Estimating File Compressibility Using File Extensions

A THESIS
SUBMITTED TO THE FACULTY OF THE GRADUATE SCHOOL
OF THE UNIVERSITY OF MINNESOTA
BY

Carson Powers

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
MASTER OF SCIENCE

Dr. Peter A. H. Peterson

July 2021

Acknowledgements

First and foremost, I would like to thank my family for their constant support and encouragement, especially during my adult life. Your years of relentless insistence that I am capable of achieving a bachelor's degree eventually paid off, and that has led me farther than I ever thought I could go.

I am also deeply grateful to my advisor, Dr. Peter Peterson, for all the insight and guidance he provided me throughout my undergraduate and graduate programs, and for inspiring me to continue research after my time at UMD. He is the reason why I chose this topic for a thesis and why I found an interest in data compression.

Lastly, I would like to thank Alex Bandt for being a warm and amicable classmate during so many of our undergraduate CS courses. With a friend at one's side, the most grueling classes and daunting projects feel like minor obligations. I'm glad I could know you for at least a short time, and I hope you have finally found peace.

Abstract

Adaptive compression systems dynamically choose a compression strategy — including no compression — by monitoring CPU usage, output rate, expected time to compress, and perhaps most importantly, the estimated compressibility of the data. Many adaptive compression systems were designed with the assumption that files with the same filename extension will compress roughly to the mean compression ratio (the ratio of compressed size to original size) of some set of files with the same extension. This implies that the compression ratio distribution follows a normal distribution. Though a normal distribution of compression ratios may seem intuitive, this assumption lacks strong empirical supporting evidence. To test this assumption, we built a tool to compress real-world files from many participants, storing the compressed size, original size, file extension, and other metadata. The results of three tests for normality indicate that none of the file extensions we analyzed have a normal distribution, though for some extensions, not all three tests agree. Furthermore, quantitative analysis reveals that files with the same extension compress according to multiple different distributions, and we identified some readily accessible metadata that can separate these files into simpler distributions. We conclude with a discussion of the utility of mean compressibility as an estimator and the implications this study has for future research in adaptive compression.

Contents

Contents iii

List of Tables vii

List of Figures ix

1 Introduction 1

2 Background 3

2.1 Overview 3

2.2 Common Compression Methods 5

2.2.1 Deflate 7

2.2.2 LZ4 & LZ4HC 8

2.2.3 LZMA/XZ 9

2.3 Adaptive Compression 9

2.3.1 Estimating Compressibility 10

2.3.2 File Extension as an Estimator 11

2.3.3 Assumption of Normality 12

2.4 Previous Work 13

2.4.1 Adaptive Compression for the Zettabyte File System 13

2.4.2 Ares 13
2.4.3 Efficient Compressibility Estimators 14
2.4.4 Datacomp 15
2.4.5 Statistical Measures as Predictors of Compression Savings . . 16

3 Experiment 17

3.1 Comprestimator 17
3.1.1 Compressors 17
3.1.2 Data Source 19
3.1.3 Algorithm 20
3.2 Methodology 22
3.2.1 Plotting 22
3.2.2 Compression Ratio 23
3.2.3 Normality Testing 23
3.2.4 Extension Selection Criteria 26

4 Results 27

4.1 File Formats 28
4.2 Results by Extension 30
4.2.1 Null Extension 30
4.2.2 BIN 32
4.2.3 DLL 33
4.2.4 DOCX 35
4.2.5 EXE 39
4.2.6 GIF 41
4.2.7 GZ 44
4.2.8 HTML 45

4.2.9 JPG 46
4.2.10 JS 48
4.2.11 JSON 49
4.2.12 MP3 51
4.2.13 PDF 53
4.2.14 PNG 54
4.2.15 RTF 56
4.2.16 SVG 58
4.2.17 TTF 59
4.2.18 TXT 61
4.2.19 WAV 63
4.3 Comparison among File Formats 67
4.3.1 Compressed File Formats 69
4.3.2 Source Code Formats 71

5 Discussion **73**
5.1 Normality 73
5.2 File Sizes 74
5.3 Mean as an Estimator 75
5.4 Mode as an Estimator 76
5.5 File Extension as a Predictor of File Type 77
5.6 Future Work 78
5.6.1 Participant Diversity 79
5.6.2 Time Estimation 79
5.6.3 Implementation in Real-World AC 80

6 Conclusion **81**

A Normality Test Results **82**

B Extensions Not Considered **116**

B.1 H, HPP, C, PY, PYC, CPP, JAVA, GO, LUA 116

B.2 TFM, TEX, VF 116

B.3 DAT 117

B.3.1 STRINGS, NIB, MANIFEST, KO, PM, MUM, MUI, etc. 117

C Rsync Compression Skip List **118**

References **119**

List of Tables

4.1 Total Files and Hours of Compression Per User 28

4.2 Most Common File Extensions Observed 29

4.3 Most Common File Extensions by Ranked Vote 30

4.4 Null Extension 31

4.5 BIN 32

4.6 DLL 34

4.7 DOCX 38

4.8 EXE 39

4.9 Average CR and Number of EXEs per Participant 41

4.10 GIF 42

4.11 GZ 44

4.12 HTML 45

4.13 JPG 47

4.14 JS 48

4.15 JSON 50

4.16 MP3 51

4.17 PDF 52

4.18 Individual PDF versions using Deflate 6 54

4.19 PNG 55
4.20 RTF 56
4.21 SVG 58
4.22 TTF 60
4.23 TXT 61
4.24 WAV 63
4.25 Compressed Format Performance with Deflate 6 69
4.26 Source Code Files with Deflate 6 71

A.1 Normal Test Results 82

C.1 File Extensions that Rsync Does Not Compress 118

List of Figures

4.1 Histogram for Null Extension 31
4.2 Histogram for BIN 33
4.3 Histogram for DLL 34
4.4 Histogram for DLL containing “key” in file metadata 36
4.5 Histogram for DLL without “key” in metadata 36
4.6 Histogram for DLL containing “key,” no limit on user contribution . . 37
4.7 Histogram for DLL without “key,” no limit on user contribution . . . 37
4.8 Histogram for DOCX 38
4.9 Histogram for EXE 39
4.10 Histogram for EXE containing compression keywords 41
4.11 Histogram for EXE containing compression keywords, no user limit . 42
4.12 Histogram for GIF 43
4.13 Histogram for GZ 44
4.14 Histogram for HTML 46
4.15 Histogram for JPG 47
4.16 Histogram for JS 49
4.17 Histogram for JSON 50
4.18 Histogram for MP3 52

4.19 Histogram for PDF 53

4.20 Histogram for PDF versions 1.0 to 1.4 using Deflate 6 55

4.21 Histogram for PNG 56

4.22 Histogram for RTF 57

4.23 Histogram for RTF 58

4.24 Histogram for SVG 59

4.25 Histogram for TTF 60

4.26 Histogram for TXT 62

4.27 Histogram for WAV 64

4.28 WAV audio, 8-bit mono 65

4.29 WAV audio, 16-bit mono 66

4.30 WAV audio, 8-bit stereo 66

4.31 WAV audio, PCM samples 68

4.32 WAV audio, ADPCM 68

4.33 Compressed Format Performance with Deflate 6 70

4.34 Source Code Files with Deflate 6 72

1 Introduction

Modern technological advances demand massive data collection schemes, and all of that data must be transported and stored. Beyond data collection for the noble causes of science, our society collects huge amounts of data for marketing, entertainment, and other efforts. The Internet Data Corporation estimated that in 2020 alone, 59 zettabytes of data would be generated, copied, and consumed [1]. By the year 2025, the amount of data stored worldwide is expected to reach 200 zettabytes [2]. To keep up with data storage requirements, computer systems must find ways to maximize the amount of information that can be stored on a medium. Reducing required storage space additionally reduces energy consumption, and with the growing threat of climate change, we have a moral duty to reduce consumption where possible. The simplest solution to maximizing data storage efficiency is data compression.

The efficiency of data compression is completely dependent on the data itself. Whenever characteristics of the data change, we may need to adjust our compression strategy. The field of Adaptive Compression (AC) creates systems that dynamically identify the best compression method for a data stream. One critical metric that an AC system needs to make decisions is an estimate of the file's compressibility. One such way of estimating is to use the file's extension.

AC systems that use file extensions to estimate compressibility must use some metric, e.g., the mean, median, or mode of a file extension's compressibility, as the estimator. If we know that file compressibility is normally distributed, these three values will be equal. We can then build an extension lookup table for an AC system to make accurate and timely compression decisions. If files with a particular extension do not compress normally, we do not know which metric, if any, will accurately predict compressibility. For example, if a particular file extension compresses according to a bimodal distribution, then the file extension will likely be a poor predictor.

To date, there has been no study to test the assumption that compressibility is normally distributed. Moreover, if a file extension does compress normally, we do not know the mean compressibility that one would use in a lookup table. Understanding the relationship between a file extension and compressibility will enable the field of adaptive compression to design future algorithms based on empirically-tested results.

In this study, we compressed a wide variety of files on multiple systems using multiple strength levels of three common compression algorithms. The results of three commonly used tests for normality provide strong evidence that none of the extensions we analyzed are associated with a normal distribution. We then calculated the mean, median, and mode (or highest mode, if the distribution appears multimodal) to make claims about using the mean or mode as estimators. For distributions that appear multimodal, we investigated whether some other readily accessible metadata (e.g., versions of a file format) can separate the data into unimodal distributions. We conclude with a discussion of the implications for future AC research.

2 Background

2.1 Overview

Much of the information we store contains redundancy. Though redundancy is occasionally beneficial, as in error-correcting codes, transmitting and storing redundant data is also wasteful. By removing redundancies from a data stream, we can represent the same information using fewer symbols. The process of converting a data stream into a smaller data stream is known as *compression*. The simplest form of compression is to drop imperceptible or insignificant bits from the data, but this technique will always lose some amount of information. Any compression method that loses information is called *lossy* compression. When we compress and then decompress a stream with a lossy method, we can never reconstruct an identical copy of the original stream. For some data, it is unacceptable to lose any information; we must be able to reconstruct an exact copy from the compressed stream. Any method that preserves all information is called *lossless* compression.

Lossy compression can discard any amount of information in the encoding process, which results in distortion when the data is decompressed. In contrast, lossless compression can only compress a stream to a lower bound — the most efficient representation of the information — beyond which, any further compression would necessarily cause information loss. This relies on a rather intuitive principle: every losslessly compressed file has exactly one decompressed representation. If two unique files F_1 and F_2 compressed to the same file C, the decompressor cannot know whether

F_1 or F_2 created C. Now, if we tried to create a lossless compressor that could reduce every file to no more than n bits, the compressor could only produce

$$\sum_{i=0}^{n} 2^n$$

possible compressed files, restricting the decompressor into producing a finite number of decompressed representations.

In 1948, Claude Shannon proved that the lower bound to which a file can be compressed is its entropy [3]. If we treat a file as a random variable X, then the entropy $H(X)$ is

$$H(X) = -\sum_{i} P(x_i) \log_2 P(x_i)$$

Though we can easily calculate the entropy of a file, in practice, it may be difficult to achieve this ideal level of compression. To achieve optimal compression for a file, we must exhaustively search for patterns in the data. Searching for increasingly longer patterns will eventually provide diminishing gains in space savings.

At its core, data compression is a trade-off between time and space. To achieve better compression of a data stream, a system must spend greater time compressing the data. Conversely, reducing the time spent compressing a stream will likely result in less compression. Though sometimes we may value the extreme optimization of either space or time, most applications desire a "happy medium" between both dimensions. A plethora of compression algorithms have been designed over the decades to match the varying applications that utilize compression.

All lossless compression algorithms have a complementary decompression algorithm, which may have very different run time compared to the compressor. Intuitively, compression usually takes more time than decompression. Searching for exploitable patterns in the data requires more effort than following instructions to rebuild the original, just as unpacking a car is much easier than finding the best way to pack it. When either encoding or decoding is significantly faster or less resource intensive than the other, we call that compression *asymmetrical* [4]. Asymmetrical compression is an attractive option for applications that do not compress and decompress with the same frequency. For example, some data may be compressed once and accessed very frequently. In this case, we may be willing to spend greater resources compressing the data while still enjoying rapid decompression later.

2.2 Common Compression Methods

Most general-purpose lossless compression methods fall into two categories: dictionary or statistical. Dictionary methods use fixed-size codes to represent variable-length strings of characters. The code lookup table, called the dictionary, specifies the string that each code represents. The modern era of dictionary-based compression began in 1977 when Jacob Ziv and Abraham Lempel developed what is now known as the LZ77 algorithm [5]. LZ77 used a sliding window composed of a search buffer and a look-ahead buffer, which essentially used the most recently seen strings as the dictionary. The following year, they created a method that replaced the sliding window with an external dictionary of previously seen strings [6]. Today, most dictionary-based encoding schemes are variations on either of these two groundbreaking algorithms.

Statistical methods generally use variable-length codes, assigning shorter codes to the most probable symbols. To use a statistical method, we must know the probability model of our source alphabet, or we at least need a good estimate. This model may be permanent, or it may be modified during the encoding process.

Huffman Coding

Perhaps the most famous statistical encoding method is Huffman Coding, a prefix code designed by MIT student David Huffman in 1952 [7]. Huffman designed his code based on two observations: first, symbols occurring with higher probability should have shorter codes, and second, the two least probable symbols should have codes of the same length. Using these two principles, he specified a bottom-up approach to generate an optimal code:

1. List all symbols s_i in descending order of probability.

2. Combine the two least probable symbols s_{n-1}, s_n into a new symbol α_1. The codes for s_{n-1} and s_n will be $\alpha_1 \circ 0$ and $\alpha_1 \circ 1$, where $\circ$ denotes concatenation.

3. Insert α_1 into the list, with $P(\alpha_1) = P(s_{n-1}) + P(s_n)$.

4. Continue from step 2 until only two symbols remain, which should be s_1 and α_{n-1}. Assign 1 to one symbol and 0 to the other.

The output will always be an optimal and uniquely decodable prefix code, though it is not guaranteed to be optimal for all possible compression methods. Huffman codes will leave zero redundancy when all symbols occur with some probability that is a negative power of two, but their performance decreases when the source alphabet is very large or the probabilities are highly skewed.

