

PROJ-H-402 – Overscore

Quentin Stievenart

April 20, 2013

1 Introduction

1.1 Goal of the project

The goal of this project is to build a system able to recognize scanned musical scores and play them using an existing open-source audio environment (*Overtone*¹). Such system is called an *Optical Music Recognition system*, or *OMR system*. Another goal of the project is to develop a text alternative to the classical musical notation, which could enable the development of truly open-source music in text form.

1.2 Overtone

Overtone¹ is an *open-source audio environment being created to explore musical ideas from synthesis and sampling to instrument building, live-coding and collaborative jamming*[2]. It provides a programmable synthesizer, using the Clojure² programming language.

As said before, one of the goal of the project is to provide a text alternative to the classical music notation. This text notation can easily be developed as a set of functions and macros on top of Overtone, and the resulting files can then directly be interpreted by Clojure, without using any additional software.

1.3 Existing OMR systems

Several OMR systems already exist, and it is a subject that has been thoroughly studied by the research community since 1966[25]. However, most implementations of OMR systems are either commercial[15, 17, 18, 19, 21, 28], or result of research work from which the source code has never been published. One of the few research work that have published their source code resulted in the Gamera MusicStaves toolkit[9], which implement various algorithm for staff line removal (see section 3.2). The only open-source OMR systems are Audiveris[5] and OpenOMR[10].

2 Musical Notation

This section will define the musical notation concepts needed to understand the rest of this document without knowledge of music theory.

A *score* is a written representation of a musical work, divided into *staves*, each *staff* consisting of five horizontal lines (the *staff lines*), and containing musical *symbols*. The musical symbols we will focus on in this work are represented in table 1.

¹: <http://overtone.github.com/>

²: <http://clojure.org/>

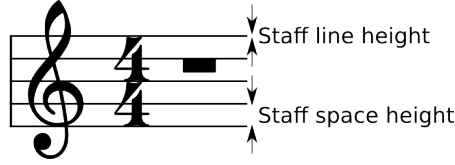





Figure 1: Reference lengths of a score

The *pitch* of a note is its frequency, and is described by a *step* and an *octave* (eg. the pitch of the note A_4 note is 440Hz^3 , where A is the step, and 4 is the octave)

Table 1: Description of some musical symbols

Name	Examples	Description
Clef		The clef is generally the first symbol appearing on a staff, and describes the correspondance between note pitch and the staff lines.
Note		A note consists of three parts: the head (of circular shape), the stem (the vertical line), and the flag (attached at the end of the stem). When multiple notes follow each other, their flag can assembled and it forms a beam . The flag of a note indicates its length, and the vertical position of the note indicates its pitch (which depends on the clef).
Sharp, Flat and Natural	$\sharp \flat \natural$	Those symbols modify the pitch of a note, and are generally placed to the left of the note they modify.
Rest		A rest is equivalent to a note with no pitch (ie. no sound is played during a certain duration).
Time signature	$\frac{4}{4} \quad \frac{3}{8}$	The time signature describes the rythmic structure of the measure.

We define the **reference lengths** as the height of a staff line (the *staff line height*, represented by d) and the distance between two staff lines (the *staff space height*, represented by n). This is represented in figure 1.

3 Architecture of OMR systems

This section will briefly describe the generic architecture of OMR systems. A more detailed description is available in the project's documentation ([doc/design/design.pdf](#)). The implementation of the OMR system for this project is described in section 4.1.

³: This document uses the gregorian notation for musical notes. A_4 represents the note A at the fourth octave. In some countries, the latin notation is more common: *do* (C), *re* (D), *mi* (E), *fa* (F), *sol* (G), *la* (A), *si* (B).

The architecture of OMR systems can be seen as a pipeline, where each stage takes as input what the previous stages produced. The main stages are: the *preprocessing* step, the *staff line processing* step, the *symbol recognition* step, and the *musical semantics* step.

3.1 Preprocessing

The preprocessing step takes as input the scanned image with no further information. Its purpose is to convert the image to a binary image (black and white), to reduce the noise of the image (the scanned image may have some defects), and to identify the reference lengths.

The conversion from colored image to black and white is done in two steps:

1. The image is converted into grayscale by applying the following formula for each pixel[14]:

$$Gray = 0.3 R + 0.59 G + 0.11 B$$

2. The image is converted from grayscale to a binary image, using a **binarization algorithm**. There exists two main classes of binarization algorithms:
 - (a) **Global thresholding** algorithms, in which a global thresholding value is computed based on the pixels of the image, and remains the same for the entire image. Pixels whose value is above this threshold are considered white, and pixels whose value is below it are considered black. The most commonly used global thresholding method is **Otsu's method**[22]
 - (b) **Adaptive binarization** algorithms, in which the threshold is different for each pixel, and depends only on the neighbor pixels. The most commonly used adaptive binarization method is **Niblack's method**[20].

Other binarizations methods exists[6, 12], and performs better on degraded music scores[7]. However, most of the scanned music scores available[1, 3, 4] are not degraded, and most of them are already binarized.

