Normalized Compression Distance

Raquel Paradinha 102491 Paulo Pinto 103234 Miguel Matos 103341
Filipe Antão 103470

Teoria Algoritmica da Informação
Departamento de Electrónica, Telecomunicações e Informática
Universidade de Aveiro

June 10, 2024

Contents

Abstract

In this report, we explain how we implemented a compression distance calculator. This calculator determines the normalized compression distance of a song and tries to guess the song on the dataset based on a sample (10s) of the song. The solution was evaluated using different compressors, and the results were presented.

1 Introduction

In an age of digital music, the ability to quickly and precisely identify a song is a very sought-after tool since people listen to music in everyday life. In contrast, in the car or the gym, where they might not have access to the details of the song, users must be provided with the means of quickly identifying the song.

This work focuses on implementing this mechanism, which relies on calculating a Normalized Compression Distance. First, the songs must be converted from a .wav format to a more adequate representation. Then, the song is compressed using different compressors, and the normalized compression distance to the other songs in the database is calculated. From these calculations, the matched song is returned.

1.1 Normalized Compression Distance

The Normalized Compression Distance (NCD) approximates the Normalized Information Distance (NID) using compression. The formula used to calculate the NCD is the following:

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

C(x) represents the number of bits required by compressor C to define x, and C(x,y) represents the number of bits needed to compress x and y together. Distances near zero indicate similarity, and those closer to one represent dissimilarity.

1.2 Changes to the *GetMaxFreqs* program

The `GetMaxFreqs` program is designed to analyze stereo audio files sampled at 44100 Hz and 16-bit resolution, such as `.wav` and `.flac` files. It processes the audio data to extract and record the most significant frequency components, creating a "signature" for each audio file. The primary function handles command-line input to specify various parameters, including verbosity, output file name, window size, shift, down-sampling factor, and the number of significant frequencies.

Upon execution, the program reads the specified audio file, converts it to mono, and down-samples the audio data. It then applies the Fast Fourier Transform (FFT) on overlapping windows of the audio signal to compute the power spectrum. The top `nf` frequencies with the highest power are identified for each window, and their indices are stored in the output file. This "signature" file can later be used for tasks such as audio comparison or identification, leveraging the distinctive frequency patterns of the audio content.

The teachers gave us this program, and we were allowed to make changes if we thought optimizations were possible. We identified two places where minor optimizations were possible:

1. **Improved Command-Line Argument Parsing**
   - **Old Version:** Multiple separate loops were used to parse each command-line argument individually.
   - **New Version:** Consolidates all command-line argument parsing into a single loop.
   - **Advantages:**
     (a) **Efficiency:** Reduces the number of loops from six to one, which minimizes the loop overhead and makes the code run more efficiently.
     (b) **Readability:** A single loop is more concise and easier to understand, making the logic of command-line argument handling clearer.
     (c) **Maintainability:** Having fewer lines of code reduces complexity, making the code easier to maintain and less prone to errors when adding or modifying argument parsing logic.
2. **Combined Loop for Power Calculation and Index Initialization**
   - **Old Version:** Two separate loops were used to calculate the power spectrum and initialize the index array.
   - **New Version:** Combines the power calculation and index initialization into a single loop.
   - **Advantages:**

(a) **Efficiency:** Reduces the overhead of running multiple loops by performing both operations within a single loop.

(b) **Cache Performance:** Accessing and modifying the power array and index array within the same loop can improve cache performance by utilizing the CPU cache more effectively.

(c) **Readability:** The combined loop maintains logical clarity while improving performance, making the code more streamlined without sacrificing comprehensibility.

2 Dataset

The dataset consists of 25 different .wav files of arbitrary songs we chose. When choosing said dataset, we tried to mix different genres of songs to provide more heterogeneous data to the program. A comprehensive table containing all the songs used can be found below:

ID	Title	Artist
1	Thrown Around	James Blake
2	CHIHIRO	Billie Eilish
3	HEAVEN TO ME	Tyler, The Creator
4	Eyes Closed	Imagine Dragons
5	Houdini	Eminem
6	Little Foot Big Foot	Childish Gambino
7	Please Please Please	Sabrina Carpenter
8	FE!N	Travis Scott
9	1093	Yeat
10	F33l Lik3 Dyin	Playboi Carti
11	Lying 4 fun	Yeat
12	on one tonight	Gunna
13	Too Sweet	Hozier
14	Music For a Sushi Restaurant	Harry Styles
15	Goodie Bag	Still Woozy
16	Chelas	Sara Correia
17	Pica Do 7	António Zambujo
18	Canção de Engate	Tiago Bettencourt
19	jingle.tm	PTM
20	Angry	The Rolling Stones
21	Dark Necessities	Red Hot Chili Peppers
22	Feel Good Inc.	Gorillaz
23	Party Rock Anthem	LMFAO
24	Let's Get It Started	The Black Eyed Peas
25	Cake By The Ocean	DNCE

Table 1: List of Songs and Their Artists

2.1 Frequency Files

Each file was then converted using the *GetMaxFreqs* function into a text representation adequate for general-purpose compressors. Since the provided solution only supported the conversion of files with a frequency rate equal to 44100Hz, some files had to be pre-processed to resample them into this frequency. This process was conducted using Audacity. The following command was used to generate the frequency signature of each audio file:

```
./GetMaxFreqs -w {music_name}.freqs dataset/{music_name}.waw
```

2.2 Test Dataset

We generated a test dataset from the music collection by extracting a random 10-second snippet from each song. Each snippet had three types of noise added (white, pink, and brown) at varying intensities.

