FEATURE EXTRACTION AND MACHINE LEARNING ON SYMBOLIC MUSIC USING THE `music21` TOOLKIT

Michael Scott Cuthbert
Music and Theater Arts
M.I.T.
cuthbert@mit.edu

Christopher Ariza
Music and Theater Arts
M.I.T.
ariza@mit.edu

Lisa Friedland
Department of Computer Science
University of Massachusetts Amherst
lfriedl@cs.umass.edu

ABSTRACT

Machine learning and artificial intelligence have great potential to help researchers understand and classify musical scores and other symbolic musical data, but the difficulty of preparing and extracting characteristics (features) from symbolic scores has hindered musicologists (and others who examine scores closely) from using these techniques. This paper describes the "feature" capabilities of `music21`, a general-purpose, open source toolkit for analyzing, searching, and transforming symbolic music data. The features module of `music21` integrates standard feature-extraction tools provided by other toolkits, includes new tools, and also allows researchers to write new and powerful extraction methods quickly. These developments take advantage of the system's built-in capacities to parse diverse data formats and to manipulate complex scores (e.g., by reducing them to a series of chords, determining key or metrical strength automatically, or integrating audio data). This paper's demonstrations combine `music21` with the data mining toolkits Orange and Weka to distinguish works by Monteverdi from works by Bach and German folk music from Chinese folk music.

1. INTRODUCTION

As machine learning and data mining tools become ubiquitous and simple to implement, their potential to classify data automatically, and to point out anomalies in that data, is extending to new disciplines. Most machine learning algorithms run on data that can be represented as numbers. While many types of datasets naturally lend themselves to numerical representations, much of the richness of music (especially music expressed in symbolic forms such as scores) resists easily being converted to the numerical forms that enable classification and clustering tasks.

The amount of preprocessing needed to extract the most musically relevant data from notation encoded in Finale or Sibelius files, or even MIDI files, is often underestimated: musicologists are rarely content to work only with pitch classes and relative note lengths—to name two easily extracted and manipulated types of information. They also want to know where a pitch fits within the currently implied key, whether a note is metrically strong or weak, what text is being sung at the same time, whether chords are in open or closed position, and so on. Such processing and analysis steps need to run rapidly to handle the large repertories now available. A robust system for data mining needs to integrate reliable and well-developed classification tools with a wide variety of methods for extracting data from large collections of scores in a variety of encodings.

The features module newly added to the Python-based, open source toolkit `music21`, provides this needed bridge between the demands of music scholars and of computer researchers. `Music21` [3] already has a well-developed and expandable framework for importing scores and other data from the most common symbolic music formats, such as MusicXML [4] (which Finale, Sibelius, MuseScore, and other notation software can produce), Kern/Humdrum [6], CCARH's MuseData [11], Noteworthy Composer, the common folk-music format ABC [10], and MIDI. Scores can easily be transformed from symbolic to sounding representations (by uniting tied notes or moving transposing instruments to C, for instance); simultaneities can be reduced to chords that represent the pitches sounding at any moment; and the key or metrical accents of a passage can be analyzed (even for passages that change key without a change in key signature).

The features module expands `music21`'s data mining abilities by adding a battery of commonly used numeric features, such as numerical representations of elements present or absent in a piece (0s or 1s, used, for example, to indicate the presence of a change in a time signature), or continuous values representing prevalence (for example, the percentage of all chords in a piece that are triadic). Collections of these features can be used to train machine learning software to classify works by composer, genre, or dance type. Or, making use of notational elements found in certain input formats, they could classify works by graphical characteristics of particular interest to musicologists study-

ing the reception of the work. Such graphical elements might identify the scribe, editor, or publisher of a piece.

In the following sections, we will describe the feature-extraction methods (FEMS) of `music21`. Because `music21` has many powerful, high-level tools for analysis and transformation, FEMS can be tailored to the characteristics of particular repertories and can be combined to create more powerful FEMS than those available in existing software packages. This paper describes how new FEMS can be added to `music21` and demonstrates their usefulness in classifying both classical and popular works.

2. FEATURE EXTRACTION IN MUSIC21

2.1 Feature Extractors from jSymbolic

One of the most useful aspects of the Features module is the integration of 57 features of the 111 implemented in Cory McKay's jSymbolic toolkit [9], a subset of his larger jMIR toolkit that classifies music encoded in MIDI [8]. (`Music21` aims for full jSymbolic compatibility in the near future.) Because `music21` is "encoding agnostic," files in any supported format now have access to these FEMS, so that MusicXML and ABC files (among others) can, without conversion, be run through the same extractors that jSymbolic provided for MIDI files. In addition, Music21 FEMS are optimized so that closely related feature extractors that require the same preprocessing routines automatically use cached versions of the processed data, rather than recreating it.