To improve the quality of the image, a basic noise reduction algorithm that eliminate isolated black pixels and fill isolated white pixels can be applied. Since scans are not always performed perfectly horizontally, a skew correction algorithm can be applied[10]. Other techniques can also be used[13], but in most cases a basic binarization method is sufficient to have a good quality binary image.

Once the image is binarized and cleaned, the reference lengths can be found using a simple algorithm based on *run-length encoding* (RLE): for each column of the image, encode the pixels using RLE (see figure 2). The most common black run corresponds to the staff line height, and the most common white run to the staff space height.

3.2 Staff Line Processing

The staff line processing step takes as input the binary image produced by the preprocessing step, and do the following:

1. Identify the position of the staff lines: this is important to be able to interpret the pitch of the notes

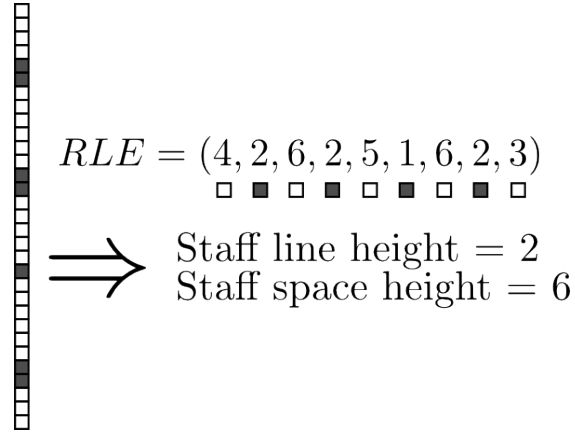


Figure 2: Finding reference lengths with run-length encoding



Figure 3: Finding staff lines with y -projection

2. Remove the staff lines: from an image analysis point of view, the staff lines make the analysis of the symbols harder, because they connect lots of symbols together. Removing the staff lines thus greatly simplifies the symbol segmentation step (described in section 3.3.1). This step is not mandatory, but very few OMR systems skip the removal of the staff lines[10].

3.2.1 Staff Line Identification

To identify staff line positions, the most widely used method[11] consists of doing y -projection of the pixels, and analyzing the maximas on this projection, since the staff lines horizontally cover most of the score. To find a staff, it is sufficient to find five maxima spaced of the staff space height. An example is given in figure 4.

3.2.2 Staff Line Removal

Lots of different algorithms exist for removing staff lines:

- Linetracking Runlength[23]
- Linetracking Chord[16]
- Carter[8]
- Fujinaga[11]
- Roach and Tatem[26]
- Skeleton[9]

However, according to [9], no algorithm performs significantly better than the others. For undeformed images, *Roach and Tatem's* algorithm performs slightly better than the others.

Some of these algorithms have been implemented and made public in a *Gamera* toolkit⁴.

3.3 Symbol Recognition

Once the staff lines have been identified and removed, the symbols should be detected and classified. The symbol recognition steps should thus output:

- the position of each symbol, which is found during the *symbol segmentation* step;
- the class of each symbol, which is found during the *symbol classification* step.

3.3.1 Symbol Segmentation

This step analyzes the binary image in order to find groups of black pixels, that should roughly correspond to musical symbols. It then refines those groups of pixels to decompose the symbols into smaller parts (eg. a note is decomposed as: the note head, the stem, and the flag (or the beam)), that can then be classified.

Unfortunately, there is not much description of algorithms performing this step in the literature. One exception is [10], which also provides a Java implementation in OpenOMR. Audiveris[5] also provides an implementation of a segmentation algorithm, but not much documentation.

3.3.2 Symbol Classification

Once the score is segmented in small segments corresponding to the musical symbols, those segments have to be classified, in order to assign to each segment a *label*, describing what type of symbol it is (eg. *notehead black*, *G clef*, *eighth rest*, ...).

This classification can be performed with well-known classification algorithms, such as *Neural Networks*, *k-Nearest Neighbors*, *Support Vector Machines*, or *Hidden Markov Models*. According to [24], the two best methods are the Support Vector Machines and the k-Nearest Neighbors, that respectively performs better on handwritten music and typeset music. However, many OMR system implementations use neural networks[5, 10].

All classification algorithms require a *training set*, consisting of already classified data that is fed to the algorithm to train it. The bigger the training set, the better the achievable classification performance. Only two public training sets are available:

- OpenOMR's training set, consisting of 727 examples;
- Audiveris' training set, consisting of 4159 examples.

This data set should then be splitted into three sets: the actual *training* set, used to train the algorithm, the *test* set, used to measure the error of the algorithm in order to know when to stop the training, and the *cross-validation* set, used to see how the algorithm behaves on previously unseen data[27]. Typically, the training set contains 50% of the data set, and the test and cross-validation sets contain 25% each.

⁴: <http://music-staves.sf.net>

3.4 Musical Semantics

The final step of an OMR system is to output the musical symbols in a way such that they can be interpreted (to be played by a synthesizer for example), or represented (in a MusicXML file for example). This is done using the data produced by the previous steps: given the positions of the staff lines, the class and positions of the symbols, we can infer the pitch and duration of the notes, as well as other characteristics of the score (the time signature, ...).

