EECE 634 Project Report

Musicar: A Text-to-Music Generator

Kassem Jaber kmj08@mail.aub.edu, Karim Dahrouge ksd05@aub.edu.lb

***Abstract*—Converting text to music is a task which can be performed using rule-based methods, statistical methods or machine learning. Our scope is to develop a statistical NLP-based model to successfully produce a sequence of aligned notes and durations to compose a melody for the text. We proposed two different approaches; the logistic regression and the NgramTagger models and decided to move forward with the NgramTagger due to constraints imposed by logistic regression. We then plotted the performance curve to tune the number of ngrams as our hyperparameter and find the optimal dev set performance. We also produced two tables for notes and durations classification without the dev set to see how the performance metrics change as the model complexity increases. The plot and the table yielded different conclusions, as the TrigramTagger with cutoff = 1 and 0 for the notes and duration models respectively performed the best, whereas the (N=4)gram model was the optimal given the table. This is probably due to statistical errors as the values are small, and may have large variance. We then synthesized the predictions by computing sinusoids and windowing them with a hamming window for higher quality tonations. The module used to playback the generated audiostream is the *sounddevice* python module.**

I. Introduction

A text to music conversion system can be developed in numerous ways to achieve diverse results under a common objective; to develop some sort of music. This music can be completely arbitrary, and its properties can be dictated by a set of rules, a system a programmer and musician named Jan 'Willem Kolkman' developed; the LangoRhythm [2]. It is short for language, algorithm and music and an intriguing TEDx talk was given by him [4] describing how the system works and a few demos of his software conversion system. The system ended up being a black box due to the number of rules being added and the significant increase in complexity as a result. It should be noted, however, that his conversion system produces music with properties which depend on the word structures, and therefore do not represent the context in which these words are used. In other words, the music doesn't have dynamic genre, or doesn't produce any music which reflects the semantics of the given speech. For example, the tempos are generated based on the average word lengths.

We want to develop a text to music conversion system utilizing Natural Language Processing (NLP) techniques and methods to convert text to a musical composition which in theory extracts the semantics of the given text to produce music with contextual meaning. However, to achieve this goal in its entirety, we would need to impose complex music theory rules to achieve a musical composition with proper structure and ratios. This will be a difficult task to complete for a course project, and therefore we will attempt to produce a musical composition hoping to achieve: melodies, tempo and structure.

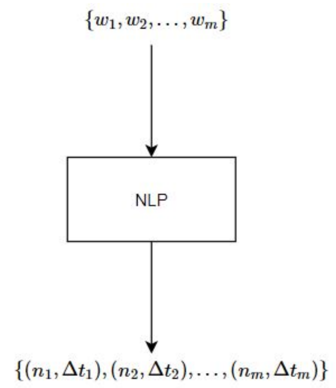

$\{w_1, w_2, \dots, w_m\}$
NLP
$\{(n_1, \Delta t_1), (n_2, \Delta t_2), \dots, (n_m, \Delta t_m)\}$

Fig. 1: Given a set of textual data, we would like to estimate the most likely frequency (i.e. note) and duration associated with it.

II. Problem Formulation

We will then formulate our classification problem as follows. Given a set of textual inputs $\{w_1, w_2, ..., w_m\}$, we wish to label each w_i with a note n_i and word duration $\Delta t_i \quad \forall i \in \{1, 2, .., m\}$ given a set of m words. **Figure 1** demonstrates the formulated problem and emphasizes the need to develop a model to convert a single input to multiple outputs.

There are several challenges we need to address and tackle to reach the final goal of text to music conversion:

- text preprocessing (stemming, lemmatization, normalization etc.)
- generate/find dataset
- model development
- model evaluation
- music synthesis

The following sections will address and describe ways to tackle the aforementioned problems.

III. Preprocessing

The first step of any NLP task is to preprocess the textual data such that words with the same semantics with respect to target objective are mapped to a single word (i.e. $z = g(w_{x1}, w_{x2}, .., w_{xk})$. However, the method to perform such a mapping is highly specific to the application of the NLP task, and therefore the given application must be studied beforehand. The preprocessing task consists of removing characters and cleaning text such that the postprocessed text represents the users typing intentions. This includes possible mispelled words and strange non-alphanumeric characters added to the

Algorithm 1 Preprocessing Algorithm

```
    Input Word token $w$
    Output Processed token $w'$
    Variable Initialization $w' \leftarrow w$
 1: procedure PREPROCESS($w$)
 2:     $p_1 \leftarrow$ '\W*([\w\d-]*\w\b)\W*'
 3:     $p_2 \leftarrow$ '\b(o)+(h)*\b'
 4:     $p_3 \leftarrow$ '\b(u)+(g)*(h)*\b'
 5:     $p_4 \leftarrow$ '\b(y)+(e)*(a)*(h)*\b'
 6:     $p_5 \leftarrow$ '([∧ y]+)(y)+'
 7:     for $i = 1 \rightarrow 5$ do
 8:         compute RegEx($p_i$)
 9:         substitute $w'$ with all capturing groups
10:         store result in $w'$
11:     end for
12:     $\epsilon \leftarrow \{\}$
13:     for each word $w_l$ in lexicon $\mathcal{L}$ do
14:         $\epsilon' \leftarrow$ min_edit_distance($w'$,$w_l$)
15:         $\epsilon \leftarrow \epsilon + \{\epsilon'\}$
16:     end for
17:     $j' \leftarrow$ argmin($\epsilon$)
18:     $w' \leftarrow \mathcal{L}_j$
19:     return $w'$
20: end procedure
```