3 Implementation

Our program is designed to calculate the Normalized Compression Distance (NCD) between a sample audio file and files in a dataset.

Our implementation consists of three main source files (‘main.cpp‘, ‘compressor.cpp‘, and ‘utils.cpp‘) and their respective header files (‘Compressor.h‘ and ‘utils.h‘). Additionally, we use a ‘Type.h‘ header file to define the enumeration of compressor types. Below, we describe the functionality and purpose of each file.

3.1 Main Program: `main.cpp`

The main program (main.cpp) efficiently handles command-line arguments, manages file operations, and orchestrates the NCD computation. It utilizes the utility functions in utils.cpp for file handling and the compression capabilities of compressor.cpp. The command-line interface offers flexibility, allowing users to specify the compression algorithm, dataset folder, and sample file, making the program adaptable to different datasets and user requirements. It includes the following functionalities:

- **Command-Line Arguments:** Parses options for specifying the compressor type, dataset folder, and sample file. The usage options are listed in Table 2.

Option	Description
`-h`	Display help information
`-v`	Run in verbose mode
`-c <compressor>`	Specify the compressor type (default: GZIP)
`-d <folder>`	Specify the dataset folder
`-f <file>`	Specify the sample file

Table 2: Command-Line Arguments

- **NCD Calculation:** Utilizes the `calculateNCD` function to compute the NCD between the sample file and each song in the dataset using the specified compression algorithm.
- **File Handling:** Uses functions from `utils.cpp` to read and list all files in the dataset directory.
- **Result Processing:** Outputs the NCD for each song and identifies the song with the minimum NCD, indicating the closest match.

3.2 Compression Functionality: `compressor.cpp`

The compressor.cpp file implements the Compressor class, which supports multiple compression algorithms such as GZIP, BZIP2, LZMA, ZSTD, LZ4, LZO, and SNAPPY. Each compression method is designed to handle large datasets efficiently, compressing data in chunks and returning the compressed size in bits—a critical aspect for accurate NCD calculation. Table 3 lists the supported compression algorithms and the libraries used to implement them:

Compression Algorithm	Library
GZIP	zlib
BZIP2	bzlib
LZMA	lzma
ZSTD	zstd
LZ4	lz4
LZO	lzo
SNAPPY	snappy

Table 3: Supported Compression Algorithms

Each method returns the size of the compressed data in bits, which will be used for NCD calculation. Below, we explain how each compression algorithm works within the `Compressor` class, grouping similar methods.

3.2.1 GZIP (zlib) and BZIP2 (bzlib)

Both GZIP and *BZIP2* use well-known libraries (*zlib* for *GZIP* and *bzlib* for *BZIP2*) to perform compression [1, 7]. The general process involves:

1. Initializing a compression stream and setting up the necessary parameters for compression.
2. Compressing the input data in chunks, allows the program to handle large data efficiently.
3. Finalizing the compression process to ensure all data is compressed and updating the total size of the compressed data.
4. Returning the total size of the compressed data in bits.

3.2.2 LZMA (lzma)

LZMA compression uses the *lzma* library [6]. The steps include:

1. Initializing a *lzma_stream* structure and configuring the encoder with appropriate parameters.
2. Compressing the input data in chunks, similar to GZIP and BZIP2.
3. Finalizing the compression process and handling any errors that might occur.
4. Returning the total size of the compressed data in bits.

3.2.3 ZSTD (zstd) and LZ4 (lz4)

Both *ZSTD* and *LZ4* are designed for high-speed compression [3, 2]. They share a similar approach:

1. Determining the maximum possible compressed size for the input data and allocating a buffer accordingly.
2. Compressing the input data and checking for errors during the process.
3. Returning the size of the compressed data in bits.

3.2.4 LZO (lzo)

LZO compression uses the *lzo* library [5], known for its speed:

1. Initializing the library and preparing the work memory required for compression.
2. Compressing the input data in a single step, handling any errors.
3. Returning the size of the compressed data in bits.

3.2.5 SNAPPY (snappy)

SNAPPY compression, implemented using the *snappy* library [4], is optimized for speed rather than maximum compression:

1. Calculating the maximum compressed length and allocating a buffer.
2. Performing the compression and checking for errors.
3. Returning the size of the compressed data in bits.

3.3 Utility Functions: `utils.cpp`

The utility functions in utils.cpp provides essential file and data manipulation capabilities. Functions like listFilesInDirectory, readFile, and concatenate simplify the main program's operations, ensuring the data is correctly prepared for compression and comparison.

- **File Listing:** `listFilesInDirectory` lists all regular files in a specified directory.
- **File Reading:** `readFile` reads the content of a file into a vector of characters.
- **Data Concatenation:** `concatenate` merges two vectors of characters.

These utilities are used by the main program to handle dataset files and prepare data for compression.

3.4 Type Definitions: `Type.h`

The Type.h header file defines an enumeration for the supported compression types, ensuring that the program can easily switch between different compression algorithms as specified by the user.

```
#ifndef TYPE_H
#define TYPE_H

enum class Type {
        GZIP,
        BZIP2,
        LZMA,
        ZSTD,
        LZ4,
        LZO,
        SNAPPY
    };

#endif // TYPE_H
```

This enumeration is used throughout the program to specify the compression algorithm.

