EXTRACTING MUSIC FEATURES WITH MIDXLOG

Roberto Basili
AI Research Group,
Dept. of Computer Science,
Systems and Production
University of Roma, Tor Vergata
Via del Politecnico 1 00133
Rome (Italy)
basili@info.uniroma2.it

Alfredo Serafini
AI Research Group,
Dept. of Computer Science,
Systems and Production
University of Roma, Tor Vergata
Via del Politecnico 1 00133
Rome (Italy)
serafini@info.uniroma2.it

Armando Stellato
AI Research Group,
Dept. of Computer Science,
Systems and Production
University of Roma, Tor Vergata
Via del Politecnico 1 00133
Rome (Italy)
stellato@info.uniroma2.it

ABSTRACT

Every machine learning process must rely, first of all, on a good collection of appropriate features from examined data sets. Even before feature selection stages and cascades of learning steps for creating complex features may lead to the creation of proper decisional spaces, the process of extracting basic features may reveal to be a long and cumbersome task, which depends on the nature of the examined information and on the way it is represented. In this sense, obtaining good and easy-to-use tools for examining collected data may help researchers on focusing on what they need, instead of dealing on how to get it. We present here our approach in easing the extraction of features from symbolic music notation, which led to the development of MidXLog, a query language for MIDI files built on top of the logic programming language Prolog. We briefly introduce some of the characteristics of this language, and then describe its use in processing MIDI files to obtain an interesting range of features for the MIREX challenge

Keywords: Query-Language for MIDI, Feature Space Representation, Feature Extraction.

1 DEVELOPING MIDXLOG

After our first exploration on music genre recognition [1], we felt the need for a tool, not only useful to ease the feature extraction process, but also able to support statistical analysis of MIDI data on the fly. We did not just required indeed a tool, but a complete query language over MIDI files.

Our approach has thus been oriented towards extending an existing language, which matched the following requirements:

- be an *interpreted language*: on the fly querying of MIDI files is a must for researcher performing extensive analysis of data. A compile–and–run approach is mostly suited for systems which already know what to do, while the possibility of interacting via console grants more possibilities during domain analysis, testing and debugging.
- be a *declarative* language: language expressions should be oriented as much as possible towards describing the results rather than explaining how to get them
- be a *Turing-complete* language: the expressive power of a Turing-complete language could help not only properly in querying midi files, but also to manipulate their content.

Our choice fell on the logic programming language Prolog [3], as for its easy-handling capabilities for lists and structures, proved to be a good basis upon which developing our query language. We thus built on top of Prolog a series of high-level primitives for querying MIDI files, which, combined with standard logical predicates, produced an easy-to-use and powerful query language: MidXLog. Instead of adopting a dedicated Prolog-driver for reading MIDI files, we developed a translator from standard MIDI format to the MidiXML format [2], using standard Java APIs for querying MIDI events. When started over a given MIDI File, the MidXLog interpreter thus calls the Java translator and then loads the produced MidiXML document into memory, as a complex structure representing MIDI general data and lists of MIDI events.

We do not enter in details on all the characteristics of the language, just limiting to mention the main general filtering primitive, which assumes the following form:

```
filter_events(F, Events, FiltEvents)
```

where Events is the list of MIDI Events, FiltEvents is the list of Filtered Events and F may be a filter of the form:

```
filter(FTYPE, Event)
```

where FTYPE may express different filters based on time intervals, channels, tracks, Note On/Off etc… and/or complex combinations of them.
The query-by-proof approach of Prolog helps in easily redefining particular filters which are subject to specific exceptions. For example, the semantics of a “Note Off” may be expressed both through a real “NoteOff” event than through a “NoteOn” event with Velocity=”0”. So, while the generic filter(type(TYPE), Event) filters events using TYPE as unique index, the specific call to filter(type(“NoteOn”), Event) or filter(type(“NoteOff”), Event) behaves differently according to the above constraint.

As an additional feature, it is possible to combine different filters through logical operators AND, OR and NOT so that, for example:

```
filter_events(((F;G),\+H), Evs, FEvs)
```

retrieves all the elements which satisfy filter `F` *or* `G` *and* which *do not* satisfy filter `H`.

As mentioned above, filtering primitives may then combined with list/set-based logical predicates and math operators which are part of Prolog, to offer a complete query language for MIDI.

2 FEATURE EXTRACTION ARCHITECTURE FOR MIREX

Before extracting features, each MIDI file is subject to a pre-processing phase in which some of the structures which will be often reused during the feature extraction step are generated. The motivations behind these preprocessing phase can be roughly bring back to two main concepts:

- *normalization*: different descriptions of the same musical information (on quantitative and qualitative grounds) must be projected in unique representational plane
- *optimization*: in this case, the information is simply maintained as it is (or even reduced), but is translated in a new form which can reduce computational time for many of the queries needed in the feature extraction phase.

In the following sections we present some of the preprocessing steps which we adopted for our feature extraction system.

