Extension on Description Based Design

Avneesh Sarwate
COS 2014
Spring 2013 Independent Work
Advisor: Rebecca Fiebrink

Abstract
In this paper I will describe my attempted improvements on Francois Pachet's Description Based Design, a system that uses machine learning to generate musical melodies. I will discuss "musically intelligent" tools, survey current music composition tools, and discuss in depth. both Description Based Design and my changes to the original system. I will also present a user study, discuss the results, and lay out the future work motivated by the user study.

1. Introduction

There has been significant work done in the development of software tools that help musicians compose music. Popular tools include sheet music and score based tools such as Finale and Sibelius, recording and mastering tools like Pro Tools and Reaper, and looping and sequencing tools such as Ableton and FL Studio. There has also been active research into creating computer applications with "musical intelligence" by combining machine learning, Markov models, and other mathematical tools with music theory. However, there have been very few tools that have been made that are both "musically intelligent" and easily usable by musicians who have no programming experience.

"Description Based Design of Melodies," a 2009 paper by Francois Pachet [11], outlines a *process* (best word?) that allows musicians to stochastically generate highly customized random melodies without requiring any knowledge of computer science or programming at all. As a proof of concept of such a system, Pachet, implements a simple version that only generates a limited type of melodies and runs some evaluations to determine the “quality” of the generated melodies.

For my independent work, I took the high level framework of Pachet’s system and redesigned and re-implemented its core components, aiming to improve the “quality” of melodies generated and to generalize it to handle any kind of melody a user would want to create. I attempt this by letting music theory research inform the design of key system components.

In this paper, I will present a brief survey of other “generative” musical composition software, explain Description Based Design (DBD), provide the details of my implementation, report the findings of a user evaluation of my system, and discuss the results and how they inform future work on this project.

2. Previous Work

We can think of existing musical composition software in terms of two parameters: “in-built (musical) intelligence” and “customizability.”

Systems with in-built intelligence are those that are designed using knowledge of musical rules and patterns, and that have the ability to apply these patterns in the modification or generation music. In general, the most popular software tools, such as the ones described by name in the introduction, have little to no musical intelligence. Notation tools such as Sibelius and Finale have very limited intelligence in that they can perform very basic transformations on sections of music, such as transposing the key of a section or inverting a melody.

Systems that express a high degree of musical intelligence are often research projects by computer music academics. The Continuator [10] is one such example. The Continuator is a real-time improvisational extension system that uses Markov models to recreate phrases (i.e, musical phrases, or, melodic segments) similar to those played by a human user on a real instrument. The user and the Continuator can trade musical phrases back and forth in real time. “Experiments in Musical Intelligence” is another system [5], created by David Cope, that used Markov models and pattern detection techniques to “learn” the styles of various classical composers and recreate music in their respective styles.

Customizable systems are those whose functionality can be changed. Music programming languages, such as Max-MSP [6], PureData [13], SuperCollider [9], ChucK [17] etc are all very customizable tools. The existence of various third party libraries makes it possible to perform almost any composition tasks using programming languages. Popular tools such as Reaper and Pro Tools have a high degree of customization available in terms of recording and signal processing, but contain nothing in terms of music generation. Digital audio work stations (DAWs) provide relatively open interfaces to connect with other systems, but provide little “customizability” within themselves, only the ability to be integrated into other systems. Overall, tools with a degree of “customizability” have a high learning curve for non-experienced users.

DBD aims to find a compromise between customizability, ease of use, and musical intelligence. It does this by limiting the area of “customizability” to the domain of one specific user task. The focus on a single task also allows for the potential to simplify and minimize the user interface.

3. Approach: Description Based Design

3.1. Process

3.1.1. Tagging

DBD presents a way to utilize machine learning to let users generate melodies without explicitly having to write them. First, the user tags the melodies produced by a random melody generator with whatever tags they desire (i.e. jazzy, funky, dissonant, smooth, etc). For each tag, a binary classifier is trained with the tagged examples constituting positive examples, and a subset of the melodies without that tag as the negative examples. The tagging process is the single task where the user can "customize." The user is free to build classifiers with any set of melodies they want.

3.1.2. Transforming

Then, for any of the trained classifiers (e.g., jazzy), the user can take that classifier and use it to transform some other melody to be more or less "jazzy". Thus, for each classifier, the user can essentially define a knob that produces "more x" or "less x" variations on a melody, where "x" is any musical adjective.

The transformation of an original melody occurs in the following way: a large number of variations of different depths are created on the source melody through the following process. Starting from a set of variations containing only the original source melody, a random melody is selected from the set, the variation function is applied on that melody, and the new melody is added to the set. Out of these variations, the variation that maximizes some function of "jazziness" (the probability of class membership to the positive "jazzy" class) and "similarity to the source melody" is selected to be the final result of the transformation. Since the binary classifiers can return a probability of class membership, this can be leveraged to allow the user to determine specifically how much more or less "jazzy" to make the melody (e.g., 10% more, 50% less) as well.

The three main components to this system are the random melody generator, the variation function, and the feature representation of the melodies.