This extensible design allows for future expansion, with minimal changes to the existing codebase, where additional compression algorithms can be incorporated.

4 Results

4.1 Compressor Performance

To evaluate the results of the compressors we chose, we created 4 test segments for each music in our dataset. These segments contained a no noise, white noise, pink noise, and Brownian noise version of each song. Each type of noise was applied at five different intensities, belonging to [0.05, 0.10, 0.15, 0.20, 0.25].

We included the three different types of noises in our analysis to verify how their specific characteristics influenced the performance of each compressor.

The plot of Figure 1 represents the accuracy of each compressor over all the test files (with and without noise). As we can see, for our song collection, *bzip2* has an accuracy of 100%, while *Snappy* and *LZO* are very subpar.

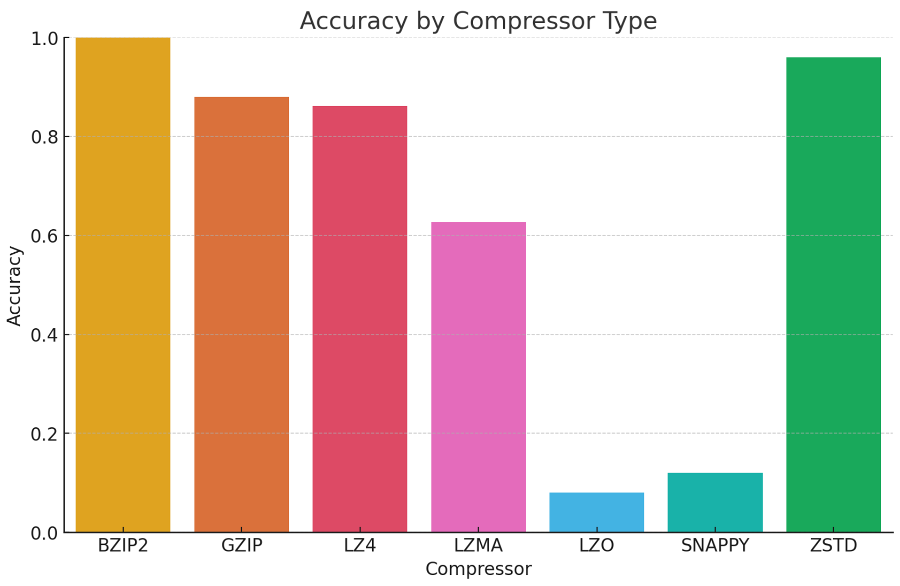


Figure 1: Accuracy per compressor

In Figure 2, we can see how the computed distances vary with the different tested compressors. *LZO* and *Snappy*, which have below 20% accuracy, produce consistently lower distance values, while the most accurate compressors all produce distance values closer to 1.

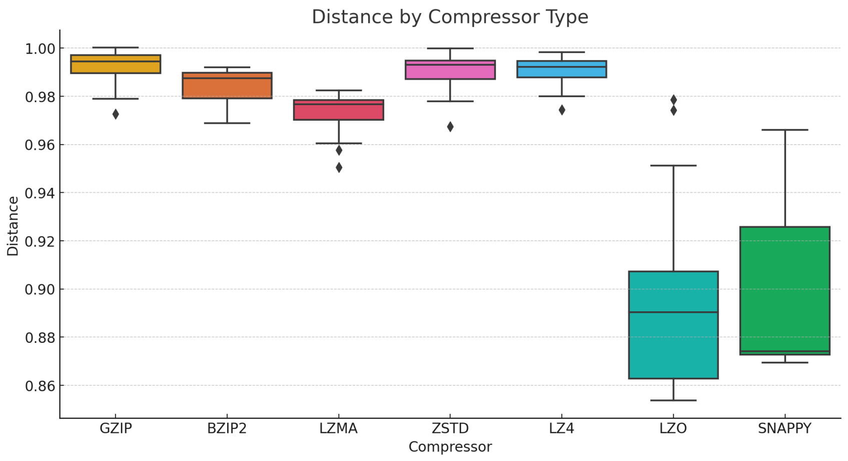


Figure 2: Distance values per compressor

Appendix A contains all the confusion matrices for each compressor's obtained results.

4.2 Noise Testing

Figure 3 plots how the overall accuracy of each compressor is affected by the amount of noise disturbing the original waves.