Arithmetic Encoding

The idea behind arithmetic encoding is that every possible message can be mapped to some unique real number — called a "tag" — in a predefined interval [4]. If we divide the interval into subintervals with sizes proportional to the probabilities of each symbol in the alphabet, we can encode a message as a unique tag by recursively selecting the interval corresponding to the next symbol to encode and re-partitioning the new interval into subintervals. For example, say we have some alphabet $\{a, b, c\}$ with probabilities $P(a) = 0.1, P(b) = 0.2, P(c) = 0.7$. For simplicity, we will choose our interval to be $[\,0, 1)$. If we wish to encode the message $a\,c\,b$, we first divide the interval into $[\,0, 0.1)$, $[\,0.1, 0.3)$, and $[\,0.3, 1)$. Since the first symbol is a, the tag will lie somewhere in $[\,0, 0.1)$. We again subdivide the interval, yielding $[\,0, 0.01)$, $[\,0.01, 0.03)$, and $[\,0.03, 0.1)$. Since the next symbol is c, the tag will lie in $[\,0.03, 0.1)$. Subdividing the interval and encoding b will give us an interval of $[\,0.037, 0.051)$. We can now choose any number in this interval, so we could choose 0.037 for simplicity. The decoder only needs to know the probability model used by the encoder in order to decode the message. When the source alphabet is small and the probabilities are highly skewed, arithmetic encoding can compress much more effectively than Huffman codes.

The drawback is that computers are not well designed for real value arithmetic; dividing real numbers is slow and will lose precision. Researchers later designed integer implementations to avoid these drawbacks [8, 9]. This class of arithmetic coding is now called *range encoding*. Range encoders can use numbers of any base, and so their implementations can greatly differ. Though base 2 encoders provide the strongest compression, base 256 encoders are faster, emitting a byte at a time instead of a bit. Since they also require fewer renormalizations (resizing the interval to prevent overflow or underflow), base 256 encoders are more common [4].

2.2.1 Deflate

The Deflate algorithm, patented by Phillip Katz in 1990 [10], forms the basis of the popular zip file archive format, the gzip application, and the zlib compression library [4, 11, 12]. Deflate has been incorporated to the HTTP protocol since at least 1999, the PNG image format since its inception [13], and the PDF format since version 1.2 [14].

Deflate uses a combination of Huffman coding and LZSS — a close relative of LZ77. While LZ77 requires *every* chunk of data be encoded as a triple *(offset, length, next symbol)*, LZSS uses an indicator bit to allow string literals when the code would be longer than the bytes. Furthermore, it condenses the codes to a pair *(offset, length)* [15]. Deflate then uses a Huffman code to encode offsets and a separate Huffman code for lengths and literals [11]. The Deflate encoder and decoder have two preset Huffman codes that can be used, or they can create unique codes based on the data. The former is used for default-level compression, and the latter is used with higher-level compression.

2.2.2 LZ4 & LZ4HC

Yann Collet designed LZ4 in 2011 to maximize compression/decompression speeds at the cost of weaker compression [16, 17]. LZ4 is strongly related to LZ77, though it uses a hash table or binary search tree to find matches, rather than LZ77's linear search function. To further improve search speed, the hash table must be small enough to fit in L1 cache. Additionally, LZ4 allows literals to prevent data expansion. LZ4HC is a recent variant of LZ4 designed for greater compression by spending more time searching for the best match. Though LZ4HC is slower, it uses the same decompressor as LZ4, and so it still has very fast decompression.

Because of its high compression and decompression speeds, LZ4 has been incorporated into the OpenZFS file system, both to save disk space [18], and because compressing and sending data to a drive may be faster than sending the uncompressed data, even with the time added to compress [19].

2.2.3 LZMA/XZ

The Lempel-Ziv Markov-chain Algorithm (LZMA) is the brainchild of Igor Pavlov designed for high compression and fast decompression [4]. Though decompression is only slightly slower than Deflate, compression takes nearly ten times longer with LZMA [20] largely due to a massive dictionary search, which also demands much more memory. In addition to delivering high compression, LZMA requires only a small memory footprint for decompressing — only 5 KB plus the dictionary size [21]. LZMA is the basis for the XZ compression application.

LZMA begins compressing by using a modified LZ77 algorithm followed by range encoding. Though there are numerous changes from LZ77, one important feature is the array of the four most recent distances. If the file is periodic in nature, certain distances will occur frequently, allowing the encoder to replace these (potentially large) distance values with a 2-bit index. Like Deflate, LZMA allows literals, preventing expansion. To locate matches in the dictionary, LZMA will hash 2, 3, or 4 bytes at a time (depending on the size of the dictionary) to locate a pointer to either a list or a binary search tree (the user can select the former for better speed, or the latter for better compression). Once the dictionary encoder has found the best match, a range encoder passes through, much like how Deflate finishes with Huffman encoding.

2.3 Adaptive Compression

Not all data compresses equally. Intuitively, the greater the redundancy, the higher we can compress a data stream. For example, a file consisting of the characters `aabbaabb` will compress much more effectively than a file containing `acbdbcad`. Data streams with minimal redundancy, such as files that have already been compressed or files containing random data, may not compress at all. A file of random data may actually *expand* if processed by a compressor, since compressors generally add a dictionary or other header information for the decompressor. It is therefore important that we understand some information about the data stream before we attempt to compress it in order to avoid unintended expansion and wasted resources.

The field of adaptive compression (AC) seeks to adapt the compression strategy to the changing environment. The simplest AC scheme will use only one compression method and strength level, and it will choose between "compression on" and "compression off." This approach adds very little overhead, but such a coarse decision model leaves many missed opportunities to optimize. AC becomes more sophisticated when it expands environmental monitoring to more-complex characteristics of the data, the available computational resources (most commonly, the current CPU usage), and the output rate. When compressing a stream to be stored locally, the output rate refers to the throughput from memory to nonvolatile storage. In the context of a network, the output rate is the available bandwidth. Even if characteristics of the data remain constant, a dramatic increase in CPU load may force AC to switch to a less expensive compressor. Likewise, a dramatic decrease in output rate may fill up the write buffer, meaning CPU resources have become relatively cheaper, and so a slower compressor becomes more attractive. Constantly optimizing the compression strategy can save time, space, and energy.

2.3.1 Estimating Compressibility

Part of optimizing the AC system is to better predict future CPU usage and future output rate. While both of these resources are somewhat predictable using recent measurements, estimating the compressibility of the data is much more difficult. AC can predict a stream's compressibility in a number of ways, including measuring compressibility of recent data [22], compressing a small sample of bytes [19], calculating the standard deviation of either the bytes [23, 24] or the difference of consecutive bytes [24], or counting the number of unique bytes that appear more frequently than some threshold [25].

The best way to estimate compressibility is to compress the data, but this has a huge time cost (and energy cost). All time spent estimating compressibility reduces the time savings from compression. For this reason, we want compressibility estimators to be fast, and so far the fastest of the aforementioned predictors requires O(n) runtime.

2.3.2 File Extension as an Estimator

A much simpler and more efficient way to estimate the compressibility of a file would be to use some metadata about the file combined with a constant-time table lookup. Some systems use file extensions — the trailing alphanumeric characters after a ".", e.g., "jpg" in somefile.jpg — to estimate compressibility [26, 27, 28, 29]. An example is the Linux tool `rsync`, commonly used for mirrors for package managers [30], which will choose to compress files with one strength level in the zlib library unless the file extension is assumed to correspond to an incompressible format[1] [26, 31]. Another common example is an optional Apache Server feature that chooses compression the way rsync does.

Though file extensions are not always available, as with network traffic, including a file extension is a widely used convention. Windows systems use file extensions to determine how to open the file, and so they are mandatory [32]; however in Linux

[1]The full list of "incompressible" file extensions is available in table C.1.

systems, file extensions are not required.

A drawback to using extensions is that one could change a file's extension to obscure its true compressibility. For example, one could change a text file's name from "file.txt" to "file.gz," making the file appear to already be compressed with gzip. Text files are usually highly compressible, but an AC system basing decisions on file extension would almost certainly not attempt to compress a pre-compressed file. While this is certainly concerning for multi-user systems, we should not expect a normal user to try to game their own filesystem, and so this concern is not always valid. With some additional education, users can be encouraged to use file extensions on their system to help optimize a local AC system.

2.3.3 Assumption of Normality

Systems that do use file extensions have all made the assumption that the compressibility of those files is normally distributed. If the compressibility of a particular file extension is a normally distributed random variable, then both the arithmetic mean and the mode (being equal to the mean) will accurately predict the long-term compression performance for that file extension. If the distribution is not normal, then mean compressibility could be significantly higher or lower than the mode, meaning a AC system would compress too many incompressible files or miss many compression opportunities. Even if this effect disappears in the limit, unpredictable short-term performance negatively affects other AC goals, like maintaining a constant flow of input and output. It is also possible that the distribution will be multimodal, signalling that there may be some other metadata that could separate the data into several unimodal distributions, which would greatly improve performance.

To date, there has been no robust study to test the assumption of normality.

Without empirical data to estimate the true distributions of compressibility, we cannot definitively say that AC based on file extension is a good or bad idea. If estimating compressibility using file extensions is indeed a good policy, we still do not know the mean compressibility for each extension, which would allow us to optimize the policy.

2.4 Previous Work

2.4.1 Adaptive Compression for the Zettabyte File System

Florian Ehmke attempted to utilize file extensions to make compression decisions in his implementation of AC in the ZFS filesystem [29]. Ehmke's approach allowed the user to prioritize energy use, compression, or system performance. The filesystem kept a lookup table of file extensions and the best compression algorithm for each of the three priority states. Whenever writing a file to disk, it would use the lookup table to make a decision. Ehmke acknowledged that the performance of his system depended on knowing which algorithm is truly best for each file extension. Without an empirical analysis of a broad range of file types, the file extension lookup table is a set of assumptions and intuitions, not an evidence-based policy.

2.4.2 Ares

Devarajan et al. developed an AC engine named "Ares" that uses input type and format to choose a compression library [33]. As a foundation for their dataset, they collected data that was broadly characters, integers, sorted integers, floats, or doubles. They then compressed and decompressed this data with many compressors, the most notable being bsc, bzip2, lz4, lzo, lzma, pithy, quicklz, snappy, and zlib. After averaging the compression ratios and output rates, they built a table that specified the best[2] compression strategies for a given data type and workload strategy. They then built a similar table of best performances based on the data format. Data formats were categorized as "binary data" (e.g., POSIX), "scientific data" (e.g., HDF5), "textual data" (e.g., XML or JSON), or "columnar data" (e.g., AVRO).

Using the dataset above, they built a compression framework that dynamically selected a compression approach using the data type and format. While they did use a file's extension as part of a larger system to infer the data format, the file extension was not used to directly make a compression decision. Additionally, they considered a very narrow range of extensions, ignoring anything uncommon or any extension that could be associated with multiple data formats. During the earlier data collection phase, though they did produce a table of "best" compressors for a given file type, they considered only five classes of data rather than the full range of file extensions. We seek to consider a much broader range of file extensions, and we will show the distribution of compression ratios rather than a single average value.

2.4.3 Efficient Compressibility Estimators

Asamoah Owusu modeled the relationship between efficient compressibility estimators (ECEs) using data gathered from browsing Wikipedia, Facebook, and YouTube [23]. He noted that since estimating compressibility by compressing samples of the data using multiple compressors is so inefficient, there is a great opportunity in using one ECE to predict the output of multiple compressors. While some ECEs are statistical measures, compression ratios from common encoders can also be viewed as ECEs. He first related the compression ratios of the dataset (gzip levels 1 and 6, xz,

[2] "Best" was quantified as a weighted sum of compression metrics, with weights adjusted based on workload priority.

and lz4) to the "average meaning entropy,"[3] Shannon entropy, the byte standard deviation, the "bytecount" (a novel method proposed by Peterson as part of Datacomp [25]), and the "heuristic" (a combination of methods developed by Titovets [34] and based on the work of Harnik et al. [35]).

While there generally was a strong association between ECEs, there were some notable exceptions. Most prominently, bytecounting could predict the compression ratio of the current browsing session, but it could not accurately predict the compression ratio for a future browsing session of the same website. This shows that while fast and accurate for current data, the average bytecount for a class of data should not be used to predict compressibility of future data. In the same vein, average meaning entropy and Shannon entropy of the previous browsing sessions were poor predictors for future browsing sessions.

Though this study grouped data by source, it examined only Internet browsing data. AC can also be used on a local filesystem, which includes a much broader range of data types. This study also focused on the compression ratio only, ignoring the computational cost associated with different compressors.

2.4.4 Datacomp

Peterson proposed estimating file compressibility using "bytecounting," which counts the number of "over-represented" bytes in a stream [25]. In this context, a byte is considered over-represented if it occurs at least $\frac{file\_size}{256}$ times, which is the expectation assuming a uniform distribution of bytes. Bytecounting (BC) is strongly related to entropy, though it is much faster and requires fewer resources to compute. He then utilized BC to implement Datacomp, a general purpose AC system that

[3] "Average meaning entropy" is Titovet's metric, which is the difference of the average byte value for this data and the average byte value for random data. The larger the difference, the higher the estimated compressibility.

monitors CPU usage, available bandwidth, and estimated compressibility to make compression decisions. Since Datacomp is designed to be useful for both local files or network transfers, it cannot rely on access to file metadata, such as extensions, to estimate compressibility, and so it did not rely on any characteristics beyond the bytes themselves to estimate compressibility.

2.4.5 Statistical Measures as Predictors of Compression Savings

Culhane explored three statistical measures of files to determine which, if any, would be good predictors of compression ratio [24]. He used the standard deviations of three values of the data: the individual bytes, the difference of consecutive bytes, and the XOR of consecutive bytes. The dataset came from a single hard drive, with the assumption that the spread of the data would be enough to justify a generalization to other files of the same type. The compression algorithms used were Huffman coding and several varieties of LZW. Ultimately, he found that the best predictor of file compressibility was the standard deviation of bytes, with accuracy above 76%.

Culhane's study provides evidence that files of the same type compress similarly, which shows that a file extension is a promising predictor for compressibility. What his study lacked was comparison to a broad set of compression algorithms; performance on LZW and Huffman coding do not necessarily predict the performance of other common compression algorithms, like LZMA and LZ4. Even Deflate, which incorporates Huffman coding, may perform much differently than pure Huffman coding, since Deflate uses Huffman coding not for the data itself, but to encode dictionary entries. We will also have much stronger evidence for generalization with data from multiple systems using different operating systems.

3 Experiment

To show the relationship between file extension and compression performance, we needed data about the compressibility of files that exist in the real world. To draw valid conclusions, we needed data sourced from many different users with varying computing resources.

