Music Identification Using Compression

Lab work no 3

Authors

Inês Baptista (98384)

João Correia (104360)

Daniel Ferreira (102885)

2023/2024

Algorithmic Information Theory

Universidade de Aveiro

Aveiro

1 Indice

Contents

2 Introduction

Music identification is a crucial task in various applications, such as music recommendation systems, music classification, and music retrieval. Traditional methods for music identification rely on audio features, such as spectrograms, mel-frequency cepstral coefficients, and chroma features. This project aims to develop an Information Theoretical approach to music identification using **compression** techniques.

The main idea is to use the Normalized Compression Distance (NCD) to measure the **similarity** between two audio segments. The NCD is an approximation of the non-computable Normalized Information Distance (NID), which is defined as the maximum of the Kolmogorov complexities of the two segments given each other. To avoid the non-computability of the Kolmogorov complexity, the NCD uses the approximation:

$$\mathrm{NCD}(x,y) = \frac{C(x,y) - \min\{C(x), C(y)\}}{\max\{C(x), C(y)\}} \tag{1}$$

where $C(x)$ denotes the number of bits required by compressor C to represent x and $C(x,y)$ indicates 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 this report, we will detail the steps needed to test our approach.

2.1 Music identification

Suppose that a certain database contains representations of several (complete) musics, denoted m_i, and that we want to identify a segment of music, represented by string x. The idea is to compute the values of $\mathrm{NCD}(x, m_i)$ for all i, and assign to x the name of the music in the database that yields the smallest value of the NCD.

3 Methodology

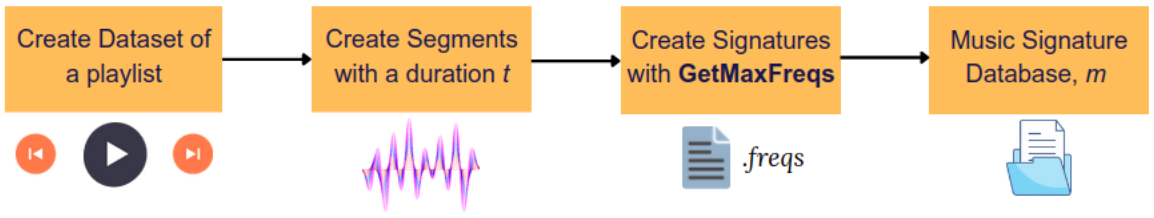

Create Dataset of a playlist
Create Segments with a duration *t*
Create Signatures with **GetMaxFreqs**
Music Signature Database, *m*
.freqs

Figure 1: Stage 1: Generate the database m_i

3.1 Stage 1: Generate the music signature database m_i

In the figure 2 below, we see the main steps to generate the database with the signatures that will be used to idenity music within audios.

1. **Playlist Generation**

   The dataset used for this project consists of 25 different musics, downloaded from YouTube. The **create_dataset.py** script automates the process of downloading audio from YouTube videos and playlists, converting the audio to a specified format (`mp3` or `wav`) with desired sample rate, bit depth, and number of channels, and saving it to a designated output directory. It supports batch processing through command-line arguments and input files containing URLs.

2. **Segments Creation**

   Segmenting the original audio files is essential because applications like *Shazam* require only a short duration of audio to accurately identify the music. The **create_audio_segment.py** script is designed to create segments of audio files with specified durations t from given audio files or directories containing audio files. It takes input arguments such as the duration of the segment in seconds, the minimum start time of the segment, and the output path. The script utilizes the `SoX` (Sound eXchange) library to manipulate audio files. The start time for each segment is either randomly chosen or in a specified range.

3. **Signature Generation**

   The audio undergoes a transformation to a representation more suitable for the intended general-purpose compressors, using the provided, by the professors, **GetMaxFreqs** program in `C++`. This process involves segmenting the audio, partially overlapping, and computing the most significant frequencies for each segment. The **GetMaxFreqs** program reads the audio data, converts it to mono, downsamples it, computes the Fast Fourier Transform (FFT) over sliding windows, identifies the top frequencies by power, and saves these frequencies to an output file `.freqs`.