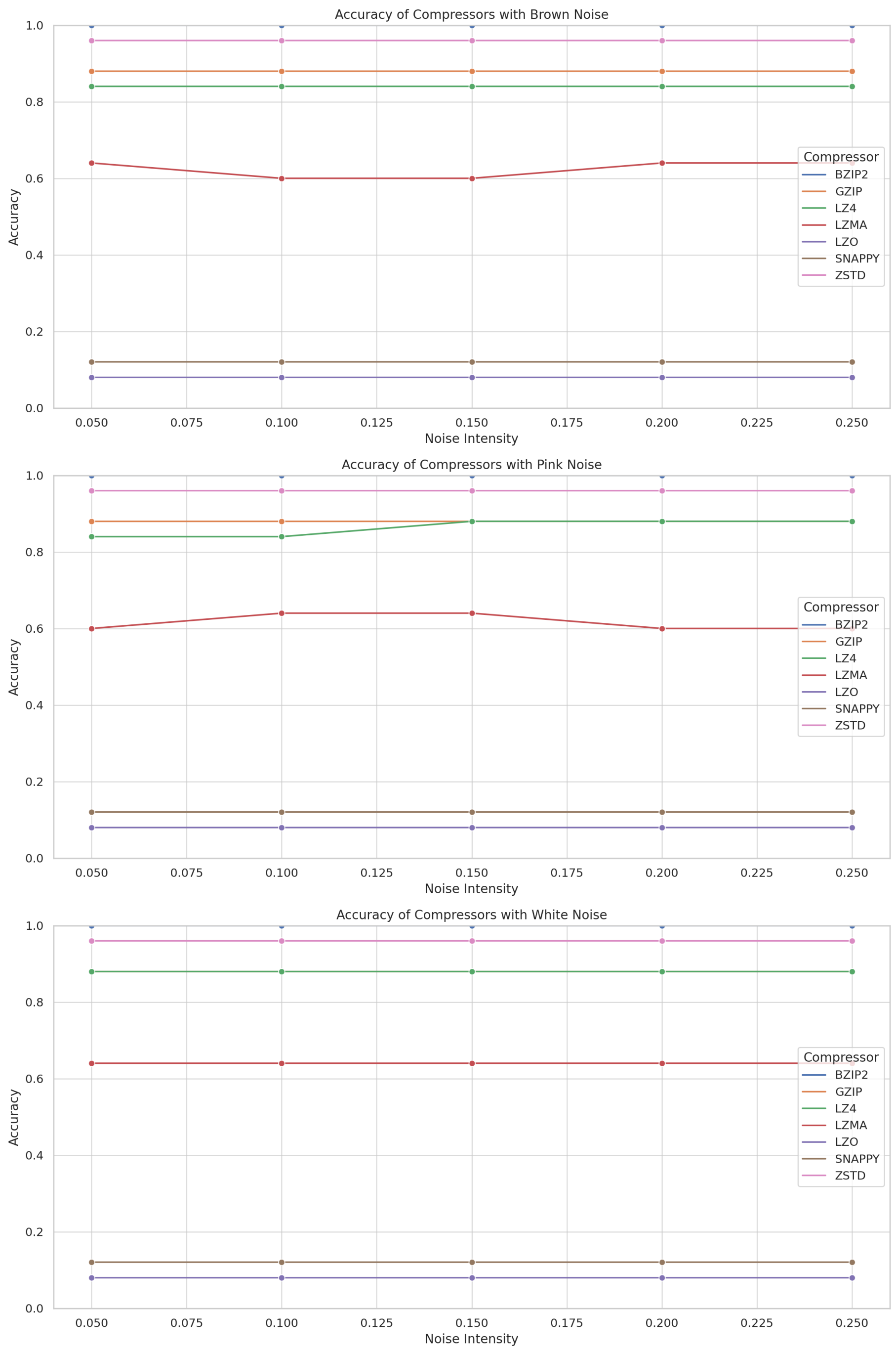


Figure 3: Accuracy per compressor over noise

5 Discussion

The findings in the previous section offer valuable insights into how different compressors perform under varying noise conditions. This section will analyze these results and explore their impact on the effectiveness and reliability of these compressors in processing music data.

5.1 Compressor Performance Analysis

The compressors' performance was evaluated based on their accuracy in handling test files with and without noise, including white, pink, and Brownian noise at various intensities. The results clearly show distinct differences in the multiple compressors' capabilities.

Bzip2 demonstrated superior performance, achieving 100% accuracy across all test files, regardless of the presence of noise. This highlights the robustness of *bzip2* in maintaining the integrity of music files even when subjected to different types of noise. The high accuracy suggests that *bzip2* can be relied upon for applications where preserving the quality of the audio data is crucial.

In contrast, *Snappy* and *LZO* exhibited significantly lower accuracy, achieving less than 20%. This apparent difference emphasizes their limited effectiveness in scenarios involving noise. The consistently lower distance values produced by these compressors further emphasize their inability to compress noisy audio signatures accurately.

5.2 Impact of Noise on Compressor Accuracy

Figure 3 illustrates the impact of different noise levels on each compressor's overall accuracy. Most compressor's performance remains constant. However, with Brownian noise, *LZMA*'s accuracy decreases slightly with minimal noise [0.1, 0.15] but returns to the same value. Interestingly enough, with Pink noise, the accuracy of the said compressor is flipped in values, starting at the decreased accuracy value and increasing a bit at the same interval. In contrast, with Brownian noise, it decreases and then goes back down again. The other compressor affected by Pink noise is *LZA*, which shows a slight accuracy increase with higher amounts of noise.

Bzip2's performance remains relatively stable even at higher noise intensities, reaffirming its robustness.

5.3 Implications and Future Work

Among the compressors tested, *bzip2* emerges as the most reliable choice for high-fidelity audio data compression applications. It excels in maintaining high accuracy across various noise types and intensities, making it well-suited for audio processing environments where data integrity is crucial.

On the other hand, the subpar performance of *Snappy* and *LZO* suggests that these compressors may not be suitable for high-fidelity audio applications, particularly in noisy environments.