As for the segmentation step, this step is not much described in the literature, and the only examples of implementation available are OpenOMR and Audiveris.

4 Implementation

This section will describe how the whole system has been implemented and its architecture. The system consists of three parts:

1. The OMR system, which takes a scanned image as input, and outputs a file in an intermediate format. The architecture of OMR systems has already been described in section 3, and the actual implementation is described in section 4.1.
2. A parser for the intermediate format that outputs a `.clj` file, using the defined musical notation, that can be played with Overtone. The intermediate format choice is described in section 4.2.
3. The defined musical notation, which consists of an extension of Overtone's features, and is described in 4.3.

4.1 OMR System

The OMR system has been implemented in such a way that it is easy to replace the implementation of one of the steps with another implementation, without depending on anything of this system (eg. the programming language). This is achieved by having a clear separation between each step, and having each step reading all its input from one or multiple files, and writing all its output to one or multiple files. This should allow future work to be easily plugged into this system.

4.1.1 Preprocessing

The preprocessing step is the simplest and works in three steps:

- if the input image is not (at least) in grayscale, convert it to grayscale using the formula given in section 3.1,
- if the input image is not already in black and white, binarize it using Otsu's algorithm[22]. This algorithm has been chosen because it is simple and well documented, and the accuracy of this step is not crucial,
- compute the reference lengths using the RLE algorithm described in section 3.1.

The whole binarization part turned out to be mostly useless, since almost all the scores available online are already binarized. However it was a good way of gaining an overview of Java's image manipulation libraries.

4.1.2 Staff Line Processing

The staff line processing step has been implemented into two sub steps.

First, *staves* are identified and isolated. A staff consists of five staff lines and the symbols they contain (along with the symbols that are near the staff lines, but not directly on them). Those staves are located in a method inspired by Fujinaga’s method for identifying staff lines[11], by analyzing a y -projection of the image, and looking for five peaks of black pixels that are separated from a similar distance. Once those peaks have been found, the boundaries of the staff are found by looking in the neighborhood of the staff lines for the y coordinate with the minimal number of black pixels.

Once those staves are identified, they are isolated into different images. Those images are then passed to the next sub step: the staff line removal step.

This sub step has not been implemented from scratch and uses Gamera’s *musicstaves* plugin, which already implements multiple staff line removal algorithms. The algorithm chosen is *Roach and Tatem’s*, since it performs better than the other ones available in this plugin[9]. The Clojure codes thus calls a Python script that uses this plugin to remove staff lines.

With the current implementation, the staff line processing step works well and produces the expected results on computer-generated scores (with minor errors in the staff line removal, that don’t affect the other steps, see figure 5).



Figure 4: The identified staff lines are displayed in red.



Figure 5: Same score, with the staff lines removed. Some horizontal segments that should not be removed are removed (eg. in the lower part of the clef).

When the staff lines are not perfectly straight, the removal algorithm might have some trouble removing them (see figure 6). To solve this, two refinements could be added to the preprocessing step:

- a *skew correction* algorithm in case the score was not perfectly aligned when it has been scanned, this is done in OpenOMR[10],
- an algorithm that behaves like Audiveris’ *grid step*[5], which scans the score for vertical lines, and generate a score with those vertical lines horizontally aligned, thus *unwarping* the image.

4.1.3 Symbol Recognition: Segmentation

The symbol segmentation step divides the staff in *segments*. This step is heavily inspired from OpenOMR’s segmentation[10], and is implemented in multiple levels:



Figure 6: Error during the staff line removal step, due to the fact that the remaining staff line is slightly curved at its end

1. The *Level-0* ($L0$, see figure 7) segmentation horizontally divides the staff into groups of symbols:
 - it first creates segments corresponding to the regions containing consecutive black pixels in their x-projection,
 - those segments are then refined by grouping the segment which are close (eg. the $L0$ segment for a dotted note should contain the note and the dot), and by removing segments that are small (which can result from noise in the score image).
2. The $L0$ segments are then analyzed to detect if they contain a note head, by looking for consecutive columns that contain a black run of a certain size (corresponding to the height of the note head, which is computed from the reference lengths extracted during the preprocessing step). The $L0$ segments that contains no note heads are kept as is, while the other go through Level-1 and Level-2 segmentation.
3. The *Level-1* ($L1$, see figure 8a) segmentation horizontally divides the $L0$ segments in order to isolate the segments that contains the note heads from the other segments (eg. a sharp). To do so, it uses the same method that is used to detect the presence of note heads to filter $L0$ segments.
4. The *Level-2* ($L2$, see figure 8b) segmentation vertically divides the $L1$ segments in order to separate vertically aligned symbols. It is done by analyzing a y -projection of the $L1$ segment, and grouping runs of black pixels that are close, while separating the ones that are not close.

