Musify : Music detection system

Diogo Mendonça 93002
Leandro Silva 93446
Margarida Martins 93169

Index

1. Overview

In this work, our goals were to develop two programs, *max_freqs* and *musify*. The first one aimed to transform a signal into individual spectral components to provide frequency information about the signal. The second program is a music recognition system that guesses the music of an audio sample using the Normalized Compression Distance (NCD) for each music in the database.

The chosen programming language was Python for the *max_freqs* program, as it is simpler to code and fastness is not too much of a requirement here. On the fly, it is only used to transform a small audio sample passed as input, and the preparation of the samples in the database only have to be called once.

On the other hand, since the *musify* program has to perform a search in the database and find the best match for the input sample as quickly as possible, we have decided to use C++ here.

2. Program options

max_freqs*:* This program only requires an optional argument with the path for an audio sample or a directory of audio samples that will be transformed into a spectrum of frequencies. If none is provided, the program will convert all samples that reside in the */music_samples/* directory.

Thus, the optional arguments are as follows:

- -f: The path of a specific audio sample to be transformed.
- -d: The directory of audio samples to be transformed.

musify: This program is the core of our application. It requires a path for the audio sample under analysis, that is, the sample that we desire to identify.

The optional arguments are as follows:

- -c: The compression level used by the various software applications that perform the file compression. Its value must be between [0, 9]. The compressors used are gzip, bzip2, and lzma.

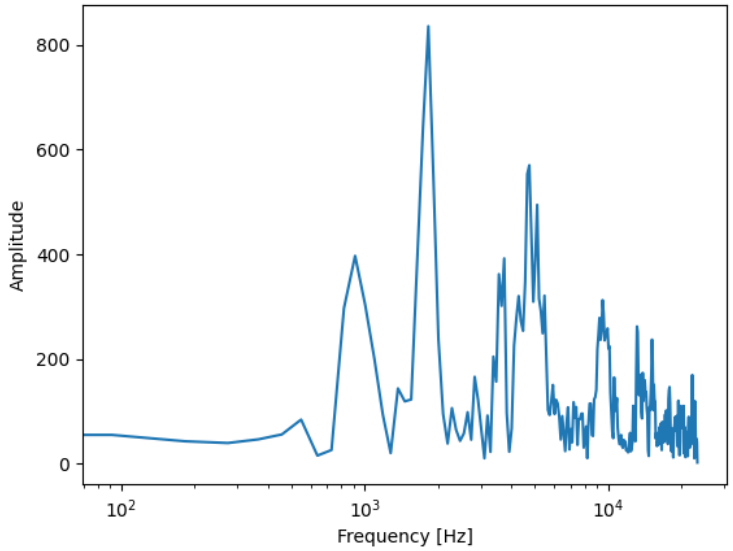
3. Functions

max_freqs

wav_signiture

Method *wav_signiture* receives a file path and its name as input. This method reads .wav files, transforms them to mono, and then writes a frequency signature of the test file by discovering the most prominent frequencies in partially overlapping segments of the track, using Fast Fourier Transforms (example **3.1**).

Example 3.1: Fast Fourier Transform of audio file



---

musify

gzip_compress

Method *gzip_compress* receives a file name and an integer representing the compression level as input. This method will compress the file passed as argument using gzip compression and return the size in bits of the compressed file.

lzma_compress

Method *lzma_compress* receives a file name and an integer representing the compression level as input. This method will compress the file passed as argument using lzma compression and return the size in bits of the compressed file.

bzip2_compress

Method *bzip2_compress* receives a file name and an integer representing the compression level as input. This method will compress the file passed as argument using bzip2 compression and return the size in bits of the compressed file.

compress_solo

This function calculates the size in bits when compressing a file using the three functions described above. These values are stored in an array passed as argument. The function also records the execution time for each different compressor.

compress_cat

The method *compress_cat* concatenates two files passed as argument and calls *compress_solo* in order to get the compression size of the resulting file.

normalized_compression_distance

This function receives a filename, a directory and a compression level as arguments. The directory is the music database and the filename is the path of the music to be detected.

