AlgoRemix: Algorithmically Remixing Songs Using Neural Networks and My Own Voice

Charles Coppieters ‘t Wallant

Advisor: Dr. Robert Fish

Submitted in partial fulfillment

of the requirements for the degree of

Bachelor of Science and Engineering

Department of Computer Science

Princeton University

April 2023

I hereby declare that I am the sole author of this thesis. I authorize Princeton University to lend this thesis to other institutions or individuals for the purpose of scholarly research.

I further authorize Princeton University to reproduce this thesis by photocopying or by other means, in total or in part, at the request of other institutions or individuals for the purpose of scholarly research.

Finally, I pledge my honor that I have not violated the honor code and that all work in this thesis is my own.

Table Of Contents

1 Introduction

When OpenAI released DALLE-2 in November of 2022, the world of art was instantly changed forever. Almost overnight, anybody with an internet connection and a smartphone could generate hyper-realistic images, paintings, drawings, 3D renders, and graphic designs of whatever they could think of. An Andy Worhol painting of a french bulldog wearing glasses?

Easy.

How about a realistic image of an armchair in the shape of an avocado?

Done.

Within days, social media was overtaken with people posting their unique creations, and YouTube videos were released comparing the images of DALLE-2 to those of professionally employed graphic designers. As the technology continued to gain popularity and momentum, people started questioning the ethics of a technology like this. Was it going to steal jobs from visual artists? Would visual art no longer be a *human* form of expression? Despite the ethical dilemmas and all the hype surrounding this technological advancement, the fact still remained: computers now had the ability to create art in a meaningful way.

Since then, AI generated art has expanded far beyond DALLE-2's text-to-image capabilities. The next big viral phenomenon came from OpenAI's ChatGPT, which made it possible to generate text from text-prompts. This opened the door for AI to create other forms of art such as poems, literature, screenplays, and any other textual forms of art. Soon after came crude attempts at AI-generated video on platforms such as Synthesia and Elai.io. Similar tools came out for architecture models, design, and theater production. However, the one form of art that has yet to be successfully generated by AI in a consumer-usable way is music.

1.1 Motivation

Because music is one of the last forms of non-physical art that the general public cannot easily generate using AI yet, my motivation for this paper is to make AI-generated music in a way that people will want to realistically use. Although AI generated music does exist, most current forms of AI generated music focus on creating every element of

completely new songs. However, I strongly believe that AI generated music will find its niche in altering existing music or as a tool to generate specific elements of music instead of entire songs at once. For example, a model could be used to generate a new instrumental for an existing song's vocals or be used to alter the drums on a pop song to make them more jazzy. The reason behind this motivation is that, similar to other AI generation tools, creating a tool that allows more people to create is more valuable than a model that does all the creating on its own. We do not need more music, we need to make it easier for people to channel their creativity and create music.

1.2 Background

For the purpose of understanding the datasets and data preprocessing that I will use in this thesis, I will define a few important music theory concepts. The most important concept to understand is the concept of beats and measures. Beats and measures make up the most basic building blocks of timing in music. The image below depicts one measure that is split up into four beats, with each black dot representing one beat.

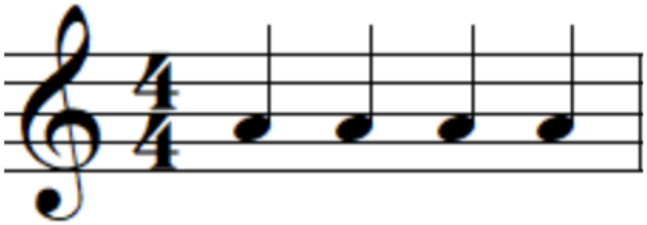

Figure 1: Visualization of a Measure With Four Beats.

Measures can theoretically be split up into any subset of beats, which is called the time signature. Time signatures are used to specify how many beats are contained in each

measure and which note value is equivalent to one beat. Time signatures are represented by a set of numbers, one on top of the other, resembling a fraction with no dividing line. The top number indicates how many beats are in a measure, while the bottom number indicates what type of note gets the beat. For example, in the 4/4 time signature depicted in the image above, there are four beats per measure and one quarter note equals one beat. Most modern music and all the music I use in this thesis has a 4/4 time signature, meaning that the measure is split into four beats. From there, each beat can be further subdivided into smaller and smaller sub-beats, which in 4/4 time are called eighth-notes, sixteenth-notes, and so on as they get smaller and smaller. The main concept to understand from this is that music has its own proprietary timing structure, which will be used to define when notes should be played after being generated by the model.

The next music theory concept to understand is the various properties of a note. At the most basic level, every note in a piece of music can be defined by its pitch, its duration, and its relative offset within a measure. Pitch refers to the highness or lowness of a sound. This is determined by the frequency of the sound, with high frequencies corresponding to higher pitches and low frequencies corresponding to lower pitches. However, the important factor to understand is that the pitch is essentially the note being played. Duration refers to how long the note is played. Referencing back to the explanation of beats and measures, duration explains exactly how many beats a note should last. Relative offset within a measure refers to where in the measure a note begins to play. The combination of pitch, duration, and relative offset make up all the basic aspects of a note.

The next music theory concept to understand is the concept of the key of a song. The key provides a framework for the chords that will be used in the song, and in turn, the notes that will sound good in context of that key. Theoretically, advanced musicians are able to make any note work in the context of any key using tension and resolution. But most music relies on playing a certain subset of notes in relation to the key to sound appealing to a general listener. For the purpose of the model I am suggesting in this thesis, I will operate under the assumption that notes generated will aim to be a note from the given key. Each key also has a specific set of chords that are based on the notes of the key's scale. These chords are called the diatonic chords. For example, in the key of C major, the notes of the scale are C, D, E, F, G, A, and B. Using these notes, we can create diatonic chords by building chords based on every other note of the scale. The chords that naturally occur in the key of C major are:

- C major (C, E, G)
- D minor (D, F, A)
- E minor (E, G, B)
- F major (F, A, C)
- G major (G, B, D)
- A minor (A, C, E)
- B diminished (B, D, F)

These chords are called diatonic chords because they are built using only the notes of the C major scale. Other chords, called non-diatonic chords, can also be used in the key of C major, but they will not be as harmonically stable as the diatonic chords. By using these diatonic chords in different combinations, a composer or songwriter can create melodies

and harmonies that fit within the tonality of the key. Understanding the chords in the context of a key is essential for understanding how music works harmonically and for creating music that sounds cohesive and tonally coherent.

Building off of the idea of keys and chords, another more complicated but equally important aspect of generating notes is the harmonic context of the note. The harmonic context refers to the chords and other notes that are being played at the same time as the note in question. This can have a significant impact on how a note is perceived by a listener because it can affect the overall sound and mood of a piece of music. The harmonic context is related to the concept of a key because notes played together generally should follow the key of the song and the chords being played at the given moment. Otherwise, if notes are not played in the proper harmonic context they can sound dissonant or unresolved which is generally less enjoyable for the average listener (again, there are exceptions for this at a higher level but for the purpose of this thesis I will operate under the assumption that notes should be played within the proper harmonic context).

The last piece of background that is important to understand is the different ways that music can be stored on a computer that allows a computer to take into account all of the important music theory concepts discussed above. The main way that music is consumed by users on our everyday devices is through audio files (MP3, WAV, FLAC). These files contain the waveform of the music which captures and plays the sound of an actual recorded performance. These files can be played on their own on essentially any device, although they do not contain any information or analysis about what notes are being played, at what time, with what rhythm, with what intensity, etc. A musician with a

trained ear may be able to determine each of those features given enough time, but from a computer standpoint these audio files do not provide much information that allows the computer to understand the key, chords, harmonic context, etc, of the given piece of music[1].

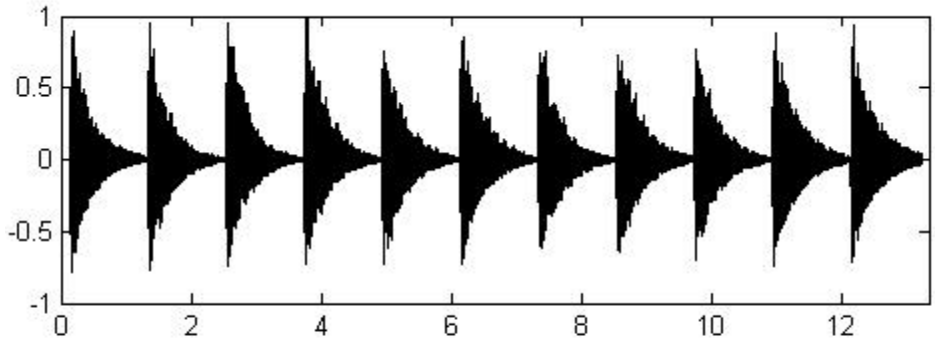

Figure 2: Visual representation of an audio file[2].

Meanwhile, MIDI is another way that music can be stored on a computer. MIDI files were invented in the early 1980s and aimed to capture the actual notes of a recorded performance. While audio files store the actual sound of a piece of recorded music, MIDI files store the "recipe" needed to recreate the performance at any time on a computer by storing each note, its length, and its velocity. These files can then be uploaded to various digital audio platforms such as GarageBand, Ableton, or LogicPro to play the MIDI file using virtual instruments[3]. The main drawback of MIDI files is that they require a sound engine or software plugin to generate a sound, while audio files can play all on their own. However, the advantage of this is that a collection of notes stored in a MIDI file can be played by any software instrument, meaning that you can record a file by playing a software saxophone and then play that same collection of notes back using any

[1] *What's the difference between MIDI and audio?* (n.d.). Roland.
[2] Catteau, B. (2021, May). *Fig. 14. The input wave-file of MIDI to WAV*. ResearchGate.
[3] *What's the difference between MIDI and audio?* (n.d.). Roland.

instrument or sound you can think of. On top of this, having access to the specific notes being played and the timing of the note sequence allows a computer to understand the various music theory concepts discussed above, such as key, rhythm, relative offset, and harmonic context. Because of these advantages, I will mainly focus on working with MIDI files in this thesis.

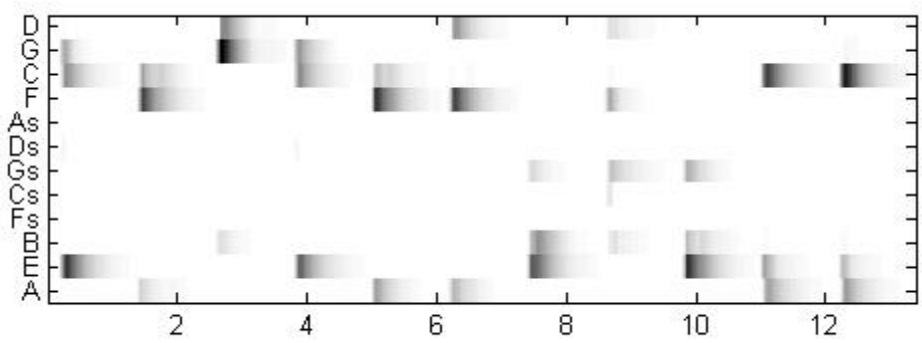

Figure 3: Visual representation of a MIDI file[4].

1.3 Goal

Based on my belief that AI music should be used to alter existing music instead of creating completely new music, in this paper I will present a machine learning model that aims to generate a remix of a song inputted by the user in five different genres: bossa nova, techno, disco, polka, and bluegrass. My model will analyze the input song for its main musical features such as beats-per-minute, measures, key, and chord progression. It will then use this information to generate a new bassline and chords as MIDI files in the chosen genre and generate lyrics matching those of the input song sung by another model trained on my own voice.

[4] Catteau, B. (2021, May). *Fig. 14. The input wave-file of MIDI to WAV*. ResearchGate.

2 Related Work

Although AI generated art has only gained significant momentum and popularity in the past few months, the idea of computers generating music dates back to the first computers in the mid-20th century. Starting in 1951, British mathematician Alan Turing and schoolteacher Christopher Strachey built the first music-programmable computer at the Computing Machine Laboratory in Manchester, England. The computer was programmed with basic melodies like "God Save the King" and "Baa, Baa, Blacksheep", which were recorded and broadcast by BBC[5]. Following this, the first computer to actually generate music instead of being programmed to play music came a few years later in 1957 when composer Lejaren Hiller and mathematician Leonardo Isaacson teamed up to program the Illinois Automatic Computer ILLIAC I to generate music. They used random number generation, probability theory, and hard-coded music theory rules to generate a four-movement classical piece for string quartets titled "Illiac Suite". This piece was outputted as sheet music and was played live by a string quartet at the University of Illinois[6].