3.2. Goals for Improvement

While DBD is a very interesting approach to creating melodies, the details of the implementation and evaluation given in the original paper seem focused on a very simplistic set of melodies. The tags that are used in the evaluation presented in the paper are special in that they are for the most part easily quantifiable. Four out of the five tags: ”brown", "serial", "short", and "long" are actually defined mathematically: "Brown" melodies are those with small intervals, "serial" melodies are those where all notes occur with equal frequency, and "long" and "short" are just melodies with "more" or "less" notes. Thus, it is easy to define a feature set that extracts the data necessary to classify these types of melodies.

More abstract tags such as "flowing" or "funky" are not so readily quantifiable, and thus it is harder to determine a feature set that would be useful in classifying these types of melodies. However, it is exactly these hard to define tags that would be most interesting to use DBD wjth when writing music.

Thus, one of the goals for this project is to implement a more robust version of a DBD system for melodies, one that can be used with all kinds of melody tags. I will try to find a feature set that represent melodies in general, a random melody generator that can create a wide range of melodies, and a variation function that can execute more types of changes than the simplistic one presented in the original paper (the variation function the original paper does not take into account rhythmic variation).

Another goal for the project is to create an interface that expedites and simplifies the process of using DBD for users with no background in computer science, programming, or machine learning. The original paper does not present a complete interface for using DBD.

The overarching goal is to take DBD from a well-defined, but high level, concept to a functioning and generally useful tool in the domain of melody writing.

4. Implementation

4.1. System Architecture

This system was built with the following software components. wxWidgets [19]: a wrapper to the wx C++ widgets library for Python. Music21 [2]: a music analysis and visualization library for Python. ChucK [13]: a live-coding language for music applications. pyOSC [14]: a Python library for the OSC network protocol. Weka [18]: a Java machine-learning library. Ableton [1]: a proprietary, professional grade digital audio workstation.

The wxWidgets library was used to build the GUI, and music21 was used to help with feature extraction. Ableton was used to play the audio. When a user would hit the "play" button, melodic information would be sent from the Python application to ChucK over OSC (using pyOSC). ChucK would then send the information to Ableton as MIDI messages. When training or classifying melodies, the Python application and the Weka classifier would exchange information via text files.

4.2. Machine Learning Features

The features I selected to represent the melodies were based upon the work of Professor Dmitri Tymoczko [16] Tymocsko argues that there are 5 fundamental properties that all Western tonal music has. These properties are:

1. *Conjunct melodic motion. Melodies tend to move by short distances from note to note.*
2. *Acoustic consonance. Consonant harmonies are preferred to dissonant harmonies, and tend to be used at points of musical stability.*
3. *Harmonic consistency. The harmonies in a passage of music, whatever they may be, tend to be structurally similar to one another.*
4. *Limited macroharmony. I use the term "macroharmony" to refer to the total collection of notes heard over moderate spans of musical time. Tonal music tends to use relatively small macroharmonies, often involving five to eight notes.*
5. *Centricity. Over moderate spans of musical time, one note is heard as being more prominent than the others, appearing more frequently and serving as a goal of musical motion.*[Tymoczko, 4]

For the melodic representation, properties 1, 4, and 5 were used. Also used were the originally devised rhythmic analogues of these three melodic properties.

1. Conjunct rhythmic motion. Adjacent notes in a melody tend to have the same or similar durations.
2. Rhythmic self-similarity. Over a moderate span of musical time, melodic phrases have the same or similar rhythms.
3. Rhythmic centricity. Notes tend to fall on the "stronger" beats of the measure (e.g, the 1 and the 3 in 4/4 time).

All melodies were represented as (noteValueList, durationValueList) pairs. The note values were the MIDI integer representations of the notes in the melody (60 = middle C, half steps are size 1). The duration value of a note was how many quarter notes long it was. For example, a quarter note would have the value 1, a half note the value 1, and a 16th note the value .25. The properties were quantized as follows:

1. Melodic Conjunctness: the average of the sizes (positive) of all intervals between notes.
2. Limited Macroharmony: The fraction of notes in the most likely key divided by the variance of the MIDI note values.
3. Melodic Centricity: The faction of notes that are the 1, 3, and 5, degrees in the most likely key.
4. Rhythmic Conjunctness: the geometric average of the >1 ratio between consecutive durations.
5. Rhythmic Self-similarity: The average "difference" between measures, as measured by the Levenshtein distance on the duration lists of the measures.

6. Rhythmic Centricity: The fraction of “important beats” where a rhythmic hit actually occurs (1 and 3 in 4/4, 1 in ¾).

The original DBD paper used 7 features, so it seemed appropriate to start with a feature set with a similar number of dimensions.

4.3. Random Generator

The random melody generator was made of these sub-components:

4.3.1. BestKey

This is a function to find the most likely key that a musical phrase is in. For each of the 12 major keys, the function assigns a utility value to each note based on what scale degree the note is in that key. It then calculates the “key utility” by summing the utilities over all notes in the phrase. It then selects the key with the greatest key utility.

4.3.2. Morph

The morph function was designed to create random “noise” on melody segments. The term “noise” in this domain meant any random change in the musical phrase that the noise is being applied to. Changing the pitch of a note, changing the duration of a note, and adding or subtracting notes are the different types of changes that the Morph function can enact on a phrase. In addition, the changing of notes can be done such that the new note is in the same key as the phrase (or, if the phrase is not entirely in any one key, the most likely key as calculated by BestKey). The probability of the different types of transformations occurring, as well as the probability of a transformation being key preserving or not, can be set by the user.

