Comparing Zstd Compression With Industry Standards (Gzip, Bzip2, LZ4): Performance And Efficiency

Rohan Desai

March 25, 2025

Abstract

With terabytes of data flowing into financial institutions like Lloyds Banking Group daily, efficient compression is critical for reducing storage costs and optimising performance. This study evaluates Zstandard (Zstd) against industry-standard compression algorithms—Gzip, Bzip2, and LZ4—by analyzing their compression ratios, speeds, and resource utilization across diverse file types, including text, PDFs, videos, images, and audio. Our findings show that Zstd consistently outperforms in compression speed and memory usage while maintaining strong compression ratios, making it particularly effective for large-scale financial datasets. Compared to Gzip, Zstd achieved up to 9x less memory usage, and 64x faster compression than bzip2. LZ4 remains the fastest but sacrifices compression efficiency, while Bzip2 offers higher compression but at a significant speed and memory cost.

Contents

1 Introduction

1.1 Background

As digital data grows exponentially, efficient compression techniques have become essential for reducing storage costs, improving transmission speeds, and optimising computational resources. The foundation of modern data compression traces back to the late 1940s with the emergence of information theory, which paved the way for various algorithms to represent information more compactly without data loss.

There are two types of compression: lossy and lossless. Lossy compression reduces file size by deleting irrelevant data that may be unnoticeable to humans, making it ideal for media formats such as images, audio, and videos. However, this data loss is irreversible, making lossy compression unsuitable for applications with critical data integrity. Lossless compression, however, ensures that no information is lost during compression and decompression, making it the preferred choice for text files, financial records and other data-sensitive applications where data loss is unacceptable.

This experiment focuses solely on lossless compression and evaluates the performance of Zstandard (Zstd) against three industry-standard algorithms: Gzip, Bzip2, and LZ4 across a diverse set of file types, including structured financial data, plain text files with random characters, videos, music, images, and PDFs. Gzip balances compression efficiency and speed. Bzip2 provides higher compression ratios but at the cost of slower speed. LZ4 prioritises speed, making it ideal for real-time applications. Zstd, a newer algorithm, claims to provide high-speed and superior compression efficiency across various data types.

This experiment aims to determine the most effective lossless compression method for diverse datasets by benchmarking these algorithms across different file formats.

1.2 Motivation

As companies become increasingly data-focused, they accumulate vast amounts of information daily. Managing and storing this data efficiently is a growing challenge, requiring effective compression techniques to optimise storage, reduce costs, and improve processing speeds. While widely used, traditional compression algorithms like Gzip, Bzip2, and LZ4 may not always offer the best balance between compression ratio and speed, especially in large-scale, high-performance environments.

Zstandard (Zstd) has emerged as a strong contender, offering high-speed compression with competitive or superior efficiency compared to industry standards. Zstd could help businesses optimise their data infrastructure, leading to significant performance and cost advantages.

1.3 Objectives

This paper aims to evaluate the performance of Zstandard (Zstd) compared to traditional compression algorithms like Gzip, Bzip2, and LZ4. The focus will be on assessing each

algorithm's compression ratio, speed, and resource (CPU and Memory) usage across various data types. By conducting a detailed analysis, the paper seeks to provide insights into the strengths and weaknesses of these algorithms, helping to make informed decisions about which compression method to adopt for optimal performance and cost efficiency.

Objectives **:**

- To compare the compression ratios achieved by Zstd, Gzip, Bzip2, and LZ4 across various data types (e.g., text, PDFs, video, images, and audio).
- To measure and compare each algorithm's speed (compression and decompression time).
- To analyse the resource usage (e.g., CPU and memory) of Zstd and other compression algorithms during compression and decompression processes.
- To identify the trade-offs between compression efficiency and speed for different data types.

1.4 Paper Organization

This paper is organized into the following sections:

- **Section 1: Introduction** – Provides an overview of the study, introduces the motivation, and outlines the objectives of comparing Zstd with other compression algorithms.
- **Section 2: Background** – Provides foundational information on data compression, explaining key concepts and introducing the algorithms discussed in the study. This section also explains terms like compression ratio, speed, and resource utilization.
- **Section 3: Methodology** – Describes the experimental setup, including the hardware and software environment, the datasets used, and the specific compression algorithms evaluated in this study. It also outlines the performance metrics used for comparison.
- **Section 4: Results** – Presents the findings from the experiments, comparing the compression ratios, speed performance, and resource utilization of each compression algorithm.
- **Section 5: Analysis and Discussion** – Analyses the results, focusing on the strengths and weaknesses of each compression algorithm based on the experimental data. This section also discusses the trade-offs between compression speed and efficiency.
- **Section 6: Legal and Ethical Considerations** – Outlines the legal and ethical considerations in this experiment.

2 Background

This section provides essential background information related to data compression, including the fundamentals of compression, different types of algorithms, and an overview of the algorithms discussed in this paper.

2.1 Preliminaries

Some terms used elsewhere in this document are defined here for clarity.

- **Uncompressed**: Describes an arbitrary set of data in its original form, before being subjected to compression. [10]
- **Compressed**: Describes the result of passing a data set through a compression algorithm. The original input has thus been compressed. [10]
- **Decompressed**: Describes the result of passing a data set through the reverse compression algorithm. When this is successful, the decompressed payload and the uncompressed payload are indistinguishable. [10]
- **Compression Ratio**: The ratio of the original data size to the compressed data size, indicates how effectively the data has been compressed.
- **Compression Speed**: How quickly data can be compressed and decompressed, measured using MB/s.
- **Resource Utilisation**: The computational resources (e.g., CPU and memory) required for compression and decompression.

2.2 Compression Algorithms

Data compression is a process used to reduce data size by encoding it in a more efficient format. In the context of lossless compression, the objective is to compress data to restore it to its original state without any loss of information. This section provides an overview of the compression algorithms used in the experiment, explaining how each algorithm functions and the techniques it uses for compression.

2.2.1 Zstd (Zstandard)

Zstandard (Zstd) is a modern compression algorithm developed by Facebook. It is designed to achieve high compression ratios and fast compression and decompression speeds. Zstd uses a combination of techniques:

- **Dictionary Compression**: Zstd uses a pre-built dictionary of common patterns or data structures that repeat frequently in input data, allowing for more efficient encoding.
- **Huffman Coding**: It applies Huffman coding to the most frequently occurring sequences in the data to represent them with shorter codes, thus reducing the total size.