Example 1 shows how a single feature extractor, borrowed from jSymbolic, can be applied to data from several different sources and datatypes. While using a single feature extractor on one or two works is not a useful way to classify these works, it is a convenient and informative way to understand the system and test the FEMS. All FEMS have documentation and code examples on the music21 website at http://mit.edu/music21. The website also gives instructions for obtaining and installing the software, as well as tutorials and references on using the toolkit.

Example 1 shows how the fraction of ascending notes in a movement of Handel's Messiah (encoded as MuseData) can be found.

```
from music21 import *
handel = corpus.parse('hwv56/movement3-05.md')
fe = features.jSymbolic.\
        DirectionOfMotionFeature(handel)
feature = fe.extract()
print feature.vector
[0.5263]
```

Example 1. Feature extraction on a MuseData score.

Example 2 shows feature extraction run first on a local file, and then on a file from the Internet. The feature extractor determines whether the initial time signature is a triple meter and returns 1 or 0. The result is returned in a Python list, since some FEMS return an array of results, such as a 12-element histogram showing the count of each pitch class. Like Example 1, this example uses file formats (ABC and MusicXML) that cannot be directly processed by jSymbolic. (In all further examples, the initial line, "from music21 import *" is omitted.)

```
# a 4/4 basse danse in ABC format
bd = converter.parse("/tmp/basseDanse20.abc")
fe = features.jSymbolic.TripleMeterFeature(bd)
print fe.extract().vector
[0]
# softly-softly by Mark Paul, in 3/4
soft = converter.parse(
 "http://static.wikifonia.org/10699/musicxml.xml")
fe.setData(soft)
print fe.extract().vector
[1]
```

Example 2. A local file and a web file in two different formats run through a triple-meter feature extractor.

2.2 Feature Extractors Native to `music21`

In addition to recreating the feature extraction methods of jSymbolic, `music21`'s features.native sub-module includes 17 new FEMS. These FEMS take advantage of the analytical capabilities built into `music21`, its ability to work with notational aspects (such as a note's spelling or representation as tied notes), or the richer, object-oriented programming environment of Python. For example, native `music21` FEMS can distinguish between correctly or incorrectly spelled triads within a polyphonic context. (The IncorrectlySpelledTriadPrevalence FEM, called on Mozart's pieces, returns approximately 0.5% of all triads, mostly reflecting chromatic lower neighbors). Notational features that do not affect playback, such as a scribe's predilection for beaming eighth notes in pairs (as opposed to in groups of four) in 4/4, can similarly form the basis for feature extraction. Feature extractors can also use a work's metadata, along with the larger capabilities of the Python language, to add powerful classification methods. An example of this is the ComposerPopularity feature, which returns a base-10 logarithm of the number of Google hits for a composer's name (see Example 3).

```
s = corpus.parse('mozart/k155', 2)
print s.metadata.composer
W. A. Mozart
fe = features.native.ComposerPopularity(s)
print fe.extract().vector
[7.0334237554869485]
```

Example 3. The ComposerPopularity feature extractor reports that there are about 10 million Google results, or approximately 10^7, for the form of Mozart's name encoded in the version of K155 movement 2 that appears in the `music21` corpus, a collection of approximately ten thousand works provided with the toolkit.

Several of the native FEMS are adaptations of jSymbolic extractors, expanded by capabilities offered by other modules in `music21`. For instance, McKay's "Quality" feature classifies a piece as either in major or in minor based on information encoded within the initial key signature of some MIDI files. For files without this information, `music21`'s enhancement of this FEM (features.native.QualityFeature) will also run a Krumhansl-Schmuckler probe-tone key analysis (with the default Aarden-Essen weightings) [7] on the work to determine the most likely mode. The native module also includes many chord-related FEMS that were proposed by McKay but not included in the present release of jSymbolic.