4.3.3. Hitvectshake

This function adds "noise" to a given rhythm. This function first translates the "duration list" representation of a melody into an "onset vector" representation, as used by Post and Toussaint [12]. An onset vector is a vector of 1's and 0's, where the ith element represents the ith quantization element of the rhythm, and is set to 1 if a "hit" occurs at that time and 0 otherwise. For example, in a rhythm representation where 16th notes are the smallest level of quantization, a straight beat in 4/4 with hits on the quarter notes would be represented by a 16 element vector with 1s on the 1st, 5th, 9th, and 13th elements. After converting the given rhythm to an onset vector, it selects a random onset and moves it at random forward or backwards 1 space, provided that there is not another onset occurring at that time. For example, "1 step" of noise applied to the straight 4/4 beat could result in an onset vector with 1s at the 1st, 6th, 9th, and 13th elements.

4.3.4. Smooth

This is a function that "flattens" a melody by removing solitary jumps. Given three sequential notes a, b, and c, (where a, b, and c are all integers representing MIDI notes) and some integer limit d, the algorithm looks for a note b such that $|a-b| > d$, $|b-c| > d$, and b-a and c-b have opposite signs. These are the conditions for a "solitary" jump, a note that deviates significantly from the range of the melody without the rest of the melody moving with it. Such notes are discordant and very rarely contribute to "good" melodies. The Smooth function takes such melodies and moves them back "closer" to the range of the melody

Two types of random melody generators were implemented in this project:

4.3.6. Rand1:

The first random melody generator relied solely on the Morph function. Starting with a “flat” phrase (one where all of the notes are the same and all of the notes have the same duration), the Morph function is applied for k rounds, where k is 5 times the number of notes in the phrase.

4.3.7. Rand2:

First, either ¾ or 4/4 is randomly selected as a time signature. Then, a random onset vector is created that is 1 measure long. This single measure is concatenated 4 times to create the rhythmic “frame” for the whole phrase. The hitvectshake function is applied a random number (uniform(0-6) of times. A “flat” melody (where all notes are the same pitch) is then added on top of the rhythmic frame, and the Morph function is called a random number of times (uniform(2*k—5*k) where k is the length of the phrase), so the noise aggregates into a distinct melody. The noise is applied so that, on average, 90% of the notes are in key. The melody is then flattened.

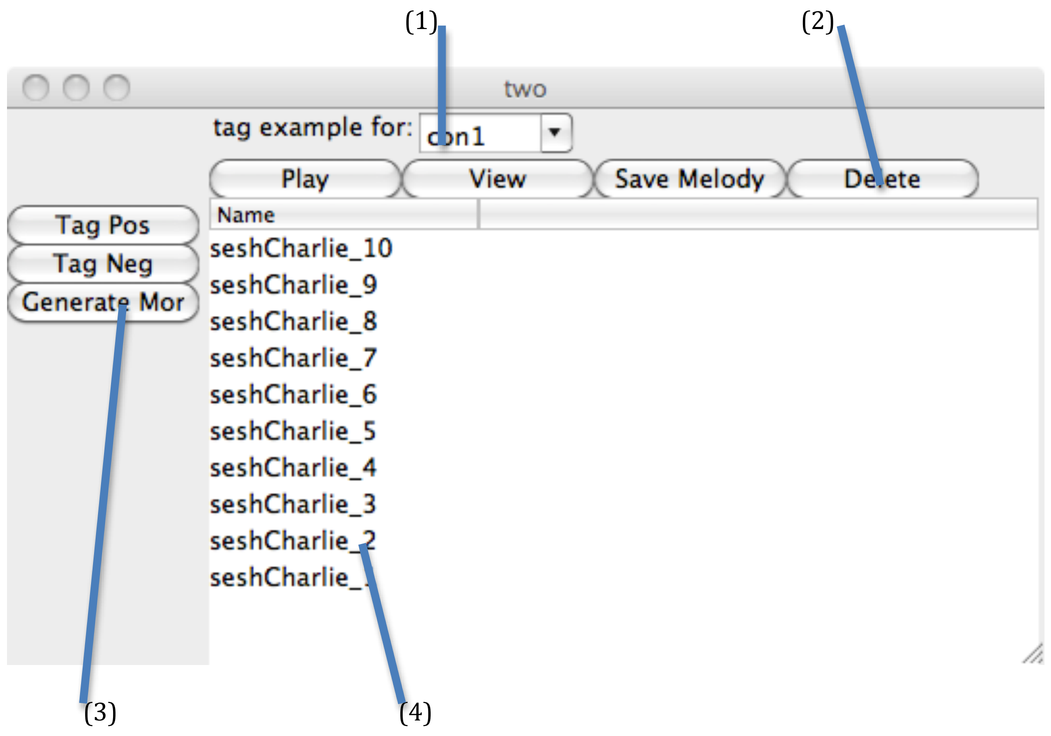
4.4. Interface

There are three main screens to the interface: The generating window, the training window, and the transforming window. Users can tag randomly generated melodies in the generating window, tweak the training set and train the learner in the training window, and create variations on melodies based on learned classifiers/tags in the transforming window.