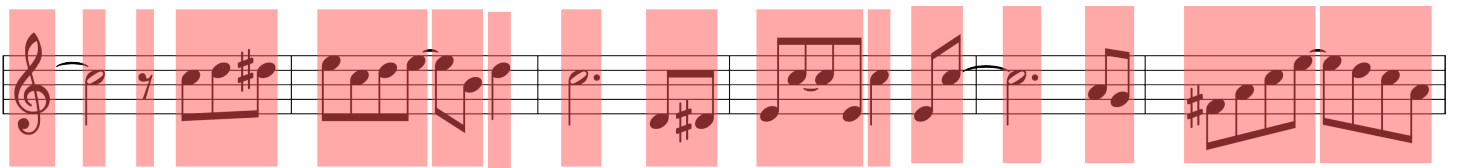


Figure 7: Level-0 segments. Note that the staff lines are still present because OpenOMR does not remove the staff lines.

All the values used in the computation (eg. to define how spaced should be two pixels to be considered *close* or *not close*) depend on parameterizable factors and on the reference lengths of the score.

This implementation works but has some issues. Depending on the values of the reference lengths (that can be tweaked by the user), it either correctly divides small symbols and incorrectly divides big symbols (see figure 9), or does not divide small symbols and correctly divides big symbols (see figure 10).

Also, the correctness of the output varies a lot from score to score, but it seems to always be possible to achieve an acceptable result by tweaking the reference lengths.

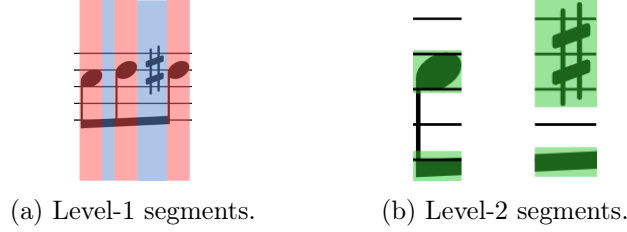


Figure 8: Final steps of segmentation.



Figure 9: Segments found for a score with reference lengths $d = 2, n = 11$



Figure 10: Segments found for a score with reference lengths $d = 1, n = 18$

4.1.4 Symbol Recognition: Classification

The symbol classification step has been implemented multiple times (and the implementation used can easily be changed, by changing a few lines in the source code of the application). The training set used is Audiveris' training set.

First of all, the input data have to be normalized in order to be manipulable by the different methods (eg. neural networks only have a fixed number of inputs while the sizes of the segments vary, so the input data has to have a fixed size). Thus, every element of the training set, and every data that will be classified, is first resized to a 20×20 image, and then converted to a 400-bit vector. This size has been chosen because it is large enough to represent most of the symbols, while not being too large, which might slow down and increase the memory usage of the application.

The first implementation uses the *k-Nearest-Neighbor* (kNN) algorithm, since it is a simple algorithm that yields really good results for classification of typeset musical scores[24]. However, this implementation is made from scratch and is really slow (it takes more than 30 seconds to classify a symbol). In order to avoid losing time optimizing it, and since the performance gap with other already existing solutions was huge, it has been chosen to use existing libraries to implement this step.

The second implementation uses neural networks, through the *Encog*⁵ library. The neural network used has the following characteristics:

- 400 input nodes (one for each pixel of the resized image),
- one hidden layer consisting of 400 nodes (arbitrary value),
- n output nodes, where n is the number of classes for the symbols, and depends on the classes represented in the training set.

⁵: <http://www.heatonresearch.com/>

When classifying, the class chosen is the one whose output node value is the highest. If no output node have a value higher than a certain threshold (set to 0.2 by default), the classifier returns `:empty` (the class corresponding to no symbol).

The neural network is trained using a training set consisting of 50% of the data set, and tested on a test set consisting of 25% of the data set (the usage of the the remaining 25%, the cross validation set, is explained later). The training set is used by Encog to adjust the weights of the neural network, while the test set is used to compute the error after adjusting those weights. The weights are adjusted until the error reaches an acceptable value, or until a number of iterations has been done.

This implementation works well, but requires a long training time (more than one hour) to reach an accepting error rate on the test set. After training, the neural network state is saved in the file system to avoid further training.

The last implementation also uses the kNN algorithm, but this time through the *OpenCV*⁶ library, directly in C++. It only needs to load the training set (storing it in 400-bit vectors), and does not require a long training as for the neural network, but works as fast.

One major problem of this step is that the majority of the data in the training set corresponds to only a few symbols: note heads covers 40% of the training set, and note-heads and beams together covers 65% of the training set. Thus, the classification step has good performance on the data that is well represented in the training set, but performs poorly on other data. Fortunately, the data well represented in the training set corresponds to the data that is more frequent on scores, so the performances on a well segmentized score are good (see figure 11). Ideally, the training set should be improved in order to represent every symbol equally.

The two usable methods (OpenCV's kNN and Encog's NN) were evaluated by computing the *precision*, the *recall*, and the *F₁-score* as follows:

- OpenCV's kNN was trained with a training set consisting of 75% of the data set, the 25% remaining being kept for the cross validation.
- Encog's NN was trained with a training set consisting of 50% of the data set, and a test set consisting of 25% of the dataset, 25% remaining also being kept for the cross validation.
- For each method, for each element in the cross validation set:
 - classify the element,
 - if the element is correctly classified, increment the *true positive* count for this class, and increment the *true negative count* of every other class,
 - if the element cannot be classified (ie. the algorithm outputs no class for this element), increment the *false negative* count for the class of this element, and the *true negative* of every other class,
 - if the element is misclassified (classified *x* while it should be classified *y*), increment the *false positive* count of *x*, the *false negative* of *y*, and the *true negative* of every other class.