Since then, computer-generated music has evolved significantly beyond the simple probabilistic models and random note generation that existed back then. Teams at

[5] Rice, D., & Galbraith, M. (2008, November 16). *Alan Turing: How His Universal Machine Became a Musical Instrument*. IEEE Spectrum.
[6] Hiller, L. (1959). *Experimental music; composition with an electronic computer*. New York, McGraw-Hill.

Google, Microsoft, Tencent, and OpenAI have made major strides in algorithmically generating music with neural networks, transformers, diffusion models, and a variety of other machine learning algorithms with the goal of teaching these machines music theory by feeding them huge amounts of music data.

2.1 Music Generation with Neural Networks

On a theoretical level, music and neural networks are essentially a perfect match. Music is a sequence-based art form, where notes, chords, and beats are arranged in time to create songs. Meanwhile, neural networks work best with sequential data because they can be used to predict the next data point based on an input sequence of data points. Combining these two is a no-brainer, which is why many of the most successful attempts at algorithmic music generation using modern machine learning techniques have been using neural networks of some kind. The main major papers released in this space are OpenAI's Jukebox and Google's Magenta. Each of these experiments had a slightly different approach and wildly different results, but the combination of both have given me a detailed understanding of what is possible at this point by applying neural networks to music generation.

OpenAI's Jukebox was a project that was announced in 2019 with the goal of creating music of all forms and genres, mainly focusing on instrumental tracks. The first iteration of this project was called MuseNet which created music by generating MIDI files. It was a deep neural network that could generate four-minute musical compositions with up to ten different instruments in styles from country to Mozart to the Beatles. It

used a slightly modified version of the architecture used by OpenAI's GPT-2, which was a large-language model using neural network transformers to predict the next word in a text-sequence[7]. MuseNet updated this architecture (a Sparse Transformer training a 72-layer network with 24 full attention heads over a context of 4096 tokens) to work with MIDI data instead of textual English in a way that, given the first few notes of a melody, the model would predict the following note. This process would be repeated over and over until an entire song was made. One of the interesting approaches that MuseNet had was how the training data was structured. Trained on a large corpus of MIDI-based datasets (ClassicalArchives, BitMidi, and MAESTRO) that covered many different styles of music, the training data was structured in a way that combined MIDI note data (pitch, volume, etc) with instrumentation and composer tags so that outputs of the model could be specified with those tags as well. This supposedly allowed the model to understand patterns specific to each genre of music it was being fed[8]. Although the technical architecture of the MuseNet approach was novel, the main feature I drew from this paper to consider in my own approach was the focus on labeling training data by genre and considering the specific distinctions between genres. No matter the approach, trying to code a model to generate "music" would always be infinitely less successful than trying to generate genre-specific models because every genre has very specific note structures, chords, rhythms, and harmonic contexts that can be learned by a model.

OpenAI's next attempt at algorithmic music generation was their Jukebox project, which aimed to train and output audio files (44kHz raw audio) instead of MIDI. Trained using a dataset of 1.2 million raw audio files, this project aimed to generate both

[7] OpenAI. (2019, November 5). *GPT-2: 1.5B release*. OpenAI.
[8] Payne, C. (2019, April 25). *MuseNet*. OpenAI.

instrumental and vocal tracks using the modified code of a large-scale image generation architecture known as VQ-VAE-2[9]. This modified architecture used a technique where the training audio was compressed to lower-dimensional space in order to discard perceptually irrelevant bits of information in the audio waveform. The model then trained and generated audio in this lower-dimensional space and then upsampled the outputs back to 44kHz raw audio[10]. The outputs from this model sound very impressive compared to the MIDI approach of MuseNet. However, one of the major shortcomings is the lack of long-term structure. The songs outputted by the model do not follow any discernible pattern that we would come to expect in modern music such as having a repeating chorus, a bridge, a build-up, etc. The outputs don't seem to follow any normal chord progressions either. Although these structures are by no means a requirement for music, most modern music listened to by the average consumer does have some kind of repeating structure and the model does not have this long-term structure in a single example output. Because of this, the main conclusion I got from the Jukebox approach was the importance and difficulty of including long-term structure in generated music.

The first project to specifically target the long-term structure issue was Google's Magenta project, an open-source project started by the Google Brain team. The Magenta project developed a range of music-related tools powered by machine learning, one of which was an AI-music generator released in February of 2022. It used a neural network transformer with the main goal of requiring the model to understand recurring elements such as phrases, verses, choruses, etc. To do this, they decided to use a self-attention model, which is an autoregressive model that references its own previous outputs at every

[9] Razavi, A. (2019, June 2). *Generating Diverse High-Fidelity Images with VQ-VAE-2*. NIPS papers.
[10] Dhariwal, P. (2020, April 30). *Jukebox: A Generative Model for Music*. OpenAI.

step of generation. A normal recurrent neural network has a fixed size memory to store elements to be referenced, which limits how much of the former output can be referenced. The self-attention model used by Magenta was structured with multiple successive layers of transformer decoders which allowed the model to reference significantly more of its former output at generation time. According to the paper, using this architecture allowed them to output songs with a long-term structure at the scale of 60 seconds, far beyond what any former attempts had been able to achieve[11]. Although this is quite impressive, this paper demonstrated a very significant pivot in my research for this thesis. Although my original plan was to attempt to generate completely new songs from start to finish using neural networks and transformers, I realized that the long-term structure issue would hold me back significantly. If teams of researchers at Google Brain could not solve it beyond 60 seconds, which is not even close to the length of a song, then it was very unlikely that I would be able to generate new music with long-term structure using neural networks with any kind of long-term structure. Based off of these papers, I decided to apply AI music generation to alter existing songs instead.

2.2 Music Generation with Diffusion Models

After realizing that neural networks presented a major issue with long-term structure, I decided to research the other approaches out there for AI-music generation. As I was writing this thesis from October 2022 to early 2023, a rapid shift in architecture approach happened towards using diffusion models instead of neural networks for music

[11] Hawthorne, C. (2022, February 15). *General-purpose, long-context autoregressive modeling with Perceiver AR*. arXiv.

generation. Diffusion models are generative models and power many AI products today, most notably the DALLE-2 image generator that introduced the consumer world to generative art described in the introduction of this thesis. Inspired by non-equilibrium thermodynamics, a diffusion model is a generative model which essentially destroys training data by repeatedly adding Gaussian noise before reversing the denoising process and learning how to recover the original data. Once trained, the diffusion model can generate data by passing randomly sampled noise back through the denoising process to create a completely new output[12]. For music, it immediately became apparent that this kind of model could work well on audio files because it was possible to add noise to them in a very similar way that noise was added to image data for DALLE-2. The main major papers released in this space came from AI labs at Tencent and Google. Each of these had different approaches but, again, gave me significant insight into what may be possible using diffusion models to generate music.

The earliest and most frequently referenced application of diffusion models to audio generation came from the AI labs at Tencent with a model called Diffsound. Released in July 2022, this paper used discrete diffusion models to generate audio samples from text inputs such as “an engine idling consistently before sputtering some” or “a dog barks and whimpers”. Although this is not directly related to music generation, the diffusion architecture proposed in this paper inspired every following paper and the use of diffusion models for sound in the first place. On a technical level, Diffsound used a non-autoregressive decoder based on the discrete diffusion model. It predicted all of the mel-spectrogram tokens of the prompted audio in one step and then refined the predicted

[12] O'Connor's, R., & O'Connor, R. (2022, May 12). *Introduction to Diffusion Models for Machine Learning*. AssemblyAI

tokens in the next step to get an output[13]. This paper was very interesting because it laid the foundation of a new kind of sound generation technique on a user level: text inputs. Using diffusion models, future approaches could make use of the architecture to allow users to request specific genres, rhythms, and other features of music theory.

The team at Google Research did exactly this with their diffusion model-based approach at music generation called Noise2Music. Released in February 2023, this model pushed the envelope by applying the techniques and architecture proposed by Diffound specifically to music, allowing potential users to generate music from text prompts. Noise2Music had 6 main attributes (genre, instrument, tempo, mood, vocal traits, and era) which could be defined in the prompt. An example prompt could be as specific as this: "A female vocalist sings this upbeat Latin pop. The song has an upbeat rhythm with a dance groove. The drumming is lively, the percussion instruments add layers and density to the music, the bass line is simple and steady, the keyboard accompaniment adds a nice melody". The dataset used to train this model was MusicCaps, which is a growing dataset of music audio files labeled with a large subset of musical features. They added to this dataset by pseudo-labeling a large unlabeled set of music audio using two deep models via zero-shot classification. The architecture of the trained model included a series of diffusion models that were applied in succession to generate the final music clip[14]. The outputs from this model were very impressive and similar in musical complexity to Jukebox and Magenta. However, the use of diffusion models did not solve long-term structure issues from neural network approaches mentioned in the last section.

---

[13] Yang, D. (2022, 20 July). *Diffsound: Discrete Diffusion Model for Text-to-sound Generation*. arXiv.
[14] Huang, Q. (2023, February 8). *Noise2Music: Text-conditioned Music Generation with Diffusion Models*. arXiv.

Despite this, the impact of this approach on my research was significant because it demonstrated the importance of allowing potential users to determine specific features of the music output using text.

3 Approach

Based on my background research and due to the continuous flow of new research papers coming out as I was writing this thesis, my approach changed significantly throughout the process of attempting to make AI-generated music in some capacity. While my original approach at the beginning of this process was simply to apply modern machine learning techniques to the creation of completely new music, I quickly realized that there was a glaring issue that no former approaches had solved: long-term structure. Neural networks could successfully generate sequences of MIDI notes that sounded musical. Neural transformers and diffusion models could successfully generate audio files that sounded like music when played in short clips. However, none of these approaches could successfully generate an actual *song*. A song has structure. A song has a chord progression. A song has sections that all relate to one another and have repetitive callbacks over long time intervals. Without this, generated audio is the musical equivalent of a run-on sentence, where new ideas keep flowing out over and over with no discernible connection to each other or structure of any kind. Seeing how this issue could not be solved in any meaningful way by teams of researchers at Google Brain with significantly more resources and experience than I have, I decided to cut my losses and figure out a different novel approach to applying modern machine learning techniques to music generation.

3.1 The Wub Machine

The approach I propose in this thesis is inspired by a website I used with my friends when I was in middle school called "TheWubMachine.com"[15]. It allows you to upload any MP3 audio file and automatically turns the input song into a dubstep remix. As you can imagine, 2013 middle school kids were all over this new technology and we used it to create dubstep versions of all our favorite songs from Daft Punk's "Get Lucky" to "Gangnam Style" by PSY (our dubstep remix was a big hit at the local beach that summer). Under the hood, the website works by chopping up the input song into small vocal fragments and overlaying them over a dubstep beat that was pre-recorded by the website's founder, Peter Sobot. For the vocal chops, he originally used a song analysis API called Echo Nest which had data on over 1 million songs and was later acquired by Spotify[16]. He used this to determine which lyrics in the song were most frequent and used those vocal chops in the generated remix outputs. Eventually, he coded his own song analysis algorithm which he has not shared publicly. For the remix instrumentals, Sobot recorded backing tracks himself that were each randomly applied to the vocal chops when a user uploaded an MP3.

Since my glory days on The Wub Machine in 2013, Sobot has added a few more styles including "Trap", "Swing", "Electro House", "Drum and Bass", and a supposedly

[15] Sobot, P. (n.d.). *The Wub Machine*. The Wub Machine.
[16] *The Echo Nest Blog — The Echo Nest Song API*. (2010, April 24). The Echo Nest Blog.

wildly popular jingle bell Christmas themed remixing style[17]. Each of these styles use a slightly different form of vocal chopping to keep the essence of the input MP3 and all backing tracks continued to be recorded by Sobot himself. For the purpose of research and proper scholarship, I of course had to try all of the new styles and I can confirm in full confidence that creating a drum and bass remix of Ice Spice is exactly as fun as it was in 2013.

3.2 Approach

Based on The Wub Machine, the approach I present in this thesis involves applying modern machine learning techniques to song remixing. Based on an input song and the choice of one of five possible genres, my model generates a remix of that song in the requested genre. As an added touch, the model generates lyrics for the remix sung by my own voice based on the lyrics of the original song. This is done in four main steps:

(1) Song Analysis: The first step of the process is to analyze the input audio file for its main music theory features including the beats per minute (BPM), the measures and beat grid, the key, and the chord progression.

(2) Instrumental Generation: The second step of the process involves generating the instrumental for the remix based on the beats per minute, measures,

[17] Khan, A. (n.d.). *Skipping Class to Build a Popular Music Remixing App*. Indie Hackers.

beatgrid, key, and chord progression analyzed above. This is done by generating a MIDI bassline and MIDI instrumental chords using LSTM neural networks.

