LEARNING SPECIES COUNTERPOINT

SERGE ALESHIN-GUENDEL, WILL BOWDITCH, CHAMP DAVIS

Contents

1. Abstract

In this paper, we outline an algorithm to create a melody given a cantus firmus (i.e. a bassline), based on the rules of first species counterpoint. The approach taken in this paper is machine learning in flavor, in particular it is a multi-class classification problem in which the implementation uses a support vector machine via Scikit learn. The sample data includes states of three bass notes and two soprano notes, where the corresponding label is precisely the interval between the third bassnote and the third soprano note (i.e. the soprano note that comes after the second soprano note, though it is not part of the state). The training data consists of excerpts of Bach chorales, Bach being one of the composers who best exemplifies good species counterpoint. We also provide a contrasting example of the result when the code learns on a different set of data that does not conform well to the rules of counterpoint. To construct the melody, we first read in a cantus firmus, starting at the first note and then proceeding monotonically by classifying one note at a time until the end of the cantus firmus is reached. The completed score is outputted in the form of an XML file that can then be analyzed by score editors such as museScore, the editor we use in this project.

2. Introduction

The idea for this project stems from the musical studies of one of the authors, who was enrolled in a Harmony class at the time of this project's conception. In the first half of said Harmony class, students are asked to analyze Bach chorales and then attempt to replicate the found patterns and techniques to compose their own melodies, all while conforming to the rules of species counterpoint. As this author also happened to be a computer science student, he immediately recognized that his repetitious homework assignments, that consisted of following a set of strict rules, could be easily and elegantly formulated as a problem in AI.

2.1. **Introduction to Species Counterpoint.**

It 1725, Johann Joseph Fux published Gradus ad Parnassum (Steps to Parnassus), in which he explicitly described the forms and rules for his five species, which provide a structure for composing a melody against a bass. Before we dive into species counterpoint, however, we will take a short digression to discuss the essential music theory needed for this paper.

2.1.1. *Music Theory Primer.*

We won't need much in terms of music theory for this paper, but the reader should be familiar with the note names on the piano and the types of intervals that can occur between them. There are twelve **pitch classes** in a standard scale (enharmonic notes omitted):

$$C,\ C\sharp,\ D,\ D\sharp,\ E,\ F,\ F\sharp,\ G,\ G\sharp,\ A,\ A\sharp,\ B$$

We will represent a note by both its pitch class and its octave. For example, middle C will be denoted by $C4$ (this is called scientific pitch notation). Further, the concept of an **interval** is crucial to species counterpoint, which can most simply be understood as the "staff distance" between notes, illustrated below.

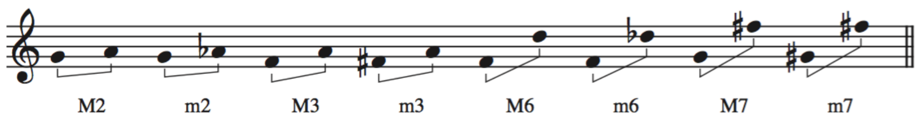

Figure 1. Illustrating different types of intervals [2]

Often, intervals are referred to simply by their number (e.g. a "third"). Further, intervals can be compounded to more than an octave and can move in either direction so that we can realize intervals such as "M10" or "P-5". The reader who is knowledgeable in the subject will recall that intervals can have modifiers to front of them (e.g. M, m, P, d, or A) to further represent their qualities. For example, seconds, thirds, sixths, and sevenths can be major (M) or minor (m). Unisons, fourths, and fifths are perfect (P). Intervals can also be augmented (A) or diminished (d). One can read more about intervals in any introductory text on music theory.

2.1.2. *History of Species Counterpoint.*

2.1.3. *First Species.* The rules of first species can be broken up into two parts: how notes in the melodic line relate to each other, and how notes in the melodic line relate to those in the bass.