mix. We also want to remove letters that emphasize the extended nature of the word in a singing context. For example, we wish to process the word "ooooh" into the word "oh". Each word has a time duration attribute that indicates for how long it is being sung, so we do not need repeated vowels to show duration.

To deal with the issue of mispelled words, we can use a given lexicon, and autocorrect the word using the minimum edit distance algorithm, and choosing the word with the minimum edit distance.

To deal with the issue of non-alphanumeric characters contaminating the textual data and repeated vowels (typically done to express extended pronunciations), we resort to simple regular expressions. We addressed four different cases:

1) cleaning any non-alphanumerics in the word while preserving possible hyphens in the middle (for compounding two words)
2) the word "oh" with more than one repeat of each letter
3) the words "uh" and "ugh" with each letter repeated more than once
4) the words yeah, which can written as ye, ya, yeh and yeah with each letter repeating more than once
5) any word ending with a y, repeated more than once

The final preprocessing algorithm is shown in **Algorithm 1**.

IV. Dataset

Typically a theoretical model is initially developed before finding or developing the dataset to ideally solve the problem, however, given the nature of this project and how limiting it can be, we decided to first find the dataset of aligned words and their musical features (i.e. notes and frequencies) to know what we will work with. After some thorough research, we found a dataset generated by Meseguer-Brocal et al. (2019) using a teacher-student machine learning paradigm (more details can be found in his paper [3]). We were sucessfully granted an academic license to access the dataset, along with a nice tutorial on how to use it in their github [1]. The dataset includes annotations of lyrics with the aligned frequency ranges and time durations. One thing to note is that we are using automatically aligned frequencies and time deltas produced by a machine learning algorithm, and naturally there will be errors. However, it seems there are only small errors in the estimates (based on audio playback of the frequencies); good enough for our application.

The dataset contains alignments for 5358 different songs. Each song has a set of syllables, words, lines and paragraphs. Meaning, for each w_{ji}, w_i, l_i or p_i respectively there exists a set of annotations $f_i = (f_{0i}, f_{1i})$ and $\Delta t_i \quad \forall i \in \mathcal{D}$, where D denotes the dataset. A visualization of the annotated dataset can be found in **Figure 2**.As a start, we extracted 22 songs composed by 'The Beatles'. We expect the conversion to be biased on the genre and the language of the band. However, later into the project we realized that training on 22 songs is not sufficient, and we need more data. To keep the premise that learning from a band will yield interesting musical interpretations, we decided to look up artists similar to 'The Beatles'. We found a list of artists, placed them in a text file and read them into Python. We extracted now a total of 66 artists with a vocabulary size of $|V| = 1560$ and total number of words $N = 14160$ as opposed to old set of 22 songs which yielded $|V| = 631$ and $|N| = 4381$ which is an improvement in corpus size with respect to the vocabulary size. However, this improvement might not be too significant, and later will hint at the need of either improving the model's ability to extract patterns given scarce data, or the need to scrape a larger dataset from Youtube or other online resources (insufficient time to do this).

V. Model Development

Before we can develop a model, we need to identify the project's goals and the given inputs and outputs. The **objective** of this project is to successfully convert text to music while satisfying some defined performance metric to be determined. The inputs to our algorithm will be plain text to be written by the user, and the output will be an aligned set of notes to the words which will be played by the computer for an audio demonstration of the conversion system. The model will be highly dictated on the corpus we have a license for; given that we have a set of aligned notes and time durations for each word, we can identify the problem to be a **classification** problem, with the training data containing the data to be predicted and their labels. This is therefore a **supervised** learning task, and will need to employ supervised learning strategies to successfully solve this problem.

A. Part-of-Speech (POS) Tagging with NGrams

Interestingly, we can reformulate this problem to be a POS tagging task. POS tagging is an NLP task which essentially

TABLE I: Feature Table

Word	**isFirstWord** $\in \{0,1\}$	**isLastWord** $\in \{0,1\}$	**Note** n_i (Hz)	**Note** n_{i-1} (Hz)	**Note** n_{i+1} (Hz)
Lu	1	0	784	N/A	784
cy	0	0	784	784	784
in	0	0	784	784	784
the	0	0	784	784	784
sky	0	0	784	784	698.46
y	0	0	698.46	784	659.25
with	0	0	659.25	698.46	587.33
di	0	0	587.33	659.25	523.25
i	0	0	523.25	587.33	493.88
a	0	0	493.88	523.25	440
monds	0	1	440.00	493.88	N/A

'tags' or labels a set of words based on it's context. This is essentially what we wish to achieve; a context-based note labelling approach for text-to-music conversion to ultimately produce contextual melodies. Fortunately, Python's *nltk* library contains an NGramTagger model trainer, which is a supervised learning context-based model which computes the ngrams of a given text and estimates the conditional probability of all classes given these ngrams. The model predicts some tag given only the previous tags, not the previous words. The maximum *a posteriori* (MAP) estimate is taken to be the most likely class to label the text.