- **Finite State Entropy (FSE)**: Zstd utilizes a more advanced entropy coding method, the Finite State Entropy, which is designed to further improve compression efficiency.
- **Multi-threading**: Zstd supports parallel processing, which can utilize multiple cores for faster compression on multi-core systems.
- **Compression Levels**: Zstd provides adjustable compression levels that allow users to balance between speed and compression ratio. The compression level typically ranging from **-7** (fastest) to **22** (highest compression).

  Compression Level Categories:

  - **Negative Levels (-7 to -1):** Ultra-fast compression with minimal CPU usage. These levels provide lower compression ratios but are well-suited for real-time applications where speed is critical.
  - **Standard Levels (1 to 22):** The default compression level is **3**.
    - Lower levels (1-5): Faster compression with moderate compression ratios.
    - Mid-range levels (6-12): A balance between speed and efficiency.
    - Higher levels (13-22): Maximum compression, but significantly slower.
  - **Expert Mode (Levels above 22):** Using special configurations, Zstd can be tuned beyond level 22 for extreme compression, although it requires more memory and CPU resources.

2.2.2 Gzip

Gzip is one of the most widely used compression formats, especially for web data transmission and file compression. It uses the **DEFLATE** algorithm, which is a combination of two key techniques:

- **LZ77 (Lempel-Ziv, 1977)**: LZ77 is a dictionary-based algorithm that replaces repeated substrings in the data with references to earlier occurrences. It maintains a sliding window of data that helps identify these repetitions, resulting in data being stored more efficiently.
- **Huffman Coding**: After applying LZ77, Gzip uses Huffman coding, a lossless entropy coding technique that assigns shorter codes to more frequent symbols and longer codes to less frequent symbols.

2.2.3 Bzip2

Bzip2 is a compression algorithm that provides higher compression ratios than Gzip but at the cost of slower performance. It uses a combination of two main techniques:

- **Burrows-Wheeler Transform (BWT)**: The Burrows-Wheeler Transform is a reversible transformation of the input data that groups similar characters, improving the effectiveness of the subsequent compression steps. The BWT is followed by a move-to-front transform, which further helps group repeating characters.
- **Huffman Coding**: After applying the BWT and move-to-front transform, Bzip2 applies Huffman coding to the resulting data, reducing its size further.

2.2.4 LZ4

LZ4 is a high-speed compression algorithm that prioritizes speed over achieving the highest compression ratios. It uses a straightforward compression technique based on:

- **Lempel-Ziv (LZ77)**: Like Gzip, LZ4 uses LZ77 for compression. It finds and replaces repeated substrings with references to earlier occurrences of the same data. However, LZ4's implementation of LZ77 is simplified and optimized for speed rather than high compression ratios.
- **Dictionary Compression**: LZ4 utilizes a smaller sliding window (dictionary) compared to algorithms like Zstd, which helps it achieve its speed but may reduce the compression ratio in comparison.

3 Methodology

This section outlines the experimental setup and performance metrics used in the experiment.

3.1 Experimental Setup

This subsection provides details on the hardware specifications, software environment, and dataset selection used for the experiment.

3.1.1 Hardware Specifications

The experiments were conducted on a system with the following hardware specifications:

- **Laptop**: MacBook Air (M1, 2020)
- **Processor**: Apple M1 (8-core CPU, 7-core GPU, 3.2 GHz)
- **Memory**: 8GB Unified Memory
- **Storage**: 512GB SSD
- **Operating System**: macOS Sonoma 14.6.1

3.1.2 Software Environment

The following software tools and libraries were used in the experiment:

- IDE: Visual Studio Code, version 1.98.2 (Universal).
- Programming Language: The experiments were conducted using Python 3.11.8.
- Zstandard (Zstd): The Zstandard compression algorithm was used in the standard Python package, version 0.23.0.
- Gzip: The built-in zlib module in Python was used for Gzip compression, which implements the DEFLATE algorithm.
- Bzip2: The built-in bz2 module in Python was used for Bzip2 compression.

- LZ4: The LZ4 compression algorithm was implemented using the lz4 Python library, version 4.4.3. We used the default compression level (3).
- psutil: The psutil library was used to measure CPU utilization and memory usage during compression and decompression, version 7.0.0.
- time: The in-built library time was used to measure the time taken to compress and decompress the data.

3.1.3 Dataset Selection

The datasets selected for testing include various file types and sizes to evaluate the efficiency of each algorithm under different conditions. The datasets include:

- **Text Files**: Randomly generated text data. These files are created using a script that generates random alphanumeric characters and symbols.
- **Text Files**: Human readable text. Samples collected from [4].
- **PDFs**: Catalogs and Documentation. Samples collected from [6].
- **Images**: Nature (PNG, JPEG, TIFF). Samples collected from [5].
- **Videos**: Animated and Nature Videos(MP4, API). Samples collected from [8].
- **Audio Files**: Music (MP3). Samples collected from [1].
- **CSV**: BigQuery Data exported as CSV. Samples collected from [2].

The datasets were chosen to reflect real-world use cases, ensuring the results apply to practical scenarios.

3.1.4 Dataset Sizes

To effectively evaluate the algorithm's performance, we have selected a range of file sizes, which can be seen in Table 1.

File Type	Small	Medium	Large	Extra Large
Text (Randomly generated with mixed characters)	10 KB	1 MB	100 MB	1 GB
Text (Human readable)	10 KB	100 KB	1 MB	10 MB
PDFs (Reports, Documents)	100 KB	1 MB	5 MB	10 MB
Images (JPG)	100 KB	800 KB	1.4 MB	17.5 MB
Videos (MP4, API)	10 MB	62 MB	127 MB	670MB
Audio (MP3)	100 KB	1 MB	5 MB	12 MB
CSV (Exported from BigQuery)	110 KB	8.3 MB	70 MB	680 MB

Table 1: File size categories for different data types used in testing.

A study analysing **49 million files** found that most text files remain under **8 MB**, with increasing trends in larger datasets [9].

3.1.5 Number of Files Tested

The number of files tested for each file type and size includes:

File Type	Small	Medium	Large	Extra Large
Text (Randomly generated with mixed characters)	10 files	10 files	10 files	10 files
Text (Human readable)	1 file	1 file	1 file	1 file
PDFs (Reports, Documents)	1 file	1 file	1 file	1 file
Images (JPG, PNG, TIFF)	1 file	1 file	1 file	1 file
Videos (MP4, MOV)	1 file	1 file	1 file	1 file
Audio (MP3, WAV)	1 file	1 file	1 file	1 file
CSV (Financial Logs, Market Data)	1 file	1 file	1 file	1 file

Table 2: Number of files tested for different data types and sizes.

Due to having a random text file generator, we are able to have access to multiple random text files, whereas, for the other file types, we deemed it sufficient to have only one sample for each file type and size needed.