The interface presents a slight change on tagging from original DBD system (described in section 3.1.1). It allows users to either positively OR negatively tag a melody. For example, a melody could be tagged as either “jazzy” or “not jazzy”. Then, only melodies tagged with some form of “jazzy” would be used to train the “jazzy” classifier.

Users could also write their own training melodies in Ableton using either a mouse/keyboard and/or a MIDI keyboard, export the melody to midi files, and then drag and drop them into the interface. The melodies must be properly quantized, or else the feature extraction will not work properly.

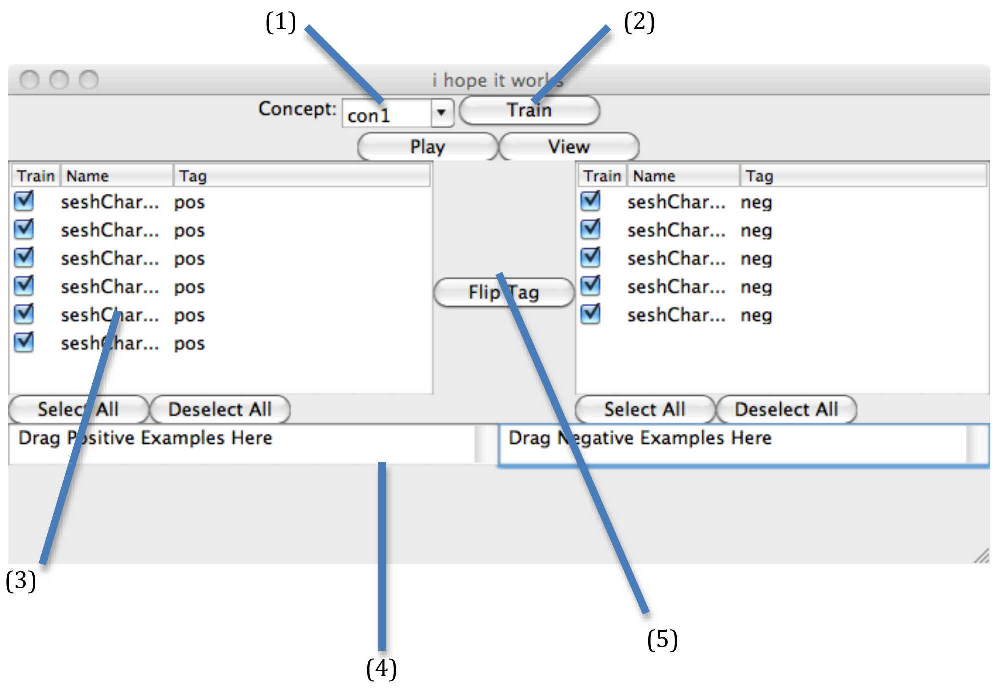
4.4.1. Generating window



1. The dropdown menu for selecting which tag name to tag the random melody with
2. For deleting unwanted melodies from the list
3. A button to generate more random melodies and display them in the window

4. The list or random melodies. "seshCharlie" is a session name, which is a command line argument given at runtime to help organize file names for saved melodies. "seshCharlie_i" is the ith random melody generated in the session.

4.4.2. Training window



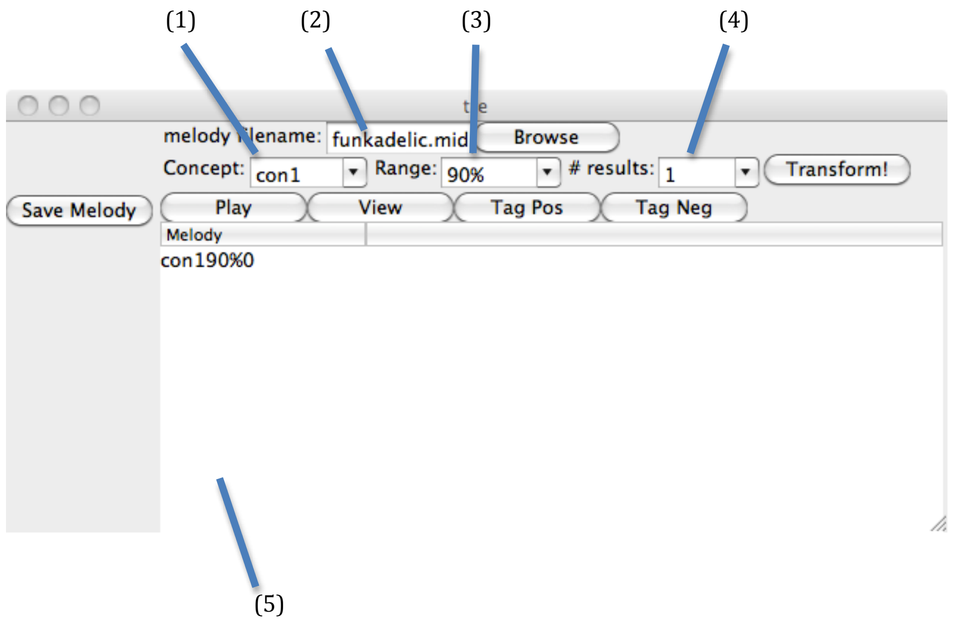
Train	Name	Tag
✓	seshChar...	pos
✓	seshChar...	pos
✓	seshChar...	pos
✓	seshChar...	pos
✓	seshChar...	pos
✓	seshChar...	pos

Train	Name	Tag
✓	seshChar...	neg
✓	seshChar...	neg
✓	seshChar...	neg
✓	seshChar...	neg
✓	seshChar...	neg

1. The dropdown menu to select which classifier/tag is being trained.
2. The button that trains the learner.
3. The list of training examples. The left list is positively tagged melodies, and the right list is negatively tagged melodies. A check means that the melody will be used to train the learner the next time the Train button is pressed. The checklist allows users to easily tweak the training set.