As an experiment, we wanted to see how bad the model will perform using only unigrams before moving on to an Ngram approach (for $N > 1$). We expect poor results mainly since music is a highly structured and creative art, which depends on the vision of the artist, the combination and structure of words this artist uses to convey meaningful emotions in the music. To capture these features, we should expect to look into more temporal structures such as ngrams, HMMs or even LSTMs, however, the scope of this project will be on ngrams.

The data was split into a training set, development set and a held-out set using the standard 80%, 10% and 10% split ratios respectively. After splitting, four different models were trained; UnigramTagger, BigramTagger,TrigramTagger and an NGramTagger. Two separate models were trained to predict for the two labels; the note and the time duration. This totals to 8 different models to be trained. In the classification stage, only two models will be used to make a prediction. Each higher order tagger is set to backoff to a lower model if the probability count is extremely low. This cutoff value will later on be tuned to determine the value which optimizes the development set's performance.

During prediction, the classifier is sometimes unable to make a proper prediction due to zero probabilities. We don't believe smoothing will improve the current model since we don't have many counts to begin with (i.e. is very sparse), so a simple workaround for this was to use the previous frequency prediction, and scale the pitch by a factor of either 2,3 or 4 (chosen at random). This makes the follow of the music smooth. A more advanced method would be to 'interpolate' the pitch for the previous and next note (assuming the next note was successfully predicted) based on some music theory-derived rules.

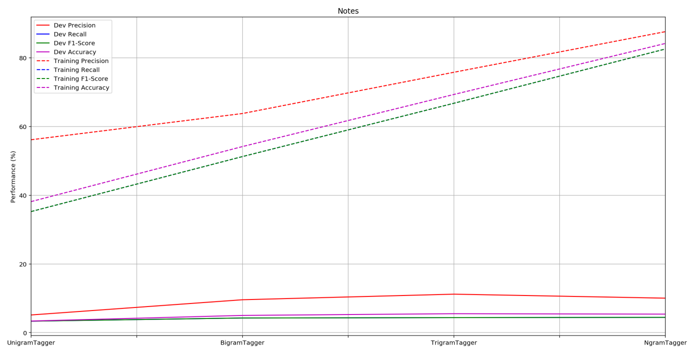
To produce a more structured musical composition, we thought it would be useful to insert in between each word a sentinel, we denoted as '$< l >$', which is typically the silent parts of speech. This silence should depend on what was previously said, and the previous tags. We therefore augment the dataset further by filling in the silent sentinel in between, labelled this silent word with a note and a duration. The duration was calculated to be the Δt between the adjacent words, and the note label will be considered 'N/A'. It should later be noted, that the model we will use does not consider previous words, and will not behave exactly as we expect it to.

B. *Logistic Regression*

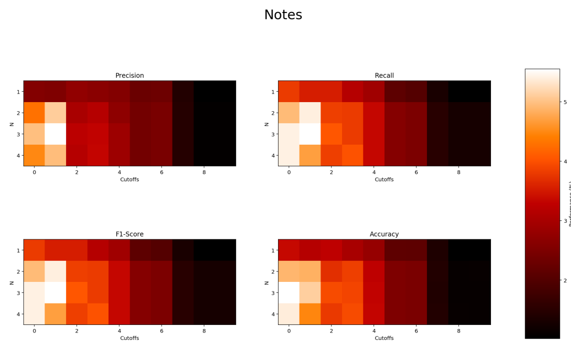
We also wanted to use a logistic regression model to make our predictions. The idea was to use features such as the position of the words in the sentence, or if the word was the first or the last one in a sentence, in addition to what we already had. However, applying a logistic regression model on this specific task proved to be very hard for multiple reasons. First of all, our classification task was between 7 classes (the 7 music notes), and this is without accounting for sharps ($\sharp$), flats ($\flat$) or even the octave we are considering. The classification choices are therefore very wide, and it is very difficult to design features that allow us to narrow down the choices of the notes. This problem could have been circumvented if the size of our data was extremely large. However, our data set was sparse, and had to be somewhat limited because of the similarity that we desired in the chosen songs. Moreover, trying to augment the already-established data set with features we desired was hard given that we obtained our data from an external source. **Table I** is an example of how the logistic regression model would have worked. We are taking as a sample the sentence "Lucy in the sky with diamonds" from the eponymous song from The Beatles. What we call here a word is a syllable with the corresponding frequency being played.

