuracy
394.    print("Fixing broken array cells as needed...")
395.
396.    def fix_broken_arr(strx):
397.        if '[' in strx:
398.            if ']' in strx:
399.                return strx
400.            else:
401.                return strx + ']'
402.
403.    for col in df.columns:
404.        df[col] = df[col].apply(lambda x: fix_broken_arr(x))
405.    # print("Headers:", pd.concat([pd.concat([names, pd.concat([nonlist, df], axis=1)], axis=1), y],
    axis=1).columns)
406.    # Headers: Index(['piece_name', 'composer', 'filename', 'duration', 'ssm_log_mel_mean',
    'formtype'], dtype='object')
407.
408.    print("Done processing cells, building training set...")
409.
410.    # d = [pd.DataFrame(df[col].astype(str).apply(literal_eval).values.tolist()).add_prefix(col) for
    col in df.columns]
411.    d = [pd.DataFrame(df[col].astype(str).apply(literal_eval).values.tolist()) for col in df.columns]
412.    df = pd.concat(d, axis=1).fillna(0)
413.    df = pd.concat([pd.concat([names, pd.concat([nonlist, df], axis=1)], axis=1), y], axis=1)  #
    print(df)
414.    train, test = train_test_split(df, test_size=0.169, random_state=0, stratify=df['formtype'])  #
    test_s=.169 gave 50%
415.    # df.to_csv(os.path.join(MASTER_DIR, 'full_modified_dataset.csv'))
416.
417.    X_train = train.iloc[:, 3:-1]
418.    # X_train_names = train.iloc[:, 0:3]
419.    y_train = train.iloc[:, -1]
420.    print("Train shape:", X_train.shape)
421.    X_test = test.iloc[:, 3:-1]
422.    # X_test_names = test.iloc[:, 0:3]
423.    y_test = test.iloc[:, -1]
424.    print("Test shape:", X_test.shape)
425.
```

```
426.    # Normalize Data
427.    """
428.    min_max_scaler = preprocessing.MinMaxScaler()
429.    X_train = min_max_scaler.fit_transform(X_train)  # Good for decision tree
430.    X_test = min_max_scaler.fit_transform(X_test)
431.    """
432.    # X_train = preprocessing.scale(X_train)
433.    # X_test = preprocessing.scale(X_test)
434.    # """
435.    mean = np.mean(X_train, axis=0)
436.    std = np.std(X_train, axis=0)
437.    X_train = (X_train - mean) / std  # Good for decision tree
438.    X_test = (X_test - mean) / std
439.    # """
440.    print("Normalized Train shape:", X_train.shape)
441.    print("Normalized Test shape:", X_test.shape)
442.
443.    # Convert to arrays for keras
444.    X_train = np.array(X_train)
445.    y_train = np.array(y_train)
446.    X_test = np.array(X_test)
447.    y_test = np.array(y_test)
448.
449.    label_encoder = LabelEncoder()
450.    old_y_train = y_train
451.    # old_y_test = y_test
452.    int_y_train = label_encoder.fit_transform(y_train)
453.    print(int_y_train.shape)
454.    # int_y_train = int_y_train.reshape(len(int_y_train), 1)
455.    int_y_test = label_encoder.fit_transform(y_test)
456.    # int_y_test = int_y_test.reshape(len(int_y_test), 1)
457.    y_train = to_categorical(label_encoder.fit_transform(y_train))
458.    y_test = to_categorical(label_encoder.fit_transform(y_test))
459.    print(y_train.shape, y_test.shape)
460.    print(label_encoder.classes_, "\n")
461.
462.    """ BASE MODEL """
463.    # DummyClassifier makes predictions while ignoring input features
464.    dummy_clf = DummyClassifier(strategy="stratified")
465.    dummy_clf.fit(X_train, y_train)
466.    DummyClassifier(strategy='stratified')
467.    dummy_clf.predict(X_test)
468.    print("Dummy classifier accuracy:", dummy_clf.score(X_test, y_test))
469.
470.    clf = tree.DecisionTreeClassifier()
471.    clf = clf.fit(X_train, y_train)
472.    clf.predict(X_test)
473.    print("Decision tree accuracy:", clf.score(X_test, y_test))
474.
475.    """ FEATURE TUNING """
476.    selector = SelectKBest(f_classif, k=15)  # 1000 if using RFE
477.    Z_train = selector.fit_transform(X_train, old_y_train)
478.    skb_values = selector.get_support()
479.    Z_test = X_test[:, skb_values]
480.    np.save(os.path.join(WEIGHT_DIR, "selectkbest_indices.npy"), skb_values)
481.    print(Z_train.shape)
482.    print(Z_test.shape)
483.    """
484.    plt.title('Feature Importance')
485.    plt.ylabel('Score')
486.    plt.xlabel('Feature')
487.    plt.plot(selector.scores_)
488.    plt.savefig('Initial_Feature_Importance.png')
489.    plt.show()
490.    """
491.    print("Indices of top 10 features:", (-selector.scores_).argsort()[:10])
492.
493.    """ KBEST MODEL """
494.    clf = tree.DecisionTreeClassifier()
```

```
495.    clf = clf.fit(Z_train, y_train)
496.    clf.predict(Z_test)
497.    # treedepth = clf.tree_.max_depth
498.    skb_score = clf.score(Z_test, y_test)
499.    print("K-Best Decision tree accuracy:", skb_score)  # Highest score: 84.3% accuracy
500.
501.    # """
502.    # Accuracy 0.211, stick with SKB? Gives good loss though
503.    clf = LinearSVC(C=0.01, penalty="l1", dual=False)
504.    clf.fit(X_train, old_y_train)
505.    rfe_selector = RFE(clf, 15, verbose=5)
506.    rfe_selector = rfe_selector.fit(Z_train, old_y_train)
507.    # rfe_selector = rfe_selector.fit(X_train, old_y_train)
508.    rfe_values = rfe_selector.get_support()
509.    # np.save(os.path.join(MASTER_DIR, "rfebest_indices.npy"), rfe_values)
510.    print("Indices of RFE important features:", np.where(rfe_values)[0])
511.    W_train = Z_train[:, rfe_values]
512.    W_test = Z_test[:, rfe_values]
513.
514.    # "" " RFE MODEL " ""
515.    clf = tree.DecisionTreeClassifier()
516.    clf = clf.fit(W_train, y_train)
517.    clf.predict(W_test)
518.    rfe_score = clf.score(W_test, y_test)
519.    print("RFE Decision tree accuracy:", rfe_score)  # Highest score: 83.7% accuracy, typically
    better than SKB
520.    """
521.    plt.figure(figsize=(30, 20))  # set plot size (denoted in inches)
522.    tree.plot_tree(clf, fontsize=10)
523.    plt.show()
524.    plt.savefig('tree_high_dpi', dpi=100)
525.    """
526.    # """
527.    # endregion
528.
529.    if skb_score > rfe_score:
530.        X_train = Z_train
531.        X_test = Z_test
532.    else:
533.        X_train = W_train
534.        X_test = W_test
535.        # treedepth = clf.tree_.max_depth
536.
537.    # TreeGrad Deep Neural Decision Forest - 83% accuracy
538.    model = TGDClassifier(num_leaves=31, max_depth=-1, learning_rate=0.1, n_estimators=100,
539.                          autograd_config={'refit_splits': True})
540.    model.fit(X_train, int_y_train)
541.    acc = accuracy_score(int_y_test, model.predict(X_test))
542.    print('TreeGrad Deep Neural Decision Forest accuracy: ', acc)
543.
544.    print('Plotting 0th tree...')  # one tree use categorical feature to split
545.    lgb.plot_tree(model.base_model_, tree_index=0, figsize=(15, 15), show_info=['split_gain'])
546.    plt.savefig('TreeGrad_Model.png')
547.    plt.show()
548.    print('Plotting feature importances...')
549.    lgb.plot_importance(model.base_model_, max_num_features=15)
550.    plt.savefig('TreeGrad_Feature_Importance.png')
551.    plt.show()
552.
553.    predictions = model.predict(X_test)
554.    # predictions = predictions.argmax(axis=1)
555.    predictions = predictions.astype(int).flatten()
556.    predictions = (label_encoder.inverse_transform(predictions))
557.    predictions = pd.DataFrame({'Predicted Values': predictions})
558.
559.    # actual = y_test.argmax(axis=1)
560.    actual = int_y_test.astype(int).flatten()
561.    actual = (label_encoder.inverse_transform(actual))
562.    actual = pd.DataFrame({'Actual Values': actual})
```

```
563.
564.    cm = confusion_matrix(actual, predictions)
565.    plt.figure(figsize=(12, 10))
566.    cm = pd.DataFrame(cm, index=[i for i in label_encoder.classes_], columns=[i for i in
    label_encoder.classes_])
567.    ax = sns.heatmap(cm, linecolor='white', cmap='Blues', linewidth=1, annot=True, fmt='')
568.    bottom, top = ax.get_ylim()
569.    ax.set_ylim(bottom + 0.5, top - 0.5)
570.    plt.title('Confusion Matrix', size=20)
571.    plt.xlabel('Predicted Labels', size=14)
572.    plt.ylabel('Actual Labels', size=14)
573.    plt.savefig('TreeGrad_Confusion_Matrix.png')
574.    plt.show()
575.    clf_report = classification_report(actual, predictions, output_dict=True,
576.                                       target_names=[i for i in label_encoder.classes_])
577.    sns.heatmap(pd.DataFrame(clf_report).iloc[:, :].T, annot=True, cmap='viridis')
578.    plt.title('Classification Report', size=20)
579.    plt.savefig('TreeGrad_Classification_Report.png')
580.    plt.show()
581.
582.    with open(os.path.join(WEIGHT_DIR, 'treegrad_model_save.pkl'), 'wb') as f:
583.        pickle.dump(model, f)
584.    with open(os.path.join(WEIGHT_DIR, 'treegrad_model_save.pkl'), 'rb') as f:
585.        model2 = pickle.load(f)
586.    acc = accuracy_score(int_y_test, model2.predict(X_test))
587.    print('TreeGrad Deep Neural Decision Forest accuracy from save: ', acc)
588.    pass
589.
590.
591.def preparePredictionData(filepath, savetoexcel=False, verbose=True):
592.    if verbose:
593.        print("Preparing MLS")
594.    mls = dus.util_main_helper(feature="mls", filepath=filepath, predict=True)
595.
596.    sngdur = 0
597.    with audioread.audio_open(filepath) as f:
598.        sngdur += f.duration
599.
600.    np.set_string_function(
601.        lambda x: repr(x).replace('(', '').replace(')', '').replace('array', '').replace("       ", '
    '), repr=False)
602.    np.set_printoptions(threshold=inf)
603.
604.    if verbose:
605.        print("Building feature table")
606.    df = pd.DataFrame(columns=['piece_name', 'composer', 'filename', 'duration', 'ssm_log_mel_mean',
    'formtype'])
607.    c_flname = os.path.basename(filepath.split('/')[-1].split('.')[0])
608.
609.    df.loc[0] = ["TBD", "TBD", c_flname, sngdur, mls[0], "TBD"]
610.    for col in df.columns:
611.        df[col] = df[col].apply(lambda x: str(x)
612.                                .replace(", dtype=float32", "").replace("],", "]")
613.                                .replace("dtype=float32", "").replace("...,", ""))
614.    if savetoexcel:
615.        df.to_excel(os.path.join(MASTER_DIR, c_flname + '.xlsx'), index=False)
616.    return df
617.
618.
619.def predictForm(midpath=None, verbose=True):
620.    if midpath is None:
621.        midpath = input("Enter path to folder or audio file: ")
622.    df = pd.DataFrame()
623.    if not os.path.exists(midpath):
624.        raise FileNotFoundError("Path not found or does not exist.")
625.    else:
626.        if os.path.isfile(midpath):
627.            # df2 = pd.read_excel(os.path.join(MASTER_DIR, 'brahms_opus117_1.xlsx'))
628.            df = preparePredictionData(midpath, savetoexcel=False, verbose=verbose)
```

