Universidade de Aveiro

School Year 2021/2022

TAI - Practical Assignment 3

Diogo Bento 93391

Pedro Laranjinha 93179

Leandro Kiame 78177

Professors Armando J. Pinho & Diogo Pratas

Introduction

For this project it was requested that we develop programs that utilize **signature calculation** and **compression algorithms** to gauge the similarity of audio tracks.

Code Steps

compile.py

This program is used to calculate the signatures of all files in a folder using a script provided to us by our teachers and output the signatures into another folder.

We do this to avoid computing all the signatures in our database every time they are used.

This program has 2 arguments:

- **--source:** folder to read files from
- **--dest:** folder to write files to

From inside the /src/ folder, one can run the code by using *python3* as follows:

```
python3 compile.py --source  <source> --dest <destination>
```

This program is intended for use with WAV and FLAC formats and outputs files with the .freqs extension.

finder.py

With this program the user can submit a WAV or FLAC file in order to compare their signature to those present in a pre-compiled database, and obtain the 10 most similar files.

Our metric of similarity is the Normalized Compression Distance, which is as follows:

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

With C denoting the length of the outcome of applying a given compression algorithm on a file and x,y being the files under scrutiny.

Since this is a measure of distance rather than similarity we take low NCD to mean that 2 given files are similar.

This program has 3 parameters:

- **--sample**: file to be compared to database
- **--db**: folder with compiled signatures

- **--gzip/--lzma/--bzip2**: sets compression algorithm used, default is **gzip**

From inside the /src/ folder, one can run the code by using *python3* as follows:

```
python3 finder.py --sample  <source> --db <signature folder>
```

finderUI.py

This program is a variant of *finder.py* that allows a user to provide multiple files in sequential order via a graphical user interface and functions in a very similar way.

Due to requiring the PyQT5 framework some additional installs will be required to run this file, the preparation commands are as follows:

```
sudo apt-get install PyQT5
sudo apt-get install PyQT5-tools
```

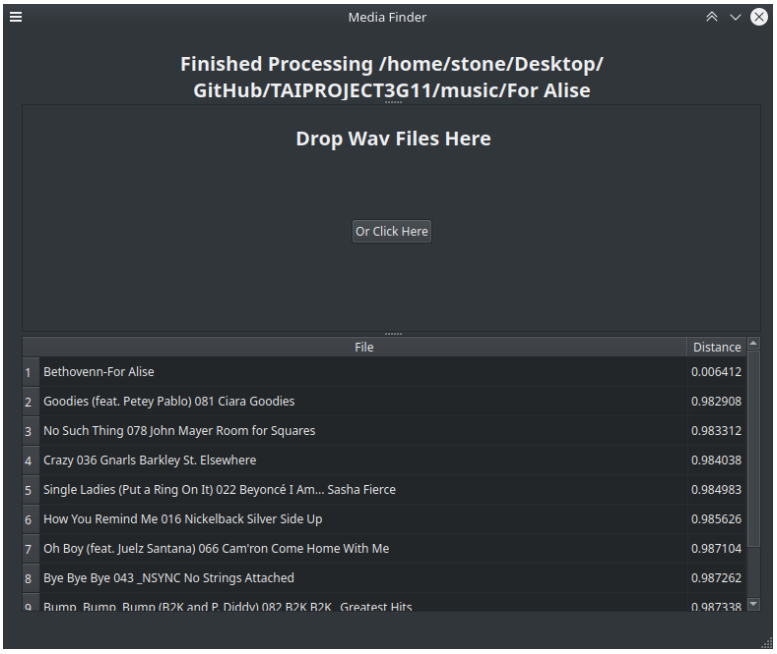

Fig. 1: Screenshot of the *finderUI.py* window

This program has 3 parameters:

- **--db**: folder with compiled signatures
- **--cache/no-cache**: whether to store signatures in memory, default is **False**

From inside the /src/ folder, one can run the code by using *python3* as follows:

```
python3 finderUI.py --db <signature folder>
```

makesamples.py

makesamples.py was created to generate test samples to compare against a given database. Given a folder with WAV files and a target folder it will output a given number of *noisified* clips.