VI. Model Evaluation

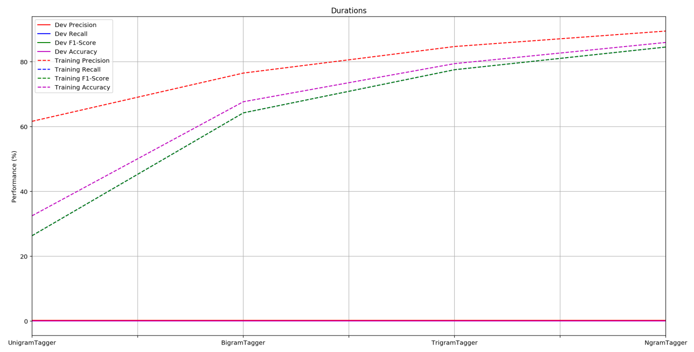
To successfully evaluate our model's performance, we need to define a performance metric to measure how 'good' our model is. This measurement is highly dependant on the application. Given that we identified the NLP task to be a classification task, we could use the conventional metrics computed in the confusion matrix such as precision, recall, false positives, false negatives, true positives, true negatives



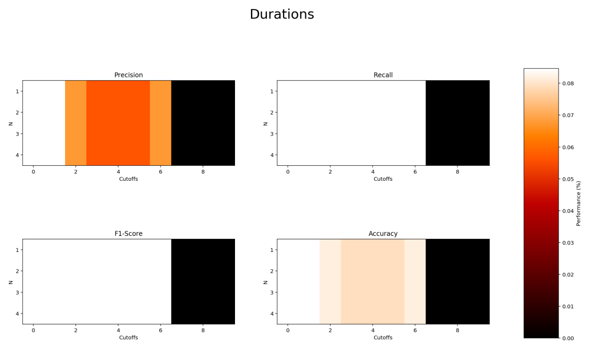
(a)



(b)



(c)



(d)

Fig. 2: a,c) Graphs of the performance curve for each conventional metric used for classification tasks. The dotted lines represents the performance on the development set, and the solid lines represents the performance for the test set. Note that the purple dashed line is overlapping the blue dashed line. Note that the curve demonstrates the performance of the model without including the silent sentinel $< l >$. b) Results for grid search show the optimal hyperparameters are $N = 3$ at cutoff=1. d) Results for grid search show the optimal hyperparameters are $N \in \{1, 2, 3, 4\}$ (1 chosen) at cutoff $\in \{0, 1\}$ (0 chosen).

and the F1-score. However, we need to note that the purpose of this project is to produce aligned melodies derived from word contexts imposed by some band. This can be a highly subjective matter in deciding whether the model is sufficient in achieving it's objective or not. We therefore are urged to introduce an *objective* and *subjective* measure of performance; for the objective measure, we will compute the confusion matrix as well as the newly designed metric based on the differences in music keys for the prediction and the labels. As for the subjective metric, this would require interviews with experts, and therefore, is not logistically feasible right now.

For the objective measures of performance, as previously mentioned, we will compute the precision, recall, F1-score, accuracy and a confusion matrix. We also defined a newer metric as an attempt to better represent the performance of the model.

We found that by calculating the performance metrics by augmenting the dataset to incorporate silent regions, the note label being consistent throughout poses an issue with artificially large performance metrics. Therefore, when it comes to computing the objective measures, we remove the silent sentinels and perform the analysis.

A. Conventional Metrics

Using *sklearn's* metric module, the accuracy, precision, recall and F1-scores are calculated for each song in the development corpus for all models. The performances are averaged over all test samples, and are plotted as a function of model complexity (i.e. number of ngrams). A graph of this performance curve can be found in **Figure 2**. The figure demonstrates a problem with overfitting for large N, and that a global optimum exists (TrigramTagger model) for the training performance constrained by the development set performance. unfortunately, this global optimum is quite small, indicating very poor performance. In fact, we would need to perform a t statistics hypothesis test to conclude any differences between the models, mainly due to them being very small in nature, and any differences could be mainly by luck due to large variance.

Tables II and III both show the performance for each NgramTagger model on the test set using a 80:10:10 splitting ratio for training, development and test set respectively. We can see that as N increases, we do get an improvement in all metrics (except key similarity). This can indicate that our dataset is bottlenecking our performance. However, it seems that the new metric shows optimal performance for the TrigramTagger model whereas the second best is the (N=4)gramTagger model. One way to confirm the optimal

Algorithm 2 Music Synthesis

```
    Inputs $\mathcal{W}$, $\mathcal{N}$, $\Delta\mathcal{T}$, $F_s$          ▷ size $m$
    Output $y[n]$                           ▷ Audio Stream Array
    Variable Initialization $y \leftarrow zeros(\Delta t_m \times F_s)$
    procedure GENERATE STREAM($\mathcal{W}$, $\mathcal{N}$, $\Delta\mathcal{T}$, $F_s$)
2:      $\mathcal{F}$ = {Note2Freq($n_i$) $\forall i \in \{1, 2, ..., m\}$}
        $\mathcal{D}$ = {Str2Int($\Delta t_i$) $\forall i \in \{1, 2, ..., m\}$}
4:      $t_{end} \leftarrow$ sum($\mathcal{D}$)
        $k_{tot} \leftarrow$ ceil($t_{end} \times F_s$)
6:      initialize array $audio\_stream$ of size $k_{tot}$
        initialize $i = 0$
8:      for each $w, f, \Delta t \in \mathcal{W}, \mathcal{F}, \mathcal{D}$ do
            $w\_size \leftarrow$ round($\Delta t \times F_s$)
10:         $x \leftarrow \{sin(2\pi f i / F_s)\} \quad \forall i \in \{1, 2, ..., w\_size\}$
            $x \leftarrow x \times$ $window\_function(w\_size)$
12:         $audio\_stream[i : i + w\_size] \leftarrow x$
            $i \leftarrow i + w\_size$
14:     end for
    end procedure
16: return $audio\_stream$
```

