Music recognition using compression

Lab Work 3

Gonçalo Machado | 98359

Introduction

Algorithmic Information Theory explores how we measure and compare the complexity and similarity of data. One useful tool in this field is the Normalized Compression Distance (NCD), which estimates how similar two pieces of data are by using data compression methods. This approach helps in various applications, such as data clustering and pattern recognition, by providing a way to compare data in a quantifiable manner.

This report focuses on using NCD to identify music tracks from a database using short audio samples. The idea is to compress the query samples and the database tracks using standard compression tools like gzip, bzip2, etc.. By calculating the NCD between the query and each track, we can find the track that is most similar to the query. This demonstrates how compression-based techniques can be effectively used for tasks like music identification.

In our study, we test the method on a database of 79 different music tracks, using small segments of these tracks for identification. We also add noise to some of the query segments to see how well the method works under challenging conditions. By converting the audio files into frequency-based signatures before compression, we aim to improve the accuracy of the NCD calculations. This research highlights the practical uses of Algorithmic Information Theory in real-world applications like music recognition.

Methodology

In this assignment we were asked to develop and test a setup using NCD for the automatic identification of music, using small samples to query the database. NCD is an approximation, through the use of compression, of the Normalized Information Distance (NID), which is defined as

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

where K(x) is the Kolmogorov complexity of string x and K(x|y) is the Kolmogorov complexity of x when y is given as additional information.

Since the Kolmogorov complexity is non-computable, the NCD uses

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

where C(x) is the number of bits required by compressor C to represent x and C(x, y) is the number of bits needed to compress x and y together (usually, the two strings are concatenated). Distances close to one indicate dissimilarity, while distances near zero indicate similarity.

In order to be able to use NCD, we will need to convert the .wav files into a representation of their most significant frequencies. This was done by using the GetMaxFreqs program made available by the professors of this course.

All code for this project was written in bash except the parsing of results and graph generating, which were done in Python, and the GetMaxFreqs which was made available by the professors. In order to better develop and test the necessary code, the several tasks required were split into several bash scripts.

In order to increase speed, the program that calculates the NCD of a test sample requires the frequency files and bits necessary for compression of the frequency files to already exist. The only part that is generated is the bits necessary to compress the concatenation of the sample and the songs in the database.

fresh_start.sh

This script's purpose is to delete all files that might have been created from an earlier experiment and are not supposed to be in the directories. These files include:

- Standardized Songs
- Test Samples
- Compressed files
- Json files
- Frequency files
- Result files

The only files that will remain in the directories after running these scripts are the bash scripts, the GetMaxFreqs files and the original songs, located in the “OriginalSongs/” directory.

original_to_standard.sh

This script is used to transform the original songs into a .wav that can be processed by the GetMaxFreqs program (which only allows .wav sampled at 44100Hz). The standard songs will be located in the “StandardSongs/” directory.

get_test_samples.sh

This script, as the name suggests, is used to retrieve the test samples that will be used to obtain results. It takes 30 random songs and chooses a random timestamp as the beginning of the test samples, and retrieves test samples with 10, 20 and 40 seconds of duration. To note that the timestamp chosen is always 40 or more seconds from the end of the song. The name of the resulting files is the name of the original song plus the interval of the sample.

Ex: Touchy_Feely_Fool-Interval-163_173.wav

The samples were saved in the “TestSamples/Original/” directory.

get_noise_samples.sh

This script takes each of the original samples and creates versions of them with white noise, by using sox **synth whitenoise** effect. Together with this effect the **vol** effect was also used to create versions with louder or quieter white noise. The values used were [0.02, 0.04, 0.08, 0.16, 0.32, 0.64]. The name of the resulting files is the name of the original sample plus the volume of white noise it contains.

Ex: Touchy_Feely_Fool-Interval-163_173_Noise_0.32.wav

The resulting samples were saved in the “TestSamples/Noise/”.

create_database_dataset.sh

This script's purpose is to generate the frequency files of every song that will be used in the database (located in the “StandardSongs/” directory), compress every frequency file in each of the compression methods available and create a json file with the bits needed to compress every song in every compression method available. All files generated by this script are placed in the “database/” directory.

create_database_test.sh

This script does the same as the previous one, but uses the samples in the “TestSamples/” directory. All files generated by this script are placed in the “Samples/” directory.

offbrand_shazam.sh

This script is the one that is used to identify the song a sample is from using the NCD.