⁶: <http://opencv.org/>

- For each class, compute the *precision* (P), the *recall* (R), and the F_1 -score as follows:

$$P = \frac{tp}{tp + fp}$$

$$R = \frac{tp}{tp + fn}$$

$$F_1 = 2 \frac{P R}{P + R}$$

where tp stands for *true positive*, fp for *false positive*, and fn for *false negative*.

Since some classes contain very few elements, they might not be present in the (randomly selected) cross-validation set, and thus the scores for those classes are set to 0. The scores is also set to 0 if no element of this class was correctly classified. The results are shown in figures 12 and 13 (each triplet of bars corresponds to the scores for one class of symbol), where we can see that the kNN method performs better than the neural network.



Figure 11: Classification results using OpenCV’s kNN algorithm, performed on a score segmentized by hand, achieving a success rate of 95%. On the same data, the neural network has a success rate of 67%. Each segment is labeled with the name of the associated class.

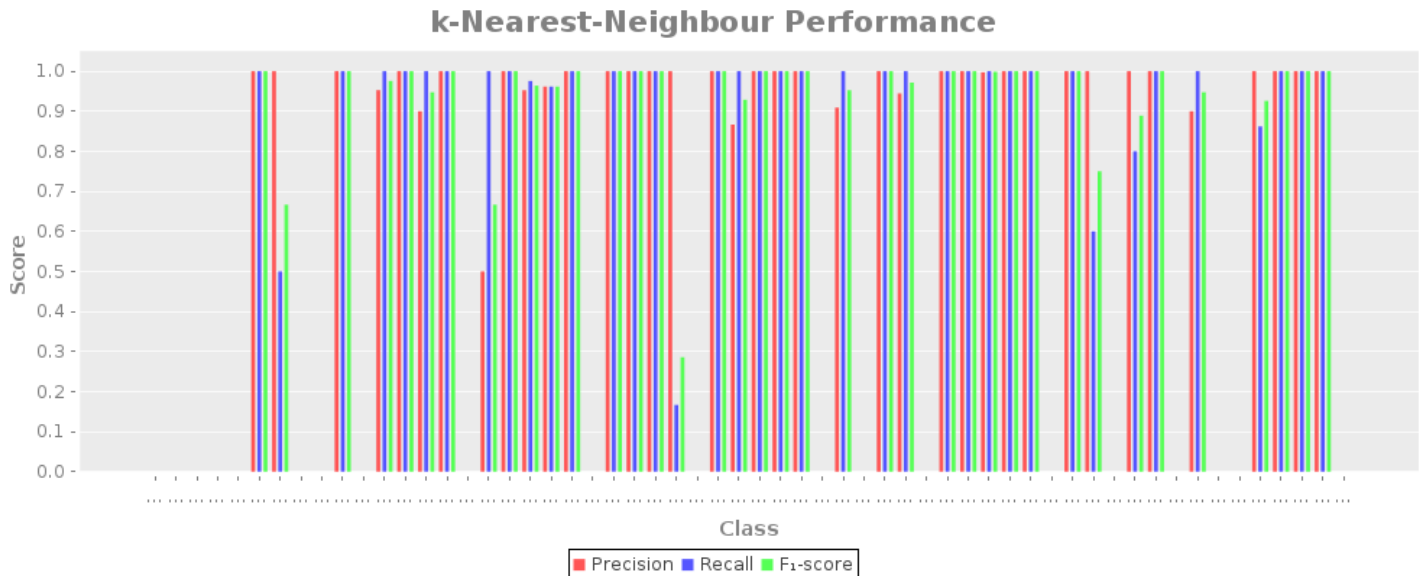


Figure 12: Performance of the kNN algorithm on a random cross-validation set (consisting of 25% of the data)