2.3 Writing Custom Feature Extractors

One of the strengths of `music21`'s feature system is the ease of writing new FEMS. After inheriting the common superclass FeatureExtractor, new FEMS can be created and used alongside existing FEMS. The core functionality is implemented in a private method called *_process()*, which sets the values of the vector of an internally stored Feature object. The FeatureExtractor superclass provides automatic access to a variety of presentations of the score, from a flat representation (using the *.flat* property) to a reduction as chords, along with histograms of commonly requested musical features such as pitch class or note duration. These representations are cached for quicker access later as keys on a property called *data* (such as self.data['chordify']). The object also allows direct access to the source score through the *stream* property.

Example 4 creates a new feature extractor that reports the percentage of notes that contain accidentals (including double sharps and flats, but excluding naturals) that are not B-flats. This feature could help chart the increased usage over the course of the Renaissance of *musica ficta*, that is, chromatic notes beyond B-flat (the only accidental common to Medieval and Renaissance music).

```
# Feature Extractor definition
class MusicaFictaFeature(
              features.FeatureExtractor):
 name = 'Musica Ficta'
 discrete = False
 dimensions = 1
 id = 'mf'

 def _process(self):
  allPitches = self.stream.flat.pitches
  # N.B.: self.data['flat.pitches'] works
  # equally well and caches the result for
  # faster access by other FEMS.
  fictaPitches = 0
  for p in allPitches:
   if p.name == "B-":
    continue
   elif p.accidental is not None \
     and p.accidental.name != 'natural':
    fictaPitches += 1
  self._feature.vector[0] = \
     fictaPitches / float(len(allPitches))

# example of usage of the new method on two pieces
# (1) D. Luca early 15th c. Gloria
luca = corpus.parse('luca/gloria.mxl')
fe = MusicaFictaFeature(luca)
print fe.extract().vector
[0.016169154228855721]
# (2) Monteverdi, late 16th c. madrigal
mv = corpus.parse('monteverdi/madrigal.3.1.xml')
fe.setData(mv)
print fe.extract().vector
[0.05728727885425442]
```

Example 4. A custom feature extractor to find *musica ficta*, applied to an early 15th-century Gloria and a late 16th-century madrigal.

3. MULTIPLE FEATURE EXTRACTORS AND MULTIPLE SCORES

Since the previous examples have extracted single features from one or two scores, similar results could have just as well been obtained through the object model or analytical routines of the `music21` toolkit. But machine learning techniques require a large group of scores and many features. The features module shines for such studies by making it easy and, through caching, fast to run many scores (or score excerpts) through many FEMS, and to graph the results or output them in the formats commonly used by machine learning programs.

3.1 Extracting Information from DataSets

The DataSet object of the features module is used for classifying a group of scores by a particular class value using a set of FEMS. Its method *addFeatureExtractors()* takes a list of FEMS that will be run on the data. (For ease of getting a large set of FEMS, each feature extractor has a short id which allows it to be found by the method *extractorsById()*. The special id "all" gets all feature extractors from both native and jSymbolic libraries.) The *addData()* method adds a `music21` Stream [1] (i.e., a score, a part, a fragment of a score, or any other symbolic musical data) to the DataSet, optionally specifying a class value (such as the composer, when the task at hand is classifying composers) and an id (such as a catalogue number or file name). For convenience, *addData()* can also take a string containing a file path to the data (in any of several formats), a URL to the score on the internet, or a reference to the work in the `mu-`

sic21 corpus. Example 5 sets up a DataSet to run three FEMS related to note length on four pieces: two by Bach, one by Handel, and an "unknown" work (also by Handel). If a file has been read in once and is unmodified since the last reading, its parsed version is cached in a Python "pickle" file for quicker reading in subsequent runs.

```
ds = features.DataSet(classLabel='Composer')
fes = features.extractorsById(['ql1','ql2','ql3'])
ds.addFeatureExtractors(fes)

b1 = corpus.parse('bwv1080', 7).measures(0,50)
ds.addData(b1, classValue='Bach', id='artOfFugue')
ds.addData('bwv66.6.xml', classValue='Bach')
ds.addData('c:/handel/hwv56/movement3-05.md',
             classValue='Handel')
ds.addData('http://www.midiworld.com/midis/other/h
andel/gfh-jm01.mid')
ds.process()
```

Example 5. Setting up and processing a DataSet with three FEMS and four scores.