```
629.        elif os.path.isdir(midpath):
630.            if midpath[-1] != "\\" or midpath[-1] != "/":
631.                if "\\" in midpath:
632.                    midpath = midpath + "\\"
633.                else:
634.                    midpath = midpath + "/"
635.            cnt = 0
636.            audio_extensions = ["3gp", "aa", "aac", "aax", "act", "aiff", "alac", "amr", "ape", "au",
    "awb", "dct",
637.                                "dss", "dvf", "flac", "gsm", "iklax", "ivs", "m4a", "m4b", "m4p",
    "mmf", "mp3", "mpc",
638.                                "msv", "nmf", "ogg", "oga", "mogg", "opus", "ra", "rm", "raw",
    "rf64", "sln", "tta",
639.                                "voc", "vox", "wav", "wma", "wv", "webm", "8svx", "cda", "mid",
    "midi", "MID", "mp4"]
640.            for (mid_dirpath, mid_dirnames, mid_filenames) in os.walk(midpath):
641.                for f in mid_filenames:
642.                    if f.endswith(tuple(audio_extensions)):
643.                        if verbose:
644.                            print("Reading file #" + str(cnt + 1))
645.                        mid_path = mid_dirpath + f
646.                        dft = preparePredictionData(mid_path, savetoexcel=False, verbose=verbose)
647.                        df = pd.concat([df, dft], ignore_index=True).reset_index(drop=True)
648.                        cnt += 1
649.        else:
650.            raise FileNotFoundError("Path resulted in error.")
651.
652.    names = df[['piece_name', 'composer', 'filename']]
653.    y = df['formtype']
654.    nonlist = df[['duration']]
655.    df.drop(columns=['piece_name', 'composer', 'filename', 'duration', 'formtype'], inplace=True)
656.    df = df[['ssm_log_mel_mean']]
657.    if verbose:
658.        print("Fixing broken array cells as needed...")
659.
660.    def fix_broken_arr(strx):
661.        if '[' in strx:
662.            if ']' in strx:
663.                return strx
664.            else:
665.                return strx + ']'
666.
667.    for col in df.columns:
668.        df[col] = df[col].apply(lambda x: fix_broken_arr(x))
669.    if verbose:
670.        print("Done processing cells, building data set...")
671.
672.    d = [pd.DataFrame(df[col].astype(str).apply(literal_eval).values.tolist()) for col in df.columns]
673.    df = pd.concat(d, axis=1).fillna(0)
674.    df = pd.concat([pd.concat([names, pd.concat([nonlist, df], axis=1)], axis=1), y], axis=1)
675.    df['duration'] = df['duration'].apply(lambda x: float(x))
676.
677.    X_test = df.iloc[:, 3:-1]  # Match 2687 from training data
678.    if X_test.shape[1] <= 2687:
679.        t_cnt = 0
680.        while X_test.shape[1] < 2687:
681.            X_test = X_test.join(pd.DataFrame(columns=['filler' + str(t_cnt)]), how='outer')
682.            t_cnt += 1
683.    else:
684.        X_test = X_test.iloc[:, :2687]
685.    X_test = X_test.fillna(0)
686.    # print(X_test.shape)
687.    X_test_names = df.iloc[:, 0:3]
688.    # y_test = df.iloc[:, -1]
689.
690.    # Normalize Data
691.    mean = np.mean(X_test, axis=0)
692.    std = np.std(X_test, axis=0)
693.    X_test = (X_test - mean) / std
```

```
694.    # print("Normalized Test shape:", X_test.shape)
695.    X_test = np.array(X_test)
696.    X_test_names = np.array(X_test_names)
697.
698.    label_encoder = LabelEncoder()
699.    label_encoder.classes_ = np.load(os.path.join(WEIGHT_DIR, 'form_classes.npy'))
700.    skb_values = np.load(os.path.join(WEIGHT_DIR, "selectkbest_indices.npy"))
701.    # kbest_indices = np.argwhere(skb_values == True)
702.    X_test = X_test[:, skb_values]
703.
704.    with open(os.path.join(WEIGHT_DIR, 'treegrad_model_save_best.pkl'), 'rb') as f:
705.        model = pickle.load(f)
706.    result = model.predict(X_test)
707.
708.    results = []
709.    names = []
710.    for i in range(X_test.shape[0]):
711.        resultlbl = label_encoder.inverse_transform([result[i]])
712.        if verbose:
713.            print("Performing predictions on", X_test_names[i][2])
714.            print("\t\tPredicted form:", resultlbl[0])
715.        results.append([resultlbl[0]])
716.        names.append([X_test_names[i][2]])
717.    return results, names
718.
719.
720.# endregion
721.
722.
723."""================================================================================================
    ================"""
724.
725.
726.# region LabelModel
727.def get_split_spectrograms(valid_only=True):
728.    tr_in = du.ReadLabelSecondsPhrasesFromFolder(FULL_LABELPATH, valid_only=valid_only,
    get_names=True)
729.    tr_xy = tr_in[0:2]
730.    tr_names = tr_in[3]
731.    tr_set = np.array(pd.DataFrame(tr_xy).transpose())
732.    all_spectrograms = []
733.    all_durations = []
734.
735.    def find(name, path):
736.        for root, dirs, files in os.walk(path):
737.            for file in files:
738.                if name in file:
739.                    return os.path.join(root, file)
740.
741.    for i in range(tr_set.shape[0]):
742.        Xt = tr_set[i][0]  # Timestamps
743.        # yt = tr_set[i][1]  # Labels
744.        fname = find(tr_names[i][0:-4], FULL_MIDI_DIR)
745.        print("Reading file: " + fname)
746.        audio_splits = []
747.        durations = []
748.        splity = du.SplitAudio("", fname)
749.        sr = splity.get_samplerate()
750.        for idx, timestamp in enumerate(Xt):
751.            if idx != len(Xt) - 1:
752.                asplit = splity.single_split(Xt[idx], Xt[idx + 1], export=False)
753.                asplit = du.audiosegment_to_ndarray(asplit)
754.                if len(asplit) == 0:
755.                    print("i1 =", Xt[idx], "\ti2 =", Xt[idx + 1])
756.                    print("Ndarr:", len(asplit))
757.                    print("Duration:", splity.get_duration(), "\tExpected:", Xt[-1])
758.                audio_splits.append(asplit)
759.                durations.append(abs(float(Xt[idx + 1] - Xt[idx])))
760.        spectrograms = []
```

```
761.        for asplit in audio_splits:
762.            mel_spec = librosa.feature.melspectrogram(y=asplit, sr=sr)
763.            # mel_spec = librosa.feature.mfcc(y=asplit, sr=sr,
764.            #                                 n_mfcc=20, dct_type=2, norm='ortho', lifter=0,
    hop_length=4096, n_fft=819)
765.            spectrograms.append(np.mean(mel_spec, axis=0))
766.            """
767.            plt.figure(figsize=(10, 4))
768.            librosa.display.specshow(mel_spec, cmap='plasma', x_axis='time')
769.            plt.colorbar()
770.            plt.ylabel('Frequency bands')
771.            plt.title('Mel Spectrogram')
772.            plt.tight_layout()
773.            plt.show()
774.            """
775.        all_spectrograms.append(spectrograms)
776.        all_durations.append(durations)
777.        print("Finished file #" + str(i + 1))
778.    np.save(os.path.join(FULL_DIR, "split_spectrograms.npy"), all_spectrograms)
779.    np.save(os.path.join(FULL_DIR, "split_durations.npy"), all_durations)
780.    pass
781.
782.
783.def get_label_dataset(valid_only=True):
784.    if not os.path.exists(os.path.join(FULL_DIR, "split_spectrograms.npy")) or not \
785.            os.path.exists(os.path.join(FULL_DIR, "split_durations.npy")):
786.        get_split_spectrograms(valid_only)
787.    all_spectrograms = np.load(os.path.join(FULL_DIR, "split_spectrograms.npy"), allow_pickle=True)
788.    all_durations = np.load(os.path.join(FULL_DIR, "split_durations.npy"), allow_pickle=True)
789.    tr_in = du.ReadLabelSecondsPhrasesFromFolder(FULL_LABELPATH, valid_only=valid_only,
    get_names=True, get_forms=True)
790.    tr_xy = tr_in[0:2]
791.    # tr_names = tr_in[3]
792.    tr_forms = tr_in[2]
793.    tr_set = np.array(pd.DataFrame(tr_xy).transpose())
794.
795.    def cleanlabel(xlbl):
796.        return ''.join([ix for ix in xlbl if not ix.isdigit()]).replace("Codetta",
    "codetta").replace("retran", "tran")\
797.            .replace("'", "").replace("True", "").replace("true", "").replace("Bri",
    "bri").replace("FugueExp", "Exp")\
798.            .replace("domProlongation", "transition").replace("h", "c").replace("Aev", "Dev")\
799.            .replace("Tcem", "Them").replace("COAA", "CODA").replace("Trio", "B")\
800.            .replace("bridge", "transition").replace("Entry", "A").replace("Episode", "B")\
801.            .replace("Exposition", "A").replace("Development", "B").replace("Recapitulation", "A")\
802.            .replace("pt", "a").replace("st", "b").replace("ct", "c").replace("var", "sec")\
803.            .replace("ccaraceribic", "characteristic").replace("Middle", "").replace("Final",
    "").replace("part", "sec")
804.        # .replace("Silence", "").replace("End", "").replace("part", "").replace("CODA",
    "ct").replace("D", "A")
805.        # These should be removed when a better architecture becomes possible and/or dataset
    increases
806.
807.    label_encoder = LabelEncoder()
808.    alllabels = set()
809.    for i in range(tr_set.shape[0]):
810.        yset = tr_set[i][1]
811.        for inneryset in yset:
812.            for iny in inneryset:
813.                tlbl = cleanlabel(iny)
814.                if (tlbl == "i" or tlbl == "g") and len(tlbl) == 1:  # To remove
815.                    tlbl = "e" if tlbl == "i" else "d"
816.                alllabels.add(tlbl)
817.    alllabels = np.array(list(filter(None, sorted(alllabels))))
818.    print(alllabels)
819.    label_encoder.fit_transform(alllabels)
820.    df = pd.DataFrame(columns=['form', 'timestamps', 'durations', 'mel_splits', 'labels'])
821.    mel_splits = []
822.    for i in range(tr_set.shape[0]):
```

```
        Xt = tr_set[i][0][:-1]  # Timestamps
        Ft = tr_forms[i]  # Form
        Ft = list(itertools.chain.from_iterable(itertools.repeat(x, len(Xt)) for x in Ft))
        Zt = all_spectrograms[i]
        for inarr in Zt:
            mel_splits.append(inarr)
        Wt = np.abs(all_durations[i])
        yt = tr_set[i][1][:-1]  # Labels
        for j in range(len(yt)):
            for lbl in range(len(yt[j])):
                tlbl = cleanlabel(yt[j][lbl])
                if (tlbl == "i" or tlbl == "g") and len(tlbl) == 1:  # To remove
                    tlbl = "e" if tlbl == "i" else "d"
                yt[j][lbl] = tlbl
            yt[j] = label_encoder.transform(np.array(list(set(yt[j]))))
        if len(Wt) != len(Xt) != len(yt) != len(Zt):
            print("Error in dataset sizes")
        df.loc[i] = [Ft, Xt, Wt, Zt, yt]

    timestamps = np.hstack(df['timestamps'])
    durations = np.hstack(df['durations'])
    forms = np.hstack(df['form'])
    labels = []
    for i in range(df.shape[0]):
        for inarr in df['labels'].iloc[i]:
            labels.append(np.array(inarr, dtype=np.int))
    labels = np.array(labels)
    mel_splits = np.array(mel_splits)
    dfmid = pd.DataFrame(columns=['mel_splits'])
    dfleft = pd.DataFrame(columns=['form', 'timestamps', 'durations'])
    dfright = pd.DataFrame(columns=['labels'])
    if len(labels) != len(durations) != len(timestamps) != len(mel_splits):
        print("Error in dataset sizes")
    for i in range(len(labels)):
        dfmid.loc[i] = [mel_splits[i]]
        dfleft.loc[i] = [forms[i], timestamps[i], durations[i]]
        dfright.loc[i] = [labels[i]]

    np.set_string_function(
        lambda x: repr(x).replace('(', '').replace(')', '').replace('array', '').replace("       ", '
').replace(
            ", dtype=float32", "").replace("dtype=float32", ""), repr=False)
    np.set_printoptions(threshold=inf)
    df2 = pd.DataFrame(dfmid['mel_splits'].values.tolist())
    df2 = df2.fillna(0)
    df2 = pd.concat([pd.concat([dfleft, df2], axis=1), dfright], axis=1)
    return df2, label_encoder


def formnn_label_lstm(mlb):
    model = Sequential()
    model.add(layers.Bidirectional(    # Try recurrent_dropout=0.4 or dropout=0.2,
    recurrent_dropout=0.2
        layers.LSTM(4, dropout=0.2, return_sequences=True), input_shape=(None, 1),
    merge_mode='concat'))
    model.add(layers.TimeDistributed(
        layers.Dense(len(mlb.classes_), activation='sigmoid')))
    model.compile(loss='binary_crossentropy', optimizer='adam', metrics=['accuracy'])
    return model


def trainLabelModel(retrain=True):
    """
    RNN Model should take in timestamp array (X) and labels (y) for training, eval only provide
    novelty timestamps
    - Convert audio into log-mel spectrogram
    - Break audio into segments between timestamps (mel arrays)
    - Pass RNN two inputs: {[timestamp1_audio], [timestamp2_audio], ...}, [timestamp1, timestamp2]
```