3.1.6 Dataset Generation

To ensure a controlled and reproducible benchmarking environment, we generated random text files using a Python script. The script allows for the specification of the number of files and their sizes in megabytes (MB). Each file consists of randomly selected alphabets, digits, punctuation, and whitespace.

The dataset creation process involves:

- Generating a user-defined number of files.
- Each file contains a mixture of characters including letters, digits, punctuation, and spaces.
- Allowing for fine-tuned control over file sizes, including fractional MB sizes.

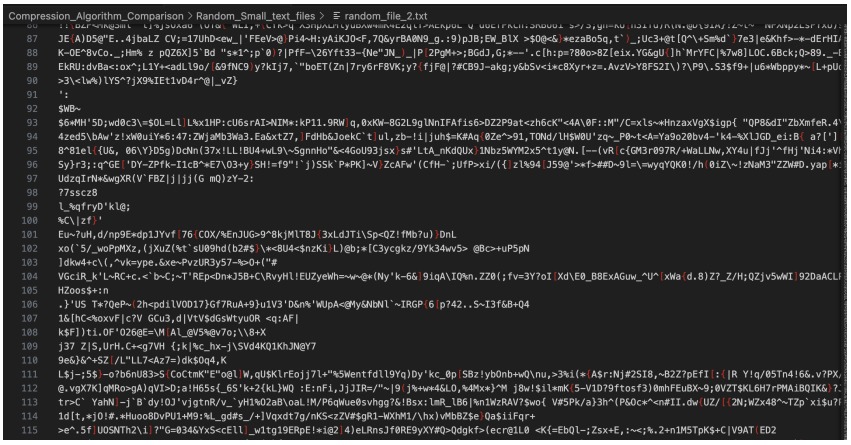

Figure 1: Example of generated random text files in the dataset folder.

3.2 Performance Metrics

To evaluate the efficiency of the compression algorithms, the following performance metrics will be considered:

- **Compression Ratio**: The compression ratio is calculated using the formula:

$$\text{Compression Ratio} = \frac{\text{Original File Size}}{\text{Compressed File Size}} \tag{1}$$

  A higher compression ratio indicates better data reduction.

- **Compression Speed**: Compression speed refers to the time taken by an algorithm to compress a given dataset. It will be measured in megabytes per second (MB/s) by dividing the original file size by the time taken for compression.

- **Decompression Speed**: Similar to compression speed, decompression speed measures how quickly an algorithm can restore the original data from its compressed form. It will be recorded in MB/s.

- **Memory Usage**: The memory (RAM) consumed during both compression and decompression processes will be monitored.

- **CPU Utilization**: CPU utilization will be measured to determine how much processing power each compression algorithm consumes during execution. This metric will be recorded as the percentage of total CPU capacity used during compression and decompression.

3.3 Statistical Analysis

To ensure a comprehensive evaluation of the algorithms, we analyse multiple statistical metrics across different performance factors. The following metrics are used:

- **Mean (μ)**: The average value of a given metric, calculated as:

$$\mu = \frac{1}{N} \sum_{i=1}^{N} x_i \tag{2}$$

  where x_i represents an individual measurement, and N is the total number of measurements.

- **Standard Deviation (σ)**: Measures the spread of values around the mean:

$$\sigma = \sqrt{\frac{1}{N} \sum_{i=1}^{N} (x_i - \mu)^2} \tag{3}$$

  A higher standard deviation indicates greater variability in the performance metric.

- **Range**: The difference between the maximum and minimum values in the dataset:

$$\text{Range} = x_{\max} - x_{\min} \tag{4}$$

4 Results

This section presents the results of the experiment conducted. The SD and Range are provided in the results appendix. It is important to note that Standard Deviation and Range are only available for text file types as more than 1 sample was tested for each size.

4.1 Compression Ratio

Algorithm	Small	Medium	Large	Extra Large	Algorithm	Small	Medium	Large	Extra Large
Randomly Generated Text Files					**Human Readable Text Files**				
gzip	1.1981	1.1992	1.1992	1.1992	gzip	2.1377	2.6584	2.7873	2.8018
bzip2	1.1825	1.2021	1.2022	1.2022	bzip2	2.3736	3.5588	4.0677	4.1319
lz4	0.9977	0.9999	0.9999	0.9999	lz4	1.4073	1.6194	1.6693	1.6745
zstd	1.1991	1.2012	1.2013	1.2013	zstd	2.1093	2.6203	2.7697	2.7688
CSV Files					**PDF Files**				
gzip	2.2966	3.3609	19.6142	4.7799	gzip	1.0925	1.1202	1.0412	16.4058
bzip2	3.0583	4.1184	32.8532	6.0261	bzip2	1.0824	1.1240	1.0403	28.8369
lz4	1.3883	1.8462	12.7139	2.9194	lz4	1.0735	1.1060	1.0318	11.1768
zstd	2.2544	3.5209	20.1928	4.6817	zstd	1.0833	1.1260	1.0437	22.4261
Image Files					**Video Files**				
gzip	1.0062	1.0019	1.0021	1.0023	gzip	1.0018	0.9998	1.0100	1.0068
bzip2	0.9995	0.9974	0.9982	1.0042	bzip2	1.0008	0.9957	1.0058	1.0123
lz4	1.0019	1.0002	1.0001	1.0000	lz4	1.0007	1.0000	1.0101	0.9999
zstd	0.9999	1.0000	1.0000	1.0009	zstd	1.0010	1.0001	1.0103	1.0000
Audio Files									
gzip	1.0430	1.0137	1.0455	1.0064					
bzip2	1.0432	1.0135	1.0471	1.0054					
lz4	1.0278	1.0003	1.0270	1.0004					
zstd	1.0461	1.0026	1.0335	1.0063					