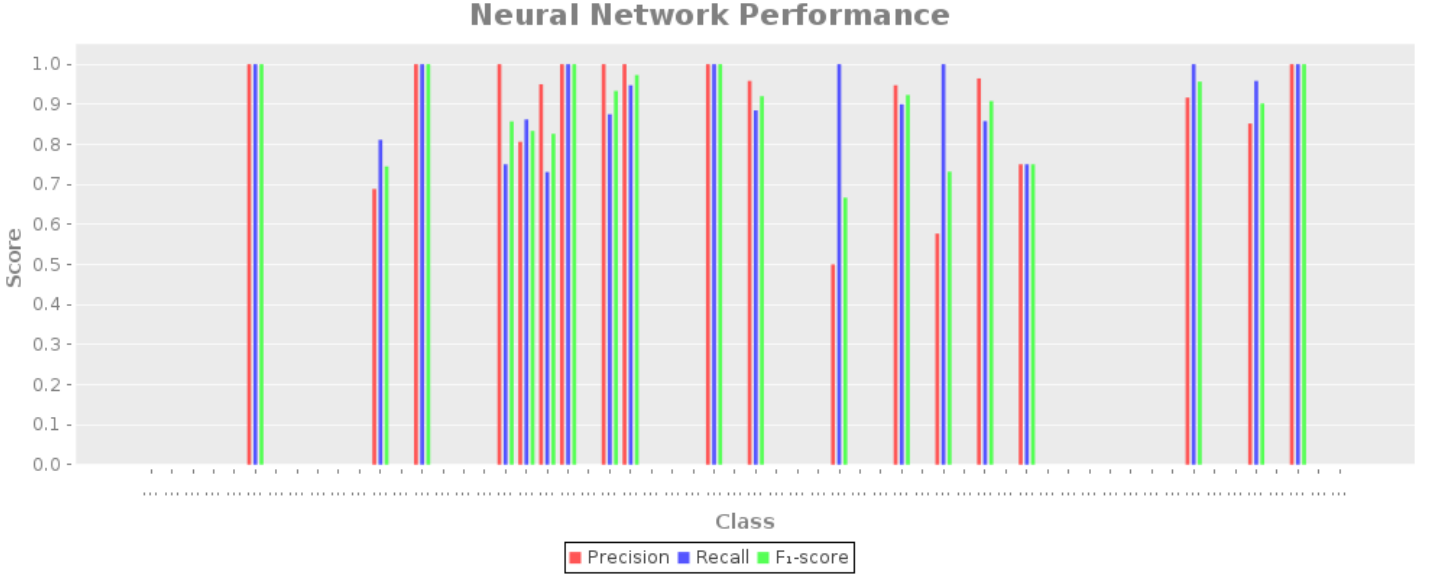


Figure 13: Performance of the neural network on a random cross-validation set (consisting of 25% of the data)

4.1.5 Musical Semantics

Since no description of this step has been found, it has been implemented from scratch, and is not based on any existent method. The implementation of this step is divided in two parts: the *grouping* of segments, and the *parsing* of those groups of segments.

The grouping part consists of grouping all the different characteristics of a musical feature together: a note head will be grouped (when possible) with its accidental (sharp, flat, ...), its beam or flag, and its dot. The elements that should possibly be grouped can be divided into three classes:

- the *pre* symbols: the accidentals and the beams,
- the *cur* symbols: all of the *pre* and *post* symbols, and the noteheads,
- the *post* symbols: the flags and the dots.

First, the symbols which are superposed vertically are grouped together (eg. a note-head with a beam below it). Then, the groups are scanned left-to-right in order to merge those that can be merged:

- a group containing only *pre* symbols can be merged with a group containing only *pre* symbols, or with a group containing a *cur* symbol,
- a group containing *cur* symbols can only be merged with a group containing only *post* symbols,
- a group containing *post* symbols can only be merged with a group containing only *post* symbols.

After this grouping part, every group should contain either only one element, or one or more elements if it contains a notehead. No group can contain more than one note-head (we make the simplifying assumption that the score does not contain two vertically

aligned noteheads, which might arise in practice, but is ignored by this OMR system for simplicity reasons).

Once the symbols are grouped, the groups are parsed in order to produce a structure representing the score. This parsing step is based on a simple grammar (given in the documentation of the project). Once the groups are parsed, the conversion from the data structure to the MusicXML file is done straightforwardly.

If the data used as input for this step is correct, the conversion is perfect on simple examples (scores that do not contain superposed noteheads, or symbols not taken into account by the system).

4.1.6 Overall Performance

While every step works correctly, with more or less good accuracy, the overall system performs poorly. This is due to the fact that every little error introduced in a step results in bigger errors in the subsequent steps, and results in completely incoherent data at the end of the process. The step that performs the poorest (and which is probably the most important step) is the segmentation step, and is thus the first step that should be improved in order to have better performance in the overall system. For further changes to the system, the changes should be done in preference to the beginning of the pipeline in order to have a better performance impact.

Also, the system is made to work with score with a very simple structure: a lot of simplifying assumptions are made. It won't work correctly with scores with more complex structure (more than one staff played at the same time, non common symbols, text annotations, repetitions of bars, ...).

4.2 Intermediate File Format

The OMR system implemented outputs an intermediate file format instead of directly outputting Overtone code. This allows the OMR system to be completely independent of the notation, and conversely. The OMR system can thus output files that can be read by other applications, and files created by other musical applications can be converted into the defined notation and played with Overtone.

The existing file formats that could be used to encode musical notation are: MusicXML (`.mxml`), MuseScore (`.mscx`), CapXML (`.capx`), MIDI (`.mid`) and NIFF (`.nif`). The format chosen is MusicXML, for multiple reasons:

- it is, as MIDI, supported by almost all musical notation softwares. MIDI however is not designed for notation, but for playback (and results in a lack of information when used to represent musical scores),
- it is designed to represent musical scores without any information loss,
- it is still maintained and actively developed, while remaining stable (the latest version is from 2011),
- the specifications of MusicXML are verbosely documented and it is an open format,

- softwares like MuseScore⁷ can do the translation between CapXML, MusicXML and MIDI.