2.1 Handling sections with different TimeSignature And/Or KeySignature

For analysis of a certain kind, it is important to keep track of the Time and Key Signature on the different sections. To this end we preferred to separate, once for all, the whole set of MIDI events in different sections where the Time and Key Signature remain constant.

The output of this step is a sequence of structures of the form:

```
(TimeSignature,KeySignature)-Section
```

where `Section` is the partition of the whole set of events which is characterized by the given `TimeSignature` and `KeySignature`, the value of which is decided by the following heuristic:

If (*"midiPiece contains only one KeySignature"*
AND
KeySignature(time(0)) == 'C')
Then guessKeySignatures(midiPiece)
Else computeKeySignatures(midiPiece)

where both computeKeySignatures and guessKeySignatures try to guess which is the harmonic tonal centre of the piece basing on note distribution of pitched instruments (patches 0-111 played on all channels apart from 10) and on other heuristics, with the sole difference that computeKeySignatures takes the number of reported fifths as a strong and reliable evidence for computing the Key Signature, while guessKeySignatures is only partially biased by the solution represented by C (or Am) but must verify this conjecture over solid data verification. This approach is due to typical lack of reported KeySignature, which is then exported by sequencers in the form of a misleading value of 0 on MIDI piece's fifths value.

Guessing in advance the central tone of the piece is crucial as we reported many of the note-related features in terms of their relative grade (with reference to the fundamental tone) instead of their absolute value.

2.2 Handling instrument sections

Due to different MIDI formatting styles which depends on user habits and conventions, we often found disturbing phenomena which fall in these two categories:

- multiple tracks per same channel/instrument (e.g. left/right hands tracks and/or channels used for piano-like instruments, or many tracks per channel 10 used for separating drum instruments)
- multiple instruments on the same channel (e.g. in line program changes) and vice versa (e.g. six-channelled instruments, necessary choice when playing guitar-synths).

For dealing with such phenomena, we almost completely ignored (except when it may reveal useful) the concept of track, in favour of analysis based exclusively on adopted:

- channels
- instruments played on those channels

We thus take account of program changes on the same track as well as of cases where the same instrument is played on two or more different tracks/channels, and produce a list of entries like the following:

```
instr(Instr,InstrSectionDelimiters)
```

where `Instr` is a given instrument which is played along the piece and `InstrSectionDelimiters` is in turn a list of elements like:

```
StartTS-EndTS-ListOfChannels
```

reporting Timestamp intervals where the instrument is played, and for every interval the list of channels hosting notes for `Instr`.

To augment performance in the feature extraction phase for channel based queries, this structure may also be rewritten in terms of "*sections of Time per Channel wrt a list of selected instruments*", so that, for example, the list of instruments' related info:

```
[instr(32, [0-355200-[2]]), instr(0,
[0-177200-[3]]), instr(29, [0-1920-
[16], 143828-212984-[16], 281236-
355200-[16]]), instr(30, [74676-
143828-[16]]), instr(7, [177200-
355200-[3]])]
```

is converted into this new form:

```
[2-[0-355200], 16-[0-1920, 74676-
212984, 281236-355200], 3-[0-355200]]
```

which is notably easier to handle once a bunch of instruments has been selected.

3 USE OF MIDXLOG FOR MIREX

MidXLog proved to be a much useful instrument for inspecting musical data. The high level MIDI querying specific query calls, together with Prolog built-in logical

and meta-logical predicates offered a whole range of operational primitives combined with the more declarative aspects of a proper query language. Unfortunately its development initiated just before our adhesion to the challenge, and this affected the number of feature categories we were able to prepare for our system. These are the features we examined:

Notes Distribution: The frequency with which each note appears in the piece (every note is a set comprehending its different octaves, and is expressed as a relative grade wrt the fundamental tone, which is in turn recalculated for every section of the piece with a stable Key Signature). Note frequency is normalized wrt the number of notes played for the whole piece on pitched instruments. For pitched instruments we intend all STANDARD MIDI instruments belonging to patches 0-111, to avoid disturbs which could arise from percussive instruments (patches 112-119) repeatedly played on only a few notes along the piece and to exclude notes played on SFX instruments (patches 119-127), the pitch of which typically bears a low correlation with a piece's tonal mood.

Drum instruments Distribution: Same as notes distribution for pitched instruments, but in this case notes are not reported as a relative grade wrt the fundamental tone, nor collapsed upon different octaves of the same tone, as each different pitch represent a totally different drum instrument.

Pitched/Percussive/SFX/Drums Percentage: Percentage of notes played by these classes of instruments.

Melodic Intervals: All the basic melodic intervals (for each instrument) within an octave are considered as a numeric feature: legal values indicate the relative frequency for each different melodic interval within the MIDI song.

Instruments (Single Instruments, Binary and Weighted): The 128 patches of the General Standard MIDI patch set surrogates the notion of instrument timbres. We reported both binary vectors which were calculated upon simple presence of the instrument in the piece, and vectors compiled upon the normalized weighted presence of Instruments/Instrument Classes, in terms of played notes per instrument wrt total number of played notes on all considered instruments.