Table 3: Mean Compression Ratios for Various File Types

4.2 Compression Time

Algorithm	Small	Medium	Large	Extra Large	Algorithm	Small	Medium	Large	Extra Large
Text (Randomly Generated)					**Text (Human Readable)**				
gzip	0.0001	0.0235	2.3544	24.4392	gzip	0.0002	0.0031	0.0325	0.3259
bzip2	0.0009	0.0589	6.0040	63.6730	bzip2	0.0009	0.0052	0.0525	0.5204
lz4	0.0000	0.0003	0.0532	1.6075	lz4	0.0000	0.0003	0.0024	0.0245
zstd	0.0000	0.0009	0.0959	0.9999	zstd	0.0001	0.0006	0.0045	0.0430
PDF Files					**Image Files**				
gzip	0.0019	0.0276	0.0974	0.0748	gzip	0.0018	0.0186	0.0285	0.4224
bzip2	0.0071	0.1091	0.3685	0.8699	bzip2	0.0078	0.0696	0.1072	1.3103
lz4	0.0001	0.0006	0.0024	0.0052	lz4	0.0000	0.0002	0.0004	0.0067
zstd	0.0002	0.0023	0.0044	0.0061	zstd	0.0001	0.0008	0.0009	0.0126
Video Files					**Audio Files**				
gzip	0.2061	1.1674	2.3523	16.0686	gzip	0.0022	0.0279	0.1339	0.2808
bzip2	0.8728	4.9955	10.0004	48.9425	bzip2	0.0088	0.0803	0.3914	0.9473
lz4	0.0043	0.0242	0.0648	0.4653	lz4	0.0001	0.0004	0.0031	0.0055
zstd	0.0062	0.0158	0.0359	0.7015	zstd	0.0006	0.0022	0.0129	0.0181
CSV Files									
gzip	0.0030	0.2089	0.4567	14.9583					
bzip2	0.0059	0.8165	5.5321	35.3825					
lz4	0.0002	0.0114	0.0349	1.0601					
zstd	0.0007	0.0273	0.0509	1.7694					

Table 4: Mean Compression Times for Various File Types

4.3 Decompression Time

Algorithm	Small	Medium	Large	Extra Large	Algorithm	Small	Medium	Large	Extra Large
Text (Randomly Generated)					**Text (Human Readable)**				
gzip	0.0000	0.0024	0.2467	3.3145	gzip	0.0000	0.0001	0.0011	0.0111
bzip2	0.0004	0.0359	3.8493	38.5726	bzip2	0.0003	0.0020	0.0186	0.1832
lz4	0.0000	0.0001	0.0302	1.1029	lz4	0.0000	0.0000	0.0004	0.0041
zstd	0.0000	0.0010	0.1081	1.2395	zstd	0.0000	0.0002	0.0010	0.0108
PDF Files					**Image Files**				
gzip	0.0003	0.0026	0.0113	0.0045	gzip	0.0001	0.0008	0.0013	0.0235
bzip2	0.0034	0.0503	0.1767	0.1094	bzip2	0.0035	0.0338	0.0510	0.6488
lz4	0.0000	0.0002	0.0009	0.0024	lz4	0.0000	0.0000	0.0001	0.0025
zstd	0.0000	0.0002	0.0007	0.0029	zstd	0.0000	0.0001	0.0001	0.0024
Video Files					**Audio Files**				
gzip	0.0079	0.0143	0.0489	1.8565	gzip	0.0003	0.0026	0.0131	0.0296
bzip2	0.4034	2.4206	4.8370	26.0902	bzip2	0.0041	0.0394	0.2155	0.4694
lz4	0.0016	0.0093	0.0291	0.3856	lz4	0.0000	0.0001	0.0016	0.0021
zstd	0.0017	0.0051	0.0165	0.0733	zstd	0.0000	0.0002	0.0012	0.0014
CSV Files									
gzip	0.0002	0.0092	0.0403	1.0716					
bzip2	0.0024	0.1492	0.5509	9.8307					
lz4	0.0000	0.0033	0.0225	0.6106					
zstd	0.0002	0.0068	0.0208	0.5348					

Table 5: Mean Decompression Times for Various File Types

4.4 CPU Usage

Algorithm	Small	Medium	Large	Extra Large	Algorithm	Small	Medium	Large	Extra Large
Text (Randomly Generated) Files					**Text (Human Readable) Files**				
gzip	100.0420	98.9470	99.5000	97.3300	gzip	99.6400	99.5300	99.9400	99.9400
bzip2	99.2490	99.7470	98.9500	95.7800	bzip2	100.0400	99.8600	99.9400	99.9200
lz4	98.9560	99.7670	83.1200	64.6000	lz4	99.3400	93.4100	99.9800	99.8100
zstd	99.5510	99.3400	97.4300	95.0700	zstd	100.1500	95.0800	99.7900	99.8600
PDF Files					**Image Files**				
gzip	99.6400	99.8800	99.7200	99.8600	gzip	99.9300	99.0700	99.9300	94.7500
bzip2	99.8500	99.9100	99.3000	99.9900	bzip2	99.8300	99.8000	99.9900	99.8400
lz4	99.1400	100.0500	99.9200	100.0000	lz4	100.4900	99.9700	100.0200	99.7800
zstd	100.0800	99.3200	99.9200	99.9900	zstd	99.7500	97.4200	99.5200	98.8100
Video Files					**Audio Files**				
gzip	99.8333	99.8667	99.3333	98.5667	gzip	99.9000	99.2600	99.4800	99.7900
bzip2	98.8000	100.0000	99.8000	99.3333	bzip2	97.6300	99.8900	99.5400	98.6300
lz4	99.5667	100.0333	87.8667	75.7333	lz4	100.3000	99.9200	95.7000	98.0800
zstd	85.9333	100.0333	95.6667	99.8667	zstd	99.8300	99.6200	98.2900	98.8300
CSV Files									
gzip	100.0000	99.8000	99.9000	99.8000					
bzip2	99.7000	100.0000	99.3000	99.6000					
lz4	98.2000	100.1000	97.4000	100.0000					
zstd	99.5000	100.0000	95.3000	100.0000					

Table 6: Mean CPU Utilization for Various File Types

4.5 Memory Usage

Algorithm	Small	Medium	Large	Extra Large	Algorithm	Small	Medium	Large	Extra Large
Text (Randomly Generated) Files					**Text (Human Readable) Files**				
gzip	0.0003	0.1325	70.7969	395.0109	gzip	0.0172	0.1187	0.0766	0.0719
bzip2	0.0239	0.3814	22.6219	1813.2125	bzip2	0.0031	0.4203	1.6641	3.2422
lz4	0.0009	0.0156	107.0594	1376.2016	lz4	0.0000	0.0078	0.0000	0.0031
zstd	0.0036	0.0239	42.9828	81.8734	zstd	0.0000	0.0547	0.0000	1.5625
PDF Files					**Image Files**				
gzip	0.0000	0.8141	0.1859	0.0000	gzip	0.0016	0.6969	1.3328	5.7812
bzip2	0.0000	2.0906	1.6938	0.0000	bzip2	0.0688	1.8906	0.7703	0.2188
lz4	0.0000	0.0078	0.0422	0.0000	lz4	0.0000	0.0078	0.0000	0.3312
zstd	0.0000	0.0484	0.0828	0.0938	zstd	0.0000	0.0375	0.4391	0.5938
Video Files					**Audio Files**				
gzip	8.3698	79.5833	118.5417	564.9792	gzip	0.0250	0.2953	0.2453	3.4750
bzip2	13.0573	44.6302	60.6667	1164.0208	bzip2	0.3547	1.0141	2.5844	3.1062
lz4	3.3385	20.5601	77.0573	286.8333	lz4	0.0187	0.0000	0.0484	0.5609
zstd	1.4635	10.0052	40.1198	100.7656	zstd	0.0469	0.0000	1.0063	1.5594
CSV Files									
gzip	0.4844	4.4844	6.9844	714.9844					
bzip2	1.0469	10.9531	7.6250	766.9688					
lz4	0.0000	0.5312	10.6094	225.4375					
zstd	0.6562	2.8125	3.8438	83.7188					

Table 7: Mean Memory Usage for Various File Types

5 Analysis and Discussion

In this section we analyse and discuss the results of the compression study across various algorithms (gzip, bzip2, lz4, zstd) and file types (randomly generated text, human-readable text, CSV, PDF, image, video, audio).