It takes 3 input parameters:

- f - The frequency file of the sample.
- c - The compression method we want to use.
- n - The number of results to present.

This scripts works by:

- Going through every song in the database:
    - Retrieving the bits necessary to compress its frequency file in the chosen compression method.
    - Retrieving the bits necessary to compress the sample's frequency file in the chosen compression method.
    - Concatenating both frequency files into a single frequency file.
    - Compressing this frequency file in the chosen compression method.
    - Calculating the bits that the compressed file needs.
    - Calculating the NCD between the database song and the sample
    - Saving it to an array
- Printing the n smaller results.

get_results.sh

This script is used to obtain the results for every combination of test samples and compression methods and append them to the file "results/results.txt". It uses the *offbrand_shazam.sh* script with the input parameter n with the value 10 to obtain the top 10 songs for each test sample.

parse_results.py

This script is used to parse the “results/results.txt” file into a JSON file that can later be used to generate the graphs and other result analysis. This file consists of an array of JSON objects, with the following structure:

```
{
    "File": "Inertia-Interval-11_21_Noise_0.32.txt",                    ##Name of test sample
    "Noise": 0.32,                                                      ##Volume of whitenoise
    "Duration": 10,                                                     ##Duration of sample (s)
    "BaseFile": "Inertia",                                              ##Original Song
    "Compression": "lzma",                                              ##Compression method
    "Top": {                                                            ##Top 10 results
      "Inertia": 0.9751857035202928,                                    ##Song and NCD value
      "AJR_-_Birthday_Party_(Official_Audio)": 0.9911755817467512,
      "AJR_-_BANG!_(Official_Video)": 0.991313386294454,
      "AJR_-_Adventure_Is_Out_There_(Official_Audio)": 0.9913798567535019,
      "AJR_-_Growing_Old_On_Bleecker_Street_(Official_Audio)": 0.9918896481256785,
      "AJR_-_Dear_Winter_(Official_Video)": 0.991957311886165,
      "AJR_-_Woody_Allen_(Official_Audio)": 0.9920832242868359,
      "AJR_-_Yes_I’m_A_Mess_(Official_Visualizer)": 0.9922881102126385,
      "AJR_-_Big_Idea_(Official_Audio)": 0.9924664264657713,
      "Steve_Aoki_-_Pretender_feat._Lil_Yachty_&_AJR_[Ultra_Music]": 0.9925499492041991
    }
    “CorrectGuessPlace”: 1                                              ##Place of correct guess
  }
```

To note that if the top 10 does not contain the correct song, the value of “CorrectGuessPlace” is 11.
The resulting JSON file is “results/parsed_results.json”.

graphs.py

This script is used to generate every graph present in the “Result and Analysis” section of this report, as well as any value that will be mentioned in that section. All resulting files will be placed in the “results/” directory.

Results and Analysis

This section will have the results regarding the project and their analysis.

The songs used for the database were all the songs, in .wav format, released by the band AJR until the date of the delivery of this report. There are 79 songs in total, with durations from 2 minutes and 30 seconds to 5 minutes. They were obtained by using online tools to convert youtube videos to .wav files.

The results for this project were obtained by using test samples from 30 different songs with different white noise volume and duration, and calculating the NCD with different compression methods.

The values for each variable were the following:

- Duration: 10, 20 and 40 seconds.
- Compression method: zip, gzip, bzip2, lzma and zstd.
- White noise volume: 0, 0.02, 0.04, 0.08, 0.16, 0.32, 0.64

For results, we first wanted to see how accurate the program was. This was done by seeing if the correct song would be the song with the lowest NCD, in the top 3 lowest, top 5 or top 10, and then calculating the accuracy which is displayed in Table 1.

Top	Number of correct guesses	Accuracy (%)
1	2923	93.11
3	3004	95.70
5	3037	96.75
10	3063	97.58
Total	3139	100

Table 1 - Accuracy of program

As we can see in Table 1, the program correctly identifies the song in 93.11% of test samples, and in only 2.42% the correct song was not in the top 10 songs with the lowest NCD value. This is a good indicator of the use of NCD as a form of identifying music.

In the next graphs and analysis, we will focus only on if the correct song is the song with the lowest NCD.