4. A drag and drop target for custom written melodies. Users can write their own melodies as training examples, save them as midi files, and then drag and drop the files onto the target to tag them. Positive examples are dropped on the left and negative on the right.
5. This button allows a user to change the tag of a selected melody, thus moving it to the opposite list.

4.4.3. Transforming window



melody filename: funkadelic.mid
Concept: con1 Range: 90% # results: 1

con190%0

1. The dropdown menu to select classifier/tag which will be used to transform the original melody
2. The selector for the melody to be transformed. The "source melody" must be a MIDI file.
3. The dropdown menu to select how much more or less like the tag the transformation will be.

4. The dropdown menu to select how many transformed melodies are desired
5. The list showing the transformed melodies.

These buttons are present in multiple windows:

- Save – allows the user to save the selected melody as a MIDI file.
- View – allows the user to view the selected melody as sheet music. This feature was deprecated during testing due to performance issues.
- Play – allows the user to play the melody through Ableton
- Tag Pos/Neg – In the Generating and Transforming windows, it allows the user to tag melodies, thus adding them to the list in the Training window.

Ben Schneiderman's design principles for "Creativity Support Tools" from his 2007 paper [15] were enormously influential in the design of this interface. Schneiderman says that designers of creativity support tools should "support exploratory search…enable collaboration…provide rich history keeping…design with low thresholds, high ceilings, and wide walls." This tool was designed primarily as an "exploratory search" tool, allowing the user to intelligently explore the vast space of musical melodies. A relatively rich history-keeping interface was developed in order to support iterative, branching search though the space of melodies. Schneider advocates "accessible but flexible" design and features. This in itself poses a problem – more features often lead to bloated interfaces. To simplify the interface, the system is designed around a single high-level task (tagging melodies), but attempts to be flexible, general, and intuitive with respect to that task.

4.5. Learning

For the learning algorithm, decision trees boosted by AdaBoost [7] were used. Decision trees were chosen because they could give fairly consistent results even with a very small number of training examples, and boosting was chosen because it could be used to capture complex relationships that may occur in larger training sets that would not be expressible by simple decision trees.

The usage of Ableton to write "custom" training examples was done with an eye towards more efficient learning. It was hypothesized that, if given the chance, users could write "stronger" positive an negative examples than they could find with the random generator, and that using these examples would allow for better learning with less examples.

5. Testing/Evaluation Procedure

One specific shortcoming with the original DBD paper was the lack of user testing. This omission seemed salient specifically because DBD was presented as a way to make musical machine learning easy to use for non-CS oriented users. Since the creation of a novel graphical interface for DBD was a goal of this system, user evaluation is especially important. This project attempted to create a testing procedure that would evaluate the system both in terms of ease of use and quality of music generated.

5.1. Testing

The user tests were set up as follows

1. Explaining the "point" of the system

2. A quick run through of the system, demonstrating all of the major features once
3. About 20 minutes to train the classifier "funky"
4. About 10-20 minutes training another classifier or to play "freely" with the system
5. Time to fill out a questionnaire, and a discussion of the test.

Due to the novelty of a "musically intelligent" tool, it was thought that parts (1) and (2) would be needed for users to gain a proper understanding of what the system did, thus allowing them to provide meaningful feedback. Part (3) was chosen as a specific goal that had some room for interpretation. It was hoped that the specificity would give users a defined objective to complete, and that the difference in the system's handling between different users' interpretation of "funky" could be observed. Part (4) was included to give users "play" time to explore the possibilities of the system more freely, and to see whether they would engage in interactions not previously planned for.

5.2. Evaluation

The questionnaire used included questions about demographic information, the simplicity of the interface, and the quality of music produced and overall usability/usefulness of the system. A paper that influenced the development of the evaluation and questionnaire was "Creativity factor evaluation: Towards a standardized survey metric for creativity support" [3]. This paper presented the idea of "flow," of moving from goal to goal and task to task without hesitation. The questionnaire given to the users included questions about their ability to navigate the interface without interruption. The questionnaire also included general questions about the user's satisfaction with the system's musical performance. It is included as an appendix.

When conducting user studies of a similar type for a previous project, it had been noted that users would not volunteer much information until asked, but would talk in much more detail when directly asked. Therefore, the questionnaire was used as a launching point for a discussion that centered on the topics of the questions given to the user. Notes on these discussions were taken.

5.3. User Selection and Demographics

Users were requested through various campus mailing lists. 3/5 had non-trivial programming experience, 1/5 had no programming experience, and 1/5 was a CS major, having both substantial programming experience and some knowledge of machine learning. 2/5 actively wrote and performed music, 1/5 actively performed but did not write, 1/5 had instrument faculty but did not regularly perform, and 1/5 actively listened to music but had no instrument nor singing faculty. All respondents to the call for testers were taken, none turned away.