Future studies should aim to experiment with larger datasets, particularly to investigate how different genres affect this approach. Experimenting with higher noise values would also help discover at what intensity of noise this approach is viable since the provided experimentation only used relatively small amounts of noise.

References

[1] Mark Adler. *zlib Usage Example.* URL: `https://zlib.net/zlib_how.html` (visited on 05/29/2024).

[2] Yann Collet. *LZ4 - Fast LZ compression algorithm.* URL: `https://android.googlesource.com/platform/external/lz4/+/83e749d/lz4.h` (visited on 06/01/2024).

[3] Yann Collet. *zstd - standard compression library.* URL: `https://chromium.googlesource.com/external/github.com/DataDog/zstd/+/a06e81606d158ee6e5bb4072e002e63fe10916ce/zstd.h` (visited on 05/29/2024).

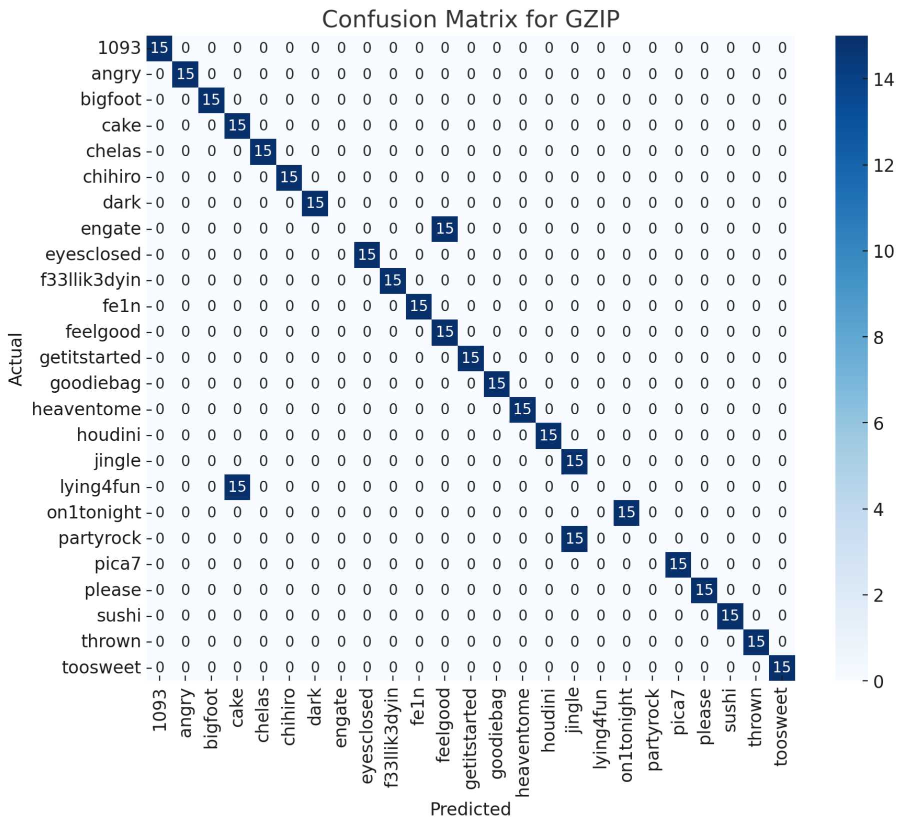
[4] Google Inc. *snappy.h.* URL: `https://chromium.googlesource.com/external/snappy/+/e72627753f7e70da00fc80e7174dc9e13f04d8ba/snappy.h` (visited on 06/06/2024).

[5] Markus Franz Xaver Johannes Oberhumer. *lzo1x.h – public interface of the LZO1X compression algorithm.* URL: `https://android.googlesource.com/platform/external/syslinux/+/refs/tags/android-cts-8.0_r16/lzo/include/lzo/lzo1x.h` (visited on 06/01/2024).

[6] Adam Sawicki. *LZMA SDK - How to Use*. URL: `https://www.asawicki.info/news_1368_lzma_sdk_-_how_to_use` (visited on 05/29/2024).

[7] Julian Seward. *Programming with libbzip2*. URL: `https://www.cs.cmu.edu/afs/cs/project/pscico-guyb/realworld/99/code/bzip2-0.9.5c/manual_3.html` (visited on 05/29/2024).

Confusion Matrix per Compressor



Actual \ Predicted	1093	angry	bigfoot	cake	chelas	chihiro	dark	engate	eyesclosed	f33llik3dyin	fe1n	feelgood	getitstarted	goodiebag	heaventome	houdini	jingle	lying4fun	on1tonight	partyrock	pica7	please	sushi	thrown	toosweet
1093	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
angry	0	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
bigfoot	0	0	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
cake	0	0	0	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
chelas	0	0	0	0	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
chihiro	0	0	0	0	0	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
dark	0	0	0	0	0	0	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
engate	0	0	0	0	0	0	0	0	0	0	0	15	0	0	0	0	0	0	0	0	0	0	0	0	0
eyesclosed	0	0	0	0	0	0	0	0	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
f33llik3dyin	0	0	0	0	0	0	0	0	0	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
fe1n	0	0	0	0	0	0	0	0	0	0	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0
feelgood	0	0	0	0	0	0	0	0	0	0	0	15	0	0	0	0	0	0	0	0	0	0	0	0	0
getitstarted	0	0	0	0	0	0	0	0	0	0	0	0	15	0	0	0	0	0	0	0	0	0	0	0	0
goodiebag	0	0	0	0	0	0	0	0	0	0	0	0	0	15	0	0	0	0	0	0	0	0	0	0	0
heaventome	0	0	0	0	0	0	0	0	0	0	0	0	0	0	15	0	0	0	0	0	0	0	0	0	0
houdini	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	15	0	0	0	0	0	0	0	0	0
jingle	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	15	0	0	0	0	0	0	0	0
lying4fun	0	0	0	15	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
on1tonight	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	15	0	0	0	0	0	0
partyrock	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	15	0	0	0	0	0	0	0	0
pica7	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	15	0	0	0	0
please	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	15	0	0	0
sushi	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	15	0	0
thrown	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	15	0
toosweet	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	15