model, is by computing k-fold cross validation and average the metrics to later perform a t-test.

With the goal to improve the performance of our model, we later computed a tuning curve on the cutoff value parameter in the NgramTagger to select the one which optimizes the test-set performance based on the aforementioned metrics. By fixing $N = 4$, we computed the tuning curve (found in **Figure 5**), which shows an obvious maxima at cutoff = 1 for the notes model, and unfortunately no obvious maxima for the durations model. After some calculations, it shows that there exists no maxima for either metric for the durations model, and therefore we chose the first one. We based the hyperparameter selection on the metric with the highest variance, which was the precision metric. The cutoff values were then calculated to be 1 and 0 for the notes and durations model respectively. To be more exhaustive in the search for our optimal parameters, we perform a grid search on the hyperparameters N and cutoff C over the range $N \in \{1, 2, 3, 4\}$ and $C \in \{0, 1, 2, ..., 9\}$ separately for notes and durations resulting in the heatmap shown in **Figure 2 (b) and (d)**. We notice for the notes model that the optimal parameters are $N = 3$ and $C = 1$, and for the durations are $N \in \{1, 2, 3, 4\}$ (4 chosen) and $C \in \{0, 1\}$ (0 chosen). By using the tuning curves previously calculated (**Figures 2 and 5**), we can notice that the training set performance curve tends to increase as N increases when tuning N, however, tends to decrease as the cutoff value increases. This is the main reason why we chose the hyperparameters for the duration model when there were conflicts within the optimal values. **Table IV** shows the final performance of our tuned model for the chosen optimal parameters $\{N = 3, (1, 0)\}$ tested against the held-out set, however, we don't seem to get much improvement. In fact, there seems to be a significant decrease in performance for all metrics. This is probably due to the random nature of 'interpolating' between the OOV terms that were unsuccessfully tagged by the model, and would yield some differences in the metrics shown in **Tables II, III and IV**; this was evident when running the algorithm multiple times. Given the variation in metric calculations, we can conclude that there seems to be minimal to no changes when tuning the cutoff hyperparameter in our model. The decrease in performance can be explained by the following:

- Dataset is too small
- Dataset is too variable, not much can be learned from it
- Music is creative
- A more sophisticated metric is required to test against

We therefore try to develop a metric to better represent the NLP task to convert text to music.

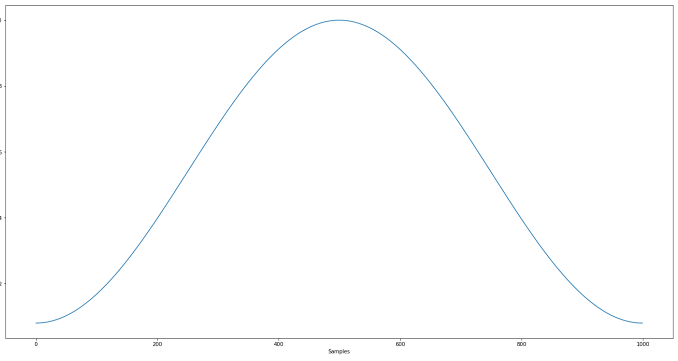


Fig. 3: A 1000 sample sized hamming window used to smoothen the sinusoid to give an enhanced tone

B. Key Similarities

Since correct music is not necessarily something absolute (i.e. playing note n instead of note w might be equally as good), a difference metric wouldn't be a suitable performance indicator. Since a song can be characterized by the Key it's in, we could estimate the key for both songs, and measure the degree of similarity between these two keys to know how different the songs are. We used the naive similarity metric, which is merely a boolean value; whether Key 1 equals Key 2. To make things more precise, we windowed the set of tokens (our case, $M = 10$) with a step size of 1, computed the keys for both the predicted note stream and the ground truth, and performed the naive AND operation on both streams. The degree of similarity is merely the sum of this new vector normalized by its length. It's important to note this metric is only suitable for the notes prediction stream, and not for the time duration stream. The last column of **Table II** demonstrates the key similarities for each NgramTagger model. We can see that the TrigramTagger obtained the optimal key similarity metric whereas the (N=4)gram obtained the second best key similarity metric. This is inline with the results found in **Figure 2**. As mentioned previously, there also seems to be some variation when calculating the key similarity metric due to the random nature of assigning tags to unknown predictions from our model. This would mean that the (N=4)gram could be the optimal model. Multiple runs and averaging should be considered. This was not possible due to the computational complexity of computing the key using the *music21* module.