(3) Vocal Generation: The last step of the process involves separating the vocals of the input audio file and generating new vocals using a generative adversarial network model trained on my own voice.

Based on my background research, the main recurring issue in AI-music generation is being able to successfully generate long-term structure in entire song-length intervals. This made most outputs from previous attempts at AI-music very unstructured, random, and far from following the general musical trends of modern music. The approach that I propose above is not susceptible to the same issue because my model relies on the long-term structure that remains with the lyrics, key, and chord progression drawn from the input song. In this way, my model is able to create new instrumentals that have long-term structure while also having some sense of familiarity by keeping the words of the vocals consistent to the input song (despite being generated to be sung by me). The following section will walk through the technical side of this approach, including all the algorithms and modern machine learning techniques I used to achieve my proposed approach.

4 Algorithms

4.1 Song Analysis - Spectrogram and Chromagram

The first step in the analysis of the input song involves preprocessing the audio file into a spectrogram. A spectrogram is a visual representation of sound depicting frequency and amplitude (loudness) in relation to time. These graphs are vital to basic music analysis because they give insight into how the intensity of different frequencies change over time which can show the sudden onsets of sounds[18]. The ability to understand sudden onsets of sounds in a numerical way is very important to my model because it can be used to find beats per minute and the beat grid of the input song. The graph below depicts the spectrogram for the song "Sweet Caroline" by Neil Diamond:

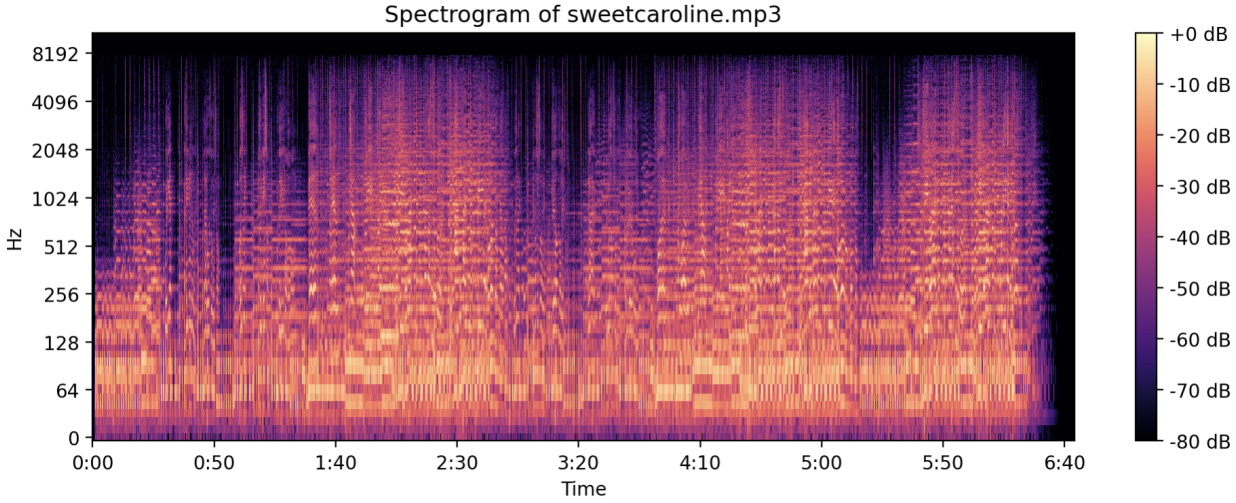

Figure 4: Spectrogram of "Sweet Caroline" by Neil Diamond.

[18] *Spectrogram View*. (2022, December 27). Audacity Manual.

The next step in the analysis of the input song involves preprocessing the audio file into a chromagram. A chromagram is a visual representation of the chroma content within a musical piece, showcasing the intensity of each of the 12 pitch classes common in Western music (C, C#, D, D#, E, F, F#, G, G#, A, A#, and B). In more simple terms, it displays how the different musical notes are distributed throughout the input audio file. These graphs are a valuable tool in music analysis because they provide insights into the harmonic structure, key, and chord progressions of a musical piece. Chromagrams can be used to analyze audio files in various ways, including music transcription, genre classification, and even emotion recognition. Researchers sometimes chromagrams to study the harmonic structure of song or to analyze the music theory techniques used by different artists[19]. In a similar vein, I utilize chromagrams as a preprocessing step to eventually allow me to derive the beats per minute, key, and chord progression of the audio files inputted to my model.

As seen in the image below, a chromagram graph is typically shown as a 2D grid, with the horizontal axis representing time and the vertical axis representing the pitch classes. Each cell in the grid corresponds to a specific pitch class at a given point in time. The intensity of the color in each cell is indicative of the presence or prominence of that particular pitch class at that specific moment. In the graph below, the red color represents a strong presence, while the blue color indicates a weaker presence or complete absence of that pitch class. This chromagram is also derived from the song “Sweet Caroline” by Neil Diamond:

[19] Verma, Y. (2021, September 19). *A Tutorial on Spectral Feature Extraction for Audio Analytics*. Analytics India Magazine.

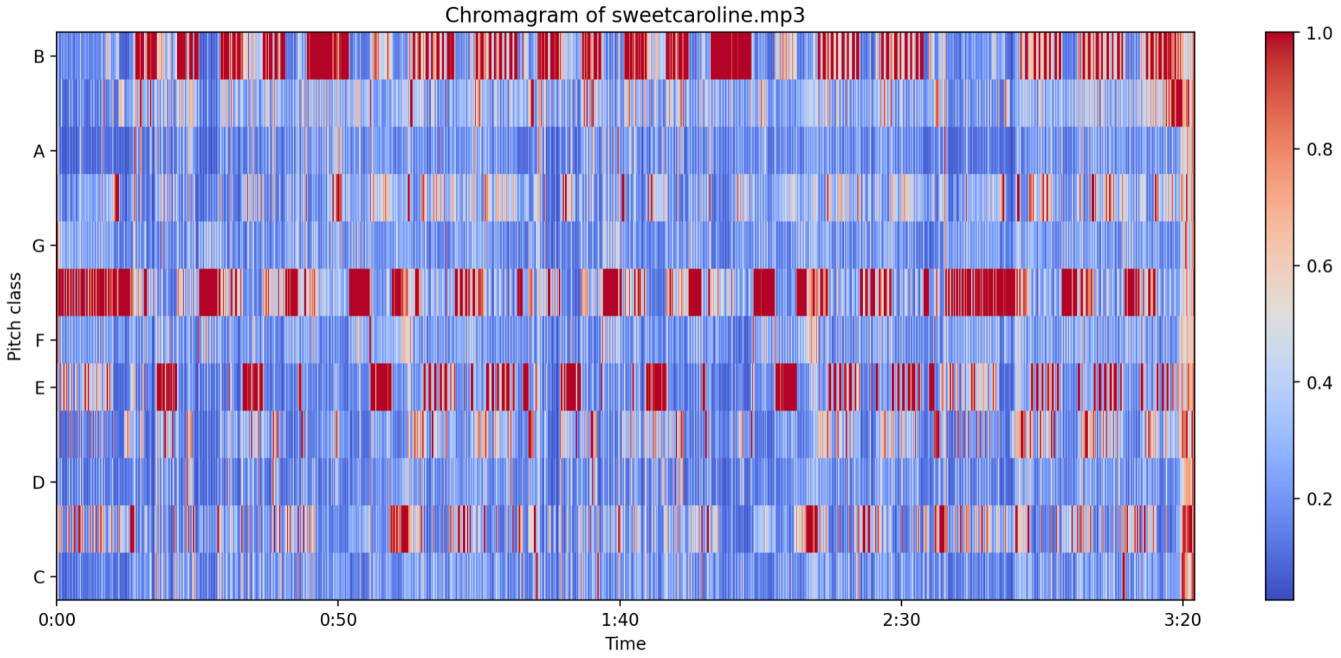

Figure 5: Chromagram of "Sweet Caroline" by Neil Diamond.

The chromagram above is computed using a process called Short-Time Fourier Transform which involves breaking down the audio signal into small segments or frames, and then analyzing the frequency content within each frame. By examining the energy of the audio signal at different frequencies, it is possible to identify which pitch classes are present and with what intensity. In my model, I use a Python package called Librosa to run the Short-Time Fourier Transform and generate the numeric values for the chromagram. Librosa is a package for music and audio processing that has various different algorithms and functions that allow for the extraction of music theory features from audio files[20]. This chromagram is vital to my model because it makes up the numerical basis of how I algorithmically determine the key and chord progression of the input song.

[20] *Librosa: Audio and Music Processing in Python*. (n.d.). Librosa.org.

4.2 Song Analysis - BPM and Beatgrid

The beats per minute (BPM) and the beatgrid of a song are important because they give an insight into the local and global timing structures of a song. The model I propose in this thesis requires an understanding of the beats per minute in order to generate the instrumental and the new vocals at the same speed as the original songs (and the same speed as each other). The beatgrid is made up of a set of markers that represent where the measures of the song are, and is required in the model because it is used in the last step to align the instrumental and vocal together.

In order to calculate the beats per minute, a four step process is required that involves (1) converting the spectrogram calculated in the last section to a Mel-scale spectrogram, (2) converting the Mel-scale spectrogram to a log-amplitude scaling, (3) calculating the onset strength envelope, and (4) using the onset strength envelope to create a self-similarity matrix that can be used to find the estimated beats per minute of the audio file.

(1) Mel-Scale Spectrogram Conversion

The Mel-scale spectrogram is a transformation of the original spectrogram that emphasizes perceptually relevant frequency bands, mimicking the human auditory system's response to the frequencies being played. This is important to take into consideration because there are a lot of frequencies that show up in the original spectrogram that the average human ear would never perceive. The Mel scale is a

perceptual scale of pitches based on the observation that human perception of frequency is non-linear and is better at lower frequencies than at higher frequencies. As a result, the Mel scale provides a more accurate representation of how we perceive sound compared to the linear frequency scale from the normal spectrogram[21]. This is calculated by applying a Mel filterbank to the normal spectrogram. The Mel filterbank (depicted in the image below) consists of a set of overlapping triangular filters, each covering a specific frequency range to match human perception.

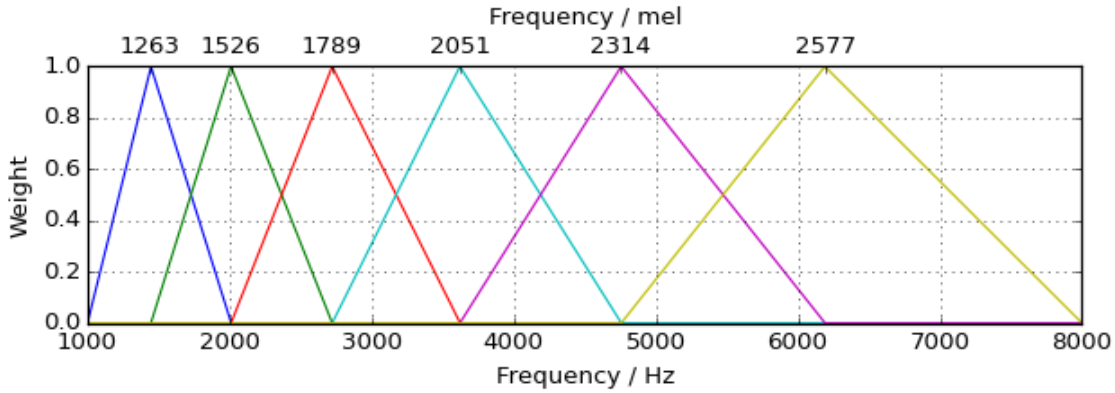

Figure 6: Visualization of the Mel filterbank[22].

The Mel-scale spectrogram is then found by summing the energy within each filter across all the frequency bands, resulting in a representation that has the same time structure as the original spectrogram, but the frequency axis is now warped according to the Mel scale. The following image shows the Mel-scale spectrogram for “Sweet Caroline” by Neil Diamond:

[21] Roberts, L. (2020, March 6). *Understanding the Mel Spectrogram - Analytics Vidhya*. Medium.
[22] Gündert, S. (2014). *Mel Filter Bank — PyFilterbank devN documentation*. GitHub Pages.