5.1 Compression Ratio

- **Randomly Generated Text Files:** Randomly generated text files have minimal structure and low redundancy, and compression algorithms tend to produce only modest reductions in file size. The lack of patterns in the data makes it challenging for compression algorithms to achieve significant compression.
  - The compression ratios for `gzip`, `bzip2`, `lz4`, and `zstd` are relatively close, with `zstd` slightly outperforming others in terms of compression efficiency.
  - `lz4` consistently shows the least compression, suggesting it's not designed for high compression efficiency but rather optimised for speed.
- **Human Readable Text Files:** Human-readable text files typically have more structure and redundancy than randomly generated files. This added redundancy allows compression algorithms to achieve better results.
  - `bzip2` and `zstd` provide higher compression ratios for human-readable text files, as they are better suited for compressing structured and highly repetitive text.
  - `gzip` and `lz4` show lower ratios, which may be attributed to their less aggressive compression algorithms compared to `bzip2` and `zstd`.
- **CSV Files:** CSV files, being structured tabular data, exhibit higher redundancy than free-form text files.
  - The compression ratios are fairly similar for `gzip` and `bzip2`, with `bzip2` still holding a slight advantage in terms of compression, particularly for larger datasets.
  - `lz4` performs poorly in this case due to its focus on speed rather than compression efficiency.
- **PDF Files:** PDF files, being binary formats with embedded images and text, have characteristics that make them more challenging to compress.
  - `gzip` and `zstd` perform comparably, while `bzip2` slightly outperforms them, offering a better compression ratio.
  - `lz4` again shows subpar performance in terms of compression efficiency for PDF files.
- **Image, Video, and Audio Files:**
  - For media files, the compression ratios for `gzip`, `bzip2`, and `zstd` are generally low, as these file types are already compressed in their native formats (e.g., JPEG, PNG, MP4).

– `lz4` still shows the least compression for these files, but its speed advantage might make it suitable for specific use cases where compression time is a priority.

5.2 Compression Time

- **`gzip`** shows a consistent increase in compression time as the file size grows. For small files, compression times are very short, but as file sizes increase, especially for large and extra-large files, the time taken increases significantly, making it less efficient for very large datasets.

- **`bzip2`** has the highest compression time across all file types. The algorithm is known for its higher compression ratios, but it requires more computational resources and time. For instance, compressing large randomly generated text files takes 6.0040 seconds, while extra-large files require up to 63.6730 seconds. The compression time increases drastically with the file size.

- **`lz4`** stands out for its speed, with compression times for large files averaging only 1.6075 seconds for randomly generated text files and 0.4653 seconds for video files. `lz4` is designed for applications where fast compression is required, often at the expense of the compression ratio. This makes it ideal for real-time applications or scenarios where speed is more important than achieving the best possible compression.

- **`zstd`** performs similarly to `gzip`, but is generally faster, especially for larger files. Compression times for randomly generated text files, for example, grow from 0.0000 seconds for small files to 0.9999 seconds for extra-large files. While its compression speed is relatively fast, `zstd` offers a balance between compression ratio and speed, making it a versatile option for various use cases.

Decompression Time

- **`gzip`** is generally fast in decompression. The times are very short for smaller files, and the decompression time increases steadily as the file size grows. For extra-large files, decompression can take up to 3.3145 seconds for human-readable text files. Despite this, `gzip` remains one of the most commonly used algorithms due to its balance between compression ratio and decompression speed.

- **`bzip2`** takes significantly longer to decompress compared to `gzip`. For instance, decompression of randomly generated text files takes 0.0359 seconds for medium-sized files and 38.5726 seconds for extra-large files. This performance is indicative of the more complex algorithm used by `bzip2`, which results in higher compression ratios but at the cost of slower decompression times.

- **`lz4`** is designed for very fast decompression. It consistently outperforms the other algorithms. This makes `lz4` an excellent choice for scenarios where decompression speed is critical.

- **`zstd`** also offers fast decompression, similar to `lz4`. For example, decompression

of randomly generated text files takes 0.0010 seconds for medium-sized files and only 1.2395 seconds for extra-large files. Like `lz4`, `zstd` provides a strong balance of speed and compression ratio, making it suitable for applications that require both efficiency and speed.

5.3 CPU Usage

CPU utilization measures the percentage of processing power consumed by each compression algorithm while compressing files of different sizes and formats.

`gzip`: The CPU usage for `gzip` remains consistently high across all file types and sizes, with utilization ranging between 97% and 100%. This indicates that `gzip` is highly optimized for CPU usage but does not significantly scale down its resource consumption for smaller files. Notably, for extra-large files, the CPU usage slightly decreases, possibly due to I/O bottlenecks.

`bzip2`: This algorithm also maintains high CPU utilization, averaging close to 100% for most file types. However, there is a noticeable dip for randomly generated text files (95.78% for extra-large files), indicating that the algorithm may experience performance degradation due to its more complex compression technique.

`lz4`: Unlike the previous two algorithms, `lz4` demonstrates significantly lower CPU utilization, especially for large and extra-large files. For example, the CPU usage drops to 64.60% for extra-large randomly generated text files and 75.73% for extra-large video files. This suggests that `lz4` prioritizes speed over CPU resource consumption.

`zstd`: `zstd` maintains CPU utilization similar to `gzip` and `bzip2`, with utilization rates around 95%-100%. However, for certain file types (e.g., video files), `zstd` shows a dip in CPU usage, reflecting its ability to dynamically adjust processing requirements based on file content and compression level.

5.4 Memory Usage

Memory consumption is another crucial factor, particularly when dealing with large datasets. Different algorithms have varying memory footprints depending on file size and type.