Melodic Line

- May use all major, minor and perfect intervals up to and including M6; also P8. May not use augmented or diminished intervals, including those that are enharmonic with M/m intervals, and no 7ths.
- No skips greater than an octave. Futher, motion should be principally stepwise
- No chromaticism other than raised sixths and sevenths in minor keys
- No consecutive skips (two or more) that outline an interval larger than an octave, or that outline a seventh.
- Octave leaps must be preceded and followed by a change in direction.
- Generally recover from a large leap by motion in the opposite direction, stepwise or smaller intervals preferably – i.e. fill in the large leap
- Upper part begins on scale degrees 1, 3, or 5; lower part begins on tonic, implies tonic harmony
- Contour should have one climax, usually in the middle (not the last note), and should not be the leading tone. In general, the leading tone should resolve to tonic unless it is part of a descending line
- Avoid strings of sequences; don't noodle
- Avoid repeated notes – may be used once only
- Cadence should be approached smoothly by stepwise motion; the upper voice ends on scale degrees 1, 3, or 5, and the lower voice on tonic, implying tonic harmony

Counterpoint

- Only consonant intervals are permitted: M/m 3, 6, P5, P8, unison. Dissonant intervals include all augmented and diminished intervals, 2nds, 7ths, P4.
- No parallel or consecutive 8ves (includes unisons) or 5ths. All 8ves or 5ths must be approached by contrary motion or oblique motion. Those approached by similar motion are direct and are not allowed.
- Don't overdo parallel 6ths and 3rds – the goal is independence of voices. Avoid large skips in the same direction at the same time in both voices.

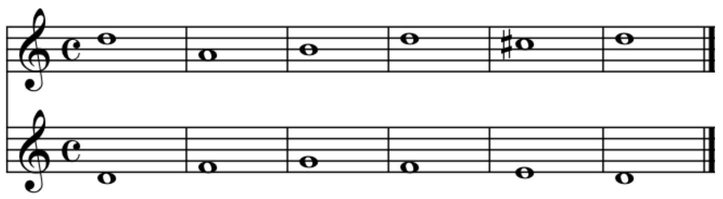

FIGURE 2. Example of first species counterpoint [1]

Figure 1 illustrates the principals of first species counterpoint being used in practice. Note the use contrary motion of the two voices how no one counterpoint interval is repeated too often.

2.2. **Current Landscape and Popular Techniques.**

Champ pls look over this and proofread/take out the links that I put here and cite them idk how

Various approaches to generating music according to the rules of first species counterpoint have been taken previously.

A program called Foox (http://ntoll.org/static/presentations/foox/index.html/10) uses genetic algorithms. A first generation of melodies are randomly generated. Their fitness is then evaluated based on how well they follow the rules of species counterpoint. The more fit melodies then breed and create melodies that are mixtures of parent melodies. These new melodies then have their fitness evaluated and the process continues until the current generation is deemed fit enough.

A paper by Farbood and Schoner (http://alumni.media.mit.edu/ schoner/papers/ICMC01$_{P}ALESTRINA.pd$ details an approach using Markov chains. Initial probability tables based on the rules of first species counterpoint are made, and then used as state transition matricies to generate counterpoint melodies using Markov chains. Human composed counterpoint pieces, along with pieces composed above are then used to train a Markov model which infers new probability tables, used to generated new counterpoint melodies.

Another paper by Herremans and Sorensen (http://www.tandfonline.com/doi/abs/10.1080/17513472.2012.738 details an approach using variable neighbourhood search. A melody is randomly generated and then improved one or two notes at a time using a local search algorithm (variable neighbourhood search). The objective function is based on the rules of first species counterpoint.

In a paper by Adiloglu and Alpaslan (http://www.sciencedirect.com/science/article/pii/S0950705106001468) a machine learning approach was taken. Training data, examples of species counterpoint, was passed into a feed-forward artificial neural network, where each input note is represented by 14 nodes which take into account all the different information about that note. After being adequately trained, the model can then produce a melody according to species counterpoint given a bass line.

The genetic algorithms approach particularly inspired us at the beginning of our research process as it contained open source code to generate melodies based on the rules of not just first species, but second, third, and fourth as well.

We initially tried to structure our problem in terms of a search problem. However it became clear over time that this wouldn't be that interesting, as the success of the program came down to having an appropriate cost function. We also considered constraint satisfaction, as the rules of species counterpoint lend itself fairly well to that sort of problem.

We ended up taking a machine learning approach to the problem similar to that of Adiloglu and Alpaslan, except using support vector machines.