3.1 Comprestimator

To gather data about the compressibility of real-world files, we built Comprestimator, a tool that compresses system files with multiple compressors and stores the results. Compression is generally a very slow process, so we wanted to make the program as low-level as possible to get a decent-sized result set from each user without asking them to run the program for a week. Though we could send the source code to each participant with instructions to compile, this would limit data collection to machines operated by computer-savvy users. We chose the Java programming language to strike a balance between portability and low-level operation. Java additionally provides many well-tested compression libraries, allowing us a wide variety of algorithm choices for this experiment.

3.1.1 Compressors

Comprestimator compresses each file with three levels of Deflate, two levels of LZ4, and two levels of LZMA. We chose these general-purpose algorithms for their popularity and very different performance characteristics.

Deflate is an obvious choice for its near-ubiquitous use. The three levels of Deflate available in Java — BEST_SPEED, DEFAULT_COMPRESSION, and BEST_COMPRESSION — correspond to gzip levels 1, 6, and 9. Each is ordered in ascending compression and descending speed. We used the Deflate implementation from the Java 8 SDK [36].

We chose LZ4 due to its increasing popularity, especially since it has become commonplace in the Zettabyte File System. We also use LZ4HC — an acronym for "LZ4 High Compression" — which is essentially the same algorithm as LZ4, but it spends greater time searching for matches in the dictionary. LZ4HC uses the same ultra-fast decompressor as LZ4. We used the LZ4 Java port version 1.7.1 available on github.com [37].

We selected LZMA since it is the main compression algorithm for the popular 7-zip application and forms the basis of the popular XZ application's LZMA2 algorithm [21]. LZMA uses much larger dictionaries than Deflate and LZ4, and so it may be able to find patterns in the data that are far too long for other compressors to detect. We used the XZ application's SDK, which supports LZMA2 [38]. LZMA2 can be thought of as a wrapper for LZMA. LZMA2 first compresses a block using LZMA, then it decides whether to store a block of uncompressed data rather than the compressed data if the compressed data has expanded. This means that LZMA and LZMA2 have virtually the same compression time, though LZMA2 reduces expansion. LZMA2 also supports multithreading and per-block dictionaries, though we did not use these features in order to maintain performance similar to 7-zip. We compressed with XZ level 6 — the default compression level — and level 9 — the highest compression level that uses a much larger dictionary and has a longer maximum match length.

3.1.2 Data Source

Comprestimator compresses all files on a filesystem with six different compressors described above, recording the original file size, the compressed file size, the time to compress, the file extension, and for Linux/Unix systems, the metadata output from the `file` command. We also needed some way of detecting duplicate files, since we need the compression data for only one instance of each file per user. Additionally, we would like to see how different machines perform for the same file, so we need to identify duplicate files across machines. Common operating systems have many pre-installed system files, so we can expect many duplicate files between machines. To protect user privacy, we opted to store a hash value of each file instead of storing the file's name or location on disk. We used the SHA-256 hash function, since its cryptographic properties ensure a high level of security and minimize the probability of a hash collision.

The risk to participants, however, was nonzero. No computationally feasible method exists to reverse the hash value to the original file; in other words, given a hash y, it is intractable to find the original x such that $h(x) = y$. Nevertheless, if we were to hash all files on our own systems, we may find one or more participants with the same hash in their data. If our file's size matches the participant's file size, we can assume these two files are identical. Thus, we can know that a participant has a specific file if we already have a copy of this file. For this reason, we will release data anonymized and only in aggregate. Understanding how we would safeguard all data, the UMN IRB deemed this study "not human subjects research," though we still required all participants to electronically sign a consent form.

We recruited 24 volunteers to run Comprestimator on their machines and send us the results. We instructed users to let the program run when the computer is otherwise idle, since other running processes could interfere with the timed compression process. Data was stored locally using an SQLite database file, which the participants later sent to us via a private dropbox.

Even with the included privacy protection, we gave participants the option to hide files from Comprestimator to prevent them from being touched. If users had any specific files or directories they wanted to hide from Comprestimator, they could enter the full path to the file or directory on a new line in a skip list. Before adding a path to the list of files, Comprestimator would check if the path started with any path on the skip list. Comprestimator never followed symbolic links, so a listed file would remain hidden even if it were linked elsewhere.

3.1.3 Algorithm

Comprestimator begins by calling the Unix `find` command or the Windows equivalent to generate a complete list of accessible files [1]. Each string representation of the path is compared to the list of files in the skip list. If no path in the skip list forms the beginning of the path in the complete list, Comprestimator would add it to the list of files it planned to process. By default, the skip list contained /dev, /proc, /sys, /snap, and /run to avoid the pseudo-files[2] they contain. Once Comprestimator has added the full list of files to process, it shuffles the list to ensure it will process a variety of files, should the program be aborted before the full list has been processed. Comprestimator then writes the list of files to be processed to a file named `enumeration.dat` so that if the program is halted and restarted, it would not need to repeat the enumeration phase.

[1]If the user does not have permission to access a file on the system, Comprestimator would also not have access.

[2]A pseudo-file is a device or process directory created by the kernel that appears to be a regular file.

Compressing every listed file requires significant time, likely upwards of 48 hours. Since we did not expect participants to relinquish control of their computers for two straight days, we designed Comprestimator so that a participant could halt and restart it without losing any previous work. The database file recorded which line in the enumeration file it had last processed so that it could resume where it had left off. This allowed users to run Comprestimator in several segments rather than all at once.

With the shuffled list of files, Comprestimator begins the main compression loop outlined in algorithm 1. Before processing a file, the program again checks that the file is not a pseudo-file and that it has not already processed this file on this machine. With both conditions satisfied, the program reads the file into a byte array in memory.

It is important to make sure that the file being compressed is completely read into memory before compressing it. This ensures the time to compress does not include I/O latency. The Java function $Files.readAllBytes()$ ensures the file is completely in memory. The memory requirement forced us to skip any files larger than about 1 GB, since larger files generally require an amount of heap space that frequently causes an Out of Memory Error.

With the entire file in memory, Comprestimator compresses the byte array and stored the results for each of the seven compressors. It stores the time to compress with microsecond-level precision with the assumption that most CPU process overhead would occur at the nanosecond level, and so nanoseconds would not be significant to calculate the true time to compress.

Algorithm 1 Compression Loop

```
1: for all files do
2:     if $file$ is not a pseudo-file and $file$ not in database then
3:         read entire file into memory
4:         result.$size$ ← size($file$)
5:         result.$hash$ ← SHA256($file$)
6:         result.$ext$ ← file extension
7:         for all compressors do
8:             result.$start$ ← system time
9:             result.$compressSize$ ← size(compress($file$))
10:            result.$stop$ ← system time
11:            store result
12:            reset compressor dictionary
```

3.2 Methodology

The primary goal of this thesis is to estimate the distributions of compressibility for common file extensions, particularly to determine whether the distributions are normal. To do this, we first use the Kolmogorov-Smirnov (KS) test for normality, the Anderson-Darling (AD) test, and Pearson's Chi-Squared test.

3.2.1 Plotting

To graph the distributions, we used a line histogram with bins of width 0.05. We chose histograms over Kernel Density Estimations, because a KDE requires knowledge of the underlying distribution(s). Lines made the differences between compressors much clearer. Though we tried using smaller bins, this smoothed the curves in a way that made it harder to separate each compressor's line. A bin width of 0.05 creates an error of ± 0.025, which is important to consider when we discuss the modes in the results. Our calculation of the mean, median, standard deviation, and all normality tests are unaffected by the binning process.

3.2.2 Compression Ratio

We use the *compression ratio* (CR) as our measure of compressibility. There are multiple definitions of CR: one is the original file size divided by the compressed size (also called *compression factor* [4]). We will use the definition of compressed size divided by original size as outlined in equation 3.1, since nearly all CRs will fall into the range between 0 and 1. With this definition, a CR of 0.1 indicates the compressed file is 10% of the original file size, and a CR of 1.05 indicates the file has expanded to 105% of its original size. Though the range of values for our definition of CR does not have tight bounds, in practice, it is rare that a file larger than 1 KB will expand to more than 110% of its original size.

$$CR = \frac{compressed\ size}{original\ size} \tag{3.1}$$

3.2.3 Normality Testing

The KS, AD, and Chi-Squared tests assume that all files with a particular extension are drawn from the same distribution. To test this assumption, we will compute the means for 100,000 samples of size 30. We will then use the Anderson-Darling test to test whether these means are normally distributed. According to the central limit theorem, if the data points all follow one distribution, then these means should be normally distributed. By extension, if the means are not normally distributed, then the CRs do not come from the same distribution. The CRs do not need to be normally distributed for the CLT to hold.

File extensions that are associated with multiple underlying distributions may have some other distinguishing property in common (e.g., different versions of the same file format) that could help us identify to which distribution a file belongs, or perhaps different users' files follow different distributions.

The primary benefit of normally distributed file CRs is that the arithmetic mean of the files' CRs would be a meaningful estimator of compressibility, as outlined in Section 2.3.3. However, it is possible that some file extensions will have a distribution that is unimodal but not normal. For these extensions, the mean CR may or may not be a useful estimator of compressibility.

Multimodal distributions challenge the hypothesis that a file extension can predict compressibility. A multimodal distribution, however, may indicate a mixture of Gaussian distributions under the umbrella of a single file extension. For these distributions, we ask the question, can we find some readily accessible metadata that can be used to separate data into multiple categories with simpler distributions? An example is the Portable Document Format (PDF). We can see the PDF's version number using the metadata we gather. If the distribution for PDFs appears to be multimodal, perhaps different PDF versions will follow different distributions.

Kolmogorov-Smirnov Test

The Kolmogorov-Smirnov test is used to test goodness-of-fit for a population. Specifically, the KS test returns a statistic s which is equal to the maximum distance between the Empirical Distribution Function for the sample and the Cumulative Distribution Function for the assumed underlying distribution (in this case, the normal distribution). We use the Scipy's *stats.kstest* function to determine the likelihood that our data does *not* fit a normal distribution. *stats.kstest* automatically computes a p-value for our sample. It is important to note that the KS test gives much more weight to the center of the data than it does to the tails [39].

Anderson-Darling Test

The Anderson-Darling test is a modification of the Kolmogorov-Smirnov test that puts extra weight on the tails. It is generally considered more sensitive than the KS test [40]. Since compressors usually detect potential data expansion and resolve to store the uncompressed data, it is very rare for a compressed file to expand by much. This means that the further the mode for a file type shifts towards 1, the less a right-hand tail will appear. Similarly, the closer the mode shifts towards 0, the left-hand tail must disappear at zero. Despite these drawbacks, the Anderson-Darling statistic remains an important metric.

We used Scipy's $stats.anderson$ function, which calculates the probability that a given set of points does *not* follow a normal distribution. The function outputs the Anderson-Darling statistic A along with five critical values, corresponding to the significance levels 0.15, 0.1, 0.05, 0.025, and 0.01. For example, the output $A = 0.8$, [0.576, 0.656, 0.787, 0.918, 1.092] would indicate that the set of input points likely do not follow a normal distribution with $p < 0.05$, since 0.787 corresponds to $\alpha = 0.05$. With these same critical values, $A = 0.7$ would indicate not normal with significance $p < 0.1$. Unfortunately, with this function, we cannot report significance beyond $p < 0.01$.

Pearson's Chi-Squared Test

Pearson's Chi-Squared test is another goodness-of-fit test for observed distributions. Unlike the KS test and the Anderson-Darling test, Chi-Squared can be applied to discrete distributions [41]. A drawback of the Chi-Squared test is that it requires a large sample to be significant, and so it will not be reliable for the extensions with the minimum number of data points. We used Scipy's $stats.chisquare$ function, which automatically computes the p-value for the given sample.

3.2.4 Extension Selection Criteria

Due to the sheer magnitude of file extensions in use, we will analyze only the most common extensions from our data. We will use three criteria to select extensions to analyze: 1) popularity by ranked vote, 2) total number of files from all users, and 3) diversity of data types. The ranked vote system prevents bias in the list of extensions towards participants who contribute massive datasets. Each user will vote their top 90 extensions, and we will manually select from this list. We expect to add or remove some extensions to ensure we analyze a variety of data types and include some extensions that will be especially interesting.

Since users may choose how much time they want to run Comprestimator, we expect some users to contribute datasets much larger than average. To prevent these users' data from dominating the results, we will select a limited number of files per participant. We decided this limit for an extension will be the smallest number of files any particular user has that is greater than 10. This should create a sample large enough to be representative while still mitigating sampling bias. If we believe this strategy fails to prevent sampling bias for any particular extension, we will consider dropping this extension from our analysis.

4 Results

We received compression data from operations on a total of 12,018,933 files from 24 participants. Table 4.1 shows how many files each participant processed and how many hours processing took[1]. It should be noted that the majority of participants were computer science students or professionals, and so source code and LaTeX files were over-represented.

The most common extension by both ranked vote and total files was the empty extension, that is, files with no extension. PNG was the second most common extension by both measures. Tables 4.3 and 4.2 list the top 90 extensions by total and ranked vote, respectively.

Of the 18 extensions analyzed, all normality tests described in Section 3.2.3 returned not normal with $p < 0.01$ at the minimum significance. For the vast majority, p approached 10^{-23}, some even too small to fit in a 64-bit float. For this reason, we will not include specific numbers for these tests. For multimodal distributions shown below, the mode refers to the single largest mode in the distribution. A represents the Anderson-Darling statistic used to test whether the CRs are from one single distribution as described in Section 3.2.3. SD refers to standard deviation.

[1]In order to collect metadata, some participants used Windows Subsystem for Linux, which has very high I/O latency. Once the file was in memory, compression speed was near equal to the equivalent in CMD.

Table 4.1: Total Files and Hours of Compression Per User

User	Total files	Hours
1	2,269,436	111
2	183,922	13
3	138,015	15
4	142,006	13
5	606,730	75
6	177,051	8
7	296,194	37
8	157,111	12
9	253,166	18
10	228,612	46
11	282,567	69
12	880,003	82
13	272,082	15
14	981,004	177
15	107,741	26
16	1,936,267	173
17	50,114	4
18	496,324	87
19	142,222	48
20	413,491	22
21	251,102	24
22	798,042	134
23	422,011	40
24	533,720	58

4.1 File Formats

We chose to analyze BIN, DLL, DOCX, EXE, GIF, GZ, HTML, JPG, JS, JSON, MP3, PDF, PNG, RTF, SVG, TTF, TXT, and WAV individually. With the exception of DOCX, all were in the top 50 by ranked vote and/or the top 75 by total files. We added DOCX to the list because it is a file extension common enough to interest other researchers. Additionally, we did a brief analysis on a group of extensions that are always compressed and a group of extensions representing source code. In the following sections, we describe these extensions and the formats they represent, and present the Comprestimator results for each.

