Instrumental Solo Generator

Prithvi Ramakrishnan, Aditya Dev Gupta
{prithvir, gaditya}@stanford.edu

Abstract

Instrumental Solo Generator[2] allows musicians to tap into a musical vocabulary that extends beyond that which they have built on their own. The system accepts a chord progression as an input and outputs musical ideas in the form of a MIDI file. A clustering algorithm is used to reduce the sample space of chords and musical ideas, after which a SVM with a linear kernel is used to suggest musical ideas for chord progression inputs. These musical ideas are intended to be used as a starting point for exploration in the composition or improvisation process. *Instrumental Solo Generator* does not aim to replace the human performer, but merely serve as a tool for idea generation as part of the music development process.

1 Introduction

The jazz and blues genres of music are grounded in improvisation, an aspect that can sometimes be deterrent to young musicians seeking to transition from the relatively straightforward classical genre. Successful jazz and blues performers often spend years in training developing an ear for thematic ideas and motifs that can be used as part of an improvised solo by accessing a personal vocabulary of musical ideas.

Clearly, there is an aspect of "learning" in the process of maturing into a successful jazz performer. Musicians encounter hundreds, or even thousands, of phrases that become a part of their vocabularly, and then selectively unleash these ideas in response to a given musical stimulus. The impetus for this project stems from the belief that the concept of learning from prior exposure to music to generate improvised musical ideas can be successfully applied to a machine learning system.

2 Dataset

2.1 Data Retrieval

This project uses MIDI files as input data. MIDI provides a precise representation of pitch and duration, datapoints that are key components of the feature space and solo generation process. MIDI transcriptions of blues and jazz performances are ubiquitous on the Internet, making data retrieval fairly straightforward.

MIDI data was scraped from an online repository[1] of jazz and blues pieces using `wget`. We avoid manually selecting MIDI files in order to preserve the objectivity of the dataset, because the driving premise of this project is to provide an unbiased realization of instrumental solo ideas from a diverse musical vocabulary. The dataset includes 607 MIDI files containing a diverse collection of jazz and blues pieces from a wide range of genres, artists, styles, and time periods, thereby diversifying the training data.

2.2 Data Processing

Music21 is a Python-based toolkit for computer-aided musicology that enables scholars and other active listeners to answer questions about music quickly and simply[1]. Music21 enables the representation of symbolic music as objects in the form of a musically intuitive hierarchial structure. This was instrumental in allowing us to parse MIDI files into training instances.

[1] http://midkar.com/jazz and http://midkar.com/blues

We use music21 to design a `ScoreParser` class that converts a MIDI file to a `Stream` object. The `Stream` object is the fundamental container of musical eleemnts in music21. Key subclasses for this project are the `Measure` and `Score` objects. Hierarchically, music21 returns a `Score` object as the top-level representation of a MIDI input. The next level of the`Score` object is a `Part` (typically an instrument, such as Piano, Saxophone, or Bass Guitar), followed by an optional decomposition into one or more `Voice` objects (a subset of a musical part, such as the treble and bass clefs in a piano part, or SATB breakdown for a vocal part). Each part or voice thereon contains an array of time offsetted `Note` objects.

We define *measure* to represent a segment of time defined by a given number of beats, each of which are assigned a particular note value. Dividing music into measures provides regular reference points to locations within a piece of music. Unfortunately, the `Score` object parsed from a MIDI file is not divided into measures. To this end, we invoke music21's `makeMeasures` method upon every `Part` to allow our data to represent reference points to time.

3 Features and Preprocessing

3.1 Feature Representation

Although the feature representation ultimately used in the clustering algorithm is derived from the raw data, it helps to understand the feature representation from a high-level perspective.

We obtain raw training instances by iterating through every measure of a MIDI file and extracting `Chord` objects for the preceding, current, and succeeding measures, along with a "musical idea" mapping expressed as an array of and `Rest` objects. Note that the sheet music diagram is a simplistic representation of our raw dataset. In reality, an individual measure may have multiple representative chords and a number of musical ideas (each contributed by the various parts/voices present in a measure). To take this into account, and also expand our feature space, we performed what is effectively a cross product over the space of feature arrays and musical ideas for a given measure.

Our feature vector can therefore be defined as $[\texttt{Chord}_{-1}, \texttt{Chord}_0, \texttt{Chord}_1]$, where the subscript of `Chord` denotes its position (in measures) relative to the measure represented by the training instance. The edge cases to the iterative approach of obtaining training instances are the first and last measures. In these cases, we express $\texttt{Chord}_{-1}$ and $\texttt{Chord}_1$ as `NoneType`, respectively.

3.2 Preprocessing

The object-based representation is effective in verifying the appropriateness of the training instances from a musical perspective. However, it is essential to obtain a numerical representation of the feature space in order to perform the clustering step. The numerical representation employed in this project captures the intervals between the musical pitches in chords and music ideas; this forms our preprocessing step.