3. Methods

The actual implementation of our algorithm isn't particularly complicated or code-intensive as we made use of existing libraries for both musical programming in python via Music21 and machine learning via Scikit learn. Further, our algorithm can be broken down into three steps, which will be discussed in greater detail in this section: (1) training the classifier, (2) constructing the melody against the given cantus firmus, (3) outputting the result.

3.1. **Representation of Music.** The first challenge we faced involved the representation of notes in python. The ideal representation needed to be easy to work with and manipulate but also capture enough information that could generalize easily. The first representation that we considered was using an integer notation, with each note assigned an integer. For example, middle C ($C4$) might be assigned the integer 0, $C\sharp 4$ would be assigned 1, and $C5$ would be assigned 12, and so on. Then basslines and melodies could be efficiently represented and thought of as vectors in Euclidean space, which we thought would work well with eventual calculations. However, the immediate problem that invalidated this idea was the lack of an obvious generalization. In particular, this works well for C major, but if we wanted to work in a different key, we would have to construct a sort of dictionary that maps a key signature to the certain integers that belonged to it. And this representation only got more complicated as we enumerated the sorts of calculations that we wanted to carry out.

3.1.1. *Music21.* The eventual solution was to use a library constructed by MIT called Music21 which takes an object oriented approach to music representation that is easy to work with, solves the key problem (as Music21 includes a "key" class), and allows the data to be efficiently and effectively outputted for analysis.

In particular, we made the most use of the Note class and the Interval class. The Note class consists of many attributes that are helpful in calculations and manipulation of data:

```
>>> f = note.Note("F5")
>>> f
<music21.note.Note F>
>>> f.name
'F'
>>> f.octave
5
```

We also use the Interval class, which serves as the backbone of the learning process in which the labels are precisely the intervals between certain notes:

```
>>> aNote = note.Note('c4')
>>> bNote = note.Note('g5')
>>> aInterval = interval.notesToInterval(aNote, bNote)
>>> aInterval
<music21.interval.Interval P12>
```

There are other methods and classes that were used in this project, though we will not discuss them in great detail. However, the full module reference can be found here: http://web.mit.edu/music21/doc/moduleReference/index.html

3.2. **The State Space.** The next phase of the project consisted of deciding the structure of the data upon which we would run the learning algorithms. As the contour of the melody is particularly important in species counterpoint, the ideal solution should somehow encapsulate the structure of the score. In particular, when we are deciding what note should come next in the melody, there should be a way to base this decision on the previous notes. Yet there was a balance that had to be met in the interest of generalization and avoiding overfitting. The route we took was to have a state consist of measures, as we decided this would be the best way to learn the rules of species counterpoint (whose strict guidelines only considers at most the previous two notes) while reducing overfitting.

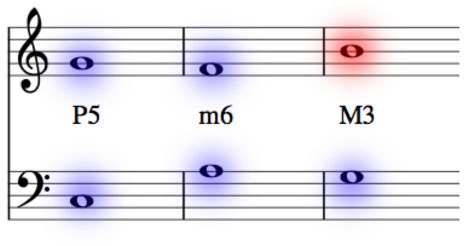

FIGURE 3. Example state

An example state can be seen above. It should be emphasized, however, that the state only consists of the blue notes, and that the red note is the note that will be classified (in fact the interval between it and its corresponding bass note), which we will discuss in a later section. The python implementation of the state can be seen below:

```
class state:
    def __init__(self, b1, b2, b3, s1, s2, pos):
        self.bassNotes = [b1,b2,b3]
        self.sopranoNotes = [s1,s2]
        self.counterpointIntervals = [interval.Interval(b1, s1),
        ↪   interval.Interval(b2, s2)]
        self.bassIntervals = [interval.Interval(b1, b2),
        ↪   interval.Interval(b2,b3)]
        self.sopranoIntervals = [interval.Interval(s1, s2)]
        self.position = pos  #Percent of the song at which the note "s3" will
        ↪   occur

    def getFeatures(self):
        #Implementation

    def getLegalMoves(self):
        #Implementation
```

3.3. **Scikit Learn and Training the Model.** Scikit-learn is an open source machine learning library for the Python programming language. The model was trained using a support vector machine implemented by Scikit-learn. Support vector machines (SVMs) are a set of supervised learning methods used for classification, regression and outliers detection.