`gzip`: The memory footprint for `gzip` is generally low for small and medium files but increases significantly for large and extra-large files. For example, the memory usage jumps from 0.1325 MB for medium randomly generated text files to 395.01 MB for extra-large files. This trend is seen across most file types, suggesting that `gzip` requires more memory as file sizes increase.

`bzip2`: `bzip2` exhibits high memory usage, particularly for extra-large randomly generated text files, reaching up to 1813.21 MB. This high memory requirement is expected given `bzip2`'s intensive compression techniques that involve multiple passes over the data.

`lz4`: In contrast, `lz4` demonstrates minimal memory consumption for smaller files, but

for larger files, memory usage increases substantially. For example, randomly generated text files require 1376.20 MB for extra-large files, though the memory footprint remains lower compared to `bzip2`.

`zstd`: `zstd` offers a more balanced approach, maintaining relatively low memory usage across different file types. Even for extra-large files, `zstd` consumes significantly less memory than `bzip2`, making it an efficient choice for environments where memory is a constraint.

5.5 Choosing the Right Algorithm

- `gzip` achieves compression ratios similar to `zstd` on average, though still lower than `bzip2`. It is the second-slowest algorithm across all file types and consumes the second-highest amount of memory among the four algorithms. If achieving a high compression ratio is important while maintaining a moderate priority on time and memory usage, this algorithm is a suitable choice.

- `bzip2` provides the highest compression ratios of all the algorithms but is also the slowest and most memory-intensive. If maximum compression is the primary goal and time or memory constraints are not a concern, this is the preferred algorithm.

- `lz4` is the fastest algorithm but offers the lowest compression ratio. It was ineffective for most file types except for human-readable text and CSV files, where it still performed the worst in terms of compression. However, it ranked second-best in memory usage on average. If speed is the top priority and compression ratio is less important, this is the ideal algorithm.

- `zstd` provides a well-balanced trade-off between compression ratio and speed. It achieves compression ratios within 10–30% of `bzip2` while being up to 10 times faster. Additionally, it decompresses up to 30 times faster than `bzip2` and has the lowest memory usage among all the algorithms compared. This makes it a versatile choice for applications requiring both efficiency and speed.

Conclusions

- `bzip2` achieves the highest compression ratio, outperforming `gzip` by up to 47% and `lz4` by over 140% on human-readable text files. However, its compression time is significantly slower—up to 3.5 times slower than `gzip` and more than 35 times slower than `lz4` for large files.

- `gzip` provides a good balance between speed and compression, compressing up to 40% faster than `bzip2` while still achieving a competitive compression ratio. However, it lags behind `bzip2` in compression efficiency by 10–47%, depending on file type.

- `lz4` is the fastest in both compression and decompression, outperforming `gzip` by up to 40x and `bzip2` by over 100x in compression speed for large text files. However, it delivers the lowest compression ratio, achieving up to 60% less compression than `bzip2`.

- `zstd` strikes a balance between compression ratio and speed, offering a compression ratio within 10–30% of `bzip2` while being up to 10x faster. It also decompresses up to 30x faster than `bzip2`, making it a versatile choice for scenarios requiring both efficiency and speed.

5.6 Limitations and Future Work

5.6.1 Limitations

Despite the thorough experimentation, several limitations were encountered:

- **Hardware Constraints:** The experiments were conducted on a single machine with limited resources (8GB RAM, Apple M1). This may affect the generalisability of results to more powerful systems or distributed environments.
- **Single System Environment:** All testing was performed on a MacBook Air with macOS, which might influence performance results due to platform-specific optimisations and limitations. Cross-platform testing could provide a more comprehensive evaluation.
- **Compression Algorithms Tested:** While the study tested popular algorithms like Gzip, Bzip2, and LZ4, emerging algorithms like Brotli or Zstandard were not included. These may offer competitive performance in modern use cases.
- **File Sizes and Count:** Although a wide range of file sizes was tested, the number of files per dataset type was limited. A more extensive testing set with varied file numbers would provide a better understanding of algorithm performance.

5.6.2 Future Work

Future work should address the following areas:

- **Platform Diversity:** Conduct tests across different platforms (Linux, Windows) and hardware configurations to assess the generalisability of results.
- **Additional Compression Algorithms:** Include newer compression algorithms such as Brotli, LZ4HC, or Zstandard to compare their performance against traditional algorithms. Specifically, evaluate different Zstandard compression levels to balance speed and compression ratio effectively.
- **Real-World Scenarios:** Test the algorithms with real-world datasets from industries like healthcare or finance, where data characteristics and size can significantly differ.
- **Compression with Encryption:** Investigate the performance of compression algorithms when combined with encryption algorithms, a common practice for secure data storage.
- **Large-Scale Testing:** Conduct large-scale testing with more datasets and file types, including massive files and high-concurrency scenarios, to measure scalability and real-time performance.

6 Legal and Ethical Considerations

This section assesses the possible legal, social, ethical, and professional implications of this experiment and explains how this experiment follows the British Computing Society's (BCS) Code of Conduct [3] [1].

6.1 Legal Issues

This experiment generated its test data, ensuring full control over the data creation process. Additionally, any external data used in the study was sourced from reputable online platforms, making it compliant with the General Data Protection Regulation (GDPR).

There are no legal or licensing concerns related to the compression algorithms used in this experiment. Zstd, Gzip, Bzip2, and LZ4 are open-source algorithms, and their use is governed by licenses that allow for free use in academic and commercial contexts. The software libraries used, including Python and its relevant modules, are licensed under the PSF License [7]. The PSF License allows for free usage, distribution, and even commercial use of Python [7].

6.2 Social Issues

This experiment was designed to be accessible and transparent. We provide clear documentation and explanations for the data compression algorithms (Zstd, Gzip, Bzip2, LZ4) and how they are applied to real-world data types, ensuring that the findings are reproducible.

6.3 Ethical Issues

This project aims to ensure fairness in the evaluation of different compression algorithms, avoiding biases in both the selection of data types and the evaluation criteria. We have used diverse datasets, including randomly generated text files and multimedia files, to ensure that the results are representative of real-world scenarios and are not skewed by any particular data type.

Finally, this project did not involve the use of humans or animals.

6.4 British Computing Society's Code of Conduct

The entire project has been conducted in accordance with the British Computing Code of Conduct [3]

The findings presented in this project accurately represent the observed results without misrepresentation, according to the code of conduct 3.e.

[1] https://www.bcs.org/media/2211/bcs-code-of-conduct.pdf

References

[1] audio samples. Accessed: 2025-03-21.

[2] Bq to csv samples. Accessed: 2025-03-21.

[3] Code of conduct for bcs members. Accessed: 2025-03-20.

[4] human text samples. Accessed: 2025-03-23.