6. Results

6.1. Pilot Test

A pilot test was run with user 1, a person with minimal experience in programming and musical composition. Two immediate flaws were present. Firstly, the melodies created by the random generator were too “random” sounding and too similar to each other, rendering the user unable to judge whether a given melody was at all funky or not. The pilot user (and, later on, another informal tester) remarked that there did not seem to be any “melodicness” in the note pattern, and that the rhythms were too unpredictable, and that there were surprisingly large intervallic jumps. Another problem was that generating the “large number of variations” (as described in section 3.1.2) based on the tag was much slower than anticipated. Rather than selection from 10k random variations, the number of random variations needed to be lowered to 100 to make the run time feasible.

6.2. Main Testing

In the second round of user tests, both users tested were familiar with programming and had some musical performance experience, but limited experience with writing music.

A trend was noticed that the generated examples were all extremely similar to the original melody being transformed. It was later realized that this was due to the fact that the number of variations created to perform the transformation was reduced. Since the transformation process (described in section 3.1.2) relied on generating new variations off of previously created ones, having a small pool of variations greatly reduced the probability that variations with greater "noise" would occur.

One user attempted to use the keyboard, but it malfunctioned, and the other avoided writing any custom melodies. The users still found it difficult to make specific judgments about the random melodies, but were able to differentiate between melodies they liked and disliked.

The third round of users involved two users, both fairly experienced with programming and fairly experienced in writing music. One of the two was experienced using DAWs as well. These two users spent much more time listening to the random melodies out of curiosity, and seemed to consider them more carefully. The user experienced in DAW's was enthusiastic about using the keyboard to create custom training examples, but the keyboard again malfunctioned. Otherwise, the tests were similar to those in the second round.

All users gave generally similar responses regarding the interface. All said that after a demonstration was given, it was easy to use the interface unsupervised. However, the three-screen layout divided user attention and made it hard to quickly pickup on what to do at the start. Though the overall layout was deemed satisfactory, a number of small improvements and feature additions to the interface were suggested.

All users had trouble tagging melodies created by the random generator as “more funky” or less funky. All users who tested the system with the second random generator were able to discern which melodies they “liked” and which ones they didn’t.

7. Discussion

7.1. Overall Determination of Success

Of the five users, only one was able to produce variations on a melody that he felt better than the original, and even then, only slightly so. In purely performance terms, the system did not succeed. Due to users’ general dissatisfaction with the training melodies created by the random generator, it is possible that the problems were in the generation of training examples rather than the feature representation, and no strong conclusions can be drawn about the effectiveness of the feature selection. It is important to note, however, that musical feature selection and random music generation are still open research areas in computer music. The insights gained from user testing were very valuable in informing directions for future research into this same topic, some of which will be discussed in the Future Work section.

7.2. User Interactions

It was interesting to note that some users, even those with piano or music writing experience, were hesitant to write custom examples using Ableton and the MIDI keyboard. This point is worth noting because custom examples were intended to be an important part of the training set. It was assumed that Ableton, being a popular interface, would be one that was not intimidating to use. If such a popular interface for writing music is still intimidating to "new" users, serious work would need to be put into new versions of the system in order to make custom example creation more accessible.

After updating the random generator, users were much more interested in the random melodies in their own right than expected. The initial intention of this system had been to help people modify and expand on music that they had already written. Two users suggested that they would start from scratch with this system, using the random melodies and transformations to get to a point where they had something to work with and could continue to make music more "organically."

A general trend that was noticed that "activeness" in writing music was strongly correlated with how involved the users were in playing with the system. Knowledge of music theory or general "experience" in playing music was not a significant factor here.

It was hoped that this system could encourage fast iteration and search of melodies but the time necessary to train a concept may be too long to keep casual users interested in the system. All users used the full 20 minutes to train the funky classifier, and the non-active musicians did not seem very engaged in the task towards the end. All users also asked how many melodies they should tag before applying a transformation, and all seemed hesitant when they were told "as many as you want." The prospect of having to do a "pre processing" task of indeterminate time, such as tagging an open ended number of melodies, seemed off-putting. Overall, the process did not seem conducive to the quick iteration. Potential solutions for speeding up user interaction will be discussed in the Future Work section.

7.3. Random Melody Generator

At its core, the implementation of “musical intelligence” is limitation on what kind of valid sequences of notes can be produced (as compared to random sequences). Different rules can be contradictory to another, and therefore it is very hard to accommodate divergent genres of music within a single set of rules. Because of this, the presence of musical intelligence can be limiting on the freedom of the system to generate all kinds of music.

This was one of the design choices faced when making the random melody generator. A generator was desired that could produce melodies all types, so that for any user tag, positive training examples could be generated. The first generator (described in section 4.3.6) assumed almost no rules. The only restrictions were that on average, 80% of the notes in the melody should fit into the best key. Unfortunately, most of the resulting melodies had minimal self-similarity or predictable patterns in their structure, sounding for the most part like random keys being hit on a piano instead of a coherent melody.