Extracting the data from a DataSet is simple once *process()* has been called. The simplest way of getting the output of multiple feature extractors is through DataSet's *write()* method, which can take a filename or a file format (if no file path is given, a file is saved to the user's "temp" directory). File formats are specified as strings that call the appropriate OutputFormat object. Music21 comes with OutputFormats for comma-separated values (csv), tab-delimited output (tab) for Orange, and Attribute-Relation File Format (arff) for Weka. The OutputFormat object is subclassable, so additional formats for R, Matlab, native Excel (an .xls reader/writer is packaged with music21), or json (for Java, Max/MSP, or other systems) can easily be developed.

Other ways of obtaining extracted features include DataSet's *getFeaturesAsList()* method, which returns a list of lists, one list of feature results for each piece, and *getString(),* which returns the data as a single string in any of the supported formats. If the optional Python package Matplotlib is installed, the data can also be graphed from within music21. Finally, because the DataSet is fully integrated with the rest of the toolkit, specific Streams can be examined in notation. Example 6 takes the DataSet object from Example 5 and examines it in several ways. Part (a) writes it out as an comma-separated file; (b) prints the attribute labels; (c) gets the entire feature output as a list of lists and prints one line of it; (d) displays the entire feature data in OrangeTab output. Part (e) examines the feature vectors and displays as pngs (via Lilypond) any scores where the most common note value is an eighth note (length = 0.5); the resulting output contains the two Handel scores. Part (f) plots the last two features (most common note length and the prevalence of that length) for each piece.

(a)

```
ds.write('/usr/cuthbert/baroqueQLs.csv')
```

(b)

```
print ds.getAttributeLabels()
```

['Identifier', 'Unique_Note_Quarter_Lengths',
'Most_Common_Note_Quarter_Length',
'Most_Common_Note_Quarter_Length_Prevalence', 'Composer']

(c)

```
fList = ds.getFeaturesAsList()
print fList[0]
```

['artOfFugue', 15, 0.25, 0.6287328490718321, 'Bach']

(d)

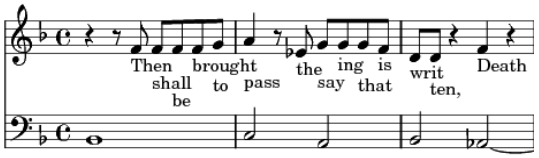
```
print features.OutputTabOrange(ds).getString()
```

Identifier	*Unique_Note…*	*Most_Common…*	*Most_Com..Prevalence*	*Composer*
string	*discrete*	*continuous*	*continuous*	*discrete*
meta				*class*
artOfFugue	*15*	*0.25*	*0.628732849072*	*Bach*
bwv66.6.xml	*3*	*1.0*	*0.601226993865*	*Bach*
hwv56/movem…	*7*	*0.5*	*0.533333333333*	*Handel*
http://www.mid...	*14*	*0.5*	*0.768951612903*	

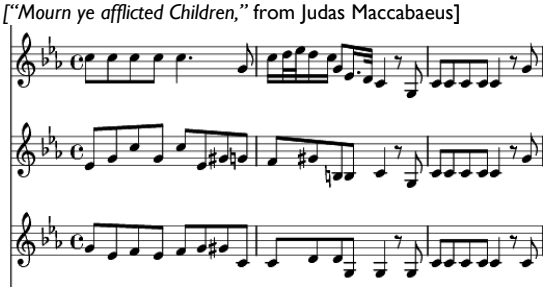
(e)

```
for i in range(len(fList)):
  if fList[i][2] == 0.5:
    ds.streams[i].show('lily.png')
```

[HWV 56 3-5, from the Messiah*]*

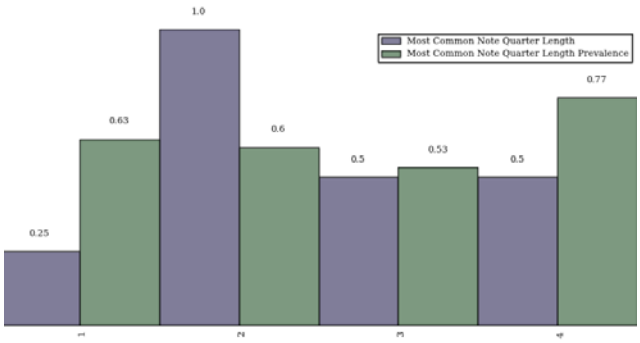


["Mourn ye afflicted Children," from Judas Maccabaeus*]*



(f)

```
p = graph.PlotFeatures(ds.streams,
          fes[1:], roundDigits = 2)
p.process()
```



Example 6. Viewing the contents of a DataSet object.