Music21 allows one to obtain a numerical representation of the pitch of a note, regardless of octave, using `Note.pitchClass`, that is to say, `pitchClass` returns 0 for both notes `C4`and `C5`, and so forth. A key requirement of the interval representation of notes is to be able to significantly distinguish musical ideas to ensure that they are clustered into different buckets when appropriate. The naive approach would be to represent intervals as an array of the difference between the pitch classes for consecutive notes. However, this would result in the musical ideas [`C4, E4, G4`] and [`C4, E-, G4`] being represented as [`4, 3`] and [`3, 4`] respectively. From a musical perspective, the ideas represent C-major and C-minor respectively, but the interval representation does not succeed in depicting these ideas as dissimilar, as it should.

To that end, we devised a *circle-of-fifths representation* to allow the interval array to incorporate tonal dissimilarities between musical ideas. The circle-of-fifths representation uses the intervals shown in *Figure 2*, and also uses a heuristic that enables us to avoid the disparity between the two relatively similar musical ideas, [`C4, E♭4`] and [`C4, E4`], that would be represented as [4] and [9] respectively, per *Figure 2*. This heuristic is detailed as folows:

Figure 1: Features

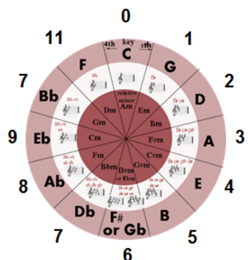

Figure 2: Circle of Fifths

We define the circle-of-fifths mapping (zero-indexed) array as $C = [0, 7, 2, 9, 4, 11, 6, 1, 8, 3, 10, 5]$, which maps pitch classes to their locations on the circle of fifths. Then, we define the difference between two notes n_1 and n_2 as

$$d(n_1, n_2) = r_{12}(C_{p(n_1)} - C_{p(n_2)})$$

where $r_k(n)$ is the remainder function programmatically conventionally represented as `n % k`, and $p(n)$ maps a note to its pitch class, indexed to $\texttt{C} = 0$.

Then, to get the circle of fifths representation of a musical idea, consisting of notes $(n_1, \ldots, n_k)$, we simply compute:

$$R(n_1, \ldots, n_k) = (d(n_1, n_2), d(n_2, n_3), \ldots, d(n_{k-1}, n_k)$$

4 Models

4.1 Clustering

The motivation for clustering[3] the chords is that even across hundreds of musical samples, it is unlikely that there will be many repeated musical ideas, note for note. Even when we normalized for key, we found that nontrivial sequences of notes (those with at least three or four pitches) were rarely duplicated. Therefore, if we wanted to apply some algorithm to pick musical ideas based on the chord progression around a given measure, we would essentially be creating an augmented Markov model, with little room for sophisticated machine learning techniques.

However, intuitively, some musical ideas are very similar, while others are quite disparite, so, assuming we can find a reasonable measure for this degree of separation between two musical ideas, we could apply a clustering algorithm to decrease the previously huge set of raw musical ideas into a finite (and fairly small) set of distinct chords.

We also clustered the chords found in midi files as well. Particularly in jazz, there are numerous different voicings for a single chord (different combinations of notes that harmonically indicate a given chord). Musicians will often add a ninth, substitute a sixth, and dynamically play other variations of chords than may be written on the score, even though they indicate the same harmonic progression. Hence, clustering chords into distinct forms will be just as useful a tool as for solo ideas. The very same clustering algorithm, in fact, is used to cluster chords.

Note that this collection of chords and musical ideas are all context-free, in that there is no connection between chords and ideas yet, and the actual mapping from sequences of chords to ideas, which create a context-aware environment for creating solos, is prepared in the *Classification* step.

The actual algorithm is constructed as follows.

First, we separate musical ideas into groups of n notes (here, we used $n = 4$). This is to facilitate constant dimension in the points, in order to run k-means. We then construct the circle-of-fifths representation of each musical idea, which is a point in $\mathbb{R}^{n-1}$. We cluster on these points, using k-means (here, we used $k = 100$). Finally re-map these cluster assignments to the original musical idea objects, to get our clusters. For chords, the algorithm is essentially the same, with the slight change that we use $k = 20$, since after normalizing for the root, there are much fewer distinct chord types than melodic ideas.

4.2 Classification

Once the clustering algorithm has mapped each idea into clusters, we apply a classification algorithm to map feature vectors, represented as chord progressions, to algorithms. Formally, let the set of musical ideas be I, and let the clusters of musical ideas be $I_1, I_2, \dots, I_{k_i}$, where we use $k_i = 100$ and $\bigcup_{i=1}^{100} I_i = I$, and let $S_I = \{I_1, I_2, \dots, I_{k_i}\}$. Similarly, for chords, we let the set of chords be C, the clusters be $C_1, C_2, \dots, C_{k_i}$ where $k_c = 20$ and $\bigcup_{i=1}^{100} C_i = C$, and let $S_C = \{C_1, C_2, \dots, C_{k_i}\}$.