Figure 4: GZIP Confusion Matrix

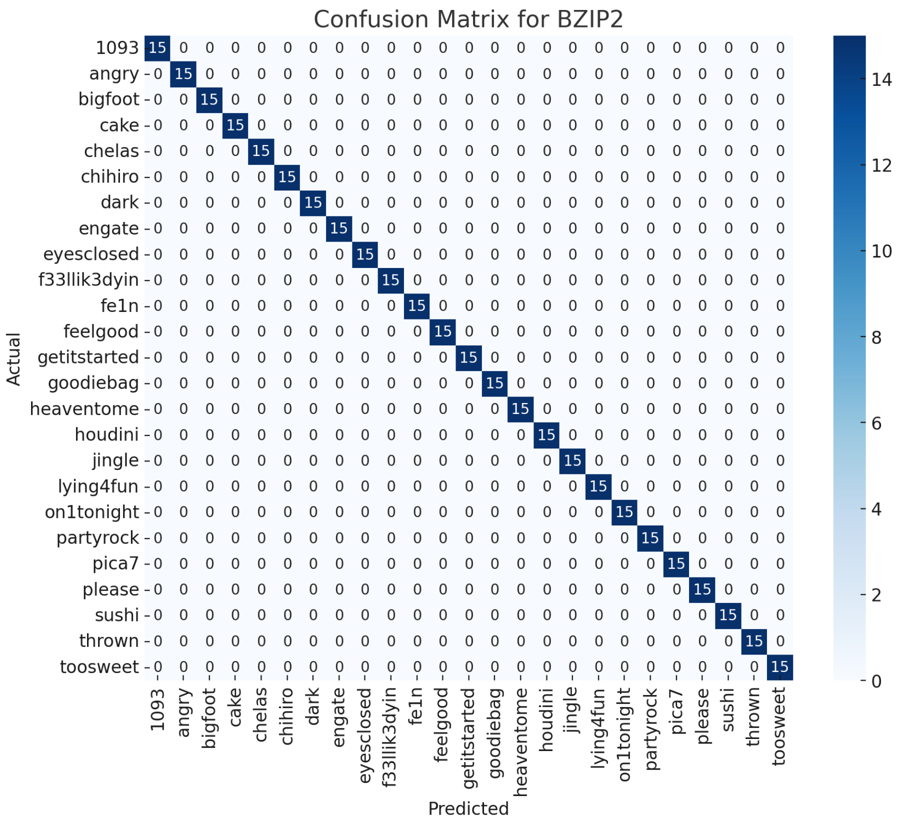


Figure 5: BZIP2 Confusion Matrix

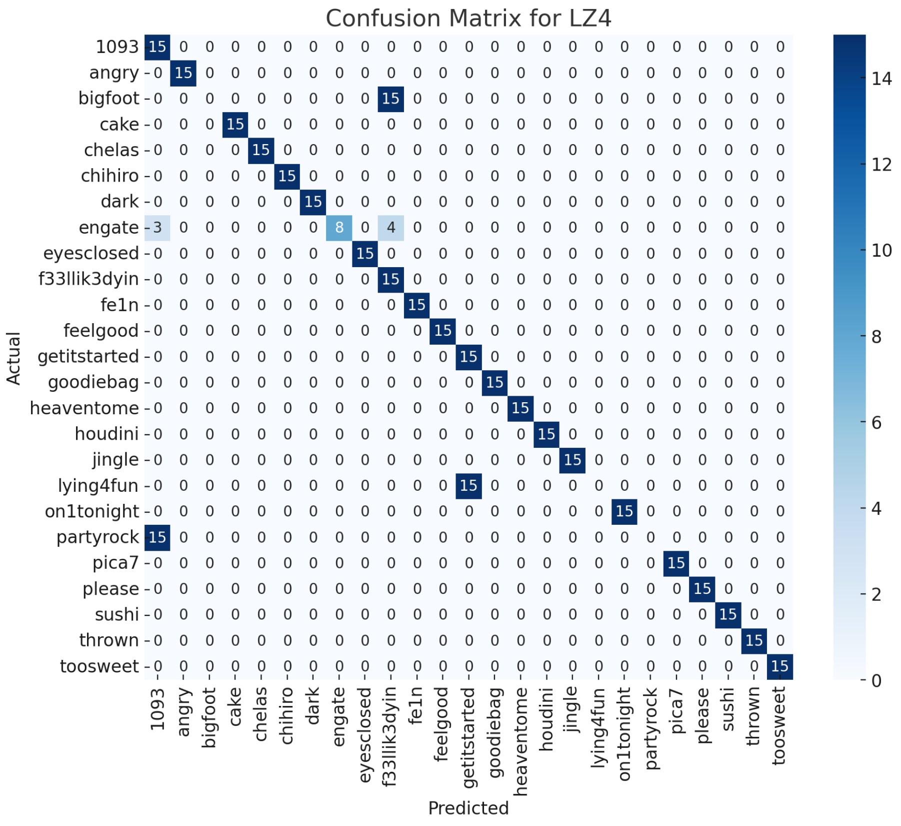


