University of Aveiro
Computer Engineering Masters
Algorithmic Theory of Information
Armando J. Pinho
Diogo Pratas

Diogo Almeida, Rodrigo Ferreira

January 21, 2021

# 1 Introduction

The third and final assignment asked us to build a compression based *shazam*-like application, at best capable of guessing the corresponding song, and at least finding the most similar songs to a certain small target audio file (*.wav*) that is used to query the database.

The database is made up of $.freqs$ files, each representing a song. These files were created using the tool made available by the teachers, $GetMaxFreqs$.

For this to happen we had to learn about some theory first, which we will briefly explain.

# 2 Comparing Files

Some technique has to be used to compare the target audio's $.freqs$ file with all those in the database.

## 2.1 Normalized Information Distance (NID)

In an ideal world, this comparison would be achieved by calculating the Normalized Information Distances (NID) for every (target-database entry) combination.

Because NID is the *holygraal* of string similarity, and because with computers anything can be represented as a string (of 0's and 1's), this can *intheory* be used to compare anything from pictures, to audio files, documents, programs, etc...

$$NID(x,y) = \frac{max\{K(x|y), K(y|x)\}}{max\{K(x), K(y)\}}$$

Where $K(n)$ represents the Kolmogorov complexity of $n$.

If we look only at the numerator, we can see the formula for absolute information distance, the denominator is what makes it express similarity.

Sadly, as previously mentioned, this is only *intheory*, because the NID isn't actually computable due to the $K(n)$'s present in its formula.

## 2.2 Kolmogorov Complexity

As we said, the function $K(n)$ or the Kolmogorov complexity is what makes the NID not computable.

To put it simply, the Kolmogorov complexity of an object is the length of the shortest computer program that produces it as output (Turing Machines are used more formally, though pseudocode is enough to understand the principle).

### 2.3 Normalized Compression Distance

While finding the exact $K(n)$ is not computable, we can still use the NID formula and approximate $K(n)$ with the use of compressors.

$$NCD_C(x,y) = \frac{C(xy) - min\{C(x), C(y)\}}{max\{C(x), C(y)\}}$$

Where $C(n)$ represents the length in bits of the compressed $n$ file.

This way we can perform string similarity with decent accuracy, though results vary depending on the compressor used.

## 3 The Code and Structure

We decided to divide the code in various files and directories.
More info on how to run the code is present on the readme file.

### 3.1 Directory Structure

In the assignment's main directory we can find several other directories:

- /songs/: this folder contains one folder per music genre, and these genre folders are where the user should add their audio files (*.wav*) to build or add to the database (in case of doubt regarding the musical genre, there is a folder named "other").
- /freqs/: this folder contains one folder per music genre, similar to /songs/, but instead of these containing audio files, they contain the corresponding audio signature files (*.freqs*) created using *GetMaxFreqs*
- /target/: this folder is where the user should add the audio files (also *.wav*) that they want to be guessed by the program.
- /src/: contains the code used which will be discussed later
- /GetMaxFreqs/: contains the program made available by the teachers to get the audio signatures (*.freqs*)
- /report/: containing the assignment's report which you are currently reading.

Inspired by the musical study linked by the teachers, we decided to divide /songs/ and /freqs/ into genres to check when guessing the song, which genre does it attribute to the sample. It's also interesting in the case that the target song isn't in the database, can it still guess the genre correctly?

It's also important to note that throughout the whole program, all audio files must be *.wav*, as we had incompatibility issues with other formats like *.mp*3.

## 3.2 Building the database

There's a python file *builder.py* completely dedicated to building the database. It has some default parameters (given to $GetMaxFreqs$) but the user can supply their own as arguments.

The program will check every genre folder in the /songs/ folder, check that the sample rate is supported by $GetMaxFreqs$ (44100), convert it if necessary and create the corresponding audio signature file in the respective genre folder in /freqs/.

## 3.3 Shazam

There's also a python file completely dedicated to guessing the target audio provided, *shazam.py*.

This program receives the same parameters as *builder.py* as well as a path to the target audio file, and possibly an experimental parameter which will be discussed later.

The program then checks if the target audio's sample rate is supported by $GetMaxFreqs$, converts it if necessary and creates its $.freqs$ file, which will then be compared using NCD to all $.freqs$ files in the database, then the top guesses will be displayed, and calculating the average result per folder, we can also display the genre similarity for each genre.