The users criticisms of the random melodies (that they were too rhythmically varied, dissonant, and had large jumps) were consistent with analysis based on Tymoczko’s properties. Most melodies in western music have very self-similar rhythmic patterns. Therefore the choice of having and entire melody’s rhythm be variations of a single measure-rhythm seemed justified when designing the second random melody generator (described in section 4.3.7). In musical composition, "out of key" notes are chosen judiciously to achieve a certain musical goal, and therefore having 20% of the notes (a rather high percentage) be out of key seemed unjustified. 90% was chosen as an acceptable percentage through listening and tweaking. The smoothing function (described in section 4.3.4) was motivated by enforcing the "conjunct motion" property (described in section 4.2) more strongly. The cutoff was set at a fifth (7 semitones) because the fifth itself is a very common and melodic sounding interval, while most intervals greater than a fifth are quite large and do not commonly sound as melodic.

A strategy for finding the balance between randomness, generality, and structure is discussed in the Future Work Section.

7.4. Machine Learning and User Interaction

During the initial design phase, it was realized that the interface design would have a substantial effect on the quality of the machine-learned classifier. If the interface was clunky or confusing, it would take users longer to tag or write training melodies. As a result, less custom examples could be written, and because less random melodies are being considered for training examples, there would be less tagged random melodies, or those tagged could be of a lower quality. Because the learning is so small scale, the difference in the training set caused by a bad interface could be significant. Based on the results of the user test, it seems reasonable to conclude that the quality of randomly generated melodies far outweighs the interface as the bottleneck for satisfactory training.

8. Future Work

The future work on this system could be split into three categories – improving the combinatorial generator, collecting more quantitative data (for multiple purposes), and improving the interface.

8.1. Interface Improvements

During development, a “view” feature was highly desired. However, there were no good Python tools for rendering sheet music with low latency. In the future, I will explore the option of linking tools written in other languages to the interface, or writing a custom sheet music renderer.

Two significant and related issues that were revealed by testing, One was that non-active musicians were hesitant in writing original melodies for the system. Another was that Ableton did not easily configure with the MIDI keyboard during the testing. One way to address both of these issues is based on a user suggestion: reorganizing the interface to let users use randomly generated melodies as source melodies to transform, thus bypassing the “music writing” step when creating variations. When combined with an improvement on the runtime of the transformation generator, these two fixes could address all of the mentioned lags not associated with tagging.

By far the slowest task performed by users was the tagging process. It was even more so for inexperienced and non-active musicians, for whom the tagging process was problematic because they did not feel comfortable making judgments on the random melodies. Improving the random melodies is one solution to this problem, and will be discussed later. But while improving the random melody generator may improve users’ comfort level with tagging, it will not make it any faster. Another solution is to (optionally) remove tagging from the user experience, and offline it. Users could be given a handful of starting concepts and simply transform with those. However, if users wanted to customize the concepts, they could add their own training examples, and select the amount of weighting to give the original concept vs the new examples. These pre-made concepts could be "hand-built," or they could be generated from crowd-sourced tags.

8.2. Data collection

There are two main areas where data-collection could help: crowd-sourcing tags to train concepts and feature selection/evaluation.

User tags could either be collected by open-sourcing the system and collecting data form users, or by farming classification to Amazon Mechanical Turk. One way to utilize mechanical Turk is the model used by Siddartha Chaudhary in his 3D modeling creativity support tool [4]. Users would be given 2 melodies, and then be asked to rank one of them as more "x" than the other, with "x" being any adjective tag of their choosing. The tags could then be cleaned, and have a synonym analysis run on them to combine melodies tagged with synonyms. The comparison pairs in each tag set could then be used to train a ranking SVM [8]. On the user interface, tagging could also be replaced with pair wise comparisons to support updating the ranking SVM models with "custom" training. From an interaction perspective, pair-wise comparisons may even be a more comfortable task for users than pos/neg tagging, as users would only have to consider 2 melodies instead of trying to remember an "average" to compare against for pair wise tagging.

Amazon Mechanical Turk tagging could also be used with standard classifier systems. Turk users could be given a series of melodies to tag with whatever adjectives they wanted. This set of adjectives could then be cleaned and synonym compressed. From this set, a few of the most common adjectives could be used as the pre-made concepts. In the second round of Turk tagging, Turk users would be given a random melody and a random adjective “x” and be asked whether the melody was more “x” or less “x”.

For either ranking SVMs or classifier systems, building up a large set of tagged/ranked examples would be extremely useful in feature selection. A number of different features and selecting methods exist, and with a set of baseline data, the effectiveness of different feature sets could be evaluated.

8.3. Random Melody Generator Improvements

As discussed previously, the main problem with designing a random melody generator was balancing the properties of structure and generality. In the current implementation, the generator adheres to certain, fairly fundamental western musical “rules.” However, there are still many “important” musical rules that were ignored in the generator. Research could be done into creating a rule based random melody generator. Given a set of rules, there could be a “degree of adherence” scale for each rule. A random “rule vector” could then be generated whose entries are the “degree of adherence” to each rule. Based on algorithms to enforce each rule to a certain degree of adherence, a random melody could be created to adhere to this rule vector. This way, the relationships of many different rules could be applied, providing for structure, but since the degree of each rule could change, there would still be a high degree of randomness. Ideally, truly “random sounding” melodies, those with a low degree of adherence to many rules, would be uncommon, as would “formulaic” melodies, those that adhered strongly to many rules.