Table 4.2: Most Common File Extensions Observed

Rank	Ext.	Total	Rank	Ext.	Votes	Rank	Ext.	Votes
1	(none)	2823530	31	m	61923	61	map	20564
2	png	498864	32	vf	60237	62	sty	19734
3	html	497473	33	cc	57190	63	mp3	18953
4	h	451014	34	exe	56093	64	vim	18854
5	dll	409941	35	bin	56080	65	s	18709
6	pyc	358332	36	plist	51102	66	ttf	18679
7	c	329101	37	page	49252	67	pl	18665
8	js	314428	38	mui	47512	68	tif	18472
9	strings	309089	39	pm	44807	69	css	17759
10	py	285446	40	class	42185	70	d	17605
11	cat	278685	41	pdf	41033	71	log	17547
12	jpg	263022	42	md	39798	72	sh	17256
13	tfm	206884	43	i	37921	73	1	16715
14	manifest	202276	44	wem	35831	74	0	16438
15	mum	170092	45	jar	33071	75	gif	16430
16	java	158671	46	file	30173	76	po	16373
17	gz	157473	47	loopdata	27851	77	rst	16217
18	xml	145239	48	wav	27401	78	php	15687
19	go	134126	49	mlx	27305	79	sys	14577
20	ko	120264	50	final	25192	80	pak	14499
21	svg	115943	51	pfb	24176	81	qml	14279
22	json	108612	52	tex	24151	82	pri	13663
23	ogg	104942	53	rb	23939	83	rs	13604
24	cpp	90874	54	a	23336	84	stringsdict	13393
25	hpp	89700	55	ts	23265	85	tga	13340
26	o	86887	56	dat	23122	86	fd	12929
27	mo	80833	57	lua	22949	87	inf	12442
28	nib	78046	58	tiff	22667	88	etl	11961
29	txt	77034	59	dds	21441	89	otf	11898
30	so	77018	60	enc	21289	90	xnb	11795

Table 4.3: Most Common File Extensions by Ranked Vote

Rank	Ext.	Votes	Rank	Ext.	Votes	Rank	Ext.	Votes
1	(none)	2862	31	log	1240	61	res	750
2	png	2739	32	java	1238	62	plist	742
3	js	2588	33	jar	1237	63	ts	721
4	html	2582	34	vim	1222	64	pod	714
5	xml	2521	35	hpp	1208	65	enc	712
6	jpg	2320	36	pak	1195	66	file	705
7	txt	2298	37	c	1162	67	winmd	684
8	json	2244	38	lua	1160	68	ko	679
9	svg	2171	39	gif	1129	69	otf	671
10	py	2093	40	sys	1047	70	ogg	669
11	bin	2028	41	tfm	1026	71	pfb	660
12	h	1987	42	htm	1023	72	pnf	652
13	pyc	1984	43	cpp	1021	73	sty	652
14	dll	1919	44	pri	1010	74	mof	649
15	ttf	1701	45	inf	992	75	afm	647
16	dat	1685	46	final	945	76	vf	623
17	gz	1661	47	tga	934	77	page	620
18	pm	1554	48	etl	913	78	go	617
19	mo	1553	49	rtf	906	79	class	612
20	css	1498	50	1	879	80	wmf	608
21	so	1446	51	rb	845	81	zip	607
22	md	1431	52	mp3	820	82	qml	602
23	exe	1399	53	o	806	83	sh	594
24	cat	1384	54	cab	805	84	adml	585
25	manifest	1370	55	a	800	85	man	574
26	pdf	1361	56	0	798	86	ps1	567
27	mum	1344	57	strings	788	87	pyi	562
28	wav	1301	58	map	783	88	cdxml	557
29	mui	1261	59	tex	769	89	tcl	553
30	pl	1245	60	ini	767	90	md5sums	544

4.2 Results by Extension

4.2.1 Null Extension

The "null extension" refers to the absence of an extension, which was the most common extension both by ranked vote and overall. The average file size was 96

Table 4.4: Null Extension

Algorithm	Mean CR	Median CR	Mode	SD	A
Deflate 1	0.612	0.577	1.00	0.277	$14,510.591,\ p < 0.01$
Deflate 6	0.597	0.563	1.00	0.287	$14,938.423,\ p < 0.01$
Deflate 9	0.597	0.563	1.00	0.288	$14,879.734,\ p < 0.01$
LZ4	0.696	0.721	1.00	0.249	$10,783.644,\ p < 0.01$
LZ4HC	0.669	0.695	1.00	0.265	$11,256.256,\ p < 0.01$
XZ 6	0.584	0.568	1.00	0.296	$13,466.838,\ p < 0.01$
XZ 9	0.584	0.568	1.00	0.296	$13,465.480,\ p < 0.01$

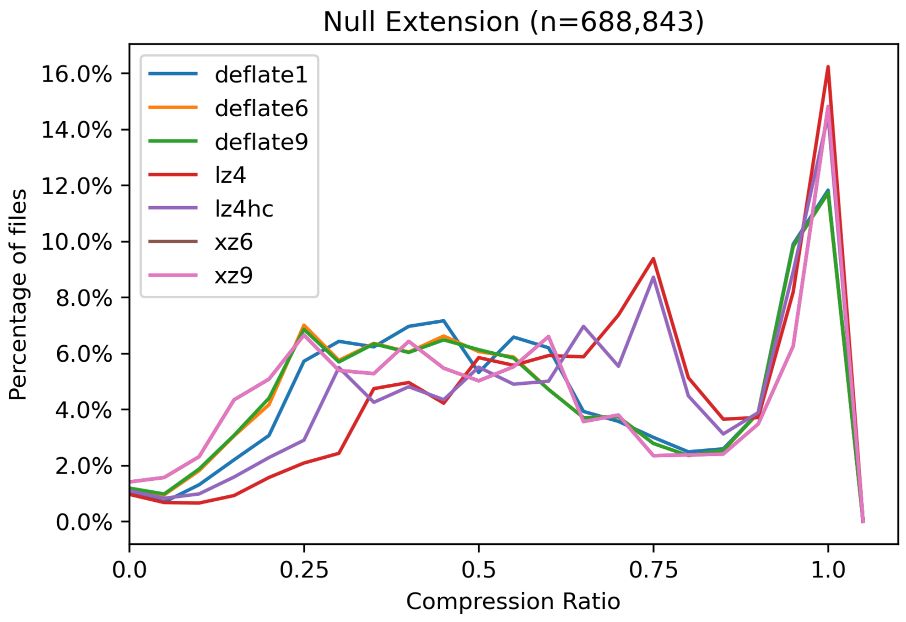

Figure 4.1: Histogram for Null Extension

KB, and the largest file processed was about 1 GB, the maximum size allowed. As expected, this data is anything but normal.

As the Anderson-Darling statistic A in Table 4.4 shows, these files certainly follow many different distributions. For reference, $A \approx 1$ is a value high enough to reject the null hypothesis with $p < 0.01$, and all values of A are above 10,783.

It is very concerning that the most common extension, representing 23.5% of all results gathered, provides very little information about compressibility. We don't know how commonly a system would interact with this type of file, i.e. 23.5% of files stored does not guarantee 23.5% accessed and transported, so the high prevalence of this files does not automatically imply disaster.

4.2.2 BIN

Table 4.5: BIN

Algorithm	Mean CR	Median CR	Mode	SD	A
Deflate 1	0.555	0.465	1.00	0.328	25.507, $p < 0.01$
Deflate 6	0.540	0.434	1.00	0.337	27.875, $p < 0.01$
Deflate 9	0.539	0.433	1.00	0.338	27.840, $p < 0.01$
LZ4	0.620	0.598	1.00	0.312	19.023, $p < 0.01$
LZ4HC	0.587	0.539	1.00	0.325	22.195, $p < 0.01$
XZ 6	0.508	0.374	1.00	0.354	30.586, $p < 0.01$
XZ 9	0.508	0.374	1.00	0.354	30.545, $p < 0.01$

Bin is an extension associated with binary (i.e. non-text) data. "Binary data" is extremely broad; it includes executable files, images, audio, and compressed data, both proprietary and open-source [42], so we doubted that it would be normally distributed.

BIN (Table 4.5 and Figure 4.2) was the 11th most common extension by ranked vote and 35th overall. The average file size was 906 KB with the largest file at 732 MB. While the means and medians are relatively close for all compressors (largest distance was 0.134 for XZ 6 & 9, smallest was 0.022 for LZ4), the mode was far away at 1.00 for all compressors. There were no striking differences in results between

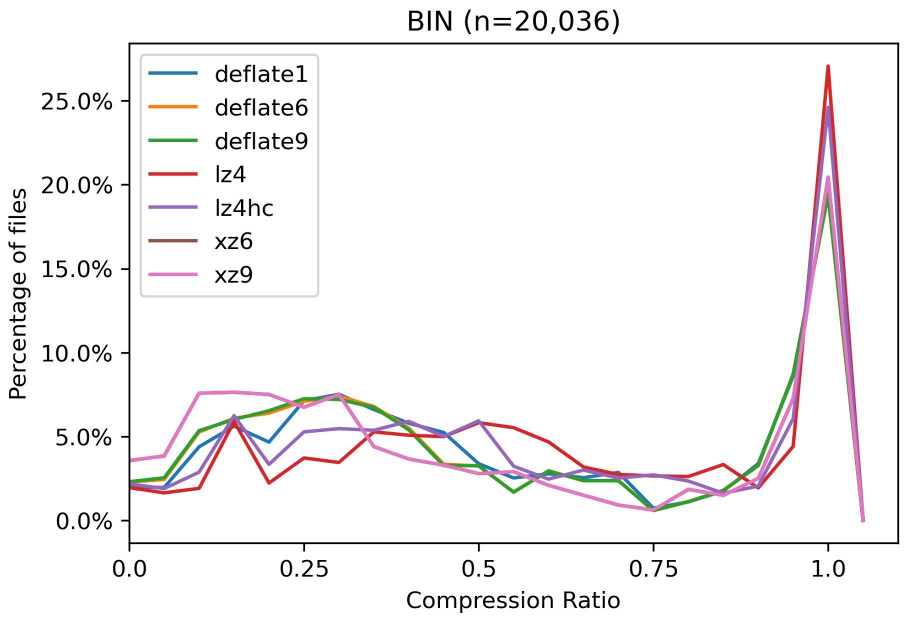

Figure 4.2: Histogram for BIN

compressors.

Figure 4.2 showed that roughly 25% of all BIN files were completely incompressible, but that leaves almost 75% of files that will have decent to excellent compression. Given how broad this category is, there are also a massive number of unique results from the `file` command, so binning files with that information will require significant time.

4.2.3 DLL

DLL is generally a Dynamic-Link Library used within Microsoft Windows operating systems. DLLs can encapsulate data from multiple formats, such as ActiveX controls (.oxc), Control Panel files (.cpl), and device drivers (.drv) [43].

DLL (Table 4.6 and Figure 4.3) was the 14th most common extension by ranked vote and the 5th overall. The average file size was 556 KB, and the largest file was 421 MB. The slightly larger average file size may make DLLs more-attractive targets

Table 4.6: DLL

Algorithm	Mean CR	Median CR	Mode	SD	A
Deflate 1	0.604	0.499	1.00	0.275	23910.964, $p < 0.01$
Deflate 6	0.585	0.471	1.00	0.287	24748.282, $p < 0.01$
Deflate 9	0.584	0.469	1.00	0.288	24729.361, $p < 0.01$
LZ4	0.678	0.614	1.00	0.237	15878.543, $p < 0.01$
LZ4HC	0.633	0.549	0.95	0.259	20155.424, $p < 0.01$
XZ 6	0.547	0.409	1.00	0.314	27016.871, $p < 0.01$
XZ 9	0.546	0.409	1.00	0.314	27015.298, $p < 0.01$

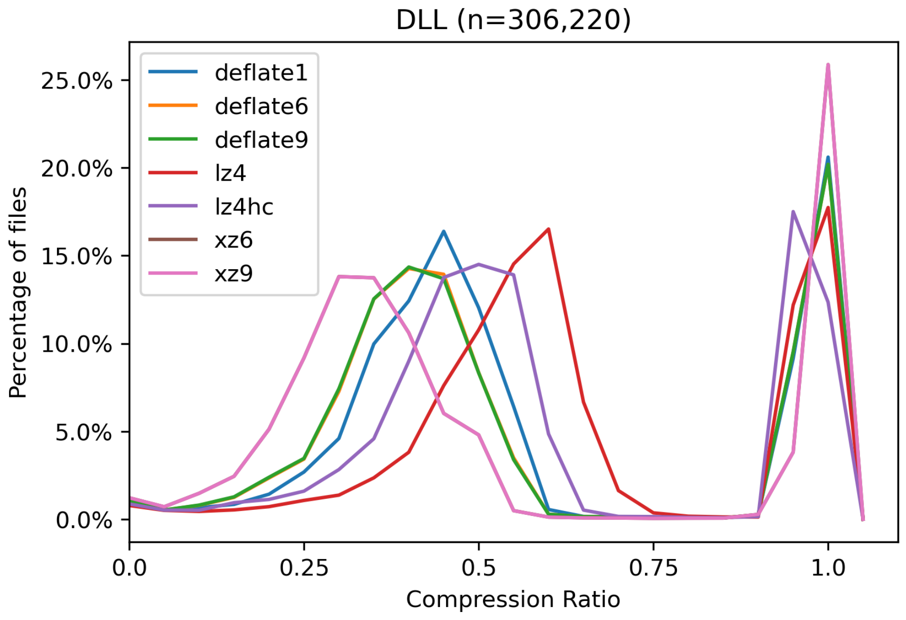

Figure 4.3: Histogram for DLL

for compression. As evidenced by both the chart and the very high values of A, all lines are multimodal, having a smaller mode very close to 1. Oddly, LZ4HC had a secondary mode near 0.9, smaller than all other compressors.

Some programs exist to compress executable DLLs [44]. The `file` command does not indicate whether the DLL has been compressed, but given the large group of CRs near 1, it is likely that many of these DLL files were already compressed.

Upon further investigation, it appeared many of the DLLs contained PGP keys (Figure 4.6 shows 380). Using a uniform number of data points per user and splitting the data points into those containing the word "key" and those without, the distributions did not change significantly (see Figures 4.4 and 4.5). When looking at all DLLs without limiting the number of files per user, nearly all files containing the word "key" in the metadata had a CR near 1. Shown in Figure 4.7), removing files containing "key" had very little difference from the original distribution.