The conversion between MusicXML and the defined musical notation (described in section 4.3) consists of a set of simple rules described in the documentation of the project ([doc/conversion/conversion.pdf](#)).

4.3 Musical Notation

This section describes the musical notation developed with the aim of having a simple text notation able to represent classical musical scores. The notation consists of a set of Clojure functions and macros that rely on Overtone's features to play the score.

4.3.1 Notes and rests

The most basic element of a score is a note. A note is expressed as its duration and its pitch:

```
(play :A4 1)
```

This corresponds to a 440Hz A (4th octave), played as a quarter note. The interpretation of the duration depends on the time signature and the tempo of the score. In this case, we assume that the time signature is 4/4, so a duration of 1 corresponds to a quarter of the entire bar (so, a quarter note).

A rest is simply a note without pitch, and is written `:rest:`

```
(play :rest 1)
```

4.3.2 Bars

A bar contains notes played at certain times. With the most basic constructs, it can be defined as a set of notes and the time they have to be played at:

```
(bar
  (beat 0 (play :C4 1))
  (beat 1 (play :A4 1))
  (beat 2 (play :G4 1))
  (beat 3 (play :C5 1)))
```

A bar can also be named, to be referred to later:

```
(defbar foo
  (beat 0 (play :C4 1))
  (beat 1 (play :A4 1))
  (beat 2 (play :G4 1))
  (beat 3 (play :C5 1)))
```

Multiple combinators simplifies the notation, and are described in section 4.3.5.

⁷: <http://musescore.org/>

4.3.3 Progressions

A progression is a set of bars to be played in sequence. It can also be defined anonymously with `prog`, or named with `defprog`:

```
(defprog foo-twice
  foo foo)
```

Progression definitions can also be simplified through the use of combinators described later.

The tempo and time signature can be changed during a progression:

```
(defprog foo-twice
  {:tempo 60} foo {:tempo 40} foo)
```

4.3.4 Songs

A song consists of a set of progressions, played simultaneously, associated with a set of instruments:

```
(defsong foo-song
  {:time-signature [4 4] :tempo 60}
  [foo-twice sampled-piano]
  [foo-twice pad])
```

4.3.5 Combinators

Notes, bars and progressions are internally represented as a function that takes a state (containing the tempo and the time signature), a time (at which to play the element), and an instrument, and returns the duration of the element. When called, those function spawn Overtone notes that will be played at the given time, and returns immediately. So, they are all considered as *elements*, and can be manipulated with the following predefined combinators:

- `play-chords`: play all the arguments at the same time,
- `play-seq`: play all the arguments one after the other,
- `simple-seq`: a macro that ease the writing of multiple notes in sequence, without needing to call `simple-seq` and `play`,
- `beat`: delay the time at which the element will be played by `n` beats,
- `repeat-elements`: repeat an element `n` times

For example:

```
(defbar foo
  (repeat-elements 2
    (play-seq
      (simple-seq 1/2 :C4 :A4 :G4)
      (play-chord
        (play :C5 1/2))
```

```

(play :A5 1/2)
(play :G5 1/2))))))

(defprog foo-prog
  (repeat-elements 2 foo))

```

If needed, other combinators can be defined easily, since they only consist of manipulating Clojure functions.

4.3.6 Playing a score

A score can be played by using the `start` function, which takes a song as argument. An element (note, bar, progression) can be played using the `start-element` function, which takes as argument at least the element and the instrument it should be played with (and optionally the tempo and time signature).

4.3.7 Improvements

The notation currently works well and is sufficient for encoding the scores used while developing the OMR system, but lacks lots of classical musical constructs. It should not be hard to extend it to support more complicated musical constructs.

One minor defect of the current implementation is that a lot of Overtone nodes (which represent notes) are spawned, and might not be *killed* if the score is stopped before it ends. Since this number of nodes is limited, it might cause errors when playing long scores, or playing other scores after having stopped one. However, in practice, this problem only arises on long debugging sessions, where lots of scores have been played and stopped.

5 Conclusion

The project has reached a state where every part is implemented, but most of them can be improved. In the OMR system, each step performs correctly, but the whole system performs poorly. This is mostly due to small errors introduced early in the pipeline, and to the small sizes of the training set available. The OMR system can thus be heavily improved: some possible improvements were discussed in this report, and more information is available in the project's `README`. Also, since the system is implemented in such a way that every step is clearly independent of the other steps, it could be used to provide a whole OMR system to test implementations of algorithms for one step of an OMR system.

The MusicXML parser and the musical notation remains basic, but are complete enough to describe some simple classical musical scores (and also most of the non-classical musical scores, where much less musical concepts are used). It should not be too hard to extend it to support more musical constructs.

All the code is heavily documented (which is unfortunately not the case of other existing OMR systems), and documents providing more information are also included in the project's documentation.

The project has been published on the author’s Github, under the name `overscore`⁸.