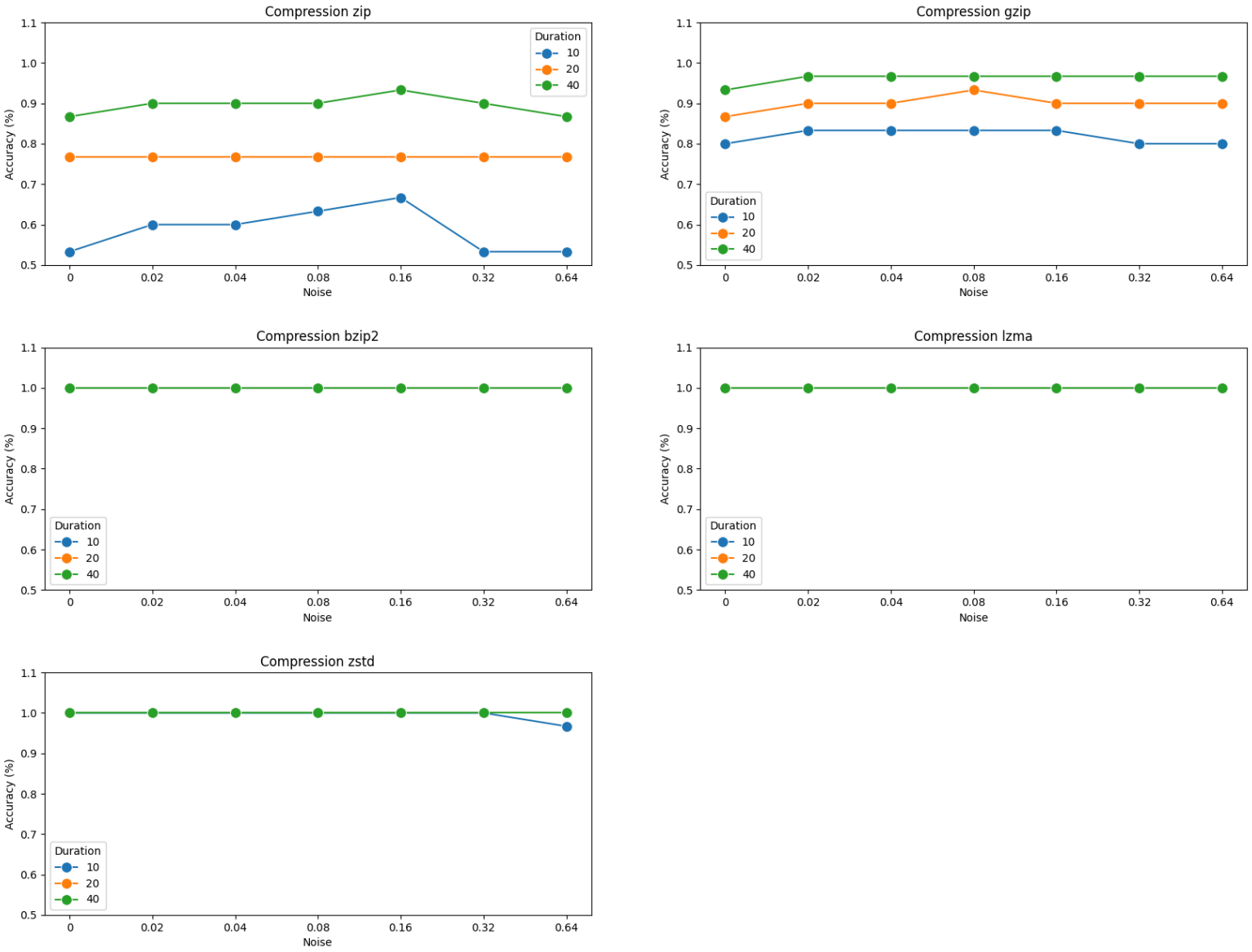

Figure 1 - Graphs with variation of duration and white noise volume for the same compression method

From looking at Figure 1, we can conclude that the **bzip2**, **lzma** and **zstd** compression methods achieved 100% accuracy on almost all samples tested (the only result lower than 100% was for **zstd** on samples with 10 seconds of duration and with a volume of 0.64, which had a 96.7% accuracy), while the **zip** and **gzip** compression methods did not reach 100% accuracy on any combination of duration and noise volume. A possible explanation for this discrepancy between the two sets of algorithms is the fact that **zip** and **gzip** both use the DEFLATE algorithm to compress files, while the other three use more complex algorithms.