```
    - Classify using simple labels only? A, B, a, b, c instead of A A' B, a, a', a'', Expos, PT, ST, etc.
        o Have to parse labels specifically for training because of this
    - If necessary, only label phrases, then phrases and parts together
    - Return label(s) for each timestamp
    - Dataset very small -- skip out on non-validation data for now?
    """

    df, label_encoder = get_label_dataset(valid_only=True)
    np.save(os.path.join(WEIGHT_DIR, 'label_classes.npy'), label_encoder.classes_)
    # Maybe drop 'form' column unless it decreases accuracy? Seems to work better with it though
    X_train = df.iloc[:, :-1]
    y_train = df.iloc[:, -1]
    print("Train shape:", X_train.shape)

    mean = np.mean(X_train, axis=0)
    std = np.std(X_train, axis=0)
    X_train = (X_train - mean) / std
    print("Normalized Train shape:", X_train.shape)

    X_train = np.array(X_train)
    int_y_train = np.array(y_train)
    mlb = MultiLabelBinarizer()
    # y_train = to_categorical(int_y_train)
    y_train = mlb.fit_transform(int_y_train)
    np.save(os.path.join(WEIGHT_DIR, 'label_mlb_classes.npy'), mlb.classes_)

    """ BASE MODEL """
    dummy_clf = DummyClassifier(strategy="stratified")
    dummy_clf.fit(X_train, y_train)
    DummyClassifier(strategy='stratified')
    dummy_clf.predict_proba(X_train)
    print("Dummy classifier accuracy:", dummy_clf.score(X_train, y_train))

    clf = tree.DecisionTreeClassifier()
    clf = clf.fit(X_train, y_train)
    clf.predict_proba(X_train)
    print("Decision tree accuracy:", clf.score(X_train, y_train))
    with open(os.path.join(WEIGHT_DIR, 'dcntree_phrase_model_save.pkl'), 'wb') as f:
        pickle.dump(clf, f)

    """
    # Fails predicting due to class imbalance
    clf = RandomForestClassifier()
    clf.fit(X_train, y_train)
    clf.predict_proba(X_train)
    print("Random Forest accuracy:", clf.score(X_train, y_train))
    with open('rndforest_phrase_model_save.pkl', 'wb') as f:
        pickle.dump(clf, f)
    """

    model = formnn_label_lstm(mlb)
    model.summary()
    if not os.path.isfile(os.path.join(MASTER_DIR, 'FormNN_LSTM_Model_Diagram.png')):
        plot_model(model, to_file=os.path.join(MASTER_DIR, 'FormNN_LSTM_Model_Diagram.png'),
                   show_shapes=True, show_layer_names=True, expand_nested=True, dpi=300)
    checkpoint = ModelCheckpoint(os.path.join(WEIGHT_DIR, "best_label_model.hdf5"), monitor='accuracy', verbose=0,
                                 save_best_only=False, mode='max', save_freq='epoch', save_weights_only=True)
    X_train = X_train[:, :, np.newaxis]
    if retrain:
        model.fit(X_train, y_train, epochs=5, batch_size=1, callbacks=[checkpoint])
    else:
        if not os.path.isfile(os.path.join(WEIGHT_DIR, 'best_label_model.hdf5')):
            model.fit(X_train, y_train, epochs=10, batch_size=1, callbacks=[checkpoint])
        model.load_weights(os.path.join(WEIGHT_DIR, 'best_label_model.hdf5'))
    feature_vectors_model = keras.models.Model(model.input, model.get_layer('time_distributed').output)
```

```
952.    X_ext = feature_vectors_model.predict(X_train)[:, :, 0]
953.
954.    dtc = tree.DecisionTreeClassifier()
955.    dtc.fit(X_ext, y_train)
956.    with open(os.path.join(WEIGHT_DIR, 'lstmtree_phrase_model_save.pkl'), 'wb') as f:
957.        pickle.dump(dtc, f)
958.    dtc_y_pred = dtc.predict(X_ext)
959.    dtc_score = dtc.score(X_ext, y_train)
960.    print("Deep LSTM Decision Tree accuracy:", dtc_score)
961.
962.    if not os.path.isfile(os.path.join(MASTER_DIR, 'LSTM_Tree.png')):
963.        plt.figure(figsize=(30, 30))
964.        tree.plot_tree(dtc, fontsize=10)
965.        plt.savefig('LSTM_Tree.png', dpi=100)
966.        plt.show()
967.
968.    def hamming_score(y_true, y_pred):
969.        acc_list = []
970.        for i in range(y_true.shape[0]):
971.            set_true = set(np.where(y_true[i])[0])
972.            set_pred = set(np.where(y_pred[i])[0])
973.            if len(set_true) == 0 and len(set_pred) == 0:
974.                tmp_a = 1
975.            else:
976.                tmp_a = len(set_true.intersection(set_pred)) / float(len(set_true.union(set_pred)))
977.            acc_list.append(tmp_a)
978.        return np.mean(acc_list)
979.
980.    print("Deep LSTM Decision Tree hamming score:", hamming_score(y_train, dtc_y_pred))
981.
982.    preds = (mlb.inverse_transform(dtc_y_pred))
983.    predictions = [label_encoder.inverse_transform(inarr) for inarr in preds]
984.    actual = [label_encoder.inverse_transform(inarr) for inarr in int_y_train]
985.    print(predictions)
986.    print(actual)
987.
988.    """
989.    # Too slow, poor accuracy (0.03% hamming score)
990.    X_train = X_train[:, :, np.newaxis]
991.    model = formnn_lstm(None, mode='concat', num_classes=len(mlb.classes_))
992.    history_loss = []
993.    history_accuracy = []
994.    for i in range(10):
995.        checkpoint = ModelCheckpoint("best_form_lstm_model.hdf5", monitor='accuracy', verbose=0,
996.                                     save_best_only=False, mode='max', save_freq='epoch',
    save_weights_only=True)
997.        model_history = model.fit(X_train, y_train, epochs=1, batch_size=1, verbose=1,
    callbacks=[checkpoint])
998.        history_loss.append(model_history.history['loss'])
999.        history_accuracy.append(model_history.history['acc'])
1000.        """
1001.
1002.        oneclass_int_y_train = []
1003.        for inarr in int_y_train:
1004.            oneclass_int_y_train.append(inarr[0])
1005.        X_train = X_train[:, :, 0]
1006.        # Only takes input for single-label classification, great accuracy though
1007.        model = TGDClassifier(num_leaves=31, max_depth=-1, learning_rate=0.01, n_estimators=100,
1008.                              autograd_config={'refit_splits': True})
1009.        model.fit(X_train, oneclass_int_y_train)
1010.        pred = model.predict(X_train)
1011.        acc = accuracy_score(oneclass_int_y_train, pred)
1012.        print('TreeGrad Deep Neural Decision Forest accuracy: ', acc)
1013.
1014.        print('Plotting 0th tree...')  # one tree use categorical feature to split
1015.        lgb.plot_tree(model.base_model_, tree_index=0, figsize=(15, 15), show_info=['split_gain'])
1016.        plt.savefig('TreeGrad_Phrase_Model.png')
1017.        plt.show()
1018.        print('Plotting feature importances...')
```

```
1019.         lgb.plot_importance(model.base_model_, max_num_features=15)
1020.         plt.savefig('TreeGrad_Phrase_Feature_Importance.png')
1021.         plt.show()
1022.
1023.         with open(os.path.join(WEIGHT_DIR, 'treegrad_phrase_model_save.pkl'), 'wb') as f:
1024.             pickle.dump(model, f)
1025.
1026.         """
1027.         predictions = dtc_y_pred
1028.         predictions = (mlb.inverse_transform(predictions))
1029.         predictions = (label_encoder.inverse_transform(predictions))
1030.         predictions = pd.DataFrame({'Predicted Values': predictions})
1031.
1032.         actual = (label_encoder.inverse_transform(int_y_train))
1033.         actual = pd.DataFrame({'Actual Values': actual})
1034.
1035.         cm = multilabel_confusion_matrix(actual, predictions)
1036.         """
1037.         pass
1038.
1039.
1040.     def predictLabels(midpath=None, verbose=True, printform=False, printresults=True):
1041.         if midpath is None:
1042.             midpath = input("Enter path to folder or audio file: ")
1043.         all_peaks = []
1044.         filenames = []
1045.         if not os.path.exists(midpath):
1046.             raise FileNotFoundError("Path not found or does not exist.")
1047.         else:
1048.             if os.path.isfile(midpath):
1049.                 peaks = du.peak_picking(midpath, returnpeaks=True, verbose=False)
1050.                 if len(peaks) == 2:
1051.                     peaks = np.insert(peaks, 1, peaks[-1] / 2)
1052.                 all_peaks.append(peaks)
1053.                 filenames.append(midpath)
1054.             elif os.path.isdir(midpath):
1055.                 if midpath[-1] != "\\" or midpath[-1] != "/":
1056.                     if "\\" in midpath:
1057.                         midpath = midpath + "\\"
1058.                     else:
1059.                         midpath = midpath + "/"
1060.                 cnt = 0
1061.                 audio_extensions = ["3gp", "aa", "aac", "aax", "act", "aiff", "alac", "amr", "ape",
    "au", "awb", "dct",
1062.                                     "dss", "dvf", "flac", "gsm", "iklax", "ivs", "m4a", "m4b",
    "m4p", "mmf", "mp3", "mpc",
1063.                                     "msv", "nmf", "ogg", "oga", "mogg", "opus", "ra", "rm", "raw",
    "rf64", "sln", "tta",
1064.                                     "voc", "vox", "wav", "wma", "wv", "webm", "8svx", "cda", "mid",
    "midi", "MID", "mp4"]
1065.                 for (mid_dirpath, mid_dirnames, mid_filenames) in os.walk(midpath):
1066.                     for f in mid_filenames:
1067.                         if f.endswith(tuple(audio_extensions)):
1068.                             if verbose:
1069.                                 print("Reading file #" + str(cnt + 1))
1070.                             mid_path = mid_dirpath + f
1071.                             peaks = du.peak_picking(mid_path, returnpeaks=True, verbose=False)
1072.                             if len(peaks) == 2:
1073.                                 peaks = np.insert(peaks, 1, peaks[-1] / 2)
1074.                             all_peaks.append(peaks)
1075.                             filenames.append(mid_path)
1076.                             cnt += 1
1077.             else:
1078.                 raise FileNotFoundError("Path resulted in error.")
1079.
1080.         formsin, namesin = predictForm(midpath, verbose=False)
1081.         # print(all_peaks, "\n", formsin, "\n", namesin)
1082.         if verbose:
1083.             print("Done predicting forms, processing file data...")
```

```
1084.
1085.         all_spectrograms = []
1086.         all_durations = []
1087.         for i in range(len(filenames)):
1088.             Xt = all_peaks[i]
1089.             audio_splits = []
1090.             durations = []
1091.             splity = du.SplitAudio("", filenames[i])
1092.             sr = splity.get_samplerate()
1093.             for idx, timestamp in enumerate(Xt):
1094.                 if idx != len(Xt) - 1:
1095.                     asplit = splity.single_split(Xt[idx], Xt[idx + 1], export=False)
1096.                     asplit = du.audiosegment_to_ndarray(asplit)
1097.                     if len(asplit) == 0:
1098.                         print("i1 =", Xt[idx], "\ti2 =", Xt[idx + 1])
1099.                         print("Ndarr:", len(asplit))
1100.                         print("Duration:", splity.get_duration(), "\tExpected:", Xt[-1])
1101.                     audio_splits.append(asplit)
1102.                     durations.append(abs(float(Xt[idx + 1] - Xt[idx])))
1103.             spectrograms = []
1104.             for asplit in audio_splits:
1105.                 mel_spec = librosa.feature.mfcc(y=asplit, sr=sr,
1106.                                                 n_mfcc=20, dct_type=2, norm='ortho', lifter=0,
    hop_length=4096, n_fft=819)
1107.                 spectrograms.append(np.mean(mel_spec, axis=0))
1108.             all_spectrograms.append(spectrograms)
1109.             all_durations.append(durations)
1110.             if verbose:
1111.                 print("Finished file #" + str(i + 1))
1112.
1113.         label_encoder_form = LabelEncoder()
1114.         label_encoder_form.classes_ = np.load(os.path.join(WEIGHT_DIR, 'form_classes.npy'))
1115.         df = pd.DataFrame(columns=['form', 'timestamps', 'durations', 'mel_splits'])
1116.         mel_splits = []
1117.         for i in range(len(filenames)):
1118.             Xt = all_peaks[i][:-1]  # Timestamps
1119.             Ft = formsin[i]  # Form
1120.             Ft = label_encoder_form.transform(Ft)
1121.             Ft = list(itertools.chain.from_iterable(itertools.repeat(x, len(Xt)) for x in Ft))
1122.             Zt = all_spectrograms[i]
1123.             for inarr in Zt:
1124.                 mel_splits.append(inarr)
1125.             Wt = np.abs(all_durations[i])
1126.             if len(Wt) != len(Xt) != len(Zt):
1127.                 print("Error in dataset sizes - Wt:", len(Wt), " Xt:", len(Xt), " Zt:", len(Zt))
1128.             df.loc[i] = [Ft, Xt, Wt, Zt]
1129.
1130.         timestamps = np.hstack(df['timestamps'])
1131.         durations = np.hstack(df['durations'])
1132.         forms = np.hstack(df['form'])
1133.         mel_splits = np.array(mel_splits)
1134.         dfmid = pd.DataFrame(columns=['mel_splits'])
1135.         dfleft = pd.DataFrame(columns=['form', 'timestamps', 'durations'])
1136.         if len(durations) != len(timestamps) != len(mel_splits) != len(forms):
1137.             print("Error in dataset sizes - durations:",
1138.                   len(durations), " timetamps:", len(timestamps), " mel_splits:", len(mel_splits),
    " forms:", len(forms))
1139.         for i in range(len(timestamps)):
1140.             dfmid.loc[i] = [mel_splits[i]]
1141.             dfleft.loc[i] = [forms[i], timestamps[i], durations[i]]
1142.
1143.         np.set_string_function(
1144.             lambda x: repr(x).replace('(', '').replace(')', '').replace('array', '').replace("
    ", ' ').replace(
1145.                 ", dtype=float32", "").replace("dtype=float32", ""), repr=False)
1146.         np.set_printoptions(threshold=inf)
1147.         df2 = pd.DataFrame(dfmid['mel_splits'].values.tolist())
1148.         df2 = df2.fillna(0)
1149.         df2 = pd.concat([dfleft, df2], axis=1)
```