[5] images samples. Accessed: 2025-03-21.

[6] Pdf samples. Accessed: 2025-03-21.

[7] Python-license. Accessed: 2025-03-20.

[8] video samples. Accessed: 2025-03-21.

[9] Jesse David Dinneen and Ba Xuan Nguyen. How big are peoples' computer files? file size distributions among user-managed collections. *Proceedings of the Association for Information Science and Technology*, 58(1):425–429, 2021.

[10] Collet Yann and Kucherawy Murray. Zstandard compression and the application/zstd media type. *No. rfc8478*, 2018.

Appendices

A Dataset Generation Code

The following Python script was used to generate the dataset for compression testing:

Listing 1: Python script for dataset generation

```
import os
import random
import string

def generate_random_files(folder_path, num_files, size_mb):
    """Generates multiple random text files in the specified folder."""
    os.makedirs(folder_path, exist_ok=True)  # Create the folder if it does
    characters = string.ascii_letters + string.digits + string.punctuation

    bytes_per_mb = 1_000_000  # Define 1MB as 1,000,000 bytes

    for i in range(1, num_files + 1):
        file_path = os.path.join(folder_path, f"random_file_{i}.txt")

        with open(file_path, 'w') as f:
            full_mb = int(size_mb)  # Whole number MB
            partial_mb = int((size_mb - full_mb) * bytes_per_mb)
# Partial MB in bytes

            # Write full MB chunks
            for _ in range(full_mb):
                f.write(''.join(random.choices(characters, k=bytes_per_mb))

            # Write remaining bytes for decimal portion
            if partial_mb > 0:
                f.write(''.join(random.choices(characters, k=partial_mb)))

        print(f"Generated {file_path} ({size_mb:.2f} MB)")

# User inputs
folder = "/Users/rohandesai/Documents/Github desktop projects/Educational-
num_files = int(input("How many files do you want? "))
size_mb = float(input("Size of each file (MB)? "))  # Allows decimal values

generate_random_files(folder, num_files, size_mb)
```

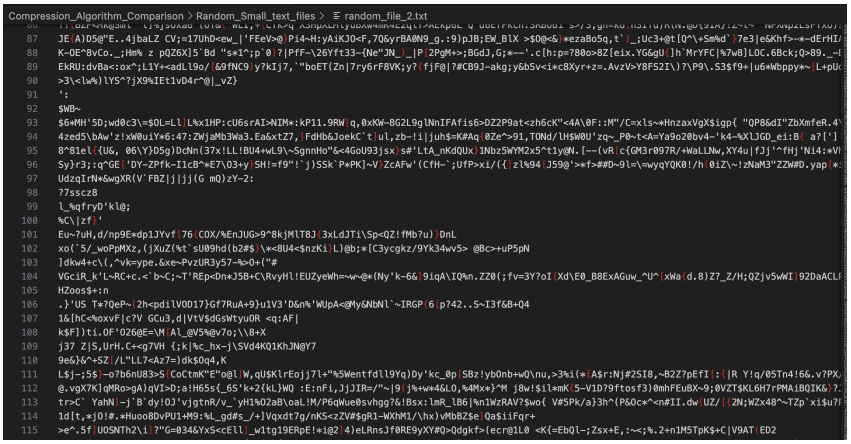

Figure 2: Example Of Randomly Generated Text File

B Results

It is important to note that Standard Deviation and Range are only available for text file types as more than 1 sample was tested for each size.

B.1 Compression Ratio

Algorithm	Small	Medium	Large	Extra Large
gzip	0.0005	0.0004	0.0005	0.0007
bzip2	0.0009	0.0003	0.0005	0.0006
lz4	0.0000	0.0000	0.0000	0.0000
zstd	0.0006	0.00006	0.0007	0.0008

Table 8: Standard Deviation (SD) of Compression Ratios For Text File Type

Algorithm	Small	Medium	Large	Extra Large
gzip	0.0016	0.0002	0.0001	0.0003
bzip2	0.0028	0.0004	0.0001	0.0005
lz4	0.0000	0.0000	0.0000	0.0000
zstd	0.0020	0.0002	0.0003	0.0007

Table 9: Range of Compression Ratios For Text File Type

B.2 Compression Time

Algorithm	Small	Medium	Large	Extra Large
gzip	0.0000	0.0050	0.0559	2.1835
bzip2	0.0001	0.0014	0.1141	5.4716
lz4	0.0001	0.0000	0.0056	0.3663
zstd	0.0002	0.0001	0.0156	0.1538

Table 10: Standard Deviation (SD) of Compression Time For Text File Type

Algorithm	Small	Medium	Large	Extra Large
gzip	0.0000	0.0491	0.1694	7.3406
bzip2	0.0006	0.0061	0.3297	17.6291
lz4	0.0007	0.0002	0.0175	1.2401
zstd	0.0017	0.0012	0.0512	0.5064

Table 11: Range of Compression Time For Text File Type

B.3 Decompression Time

Algorithm	Small	Medium	Large	Extra Large
gzip	0.0000	0.0002	0.0091	0.1710
bzip2	0.0000	0.0015	0.7305	2.9685
lz4	0.0000	0.0000	0.0091	0.3948
zstd	0.0000	0.0001	0.0056	0.3379

Table 12: Standard Deviation (SD) of Decompression Time For Text File Type

Algorithm	Small	Medium	Large	Extra Large
gzip	0.0000	0.0018	0.0236	0.5398
bzip2	0.0002	0.0071	2.3691	9.3923
lz4	0.0000	0.0001	0.0337	1.3272
zstd	0.0004	0.0005	0.0183	0.9249

Table 13: Range of Decompression Time For Text File Type

B.4 CPU Usage

Algorithm	Small	Medium	Large	Extra Large
gzip	0.1296	5.3690	0.5538	5.3402
bzip2	2.2597	0.3815	0.9372	5.0334
lz4	7.9602	1.2435	4.4746	10.6722
zstd	8.1090	4.5427	6.9723	9.1130

Table 14: Standard Deviation (SD) of CPU Utilization For Text File Type

Algorithm	Small	Medium	Large	Extra Large
gzip	1.4000	51.8000	1.6000	17.7000
bzip2	16.4000	2.4000	2.8000	15.3000
lz4	79.0000	8.6000	14.5000	26.6000
zstd	81.6000	44.4000	22.4000	29.3000

Table 15: Range of CPU Utilization For Text File Type

B.5 Memory Usage

Algorithm	Small	Medium	Large	Extra Large
gzip	0.0031	1.0281	109.4554	639.7218
bzip2	0.0832	2.8852	102.9966	226.0946
lz4	0.0066	0.1014	52.3360	663.1827
zstd	0.0243	0.1571	14.8867	608.0458

Table 16: Standard Deviation of Memory Usage For Text File Type

Algorithm	Small	Medium	Large	Extra Large
gzip	0.0312	8.9844	384.2500	1692.9219
bzip2	0.4688	21.5781	344.4219	716.4688
lz4	0.0625	0.9375	147.9531	1167.5625
zstd	0.1875	1.4688	44.8750	171.0781

Table 17: Range of Memory Usage For Text File Type

C Code For Compression Performance Evaluation

The following Python code is used to measure the performance of various compression algorithms, including gzip, bzip2, lz4, and zstd. It evaluates compression time, decompression time, memory usage, CPU utilization, and compression ratio for each algorithm on different file sizes.