We can also see, on the zip and gzip compression methods, that the bigger the duration the more accurate the program becomes, which is to be expected since the algorithm that both most commonly use, DEFLATE, uses, among other methods, replacement of repeated sections to compress files so the more sections files have in common, the better they are compressed, and the lower the NCD is between files.

A more unexpected result obtained was that the increase of white noise did not affect the accuracy in most cases, and in some it even increased the accuracy when comparing to the same conditions but with less volume. This might be due to the low amount of tests and samples used. This result is more visible in the results obtained with the **gzip** compression method in samples with 10 seconds, which might also be because the white noise creates more repeated sections within the sample, thus increasing the compression ratio slightly.

By analyzing Figures 2 and 3 we can confirm what we concluded by analyzing Figure 1, but also make more relations. For instance, we can see in Figure 2 that the more duration the samples have, the less spread the results are, which means that longer samples are less affected by noise and compression methods. We can also see that for samples with both 10 and 20 seconds, a white noise volume of 0.16 seems to help with accuracy.

The data used to generate this graphs, as well as all the PNG files containing the graphs are available in the “results/” directory.

Conclusion

The objective of this project was to test if the NCD, or Normalized Compression Distance, could be used to identify the music a sample is part of. After developing and testing the programs mentioned in this report, we can say that the NCD can in fact be used in music identification.

The results obtained showed that the overall accuracy of the developed program was 93.1% for identifying the correct song using a sample, and an accuracy of 97.58% for having the correct song in the top 10 songs with lowest NCD.

Furthermore, we can confidently say that the duration of the sample is directly related to the accuracy of the program, with the longer the sample the better the accuracy.

As for the impact of white noise volume, we unexpectedly found out that white noise does not have an impact on accuracy, and even improves it in some cases, but we can’t say that this is true in reality and might be due to lack of testing or random chance.

We also found that the compression methods impact the accuracy, since the lzma, zstd and bzip2 had an impressive 100% accuracy on most cases, while gzip and zip never reached 100% on any case. This is likely due to the algorithms used by each compression method, which shows that the better the algorithm of the compression method chosen, the more accurate the program is.

Future work

If we were to further work on this project, we would first like to do more tests with songs of different genres, using different instruments and with/without vocals to see if there was a relation between those and the accuracy of the programs.

We would also analyze different parameters to have a better understanding of the performance of the program, as well as the quality of the results and the conclusions we reach from them.

As for improving the program itself, we could try to write it in other languages to make it faster, like C or newer programming languages that leverage GPUs like CUDA.

We would improve the code itself and make it more readable, automation-friendly and with more documentation.

Annexes

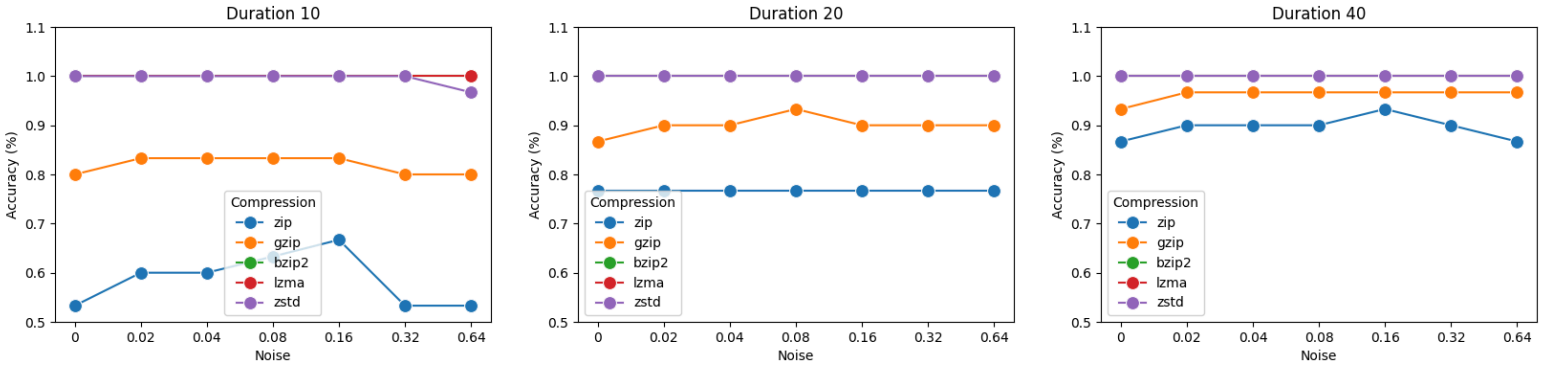

Figure 2 - Graphs with variation of compression method and white noise volume for the same duration