```
1150.         df2 = df2.fillna(0)
1151.
1152.         if verbose:
1153.             print("Done processing files, building data set...\n")
1154.         label_encoder = LabelEncoder()
1155.         label_encoder.classes_ = np.load(os.path.join(WEIGHT_DIR, 'label_classes.npy'))
1156.         X_test = df2
1157.         if X_test.shape[1] <= 1341:  # Match shape to training data
1158.             t_cnt = 0
1159.             while X_test.shape[1] < 1341:
1160.                 X_test = X_test.join(pd.DataFrame(columns=['filler' + str(t_cnt)]), how='outer')
1161.                 t_cnt += 1
1162.         else:
1163.             X_test = X_test.iloc[:, :1341]
1164.         X_test = X_test.fillna(0)
1165.
1166.         min_max_scaler = preprocessing.MinMaxScaler()
1167.         X_test = min_max_scaler.fit_transform(X_test)
1168.         X_test = np.array(X_test)
1169.
1170.         mlb = MultiLabelBinarizer()
1171.         mlb.classes_ = np.load(os.path.join(WEIGHT_DIR, 'label_mlb_classes.npy'),
    allow_pickle=True)
1172.         model = formnn_label_lstm(mlb)
1173.         X_test = X_test[:, :, np.newaxis]
1174.         # print(X_test)
1175.         model.load_weights(os.path.join(WEIGHT_DIR, 'best_label_model.hdf5'))
1176.         # model.summary()
1177.         feature_vectors_model = keras.models.Model(model.input,
    model.get_layer('time_distributed').output)
1178.         X_ext = feature_vectors_model.predict(X_test)[:, :, 0]
1179.         # with open('lstmtree_phrase_model_save_best.pkl', 'rb') as f:
1180.         with open(os.path.join(WEIGHT_DIR, 'lstmtree_phrase_model_save_best.pkl'), 'rb') as f:
1181.             dtc = pickle.load(f)
1182.         dtc_y_pred = dtc.predict(X_ext)
1183.         preds = (mlb.inverse_transform(dtc_y_pred))
1184.         all_predictions = [label_encoder.inverse_transform(inarr) for inarr in preds]
1185.         lstmpredictions = []
1186.         for i in range(len(all_peaks)):
1187.             inarr = []
1188.             for j in range(len(all_peaks[i])-1):
1189.                 inarr.append(all_predictions.pop())
1190.             lstmpredictions.append(inarr)
1191.
1192.         with open(os.path.join(WEIGHT_DIR, 'dcntree_phrase_model_save_best.pkl'), 'rb') as f:
1193.             dtc = pickle.load(f)
1194.         dtc_y_pred = dtc.predict(X_test[:, :, 0])
1195.         preds = (mlb.inverse_transform(dtc_y_pred))
1196.         all_predictions = [label_encoder.inverse_transform(inarr) for inarr in preds]
1197.         dtcpredictions = []
1198.         for i in range(len(all_peaks)):
1199.             inarr = []
1200.             for j in range(len(all_peaks[i]) - 1):
1201.                 inarr.append(all_predictions.pop())
1202.             dtcpredictions.append(inarr)
1203.
1204.         with open(os.path.join(WEIGHT_DIR, 'treegrad_phrase_model_save_best.pkl'), 'rb') as f:
1205.             model = pickle.load(f)
1206.         preds = model.predict(X_test[:, :, 0])
1207.         all_predictions = label_encoder.inverse_transform(preds).tolist()
1208.         tgradpredictions = []
1209.         for i in range(len(all_peaks)):
1210.             inarr = []
1211.             for j in range(len(all_peaks[i]) - 1):
1212.                 inarr.append(all_predictions.pop())
1213.             tgradpredictions.append(inarr)
1214.
1215.         # Maybe add "average guess" -> pick most guessed label(s)?
1216.         # Model prioritizes large form label, understandably so
```

```
1217.         alldf = []
1218.         for i in range(len(filenames)):
1219.             df = pd.DataFrame(columns=['Guesser:', 'LSTMTree', 'DcnTree', 'TreeGrad'])
1220.             cnt = 0
1221.             df.loc[cnt] = [namesin[i][0], '', '', '']
1222.             if printform:
1223.                 df.loc[cnt + 1] = [formsin[i][0], '', '', '']
1224.                 cnt += 1
1225.             df.loc[cnt + 1] = ['Guesser:', 'LSTMTree', 'DcnTree', 'TreeGrad']
1226.             df.loc[cnt + 2] = [0.000, 'Silence', 'Silence', 'Silence']
1227.             cnt += 3
1228.             for j in range(len(all_peaks[i]) - 1):
1229.                 if j == 0:
1230.                     df.loc[cnt] = [all_peaks[i][j]+.1, lstmpredictions[i][j], dtcpredictions[i][j],
    tgradpredictions[i][j]]
1231.                 else:
1232.                     df.loc[cnt] = [all_peaks[i][j], lstmpredictions[i][j], dtcpredictions[i][j],
    tgradpredictions[i][j]]
1233.                 cnt += 1
1234.             df.loc[cnt] = [all_peaks[i][-1], 'End', 'End', 'End']
1235.             alldf.append(df)
1236.             cnt += 3
1237.
1238.         print(len(alldf))
1239.
1240.         if printresults:
1241.             pd.set_option('display.precision', 3)
1242.             for i in range(len(alldf)):
1243.                 print(alldf[i].to_string(index=False, header=False), "\n")
1244.         pass
1245.
1246.
1247.     # endregion
1248.
1249.
1250.     """=============================================================================================
    ======================"""
1251.
1252.
1253.     def predictFormAndLabels():
1254.         midpath = input("Enter path to folder or audio file: ")
1255.         predictLabels(midpath, verbose=True, printform=True, printresults=True)
1256.
1257.
1258.     if __name__ == '__main__':
1259.         print("Hello world!")
1260.         # validate_directories()
1261.         # get_total_duration()
1262.         # generate_label_files()
1263.         # prepare_model_training_input()
1264.         # prepare_train_data()
1265.         # buildValidationSet()
1266.         # create_form_dataset()
1267.         # generate_augmented_datasets()
1268.         # prepare_lstm_peaks()
1269.
1270.         # trainFormModel()
1271.         # predictForm()
1272.
1273.         # trainLabelModel(retrain=False)
1274.         # predictLabels()
1275.
1276.         predictFormAndLabels()
1277.         print("\nDone!")
1278.
```

D.2 Data_utils.py

```
1.  import re
2.  import librosa
3.  import matplotlib.pyplot as plt
4.  import librosa.display
5.  import os
6.  import tensorflow as tf
7.  from pydub import AudioSegment
8.  import skimage.measure
9.  from skimage.transform import resize
10. import scipy
11. from scipy.spatial import distance
12. import librosa.segment
13. from sklearn.neighbors import NearestNeighbors
14. import math
15. from scipy import signal
16. import numpy as np
17. from sklearn.preprocessing import LabelEncoder
18. from sklearn.preprocessing import OneHotEncoder
19.
20. MASTER_DIR = 'D:/Google Drive/Resources/Dev/Python/Machine Learning/Master Thesis/'
21.
22. # Output filepath for training images and labels
23. DEFAULT_FILEPATH = os.path.join(MASTER_DIR, 'Images/Train/')
24. DEFAULT_LABELPATH = os.path.join(MASTER_DIR, 'Labels/')
25.
26.
27. class SplitAudio:
28.     def __init__(self, folder, filename, setmono=True):
29.         self.folder = folder
30.         self.filename = filename
31.         # self.filepath = folder + '\\' + filename
32.         self.audio = AudioSegment.from_file(self.filename)
33.         if setmono:
34.             self.audio = self.audio.set_channels(1)
35.
36.     def get_duration(self):
37.         return self.audio.duration_seconds
38.
39.     def get_samplerate(self):
40.         return self.audio.frame_rate
41.
42.     def single_split(self, from_sec, to_sec, split_filename="", export=True):
43.         t1 = from_sec * 1000
44.         t2 = to_sec * 1000
45.         split_audio = self.audio[t1:t2]
46.         if export:
47.             split_audio.export(self.folder + '/' + split_filename[split_filename.index("/")+1:],
    format="wav")
48.         else:
49.             return split_audio
50.
51.     def multiple_split(self, sec_per_split, verbose=True):
52.         total_sec = math.ceil(self.get_duration())
53.         for i in range(0, total_sec, sec_per_split):
54.             split_fn = self.filename[:self.filename.index('.')] + '_' + str(i) + '.wav'
55.             self.single_split(i, i + sec_per_split, split_fn)
56.             if verbose:
57.                 print(str(i) + " Done")
58.                 if i == total_sec - sec_per_split:
59.                     print("All splits completed successfully")
60.                 else:
61.                     print("Error during audio splitting")
62.
63.
```

```
64. def audiosegment_to_ndarray(audiosegment, getSR=False):
65.     samples = audiosegment.get_array_of_samples()
66.     samples_float = librosa.util.buf_to_float(samples, n_bytes=2,
67.                                               dtype=np.float32)
68.     if audiosegment.channels == 2:
69.         sample_left = np.copy(samples_float[::2])
70.         sample_right = np.copy(samples_float[1::2])
71.         sample_all = np.array([sample_left, sample_right])
72.     else:
73.         sample_all = samples_float
74.
75.     if getSR:
76.         return [sample_all, audiosegment.frame_rate]
77.     else:
78.         return sample_all
79.
80.
81. # Novelty Function
82. def peak_picking(filename, name="", foldername="", filepath=DEFAULT_FILEPATH, returnpeaks=True,
    verbose=True):
83.     # window_size = 0.209  # sec/frame
84.     samples_frame = 8192  # samples_frame = math.ceil(window_size*sr)
85.     # hop_size = 0.139  # sec/frame
86.     hop_length = 6144  # hop_length = math.ceil(hop_size*sr) #overlap 25% (samples/frame)
87.     sr_desired = 44100
88.     if filepath != DEFAULT_FILEPATH:
89.         pass
90.
91.     y, sr = librosa.load(filename, sr=None)
92.
93.     if sr != sr_desired:
94.         y = librosa.core.resample(y, sr, sr_desired)
95.         sr = sr_desired
96.
97.     stft = np.abs(librosa.stft(y, n_fft=samples_frame, hop_length=hop_length))
98.     # fft_freq = librosa.core.fft_frequencies(sr=sr, n_fft=samples_frame)
99.
100.    # Plot Mel-Spectogram from SFTF
101.    if verbose:
102.        librosa.display.specshow(librosa.amplitude_to_db(stft, ref=np.max), y_axis='log',
    x_axis='frames')
103.        plt.title('Power spectrogram')
104.        plt.colorbar(format='%+2.0f dB')
105.        plt.tight_layout()
106.        plt.show()
107.
108.    chroma = librosa.feature.chroma_stft(S=stft, sr=sr, n_fft=samples_frame, hop_length=hop_length)
109.
110.    # Plot PCPs or Chroma from spectogram
111.    if verbose:
112.        plt.figure(figsize=(10, 4))
113.        librosa.display.specshow(chroma, sr=sr, y_axis='chroma', x_axis='frames', cmap="coolwarm")
114.        plt.colorbar()
115.        plt.title('Chromagram')
116.        plt.tight_layout()
117.        plt.show()
118.        print("Chroma dimensions are: [chroma vectors, N']")
119.        print("Chroma dimensions are: [", chroma.shape[0], ",", chroma.shape[1], "]")
120.
121.    # vector x_hat construction. x in Serra's paper is chroma here
122.    m = round(5 * sr / hop_length)
123.    tau = 1
124.    w = (m - 1) * tau
125.    chroma = np.concatenate((np.zeros((chroma.shape[0], w)), chroma), axis=1)
126.    x = [np.roll(chroma, tau * n, axis=1) for n in range(m)]
127.    x_ = np.concatenate(x, axis=0)
128.
129.    X_hat = x_[:, w:]  # (w, frames)
130.
```

