Ismir 2004 classification contest

<u>The framework and how it works</u>

The framework consists mainly in 3 actions you can do:

- extract all the descriptors for a given set of files
- train a model given the classes definitions
- evaluate your model and rank your system accordingly

This is accomplished by 3 scripts:

1) ExtractDescriptors.py classfile

    input:

        classfile : the path to a file containing the classes definitions; see format in annex A.

    This script will successively call your 'DescriptorExtractor' method on all files in the model.

2) TrainModel.py classfile

    input:

        classfile: the same file as given in 1)

    This script will just call your 'TrainModel' method with the model containing the feature files instead of the wavfiles as an argument; see format in annex A.

3) EvaluateModel.py modelfile querydirectory   or
   EvaluateSimilarityModel.py modelfile querydirectory

    input:

        modelfile: this is the file that should have been written by the user's program when asked to train a model. The name should be the name of the classes definition file with '.featureclasses.model' appended.

        querydirectory: the path to a directory containing a set of queries, each one being a file, which format is described in annex B.

    This script will successively call your 'EvaluateModel' (resp. 'EvaluateSimilarityModel') method on all queries in the directory and compute a ranking for the whole system.

<u>Installation - what you have to plug in to run your own tests</u>

Configuration

First of all, you should run the configure.py script that's in the root directory. Alternatively, once you get used to it, you can directly edit the 'config' file.

It will ask you which framework you prefer to use (among 'shell', 'matlab', 'python'). If you don't know what to choose, you may want to choose 'shell', as it allows you to call your program in the way you want. The others are here if you prefer tighter bindings with a language so you don't have to handle administrative tasks such as reading class files and such. (see the skeletons associated with each binding).

It will then ask you where your audio files are located. You should provide an absolute path to the root of where you put all of your audio files. All relative paths used in the class files are relative to this path.

It will also ask you about the extension of your features files. You should type the extension with the dot included, ie: '.features' for instance. Thus, you will be able to train different models using different features for each of your datasets.

Finally, it will ask you how many results should be returned in the similarity evaluation. 10 seems to be a good value, but if you want to try other values, you can do so by changing this variable.

You can then implement your algorithm. **The files you need to write will have to be located in the 'bin' directory for them to be called correctly**.

Implementation

There are mainly 3 tasks you have to do to run your own tests, which consist in writing the 3 programs that are going to be called by the scripts.

NB: depending on the language chosen, some minor details may vary (as the file extension), but the results expected are all the same, and thus the process will be described using the shell method (because it allows more flexibility, as you can call any of your programs to do the task)

1) DescriptorExtractor.sh wavfile
    input:
        wavfile: the path to a wavfile, mono, 22.05KHz
        outputfile: the path to the file you should write
    output:

your program should write a file which name is the one given as argument.

The format of the file is the one you will use, and is absolutely free.

2) TrainModel.sh featuresclassfile
   - input:
     - featuresclassfile: the path to a classes definition file, with the elements being the feature files. The format is the one described in annex A.
   - output:
     - your program should write a file that describes the classes models, and that is going to be used for the evaluation. Its name should be the name of the classfile with '.model' appended.

3) EvaluateModel.sh classmodel resultfile queryfeaturefile or EvaluateSimilarityModel.sh classmodel resultfile queryfeaturefiles
   - input:
     - classmodel: the path to the model you wrote with TrainModel.sh
     - resultfile: the path to the file in which you will have to write your results.
     - queryfeaturefile: the path(s) to one (or more) feature files that has (have) been computed from the wavfile(s) we wish to classify
   - output:
     - you should write the result in the given resultfile. Depending on the contest, its format will either be just the class it is in, or a similarity list (one item per line).

Annex A : format of the classes files

Classes are stored in a file, which format is pretty simple. It is composed of lines, each line representing a class.
On each line, we can find, separated by whitespaces, first the name of the class and then the elements constituent of that class. The elements are, depending on the situation, either the relative paths to the wavfiles, or the absolute paths to the features files. The two models are equivalent, they are just provided for convenience.
Usually, the second will have the name of the first with your extension for features files appended.

Example 1: the file with wavfiles (artists.classes)

```
artist1 album17/song1.wav album17/song2.wav album17/song3.wav
artist2 album23/song1.wav album23/song23.wav
...
```

Example 2: the same file with .features files (artists.classes.features)

```
artist1 /export/audio/album17/song1.wav.features /export/audio/...
artist2 /export/audio/album23/song1.wav.features /export/audio/album23/...
...
```

<u>**Annex B : format of the query files**</u>

A query is stored in a file. The file consists in 2 (or more) lines:
the first one is the name of the wavfile to query, then:

1) for artist and genre identification:
   the second line is the result that the EvaluateModel program should return
2) for artist similarity:
   the following lines are the results that the EvaluateSimilarityModel program should return

Example: in the case of artist identification, a query file could be composed of

```
peter_gabriel/vanilla_sky/04_-_solsbury_hill.wav
peter_gabriel
```

A set of queries is just a directory containing all the query files.

<u>**Annex C : simple scenario**</u>

Let's suppose you want to take part in the artist similarity contest.
First, configure your base audio path using the configure.py script or by editing the 'config' file by hand.
You then have to write the 3 scripts:
DescriptorExtractor.sh, that you are going to put (like all the others) in the 'bin' directory.
TrainModel.sh, and EvaluateSimilarityModel.sh

To evaluate it, you should do:

1) change directory to the base path of the framework.

2) If it isn't done already, run the ExtractDescriptors.py script. Let's use for instance the uspop1.artists file in data/classes:

   ```
   ./ExtractDescriptors.py data/classes/uspop1.artists
   ```

3) Train your model:

   ```
   ./TrainModel.py data/classes/uspop1.artists
   ```

4) Evaluate it, using the queries from the uspop2 artists set:

```
./EvaluateSimilarityModel.py
data/classes/uspop1.artists.featureclasses.model
data/queries/uspop2/
```

The results should appear on the console after a while, indicating your score according to different measures, which are:
The measures are similarity distances between two ordered lists. As there is no single answer to this problem, we provide here a few measures to choose from:

- Overlapping items : this is the number of common items between the lists.

- Top N rank agreement: this measure is taken from [uspop2002].

- MTG distance1: this is an experimental measure, which is the weighted sum of the distance from the original artist in the reference matrix. The weight function is an empirical monotonically decreasing function.
  Thus, the lower, the better.

- MTG distance2: this is an experimental measure as well. It is the weighted sum of the distance from the original artist in the reference matrix. The weight function is 1/x. It is then normalized by $\sum_{r=1}^{maxrank} \frac{c_r}{r}$, where maxdist stands for the worst ranking of the artists of the list, and c(r) is the cardinal of the set of all elements at distance r from the original artist. Range is [0,1].