3.2 Stage 2: Identify a music x given an audio x

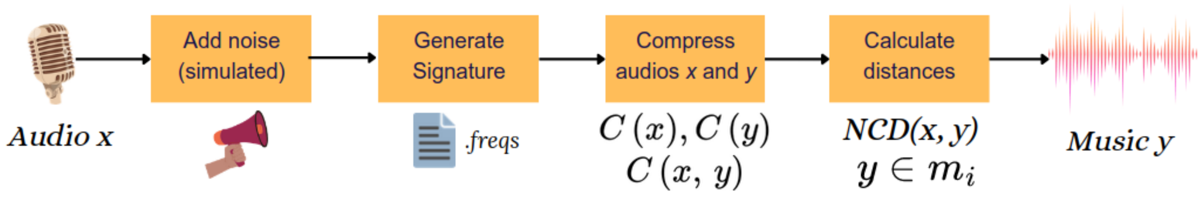


Figure 2: Stage 2: Identify a music y given an audio x containing a music and simulated noise on the background.

1. **Simulating Noise:** To assess the reliability of our music identification system, noise was added to the audio segments used for querying the database. The `create_noise.py` script is utilized to generate different types of noise and add them to the audio segments. Various types of noise can be added, including:

   - **White noise:** Random noise with equal intensity at different frequencies.
   - **Pink noise:** Noise with more energy in lower frequencies, similar to many natural sounds

- **Brown noise:** Noise with even more energy in lower frequencies compared to pink noise.
- **Blue noise:** Noise with more energy in higher frequencies.
- Any **YouTube video**.

The **intensity** of the added noise can be controlled to create different signal-to-noise ratios, allowing for testing the music identification technique under various noise conditions.

2. **Signature Generation**

   The audio undergoes a transformation to a representation more suitable for the intended general-purpose compressors, using the provided, by the professors, **GetMaxFreqs** program in `C++`. This process involves segmenting the audio, partially overlapping, and computing the most significant frequencies for each segment. The **GetMaxFreqs** program reads the audio data, converts it to mono, downsamples it, computes the Fast Fourier Transform (FFT) over sliding windows, identifies the top frequencies by power, and saves these frequencies to an output file `.freqs`.

3. **Compreesion**

   To compute the Normalized Compression Distance (NCD), we compress both x and y separately to determine their individual compressibility, denoted as $C(x)$ and $C(y)$ respectively. Additionally, we compress their concatenation of $x+y$ to capture shared patterns, represented by $C(x, y)$. These lengths of each compression serve as the basis for computing the NCD, indicating the similarities or dissimilarities between the two entities based on the information extracted through compression, also known as **entropy**. For this task, we used the standard compressors **gzip**, **bz2**, **lzma**, **zstd**, **zlib**, **lz4**, **lz4**, and **snappy**. Later in the analysis, we analyze which one provided the best results.

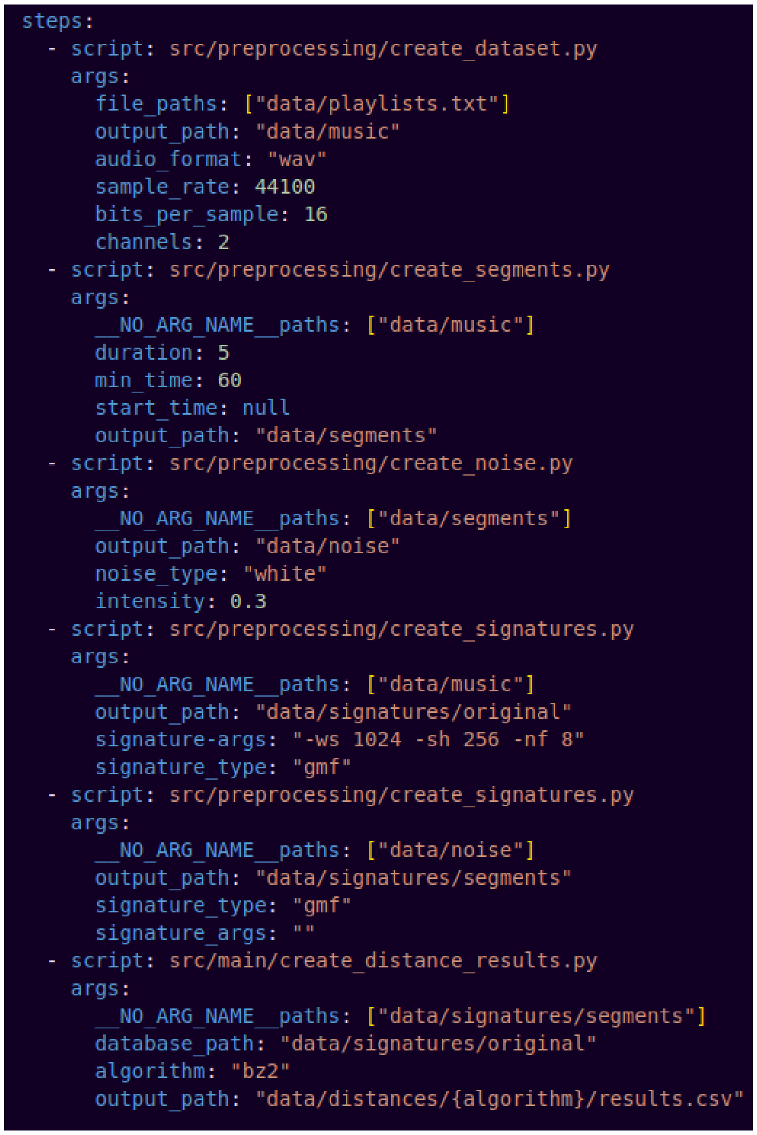
4. **NCD Calculation:** The database is queried using small samples of the music pieces. For each query, the Normalized Compression Distance (NCD) is calculated, and the music in the database yielding the smallest NCD value is assigned to the query. The NCD computation is performed for each pair of audio segments in the database, serving as a metric to gauge the similarity between the segments.