```
    N_prime = chroma.shape[1]
    N = N_prime - w

    # Plot x, x_ and resulting x_hat
    # x (first chroma)
    if verbose:
        plt.figure(figsize=(15, 7))
        plt.title('First chroma vector: x[0]')
        plt.imshow(np.asarray(x[0]), origin='lower', cmap='plasma', aspect=2)
        plt.show()

    # x_
    if verbose:
        plt.figure(figsize=(15, 7))
        plt.title('x_')
        plt.imshow(x_, origin='lower', cmap='plasma', aspect=0.5)
        plt.show()

    # x_hat
    if verbose:
        plt.figure(figsize=(15, 7))
        plt.title('x_hat')
        plt.imshow(X_hat, origin='lower', cmap='plasma', aspect=0.5)
        plt.show()
        print("X_hat dimensions are: [chroma vectors * m (in samples), N'] = [", chroma.shape[0], "*", m, ", N']")
        print("X_hat dimensions are: [", X_hat.shape[0], ",", X_hat.shape[1], "]")

    # Recurrence matrix from librosa
    recurrence = librosa.segment.recurrence_matrix(chroma, mode='affinity', k=chroma.shape[1])
    if verbose:
        plt.figure(figsize=(7, 7))
        plt.title('Recurrence matrix from chroma vector from LIBROSA')
        plt.imshow(recurrence, cmap='gray')
        plt.show()

    # Plot recurrence matrix of vector x with librosa
    recurrence2 = librosa.segment.recurrence_matrix(x, k=14, sym=True)
    if verbose:
        plt.figure(figsize=(7, 7))
        plt.title('Recurrence matrix of x vector with k=13 neighbors from LIBROSA')
        plt.imshow(1 - recurrence2, cmap='gray')
        plt.show()

    # KNN
    K = 14  # K = round(N*0.03)
    nbrs = NearestNeighbors(n_neighbors=K).fit(X_hat.T)
    distances, indices = nbrs.kneighbors(X_hat.T)
    R = np.zeros((N, N))
    for i in range(N):
        for j in range(N):
            if (i in indices[j]) and (j in indices[i]) and (i != j):
                R[i, j] = 1

    # Plot recurrence matrix of vector R (same as above)
    if verbose:
        plt.figure(figsize=(7, 7))
        plt.title('Recurrence matrix R')
        plt.imshow(1 - R, cmap='gray')
        plt.show()

    L = librosa.segment.recurrence_to_lag(R, pad=False)  # None

    # Lag Matrix calculated from R
    if verbose:
        plt.figure(figsize=(7, 7))
        plt.title('Lag Matrix')
        plt.imshow(1 - L, cmap='gray')
        plt.show()
```

```
199.
200.    # Smoothing signal with Gaussian windows of 30 samples length
201.    s1 = round(0.3 * sr / hop_length)
202.    st = round(30 * sr / hop_length)
203.    sigma1 = (s1 - 1) / (2.5 * 2)
204.    sigmat = (st - 1) / (2.5 * 2)
205.    g1 = signal.gaussian(s1, std=sigma1).reshape(s1, 1)  # g1 in paper
206.    gt = signal.gaussian(st, std=sigmat).reshape(st, 1)  # gt in paper
207.    G = np.matmul(g1, gt.T)
208.
209.    # Plot Gaussian window
210.    if verbose:
211.        plt.plot(gt)
212.        plt.title("Gaussian window ($\sigma$=7)")
213.        plt.ylabel("Amplitude")
214.        plt.xlabel("Sample")
215.        plt.show()
216.
217.    # Gaussian kernel G
218.    if verbose:
219.        plt.figure(figsize=(7, 7))
220.        plt.title('Gaussian kernel G')
221.        plt.imshow(1 - G, origin='lower', cmap='gray', aspect=40)
222.        plt.show()
223.
224.    # Applyin gaussian filter to Lag matrix
225.    P = signal.convolve2d(L, G, mode='same')
226.
227.    # Plot R matrix after Gaussian smoothing
228.    P2 = librosa.segment.lag_to_recurrence(P, axis=-1)
229.    if verbose:
230.        plt.figure(figsize=(7, 7))
231.        plt.title('Recurrence matrix R after gaussian')
232.        plt.imshow(1 - P2, cmap='gray')
233.        plt.show()
234.
235.    # Plot Lag matrix after Gaussian smoothing
236.    if verbose:
237.        plt.figure(figsize=(7, 7))
238.        plt.title('Lag matrix L after gaussian')
239.        plt.imshow(1 - P, cmap='gray')
240.        plt.show()
241.
242.    # Novelty curve
243.    c = np.linalg.norm(P[:, 1:] - P[:, 0:-1], axis=0)
244.    c_norm = (c - c.min()) / (c.max() - c.min())  # normalization of c
245.
246.    # Plot novelty function with boundaries
247.    frames = range(len(c_norm))
248.    if verbose:
249.        plt.figure(figsize=(10, 4))
250.        plt.title('Novelty function vector c')
251.        plt.xlabel('Frames')
252.        plt.plot(frames, c_norm)
253.        plt.show()
254.
255.    # Peaks detection - sliding window
256.    delta = 0.05  # threshold
257.    lamda = round(6 * sr / hop_length)  # window length
258.    peaks_position = signal.find_peaks(c_norm, height=delta, distance=lamda, width=round(0.5 * sr /
    hop_length))[
259.        0]  # array of peaks
260.    # peaks_values = signal.find_peaks(c_norm, height=delta, distance=lamda, width=round(0.5 * sr /
    hop_length))[1][
261.    #     'peak_heights']  # array of peaks
262.    b = peaks_position
263.    # Adding elements 1 and N' to the begining and end of the arrray
264.    if len(b) == 0 or b[0] != 0:
265.        b = np.concatenate([[0], b])  # b: segment boundaries
```

```
266.    if b[-1] != N_prime - 1:
267.        b = np.concatenate([b, [N - 1]])
268.
269.    # Plot novelty function with boundaries
270.    frames = range(len(c_norm))
271.    if verbose:
272.        plt.figure(figsize=(10, 4))
273.        plt.title('Novelty function vector c (red lines are peaks)')
274.        plt.xlabel('Frames')
275.        for i in range(len(b)):
276.            plt.axvline(b[i], color='r', linestyle='--')
277.        plt.plot(frames, c_norm)
278.        plt.show()
279.
280.    if returnpeaks:
281.        peaktimes = []
282.        for i in range(len(b)):
283.            timeSecondsDecimal = b[i] / sr * hop_length
284.            peaktimes.append(timeSecondsDecimal)
285.        return peaktimes
286.
287.    # Cumulative matrix: Q
288.    Q = np.zeros_like(R)
289.    for u in range(b.shape[0] - 1):
290.        for v in range(b.shape[0] - 1):
291.            Q_uv = np.copy(R[b[u]:b[u + 1], b[v]:b[v + 1]])
292.            for i in range(1, Q_uv.shape[0]):
293.                for j in range(1, Q_uv.shape[1]):
294.                    if i == 1 and j == 1:
295.                        Q_uv[i, j] += Q_uv[i - 1, j - 1]
296.                    elif i == 1:
297.                        Q_uv[i, j] += max(Q_uv[i - 1, j - 1], Q_uv[i - 1, j - 2])
298.                    elif j == 1:
299.                        Q_uv[i, j] += max(Q_uv[i - 1, j - 1], Q_uv[i - 2, j - 1])
300.                    else:
301.                        Q_uv[i, j] += max(Q_uv[i - 1, j - 1], Q_uv[i - 2, j - 1], Q_uv[i - 1, j - 2])
302.
303.            Q[b[u]:b[u + 1], b[v]:b[v + 1]] = Q_uv
304.
305.    # Cumulative matrix plot
306.    plt.figure(figsize=(7, 7))
307.    plt.title('Cumulative matrix Q')
308.    plt.imshow(1 - Q, cmap='gray')
309.    plt.show()
310.
311.    # Normalization of Q matrix: Segment similarity matrix S
312.    num_segments = b.shape[0] - 1
313.    S = np.zeros((num_segments, num_segments))
314.    for u in range(b.shape[0] - 1):
315.        for v in range(b.shape[0] - 1):
316.            S[u, v] = np.max(Q[b[u]:b[u + 1], b[v]:b[v + 1]]) / min(b[u + 1] - b[u], b[v + 1] - b[v])
317.
318.    # Plot Segment similarity matrix S
319.    plt.figure(figsize=(7, 7))
320.    plt.title('Segment matrix S')
321.    plt.imshow(1 - S, cmap='gray')
322.    # for i in range(len(b)):
323.    # plt.axvline(b[i], color='r', linestyle='--')
324.    # plt.axhline(b[i], color='r', linestyle='--')
325.    plt.show()
326.
327.    # Transitive Binary Similarity Matrix: S_hat
328.    S_hat = S > S.mean() + S.std()
329.    S_hat_norm = np.matmul(S_hat, S_hat)
330.    while (S_hat_norm < S_hat).all():
331.        S_hat = S_hat_norm
332.        S_hat_norm = np.matmul(S_hat, S_hat)
333.        S_hat_norm = S_hat_norm >= 1
334.
```

```
335.    # Plot transitive binary similarity matrix S_hat
336.    plt.figure(figsize=(7, 7))
337.    plt.title('Segment transitive binary similarity matrix S_hat')
338.    plt.imshow(1 - S_hat_norm, cmap='gray')
339.    plt.show()
340.
341.    # Image vs ground truth - Plot S with labels
342.    S_frames = np.zeros_like(Q)
343.    for u in range(b.shape[0] - 1):
344.        for v in range(b.shape[0] - 1):
345.            S_frames[b[u]:b[u + 1], b[v]:b[v + 1]] = S_hat_norm[u, v]
346.
347.    label_path = DEFAULT_LABELPATH + foldername
348.    file = "/" + os.path.basename(name) + ".txt"
349.    nums, lbls, form = ReadDataFromtxt(label_path, file)
350.    labels_array = np.asarray(nums)
351.    array = labels_array.astype(np.float)
352.
353.    plt.figure(figsize=(7, 7))
354.    plt.title('Segment Similarity Matrix S with labels')
355.    plt.imshow(1 - S_frames, cmap='gray')
356.    for i in range(len(array)):
357.        plt.axvline(array[i] * sr / hop_length, color='b', linestyle='--')
358.        plt.axhline(array[i] * sr / hop_length, color='b', linestyle='--')
359.    plt.show()
360.    print()
361.
362.    fig = plt.figure(figsize=(7, 7))
363.    plt.imshow(1 - S_frames, cmap='gray')
364.    for i in range(len(array)):
365.        plt.axvline(array[i] * sr / hop_length, color='b', linestyle='--')
366.        plt.axhline(array[i] * sr / hop_length, color='b', linestyle='--')
367.    ax = fig.add_subplot(111)
368.    ax.axes.get_xaxis().set_visible(False)
369.    ax.axes.get_yaxis().set_visible(False)
370.    ax.set_frame_on(False)
371.    # filename = filepath + "SSLMCRM/" + os.path.basename(name) + 'crm.png'
372.    # plt.savefig(filename, bbox_inches='tight', pad_inches=0)  # dpi=400, transparent=True
373.    fig.clf()
374.    plt.close(fig)
375.    del ax, fig
376.
377.    # Plot novelty function with boundaries
378.    frames = range(len(c_norm))
379.    plt.figure(figsize=(10, 4))
380.    plt.title('Novelty function vector c (red lines are peaks and black lines are labels)')
381.    plt.xlabel('Frames')
382.    timeDifs = []
383.    # dbltb = "\t\t"
384.    # nspc = ""
385.    for i in range(len(array)):
386.        plt.axvline(array[i] * sr / hop_length, color='black', linestyle='-')
387.    for i in range(len(b)):
388.        plt.axvline(b[i], color='r', linestyle='--')
389.        # timeSecondsDecimal = b[i] / sr * hop_length
390.        """
391.        # DEMO EVENT COMPARISON
392.        timeStr = str(datetime.timedelta(seconds=timeSecondsDecimal))
393.        gtTimeStr = 0
394.        timeDifference = 0
395.        if i < len(array):  # Demonstration only
396.            gtTimeStr = str(datetime.timedelta(seconds=array[i]))
397.            timeDifference = array[i] - timeSecondsDecimal
398.        print(f"Event: {timeStr}\t\t{dbltb if i == 0 else nspc}Ground Truth: {gtTimeStr}\t\t{dbltb if
    i == 0 else nspc}"
399.              f"Difference: "
400.              f"{'{:.6f}'.format(timeDifference) if timeDifference < 0 else '{:
    .6f}'.format(timeDifference)}\t\t"
401.              # f"G.T. Labels: {lbls[0]}")
```