Should future work be made on this system, multiple points could be improved, touching to different subjects:

- substeps could be added to the preprocessing: deskewing, dewarping, noise-reduction, ... (simple image analysis)
- the segmentation step could be improved (more complex image analysis)
- the training set could be completed with more classes of symbols, and with more data on existing symbols (machine learning)
- the notation could be extended to support more musical constructs (language design, musical theory)

References

- [1] IMSLP: The Petrucci Music Library. <http://imslp.org/wiki/>.
- [2] Overtone website. <http://overtone.github.com/>.
- [3] Project Gutenberg, The Sheet Music Project. http://www.gutenberg.org/wiki/Gutenberg:The_Sheet_Music_Project.
- [4] The Mutopia Project: Free sheet music for everyone. <http://www.mutopiaproject.org/>.
- [5] H. Bitteur. Audiveris handbook. <http://audiveris.kenai.com/docs/manual/handbook.html>.
- [6] A. D. Brink and N. E. Pendock. Minimum cross-entropy threshold selection. *Pattern Recognition*, 29(1):179–188, 1996.
- [7] J. A. Burgoyne, L. Pugin, G. Eustace, and I. Fujinaga. A comparative survey of image binarisation algorithms for optical recognition on degraded musical sources. In *ISMIR*, pages 509–512, 2007.
- [8] N. P. Carter and R. A. Bacon. Automatic recognition of printed music. In H. S. Baird, H. Bunke, and K. Yamamoto, editors, *Structured Document Image Analysis*, pages 456–65. Springer-Verlag, Berlin, 1992.
- [9] C. Dalitz, M. Droettboom, B. Pranzas, and I. Fujinaga. A comparative study of staff removal algorithms. *IEEE Trans. Pattern Anal. Mach. Intell.*, 30(5):753–766, May 2008.
- [10] A. Desaedeleer. Reading Sheet Music. Master’s thesis, Imperial College London, Technology and Medicine, Department of Computing, London, 2006.
- [11] I. Fujinaga. Staff detection and removal. In Susan George, editor, *Visual Perception of Music Notation: On-Line and Off-Line Recognition*, pages 1–39. Idea Group Inc., 2004.

⁸: <http://github.com/acieroid/overscore>

- [12] B. Gatos, I. Pratikakis, and S. J. Perantonis. An adaptive binarization technique for low quality historical documents. In Simone Marinai and Andreas Dengel, editors, *Document Analysis Systems*, volume 3163 of *Lecture Notes in Computer Science*, pages 102–113. Springer, 2004.
- [13] R. Göcke. Building a system for writer identification on handwritten music scores. In *Proceedings of the IASTED international conference on signal processing, pattern recognition, and applications*, pages 205–255, Anaheim, 2003. Acta Press.
- [14] J. He, Q. D. M. Do, A. C. Downton, and J. H. Kim. A comparison of binarization methods for historical archive documents. In *Proceedings of the Eighth International Conference on Document Analysis and Recognition*, ICDAR '05, pages 538–542, Washington, DC, USA, 2005. IEEE Computer Society.
- [15] Visiv Ltd. SharpEye. <http://www.visiv.co.uk/>.
- [16] P. Martin and C. Bellissant. Low-level analysis of music drawing images. *Proceedings of the International Conference on Document Analysis and Recognition*, pages 417–25, 1991.
- [17] Leoné MusicReader. MusicReader. <http://www.musicreader.net/>.
- [18] Musitek. SmartScore. <http://www.musitek.com/smartscore-pro.html>.
- [19] Neuratron. Neuratron PhotoScore Ultimate 7. <http://www.neuratron.com/photoscore.htm>.
- [20] W. Niblack. *An introduction to digital image processing*. Strandberg Publishing Company, Birkerød, Denmark, Denmark, 1985.
- [21] Forte Notation. Forte Scan Light. <http://www.fortenotation.com/en/products/sheet-music-scanning/forte-scan-light/>.
- [22] N. Otsu. A Threshold Selection Method from Gray-level Histograms. *IEEE Transactions on Systems, Man and Cybernetics*, 9(1):62–66, 1979.
- [23] R. J. Randriamahefa, J. P. Cocquerez, C. Fluhr, F. Pépin, and S. Philipp. Printed music recognition. *Proceedings of the International Conference on Document Analysis and Recognition*, pages 898–901, 1993.
- [24] A. Rebelo, G. Capela, and J. S. Cardoso. Optical recognition of music symbols: A comparative study. *International Journal on Document Analysis and Recognition*, 13:19–31, 2010.
- [25] A. Rebelo, I. Fujinaga, F. Paszkiewicz, A. R. S. Marcal, C. Guedes, and J. S. Cardoso. Optical music recognition - state-of-the-art and open issues. *International Journal of Multimedia Information Retrieval*, 2012.
- [26] J. W. Roach and J. E. Tatem. Using domain knowledge in low-level visual processing to interpret handwritten music: an experiment. *Pattern Recognition*, 21(1):33–44, January 1988.
- [27] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach (3. internat. ed.)*. Pearson Education, 2010.
- [28] Capella Software. Cappella Scan. <http://www.capellasoftware.com/General/capellascan-overview.html>.