Listing 2: Python code for measuring compression and decompression performance.

```
import time
import os
import psutil
import zlib
import bz2
import lz4.frame
import zstandard as zstd
import statistics
import csv

# Number of iterations per file
NUM_RUNS = 10

# Define compression algorithms
ALGORITHMS = {
    "gzip": lambda data: zlib.compress(data),
    "bzip2": lambda data: bz2.compress(data),
    "lz4": lambda data: lz4.frame.compress(data),
    "zstd": lambda data: zstd.ZstdCompressor().compress(data),
} # add level = in the zstd.ZstdCompressor() to set the level or by defaul

# Define decompression algorithms
DECOMPRESSION = {
    "gzip": lambda data: zlib.decompress(data),
    "bzip2": lambda data: bz2.decompress(data),
    "lz4": lambda data: lz4.frame.decompress(data),
    "zstd": lambda data: zstd.ZstdDecompressor().decompress(data),
}

# Define file size categories based on filename patterns
SIZE_CATEGORIES = {
    "small": "Small",
    "medium": "Medium",
    "xl": "XL",
    "large": "Large",
}

# Directory to store compressed files
COMPRESSED_FOLDER = "Path/to/folder"

def detect_file_size(filename):
```

```
    """Determines file size category based on filename."""
    filename_lower = filename.lower()
    if 'xl' in filename_lower:
        return 'XL'
    for keyword, category in SIZE_CATEGORIES.items():
        if keyword in filename_lower:
            return category
    return None

def ensure_directory_exists(directory):
    """Ensures a directory exists; creates it if not."""
    if not os.path.exists(directory):
        os.makedirs(directory)

def measure_performance(file_path):
    """Measures compression and decompression performance for a single file
    results = {algo: {"Compression Ratio": [], "Compression Time (s)": [],
                      "Decompression Time (s)": [], "Memory Usage (MB)": []
                      "CPU Utilization (%)": []} for algo in ALGORITHMS}

    with open(file_path, "rb") as f:
        data = f.read()

    original_size = len(data)

    for algo, compress_func in ALGORITHMS.items():

        for run in range(NUM_RUNS):

            process = psutil.Process(os.getpid())
            start_cpu = process.cpu_percent(interval=None)
            start_mem = process.memory_info().rss
            start_time = time.time()

            compressed_data = compress_func(data)

            end_time = time.time()
            end_cpu = process.cpu_percent(interval=None)
            end_mem = process.memory_info().rss

            results[algo]["Compression Time (s)"].append(end_time - start_t
            results[algo]["Compression Ratio"].append(original_size / len(c
            results[algo]["Memory Usage (MB)"].append((end_mem - start_mem)
# Convert to MB
            results[algo]["CPU Utilization (%)"].append(end_cpu - start_cpu
# CPU usage percentage

            # Measure decompression time
```

```
            start_time = time.time()
            decompressed_data = DECOMPRESSION[algo](compressed_data)
            end_time = time.time()
            results[algo]["Decompression Time (s)"].append(end_time - start

            # Verify integrity
            assert decompressed_data == data, f"Decompression failed for {a

        # Store compressed file after last run
        algo_folder = os.path.join(COMPRESSED_FOLDER, algo)
        ensure_directory_exists(algo_folder)
        compressed_file_path = os.path.join(algo_folder, os.path.basename(f

        with open(compressed_file_path, "wb") as f:
            f.write(compressed_data)

        # Take the mean of 10 runs
        for metric in results[algo]:
            results[algo][metric] = statistics.mean(results[algo][metric])

    return results

def process_folder(folder_path):
    """Processes files and groups results by size categories."""
    all_results = {size: {algo: {"Compression Ratio": [], "Compression Time
                                 "Decompression Time (s)": [], "Memory Usag
                                 "CPU Utilization (%)": []} for algo in ALG
                   for size in SIZE_CATEGORIES.values()}

    for file_name in os.listdir(folder_path):
        file_path = os.path.join(folder_path, file_name)

        if os.path.isfile(file_path):
            file_size_category = detect_file_size(file_name)

            if file_size_category:
                print(f"Processing {file_name} ({file_size_category})...")
                file_results = measure_performance(file_path)

                for algo in file_results:
                    for metric in file_results[algo]:
                        all_results[file_size_category][algo][metric].appen

    return all_results

def calculate_mean(results):
    """Calculates mean for each metric."""
    stats = {}
```

```
    for size, algos in results.items():
        stats[size] = {}
        for algo, metrics in algos.items():
            stats[size][algo] = {}
            for metric, values in metrics.items():
                if values:
                    stats[size][algo][metric] = statistics.mean(values)
                else:
                    stats[size][algo][metric] = 0  # Handle empty lists
    return stats

def save_results_to_csv(stats, filename="compression_results.csv"):
    """Saves the statistical results to a CSV file."""
    with open(filename, mode='w', newline='') as file:
        writer = csv.writer(file)
        header = ["Algorithm", "File Size", "Compression Ratio", "Compressi
                  "Decompression Time (s)", "Memory Usage (MB)", "CPU Utili
        writer.writerow(header)

        for size in stats:
            for algo in ALGORITHMS:
                writer.writerow([
                    algo,
                    size,
                    f"{stats[size][algo].get('Compression Ratio', 'N/A'):.4
                    f"{stats[size][algo].get('Compression Time (s)', 'N/A'
                    f"{stats[size][algo].get('Decompression Time (s)', 'N/A
                    f"{stats[size][algo].get('Memory Usage (MB)', 'N/A'):.4
                    f"{stats[size][algo].get('CPU Utilization (%)', 'N/A'):
                ])

    print(f"Results saved to {filename}")

if __name__ == "__main__":
    folder_path = "Path/to/folder"

    ensure_directory_exists(COMPRESSED_FOLDER)

    results = process_folder(folder_path)
    stats = calculate_mean(results)

    save_results_to_csv(stats)
```