3.3 Automations with pipeline execution

Both stages can be automated through a **pipeline** with custom made `YAML` files. These files define the steps to be taken. For example, a `YAML` file is created with the structure shown in Figure 3.

The pipeline is then executed using the `src/pipeline.py` script. It is important to note that steps within the pipeline can be excluded to avoid repetition. For example, `create_dataset.py` only needs to be run once, as does the creation of original signatures. Furthermore, as depicted in Figure 3, the process includes the calculation of all distances, essentially forming a Cartesian product. However, this might not always be necessary and the `YAML` file can be tailored based to just identify one segment.

When running the pipeline multiple times, be cautious of calculating results for duplicate files. It is recommended to delete directories that are not overwritten, such as those containing audio segments. There is an option to automate this process or use the `src/clean.py` script. For initial setup, run the `src/bash/install.sh` script to install the necessary libraries.



```yaml
steps:
  - script: src/preprocessing/create_dataset.py
    args:
      file_paths: ["data/playlists.txt"]
      output_path: "data/music"
      audio_format: "wav"
      sample_rate: 44100
      bits_per_sample: 16
      channels: 2
  - script: src/preprocessing/create_segments.py
    args:
      __NO_ARG_NAME__paths: ["data/music"]
      duration: 5
      min_time: 60
      start_time: null
      output_path: "data/segments"
  - script: src/preprocessing/create_noise.py
    args:
      __NO_ARG_NAME__paths: ["data/segments"]
      output_path: "data/noise"
      noise_type: "white"
      intensity: 0.3
  - script: src/preprocessing/create_signatures.py
    args:
      __NO_ARG_NAME__paths: ["data/music"]
      output_path: "data/signatures/original"
      signature-args: "-ws 1024 -sh 256 -nf 8"
      signature_type: "gmf"
  - script: src/preprocessing/create_signatures.py
    args:
      __NO_ARG_NAME__paths: ["data/noise"]
      output_path: "data/signatures/segments"
      signature_type: "gmf"
      signature_args: ""
  - script: src/main/create_distance_results.py
    args:
      __NO_ARG_NAME__paths: ["data/signatures/segments"]
      database_path: "data/signatures/original"
      algorithm: "bz2"
      output_path: "data/distances/{algorithm}/results.csv"
```

Figure 3: Pipeline Configuration YAML file for automation purposes

3.4 Enhancing Performance With Multiprocessing

To improve the performance of the audio signature generation process, the `create_gmf_signatures` function sets up a multiprocessing pool to generate signatures in **parallel**. It uses `Pool` from `multiprocessing` `Python` library.