The noise is generated by adding a zero-mean normal distribution array to the provided signal.

This program has 2 arguments:

- **--source:** folder to read files from
- **--dest:** folder to write files to
- **--min-length**: minimum sample length in seconds, default is **5.0**
- **--max-length**: maximum sample length in seconds, default is **15.0**
- **--samples-per-track**: samples to extract per track, default is **10**
- **--noise:** multiplicative factor applied to the noise signal, if listening is intended very small values are recommended, default is **0.01**

From inside the /src/ folder, one can run the code by using *python3* as follows:

```
python3 makesamples.py --source  <source> --dest <destination>
```

Test Dataset

To create the dataset we got 26 random popular songs from the 2000's and 4 well known songs from the classical genre.

We used the 4 songs from the classical genre to test if our programs could find similar songs and not just the correct one, as while it is difficult to say how similar pop songs are, we know that classical music tends to be very different from pop songs.

Results

In this section we see how results change based on these 3 arguments:

- Size of the sample
- Amount of noise added to the sample
- Compressor used in the distance calculation

To not bias the results towards the start, middle, or end of the songs, 10 different samples from random parts of each song were used per sample size and noise.

First, we confirmed that with a lower noise and a bigger sample, the song found tends to be correct more often.

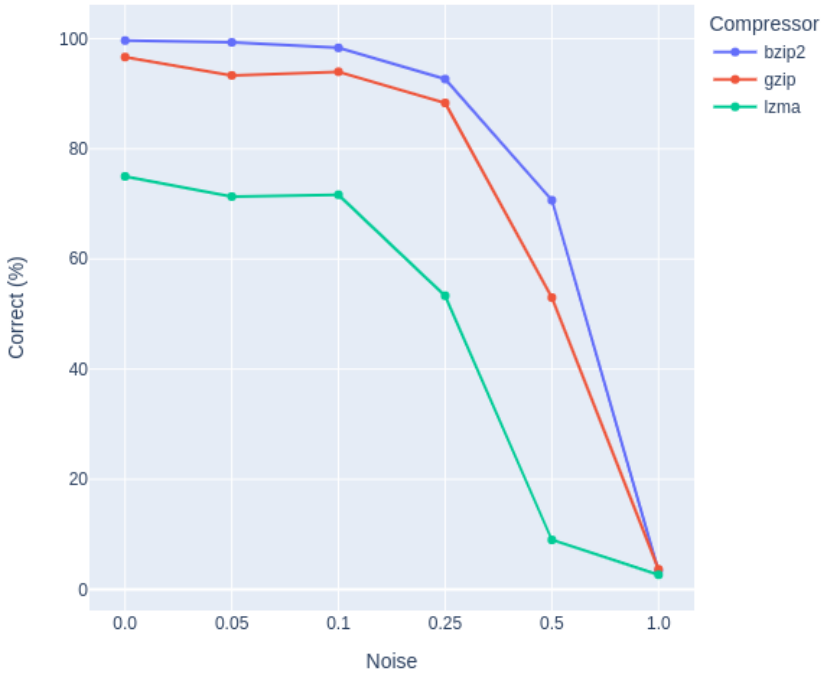

Fig. 2: Percentage of correctly found songs with sample size=5s

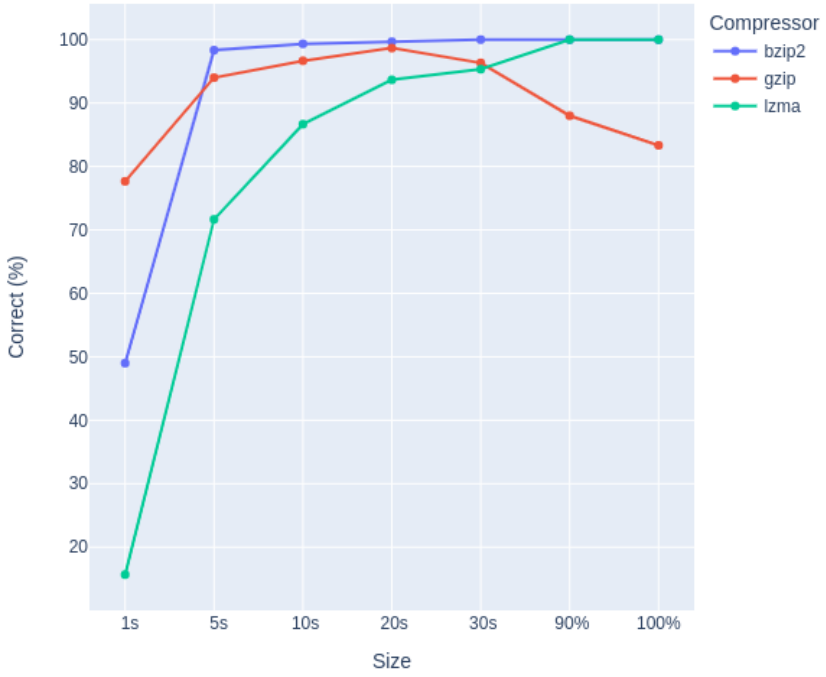

Fig. 3: Percentage of correctly found songs with noise=0.1

We also found out that the **bzip2** compressor is usually the most correct, followed by **gzip** and lastly **lzma**. This order only changes either when the sample size is very small, where **gzip** becomes the most correct, or when the sample size is close or equal to the size of the song, where **gzip** becomes the least correct.

Fig. 4: Percentage of correctly found songs

Lastly, we checked how many classical songs were in the top 4 whenever one of the 4 classical songs was searched, to see how good our program is at finding similar songs.

Like in the last analysis, a lower noise and a bigger sample usually means more songs in the top 4, with the exception of samples close to or equal to the size of the song, where **gzip** and **bzip2** get substantially lower results.

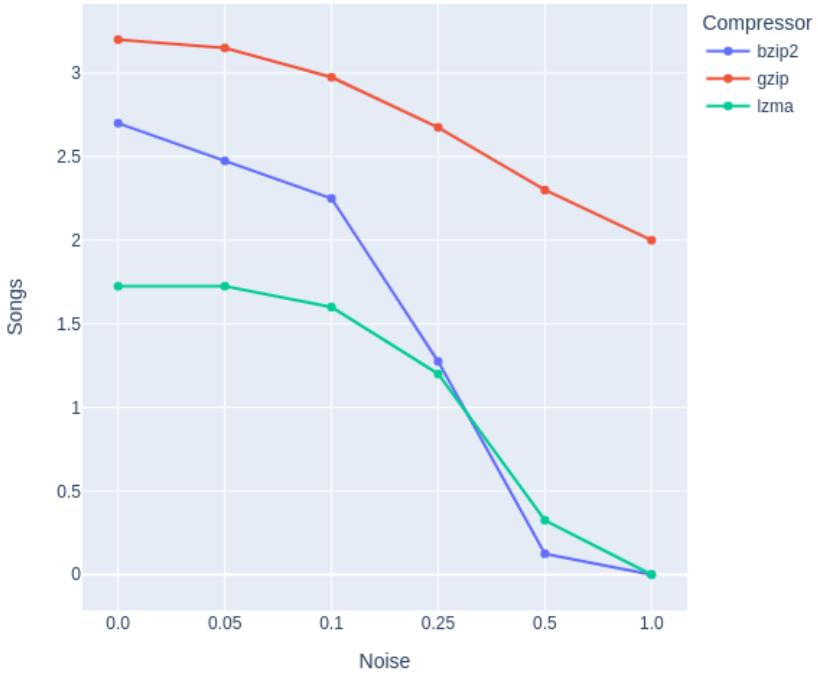

Fig. 5: Average number of classical songs found in the top 4 with sample size=30s