```
402.                 f"G.T. Labels: {lbls[i]}")
403.         timeDifs = np.append(timeDifs, abs(timeDifference))
404.         """
405.     plt.plot(frames, c_norm)
406.     plt.show()
407.     print("\nAverage (absolute) time difference: ±" + str(np.average(timeDifs)))
408.
409.
410.# region ReadFiles
411.def ReadNumbersFromLine(line):
412.     number = re.split(r'\s\s*', line)[0]
413.     number = float(number)
414.     return number
415.
416.
417.def ReadLabelsFromLine(line):
418.     labels = re.split(r'\s\s*', line)[1:]
419.     for i in range(len(labels)):
420.         labels[i] = labels[i].replace(',', '')
421.     return np.asarray(labels).astype(object)
422.
423.
424.def ReadImagesFromFolder(directory):
425.     imgs = []
426.     for (img_dir_path, img_dnames, img_fnames) in os.walk(directory):
427.         for f in img_fnames:
428.             img_path = img_dir_path + f
429.             img = plt.imread(img_path)
430.             img = resize(img, (200, 1150, 4))
431.             imgs.append(img)
432.     return imgs
433.
434.
435.def ReadDataFromtxt(directory, archive):
436.     numbers = []
437.     labels = []
438.     cnt = 1
439.     # for _ in listdir(directory):
440.     cnt += 1
441.     file = open(directory + archive, "r")
442.     form = next(file).strip()
443.     for line in file:
444.         numbers.append(ReadNumbersFromLine(line))
445.         labels.append(ReadLabelsFromLine(line.rstrip()))
446.     file.close()
447.     return numbers, np.asarray(labels).astype(object), form
448.
449.
450.def ReadLabelSecondsPhrasesFromFolder(lblpath=DEFAULT_LABELPATH, stop=-1, valid_only=False,
451.                                       get_names=False, get_forms=False):
452.     nums = []
453.     lbls = []
454.     forms = []
455.     fnames = []
456.     for (lbl_dir_path, lbl_dnames, lbl_fnames) in os.walk(lblpath):
457.         for f in lbl_fnames:
458.             if valid_only:
459.                 num_lines = sum(1 for _ in open(lbl_dir_path + f))
460.                 if num_lines <= 3:
461.                     # print("File has not been labeled with ground truth yet. Skipping...")
462.                     continue
463.             if stop != -1:
464.                 stop -= 1
465.                 if stop == 0:
466.                     break
467.             # prepend_line(lbl_dir_path + '/' + f, lbl_dir_path.split('/')[-1])  # Run once for
    master label set
468.             numsIn, lblsIn, formsIn = ReadDataFromtxt(lbl_dir_path + '/', f)
469.             numsIn = np.array(numsIn, dtype=np.float32)
```

```
470.            nums.append(numsIn)
471.            lbls.append(lblsIn)
472.            forms.append([formsIn])
473.            fnames.append(f)
474.
475.    # Convert Forms to One Hot encoding
476.    values = np.array(forms)  # print(values)
477.    label_encoder = LabelEncoder()
478.    label_encoder.classes_ = np.load(os.path.join(WEIGHT_DIR, 'form_classes.npy'))
479.    integer_encoded = label_encoder.transform(values)  # print(integer_encoded)
480.    onehot_encoder = OneHotEncoder(sparse=False)
481.    integer_encoded = integer_encoded.reshape(len(integer_encoded), 1)
482.    onehot_encoded = onehot_encoder.fit_transform(integer_encoded)  # print(onehot_encoded)
483.    # onehot_encoded = to_categorical(integer_encoded, len(label_encoder.classes_))
484.    # inverted = label_encoder.inverse_transform([argmax(onehot_encoded[0, :])])  # Return original
    label from encoding
485.    # np.save(os.path.join(MASTER_DIR, 'form_classes.npy'), label_encoder.classes_)
486.    # print(label_encoder.classes_)
487.    """
488.    # Convert Phrases to One Hot encoding
489.    values = np.array([np.array([np.array(y) for y in x]) for x in lbls])  # print(values)
490.    print(values)
491.    label_encoder = LabelEncoder()
492.    integer_encoded = label_encoder.fit_transform(values)  # print(integer_encoded)
493.    onehot_encoder = OneHotEncoder(sparse=False)
494.    integer_encoded = integer_encoded.reshape(len(integer_encoded), 1)
495.    onehot_labels = onehot_encoder.fit_transform(integer_encoded)  # print(onehot_encoded)
496.    # inverted = label_encoder.inverse_transform([argmax(onehot_encoded[0, :])])  # Return original
    label from encoding
497.    """
498.    if get_names:
499.        if get_forms:
500.            return nums, np.asarray(lbls), integer_encoded, np.asarray(fnames)
501.        return nums, np.asarray(lbls), tf.expand_dims(onehot_encoded, axis=-1), np.asarray(fnames)
502.    return nums, np.asarray(lbls), tf.expand_dims(onehot_encoded, axis=-1)
503.
504.
505.def prepend_line(file_name, line):
506.    """Insert string as a new line at the beginning of a file"""
507.    dummy_file = file_name + '.bak'
508.    with open(file_name, 'r') as read_obj, open(dummy_file, 'w') as write_obj:
509.        write_obj.write(line + '\n')
510.        for line in read_obj:
511.            write_obj.write(line)
512.    os.remove(file_name)
513.    os.rename(dummy_file, file_name)
514.    print("Finished prepending to " + file_name)
515.
516.
517.# endregion
518.
```

D.3 Data_utils_input.py

```
1.  import glob as gb
2.  import librosa
3.  import librosa.display
4.  import numpy as np
5.  import time
6.  import skimage.measure
7.  import os
8.  import scipy
9.  from scipy.spatial import distance
10. import pandas as pd
11. import tensorflow.keras as k
12. import data_utils as du
13.
14. start_time = time.time()
15.
16.
17. # region DataPreparation
18. def compute_ssm(X, metric="cosine"):
19.     """Computes the self-similarity matrix of X."""
20.     D = distance.pdist(X, metric=metric)
21.     D = distance.squareform(D)
22.     for i in range(D.shape[0]):
23.         for j in range(D.shape[1]):
24.             if np.isnan(D[i, j]):
25.                 D[i, j] = 0
26.     D /= D.max()
27.     return 1 - D
28.
29.
30. def mel_spectrogram(sr_desired, filepath, window_size, hop_length):
31.     """This function calculates the mel spectrogram in dB with Librosa library"""
32.     y, sr = librosa.load(filepath, sr=None)
33.     if sr != sr_desired:
34.         y = librosa.core.resample(y, sr, sr_desired)
35.         sr = sr_desired
36.
37.     S = librosa.feature.melspectrogram(y=y, sr=sr, n_fft=window_size, hop_length=hop_length,
    n_mels=80, fmin=80,
38.                                        fmax=16000)
39.     S_to_dB = librosa.power_to_db(S, ref=np.max)  # convert S in dB
40.     return S_to_dB  # S_to_dB is the spectrogam in dB
41.
42.
43. def fourier_transform(sr_desired, name_song, window_size, hop_length):
44.     """This function calculates the mel spectrogram in dB with Librosa library"""
45.     y, sr = librosa.load(name_song, sr=None)
46.     if sr != sr_desired:
47.         y = librosa.core.resample(y, sr, sr_desired)
48.         sr = sr_desired
49.     stft = np.abs(librosa.stft(y=y, n_fft=window_size, hop_length=hop_length))
50.     return stft
51.
52.
53. def max_pooling(stft, pooling_factor):
54.     x_prime = skimage.measure.block_reduce(stft, (1, pooling_factor), np.max)
55.     return x_prime
56.
57.
58. def sslm_gen(spectrogram, pooling_factor, lag, mode, feature):
59.     padding_factor = lag
60.     """This part pads a mel spectrogram gived the spectrogram a lag parameter
61.     to compare the first rows with the last ones and make the matrix circular"""
62.     pad = np.full((spectrogram.shape[0], padding_factor), -70)  # 80x30 frame matrix of -70dB
    corresponding to padding
```

```
63.     S_padded = np.concatenate((pad, spectrogram), axis=1)  # padding 30 frames with noise at -70dB at
    the beginning
64.
65.     """This part max-poolend the spectrogram in time axis by a factor of p"""
66.     x_prime = max_pooling(S_padded, pooling_factor)
67.     x = []
68.     if feature == "mfcc":
69.
70.         """This part calculates a circular Self Similarity Lag Matrix given
71.         the mel spectrogram padded and max-pooled"""
72.         # MFCCs calculation from DCT-Type II
73.         MFCCs = scipy.fftpack.dct(x_prime, axis=0, type=2, norm='ortho')
74.         MFCCs = MFCCs[1:, :]  # 0 componen ommited
75.
76.         # Bagging frames
77.         m = 2  # baggin parameter in frames
78.         x = [np.roll(MFCCs, n, axis=1) for n in range(m)]
79.     elif feature == "chroma":
80.         """This part calculates a circular Self Similarity Lag Matrix given
81.                 the chromagram padded and max-pooled"""
82.         PCPs = librosa.feature.chroma_stft(S=x_prime, sr=sr_desired, n_fft=window_size,
    hop_length=hop_length)
83.         PCPs = PCPs[1:, :]
84.
85.         # Bagging frames
86.         m = 2  # Bagging parameter in frames
87.         x = [np.roll(PCPs, n, axis=1) for n in range(m)]
88.
89.     x_hat = np.concatenate(x, axis=0)
90.
91.     # Cosine distance calculation: D[N/p,L/p] matrix
92.     distances = np.zeros((x_hat.shape[1], padding_factor // p))  # D has as dimensions N/p and L/p
93.     for i in range(x_hat.shape[1]):  # iteration in columns of x_hat
94.         for l in range(padding_factor // p):
95.             if i - (l + 1) < 0:
96.                 cur_dist = 1
97.             elif i - (l + 1) < padding_factor // p:
98.                 cur_dist = 1
99.             else:
100.                cur_dist = 0
101.                if mode == "cos":
102.                    cur_dist = distance.cosine(x_hat[:, i],
103.                                               x_hat[:, i - (l + 1)])  # cosine distance between
    columns i and i-L
104.                elif mode == "euc":
105.                    cur_dist = distance.euclidean(x_hat[:, i],
106.                                                  x_hat[:, i - (l + 1)])  # euclidian distance
    between columns i and i-L
107.                if cur_dist == float('nan'):
108.                    cur_dist = 0
109.            distances[i, l] = cur_dist
110.
111.    # Threshold epsilon[N/p,L/p] calculation
112.    kappa = 0.1
113.    epsilon = np.zeros((distances.shape[0], padding_factor // p))  # D has as dimensions N/p and L/p
114.    for i in range(padding_factor // p, distances.shape[0]):  # iteration in columns of x_hat
115.        for l in range(padding_factor // p):
116.            epsilon[i, l] = np.quantile(np.concatenate((distances[i - l, :], distances[i, :])),
    kappa)
117.
118.    # We remove the padding done before
119.    distances = distances[padding_factor // p:, :]
120.    epsilon = epsilon[padding_factor // p:, :]
121.    x_prime = x_prime[:, padding_factor // p:]
122.
123.    # Self Similarity Lag Matrix
124.    sslm = scipy.special.expit(1 - distances / epsilon)  # aplicación de la sigmoide
125.    sslm = np.transpose(sslm)
126.    sslm = skimage.measure.block_reduce(sslm, (1, 3), np.max)
```