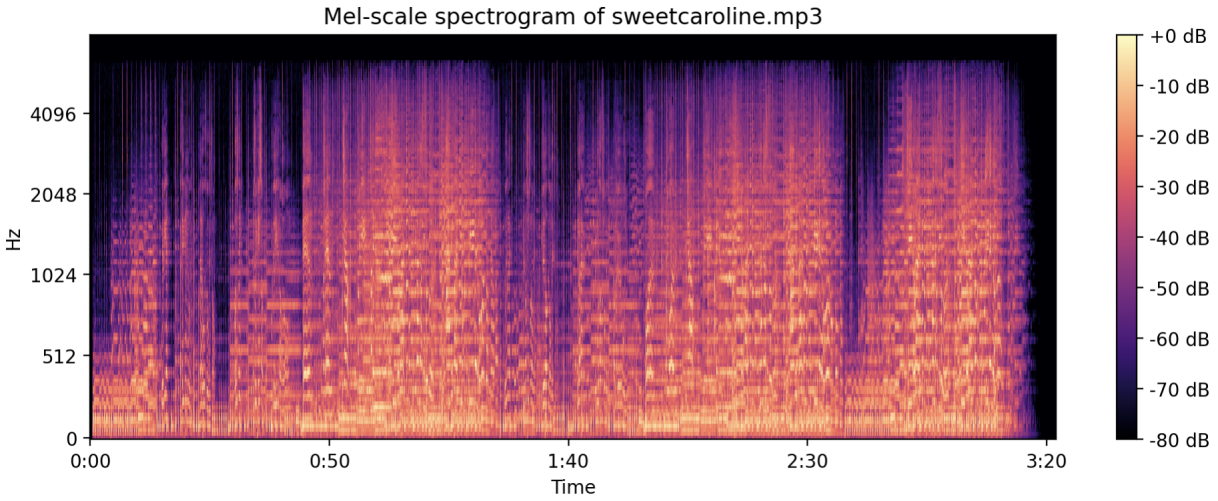

Figure 7: Mel-scale Spectrogram of "Sweet Caroline" by Neil Diamond.

This graph emphasizes the frequency bands that are most relevant to human perception to make the beats per minute estimation more accurate.

(2) Log-Amplitude Scaling Conversion

The next step in the beats per minute estimation is converting the Mel-scale spectrogram into a log-amplitude scaling. This step is the equivalent of the last step but for the amplitude (loudness) or the audio file. Humans perceive loudness in a logarithmic way, meaning that the difference in loudness between two sounds is more significant when the sounds have lower amplitudes to when they have higher amplitudes[23]. In other words, we are more sensitive to changes in amplitude at lower levels than at higher levels. Log-amplitude scaling is simply taking the log of the amplitude values in the Mel-scale spectrogram which compresses the dynamic range, emphasizing the

[23] Mantione, P. (2018, July 3). *The Fundamentals of Amplitude and Loudness — Pro Audio Files*. Pro Audio Files.

differences in amplitude at lower levels and de-emphasizing the differences at higher levels. The image below depicts the log-amplitude scaling of "Sweet Caroline" by Neil Diamond:

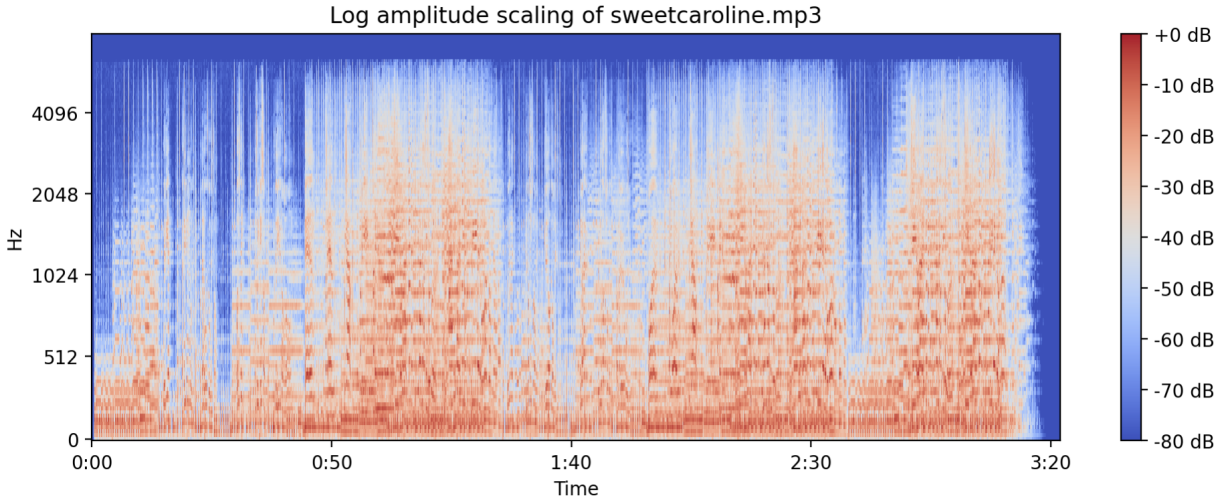

Figure 8: Log Amplitude Scaling of "Sweet Caroline" by Neil Diamond.

(3) Onset Strength Envelope Calculation

The next step in the beats per minute estimation is calculating the onset strength envelope, which identifies the significant changes in energy in the audio file. On a local scale, these significant changes in energy tend to correspond to rhythmic events like beats[24], which we can use to finally estimate beats per minute. Using the log-amplitude Mel-scale spectrogram, the onset strength envelope is calculated by computing the first order difference along the time axis, measuring the local changes in energy from one frame to the next. Next, a half-wave rectification is applied which sets all the negative

[24] Bello, J. P. (n.d.). *A Tutorial on Onset Detection in Music Signals*. School of Electronic Engineering and Computer Science.

values to zero. This is important because it means the result only measures positive increases in energy which generally apply to the start of beats[25]. Finally, the positive values of the first order difference are summed across the frequency bands which results in a one-dimensional array representing the onset strength envelope. The image below depicts the onset strength envelope of "Sweet Caroline" by Neil Diamond:

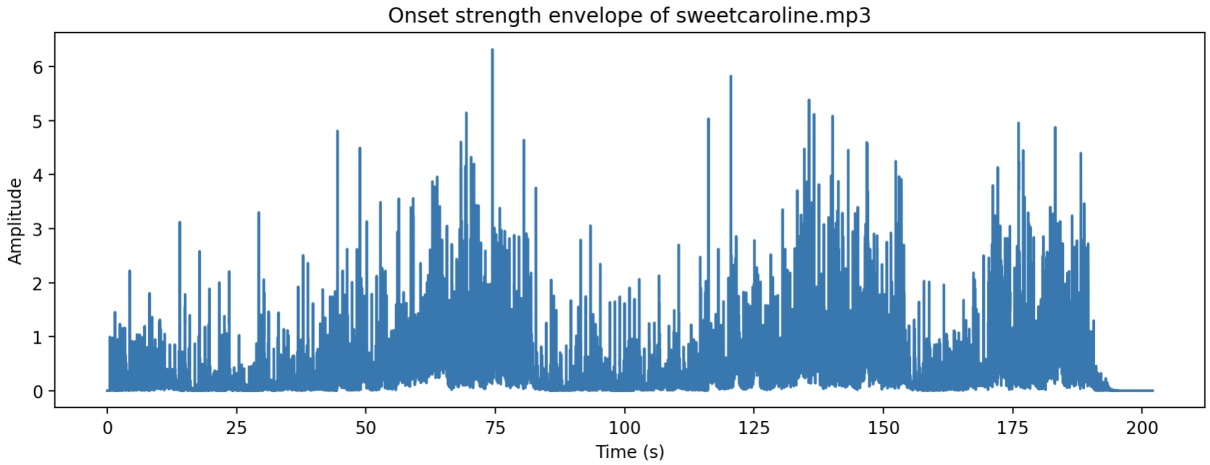

Figure 9: Onset Strength Envelope of "Sweet Caroline" by Neil Diamond.

(4) BPM Estimation

The last step is to use the onset strength envelope to run an autocorrelation algorithm that finds repeating patterns within the onset strength envelope to estimate beats per minute. First, a self-similarity matrix is computed by taking the dot product of the onset strength envelope and its shifted versions at every other time step. Next, all the elements along diagonals parallel to the main diagonal are summed together which computes the correlation between the onset strength envelope at a given point with all the

[25] Bello, J. P. (n.d.). *A Tutorial on Onset Detection in Music Signals*. School of Electronic Engineering and Computer Science.

shifted versions. High autocorrelation values indicate a strong periodic pattern at the corresponding time lag, suggesting the presence of a repeating rhythmic structure[26]. The beats per second is then estimated by finding the time interval with the highest autocorrelation value in seconds, taking the inverse of this value, and multiplying it by 60 to get the beats per minute estimation.

Once the beats per minute are estimated, finding the beat grid is relatively simple using an algorithm called the "Dynamic Bayesian Network Beat Tracker". This algorithm finds the optimal sequence of beat positions that maximizes the alignment with the onset strength envelope[27], outputting an array of beat positions that make up the beat grid. The following image depicts the beat grid calculated for "Sweet Caroline" by Neil Diamond. The estimated beats per minute for this song was 129.2, which is accurate compared to the MP3 input which stands at 129 beats per minute.

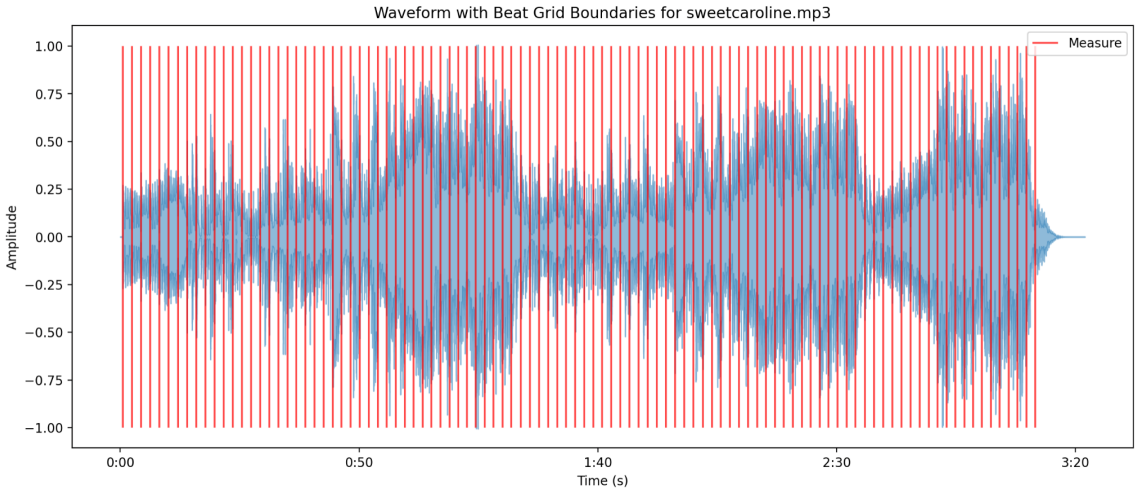

Figure 11: Beat Grid of "Sweet Caroline" by Neil Diamond.

[26] Clarke, H. D. (2005). *Autocorrelation*. ScienceDirect.
[27] Bock, S. (n.d.). *Joint Beat and Downbeat Tracking with Recurrent Neural Networks*. Institute of Computational Perception.

4.3 Song Analysis - Key

The next step in the music theory analysis of the audio file is finding the key of the input song. This part of the analysis relies on using the chromagram calculated in *Section 4.1* as an input and uses the Krumhansl-Schmukler algorithm to find the key. For the purpose of this thesis and to be able to use the Krumhansl-Schmukler algorithm, I assume that all input songs follow the twelve pitch classes seen in modern Western music (C, C#, D, D#, E, F, F#, G, G#, A, A#, and B) and have a key matching one of these pitch classes.

Operating under this assumption, a preprocessing step for the Krumhansl-Schmukler algorithm is to use the chroma values calculated from the chromagram to find how likely it is that each pitch class appears in the input audio file. This is done by calculating the average chroma value of each pitch class, which is displayed in the image below for the song “Sweet Caroline” by Neil Diamond:

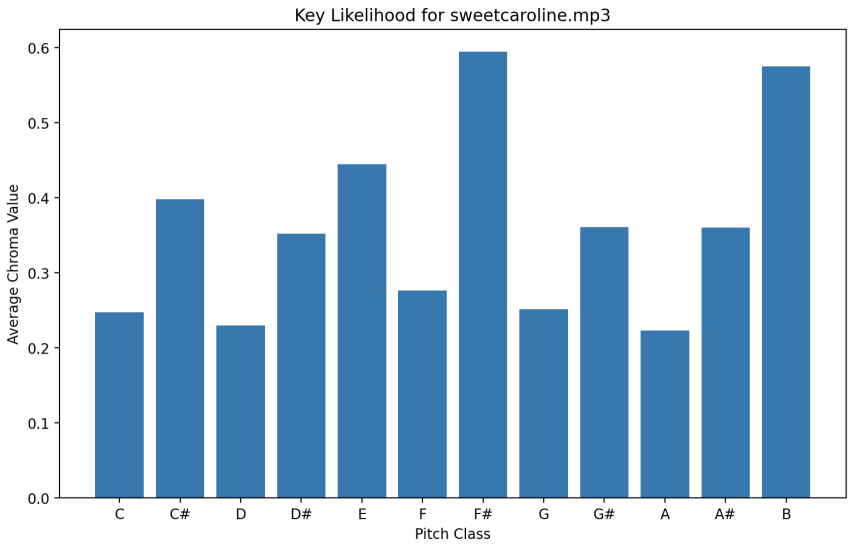

Figure 10: Key Likelihood for “Sweet Caroline” by Neil Diamond.