3.3.1. *Scikit Learn Preliminaries.*

3.3.2. *Feature Extraction.* One of the major steps in this project is deciding what features to use when training the data. That is, how to best represent the data with the overarching themes of scalability and overfitting ever-present in the back of our minds. Essentially what ended up working best was just to work in intervals and essentially regarding the states as structures not embedded in the staff so that we consider states equivalent up to vertical shifts up or down the staff. This allows for better understanding of the counterpoint that is at work behind the scenes, as how the melody and bass are interacting is much more important than seeing what specific notes are being play, plus including absolute pitches in our features would lock us into a key signature; that is, working in intervals allows us to work in all keys.

As for the exact values for the features, we simply took the number of semitones of each of the five possible intervals that could be considered, (along with one position feature), and normalized each by the maximum expected value for the feature. That is, counterpoint intervals we expect to be larger on average so we normalize those by 32.0. We normalize the melodic intervals by 8.0 as we don't anticipate many intervals above a fifth here. It should be emphasized that the features we used were found through experimentation, and that such features may very well not be the most effective set of values for the problem. Below is the python implementation of the features for each space:

```python
class state:
    def __init__(self, b1, b2, b3, s1, s2, pos):
        #Implementation

    def getFeatures(self):
            return [self.counterpointIntervals[0].semitones/32.0,
             ↪   self.counterpointIntervals[1].semitones/32.0,
             ↪   (self.bassIntervals[0].semitones/8.0),
             ↪   (self.bassIntervals[1].semitones/8.0),
             ↪   (self.sopranoIntervals[0].semitones/8.0), self.position]

    def getLegalMoves(self):
        #Implementation
```

We return the features in the form of a standard feature vector, as this is the format required for Scikit learn to train on.

3.3.3. *Training the Model.* In this section we talk about how we trained the model, in terms of the data set used as well as the implementation thereof.

It should be noted that finding a good data set to learn on was a nontrivial task, and that our data had to be manually manipulated in order to fit the constraints of our problem. In particular, through Music21 we obtained a corpus of several hundred Bach chorales. The issue though was that these chorales were written for four voices—soprano, alto, tenor, bass—while we are only working with a bass and a soprano. Further, there was not a one-to-one correspondence between bass and soprano notes that is necessary for first species, whence each score required one to manually go through the score and remove passing tones, keeping only those notes that occurred on a strong beat. Lastly, even Bach didn't always restrict his compositions to first species, making our data set less than perfect. We will discuss an "accuracy" metric in the results section.

Nonetheless, the code (found below) is relatively straightforward. Iterating over all scores in a directory (our set of chorales), we were able to read in these scores via Music21 and extract the soprano line and the bass line to separate lists (in fact we used the Music21 representation of a list, called a stream, which allowed for easy XML output).

With two lists of notes at hand, we were able to iterate through the streams in a sweeping window fashion, extracting features along the way. That is, for each sequence of three bass notes, we take the first two corresponding soprano notes and create a state with this data (discussed in a previous section). From this state we can then extract the features and add this to the 2D array [n-samples, n-features] required by Scikit learn. The interval between the third bass note and the third soprano note (not in the state) is then added to the target array. Note that we omit the modifier of the interval. That is, our labels don't consist of strings like 'm3' or 'P4', but rather '3' and '4', etc

```
data = []
targets = []

#Reading in the training data.
for file in os.listdir("/Users/Path"):
        if file.endswith(".xml"):
            name= "/Users/Path/"+file
            score = converter.parse(name)

            sopranoMeasures = score.parts[0].getElementsByClass("Measure").flat
            sopranoNotes = sopranoMeasures.getElementsByClass("Note")

            bassMeasures = score.parts[1].getElementsByClass("Measure").flat
            bassNotes = bassMeasures.getElementsByClass("Note")

            stateSpace = []
            for i in range (2, len(sopranoNotes)):
                stateSpace=stateSpace+[state(bassNotes[i-2], bassNotes[i-1],
                ↪   bassNotes[i], sopranoNotes[i-2], sopranoNotes[i-1],
                ↪   (1.0*i)/len(bassNotes))]
                targets = targets+[interval.Interval(bassNotes[i],
                ↪   sopranoNotes[i]).directedName[1:]]

            for s in stateSpace:
                data = data+[s.getFeatures()]

#Fitting the Model
clf = svm.SVC(C=10000)
clf.fit(data,targets)
```

After we have finished analyzing every score in the current directory, our data can then fitted to the corresponding targets, which is as easy as initializing a support vector machine in Scikit learn and calling the fit function.