Instrument Classes and Drum-kits: Analogous vectors of those for single instruments, but related to instrument classes. Each GSM patch is associated to exactly one of the common sixteen different instrument classes (i.e. Piano-like instruments, Strings, Synth Pads, Brass and so on). For drums, we considered the 8 different drum-sets always associated with the MIDI channel 10. The different instrument classes and drum kits are here expressed both as boolean features and weighted normalized features.

Tempo Related Features: such as:

- *number of tempo changes* inside the piece
- *standard deviation of stable times* (stable times are times where the tempo remains unchanged). Times have been left as they were and being normalized wrt the length of the piece.
- *standard deviation of relevant stable times only* (relevant times exclude local and gradual tempo changes, as for *crescendo* over time).
- *weighted mean time*: mean time taken upon weighted measures of tempo changes, based on their stable time extensions

Time-Signature Related Features: such as:

- *number of TS* changes inside the piece
- *standard deviation of stable times* (stable times are times where the TS remains unchanged). Times have been left as they were and being normalized wrt the length of the piece.
- *standard deviation of relevant stable times only* (relevant times exclude local TS changes, as for TS patch bars).
- *Distribution of TS along the piece*: distribution of different TS along the piece, weighted over their persistence in the piece. Several Time Signatures have been considered as a possible feature, while a generic "OthersTimeSignature" has been adopted for weighting unknown TSs.

Key-Signature Related Features: such as:

- *number of KS* changes inside the piece
- *standard deviation of stable times* (stable times are times where the KS remains unchanged). Times have been left as they were and being normalized wrt the length of the piece.
- *standard deviation of relevant stable times only* (relevant times exclude local KS changes, as for certain kinds of modulations).
- *Distribution of KS along the piece*: distribution of different KS along the piece, weighted over their presence (time) in the piece.

Pitch Wheel Related Features: This class of features gives some more notion about playing style of the music piece (synth-soloing, guitar bending etc…). For this contest we limited our investigation to analyzing which instruments exhibit a *Pitch Wheel* control during their performance.

4 CONCLUSIONS

The final MIREX results are more than encouraging. The two partecipating systems were based on standard machine learning algorithms (Naïve-Bayes (NB) and Decision trees) and no specific tuning on the MIREX task has been carried on. Most of the effort has in fact been spent on the development of the MidXLog-based infrastructure (e.g. design and development of the language itself, algorithms for feature matching in the logical formalism adopted). As a result not much time was available to extend the range and type of features according to the target musical genres [4]. Nonetheless, the NB classifier on the set of extracted features simpler than other proposals ranked slightly below the best overall system.

A first analysis could suggest that normalization of MIDI data, handled during the preprocessing stages in

MidXLog, may have played a strong role in removing some MIDI idiosyncrasies which may have been dangerous. A relevant aspect is the relatively small number of feature types (about 20) with respect to other proposals. In particular, some of these are novel, as far as we know, in the literature. Extensive testing will be required to judge their individual quality and how they impact on global performances if combined with already explored features (as in [4]). A further dimension to explore is the learning algorithmics: for example, in [1] we noticed that NB and decision trees were not always optimizing the classification accuracy. Alternative learning methods ([5]) or an analysis of more complex learning processes (e.g. stacking or voting among cascades of classifiers) is thus also a future line of research not explored in this work.

The statistics behind our score, and their comparison with the best results of MIREX suggests that large margins for improvement exist. The overlap between the two outcomes is rather low. In particular, our 55% accuracy for recognition of the "classical" genre is strongly under the 100% of the best reported result. Notice how this task is easier according to some our previous tests ([1]). On the other hand, some of our 100% accuracy are achived over very specific genres (e.g. Ragtime). This is surprising as no specific feature set has been designed targeted to those genres. These discrepancies let us foresee possible improvements when extension including features inspired by other systems are applied. This would be a suitable outcome from the MIREX panel discussion with the other participants.

ACKNOWLEDGEMENTS

We would like to thank Stephen Downie and M. Cameron Jones for all their support and patience during the run of the contest experiments.

REFERENCES

[1] Basili R., Serafini A., Stellato A.: "Classification of Musical Genre: A Machine Learning Approach". ISMIR 2004. 5th International Conference on Music Information Retrieval, Barcelona, Spain, October 13, 2004

[2] Good, M., "MusicXML: An Internet-Friendly Format for Sheet Music". In XML 2001 Conference Proceedings, Orlando, FL, December 9-14, 2001

[3] Kowalsky. R. Predicate logic as a programming language. In Proceedings of the 1FlPS Conference 1974. North-Holland, Amsterdam, pp. 569-574

[4] McKay, C. 2004. Automatic genre classification of MIDI recordings. M.A. Thesis. McGill University, Canada.

[5] N. Cristianini, J. Shawe-Taylor, An Introduction to Support Vector Machines and other Kernel-based learning methods, Cambridge University Press, 2000.