Average file sizes changed significantly depending on the range of CRs. The average size of files with a CR greater than 0.95 was 37 KB, greater than 0.9 was 41 KB, less than 0.9 was 761 KB, and less than 0.75 was 755 KB. The average CR for files less than about 100 KB was 0.65. The average CR increased steadily as the sizes decreased, reaching a peak of 0.96 for files less than 2 KB. In the opposite direction, the average CR for files larger than 20 MB was 0.27. This does not perfectly bin the results, however, since at least one file larger than 20 MB had a CR greater than 0.9.

4.2.4 DOCX

DOCX is the Microsoft Word Open XML format introduced in 2009 [45]. DOCX was introduced to replace MS Word's older DOC format. It includes information about font type, size, and color, margins, embedded images, etc. DOCX is automatically losslessly compressed with Deflate.

DOCX (Table 4.7 and Figure 4.8) was the 154th most common overall. It did not appear in the ranked vote, which spanned the top 160 extensions. Even though DOCX was relatively uncommon, people interact with DOCX often and know the extension fairly well, so we considered its results interesting. The average DOCX file size was

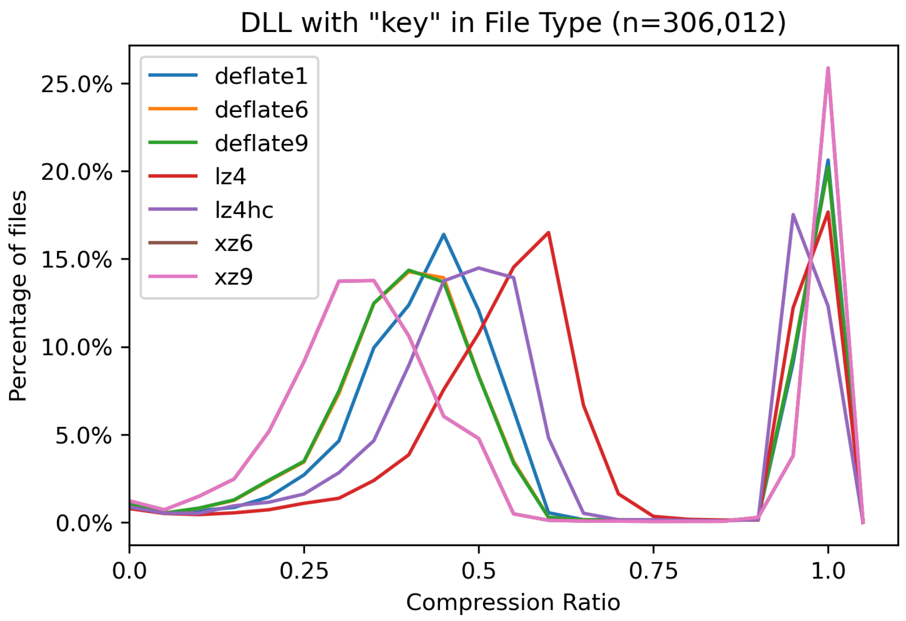

Figure 4.4: Histogram for DLL containing "key" in file metadata

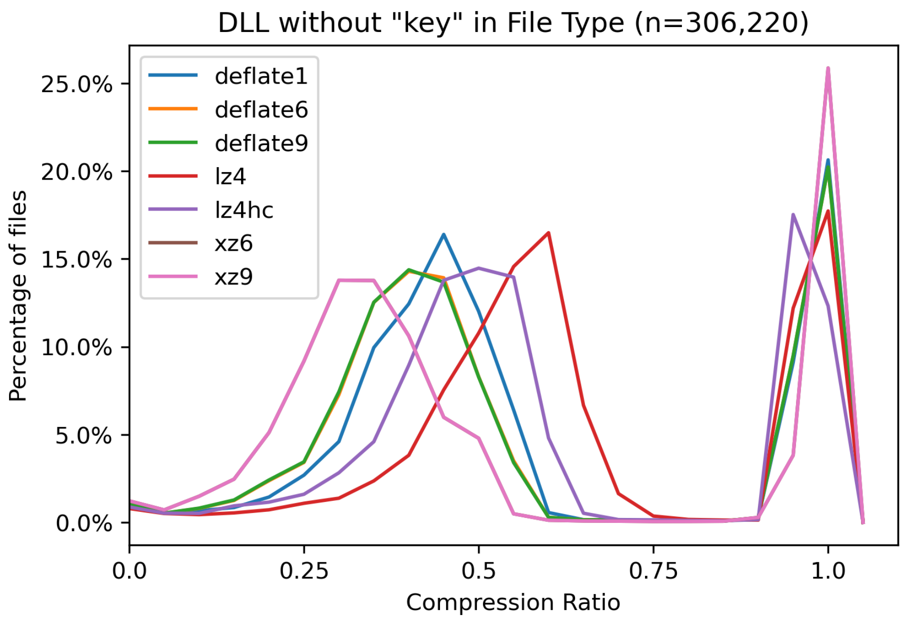

Figure 4.5: Histogram for DLL without "key" in metadata

738 KB with a maximum size of 32 MB. As expected, CRs were fairly high (DOCX is automatically compressed with Deflate), though the mode was a surprisingly low 0.8.

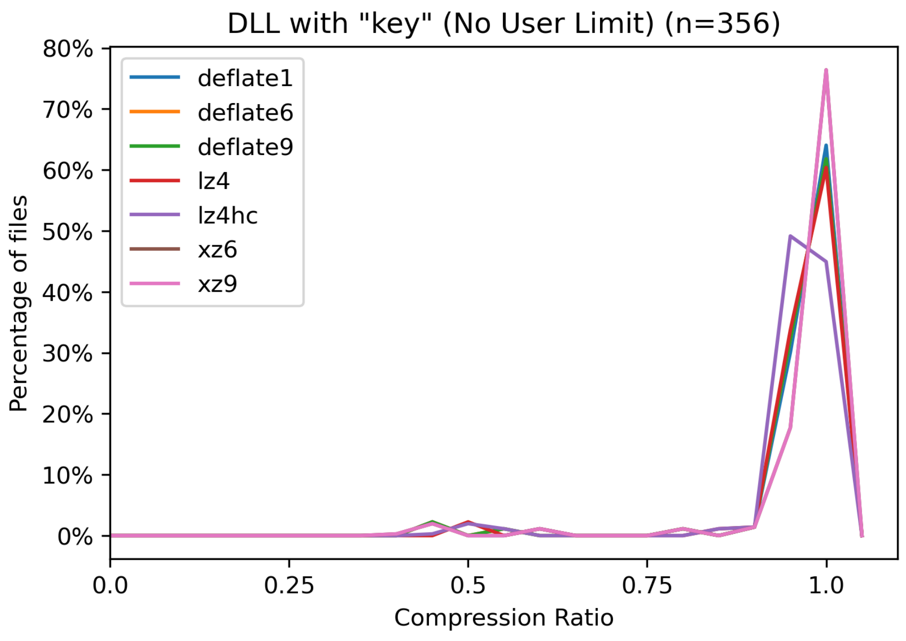

Figure 4.6: Histogram for DLL containing “key,” no limit on user contribution

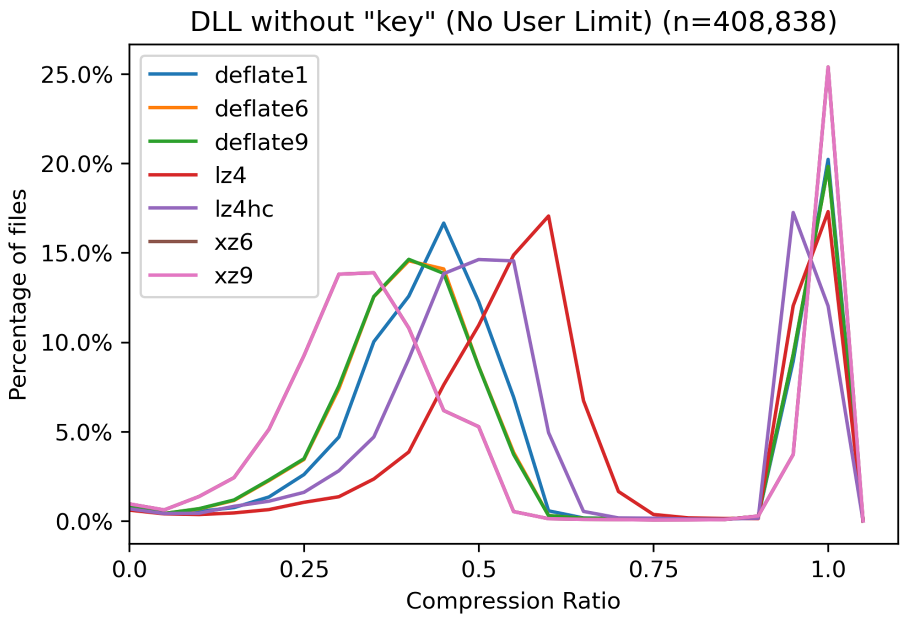

Figure 4.7: Histogram for DLL without “key,” no limit on user contribution

All compressors had a mean, median, and mode very close together, with LZ4 having the largest distance between mean and mode at 0.066. Both XZ levels had the smallest

Table 4.7: DOCX

Algorithm	Mean CR	Median CR	Mode	SD	A
Deflate 1	0.853	0.849	0.80	0.069	6.144, $p < 0.01$
Deflate 6	0.849	0.845	0.80	0.070	5.976, $p < 0.01$
Deflate 9	0.849	0.844	0.80	0.070	5.907, $p < 0.01$
LZ4	0.862	0.856	0.80	0.070	5.748, $p < 0.01$
LZ4HC	0.853	0.846	0.80	0.071	5.892, $p < 0.01$
XZ 6	0.847	0.843	0.80	0.077	9.531, $p < 0.01$
XZ 9	0.847	0.843	0.80	0.077	9.531, $p < 0.01$

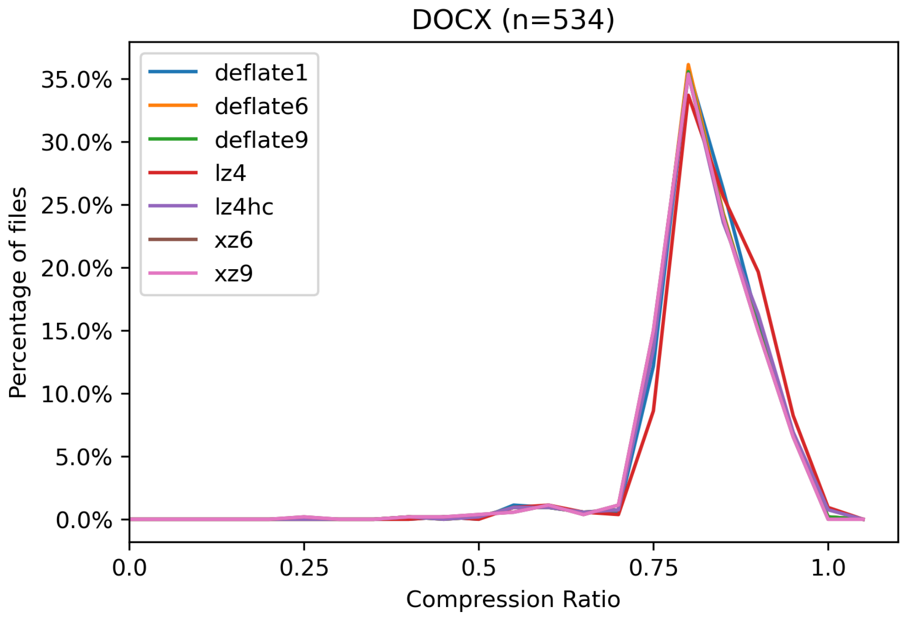

Figure 4.8: Histogram for DOCX

distance at 0.05 between median and mode. Their near-identical performance may have been related to the small average file size; XZ 6 is already looking for matches so long that higher levels of XZ can provide no additional benefit. The average file size among files with a CR lower than 0.75 was 370 KB, suggesting that smaller files may be lightly compressed.

4.2.5 EXE

Table 4.8: EXE

Algorithm	Mean CR	Median CR	Mode	SD	A
Deflate 1	0.621	0.513	1.00	0.268	2426.411, $p < 0.01$
Deflate 6	0.603	0.487	1.00	0.279	2521.368, $p < 0.01$
Deflate 9	0.602	0.486	1.00	0.279	2520.533, $p < 0.01$
LZ4	0.699	0.640	1.00	0.226	1409.943, $p < 0.01$
LZ4HC	0.653	0.574	0.55	0.250	1849.589, $p < 0.01$
XZ 6	0.564	0.427	1.00	0.306	2765.054, $p < 0.01$
XZ 9	0.564	0.427	1.00	0.306	2762.860, $p < 0.01$

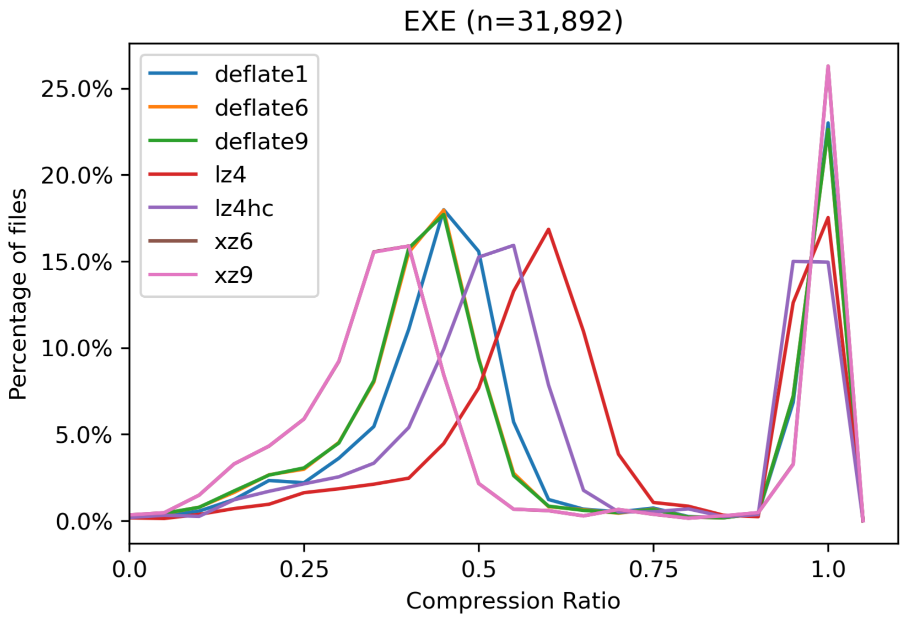

Figure 4.9: Histogram for EXE

EXE refers to an executable file for Microsoft operating systems [46]. Even though EXE is specific to PCs, we chose to analyze this extension to explore executable files in general. Many Mac and Linux executables do not use an extension, so header information is necessary to identify them. We expect machine code to compress well; however, some executables are already compressed [4]. Compressed and uncompressed executables share the EXE extension, so the extension alone will likely tell us very little about compressibility.