It should be emphasized that the value of C was found through experimentation, and furthermore, we found that a regular support vector machine with Gaussian kernel actually performed better than a linear support vector machine, which was recommended by Scikit learn to use on relatively small data sets such the one we are working with.

3.4. **Constructing the Melody.** After the model has been fitted, constructing the melody is relatively straightforward. The implementation is seen below. To start, we realize that because we're working with sequences of three notes, we must generate the first two notes, the first one being chosen from the tonic, scale degree three, or scale degree five, and the second of which is chosen randomly from there to conform to the rules of species counterpoint.

After the first two notes are generated, we proceed monotonically in a sweeping window fashion as previously described. We construct a state from the current five notes, and after extracting the features from this state, we use the trained model to predict what the next interval will be above the third bass note. We then input the respective note into the melody stream and shift the window over by one note, proceeding until we reach the end of the inputted cantus firmus.

```
#What key we are working in
key = key.KeySignature(0)

#Reading in the cantus firmus
cf_score = converter.parse('/Users/Champ/Desktop/AI/basslines/cf007-16bar.xml')

#Getting the cantus firmus' notes (the bassline)
cf_measures = cf_score[2].getElementsByClass("Measure").flat
cf_notes = cf_measures.getElementsByClass("Note")

#Creating the melody
melody=[]

#Randomly generating the first note to be on the first, third, or fifth (assume C
↪   major for current implementation)
possibleFirstNotes=[note.Note('G4', quarterLength=4),note.Note('C5',
↪   quarterLength=4),note.Note('E5', quarterLength=4)]
melody.append(possibleFirstNotes[random.randint(0,2)])

#Randomly generating the second note
legal_moves = ['M3', 'm3', 'M2', 'm2', 'P1', 'm-2', 'M-2', 'm-3', 'M-3']
second_note = melody[0].transpose(interval.Interval(
↪   legal_moves[random.randint(0,len(legal_moves)-1)]))
if key.accidentalByStep(second_note.step) != second_note.pitch.accidental:
        second_note.pitch.accidental = key.accidentalByStep(second_note.step)


#Making sure the second note follows the counterpoint rules i.e. no 2nds, 4ths, or
↪   7ths against the bass

while(interval.Interval(cf_notes[1], second_note).simpleName[-1]=='2' or
↪   interval.Interval(cf_notes[1], second_note).simpleName[-1]=='4' or
↪   interval.Interval(cf_notes[1], second_note).simpleName[-1]=='7'):
        second_note = melody[0].transpose(interval.Interval(
        ↪   legal_moves[random.randint(0,len(legal_moves)-1)]))
        if key.accidentalByStep(second_note.step) !=
        ↪   second_note.pitch.accidental:
                second_note.pitch.accidental =
                ↪   key.accidentalByStep(second_note.step)

melody.append(second_note)


#Adding the rest of the notes, one by one
for j in range (2, len(cf_notes)):
        current_state = state(cf_notes[j-2], cf_notes[j-1], cf_notes[j],
        ↪   melody[j-2], melody[j-1], (1.0*j)/len(cf_notes))
        next_interval = clf.predict([current_state.getFeatures()])[0]

        if next_interval in ['1','4','5','8','11','12','15','18','19','22','25']:
                next_interval='P'+next_interval
        else:
                next_interval='M'+next_interval

        next_note = cf_notes[j].transpose(interval.Interval(next_interval))

        #make sure the next note fits in the key
        if key.accidentalByStep(next_note.step) != next_note.pitch.accidental:
                next_note.pitch.accidental = key.accidentalByStep(next_note.step)

        #append it to the melody
        melody.append(next_note)
```

A *key* part of this code is in making sure the next note fits into the key signature. This is a subtle point that requires some knowledge of music theory. As our labels consist of simply the staff distance, we then add back a quantifier so we can make a Music21 object out of the interval to then transpose the current third bass note to get the correct next melody note. But because a “P4” or a ”M3” above a certain bass note might not be in the key that we are working in, we simply correct the note to get the correct accidental. All of this is to say that our program **does not worry about the quantified interval (m, M, P, A, d)**, all we care about is the staff distance, and then we find the correct note based on the key.