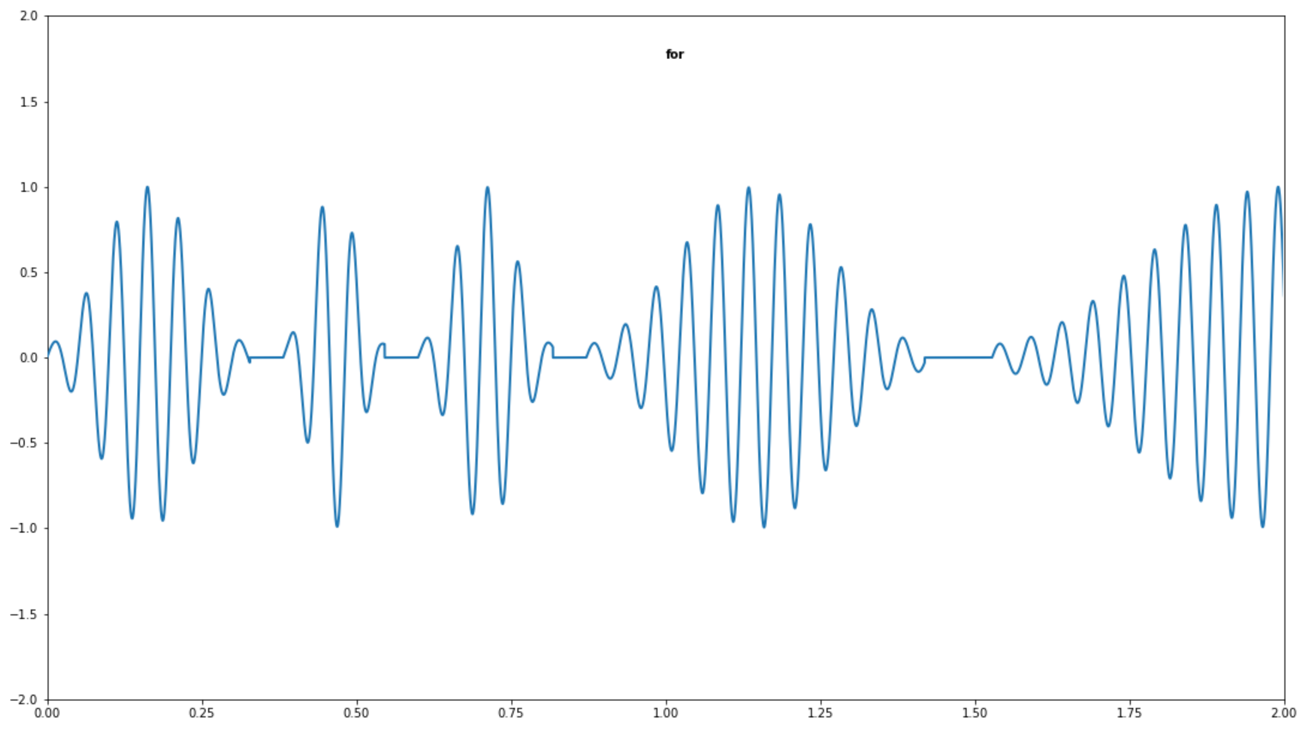


*Fig. 4: A visual example of **Algorithm** 2, however, note that the true frequency is modulated to a lower frequency for visual purposes.*

VII. Music Synthesis

We found two different modules for playing music; the basic one using *winsound* and the more complex one using *music21*. The reader must take note that the winsound module only works for windows machines. Music21 is a computational tool on Python for researchers in music theory. It has a module for generating an MIDI stream to play a set of notes with their durations. The winsound module plays music by either extracting a .wav file (or other formats), or generates a 'beep' at a certain frequency and lasts for a number of milliseconds. Music21 can play the keyboard given a set of notes with their durations, composing beautifully sounding music compared to winsound, however the duration of the notes is not absolute. This means that we can't specify how long each note lasts per second, but only how long a type of note lasts with respect to a time signature, tempo and the note type (full note, half note, quarter note etc.). This is a bit more complicated to process with, and after some experimentation we weren't able to generate music with the correct timings, and therefore resorted to the winsound module.

After thorough testing, we realized that the winsound module doesn't perform too well in regards to maintaining tempo and respecting the time durations for each note. This is probably due to an amount of delay caused by the repetitive on/off switching of the speakers. To develop a work-around, we resorted to manually generating the audio stream by computing the pitch, or fundamental frequency of the notes to naively generate the beep sound using a simple sinusoid with the given note frequency as follows:

$$y[i] = sin(2\pi f i) \tag{1}$$

For note n_i. Namely, given a reference note n_0 (typically C_0), we can calculate the frequency as follows:

$$f = n_0 2^{\frac{N}{12}} \tag{2}$$

By default, we assumed a sampling frequency $f_s = 44.1 \times 10^3$. This value can be changed by the user when calling the *play* function, or the *gen_audio_stream* function under the *play_song* module. For each given duration, we calculate the window size to multiple with the sin wave to generate the note and it's duration, and append it to the audio stream. The algorithm to generate the audio stream can be found in **Algorithm 2**. The inputs to the algorithm are the outputs of the taggers. Therefore, the notes are a set of ordered strings describing the note and the octave at which this note will be played in, the duration are a set of ordered strings indicating the absolute duration in ms to which each note will be played. For this information to be useful, we convert the notes to frequencies using (2) and store them in an array and we convert the durations in strings to integers and divide by 1000 to convert the units to seconds.