This visualization shows which pitch classes are most dominant in the input audio file, which in this case is F# and B. Although it would seem like this should be enough information to calculate the key (with F# just about edging out B for the most dominant position), this is not the case because simply averaging the chroma values does not take into account tonal hierarchy and other more complicated music theory concepts like overtones and relationships between pitches in the same key. This method may work by chance in certain cases but is not very accurate (hint: it's wrong for this "Sweet Caroline" audio file). Averaging chroma values also does not give insight into whether the key is major, minor, dominant, etc, which is an important feature of the key as well.

Because of this, the next step is to apply the Krumhansl-Schmukler algorithm which relates the average chroma values calculated above to predefined major and minor key profiles. The predefined key profiles are derived from statistical analyses of pitch class distributions in Western music representing the average pitch class frequency where the individual pitch classes (C, C#, D, D#, E, F, F#, G, G#, A, A#, and B) are given values from 0 to 11 respectively. The key profiles depict the average frequency of pitches in a given key, and are as follows[28]:

Major Key Profiles

Key	C	C#	D	D#	E	F	F#	G	G#	A	A#	B
Freq	6.35	2.23	3.48	2.33	4.38	4.09	2.52	5.19	2.39	3.66	2.29	2.88

[28] Hart, R. (2012, August 19). *Key-finding algorithm*. Robert Hart's Homepage.

Minor Key Profiles

Key	C	C#	D	D#	E	F	F#	G	G#	A	A#	B
Freq	6.33	2.68	3.52	5.38	2.60	3.53	2.54	4.75	3.98	2.69	3.34	3.17

Using these values as a reference, we then run the Krumhansl-Schmukler algorithm which has two steps:

(1) Normalize the chroma data (from the chromagram) by dividing each column of the chroma matrix by the sum of its elements. This ensures that the energy distribution across the twelve pitch classes sums to 1 for each time frame which removes any variability in loudness over the audio file.

(2) Calculate the correlation between the normalized chroma matrix and the major and minor key profiles. The correlation algorithm used is called circular correlation which computes the correlation between two sound signals by shifting one signal circularly with respect to the other and calculating the correlation for each shift. The maximum correlation value indicates the similarity between the two signals, demonstrating the similarity between the chroma features and the predefined key profiles to estimate the key[29]. The graph below depicts the these correlations for “Sweet Caroline” by Neil Diamond:

[29] Schmuckler, M. A. (2005, November). *(PDF) Perceptual Tests of an Algorithm for Musical Key-Finding.* ResearchGate.

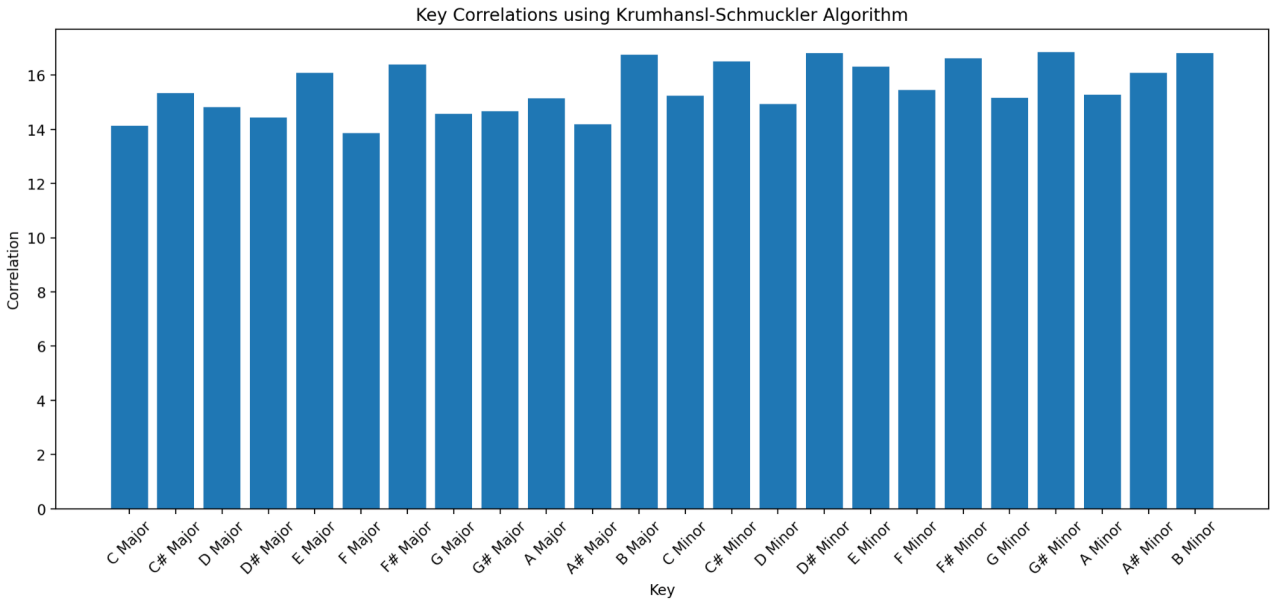

Figure 11: Key Correlations for "Sweet Caroline" by Neil Diamond.

Based on the run through of the Krumhansl-Schmukler algorithm above, the estimated key of "Sweet Caroline" is B major (which is correct).

4.4 Song Analysis - Chord Progression

The next stage of the input audio file analysis is estimating the chord progression of the song. This step relies on the use of the chromagram values and many of the techniques used to calculate the key in the last section. For this section I operate under the assumption that there is no more than one chord per measure, meaning that the output has one chord for every measure in the input song. The process to estimate the chord progression has four steps, which are as follows:

(1) This step utilizes the beat grid calculated in *Section 4.2*. The goal of this step is to sync the chroma features with the beat grid to focus on the harmonic context relevant to the specific beat being estimated for. This is done by taking the median chroma value for each pitch class within each measure interval. The median is used because it is less sensitive to outliers than the mean chroma value, meaning the estimation for that beat will be more accurate in the presence of random noise or other unrelated frequencies.

(2) For each measure, the root note of the estimated chord is calculated by calculating the highest average chroma value within the measure. This process is very similar to the first step of the key-estimation approach from *Section 4.3*. Estimating the root note of the chord is important because the root note makes up the basis for the rest of the chord for that measure.

(3) For each measure, the chord quality (major, minor, or dominant) is estimated based on the root note. This step uses a Python library called Music21 to analyze the scale degree of the chord root note in the context of the estimated key. Music21 is an open source library of functions that can be used to analyze audio files in various ways. The function that I use from the library first constructs a major scale for the estimated key then determines the scale degree of the chord root note by finding its index in the scale. Based on the scale degree and the mode of the key (major or minor), the function returns the chord quality following common harmonic patterns[30].

[30] *Music21: a Toolkit for Computer-Aided Musicology*. (n.d.). Massachusetts Institute of Technology.

(4) The last step combines the root note and the chord quality and assigns that chord to the measure being analyzed. The steps are then repeated for every measure in the input audio file.

The image below depicts the estimated chord progression for “Sweet Caroline” by Neil Diamond:

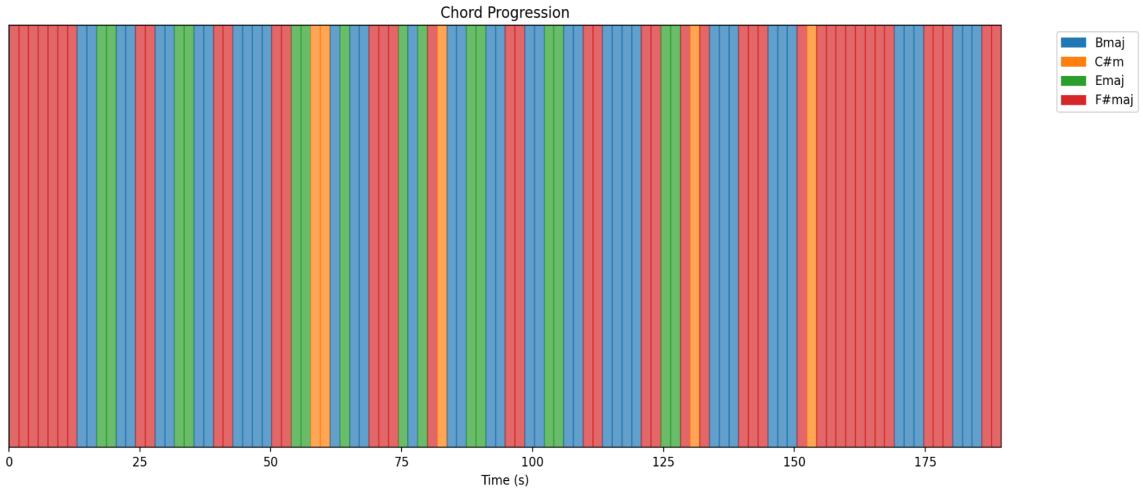

Figure 12: Estimated Chord Progression for “Sweet Caroline” by Neil Diamond.

4.5 Remix Generation - Instrumental

The first step of the remix generation process is the instrumental, which includes generating both a new bassline and new chords. In my approach, I train a neural network and generate each of these tracks individually to mimic the real world where each of these tracks are played by different instruments. This is made possible because I am adopting the harmonic context from the chord progression of the input songs, meaning that at generation time, the bassline and the chords can be generated completely separately and still sound harmonically together because they are both based on the same

chord progression. The main reason why I train and generate the bassline and chords separately is that each genre in my model has very distinct bass and chord patterns that are very different from one another. Training separate models for each instrument allows the neural network to more effectively learn the patterns of each instrument with less training, which is optimal given that training is computationally expensive and very time consuming.

Although I use slightly different approaches for my bassline and chord progression models, there is some preprocessing that is consistent for both models that I will explain before going into the specifics of each model. To start, it is important to understand what kind of training data I will be feeding into my models. All training data is made up of one measure MIDI files labeled by a chord. As a reminder, one measure is made up of four beats (not necessarily true for all music but I am assuming this for the purpose of this thesis). The reason why each training MIDI file is only one measure is because the chord progression I estimate for the input song is estimated on a measure-by-measure basis. At generation time, my models will therefore generate one measure of MIDI notes based on a given chord, which will then be consistent with the chord progression analyzed from before. As for labels, each MIDI file's name contains the label, which is a string containing the key and the mode of the chord. For example, a MIDI file which has one measure of a bassline in the key C Major would be named and labeled "cmaj.mid". Meanwhile, C Minor would be named "cm.mid" and C Dominant would be named "c7.mid". It is important to note that the key can also have accidentals such as "C#", which is the note one-half step above C, so files can be named "c#maj.mid", "c#m.mid", "c#7.mid", and so on.

The next preprocessing step that is consistent for all my instrumental models is the encoding of the chord labels. In order to feed them into a neural network, the chord label string must be turned into some kind of numerical value. Given this array of chord labels: ['a#7', 'a#m', 'a#maj', 'a7', 'am', 'amaj', 'b7', 'bm', 'bmaj', 'c#7', 'c#m', 'c#maj', 'c7', 'cm', 'cmaj', 'd#7', 'd#m', 'd#maj', 'd7', 'dm', 'dmaj', 'e7', 'em', 'emaj', 'f#7', 'f#m', 'f#maj', 'f7', 'fm', 'fmaj', 'g#7', 'g#m', 'g#maj', 'g7', 'gm', 'gmaj'], I encoded each of them in order from 0 to 35. The reason I put them in the order above is that I grouped the most similar chords together so that they would be close to each other on a numeric scale.