Another line of research that was considered was generating melodies from a feature vector. The general idea was to generate melodies whose points in feature space were some small distance away from the given feature vector. This line of research was abandoned at the start of the semester as being out of the scope of the general project.

There is also an alternative to having a random melody generator. We could start with a large set premade, “satisfactory” melodies – those that have been determined (by some criteria) to be melodies that are “non random” enough for a general user to work with. Once a user starts training on them, the system would start suggesting to the user melodies “unclearly” classified by the system to tag. This way, the system could strengthen the weaker “areas” of the classifier. This process of machine recommended tagging would also work with random melodies if there were a way to generate melodies conforming to a certain feature vector.

9. Conclusion

Though the current implementation of DBD did not produce the desired results, the wide range of solutions possible for the faults of the current system suggest that further research on this project could be fruitful.

Description Based Design, with its accessibility to non-CS oriented musicians and its simple, well-defined use case, has the potential to integrate well with existing music software. If DBD were implemented as a plug-in to an existing interface, users could easily tag and transform melodies from right within their scores, seamlessly harnessing “new” technology without having to give up familiar interfaces.

“CS knowledge agnostic” tools like DBD would be accessible to everyday musicians and could help bridge the gap between “experimental” computer music technology and mainstream musical tools.

10. References

[1] Ableton. https://www.ableton.com/

[2] Ariza, C., Cuthbert, M. 2000. Music21: A toolkit for computer-aided musicology and symbolic music data. *Proc. ISMIR*

[3] Carroll, E., Latulipe, C., Fung R., and Terry, M. 2009. Creativity factor evaluation: Towards a standardized survey metric for creativity support. *Proc. ACM Creativity & Cognition*, 127–136

[4] Chaudhuri, Siddhartha and Koltun, Vladlen. 2010. Data-driven suggestions for creativity support in 3D modeling. *ACM Trans. Graph*. 29(6), Article 183 (December 2010), 10 pages.

[5] Cope, David. Experiments in musical intelligence. http://artsites.ucsc.edu/faculty/cope/experiments.htm

[6] Cycling 74. Max/MSP. http://cycling74.com/products/max/

[7] Freund, Yoav. Schapire, Robert. 1999. A Short Introduction to Boosting. *Journal of Japanese Society for Artificial Intelligence*, 14(5):771-780.

[8] Joachims, Thorsten. 2002. Optimizing Search Engines Using Clickthrough Data. *Proceedings of the ACM Conference on Knowledge Discovery and Data Mining (KDD).*

[9] McCartney, James. 2002. Rethinking the Computer Music Language: SuperCollider. *Computer Music Journal*, 26(4). 61-68.

[10] Pachet. Francois. 2003. The Continuator: Musical interaction with style. *Journal of New Music Research*, 32(3): 333–341

[11] Pachet, Francois. 2009. Description-based design of melodies. *Computer Music Journal* 33(4): 56–68.

[12] Post, Olaf. Toussaint, Godfried. 2011. The Edit Distance as a Measure of Perceived Rhythmic Similarity. *Empirical Musicology Review*, 6(3). 164-179

[13] Puckette, M. 1996. "Pure Data: another integrated computer music environment." Proceedings, Second Intercollege Computer Music Concerts, Tachikawa, Japan, pp. 37-41

[14] pyOSC. https://trac.v2.nl/wiki/pyOSC

[15] Shneiderman, B. 2007. Creativity support tools: Accelerating discovery and innovation. *Communications of the ACM* 50(12): 20–32

[16] Tymoczko, Dmitiri. *A Geometry of Music: Harmony and Counterpoint in the Extended Common Practice*. Oxford: Oxford University Press, 2011. 2-4

[17] Wang, Ge. 2008. The ChucK audio programming language: A strongly-timed and on-the-fly environ/mentality. PhD thesis, Princeton University, Princeton, NJ, USA

[18] Witten, Ian, and Eibe Frank. *Data Mining: Practical Machine Learning Tools and Techniques*. San Francisco: Morgan Kaufmann Publishers, 2005 .

[19] wxWidgets. http://www.wxwidgets.org/.

Appendix I: User Evaluation Questionnaire

Background
On a scale of 1-10, how would you rate your own music theory knowledge (1 being no knowledge, 10 being equivalent to a B.A in music theory)?
Answer:

On a scale of 1-10 (1 being none, 10 being recording an album's worth of finished music), how would you rate your experience with writing original music (1 being none, 10 being equivalent to writing an album's worth of finished music)?
Answer:

On a scale of 1-10, what would you rate your programming experience (1 being none, 10 being equivalent to a BA in computer science)?
Answer:

Describe your songwriting process in general. What parts do you tend to write before others rhythms before melodies before chords, etc)?
Answer:

Overall Perception
Do you think you would experiment with/use this system while writing music? Why or why not?

How and when would you use this system, if at all?

Would this system change your songwriting process at all? Why or why not, and how?

In what way could computers be most helpful to your compositional process?

Did you feel that the music the system produced reflected the tags it was trained for?

Interface
Explain in your own words what this system does:

How you would you train a classifier for a new tag?

How often did you feel like you needed instructions on how the interface worked or what things were?

How often did you feel that you were not sure what to do next or how to proceed with the task at hand?