Listing 1: Python code for the ‘create_gmf_signatures‘ function

```python
def create_gmf_signatures(paths, output_path, args, verbose=False):
    os.makedirs(output_path, exist_ok=True)

    if not check_dependencies() or not compile_get_max_freqs():
        return

    with Pool(cpu_count()) as pool:
        tasks = [(path, output_path, args) for path in paths]
        results = pool.map(generate_signature, tasks)

    if verbose:
        for path, result in zip(paths, results):
            if result:
                print(f"Generated signature for {path}")
            else:
                print(f"Failed to generate signature for {path}")
```

We define a function `generate_signature` that generates the signature for a single audio file. This function will be used by the `Pool` object to process files concurrently:

Listing 2: Python code for the 'generate_signature' function

```python
def generate_signature(args):
    path, output_path, signature_args = args
    output_file = os.path.join(
        output_path,
        os.path.basename(path).rsplit('.', 1)[0] + ".freqs")
    signature_args = signature_args.split()
    if subprocess.run([
            "GetMaxFreqs/bin/GetMaxFreqs",
            "-w",
            output_file
            ]
            + signature_args + [path]
        ).returncode != 0:
        print(f"Failed to generate signature for {path}")
        return False
    return True
```

3.5 Enhancing Performance with Caching

The function `get_compressed_length` utilizes a cache mechanism to compress each signature only once, ensuring efficiency by storing and retrieving previously computed results.

Listing 3: Python code for the 'get_compressed_length' function

```python
def get_compressed_length(algorithm, data, cache, cache_key):
    if cache_key not in cache:
        cache[cache_key] = len(compress_file(algorithm, data))
    return cache[cache_key]
```

4 Results and analysis

4.1 Parameter Tuning

The program **GetMaxFreqs** generates signatures with adjustable parameters. The **Window Size (-ws)** determines FFT window size, impacting frequency resolution and computational cost. The **Window Overlap (-sh)** controls sample overlap, enhancing temporal resolution. The **Number of Significant Frequencies (-nf)** limits considered frequencies, balancing computational efficiency with information retention.

In the Figure 4, the accuracy is higher when $ws = 2560$, $sh = 0.0625$ and $nf = 3$.

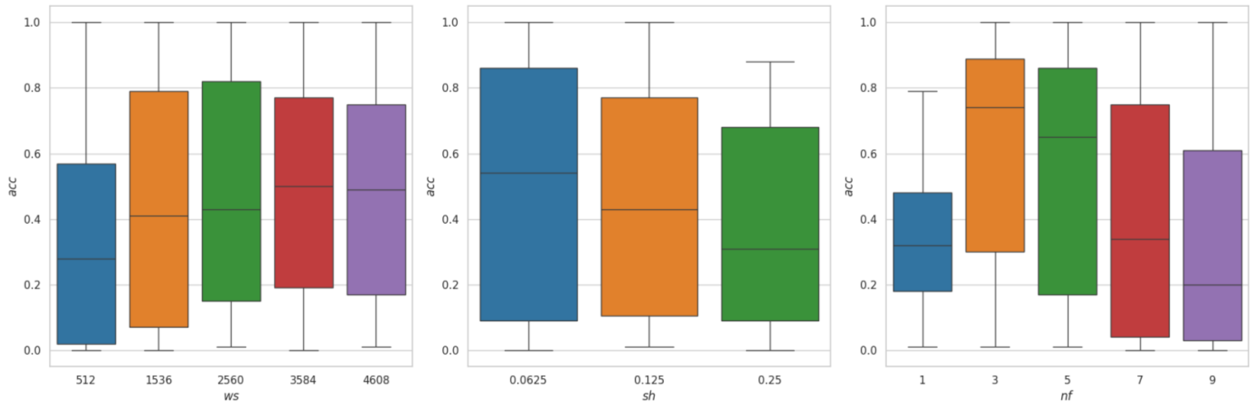


Figure 4: Accuracy results in function of each of the parameters **ws**, **sh** and **nf**.