3.2 Using Feature Data for Classification

Once the DataSet object has been plotted or viewed as musical data to check the results for obvious errors, then the outputted data can be fed into any number of standard data mining packages for analyses such as clustering or classification. The package Orange (http://orange.biolab.si) integrates well with `music21` since it provides a Python interface to its classification algorithms (in addition to having a GUI); other toolkits such as Weka [5] can also easily be used. Below, we include sample code for using Orange, but the results of Examples 8 and 9 were produced in Weka. Complete code examples, along with our sample data, can be found in the demos directory in the `music21` distribution.

4. DEMONSTRATIONS AND RESULTS

We end this paper with two demonstrations of the power of feature extraction in `music21` to enable automatic classification of musical styles and composers from symbolic data encoded in many formats. The first example uses 24 pitch- and rhythm-based feature extractors (p1–16, 19–21, and r31–35) to classify monophonic folksongs from four files in the Essen folksong database as being from either China or Central Europe (mostly Germany). Two files, folkTrain.tab and folkTest.tab, are created according to the same model as Example 5. (Full source for this part of the example is available in the `music21` distribution as demos/ismir2011/prepareChinaEurope().) The files contain 969 and 974 songs, respectively, and the extractors described above result in 174 features, although about half are discarded during preprocessing because they have the same value for every song.

Example 7 applies two classification methods (or learners) to the pair of data files, using the songs in the first file for training the classifier and those in the second for testing (i.e., validating) the classifier's predictions. The first method, MajorityLearner, simply chooses the classification that is most common in the training data (e.g., for the data in Examples 5-6, it would label the unknown data as Bach, because Bach is represented twice as often as Handel in the labeled data), and thus reports a baseline accuracy for other classification methods to be measured against. The second method, k-nearest neighbors (kNN) [12], assigns to each test example the majority label among the *k* most similar training examples. After assigning an origin to each song in folkTest, the program consults the correct answer or "ground truth," and in the end it prints the fraction of songs correctly labeled by each classifier: 69% for the baseline (MajorityLearner) and over 94% for kNN. The performance of kNN over MajorityLearner stands only to increase with the development, in the near future, of FEMS more suited to the nuances of folk music.

```
import orange, orngTree
trainData = orange.ExampleTable('/folkTrain.tab')
testData  = orange.ExampleTable('/folkTest.tab')

majClassifier = orange.MajorityLearner(trainData)
knnClassifier = orange.kNNLearner(trainData)

majWrong = 0
knnWrong = 0

for testRow in testData:
  majGuess = majClassifier(testRow)
  knnGuess = knnClassifier(testRow)
  realAnswer = testRow.getclass()
  if majGuess == realAnswer:
    majCorrect += 1
  if knnGuess = realAnswer:
    knnCorrect += 1

total = float(len(testData))
print majCorrect/total, knnCorrect/total
0.68788501026694049       0.94353182751540043
```

Example 7. Using data output from the features module of `Music21` to classify folksongs in Orange.

In Example 7, the training and testing data are split approximately 50-50. We can increase both the amount of data used to train the models and the number of predictions they make by using a technique called 10-fold cross-validation. Example 8 shows the results of doing this, on the same data, using a variety of classifiers in Weka.

Classifier	Accuracy
Majority (baseline)	63%
Naïve Bayes	79%
Naïve Bayes (using supervised discretization option)	91%
Decision tree	93%
Logistic regression	95%
K-nearest neighbor (using k = 3)	96%

Example 8. Accuracy of classifiers for distinguishing Chinese from Central European folk music.

While kNN was the best classifier in all our experiments, decision tree-based classification systems [2] can be helpful for users wishing to understand how a classifier decides which features are important. Example 9 shows a decision tree built to distinguish the vocal works of Bach and Monteverdi. Given a data set of 46 works from each composer, and the same features used previously, the classifier has selected just 6 features as informative when building this tree. (In a 10-fold cross-validation experiment, trees like this achieved about 86% classification accuracy.)