EXE (Table 4.8 and Figure 4.9) was the 23rd most common extension by ranked vote and 34th overall. The average file size was 2 MB, and the largest file was 1 GB (the maximum allowable size). EXE had a multimodal distribution resembling that of DLL. Like DLL, LZ4HC inexplicably performed better on data with CRs near 1. LZ4 similarly performed better on the less-compressible files, though it did not have this performance advantage on DLLs.

Many EXEs are already compressed, which should be a strong predictor that the file will not compress further. Further investigations, however, did not provide much supporting evidence. We assumed a file was compressed if the metadata contained the substrings "compress" or "extract," which frequently corresponded to the strings "compressed" or "self-extracting." Including only the files containing compression keywords provided no additional information for the distribution using a uniform number of points per user (Figure 4.10); however, the distribution with unlimited points per user (Figure 4.11) showed that including the compression keywords strongly predicted the file would have a large CR. Strangely, there were still a significant number of supposedly compressed EXEs with CRs around 0.1 to 0.4.

The average CR for each participant was relatively stable. Two participants had a mean CR of 0.41 for Deflate 6 (about 0.19 below the global average), and one had an average of 0.73 (about 0.13 above global average), but most other participants had averages in the 0.5 to 0.7 range. Table 4.9 shows each participant's average CR and how many EXEs each user contributed when we ignored user limits.

Table 4.9: Average CR and Number of EXEs per Participant

User ID	Avg CR	Num Files
1	0.56	45
2	0.66	3500
3	0.73	3138
4	0.62	3967
5	0.33	1608
6	0.64	11
7	0.66	4531
8	0.52	9
9	0.52	9
10	0.67	4154
11	0.61	4347
12	0.58	6753
13	0.41	62
14	0.49	440
15	0.60	4183
16	0.30	1058
17	0.51	8
18	0.54	3614
19	0.70	3180
20	0.41	143
21	0.52	9
22	0.57	7094
23	0.66	4216
24	0.54	14

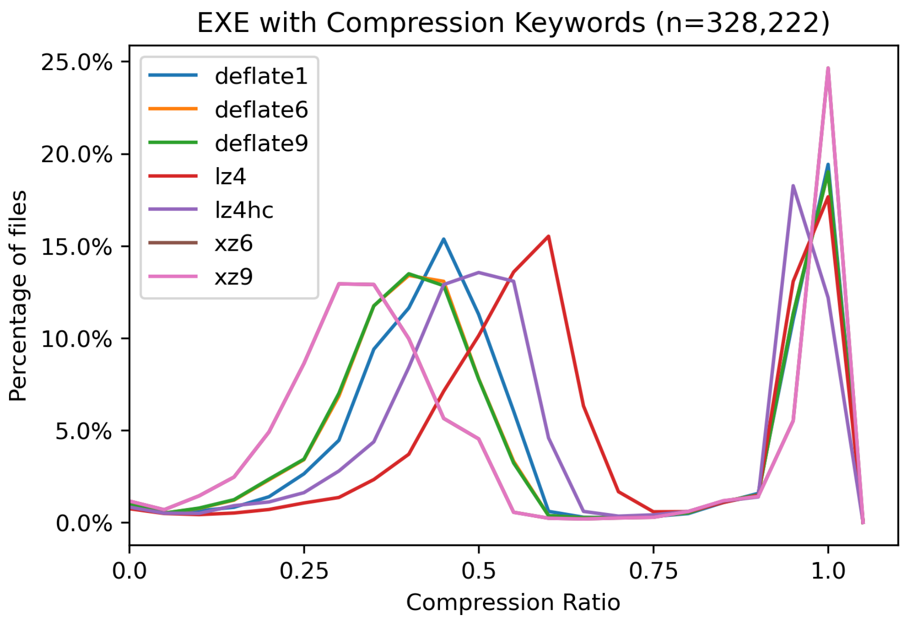

Figure 4.10: Histogram for EXE containing compression keywords

4.2.6 GIF

GIF, or Graphics Interchange Format, is a format that stores image data compressed with a variant of LZW [4]. Though LZW is a lossless compressor, GIF image input is restricted to 256 colors, meaning a higher-quality image will lose some de-

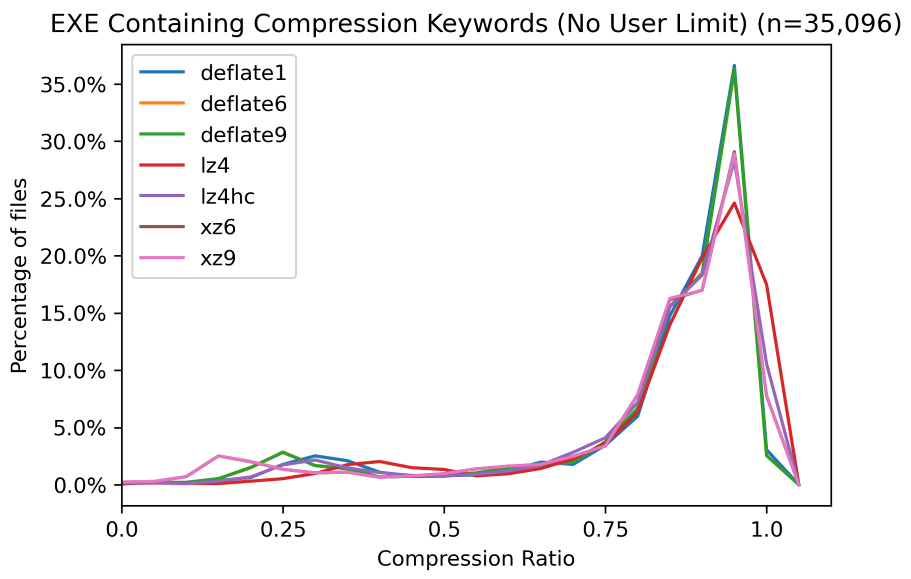

Figure 4.11: Histogram for EXE containing compression keywords, no user limit

Table 4.10: GIF

Algorithm	Mean CR	Median CR	Mode	SD	A
Deflate 1	0.903	0.978	0.95	0.165	746.165, $p < 0.01$
Deflate 6	0.901	0.977	0.95	0.167	739.436, $p < 0.01$
Deflate 9	0.901	0.977	0.95	0.167	739.610, $p < 0.01$
LZ4	0.913	0.984	1.00	0.163	856.342, $p < 0.01$
LZ4HC	0.909	0.981	1.00	0.166	819.544, $p < 0.01$
XZ 6	0.880	0.954	0.95	0.173	531.531, $p < 0.01$
XZ 9	0.880	0.954	0.95	0.173	531.531, $p < 0.01$

tail before compression. Though GIF was never intended to be an animation format [47], one can animate GIFs by encapsulating multiple frames within a single file, each with its own time delay. Because LZW is a one-dimensional compressor, GIF cannot exploit color similarities between vertically adjacent pixels, or temporally adjacent pixels, in the case of animated GIFs.

GIF (Table 4.10 and Figure 4.12) was the 39th most common extension by ranked

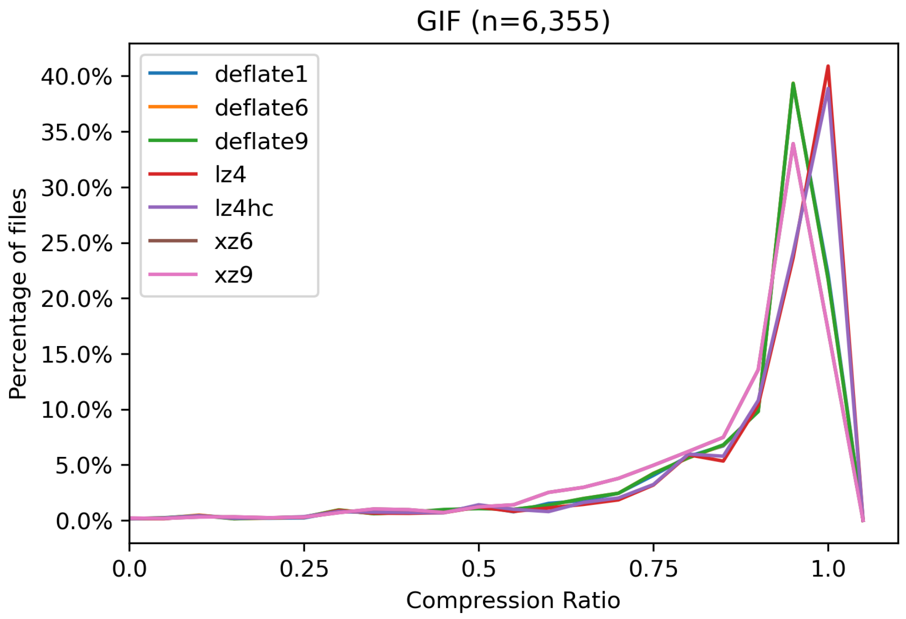

Figure 4.12: Histogram for GIF

vote and 75th overall. The average file size was 73 KB, and the largest was 22 MB. The distribution is very clearly not normal, with all modes around 0.95 or higher. There is still a significant group of files compressible to 80% or less of the original size. Since the distribution is so heavily weighted near 1, the mean, median, and mode for all compressors are somewhat close, but the mean is universally smaller than both the median and mode. The difference between the mean and the mode is as much as 0.078 (for XZ 6 & 9), which could lead to many naïve attempts to compress incompressible files if using the mean as an estimator. Perhaps the most important metric is the median, which shows that half the files have a CR larger than 0.962 for XZ. Even if this were the median for the fastest compressor, it still may not be profitable. The minority of files with CRs around 0.5 are skewing the mean much lower than we would expect it to be.

Table 4.11: GZ

Algorithm	Mean CR	Median CR	Mode	SD	A
Deflate 1	1.000	1.004	1.00	0.030	14377.647, $p < 0.01$
Deflate 6	1.000	1.004	1.00	0.030	14376.987, $p < 0.01$
Deflate 9	1.000	1.004	1.00	0.030	14381.956, $p < 0.01$
LZ4	1.002	1.004	1.00	0.027	19089.815, $p < 0.01$
LZ4HC	1.000	1.004	1.00	0.028	17329.813, $p < 0.01$
XZ 6	1.011	1.014	1.00	0.034	7328.876, $p < 0.01$
XZ 9	1.011	1.014	1.00	0.034	7329.839, $p < 0.01$

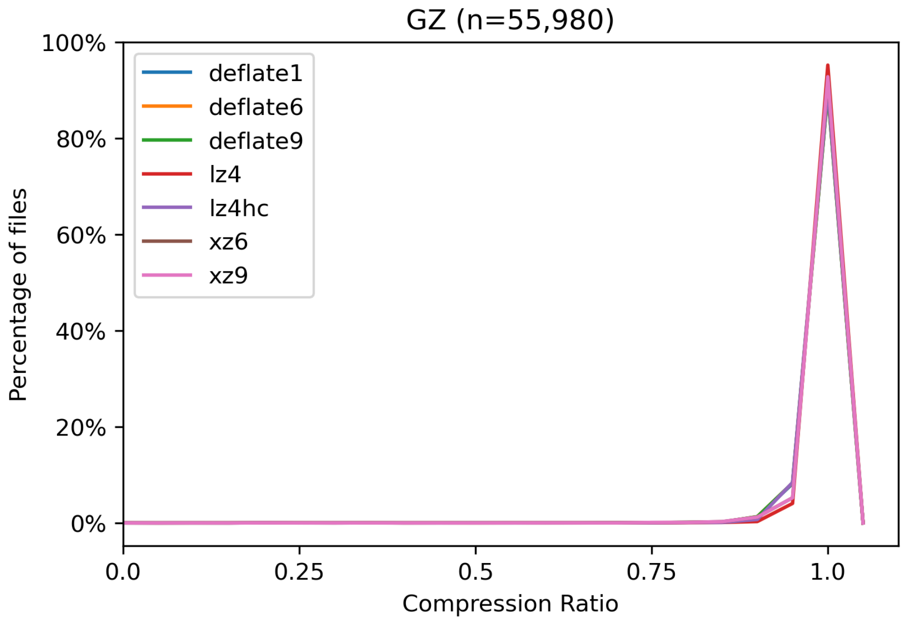

Figure 4.13: Histogram for GZ

4.2.7 GZ

GZ is the extension for gzip-compressed files. Gzip is based on Deflate, which as described in Section 2.2.1 the background, is a very effective compression algorithm. For this reason, we expect very little compression or some expansion when trying to compress pre-compressed files.

GZ (Table 4.11 and Figure 4.13) was the 17th most common extension both by ranked vote and overall. The average file size was 132 KB with a maximum size of 1 GB (the largest size considered). It is not at all surprising that the mean, median, and mode are so very high. All medians were larger than 1, meaning that more than half of all files expanded when attempting to compress these pre-compressed files.

4.2.8 HTML

Table 4.12: HTML

Algorithm	Mean CR	Median CR	Mode	SD	A
Deflate 1	0.330	0.325	0.25	0.101	378.407, $p < 0.01$
Deflate 6	0.303	0.294	0.20	0.105	706.272, $p < 0.01$
Deflate 9	0.303	0.294	0.20	0.106	711.838, $p < 0.01$
LZ4	0.444	0.438	0.35	0.136	396.095, $p < 0.01$
LZ4HC	0.405	0.392	0.30	0.142	761.319, $p < 0.01$
XZ 6	0.308	0.293	0.20	0.118	1020.903, $p < 0.01$
XZ 9	0.308	0.293	0.20	0.118	1020.903, $p < 0.01$

HTML and HTM denote Hypertext Markup Language files used for internet browsers. HTML is an XML-based text language, and so we expect these files to compress like other XML-style files. HTML files may contain some JavaScript using the `<script></script>` tags. Though we also expect JavaScript to compress well (see Section 4.2.10), if JS has a mean far from that of pure HTML, this may produce a second mode.