Now, formally, our classificaiton algorithm is an algorithm that models a mapping from $S_C \times S_C \times S_C \to S_I$, thus mapping a combination of chord clusters to a musical idea cluster.

The algorithm we used to do this was SVM with a Linear kernel. We experemented with some other algorithms, which are described in more details in *Results*.

Note that since each training point is a sequence of chords and notes, centered by a measure, we get a training point for each measure of each MIDI file, generating a large amount of training data.

4.3 Prediction

Combining these two models for the music, we have a machine learning algorithm that produces a cluster of music ideas given a sequence of chords. Now, to produce actual music, we have to construct sequences of notes and rests from this sequence of clusters of chords. Since this operation does not involve any machine learning, we used a fairly naïve approach for this, simply selecting an arbitrary member of the cluster of musical ideas selected that structurally fits into the structure of the measure being constructed. This structural check is done simply with a rhythmic, avoiding sequences of rests and rhythmic inconsistencies that would make the music sound choppy.

5 Results

Since the quality of the music cannot be objectively numerically analyzed, we tested the quality of the classification algorithm. To do this, we first broke up the data in half by songs. Specifically, we decided not to break it up by training points after randomizing. Since each song has a chord progession that's persistent throughout the piece, it's likely that many training points will overlap. Then, the testing set will consist of feature vectors that are from the same song as some feature vectors in the training set, making the performance of the algorithm extremely optimistic. Partitioning training set and the testing set by song avoids this fallacious behavior.

Once the data was partitioned, we trained the classification model on the training data. For each testing instance, in each measure, we simply compared the cluster index generated by the classification algorithm to the actual cluster of the musical idea presented in the measure, as would be done in a normal performance analysis of a classification algorithm.

6 Discussion

Using a SVM with a Linear kernel for the classification produced the best results, with $\sim 18\%$ accuracy when trained on the small training set and $\sim 73\%$ accuracy when trained on the large training set. We can rationalize this vast difference fairly easily. With the small dataset, which contained only 2000 data points, ranged only over a few files, and so the musical vocabulary may not have been broad enough to cover the new musical concepts that would have been included in the testing set. However, once the training set was increased to around 18000 data points,

Table 1 Results

Model	Performance	
	Small dataset	Large dataset
SVM (RBF kernel)	0.1782	0.7271
SVM (Linear kernel)	0.1782	0.7271
SVM (Polynomial kernel)	0.0891	0.3148
Random forest (gini criteria)	0.0977	0.5483
Random forest (entropy criteria)	0.1063	0.6301
Random forest (5 estimators)	0.1006	0.4846
Random forest (10 estimators)	0.0862	0.5180
Random forest (20 estimators)	0.1264	0.5361
Random forest (50 estimators)	0.1149	0.6665

the musical vocabulary was complete enough to determine the cluster of solo ideas fairly accurately.

7 Conclusion

Instrumental Solo Generator succeeds in transforming a large, diverse repository of MIDI jazz pieces into a feature space that is capable of suggesting musical ideas given a user supplied musical basis. The final product is capable of accepting a sequence of chords from the user as input, and outputting a sequence of musical ideas in the form of a MIDI file. This presents itself as a useful tool for jazz or blues composer or performer hoping to tap into a musical vocabulary that extends beyond his or her own.

8 Future Work

The current implementation uses naïve approaches and heuristics to determine the identity of the solo part for each measure, which is used to decide which instruments will be used to provide chords and which instrument will be used to populate solo ideas in different sections of the music. By using more intelligent approaches that aim to minimize an entropy value to prevent the solo part from changing frequently.

Several midi files also have a "Percussion" part, which has different pitches corresponding to different unpitched instruments. This adds significant noise to the solo ideas and chords clusters. Writing a script that removed all the percussion parts would have readily improved overall performance.

Finally, adding a feature for the genre of jazz that the piece is taken from would also improve results. Jazz solos that might sound good in a blues setting might not sound as good in a bebop or bossa nova style.

References

[1] Cuthbert, M. and Ariza, C. "music21: A Toolkit for Computer-Aided Musicology and Symbolic Music Data." in J. Stephen Downie and Remco C. Veltkamp (Eds.). 11th International Society for Music Information Retrieval Conference (ISMIR 2010), August 9-13, 2010, Utrecht, Netherlands. pp. 637-642 http://web.mit.edu/music21

[2] Gupta, A. and Ramakrishnan, P. "Instrumental Solo Generator". https://github.com/adityadevgupta/cs229project

[3] Pedregosa et al. "Scikit-learn: Machine Learning in Python" in JMLR 12, pp. 2825-2830, 2011. http://scikit-learn.org/stable/index.html