Figure 6: LZ4 Confusion Matrix

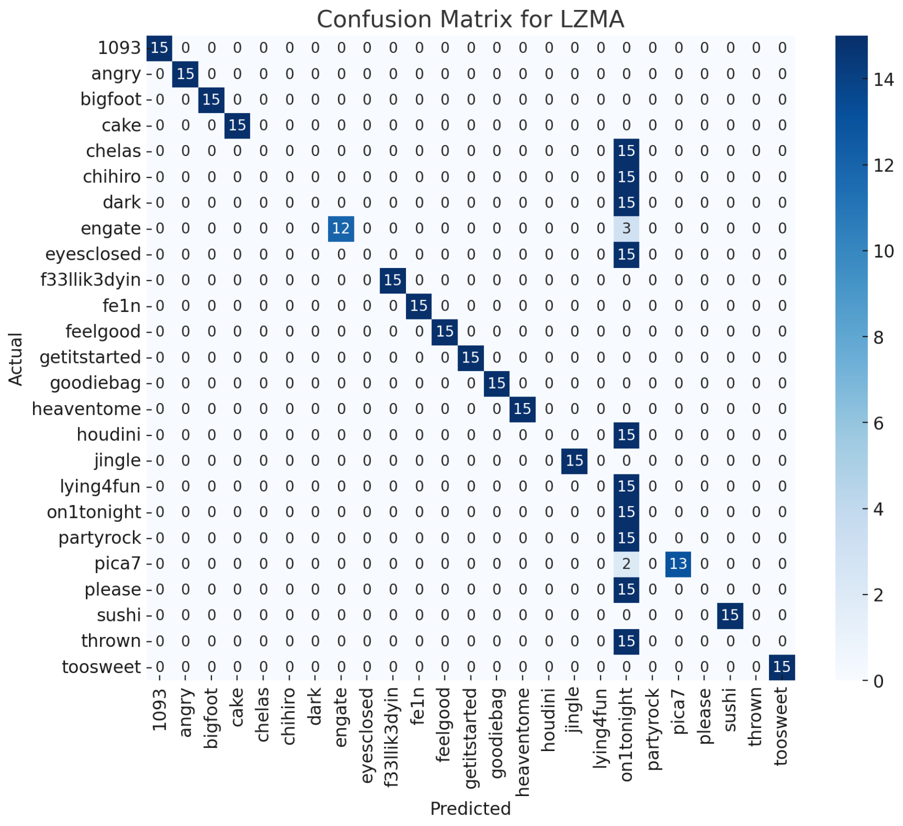


Figure 7: LZMA Confusion Matrix

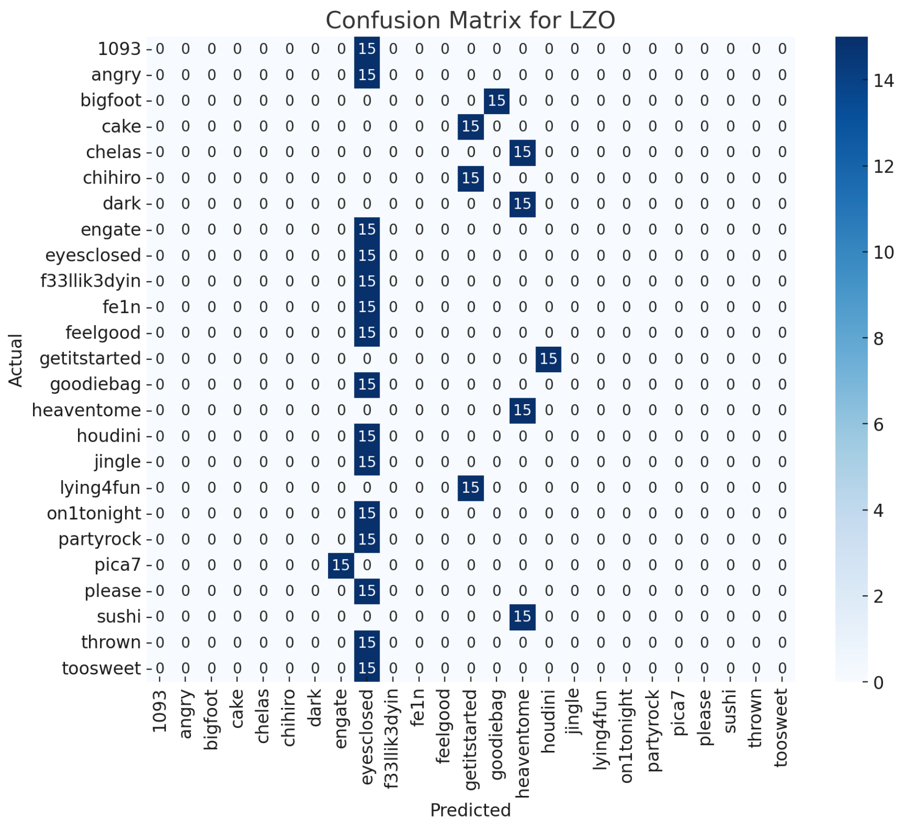


Figure 8: LZO Confusion Matrix