Another important aspect of preprocessing is how I turn MIDI notes into numerical data. I do this using a Python library called "pretty_midi"[31] to load a MIDI file and parse the note information from it. I then have a function that loops through each note of each MIDI file and extracts the start time, end time, pitch, and velocity for every note. All four of these values are in a numerical format. These four features are then stored as a tuple for each note representing the note in a purely numerical format that can be fed into the model as training data. However, there is still more preprocessing that needs to be done in order to have proper training data. First, the code normalizes the start time, end time, pitch, and velocity of each note to be between 0 and 1 by dividing the raw values by the maximum possible value. The normalized values are then stored as a new tuple for each note. Normalizing training data is important because it helps make the data more consistent, can avoid bias, and improves accuracy in general. I then pad the sequence of normalized notes to a fixed length, which is necessary because the input sequences for the neural network layer must all be the same length. Some training files may have more notes than others, which this step takes care of. If a sequence is shorter

[31] Raffel, C. (n.d.). *pretty_midi · PyPI*. PyPI.

than the maximum length, it is padded with a 0 to make it the same length as the longest sequence. Finally, I convert the list of padded normalized notes into a 3D array where the first dimension represents the number of examples for a given label, the second dimension represents the number of time steps in the sequence, and the third dimension represents the number of features for each time step (i.e., start time, end time, pitch, and velocity).

Based on the structure of the training data and the encoding of the chord labels, both the bassline and chord models have a basic training and generation architecture as follows:

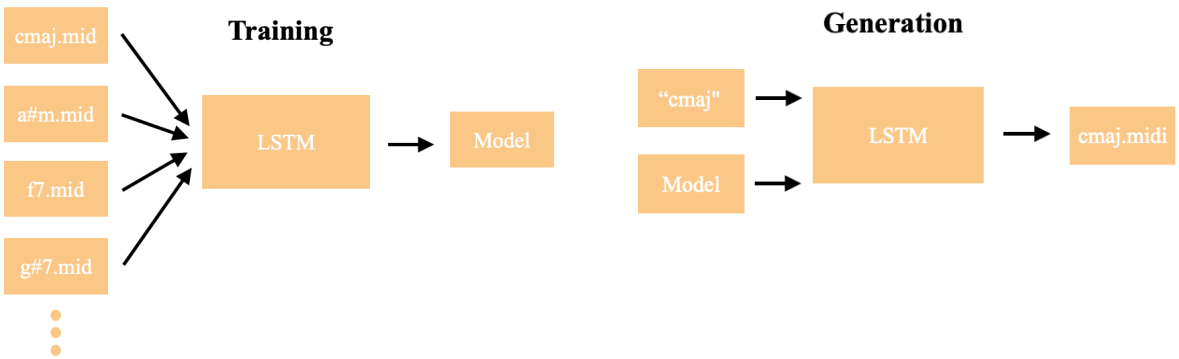

Figure 13: General Architecture of my LSTM Generation Model.

(1) The Bassline

To generate a new bassline for the input song, we first consider the array of chords outputted in *Section 4.4: Song Analysis - Chord Progression* which represents the chord progression of the input song. As explained above, these chord progressions are

estimated on a measure-by-measure basis, meaning that the model that I create must also output chords on a measure-by-measure basis using chord string labels as an input and outputting one measure bassline MIDI files for each chord. In general, all of the genres that I support using my model have very distinct bassline rhythms and notes. For the purpose of conciseness, I will not go into explaining each genre and instead will use the bossa nova style as an example for each generation model. However, every genre's bassline model has the exact same architecture and was trained using training data structured in the exact same way, so the explanation provided below is interchangeable between all genres.

For the training data, I recorded basslines in all the styles myself using a MIDI keyboard and a computer program called Ableton, which is a digital audio workstation that makes recording and playing back MIDI very easy. I recorded each bassline in every single key and tried to incorporate as many different variations of basslines for each genre. For the bossa nova style, the typical bassline follows a very simple general pattern as seen below[32]:

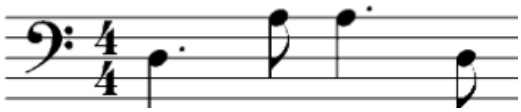

Figure 14: Bossa Nova Bassline Example.

The bassline alternates between the root note and the fifth note of the chord in a syncopated rhythm that hits the first beat, the second-and-a-half beat, the third beat, and the fourth-and-a-half beat. Similar to many basslines, the pattern played is repeated for each chord, although with slight variations. This repetition is what allows my model to

[32] *How to Play Bossa Nova (Afro-Brazilian Jazz)*. (n.d.). The Jazz Piano Site.

have long-term structure while still sounding harmonically good, because my model attempts to generate a similar pattern for each chord in the chord progression.

The machine learning model I decided to use for my bassline model was a Long Short-Term Memory (LSTM) neural network, which is a type of recurrent neural network that is specifically well-designed for sequence-based tasks. Music is a time-based form of art and deals with sequences of notes, meaning that a sequence-based model is a good approach to music generation. Because the papers I researched in my related work section applied transformer-based neural networks to music generation, I decided to use a different kind of sequence-based model to try something new. This led me to LSTMs. These models are a unique kind of recurrent neural network because they have a memory cell that allows the network to remember patterns and relationships between inputs and the training data[33]. Because my goal was to build a network that would learn the patterns in rhythm and notes that are apparent in the many genres I wanted to support in my model, LSTMs seemed like the most logical sequence-based model to apply to my approach.

The architecture of the LSTM is as follows:

```
model = Sequential([
    Embedding(input_dim=num_classes, output_dim=128, input_length=1),
    LSTM(128, return_sequences=True),
    LSTM(128, return_sequences=False),
    Dense(max_len * 4, activation='relu')
])
```

Figure 15: LSTM Model Code.

[33] Brownlee, J. (2017, May 24). *A Gentle Introduction to Long Short-Term Memory Networks by the Experts - MachineLearningMastery.com*. Machine Learning Mastery.

This model has four layers. The first layer is an embedding layer that maps each chord label input to a dense vector of length 128. The “input_dim” parameter specifies the number of unique chord labels in the input data, and the “output_dim” parameter specifies the dimensionality of the dense vectors. The “input_length” parameter specifies the length of each input sequence, which is 1 in this case since each chord label is processed independently. The embedding layer is important because it allows the model to represent the categorical input data (chord labels) in a continuous space, which can help the model learn the complex relationships between the input and output.

The second layer is the first of two LSTM layers in the model. It has 128 memory cells, which allows it to learn patterns and relationships in the input data over long sequences. I adjusted this value while training to find a good medium between training time and how well the model was able to recognize the distinct bass patterns in the training data. The “return_sequences=True” parameter specifies that the layer should output a sequence of hidden states for each input in the sequence, rather than just the last one. This is important because it allows the second LSTM layer to process the entire sequence.

The third layer is the second LSTM layer. It also has 128 memory cells, and the “return_sequences=False” parameter specifies that it should output only the last hidden state of the sequence. This means that it takes in the output of the first LSTM layer (a sequence of hidden states) and processes it to generate a single output vector that summarizes the entire sequence. I also played around with having anywhere from one to five LSTM layers, however I found that having two LSTM layers had the most success

between low training time and a high accuracy for identifying the bass patterns in the training data.

The last layer of the model is a dense layer. It has “max_len * 4” units, which is the maximum length of the input sequence multiplied by 4 (since each input consists of 4 features: start time, end time, pitch, and velocity). The activation function is set to “relu”, which essentially means that the output values will be non-negative. The purpose of this layer is to generate the output bassline sequence based on the input chord label and the processed sequence of hidden states from the LSTM layers. The weights of the layer determine how much each input feature contributes to each output feature. Finally, the “relu” activation function helps to introduce non-linearity into the output of the layer, which can help the model learn more complex relationships between the input chord label and output notes.

I trained the bossa nova model using a set of one measure MIDI files in the bossa nova style that I recorded myself as explained above. I then repeated this for each genre. As a result, I have a separate bassline model for each genre that can be used to generate a new bassline in that genre. On the generation side, I take the chord progression estimated from the input song and loop through the array of chords, generating a new bassline for each measure (since each chord in the chord progression estimation represents one measure). After generating a new one measure bassline, I essentially do reverse-preprocessing where I take the numerical values outputted by the model, de-normalize them, and then convert them back to MIDI notes using the “pretty_midi” Python library. I then concatenate all of the one measure MIDI bassline outputs into one long MIDI track, which then can be used for the input song remix.

(2) The Chords

The chord progression model is exactly the same as the bassline model architecturally, however it differs in the training data structure and post-processing. Chords are an interesting part of music to try and generate because even small time scales like single-measure chunks have a lot of individual notes in them. And the more notes being generated at once, the higher chance of a note being out of harmonic context or clashing with the rest of the generation. Unless a generative model is extremely overfitted to the training data, there is always a chance that the numeric values outputted by the model represent a note slightly out of harmonic context. When I tried to train the chord model in the exact same way as the bassline model, the outputs were significantly less harmonically and rhythmically coherent simply because of the volume of notes in each measure. Another major issue was that chords tend to have more diverse rhythms on a measure-by-measure basis than basslines, which the LSTM was having trouble learning. Every output came out as a jumbled mess of notes and rhythms that was a feeble attempt at understanding all the various notes and rhythms that came up in the one measure MIDI training data.

To solve this problem, I took a different approach for the chords model. Instead of focusing on both notes and rhythms, I decided that rhythm was the most important part of the chords. In general, the actual notes being played as part of a chord are very interchangeable and harmonically, it doesn't really matter which notes are played as long as they are part of the chord. For example, a chord can be made up of only three notes (the root, the third, and the fifth) and at the same time could also be made up of ten notes

(the root, the third, the fifth, the seventh, the ninth, etc). However, on a very basic level, both of these chords serve the same purpose in creating a fuller instrumental in the given harmonic context. For this reason, I decided to use only rhythms as training data and then hardcode chord pitches for major chords, minor chords, and dominant chords.

For the training data, I recorded one measure rhythm sequences on the root note of each chord in each genre using a MIDI keyboard and Ableton again. I recorded each rhythm sequence in every single key and tried to incorporate as many different variations of rhythm sequences for each genre. For the bossa nova style, the typical chord and rhythm sequence follows a very simple general pattern as seen below[34]:

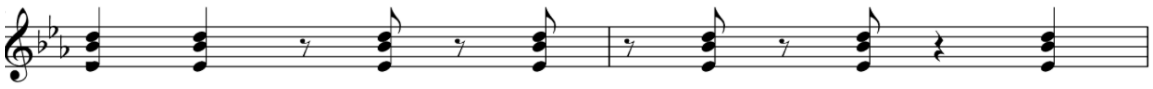

Figure 16: Bossa Nova Chord/Rhythm Example Sequence.

The actual notes don't really matter as long as they fit the harmonic context of the input chord, but the rhythm is very important to the bossa nova style. This is technically a two measure sequence, although I adapted this to fit my one measure structure by recording each measure as its own file. Each style my model supports has a relatively distinct rhythm similar to a bossa nova.

Once I had recorded all of my rhythm sequences in every key for each genre, I then preprocessed these files the same way I did for the bassline model and fed it into the same LSTM architecture.

[34] *How to Play Bossa Nova (Afro-Brazilian Jazz)*. (n.d.). The Jazz Piano Site.

On the generation side, the model generates only the root notes of each input chord with the rhythms learned from the training data. Once these MIDI files are generated, I have a post-processing function that then adds the hard-coded thirds, fifths, and sevenths of each chord, taking into account whether the chord is major, minor, or dominant. These hard-coded notes follow the same rhythm as the root notes generated by the model, creating one measure simple chord patterns with rhythms specific to the given genre. Similar to the bassline model, I have a separate chord model for each genre that can be used to generate a new chord sequence in that genre. I take the chord progression estimated from the input song and loop through the array of chords, generating a new chord sequence using the model for each measure (since each chord in the chord progression estimation represents one measure).

4.6 Remix Generation - Vocal

The final step of the remix generation is generating vocals from the input song using a model trained on my own voice. This step involves using an open source voice conversion model called the SoftVC VITS Voice Conversion from the Voice Conversion Toolbox (VCTK)[35]. This project uses a VITS algorithm, which is a voice conversion technique that uses a neural network to transform source audio files of a voice into a target voice. The neural network architecture used is called a generative adversarial network (GAN) framework, which consists of two neural networks: a generator and a discriminator. The generator can generate samples that resemble the training data while

[35] *34j/so-vits-svc-fork: so-vits-svc fork with realtime support, improved interface and more features.* (2023, April 16). GitHub.

the discriminator evaluates the samples to determine whether they are real or generated by the model. An example of this is depicted below:

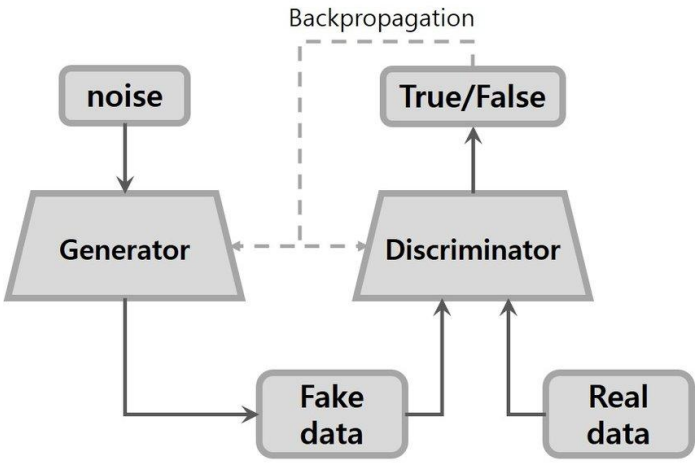

Figure 17: Generative Adversarial Network Architecture[36].