```
127.
128.    # Check if SSLM has nans and if it has them, substitute them by 0
129.    for i in range(sslm.shape[0]):
130.        for j in range(sslm.shape[1]):
131.            if np.isnan(sslm[i, j]):
132.                sslm[i, j] = 0
133.
134.    # if mode == "euc":
135.    #     return sslm, x_prime
136.
137.    # return sslm
138.    return sslm, x_prime
139.
140.
141.def ssm_gen(spectrogram, pooling_factor):
142.    """This part max-poolend the spectrogram in time axis by a factor of p"""
143.    x_prime = max_pooling(spectrogram, pooling_factor)
144.
145.    """This part calculates a circular Self Similarity Matrix given
146.    the mel spectrogram padded and max-pooled"""
147.    # MFCCs calculation from DCT-Type II
148.    MFCCs = scipy.fftpack.dct(x_prime, axis=0, type=2, norm='ortho')
149.    MFCCs = MFCCs[1:, :]  # 0 componen ommited
150.
151.    # Bagging frames
152.    m = 2  # baggin parameter in frames
153.    x = [np.roll(MFCCs, n, axis=1) for n in range(m)]
154.    x_hat = np.concatenate(x, axis=0)
155.    x_hat = np.transpose(x_hat)
156.
157.    ssm = compute_ssm(x_hat)
158.
159.    # Check if SSLM has nans and if it has them, substitute them by 0
160.    for i in range(ssm.shape[0]):
161.        for j in range(ssm.shape[1]):
162.            if np.isnan(ssm[i, j]):
163.                ssm[i, j] = 0
164.
165.    return ssm
166.# endregion
167.
168.
169.window_size = 2048  # (samples/frame)
170.hop_length = 1024  # overlap 50% (samples/frame)
171.sr_desired = 44100
172.p = 2  # pooling factor
173.p2 = 3  # 2pool3
174.L_sec_near = 14  # lag near context in seconds
175.L_near = round(L_sec_near * sr_desired / hop_length)  # conversion of lag L seconds to frames
176.
177.MASTER_DIR = 'D:/Google Drive/Resources/Dev/Python/Machine Learning/Master Thesis/'
178.DEFAULT_LABELPATH = os.path.join(MASTER_DIR, 'Labels/')
179.TRAIN_DIR = 'F:/Master Thesis Input/NewTrain/'
180.MIDI_DIR = os.path.join(MASTER_DIR, 'Data/MIDIs/')
181.
182.
183.def util_main_helper(feature, filepath, mode="cos", predict=False, savename=""):
184.    sslm_near = None
185.    if feature == "mfcc":
186.        mel = mel_spectrogram(sr_desired, filepath, window_size, hop_length)
187.        if mode == "cos":
188.            sslm_near = sslm_gen(mel, p, L_near, mode=mode, feature="mfcc")[0]
189.            # mls = max_pooling(mel, p2)
190.            # Save mels matrices and sslms as numpy arrays in separate paths
191.            # np.save(im_path_mel_near + song_id, mls)
192.        elif mode == "euc":
193.            sslm_near = sslm_gen(mel, p, L_near, mode=mode, feature="mfcc")[0]
194.            if sslm_near.shape[1] < max_pooling(mel, 6).shape[1]:
195.                sslm_near = np.hstack((np.ones((301, 1)), sslm_near))
```

```
            elif sslm_near.shape[1] > max_pooling(mel, 6).shape[1]:
                sslm_near = sslm_near[:, 1:]
    elif feature == "chroma":
        stft = fourier_transform(sr_desired, filepath, window_size, hop_length)
        sslm_near = sslm_gen(stft, p, L_near, mode=mode, feature="chroma")[0]
        if mode == "euc":
            if sslm_near.shape[1] < max_pooling(stft, 6).shape[1]:
                sslm_near = np.hstack((np.ones((301, 1)), sslm_near))
            elif sslm_near.shape[1] > max_pooling(stft, 6).shape[1]:
                sslm_near = sslm_near[:, 1:]
    elif feature == "mls":
        mel = mel_spectrogram(sr_desired, filepath, window_size, hop_length)
        sslm_near = ssm_gen(mel, pooling_factor=6)

    """
    # UNCOMMENT TO DISPLAY FEATURE GRAPHS
    # recurrence = librosa.segment.recurrence_matrix(sslm_near, mode='affinity',
    k=sslm_near.shape[1])
    plt.figure(figsize=(15, 10))
    if feature == "mls":
        plt.title("Mel Log-scaled Spectrogram - Self-Similarity matrix (MLS SSM)")
        plt.imshow(sslm_near, origin='lower', cmap='plasma', aspect=0.8)  # switch to recurrence if
    desired
    else:
        plt_title = "Self-Similarity Lag Matrix (SSLM): "
        if feature == "chroma":
            plt_title += "Chromas, "
        else:
            plt_title += "MFCCs, "
        if mode == "cos":
            plt_title += "Cosine Distance"
        else:
            plt_title += "Euclidian Distance"
        plt.title(plt_title)
        plt.imshow(sslm_near.astype(np.float32), origin='lower', cmap='viridis', aspect=0.8)
        # switch to recurrence if desired
    plt.show()
    """
    if not predict:
        # Save matrices and sslms as numpy arrays in separate paths
        np.save(filepath, sslm_near)
    else:
        return sslm_near


def util_main(feature, mode="cos", predict=False, inpath=TRAIN_DIR, midpath=MIDI_DIR):
    img_path = ""

    if feature == "mfcc":
        if mode == "cos":
            img_path = os.path.join(inpath, 'SSLM_MFCC_COS/')
        elif mode == "euc":
            img_path = os.path.join(inpath, 'SSLM_MFCC_EUC/')
    elif feature == "chroma":
        if mode == "cos":
            img_path = os.path.join(inpath, 'SSLM_CRM_COS/')
        elif mode == "euc":
            img_path = os.path.join(inpath, 'SSLM_CRM_EUC/')
    elif feature == "mls":
        img_path = os.path.join(inpath, 'MLS/')

    if not os.path.exists(img_path):
        os.makedirs(img_path)

    num_songs = sum([len(files) for r, d, files in os.walk(midpath)])
    i = 0
    for folder in gb.glob(midpath + "*"):
        for file in os.listdir(folder):
            # foldername = folder.split('\\')[-1]
```

```
263.            name_song, name = file, file.split('/')[-1].split('.')[0]
264.            start_time_song = time.time()
265.            i += 1
266.            song_id = name_song[:-4]  # delete .ext characters from the string
267.            print("\tPreparing", song_id, "for processing...")
268.            if str(song_id) + ".npy" not in os.listdir(img_path):
269.                util_main_helper(feature, folder + '/' + name_song, mode, predict, savename=img_path
    + song_id)
270.                print("\t\tFinished", i, "/", num_songs, "- Duration: {:.2f}s".format(time.time() -
    start_time_song))
271.            else:
272.                print("\t\tAlready completed. Skipping...\n\t\tFinished", i, "/", num_songs)
273.            # return
274.    print("All files have been converted. Duration: {:.2f}s".format(time.time() - start_time))
275.
276.
277.def validate_folder_contents(labels, midis, mlsdir, sslm1, sslm2, sslm3, sslm4):
278.    """Ensure all folders contain files of the same name"""
279.    labelfiles = os.listdir(labels)
280.    midifiles = os.listdir(midis)
281.    mlsfiles = os.listdir(mlsdir)
282.    sslm1files = os.listdir(sslm1)
283.    sslm2files = os.listdir(sslm2)
284.    sslm3files = os.listdir(sslm3)
285.    sslm4files = os.listdir(sslm4)
286.
287.    for i in range(len(labelfiles)):
288.        c_lbl = os.path.splitext(labelfiles[i])[0]
289.        c_midi = os.path.splitext(midifiles[i])[0]
290.        c_mls = os.path.splitext(mlsfiles[i])[0]
291.        c_sslm1 = os.path.splitext(sslm1files[i])[0]
292.        c_sslm2 = os.path.splitext(sslm2files[i])[0]
293.        c_sslm3 = os.path.splitext(sslm3files[i])[0]
294.        c_sslm4 = os.path.splitext(sslm4files[i])[0]
295.
296.        if c_lbl != c_midi or c_lbl != c_mls or\
297.                c_lbl != c_sslm1 or c_lbl != c_sslm2 or c_lbl != c_sslm3 or c_lbl != c_sslm4:
298.            err = FileNotFoundError("File discrepency at index " + str(i))
299.            print("Current labels: ")
300.            print(f"Label: {c_lbl}\nMIDI: {c_midi}\nMLS: {c_mls}\nSSLM-CRM-COS: {c_sslm1}"
301.                  f"\nSSLM-CRM-EUC: {c_sslm2}\nSSLM-MFCC-COS: {c_sslm3}\nSSLM-MFCC-EUC: {c_sslm4}")
302.            raise err
303.
304.    if len(labelfiles) != len(midifiles) or len(labelfiles) != len(mlsfiles) or \
305.            len(labelfiles) != len(sslm1files) or len(labelfiles) != len(sslm2files) or\
306.            len(labelfiles) != len(sslm3files) or len(labelfiles) != len(sslm4files):
307.        raise ValueError("Not all directories contain the same number of files")
308.
309.
310.# region Transformations
311.def gaussian(x, mu, sig):
312.    """Create array of labels"""
313.    return np.exp(-np.power((x - mu) / sig, 2.) / 2)
314.
315.
316.def borders(image, label, labels_sec, label_form):
317.    """This function transforms labels in sc to gaussians in frames"""
318.    pooling_factor = 6
319.    num_frames = image.shape[2]
320.    repeated_label = []
321.    for i in range(len(labels_sec) - 1):
322.        if labels_sec[i] == labels_sec[i + 1]:
323.            repeated_label.append(i)
324.    labels_sec = np.delete(labels_sec, repeated_label, 0)  # labels in seconds
325.    labels_sec = labels_sec / pooling_factor  # labels in frames
326.
327.    # Pad frames we padded in images also in labels but in seconds
328.    sr = sr_desired
329.    padding_factor = 50
```

```
330.    label_padded = [labels_sec[i] + padding_factor * hop_length / sr for i in
    range(labels_sec.shape[0])]
331.    vector = np.arange(num_frames)
332.    new_vector = (vector * hop_length + window_size / 2) / sr
333.    sigma = 0.1
334.    gauss_array = []
335.    for mu in (label_padded[1:]):  # Ignore first label (beginning of song) due to insignificance
    (0.000 Silence)
336.        gauss_array = np.append(gauss_array, gaussian(new_vector, mu, sigma))
337.    for i in range(len(gauss_array)):
338.        if gauss_array[i] > 1:
339.            gauss_array[i] = 1
340.    return image, label[1:], gauss_array, label_form
341.
342.
343.def padding_MLS(image, label, labels_sec, label_form):
344.    """This function pads 30frames at the begining and end of an image"""
345.    sr = sr_desired
346.    padding_factor = 50
347.
348.    def voss(nrows, ncols=16):
349.        """Generates pink noise using the Voss-McCartney algorithm.
350.
351.        nrows: number of values to generate
352.        rcols: number of random sources to add
353.
354.        returns: NumPy array
355.        """
356.        array = np.empty((nrows, ncols))
357.        array.fill(np.nan)
358.        array[0, :] = np.random.random(ncols)
359.        array[:, 0] = np.random.random(nrows)
360.
361.        # the total number of changes is nrows
362.        n = nrows
363.        cols = np.random.geometric(0.5, n)
364.        cols[cols >= ncols] = 0
365.        rows = np.random.randint(nrows, size=n)
366.        array[rows, cols] = np.random.random(n)
367.
368.        df = pd.DataFrame(array)
369.        df.fillna(method='ffill', axis=0, inplace=True)
370.        total = df.sum(axis=1)
371.
372.        return total.values
373.
374.    n_mels = image.shape[1]  # Default(80) - fit padding to image height
375.    y = voss(padding_factor * hop_length - 1)
376.    S = librosa.feature.melspectrogram(y=y, sr=sr, n_fft=window_size, hop_length=hop_length,
377.                                       n_mels=n_mels, fmin=80, fmax=16000)
378.    S_to_dB = librosa.power_to_db(S, ref=np.max)
379.    pad_image = S_to_dB[np.newaxis, :, :]
380.
381.    # Pad MLS
382.    S_padded = np.concatenate((pad_image, image), axis=-1)
383.    S_padded = np.concatenate((S_padded, pad_image), axis=-1)
384.    return S_padded, label, labels_sec, label_form
385.
386.
387.def padding_SSLM(image, label, labels_sec, label_form):
388.    """This function pads 30 frames at the begining and end of an image"""
389.    padding_factor = 50
390.
391.    # Pad SSLM
392.    pad_image = np.full((image.shape[1], padding_factor), 1)
393.    pad_image = pad_image[np.newaxis, :, :]
394.    S_padded = np.concatenate((pad_image, image), axis=-1)
395.    S_padded = np.concatenate((S_padded, pad_image), axis=-1)
396.    return S_padded, label, labels_sec, label_form
```