The experimental parameter concatenates the audio file to itself until it has reached the database's average duration, and only then creates the corresponding $.freqs$ file. The results of this experimental parameter will be discussed in the Results section.

### 3.3.1 Audio

Besides those more important python files, there is also *audio.py*.
This module is mainly used for audio file manipulation, operations like:

- trimming audio: cutting a sample from an audio file to use as target
- concatenating audio: concatenating various audio files.
- adding noise: adding whitenoise to an audio file
- converting sample rate: converting an audio file's sample rate to 44100 to be compatible with $GetMaxFreqs$

### 3.3.2 Structuring

Finally, there is also a file, *structure_songs.py* which should be executed first of all the programs, as it builds the necessary structure for the user to add his own songs to the database.

# 4 Results

It's tricky to display results for such an application, first, because there are a lot of parameters (and thus, possible combinations), and though we did our fair share of testing, we surely missed some things.

## 4.1 GetMaxFreqs Parameters

First off, it should be noted that our results were always better when using the same $GetMaxFreqs$ parameters to build and to query the database, so for the sake of time, we're not even considering any other approach in this matter.

First lets clarify what each $GetMaxFreqs$ parameter means:

- WS: size of each window to be evaluated;
- SH: jump/shift from each window to the next;
- NF: number of signature frequencies to be collected from each window;
- DS: downsampling to be applied to the file.

### 4.1.1 Accuracy

We settled on a few combinations of parameters which seemed to work for us, namely:

- WS: 4096;
- SH: 128;
- NF: 4;
- DS: 4.

It should be noted that this combination was chosen because it was the best at guessing the target correctly, when it belonged to the database. Other combinations (mostly with similar parameters but smaller NF) worked better at finding similar songs when the target wasn't present in the database, but as that wasn't the main objective of this work, we discarded such possibilities.

### 4.1.2 Time

As most of out time running code was spent building and rebuilding the database with different parameters we could more or less see how each affected the execution time.

- WS: didn't have the biggest impact but on average larger windows made the program run for longer;
- SH: One of the most influential regarding time, small values made the program run for much longer, as more windows had to be analyzed.

- NF: didn't seem to have much of an impact.
- DS: larger values significantly decrease running time and vice versa.

DS and SH were the main offenders when it came to the time we had to wait building the database.

### 4.2 Target Files

Target files also had a major impact on the application's accuracy, on average, the bigger the target file, the more accurate were the applications guesses, and vice-versa.

The part of the song the audio file represents is also important, it should be as specific to the song as possible for better results.

For instance, when querying a guitar section of a generic pop song, despite the number 1 guess being correct in this case, the other closest guesses were all of songs in the rock or metal genre, which obviously have their fair share of guitar solos.

### 4.3 Noise

We also tested different queries with different volumes of noise added to them.

| *compressor/noiselevel* | 0 | 0.05 | 0.1 | 0.15 | 0.2 | 0.5 | 0.75 |
|---|---|---|---|---|---|---|---|
| lzma | 1:0.91 | 1:0.91 | 1:0.92 | 1:0.93 | 1:0.92 | 1:0.93 | 1:0.94 |
| gzip | 1:0.96 | 1:0.96 | 1:0.96 | 1:0.96 | 1:0.96 | 1:0.97 | 1:0.97 |
| bz2 | 1:0.90 | 1:0.90 | 1:0.91 | 1:0.91 | 1:0.91 | 1:0.91 | 1:0.92 |
| zlib | 1:0.96 | 1:0.96 | 1:0.96 | 1:0.96 | 1:0.97 | 1:0.97 | 1:0.98 |
| lzma_legacy | 1:0.91 | 1:0.92 | 1:0.92 | 1:0.93 | 1:0.93 | 1:0.94 | 1:0.95 |

Table 1: rank and rounded NCD value of the $Drake - nonstop$ song guessed by each compressor with different noise levels for a $Drake - nonstop$ 15 second sample

Pretty impressive performance all around, considering that with 0.75 volume the song is barely audible.

All the modified song files were created using our *audio.py* with the "-noise" parameter, more information on the readme, some might also be present in the target/ folder for testing.

## 4.4 Compressors

The final results are regarding the compressors, we used 5 different compressors in our experiments:

- gzip;
- lzma;
- lzma legacy version;
- bz2;
- zlib.