HTML (Table 4.12 and Figure 4.14) was the 4th most common extension by ranked vote and the 3rd overall. The average file size was a small 18 KB, and the

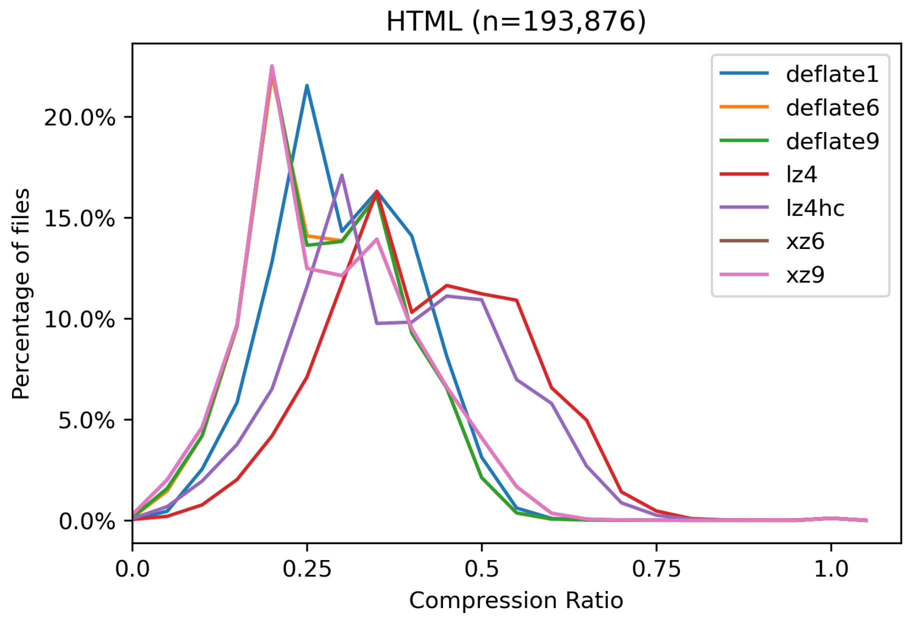

Figure 4.14: Histogram for HTML

largest size was 44 MB. The group of files near 1 is very surprising for an XML-based format. There were 283 files (not limited by user) with a CR larger than 0.9. The average size for this group of files was 2.2 KB, suggesting they may just be too small to compress well.

At first glance, all lines appear to be trimodal. Unlike many of the lines we have seen, the modes appear to be universally smaller than the mean. Deflate 1 may be the exception, but the binning process (described in Section 3.2.1) reduces precision. Also unlike previous distributions, the percentage of files at the modes is quite variable. There was a bit of a difference in GIF (see Table 4.10), but here, there is a difference of almost 10% between LZ4 and XZ 6 & 9.

4.2.9 JPG

JPEG and JPG are interchangeable extensions used for images compressed with some version of JPEG. JPEG is an image compression method only, not a standard

Table 4.13: JPG

Algorithm	Mean CR	Median CR	Mode	SD	A
Deflate 1	0.942	0.982	0.95	0.111	6040.397, $p < 0.01$
Deflate 6	0.940	0.981	0.95	0.113	5996.984, $p < 0.01$
Deflate 9	0.940	0.981	0.95	0.114	5999.496, $p < 0.01$
LZ4	0.950	0.989	0.95	0.108	6082.688, $p < 0.01$
LZ4HC	0.944	0.985	0.95	0.111	5831.245, $p < 0.01$
XZ 6	0.937	0.982	0.95	0.119	5698.467, $p < 0.01$
XZ 9	0.937	0.982	0.95	0.119	5698.345, $p < 0.01$

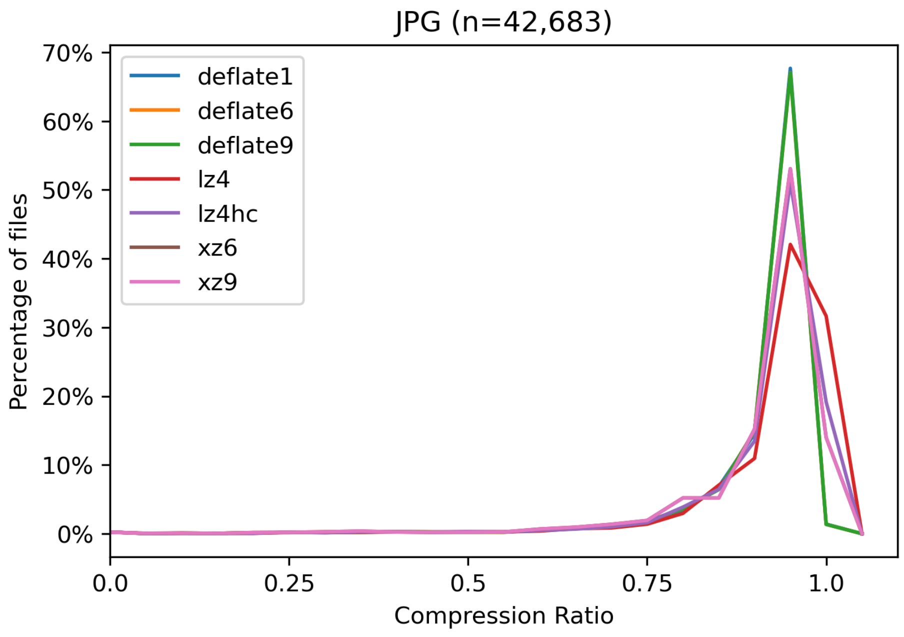

Figure 4.15: Histogram for JPG

for a complete file. As such, pixel aspect ratio, color map, and interleaved bitmap rows must be independently coded [4]. Further complicating this extension, JPEG has many modes. Compression can be lossless, though it is almost always lossy. Lossy compression can encode the file linearly (left to right, top to bottom), in blocks, or hierarchically, which allows the user to view lower-resolution blocks of an image before the higher-resolution portions have been decompressed. Like all already-compressed files, we expect very little compression.

JPG (Table 4.13 and Figure 4.15) was the 6th most common by ranked vote and the 12th overall. The average file size for JPG was 753 KB with the largest file at 66 MB. The lines for JPG follow the pattern that we saw for other pre-compressed data. The mean, median, and mode for each line are very close to 1, and the means are skewed to the left due to left-hand tail.

4.2.10 JS

Table 4.14: JS

Algorithm	Mean CR	Median CR	Mode	SD	A
Deflate 1	0.374	0.371	0.35	0.096	26.773, $p < 0.01$
Deflate 6	0.345	0.340	0.30	0.103	44.739, $p < 0.01$
Deflate 9	0.344	0.340	0.30	0.104	43.293, $p < 0.01$
LZ4	0.515	0.513	0.45	0.129	5.531, $p < 0.01$
LZ4HC	0.466	0.460	0.40	0.143	46.279, $p < 0.01$
XZ 6	0.355	0.346	0.30	0.121	121.820, $p < 0.01$
XZ 9	0.355	0.346	0.30	0.121	121.819, $p < 0.01$

JS is the extension for JavaScript files. We expect JavaScript to compress well because it contains so many patterns. English text compresses very well[2], and many JavaScript functions and variable names use English text. If the programmer who created a file regularly uses one coding style, e.g., puts the left curly brace on the same line as the function signature, patterns will be even stronger.

[2]Using an alphabet of 95 ASCII characters, Brown et al. estimated an upper bound on the entropy of English text at 1.75 bits per letter [48]. Assuming a text file uses one byte per character, that would leave a CR near 0.22.

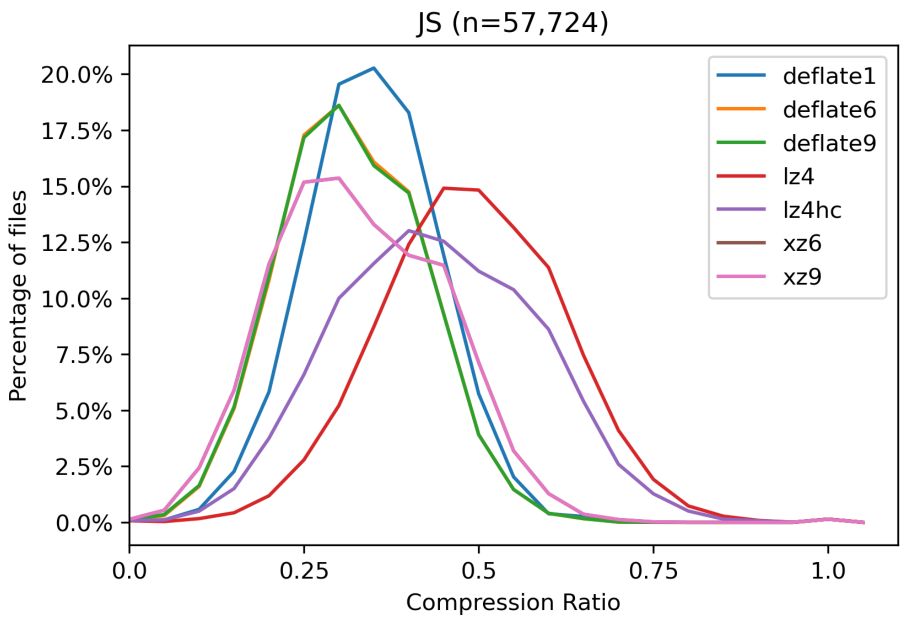

Figure 4.16: Histogram for JS

JS (Table 4.14 and Figure 4.16) was the 3rd most common extension by ranked vote and the 8th overall. The average file size was 31 KB, and the maximum was 51 MB. All JS lines are the closest to normal that we have seen so far. Interestingly, Deflate 6 & 9 performed slightly better than both levels of XZ. Like HTML (Figure 4.14), the percentage of files at the modes is highly variable. There appears to be a difference of 8 or 9% between LZ4HC and Deflate 1.

In other extensions with fairly bell-shaped curves, it's common for LZ4 and LZ4HC to have larger standard deviations, making them look "flatter." Speed is definitely the only reason to choose these compressors, because they stand out from Deflate and XZ much more than the differences between Deflate and XZ.

4.2.11 JSON

JSON, or JavaScript Object Notation, is a language-independent data-interchange format [49]. Its text-based structure makes it human-readable, easy for machines to

Table 4.15: JSON

Algorithm	Mean CR	Median CR	Mode	SD	A
Deflate 1	0.350	0.373	0.40	0.121	380.764, $p < 0.01$
Deflate 6	0.322	0.343	0.40	0.124	318.110, $p < 0.01$
Deflate 9	0.321	0.342	0.40	0.125	325.304, $p < 0.01$
LZ4	0.477	0.515	0.60	0.165	500.484, $p < 0.01$
LZ4HC	0.429	0.467	0.55	0.168	491.170, $p < 0.01$
XZ 6	0.332	0.353	0.40	0.141	364.152, $p < 0.01$
XZ 9	0.332	0.353	0.40	0.141	364.169, $p < 0.01$

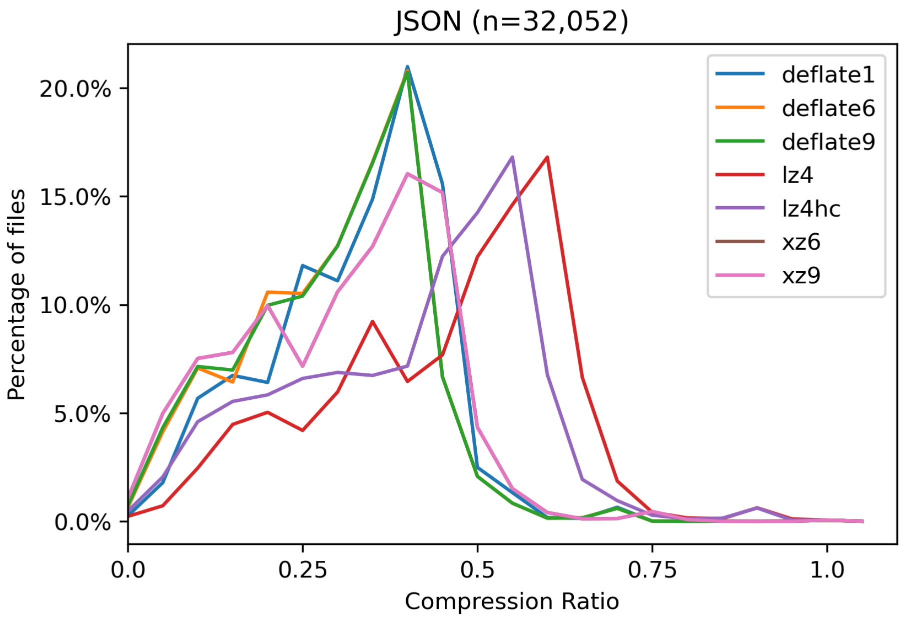

Figure 4.17: Histogram for JSON

parse, and therefore, likely compressible. Compressibility may rely heavily on the data structure(s) the JSON file contains or based on coding styles. For example, a programmer who frequently replaces long field names with the *@JsonProperty* annotation may have less-compressible files compared to a programmer who ignores this feature.

Table 4.16: MP3

Algorithm	Mean CR	Median CR	Mode	SD	A
Deflate 1	0.923	0.956	0.95	0.107	171.563, $p < 0.01$
Deflate 6	0.920	0.951	0.95	0.108	167.214, $p < 0.01$
Deflate 9	0.920	0.951	0.95	0.108	166.742, $p < 0.01$
LZ4	0.935	0.976	0.95	0.107	187.135, $p < 0.01$
LZ4HC	0.924	0.959	0.95	0.108	171.890, $p < 0.01$
XZ 6	0.916	0.947	0.95	0.111	157.784, $p < 0.01$
XZ 9	0.916	0.947	0.95	0.111	157.458, $p < 0.01$

JSON (Table 4.15 and Figure 4.17) was the 8th most common extension by ranked vote and the 22nd overall. The average file size was 56 KB, and the largest file was 585 MB. Deflate and XZ had fairly similar performance, but as usual, LZ4 and LZ4HC had noticeably worse performance. The average file size for the files with a CR larger than 0.8 was 14 KB, suggesting that their poor compressibility is due to smaller file sizes.

4.2.12 MP3

MP3 is the well-known extension for the MPEG-3 audio file format, a lossy compression format that utilizes a Modified Discrete Cosine Transform (MDCT). Its open-source counterpart is Ogg Vorbis [50], identifiable with the extension OGG [51]. Though the OGG extension was more common by total files, MP3 had a higher ranked vote, and MP3 is better known. We chose to analyze only MP3 with the assumption that two MDCT-based formats would have similar compression performance.