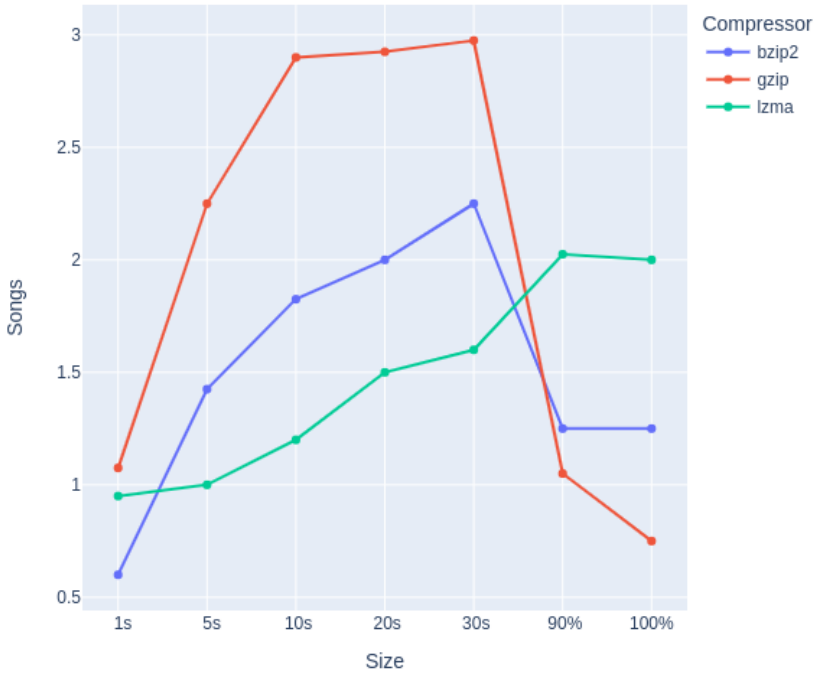

Fig. 6: Average number of classical songs found in the top 4 with noise=0.1

Even so, the **gzip** compressor tends to find the most songs in the top 4, followed by **bzip2** and then by **lzma**. Another exception is when the sample size is very small, where **lzma** can outperform the other 2.

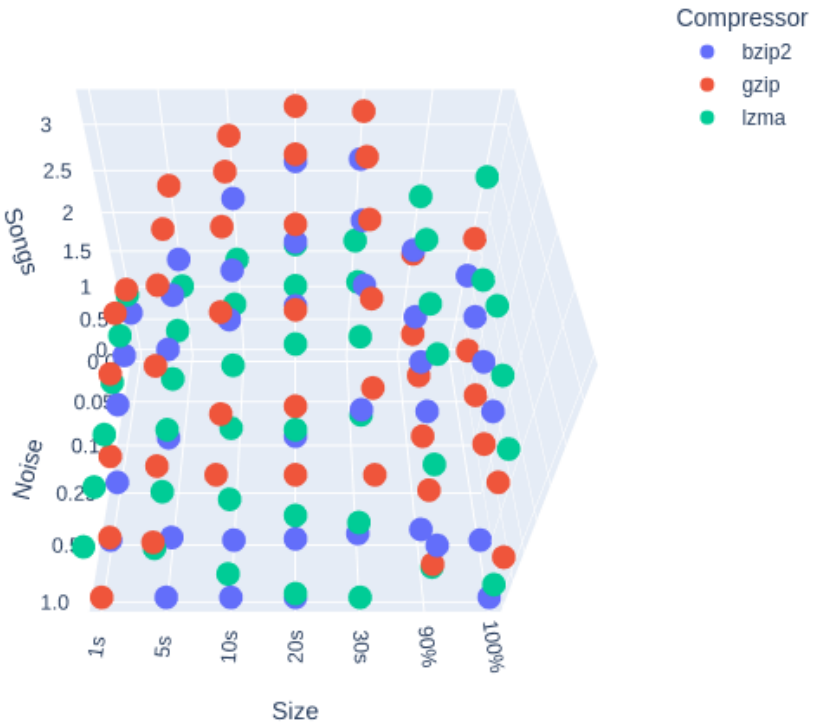

Fig. 7: Average number of classical songs found in the top 4

All the graphs used in this report and their interactive versions can be found in the folder **/report/plots**.
To help with the result and graph generation we made 2 programs:

- **datagen.py:** generates the data needed to create the graphs
- **plot.py:** uses the data generated by **datagen** to create graphs for analysis