3.5. MuseScore and XML Output.

```
#Preparation for output
melody_output = stream.Part(id='part 1')
melody_stream = stream.Stream() #used in finding counterpoint errors

bass_output = stream.Part(id='part 0')
bass_output.append(clef.BassClef())
bass_stream = stream.Stream() #used in finding counterpoint errors

current_measure = 0
for b in cf_notes:
        measure=stream.Measure(number=current_measure)
        measure.append(b)
        bass_output.append(measure)
        bass_stream.append(b)
        current_measure+=1



current_measure = 0
index = 0
for s in melody:
        s.addLyric(interval.Interval(cf_notes[index],s).simpleName)
        measure=stream.Measure(number=current_measure)
        measure.append(s)
        melody_output.append(measure)
        melody_stream.append(s)
        current_measure+=1
        index+=1


#Showing the result
output = stream.Score()
output.insert(0,melody_output)
output.insert(1,bass_output)
output.show()


#Accuracy check
cp = alpha.counterpoint.species.ModalCounterpoint(stream1 = bass_stream, stream2
 ↪   = melody_stream)
print "Number of Bad Harmonies: ", cp.countBadHarmonies(cp.stream1, cp.stream2)
print "Accuracy: ", 100-(cp.countBadHarmonies(cp.stream1,
 ↪   cp.stream2)/(1.0*len(melody))*100.0), "%"
```

4. Results and Discussion

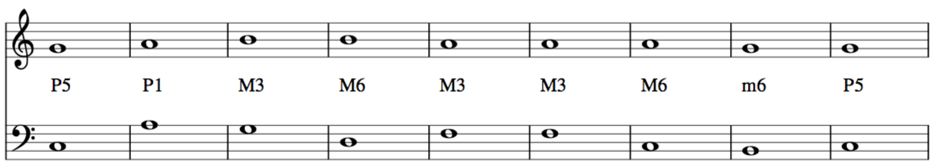

Figure 4. Example 8-bar output trained on Bach chorles

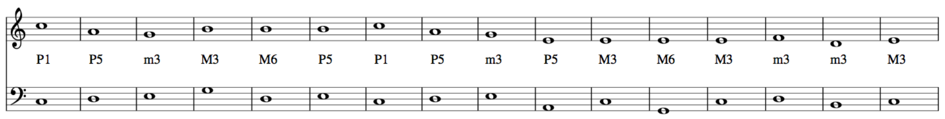

Figure 5. Example 16-bar output trained on Bach chorales

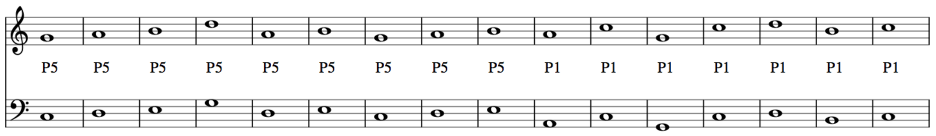

Figure 6. Example 16-bar output trained on fifths and octaves

5. Conclusions

-Definitely possible to make melody given bass line using computers -To move to other species we need appropriate training data -To move to other genres wouldn't actually be that bad cause we wouldn't need to worry about the rules that much I think, more about style

Acknowledgments. It is a pleasure to thank Sergio Alvarez for introducing to us the machine learning techniques required to complete this project and for providing technical guidance along the way.

References

[1] https://commons.wikimedia.org/wiki/File:Species1.png
[2] Kostka, Payne, Almén Tonal Harmony. McGraw-Hill. 2013.
[3] Tobias Oekiter, Hubert Partl, Irene Hyna and Elisabeth Schlegl. The Not So Short Introduction to $\LaTeX\,2_\varepsilon$. http://tobi.oetiker.ch/lshort/lshort.pdf.