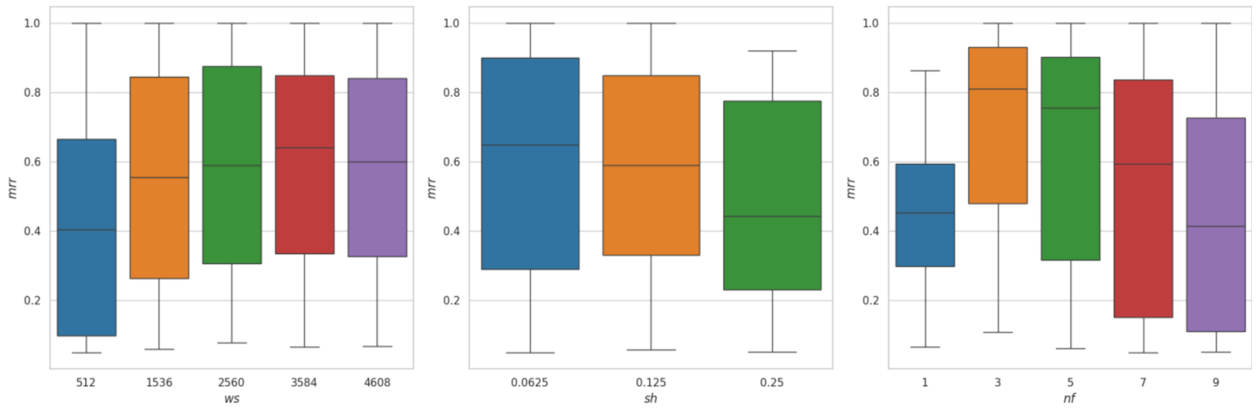


Figure 5: Mean Reciprocal Rank (MRR) results in function of each of the parameters **ws**, **sh** and **nf**.

In the Figure 5, Mean Reciprocal Rank (MRR) is also higher when ws = 2560, sh = 0.0625 and nf = 3. A high MRR (Mean Reciprocal Rank) means that the relevant items are ranked very high in the list of returned results and follows the formula:

$$\text{MRR} = \frac{1}{|Q|} \sum_{i=1}^{|Q|} \frac{1}{\text{rank}_i} \tag{2}$$

4.1.1 Performance Evaluation

In the Figure 6, we see the time for different parameter combinations. Each time meaning:

- **t1** is the time to generate the signatures of the complete songs.
- **t2** is the time to generate the signatures of the segments.
- **t3** is the time to calculate the distances.

From the figure, we see that the parameters don't have an impact of **t2** - which makes sense has the segments have all the same duration. Time **t1** increases as **sh** increases and **ws** and **nf**

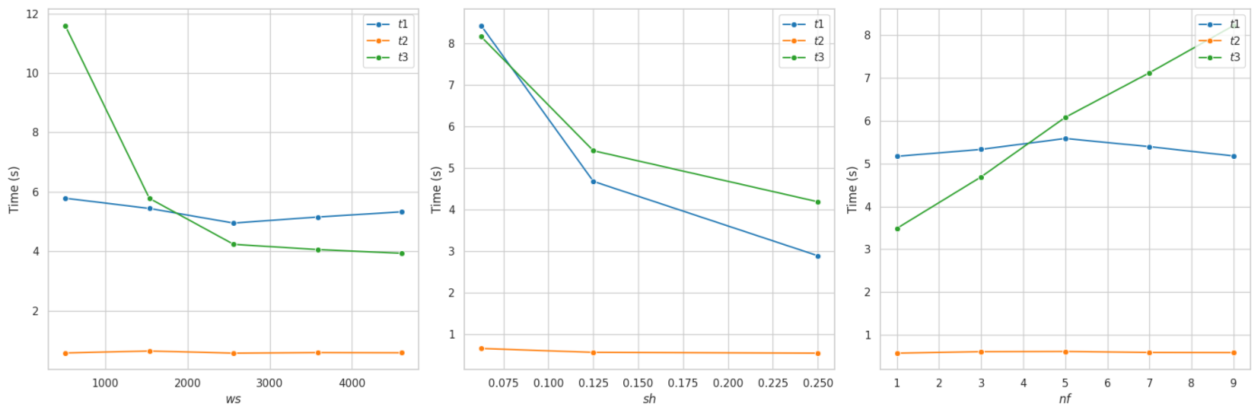


Figure 6: Time *t1, t2 and t2* in function of the parameters **ws**, **sh** and **nf**.

also don't have a big impact on this timing. The time to calculate the distances, **t3**, is influenced by all the parameters. It is inversely proportional to **ws** and **sh** (decreases as **ws** and **sh** decrease) and directly proportional to **nf** (increases as **nf** increases).

4.2 Compressors comparison

We also compare different compressors - **gzip**, **bz2**, **lzma**, **zstd**, **zlib**, **lz4**, **lz4**, and **snappy** to find the one who retrieves higher accuracy.

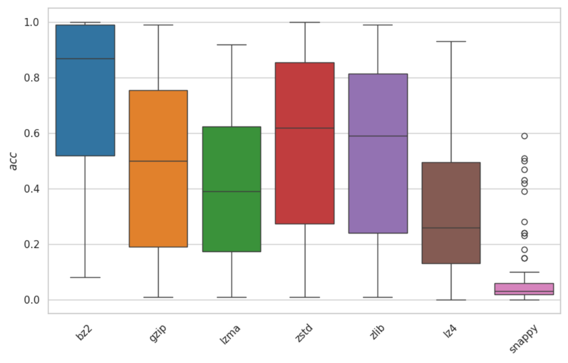


Figure 7: Accuracy results in function of each of the compressors.