To generate a simple beep sound, the window function in line 11 of algorithm 2 is set to 1 for all samples $i \in \{1, 2, .., w\_size\}$. To generate a more sophisticated tune, we

TABLE II: Performance Metrics for Note Classification

Model\Performance	**Accuracy**	**Precision**	**Recall**	**F1-Score**	**Key Similarity**
UnigramTagger	0.12430518502592276	0.1787664793872379	0.12430518502592276	0.13231488295669733	0.20021131343018853
BigramTagger	0.12167214046840706	0.20172357254761739	0.12167214046840706	0.13679366668151247	0.16541971995257956
TrigramTagger	0.14728295368096786	0.20151480822915105	0.14728295368096786	0.15816314742075363	0.24439620239048132
NgramTagger (N=4)	0.159682093099102	0.2181802944870629	0.159682093099102	0.1729639042209652	0.2253941875095232

TABLE III: Performance Metrics for Duration Classification

Model\Performance	**Accuracy**	**Precision**	**Recall**	**F1-Score**
UnigramTagger	0.0016064211434539757	0.03221188696984528	0.0016064211434539757	0.0028479056427378486
BigramTagger	0.0054466669191836235	0.04143172189783855	0.0054466669191836235	0.008336684545867569
TrigramTagger	0.00851886353976734	0.040433257996148844	0.00851886353976734	0.010603728187385262
NgramTagger (N=4)	0.00851886353976734	0.040433257996148844	0.00851886353976734	0.010603728187385262

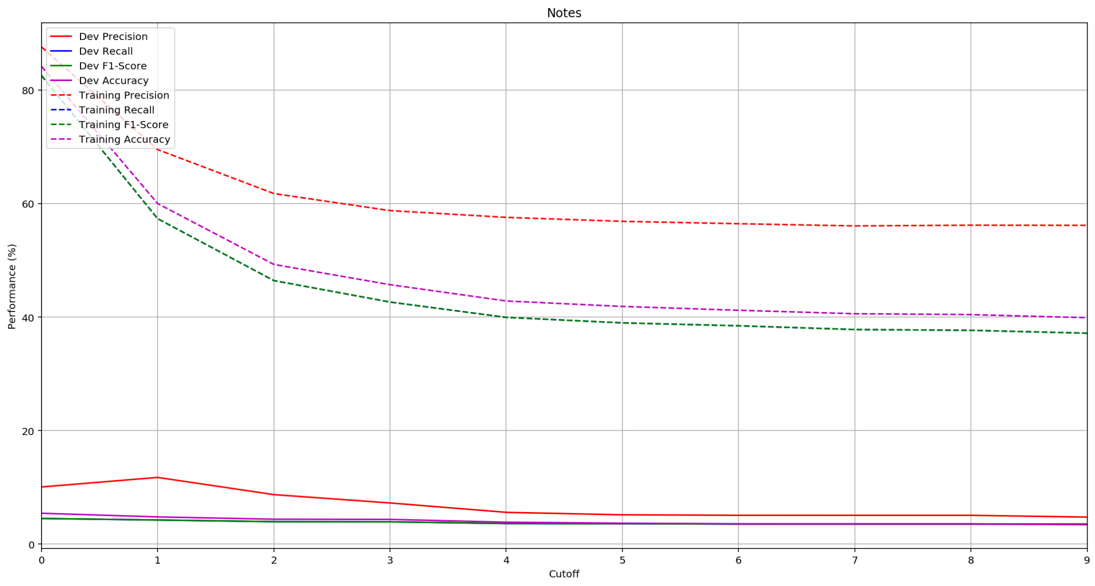
TABLE IV: Performance Metrics for Notes Classification with Cutoff = 1

Model\Performance	**Accuracy**	**Precision**	**Recall**	**F1-Score**	**Key Similarity**
UnigramTagger	0.101873333997038074	0.14963209602120986	0.101873333997038074	0.11176503449757731	0.14982109451739187
BigramTagger	0.10276686765292793	0.17073808873999008	0.10276686765292793	0.11945165831665931	0.16048252164111457
TrigramTagger	0.1113264350159413	0.17745601422811133	0.1113264350159413	0.12836901064305814	0.17256930166574927
NgramTagger (N=4)	0.11743343210969058	0.18474641550806173	0.11743343210969058	0.13447511130526782	0.18845414054484552

can consider using a hamming window (shown in **Figure 3**. After multiplying the sinusoids with the hamming window, as we expect from the smooth edges of this window, we get a smooth transition from note to note which is pleasing and produces a tone of higher quality. We then playback the audio using the *sounddevice* module, which takes as inputs the generated audio stream and the sampling frequency. An example of a song being played with the algorithm and hamming window can be found in **Figure 4**.