```
397.
398.
399.def normalize_image(image, label, labels_sec, label_form):
400.    """This function normalizes an image"""
401.    image = np.squeeze(image)  # remove
402.
403.    def normalize(array):
404.        """This function normalizes a matrix along x axis (frequency)"""
405.        normalized = np.zeros((array.shape[0], array.shape[1]))
406.        for i in range(array.shape[0]):
407.            normalized[i, :] = (array[i, :] - np.mean(array[i, :])) / np.std(array[i, :])
408.        return normalized
409.
410.    image = normalize(image)
411.    # image = (image-np.min(image))/(np.max(image)-np.min(image))
412.    image = np.expand_dims(image, axis=0)
413.    return image, label, labels_sec, label_form
414.# endregion
415.
416.
417.# Load MLS and SSLM Data
418.class BuildDataloader(k.utils.Sequence):
419.    def __init__(self, images_path, label_path=DEFAULT_LABELPATH, transforms=None, batch_size=32,
    end=-1, reshape=True):
420.        self.songs_list = []
421.        self.images_path = images_path
422.        self.images_list = []
423.        self.labels_path = label_path
424.        self.labels_list = []
425.        self.labels_sec_list = []
426.        self.labels_form_list = []
427.        self.batch_size = batch_size
428.        self.n = 0
429.        self.reshape = reshape
430.
431.        print("Building dataloader for " + self.images_path)
432.        cnt = 1
433.        for (im_dirpath, im_dirnames, im_filenames) in os.walk(self.images_path):
434.            for f in im_filenames:
435.                if f.endswith('.npy'):
436.                    self.songs_list.append(os.path.splitext(f)[0])
437.                    # print("Reading file #" + str(cnt))
438.                    img_path = im_dirpath + f
439.                    image = np.load(img_path, allow_pickle=True)
440.                    if image.ndim == 1:
441.                        raise ValueError("Erroneous file:", img_path, "Shape:", image.shape,
    image.ndim)
442.                    else:
443.                        # image = resize(image, (300, 500))
444.                        # image = (image - image.mean()) / (image.std() + 1e-8)
445.                        if reshape:
446.                            image = np.mean(image, axis=0)
447.                        else:
448.                            image1 = np.mean(image, axis=0)
449.                            image2 = np.var(image, axis=0)
450.                            image = np.array([image1, image2])
451.                    self.images_list.append(image)
452.                    cnt += 1
453.                    if end != -1:
454.                        if cnt == end + 1:
455.                            break
456.        lbls_seconds, lbls_phrases, lbl_forms =
    du.ReadLabelSecondsPhrasesFromFolder(lblpath=self.labels_path, stop=cnt)
457.        self.labels_list = lbls_phrases
458.        self.labels_sec_list = lbls_seconds
459.        self.labels_form_list = lbl_forms
460.        self.transforms = transforms
461.        self.max = self.__len__()
462.
```

```
463.     def __len__(self):
464.         return len(self.images_list)
465.
466.     def __getitem__(self, index):
467.         # print("LEN: " + str(self.max) + " TRU LEN: " + str(len(self.images_list)) + " INDX: " +
    str(index))
468.         image = self.images_list[index]
469.         # print(image.shape, image.ndim)
470.         # print(image)
471.         # if image.ndim == 1:
472.         #     print(image)
473.         if self.reshape:
474.             image = image[np.newaxis, :, np.newaxis]
475.         labels = self.labels_list[index]
476.         # print("Labels: ", str(len(labels)), "Images: ", str(len(image)), image.shape)
477.         labels_sec = self.labels_sec_list[index]
478.         labels_form = self.labels_form_list[index]
479.         song_name = self.songs_list[index]
480.         if self.transforms is not None:
481.             for t in self.transforms:
482.                 image, labels, labels_sec, labels_form = t(image, labels, labels_sec, labels_form)
483.         return image, [labels, labels_sec, labels_form, song_name]
484.
485.     def __next__(self):
486.         if self.n >= self.max:
487.             self.n = 0
488.         result = self.__getitem__(self.n)
489.         self.n += 1
490.         return result
491.
492.     def getNumClasses(self):
493.         return len(self.labels_form_list[1])
494.
495.     def getLabels(self):
496.         return self.labels_form_list
497.
498.     def getImages(self):
499.         return self.images_list
500.
501.     def getCurrentIndex(self):
502.         return self.n
503.
504.     def getSong(self, index):
505.         return self.songs_list[index]
506.
507.     def getFormLabel(self, index):
508.         return self.labels_form_list[index]
509.
510.     def getDuration(self, index):
511.         return self.labels_sec_list[index][-1]
512.
513.
514.def get_midi_dataframe(building_df=False):
515.     df = pd.DataFrame(columns=['spectral_contrast_mean', 'spectral_contrast_var'])
516.     if building_df:
517.         df2 = pd.DataFrame(columns=['chroma_stft_mean', 'chroma_stft_var',
518.                                     'chroma_cqt_mean', 'chroma_cqt_var',
519.                                     'chroma_cens_mean', 'chroma_cens_var',
520.                                     'mel_mean', 'mel_var',
521.                                     'mfcc_mean', 'mfcc_var',
522.                                     'spectral_bandwidth_mean', 'spectral_bandwidth_var',
523.                                     'spectral_centroid_mean', 'spectral_centroid_var',
524.                                     'spectral_flatness_mean', 'spectral_flatness_var',
525.                                     'spectral_rolloff_mean', 'spectral_rolloff_var',
526.                                     'poly_features_mean', 'poly_features_var',
527.                                     'tonnetz_mean', 'tonnetz_var',
528.                                     'zero_crossing_mean', 'zero_crossing_var',
529.                                     'tempogram_mean', 'tempogram_var',
530.                                     'fourier_tempo_mean', 'fourier_tempo_var'])
```

```
        df = pd.concat([df, df2], axis=1)
    return df


def get_audio_features(df, cnt, mid_path, building_df=False):
    X, sample_rate = librosa.load(mid_path, res_type='kaiser_fast', duration=3, sr=44100, offset=0.5)
    contrast = librosa.feature.spectral_contrast(y=X, sr=sample_rate)
    """ Plot spectral contrast
    plt.figure(figsize=(10, 4))
    librosa.display.specshow(contrast, cmap='plasma', x_axis='time')
    plt.colorbar()
    plt.ylabel('Frequency bands')
    plt.title('Spectral contrast')
    plt.tight_layout()
    plt.show()
    """
    contrast = np.mean(contrast, axis=0)
    contrast2 = np.var(contrast, axis=0)
    if building_df:
        chroma_cens = librosa.feature.chroma_cens(y=X, sr=sample_rate)
        chroma_cqt = librosa.feature.chroma_cqt(y=X, sr=sample_rate)
        chroma_stft = librosa.feature.chroma_stft(y=X, sr=sample_rate)
        mel_spec = librosa.feature.melspectrogram(y=X, sr=sample_rate)
        mfcc_spec = librosa.feature.mfcc(y=X, sr=sample_rate)
        spec_bdwth = librosa.feature.spectral_bandwidth(y=X, sr=sample_rate)
        spec_centrd = librosa.feature.spectral_centroid(y=X, sr=sample_rate)
        spec_flatns = librosa.feature.spectral_flatness(y=X)
        spec_rolloff = librosa.feature.spectral_rolloff(y=X, sr=sample_rate)
        poly_feat = librosa.feature.poly_features(y=X, sr=sample_rate)
        tonnetz = librosa.feature.tonnetz(y=X, sr=sample_rate)
        zero_cross = librosa.feature.zero_crossing_rate(y=X)
        tempogram = librosa.feature.tempogram(y=X, sr=sample_rate)
        fouriertemp = librosa.feature.fourier_tempogram(y=X, sr=sample_rate)  # Not used in model,
    repurpose for others?

        df.loc[cnt] = [contrast, contrast2,  # 0, 1
                       np.mean(chroma_cens, axis=0), np.var(chroma_cens, axis=0),  # 2, 3
                       np.mean(chroma_cqt, axis=0), np.var(chroma_cqt, axis=0),  # 4, 5
                       np.mean(chroma_stft, axis=0), np.var(chroma_stft, axis=0),  # 6, 7
                       np.mean(mel_spec, axis=0), np.var(mel_spec, axis=0),  # 8, 9
                       np.mean(mfcc_spec, axis=0), np.var(mfcc_spec, axis=0),  # 10, 11
                       np.mean(spec_bdwth, axis=0), np.var(spec_bdwth, axis=0),  # 12, 13
                       np.mean(spec_centrd, axis=0), np.var(spec_centrd, axis=0),  # 14, 15
                       np.mean(spec_flatns, axis=0), np.var(spec_flatns, axis=0),  # 16, 17
                       np.mean(spec_rolloff, axis=0), np.var(spec_rolloff, axis=0),  # 18, 19
                       np.mean(poly_feat, axis=0), np.var(poly_feat, axis=0),  # 20, 21
                       np.mean(tonnetz, axis=0), np.var(tonnetz, axis=0),  # 22, 23
                       np.mean(zero_cross, axis=0), np.var(zero_cross, axis=0),  # 24, 25
                       np.mean(tempogram, axis=0), np.var(tempogram, axis=0),  # 26, 27
                       np.mean(fouriertemp, axis=0), np.var(fouriertemp, axis=0)]  # 28, 29
    else:
        df.loc[cnt] = [contrast, contrast2]
    return df


# Load MIDI Data
class BuildMIDIloader(k.utils.Sequence):
    def __init__(self, midi_path, label_path=DEFAULT_LABELPATH,
                 transforms=None, batch_size=32, end=-1, reshape=True, building_df=False):
        self.songs_list = []
        self.midi_path = midi_path
        self.midi_list = pd.DataFrame()
        self.labels_path = label_path
        self.labels_list = []
        self.labels_sec_list = []
        self.labels_form_list = []
        self.batch_size = batch_size
        self.n = 0
        self.reshape = reshape
```

```
599.
600.        print("Building dataloader for " + self.midi_path)
601.        df = get_midi_dataframe(building_df)
602.        cnt = 1
603.        audio_extensions = ["3gp", "aa", "aac", "aax", "act", "aiff", "alac", "amr", "ape", "au",
    "awb", "dct",
604.                            "dss", "dvf", "flac", "gsm", "iklax", "ivs", "m4a", "m4b", "m4p", "mmf",
    "mp3", "mpc",
605.                            "msv", "nmf", "ogg", "oga", "mogg", "opus", "ra", "rm", "raw", "rf64",
    "sln", "tta",
606.                            "voc", "vox", "wav", "wma", "wv", "webm", "8svx", "cda", "mid", "midi",
    "MID", "mp4"]
607.        for (mid_dirpath, mid_dirnames, mid_filenames) in os.walk(self.midi_path):
608.            for f in mid_filenames:
609.                if f.endswith(tuple(audio_extensions)):
610.                    self.songs_list.append(os.path.splitext(f)[0])
611.                    print("Reading file #" + str(cnt))
612.                    mid_path = mid_dirpath + f
613.                    # print("Working on file: " + f)
614.                    df = get_audio_features(df, cnt-1, mid_path, building_df)
615.                    cnt += 1
616.                    if end != -1:
617.                        if cnt == end:
618.                            break
619.        # df = pd.DataFrame(df['spectral_contrast'].values.tolist())
620.        print(cnt)
621.        df = df.fillna(0)
622.        if reshape:
623.            mean = np.mean(df, axis=0)
624.            std = np.std(df, axis=0)
625.            df = (df - mean) / std
626.            df = np.array(df)
627.            df = df[:, :, np.newaxis]
628.        else:
629.            df = np.array(df)
630.        self.midi_list = df
631.        lbls_seconds, lbls_phrases, lbl_forms =
    du.ReadLabelSecondsPhrasesFromFolder(lblpath=self.labels_path, stop=cnt)
632.        self.labels_list = lbls_phrases
633.        self.labels_sec_list = lbls_seconds
634.        self.labels_form_list = lbl_forms
635.        self.transforms = transforms
636.        self.max = self.__len__()
637.
638.    def __len__(self):
639.        return self.midi_list.shape[0]
640.
641.    def __getitem__(self, index):
642.        mid = self.midi_list[index]
643.        labels = self.labels_list[index]
644.        labels_sec = self.labels_sec_list[index]
645.        labels_form = self.labels_form_list[index]
646.        song_name = self.songs_list[index]
647.        if self.transforms is not None:
648.            for t in self.transforms:
649.                mid, labels, labels_sec, labels_form = t(mid, labels, labels_sec, labels_form)
650.        return mid, [labels, labels_sec, labels_form, song_name]
651.
652.    def __next__(self):
653.        if self.n >= self.max:
654.            self.n = 0
655.        result = self.__getitem__(self.n)
656.        self.n += 1
657.        return result
658.
```