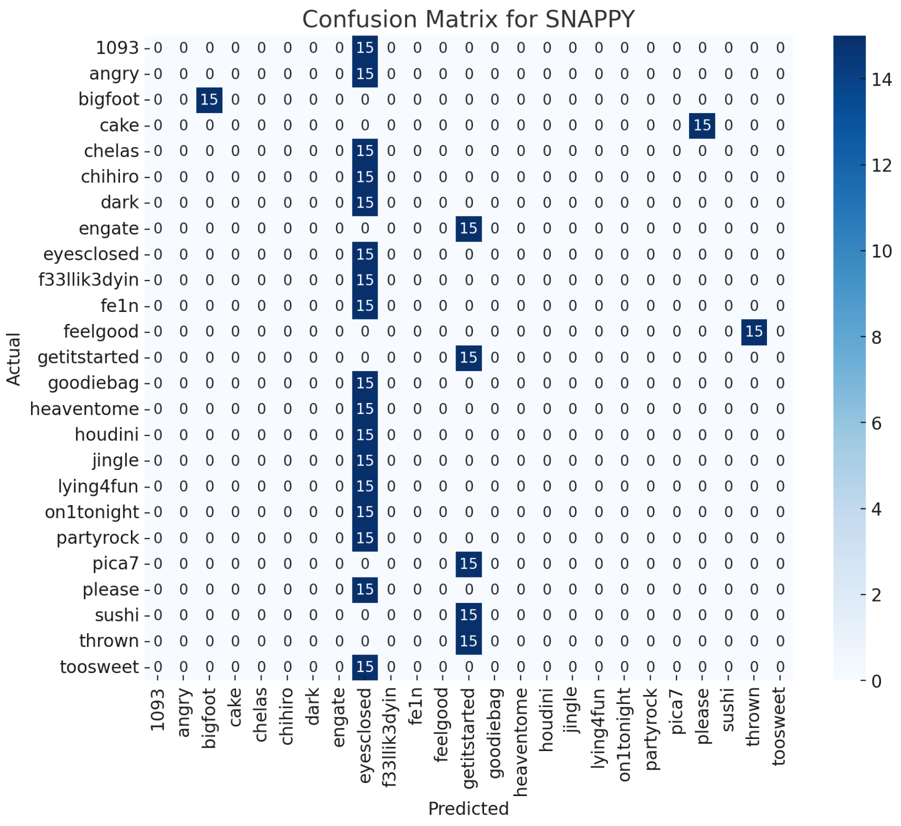


Figure 9: Snappy Confusion Matrix

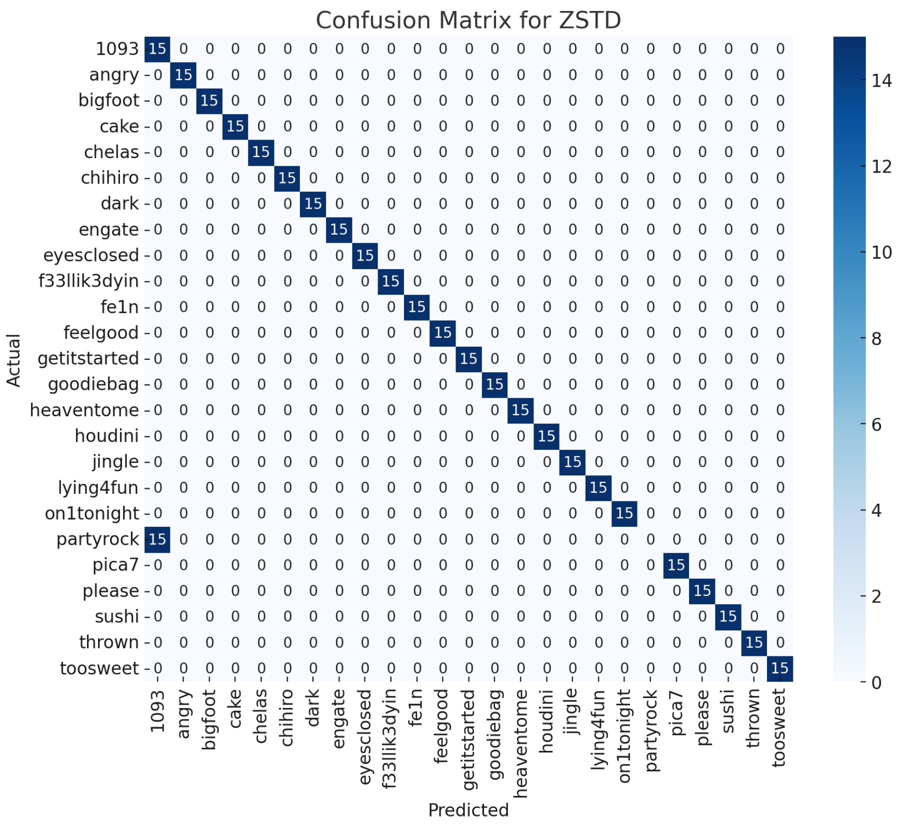


Figure 10: zstd Confusion Matrix