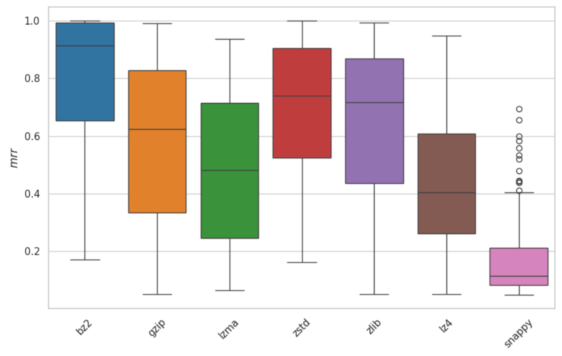


Figure 8: Mean Reciprocal Rank (MRR) results in function of each of the compressors.

In Figures 7 and 8, a clear trend emerges: the accuracy and Mean Reciprocal Rank (MRR) metrics favor the **bz2** compressor, showcasing its superiority over other contenders, with **zstd** closely following as the second-best performer. Conversely, **snappy** achieves very poor results for both metrics, solidifying its position as the worst compressor for this task.

This aligns with our understanding of the compression algorithms at play. **Snappy**, designed for speed and efficiency, prioritizes fast compression and decompression over achieving high compression ratios. This focus on speed likely leads to discarding some of the musical details that are crucial for accurate music identification.

Music thrives on repetitive patterns and transient elements. Compressors that excel at capturing these subtleties perform better in music identification tasks. **bz2** and **zstd**, with their more complex algorithms, seem adept at identifying and representing these repeating patterns. This translates to more accurate comparisons between the compressed query clip and music pieces in the database, reflected in the higher accuracy and MRR values.

Interestingly, **lz4**, another speed-focused compressor, outperformed snappy significantly. While both prioritize speed, it appears **lz4**'s design choices allow it to retain more musical information. This reinforces the notion that not all fast compressors are created equal for music identification. Nonetheless, the results for **lz4** still fall short of those achieved by **bz2** and **zstd**, highlighting the significance of striking a balance between speed and compression quality for precise music identification.

4.3 Best combinations

To identify optimal parameter combinations, we did a gridsearch through the parameters for each compressor - **bz2**, **gzip**, **lzma**, **zstd**, **zlib**, **lz4**, **snappy**. Due to the necessary trade-off between precision and computing time, we used an additional composite metric balancing accuracy and computational efficiency. We computed the total processing time (sum of $t1$, $t2$, and $t3$) and normalized it to a range of 0 to 1, using *min-max scaling*.

The composite metric, a weighted sum of accuracy (60%) and normalized total time (40%), prioritizes high accuracy while penalizing excessive computation time. The top 6 combinations can be found in Table 1 and the tested parameters intervals were the following:

- **Window Sizes (ws)**: {512, 1536, 2560, 3584, 4608}
- **Window Overlap (sh)**: { $\frac{1}{16}$, $\frac{1}{8}$, $\frac{1}{4}$ of ws }
- **Number of Significant Frequencies (nf)**: {1, 3, 5, 7, 9}

Table 1: Top 6 combinations resulting from varying the parameters above.

Combination	ws	sh	nf	Compressor	acc	MRR	$t1$	$t2$	$t3$
1	1536	192	5	bz2	1.0	1.0	4.8069	0.6107	5.2554
2	2560	160	3	gzip	0.99	0.9909	7.2340	0.5786	2.2616
3	2560	160	7	lzma	0.87	0.8997	7.3348	0.6003	22.2095
4	2560	160	3	zstd	1.0	1.0	7.2340	0.5786	2.7799
5	2560	160	3	zlib	0.99	0.9925	7.2340	0.5786	3.3078
6	4608	288	5	lz4	0.93	0.9482	8.4333	0.6347	1.4662
7	2560	160	1	snappy	0.59	0.6942	8.1086	0.6971	1.4702

4.4 Impact of Segment Size

The *segment size* is one of the most important parameters as the more information we have on an audio segment of a music, more accurate the results as we have more information of the audio/song. However it is still interesting to test the *threshold* where our model starts being more efficient. For this, we evaluated the accuracy for multiple *segment size* values for the best found compressors

bz2 and **lzma**.