Although it is not always possible to explain the algorithm's choices intuitively, some of them make sense upon examination. For example, although Monteverdi uses sharped notes, he does not ever use sharps in his key signatures, and thus sharped notes remain uncommon in his pieces. The decision tree picks up on this predilection in its

top-level split, the single most informative rule learned (final line of Example 9): if more than 14.4% of the piece's notes are MIDI note 54 (F#3), then the piece is by Bach (true all 30 out of 30 times in the data set).

```
Basic_Pitch_Histogram_54 <= 0.144578
| Initial_Time_Signature_0 <= 3: Bach (4.0)
| Initial_Time_Signature_0 > 3
| | Range <= 32: Bach (6.0)
| | Range > 32
| | | Basic_Pitch_Histogram_64 <= 0.05: Bach (3.0)
| | | Basic_Pitch_Histogram_64 > 0.05
| | | | Basic_Pitch_Histogram_60 <= 0.921569: Monteverdi (47.0/1.0)
| | | | Basic_Pitch_Histogram_60 > 0.921569
| | | | | Relative_Strength_of_Top_Pitches <= 0.96875: Bach (4.0)
| | | | | Relative_Strength_of_Top_Pitches > 0.96875: Monteverdi (2.0)
Basic_Pitch_Histogram_54 > 0.144578: Bach (30.0)
```

Example 9. Decision tree algorithm applied to distinguish Bach and Monteverdi's choral pieces.

The results of these classification tests of folk and baroque music demonstrate `music21`'s utility in automatically determining musical style from a score without human intervention. Sophisticated style analysis tools open up opportunities in other areas, such as more accurate notation and playback. For instance, a program could choose appropriate instruments for digital performance depending on the estimated location in which the piece was composed: fiddles for Irish jigs, kotos and *shō* for Japanese folk music. By lowering the barriers to using feature extraction, `music21` can bring the fruits of MIR to a wide audience of computer music professionals.

5. FUTURE WORK

Though these tools are extremely powerful already, the development of new FEMS in `music21` and application of these features to the classification of musical scores is still in its infancy. The authors and the `music21` community will continue to add new feature extractors to solve problems that range from assigning composer names to anonymous works of the Middle Ages and Renaissance, to genre classification of popular music leadsheets. to charting the slow change in use of chromatic harmony in the nineteenth century. More sophisticated data mining tools such as support vector machines and clustering algorithms can be explored to improve the accuracy of the classification methods. The newest releases of `music21` can take audio data as input; thus we hope to combine MIR of symbolic music data with feature extraction methods applied to audio files, inching closer to the goal of creating software for sophisticated musical listening.

6. ACKNOWLEDGEMENTS

Development of the feature extraction aspects of the `music21` toolkit is supported by funds from the Seaver Institute. Thanks to Seymour Shlien and Ewa Dahlig-Turek for permission to distribute ABC versions of the Essen folksong database with `music21`.

7. REFERENCES

[1] C. Ariza and M. Cuthbert: "The `music21` Stream: A New Object Model for Representing, Filtering, and Transforming Symbolic Musical Structures," *Proceedings of the International Computer Music Conference*, 2011.

[2] L. Breiman et al.: *Classification and Regression Trees*. Chapman & Hall, Boca Raton, 1984.

[3] M. Cuthbert and C. Ariza: "`music21`: A Toolkit for Computer-Aided Musicology and Symbolic Music Data," *Proceedings of the International Symposium on Music Information Retrieval*, pp. 637–42, 2010.

[4] M. Good: "An Internet-Friendly Format for Sheet Music." *Proceedings of XML 2001*.

[5] M. Hall, E. Frank, G. Holmes, B. Pfahringer, P. Reutemann, I. H. Witten: "The WEKA Data Mining Software: An Update." *SIGKDD Explorations*, *11*(1), 2009.

[6] D. Huron: "Humdrum and Kern: Selective Feature Encoding." In *Beyond MIDI: the Handbook of Musical Codes*. E. Selfridge-Field, ed. MIT Press, Cambridge, Mass., pp. 375–401, 1997.

[7] C. Krumhansl: *Cognitive Foundations of Musical Pitch.* Oxford University Press, Oxford, 1990.

[8] C. McKay: "Automatic Music Classification with jMIR," Ph.D. Dissertation, McGill University, 2010.

[9] C. McKay and I. Fujinaga: "jSymbolic: A feature extractor for MIDI files." *Proceedings* of *the International Computer Music Conference*, pp. 302–5, 2006.

[10] I. Oppenheim, C. Walshaw, and J. Atchley. "The abc standard 2.0." http://abcnotation.com/wiki/abc:standard:v2.0. 2010.

[11] C. S. Sapp: "Museinfo: Musical Information Programming in C++." http://museinfo.sapp.org, 2008.

[12] G. Shakhnarovich, T. Darrell, and P. Indyk: *Nearest-Neighbor Methods in Learning and Vision,* MIT Press, Cambridge, Mass. 2006.