For the SoftVC VITS model, the generator is made up of an encoder, a decoder, and a conversion module. The encoder maps the input source audio files of the voice to a low-dimensional latent space. The decoder then takes the latent representation and generates a spectrogram of the target voice. And lastly, the conversion module transforms the spectrogram of the original voice to the target spectrogram.

Meanwhile, the discriminator is trying to figure out whether the data it is getting passed is a real voice recording audio file or a file generated by the generator. For the SoftVC VITS model, the decoder is made up of a convolutional neural network that evaluates the spectrograms of the input audio files to determine whether they are real or generated. While training, this architecture is trained to minimize the difference between the generated spectrogram and the spectrogram of the training data. This is done using a combination of mean squared error and adversarial loss to

[36] Li, Y. (2023, March 11). *Seismic Data Augmentation Based on Conditional Generative Adversarial Networks*. ResearchGate.

ensure that the generated spectrogram is indistinguishable from a real spectrogram of the target voice[37].

Now for the fun part: the training data. Since this is an open source architecture, I did not design this model myself and the major work I did was on creating a large subset of high-quality training data of my own voice. I did this by recording myself speaking and singing a large dataset of phrases in English known as *The Harvard Sentences*. These 720 sentences are a set of standardized phrases commonly used in speech recognition, perception, and audiology research. Developed by the Harvard Psychoacoustic Lab in the 1940s, these sentences are made to contain an equal distribution of phonetic sounds representing almost every possible sound that is in the English language[38]. Although these sentences are not usually applied to machine learning models, I thought that feeding the SoftVC VITS model with a training corpus that has almost every sound in the English language would make the model as close to my voice as possible. I even attempted to sing certain sentences in multiple pitches to have representations of my voice at higher and lower pitches compared to everyday speaking. Using this large training dataset, I trained the SoftVC VITS model for 10,000 epochs to create a general generative model of my voice.

In order to generate the vocals of the input song in my own voice, I also had to create a way to get the vocals of the input song on their own without the instrumental part of the audio file. I did this using another open source model called Spleeter. This model can separate music tracks into individual components and instruments using a fully convolutional neural network with a U-Net like architecture. This consists of a series of convolutional layers that process an input audio file and produce a spectrogram for each component of the input song. The network was trained in a supervised way where the training data was made up of audio files paired with

---

[37] *34j/so-vits-svc-fork: so-vits-svc fork with realtime support, improved interface and more features.* (2023, April 16). GitHub.
[38] Kwong, A. (n.d.). *Harvard Sentences*. CS @ Columbia.

already separated components[39]. This architecture is inspired by biomedical image segmentation but adapted to work on audio file spectrograms. Similar to my audio file analysis at the beginning of this thesis, the input audio file is transformed into the log-magnitude spectrogram before being fed into the model. This is a fully pretrained model, which allowed me to simply input the input audio file and get the vocal and instrumental components.

After getting the vocal component using the Spleeter model, I then feed those vocals into the SoftVC VITS model trained on my own voice to generate the lyrics of the input song in my own voice. The combination of these vocals with the instrumental generated above makes up the algorithmically generated remix of the input song in its entirety.

[39] Deezer. (2021, September 3). *deezer/spleeter: Deezer source separation library including pretrained models.* GitHub.

5 Results

After generating the new MIDI bassline, new MIDI chords, and new vocals for an input song in a given genre, each of these files can then be uploaded into any digital audio workstation such as Ableton, Logic Pro, or FL Studio to hear the result. A few example remixes based on “Sweet Caroline” by Neil Diamond can be found in the “Output_Examples” folder of the GitHub below:

https://github.com/cwallant/AlgoRemix/tree/main/Output_Examples

6 Evaluation

Because the output of the model I propose in this thesis is music, there isn't really a clear method of evaluation that analyzes the output in an objective, quantitative way. Music is inherently a subjective medium that is used by people for many different purposes and settings. Some people enjoy music to relax while laying out in the sun on a warm day. Others want their music to hype them up for a workout or party. Creating a general music model based on a few genres like I have in this thesis could serve any of those use cases and certain people may like or dislike the outputs because of their specific knowledge of music theory, their preference of genres, or their general relationship with music. With this in mind, the two methods of evaluation that I believe are most useful to the model I propose in this thesis are a genre classification evaluation and a subjective listening evaluation using volunteer test subjects.

6.1 Genre Classification Evaluation

The genre classification evaluation I used is an attempt at quantitatively measuring my model's success at generating each of the five genres it was trained on: bossa nova, techno, disco, polka, and bluegrass. To do this, I used a popular genre classification architecture that is based on a convolutional neural network approach[40].

[40] *Python · GTZAN Dataset - Music Genre Classification*. (n.d.). Kaggle.

This approach trains a convolutional neural network on a corpus of 30 second long clips of songs labeled by their genre. The audio files are then preprocessed into their respective Mel-spectrograms (just as I did for my song analysis), which are then fed into the convolutional neural network in an architecture that looks something like this:

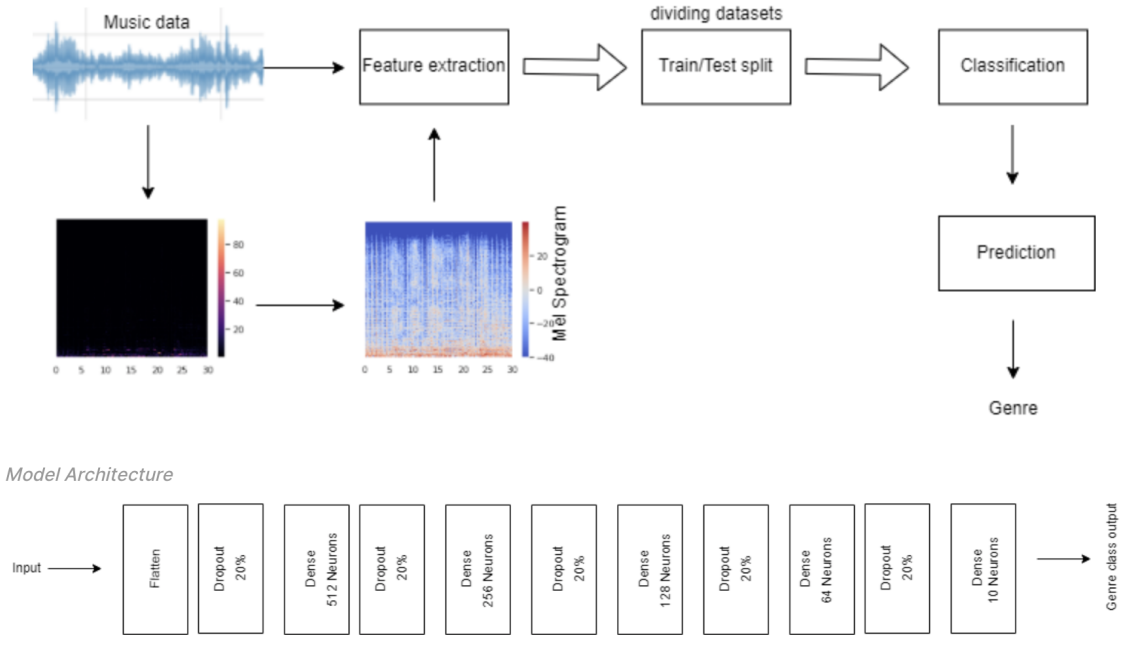

Figure 18: Methodology and Architecture of Genre Classification Model[41].

The original model is trained on a dataset called GTZAN, which is a collection of 1000 30-second music audio files in ten different genres[42]. However, this dataset does not include techno, bossa nova, polka, or bluegrass as genres. To work around this problem, I built my own dataset of 100 30-second audio files for each of my genres, totaling 500 audio files. I then appended my dataset to the GTZAN dataset because I wanted as many genres as possible in my dataset (in case the model identifies my generations as

[41] *Python · GTZAN Dataset - Music Genre Classification*. (n.d.). Kaggle.
[42] Tzanetakis, G. (2022, December 6). *gtzan*. TensorFlow.

completely separate genres). After modifying the dataset, I retrained the model architecture proposed above to create my genre classification model.

After training the model, I then fed my "Sweet Caroline" remixes into the model to see what genres it would classify each remix as. The results are as follows:

Song Type	Original Song	Bossa Nova Remix	Techno Remix	Disco Remix	Polka Remix	Bluegrass Remix
Estimated Genre	Pop	Bossa Nova	Techno	Disco	Polka	Polka

Figure 19: Genre Classification Results.

For the most part, my model seems to be successful in generating remixes in new genres that follow the general patterns and Mel-spectrogram features related to those genres. However, both the polka and the bluegrass remixes were classified as polka, which was an interesting surprise. Upon listening to the training data, this does somewhat make sense because the bassline pattern that my model generates for bluegrass remixes is generally quite similar to the basslines generated for polka remixes. The bluegrass music used to train the genre classification model is much more complicated and has a lot more rhythmic and harmonic variation, which demonstrates a limitation of my model in that it can only generate simple rhythms and harmonic patterns at this point in time. Although the rest of the genres were correctly classified, each of the genres I chose (apart from bluegrass and polka) are sonically very different from one another which likely led to the success of this genre classification model. I would be interested to see if I trained my model in the future on two adjacent genres of jazz (bebop and swing) to see if my model could still pass this genre classification evaluation for two very similar genres.

6.2 Subjective Listening Evaluations

The next part of the evaluation of my model was the subjective listening evaluation I ran with volunteer testers. Based on my original goal to make AI generated music that people would enjoy listening to, most of my subjective listening evaluation focused on how much people enjoyed the remix that was made compared to the original input song. The input song I used for evaluation was also “Sweet Caroline” by Neil Diamond.

My testing group consisted of twenty volunteers. I made sure this group had varying experience in music theory and varying music listening habits. At the beginning of the evaluation survey, I had them explain in detail their listening habits and rank their familiarity with music theory to give me a better idea of what their opinions might be influenced by. Twelve of the participants gave themselves a music theory rating of four and below and the other eight rated themselves five or above and had experience playing an instrument. Having this split was important because I wanted people familiar with music theory to evaluate my results in order to have a more detailed evaluation of the harmonic features of my generated remixes. I also had each of the participants rank the five genres my model supports so I could see if their preferences for genres might impact their preferences for the remixes generated in each genre.

For the listening evaluation, I had them listen to the original version of “Sweet Caroline” and rank how much they liked the song. I then had them listen to each of the remixes one after another and rank how much they enjoyed the remix compared to the original. The average rating for the original version and each remix is as follows:

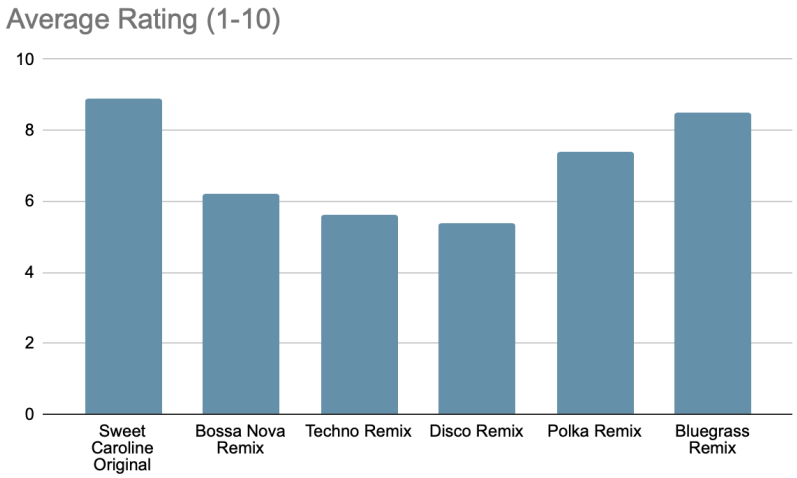

Figure 20: Average Ratings for Each Remix.