VIII. Conclusion

We can conclude this project by saying that although we have developed a model which can successfully play some melodies given some textual inputs under the aforementioned performance metrics, we believe that we can produce more accurate models by looking into more complex supervised learning approaches. We could add more complexity to the NgramTagger by considering the previous words as well. Support Vector Machines (SVM) would be a nice model to look into in the future of this project although we were not able to produce and test this model due to time constraints. Our current dataset is not sufficient to train any model with great performance, so we would have to look into ways to scrape online resources to expand our dataset, in specific, build 'The Beatles' dataset to contain a large enough set of songs to train with. One way we dealt with the scarcity of 'The Beatles' songs, was to include similar artists. We believe that this may have a factor to the poor performance of the models, since although these artists are similar, they have their own language styles. Such changes do not correlate well with the classes of interest (i.e. notes and durations), and therefore yield low performances. We computed a performance curve and an exhaustive grid search using the performance on the development set in hoped to improving the performance of our model with respect to the baseline (N=4 with no cutoff) by selecting the optimal model, as well as computing a performance table with the new metric. Both seem to show differences as the TrigramTagger was optimal in **Figure 2**, whereas the NgramTagger (for $N = 4$) was optimal in **Table II**. This can be explained due to statistical errors as the differences between the optima and the other points are quite small, and given the little set of data indicates larger variance and therefore doesn't provide a confirmation of the given result. We later synthesized the predicted notes and durations by generating sinusoids, each one multiplied by a hamming window to produce a higher quality tone than that provided by winsound using *sounddevice* to synthesize the generated audio stream.



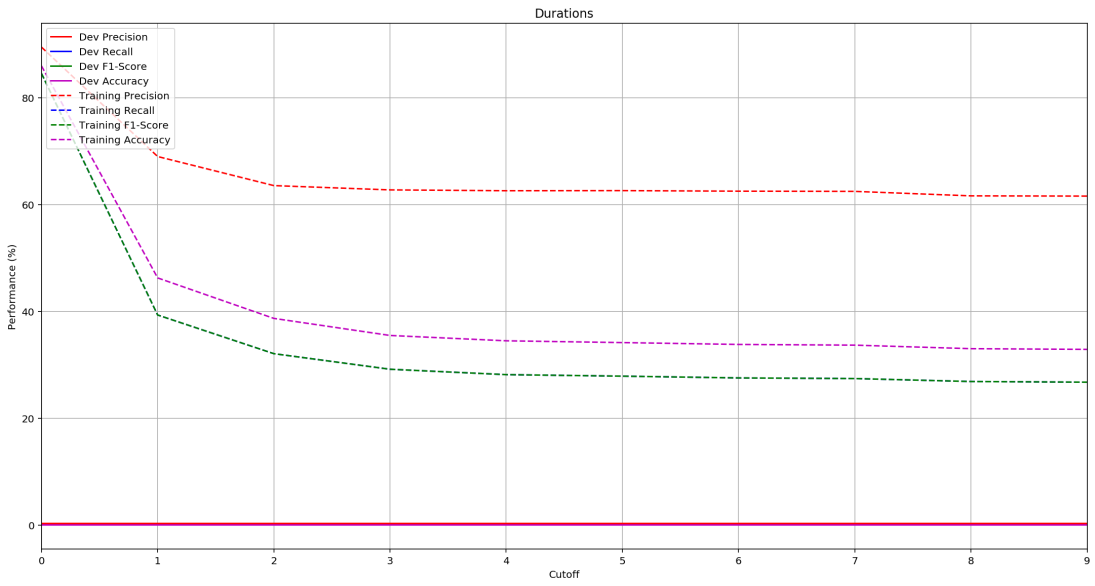


Fig. 5: Cutoff hyperparameter tuning curves for the notes and duration models. An obvious maxima is shown for the notes model as cutoff=1 and none for the durations model.

References

[1] gabolsgabs. *DALI: a large Dataset of synchronised Audio, LyrIcs and vocal notes.* 2019. URL: https://github.com/gabolsgabs/DALIA (visited on 12/26/2019).

[2] Jan Willem Kolkman. *LangoRhythm 2.0 - Text to music generator*. 2015. URL: https://kickthejetengine.com/langorhythm/ (visited on 12/26/2019).

[3] Gabriel Meseguer-Brocal, Alice Cohen-Hadria, and Geoffroy Peeters. “DALI: a large Dataset of synchronized Audio, LyrIcs and notes, automatically created using teacher-student machine learning paradigm”. In: (June 2019).

[4] TEDx Talks. *LangoRhythm, every speaker is a musician — Jan Willem Kolkman — TEDxMaastricht*. 2015. URL: https://www.youtube.com/watch?v=N7a1Ua6kvcA (visited on 12/26/2019).