### 4.4.1 Accuracy

Interestingly, different combinations of $GetMaxFreqs$ parameters worked better for different compressors.

The main difference being with the lzma's, while the others worked well with a reduced NF, the lzma's had a significant improvement in performance with higher NF.

The /freqs/ database delivered was built using the parameters that gave us the best results with lzma as the best parameters found for this one, worked decently for all the others, while the inverse didn't happen.

In regards to ranking the compressors, purely in terms of accuracy, for the current combination of parameters, the lzma's and bz2 would be a tier above, zlib and gzip would be very close to each other but slightly worse than the other 2.

Regarding songs not in the database, for the current combination of parameters does give weird results on occasion, but that is discussed later.

In general, for songs present in the database, with a decent sample target (+-15 seconds,representative), all compressors guessed correctly most of the time.

### 4.4.2 Time

Regarding time, across all tests gzip, bz2 and zlib always had similar execution times.

What stood out was lzma, it was always the compressor that took the longest.

Though none of the compressors took more than a few seconds to do their job so it wasn't a problem whatsoever.

# 5 Observations

## 5.1 Difficulties and Solutions

We had some issues in the beginning with the scores we were getting because we were using songs with significantly different lengths in our database, it was making it so that songs with around 1 minute length were always the top guesses for the target regardless of its source.

We judged this was because of the difference in length, so we decided to make a more well rounded database trying to average around 3:30 or 4:00 minutes per song.

We also tried to keep the number of songs per genre balanced, to get a better guess at the target's genre.

## 5.2 Observations

One of the more immediate and most shocking observations we made was that we expected the values of the correct guesses to be way lower. In theory, when the files are similar the NCD tends to 0, but in our case, even with the correct target and song combination, we still got values around 0.90 for the correct guess (though the others were always hovering around 1.00 so the guesses were still correct).

We also judged this to be an influence of the difference in length between target and source song, because when querying a full song present in the database as a target, the results were as expected very close to 0.

Another observation was that the genre guess wasn't always very accurate, this can be attributed to several factors:

- Bad choice of songs for each genre.
- Small target sample.
- Not very representative target.

We say bad choice of songs for each genre, because we weren't very strict with our distribution of songs, and thus, many subgenres of metal can be inside metal folder, and likewise for the other genres.

Small target sample is an explanation when the target sample is very small ($< 10$ seconds).

Not representative target refers to the part of the song selected to use as a target, the more generic the better the genre guess, but very unique parts can lead to answers all over the place.

Regarding our experiment of self concatenating a small audio sample to match the length of an average song in the database, the results on average showed a slight improvement. All compressors benefited slightly (around 0.01) from this experiment except for zlib that got slightly worse results.

# 6 Conclusions

When looking at the problem of audio recognition from an outside perspective, it can be daunting, where to start, how to go about it? But as seems to be the norm, compression can be used to address and simplify the issue.

We think it's fair to say that compression is a very versatile area, along this course we had several different assignments regarding it, and they all had very interesting applications that addressed real world problems, so we are leaving much more educated as well as interested in the field that is compression.

# 7 Notes

All songs were downloaded directly from youtube playlists using youtube-dl, and the titles of the corresponding videos were used to name the audio files (explaining some weird names). The playlists used were:

- Jazz:

  `https://www.youtube.com/playlist?list=PL9QZLUOS8sToh4ML-s2sCxyBTdx8olbnx`

- Rock:

  `https://www.youtube.com/playlist?list=PL9QZLUOS8sTrlOiSoNLcisouIt-rzg6lK`

- Metal:

  `https://www.youtube.com/playlist?list=PL9QZLUOS8sTqWTI1yBjxEU-Mj-DokPqX2`

- Pop:

  `https://www.youtube.com/playlist?list=PL9QZLUOS8sTqmQmpHHf1s3WZrQ76-C8lj`

- Classical:

  `https://www.youtube.com/playlist?list=PL9QZLUOS8sTpRMM9sn0WHee0Otc_ByzNX`

- EDM:

  `https://www.youtube.com/playlist?list=PL9QZLUOS8sTqKwmKnvqHHhAOSwleiqYNF`

- Hip hop:

  `https://www.youtube.com/playlist?list=PL9QZLUOS8sTpAM3r4MH4s7UOT0AagU7Yj`