The results are presented in Figure 9. Interestingly, the model demonstrated high efficiency starting at a segment size of 10 seconds, which coincidentally aligns with our initial choice. For segment sizes of 10 seconds, both compressors achieved reasonable results, with ≈ 97.5% for **bz2** and 90% for **lzma**. However, for smaller segments, ≈ 2 seconds, **lzma** performed poorly, while **bz2** still delivered satisfactory accuracy, reaching ≈ 92.5%.

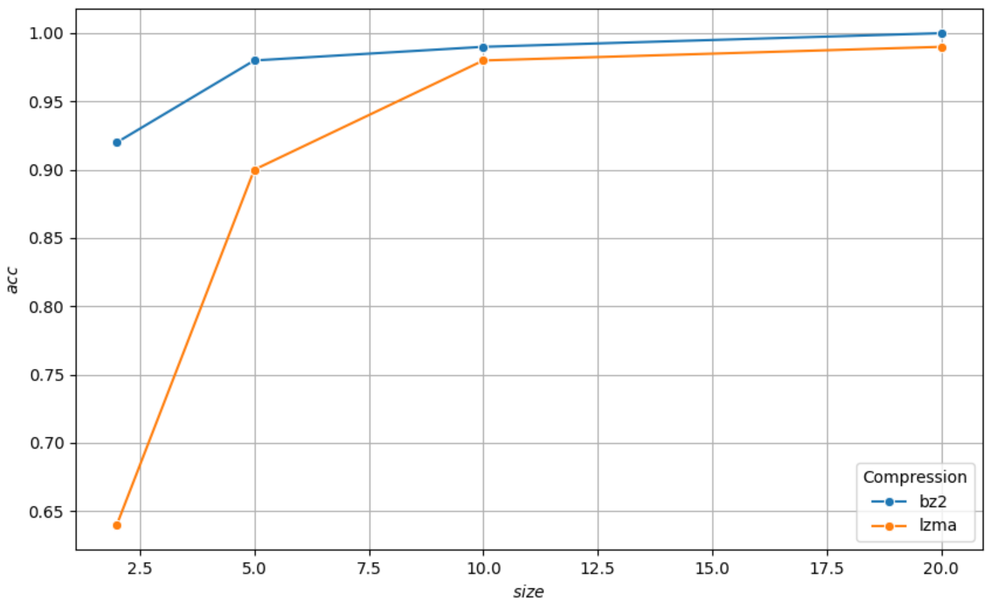


Figure 9: Accuracy as a function of the segment size for the 2 best compressors found.

4.5 Performance Evaluation with Different Noise Types

Since the primary use cases for music identifier apps (e.g., Shazam) occur in the real world, which might be quite loud, adding noise to our audios helps determine if our model will perform effectively identifying music in a real world setting.

Analyzing Figure 10, we noticed that the model performed much better when exposed to **white noise** compared to other types of noise. This could be because **white noise** has a uniform frequency spectrum, meaning all frequencies have equal energy. This makes it easier to pick out the music signal from the noise. On the other hand, **brown** and **pink noise** have spectral characteristics where the energy decreases as frequency increases. This can hide certain parts of the music signal, making it harder to detect. Additionally, **brown noise** has a memory effect due to its specific pattern, which can mess with the timing of the music signal and make it harder to spot.

4.6 Impact of Noise Intensity on the Performance

To further assess our model's **robustness** and **reliability**, we subjected our pipeline (Figure 3) to varying levels of noise intensity, testing the top 6 configurations from Table 1, excluding **snappy** due to imprecise results.