Surprisingly, the highest rated remix was consistently the bluegrass remix. Despite not a single participant listing bluegrass as one of their preferred genres in their listening habits, the participants seemed to enjoy how the bluegrass style sounded with the "Sweet Caroline" lyrics. The polka remix also ranked high for a similar reason. Participants noted that mixing the polka style with a more mainstream song like "Sweet Caroline" was quite funny which accounted for the high rating. The bossa nova, techno, and disco remixes were rated lower and did not have as much excitement about them. Before revealing what genre each remix was supposed to be based on, I had each participant listen to some example songs for each genre and then see if they could identify which genre each of my remixes was supposed to recreate. Surprisingly, this test had a 100% success rate which I think is in part due to the fact that the five genres I chose are very separate from each other and sound relatively distinct.

One last thing to note is that all the remixes ranked lower than the original song, with most participants explaining that the remixes were not as harmonically complicated

and more repetitive than the original. This presents a potential limitation of my model because it relies on a measure-by-measure approach at generation, which creates a lot of potential stylistic repetition. Although my original goal was to have significant repetition to solve the long-term structure problem, I may have gone a little too far in the direction of structure. Despite this limitation, participants were generally impressed with the remixes generated and said they could see themselves listening to some of the remixes in their free time.

Another aspect of my subjective listening evaluation was the evaluation of the generated vocals. I first asked the participants to rate the vocals from one to ten. I then told them that the vocals were trained on a model to sound like me and asked them to rate how closely the model sounded to my normal speaking voice. The average rating for the vocals was 7.4/10 based on all 20 submissions. The main explanation given for lower scores was that the vocal model has a hard time generating the proper words at higher pitches, but otherwise sounded great at the lower pitches. 11 out of the 20 participants asked if the vocal was me singing before I told them it was, and 4 out of the remaining 9 participants said the model did sound like me after learning that it was trained to match my voice. This evaluation was generally positive in the sense that most participants did say the model sounded like me. However, the evaluation also pointed out a limitation of my model in that it had a very hard time generating vocal pitches that were outside my normal speaking (and attempted singing) range that I recorded for the training data.

7 Conclusion

In this thesis, I have demonstrated the application of neural networks to AI music generation through the creation of multi-genre remixes of an input song. In my research on related approaches to AI music generation, I identified a recurring issue with long-term structure where all previous approaches were unable to create full length songs with modern verse-chorus and chord progression structures. To solve this, I decided to apply AI music generation specifically to altering existing music through song remixing. Generating remixes of an input song solved the long-term structure issue because the remixes relied on the verse-chorus structure and chord progressions of the input song. For the actual musical generation aspect of this approach, I used an LSTM neural network to train models for five different genres including bossa nova, techno, disco, polka, and bluegrass. Capable of generating new basslines and chords based on the chord progression of the input song, these models made up the bulk of the music generation in my approach. I also trained a generative adversarial neural network on a dataset of audio files of my own voice to generate vocals for the remix. The vocals were based on the lyrics from the input song but were generated to sound like my own voice.

Based on my quantitative genre classification evaluation, my models achieved my goal of generating remixes of an input song in multiple genres. The genre classification algorithm correctly identified 80% of my remixes in the correct genre. I also ran a subjective listening test using volunteers to see if real people would actually enjoy the

generated music. This had relatively positive results as well, but also identified some limitations of my models.

The main limitation of my models is that the generated music is simple and somewhat repetitive. Based on my goal to generate music with local and long-term structure, I generated the new basslines and chords on a measure-by-measure basis. This led to a lot of the generation being repetitive, with my model generating similar results for each measure except pitch-shifted to the relevant chord. In the future, I would attempt to train my model on a much larger corpus of data that considers longer local sequences. For example, I would record four-measure bassline sequences over multiple chords instead of the one-measure single-chord recordings I used for my training data. I believe this would help make the music generation less repetitive measure-to-measure.

Another limitation that was suggested was on the vocal side. The model trained on my voice had trouble generating lyrics outside my normal vocal range. Although I attempted to sing in the recordings I made for my training data, I did not successfully record every sound in the English language in every pitch. My training data lacked a lot of sound in the higher pitch range. In the future, I would tackle this problem by attempting to sing each of the Harvard Sentences at every pitch in a normal singing range, however this would take a *significant* amount of time (on the scale of a few weeks of daily recording).

Beyond fixing the limitations in my model, there are a lot of directions this project could be taken in the future. I can very realistically see a world where music becomes algorithmically altered to match an individual's preferences live while they listen. For example, an individual who has been identified as a jazz fan could play a

playlist of this year's top songs and have all the songs be altered to be more jazz-y to fit this preference. The model I present in this thesis is a first step in that direction and represents a major potential shift towards the future of music.

8 Appendix

8.1 Bibliography

Bello, J. P. (n.d.). *A Tutorial on Onset Detection in Music Signals*. School of Electronic Engineering and Computer Science. Retrieved April 10, 2023, from http://www.eecs.qmul.ac.uk/former/people/jbc/Documents/Bello-TSAP-2005.pdf

Bock, S. (n.d.). *Joint Beat and Downbeat Tracking with Recurrent Neural Networks*. Institute of Computational Perception. Retrieved April 10, 2023, from http://www.cp.jku.at/research/papers/Boeck_etal_ISMIR_2016.pdf

Brownlee, J. (2017, May 24). *A Gentle Introduction to Long Short-Term Memory Networks by the Experts - MachineLearningMastery.com*. Machine Learning Mastery. Retrieved April 16, 2023, from https://machinelearningmastery.com/gentle-introduction-long-short-term-memory-networks-experts/

Catteau, B. (2021, May). *Fig. 14. The input wave-file of MIDI to WAV*. ResearchGate. Retrieved April 7, 2023, from https://www.researchgate.net/figure/The-input-wave-file-of-MIDI-to-WAV-rendered-data-and-the-derived-chroma-vectors_fig10_221649522

Clarke, H. D. (2005). *Autocorrelation*. ScienceDirect. Retrieved April 10, 2023, from https://www.sciencedirect.com/science/article/pii/B0123693985001572

Deezer. (2021, September 3). *deezer/spleeter: Deezer source separation library including pretrained models.* GitHub. Retrieved April 17, 2023, from https://github.com/deezer/spleeter

Dhariwal, P. (2020, April 30). *Jukebox: A Generative Model for Music*. OpenAI. Retrieved April 9, 2023, from https://openai.com/research/jukebox

The Echo Nest Blog — The Echo Nest Song API. (2010, April 24). The Echo Nest Blog. Retrieved April 9, 2023, from https://blog.echonest.com/post/545703077/the-echo-nest-song-api

Gündert, S. (2014). *Mel Filter Bank — PyFilterbank devN documentation*. GitHub Pages. Retrieved April 10, 2023, from https://siggigue.github.io/pyfilterbank/melbank.html

Hart, R. (2012, August 19). *Key-finding algorithm*. Robert Hart's Homepage. Retrieved April 10, 2023, from http://rnhart.net/articles/key-finding/

Hawthorne, C. (2022, February 15). *General-purpose, long-context autoregressive modeling with Perceiver AR*. arXiv. Retrieved April 9, 2023, from https://arxiv.org/abs/2202.07765

Hiller, L. (1959). *Experimental music; composition with an electronic computer*. New York, McGraw-Hill. https://archive.org/details/experimentalmusi00hill/page/n9/mode/2up

How to Play Bossa Nova (Afro-Brazilian Jazz). (n.d.). The Jazz Piano Site. Retrieved April 16, 2023, from https://www.thejazzpianosite.com/jazz-piano-lessons/jazz-genres/how-to-play-bossa-nova/

Huang, Q. (2023, February 8). *Noise2Music: Text-conditioned Music Generation with Diffusion Models*. arXiv. Retrieved April 9, 2023, from https://arxiv.org/abs/2302.03917

Khan, A. (n.d.). *Skipping Class to Build a Popular Music Remixing App*. Indie Hackers. Retrieved April 9, 2023, from https://www.indiehackers.com/interview/skipping-class-to-build-a-popular-music-remixing-app-262da11241

Kwong, A. (n.d.). *Harvard Sentences*. CS @ Columbia. Retrieved April 17, 2023, from https://www.cs.columbia.edu/~hgs/audio/harvard.html

Li, Y. (2023, March 11). *Seismic Data Augmentation Based on Conditional Generative Adversarial Networks*. ResearchGate. Retrieved April 16, 2023, from https://www.researchgate.net/publication/347552182_Seismic_Data_Augmentation_Based_on_Conditional_Generative_Adversarial_Networks

Librosa: Audio and Music Processing in Python. (n.d.). Librosa.org. Retrieved April 9, 2023, from https://librosa.org

Mantione, P. (2018, July 3). *The Fundamentals of Amplitude and Loudness — Pro Audio Files*. Pro Audio Files. Retrieved April 10, 2023, from https://theproaudiofiles.com/amplitude-and-loudness/

Music21: a Toolkit for Computer-Aided Musicology. (n.d.). Massachusetts Institute of Technology. Retrieved April 10, 2023, from https://web.mit.edu/music21/

O'Connor's, R., & O'Connor, R. (2022, May 12). *Introduction to Diffusion Models for Machine Learning*. AssemblyAI. Retrieved April 9, 2023, from

https://www.assemblyai.com/blog/diffusion-models-for-machine-learning-introduction/

OpenAI. (2019, November 5). *GPT-2: 1.5B release*. OpenAI. Retrieved April 8, 2023, from https://openai.com/research/gpt-2-1-5b-release

Payne, C. (2019, April 25). *MuseNet*. OpenAI. Retrieved April 8, 2023, from https://openai.com/research/musenet

Python · GTZAN Dataset - Music Genre Classification. (n.d.). Kaggle. Retrieved April 18, 2023, from https://www.kaggle.com/code/jvedarutvija/music-genre-classification

Raffel, C. (n.d.). *pretty_midi · PyPI*. PyPI. Retrieved April 16, 2023, from https://pypi.org/project/pretty_midi/

Razavi, A. (2019, June 2). *Generating Diverse High-Fidelity Images with VQ-VAE-2*. NIPS papers. Retrieved April 9, 2023, from https://proceedings.neurips.cc/paper_files/paper/2019/file/5f8e2fa1718d1bbcadf1cd9c7a54fb8c-Paper.pdf

Rice, D., & Galbraith, M. (2008, November 16). *ALAN TURING: HOW HIS UNIVERSAL MACHINE BECAME A MUSICAL INSTRUMENT*. IEEE Spectrum. Retrieved March 30, 2023, from https://spectrum.ieee.org/alan-turing-how-his-universal-machine-became-a-musical-instrument

Roberts, L. (2020, March 6). *Understanding the Mel Spectrogram | by Leland Roberts | Analytics Vidhya*. Medium. Retrieved April 10, 2023, from

https://medium.com/analytics-vidhya/understanding-the-mel-spectrogram-fca2afa2ce53

Schmuckler, M. A. (2005, November). *(PDF) Perceptual Tests of an Algorithm for Musical Key-Finding.* ResearchGate. Retrieved April 10, 2023, from https://www.researchgate.net/publication/7503748_Perceptual_Tests_of_an_Algorithm_for_Musical_Key-Finding

Sobot, P. (n.d.). *The Wub Machine*. The Wub Machine. Retrieved April 9, 2023, from https://the.wubmachine.com

Spectrogram View. (2022, December 27). Audacity Manual. Retrieved April 10, 2023, from https://manual.audacityteam.org/man/spectrogram_view.html

34j/so-vits-svc-fork: so-vits-svc fork with realtime support, improved interface and more features. (2023, April 16). GitHub. Retrieved April 16, 2023, from https://github.com/34j/so-vits-svc-fork

Tzanetakis, G. (2022, December 6). *gtzan*. TensorFlow. Retrieved April 18, 2023, from https://www.tensorflow.org/datasets/catalog/gtzan

Verma, Y. (2021, September 19). *A Tutorial on Spectral Feature Extraction for Audio Analytics* -. Analytics India Magazine. Retrieved April 9, 2023, from https://analyticsindiamag.com/a-tutorial-on-spectral-feature-extraction-for-audio-analytics/

What's the difference between MIDI and audio? (n.d.). Roland. Retrieved April 7, 2023, from https://www.roland.com/uk/blog/midi-vs-audio/

Yang, D. (2022, 20 July). *Diffsound: Discrete Diffusion Model for Text-to-sound Generation*. arXiv. Retrieved April 9, 2023, from https://arxiv.org/abs/2207.09983

8.2 Code

All the code can be found at the following GitHub Repository:

https://github.com/cwallant/AlgoRemix