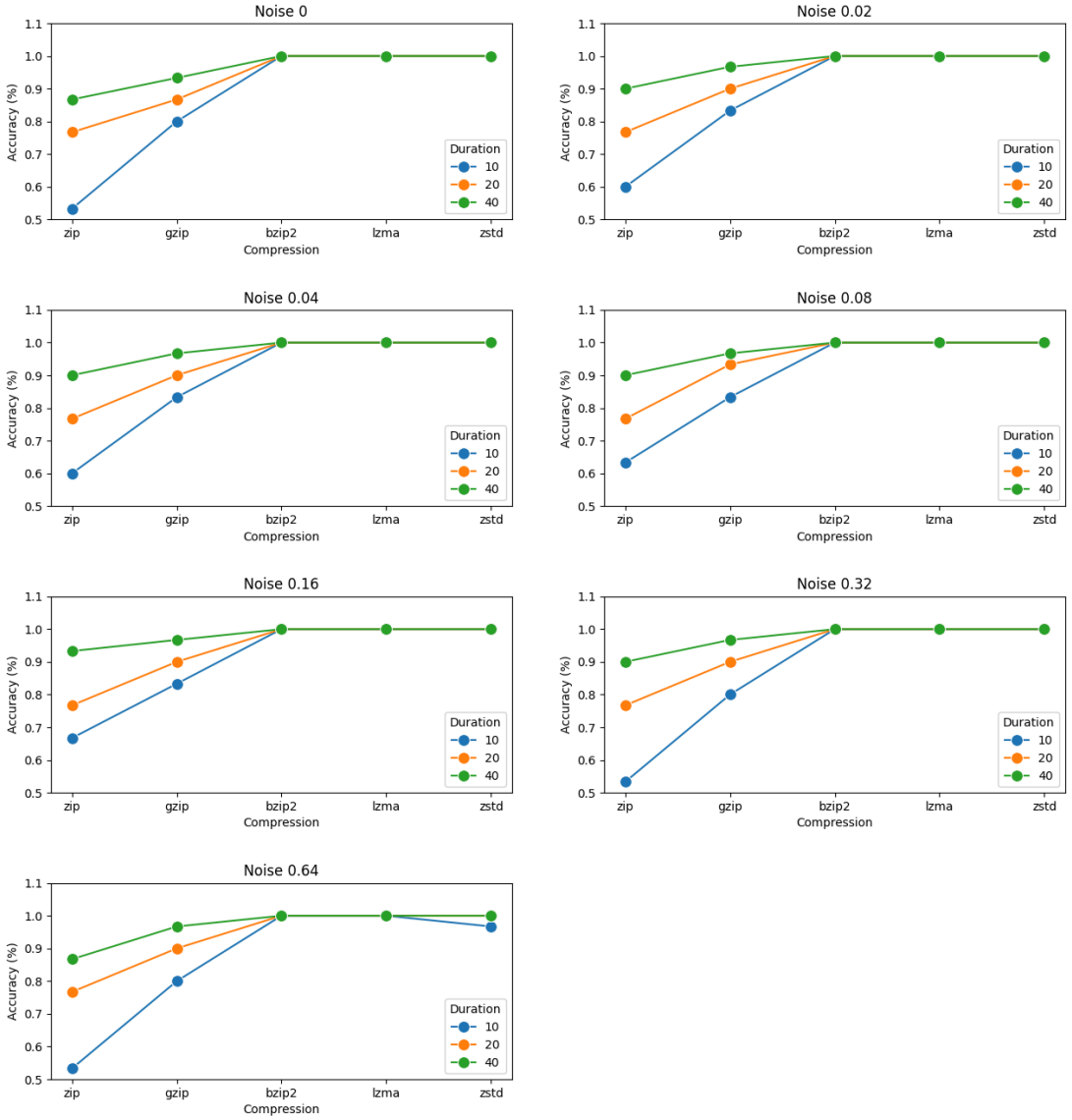

Figure 3 - Graphs with variation of compression method and duration for the same white noise volume