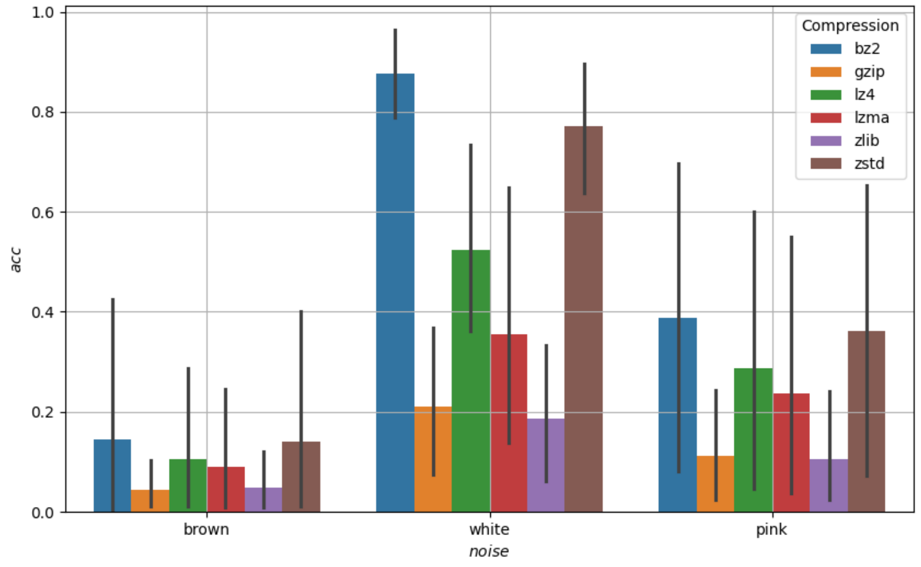


Figure 10: Accuracy as a function of the type of noise added to the audio for all the compressors.

The results in Figure 11 show that while our model showed great results, it has difficulties in identifying musics with very noisy audios ($intensity = 5$). These noises are not representative of the noise in the real world so we do not know the behaviour of our model in a real scenario.

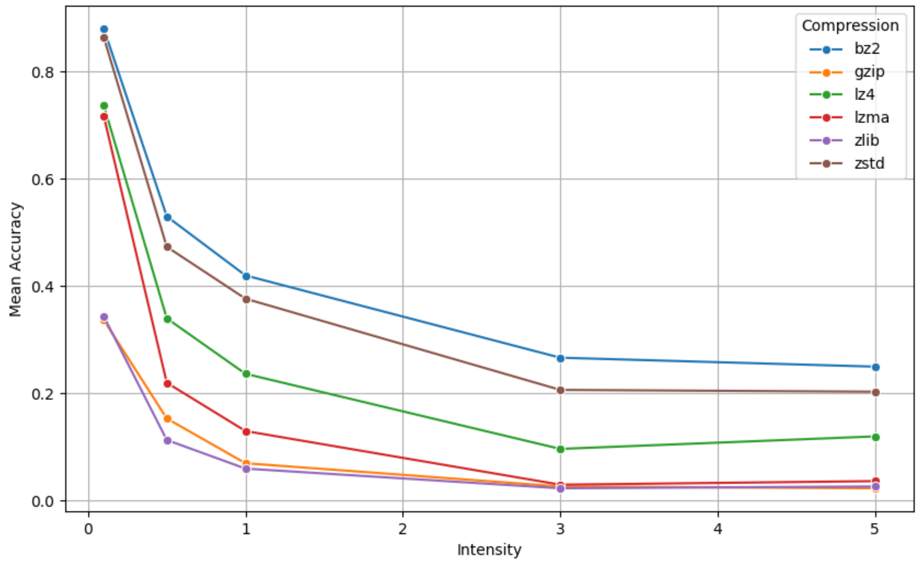


Figure 11: Mean Accuracy as a function of different noise intensities for the different compressors.

5 Conclusion

The project aims to develop a novel approach to music identification using compression techniques, with the results demonstrating the effectiveness of the NCD-based approach in identifying music. The reliability of the technique was assessed by adding noise to the segments used for querying.

We notice that the model performed significantly better for white noise than the other types of noises. Additionally, we observe considerable variability in accuracy results for each compressor across different types of noise, indicating the sensitivity of the model's performance to the characteristics of the noise and the compression method used.

The best compressor found was the **bz2** compressor, closely followed by **zstd**, in terms of accuracy and Mean Reciprocal Rank (MRR) metrics, while **snappy** demonstrates the worst poor performance. This observation aligns with the known characteristics of compression algorithms; compressors like **bz2** and **zstd**, adept at capturing musical patterns, perform better in music identification tasks compared to **snappy**, which prioritizes speed over compression quality.

Notably, **lz4**, while faster than **snappy**, falls short of the performance achieved by **bz2** and **zstd**, underscoring the importance of a balanced approach between speed and compression quality for accurate music identification.

Despite the satisfactory results, these simulated noises are not representative of the noise in the real world (it can vary a lot depending on the user location) so we do not know the behaviour of our model in a real scenario. It's also noteworthy to say that our database is very small and the process may not yield satisfactory results for larger datasets, as it consumes a significant amount of disk memory. Implementing it for a large number of songs may prove impractical.