The function will loop through every music in the database, calculating the Normalized Compression Distance between the music to be detected and the database music using the *compress_solo* and *compress_cat* functions.

For each different compressor, the smallest NCD is recorded.

4. Results

In order to verify if the *musify* program is working correctly, it is necessary to calculate the accuracy of its predictions, for each of the used compression methods. For this, samples already present in the database were fed to the program, and it was checked if the program returned the same file name, for each compression type.

Example 4.1: *Musify* accuracy (no noise, compression level 9):
gzip accuracy - 100%
lzma accuracy - 100%
bzip accuracy - 100%

Another test was performed to see the effect of the addition of noise to test samples on *musify*. As we can see in example **4.2**, the accuracy values dropped significantly when compared to the values obtained without noise, with gzip presenting the best results out of the 3 compression methods.

Next, different compression levels were experimented with the same noisy samples, as can be seen in examples **4.3**. These noisy samples were built by adding an audio with background noise to the samples. The background noise is the typical sounds heard in a cafeteria. This way, we can simulate a more realistic usability scenario. *Musify* is still able to identify most of the songs, with gzip being the most inaccurate compressor.

Using the program, white noise was added to the samples, however this change resulted in small or no changes in the results obtained previously.

Example 4.2: *Musify* accuracy (with noise, compression level 9):
gzip accuracy - 60%
lzma accuracy - 64%
bzip2 accuracy - 64%

Example 4.3: *Musify* accuracy (with noise, compression level 1):
gzip accuracy - 56%
lzma accuracy - 64%
bzip2 accuracy - 64%

For the previous results, all sample frequency signatures were obtained using a window size of 1024 in the program *maxfreqs.py*, which might not be the optimal value for the latter use in *musify*. As we can see in table **4.5**, the highest accuracy value is for gzip compression, with a window size of 2048. This is good, as using gzip is approximately 2 times quicker than bzip2, and 4 times quicker than lzma.

Table 4.5: Window size influence on model accuracy:

Window Size	gzip accuracy	lzma accuracy	bzip2 accuracy
1024	60%	60%	62%
2048	68%	64%	64%
4096	56%	56%	60%

Table 4.6: Average compression time:

	gzip	lzma	bzip2
Average Time (ms)	66	218	108

5. Recognizing Animal Sounds From Different Sources

Aside from the main objective of identifying a sound based on a copy sample, that might or not be mixed with some noise, we decided to try to identify an animal sound from different sources.

We prepared a database with samples from eight different animals: a cat, a cow, an elephant, a goat, a horse, a sheep, a dog, and a wolf. Then, we verified what was the best match for other samples not presented in the database, but from the same animals.

The results were very poor. We attempted to test with others six samples from a cow, a wolf and a cat, which we thought to be the most distinguishable, but only the wolf was correctly guessed.

test sample	gziz best sample	lzma best sample	bzip2 best sample
cow_test1	wolf	wolf	cow
cow_test2	wolf	wolf	wolf
wolf_test1	wolf	wolf	wolf
wolf_test2	wolf	wolf	wolf
cat_test1	sheep	cow	goat
cat_test2	sheep	wolf	goat

This is because the trained animal sound that should match with the test animal sound has a different tone. We, as humans, are able to recognize it because they have the same timbre. If the tone varies, the frequency is not the same, and thus the application fails to recognize it. The reasons for having so much wolf guesses might be for it having a very clear and almost unique frequency, which is able to make its compression more compact.

To remind that this experiment is not a problem in our product, since its goal is to recognize music samples with a sub-sample from it, but it would be a nice feature to have. To accomplish it, we would probably have to resort to machine learning techniques.

6. Conclusion

With this work, and with the experience obtained from past works, we can conclude that using compression techniques to identify similar contents is very efficient and has proved to achieve good results. It is incredible how it can be applied everywhere, in different contexts and with different types of data.

Musify takes advantage of this to identify music samples, but also extracts the maximum frequencies from the audio signals to remove the noise, that is the not so important features from the signal, so that it can have a higher efficacy. Without this extraction, we think that the results would worsen significantly.