MP3 (Table 4.16 and Figure 4.18) was the 52nd most common extension by ranked vote and the 63rd overall. The average MP3 size was 5 MB with the largest file at

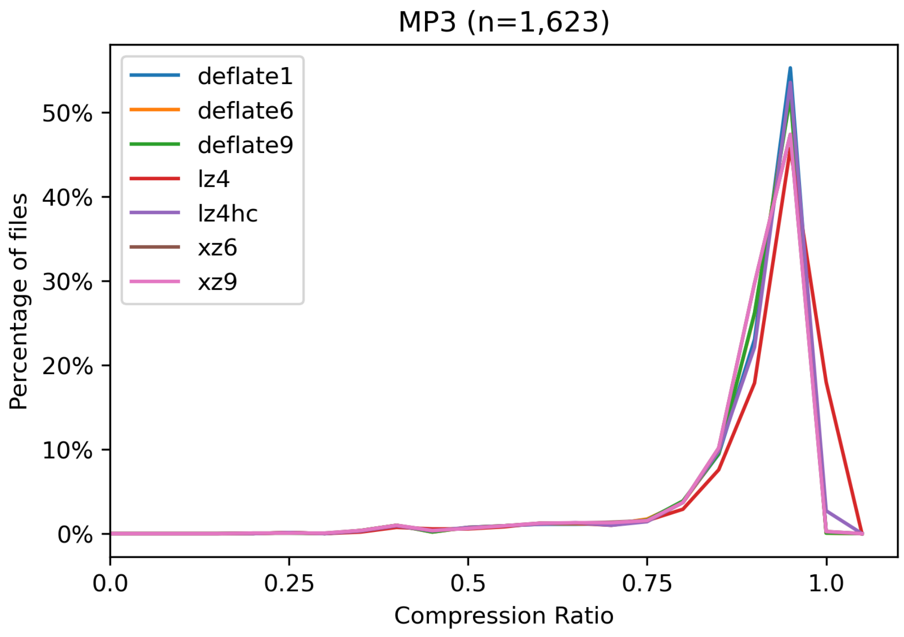

Figure 4.18: Histogram for MP3

Table 4.17: PDF

Algorithm	Mean CR	Median CR	Mode	SD	A
Deflate 1	0.835	0.903	0.95	0.178	765.015, $p < 0.01$
Deflate 6	0.826	0.897	0.95	0.186	747.393, $p < 0.01$
Deflate 9	0.825	0.896	0.95	0.186	745.065, $p < 0.01$
LZ4	0.856	0.915	0.95	0.160	707.205, $p < 0.01$
LZ4HC	0.839	0.901	0.95	0.171	684.823, $p < 0.01$
XZ 6	0.813	0.884	0.95	0.198	656.093, $p < 0.01$
XZ 9	0.813	0.884	0.95	0.198	655.635, $p < 0.01$

991 MB. Each compressor's distribution closely resembles all other distributions for pre-compressed files.

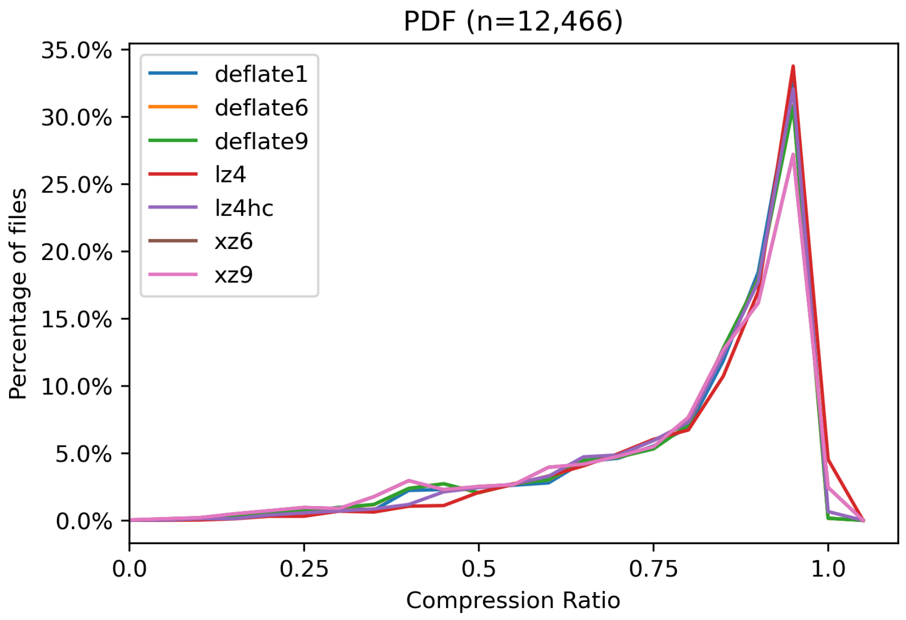

Figure 4.19: Histogram for PDF

4.2.13 PDF

PDF (Portable Document Format) was created in the early 1990s by Adobe Systems [52]. Later-version PDFs can contain interactive content, audio, video, JavaScript, vector graphics, and more. Updates to objects within the document are often appended to the document rather than modifying the original object [53]. Individual objects can be compressed (e.g., images can be compressed with JPEG or Run-Length Encoding), and since version 1.2, the entire document can be compressed with lossless algorithms like Deflate or LZW.

PDF (Table 4.17 and Figure 4.19) was the 26th most common extension by ranked vote and the 41st overall. The average file size was 872 KB with a largest size of 664 MB. The given shape of the lines strongly suggests that the majority of these PDFs were already compressed. Unfortunately, the metadata gives only the version of the PDF and does not tell us whether the entire file or portions of the file are compressed,

Table 4.18: Individual PDF versions using Deflate 6

Version	Mean	Median	Mode
1.0	0.584	0.652	0.75
1.1	0.722	0.838	0.95
1.2	0.787	0.826	0.80
1.3	0.845	0.889	0.90
1.4	0.784	0.845	0.90

though at the minimum we know that text will not be compressed if the version is 1.0 or 1.1. We also have no way to know what each PDF contains. PDFs with many embedded images likely compress very differently from their text-heavy counterparts.

Though the distributions resemble those for pre-compressed data, there is a much longer tail to the right. A great number of PDFs are still very compressible even though the mode is universally at 0.95.

Comparison by Version

Table 4.18 and Figure 4.20 suggest that, as expected, versions 1.0 and 1.1 predict greater compressibility, but $n = 7$ is a very small sample to draw any conclusions. Versions 1.3 and 1.4 have equal modes, but the mean and median for version 1.4 are both lower than those of 1.3.

4.2.14 PNG

PNG is a Deflate-compressed image format created to replace GIF [54]. It supports multiple levels of compression, so while some PNGs may compress slightly more, we expect generally poor performance.

PNG (Table 4.19 and Figure 4.21) was the 2nd most common extension both by

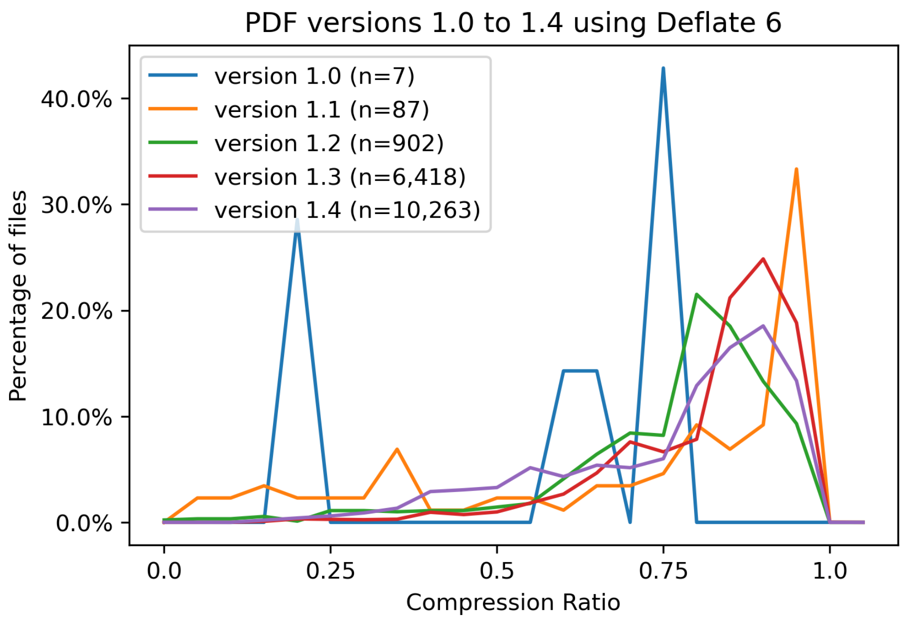

Figure 4.20: Histogram for PDF versions 1.0 to 1.4 using Deflate 6

Table 4.19: PNG

Algorithm	Mean CR	Median CR	Mode	SD	A
Deflate 1	0.930	0.997	1.00	0.160	539.405, $p < 0.01$
Deflate 6	0.929	0.997	1.00	0.162	542.535, $p < 0.01$
Deflate 9	0.929	0.997	1.00	0.163	542.595, $p < 0.01$
LZ4	0.941	1.002	1.00	0.152	603.117, $p < 0.01$
LZ4HC	0.934	0.997	1.00	0.156	564.665, $p < 0.01$
XZ 6	0.930	0.999	1.00	0.167	467.808, $p < 0.01$
XZ 9	0.930	0.999	1.00	0.167	467.808, $p < 0.01$

ranked vote and overall. The average size was 84 KB, and the largest size was 379 MB. Its distributions resembled all other pre-compressed file distributions.

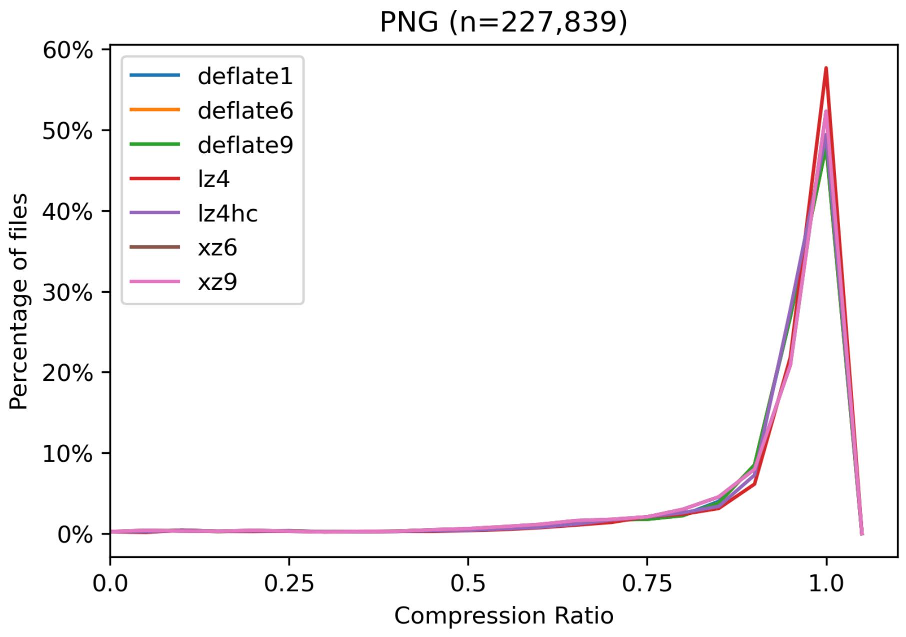

Figure 4.21: Histogram for PNG

Table 4.20: RTF

Algorithm	Mean CR	Median CR	Mode	SD	A
Deflate 1	0.565	0.411	1.00	0.342	335.296, $p < 0.01$
Deflate 6	0.539	0.381	1.00	0.362	333.748, $p < 0.01$
Deflate 9	0.538	0.381	1.00	0.364	330.337, $p < 0.01$
LZ4	0.647	0.577	1.00	0.299	198.314, $p < 0.01$
LZ4HC	0.600	0.528	1.00	0.336	219.204, $p < 0.01$
XZ 6	0.542	0.398	1.00	0.373	285.454, $p < 0.01$
XZ 9	0.542	0.398	1.00	0.373	285.454, $p < 0.01$

4.2.15 RTF

Rich Text Format (RTF) was created by Microsoft in 1987 to hold text with italics, bold-face, multiple fonts, among other features [55]. RTF can include embedded images of several formats, including PNG and JPG. RTF is highly portable across

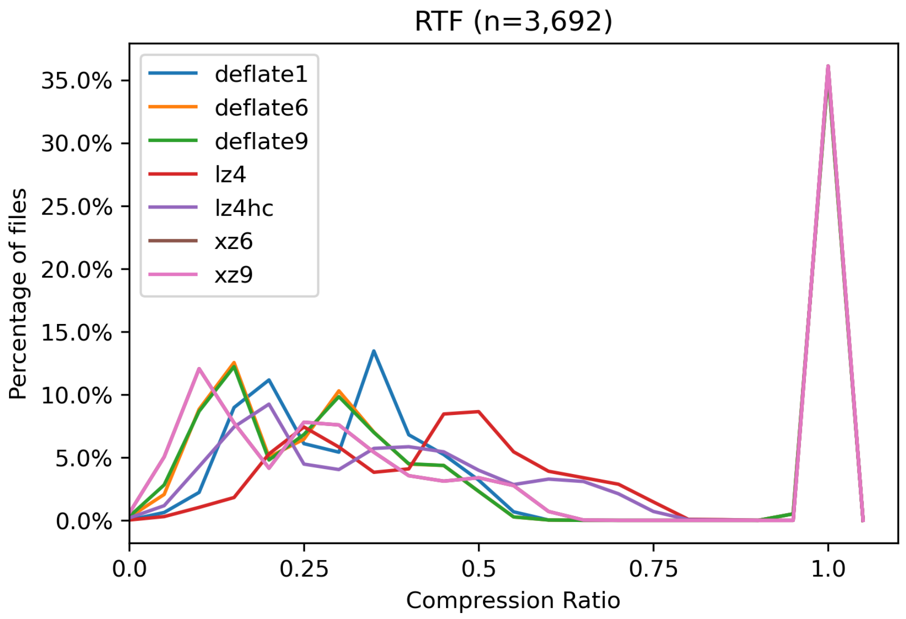

Figure 4.22: Histogram for RTF

operating systems, making it an attractive format and increasing its prevalence.

RTF (Table 4.20 and Figure 4.22) was the 49th most common extension by ranked vote and the 111th overall. We included RTF, since it is still a relatively common and portable document format, and so we considered its results interesting. We were very surprised to find a massive group of totally incompressible files. Upon further investigation, many files simply had "data" as the results of the `file` command. The average CR for these files was 1.01. No version of Rich Text Format supports compression, so we assume these files are something other than Rich Text. If we include only files containing "Rich Text Format" in the metadata, we get Figure 4.23, the distribution we expected for this extension. (This chart did not limit the number of files per user.)

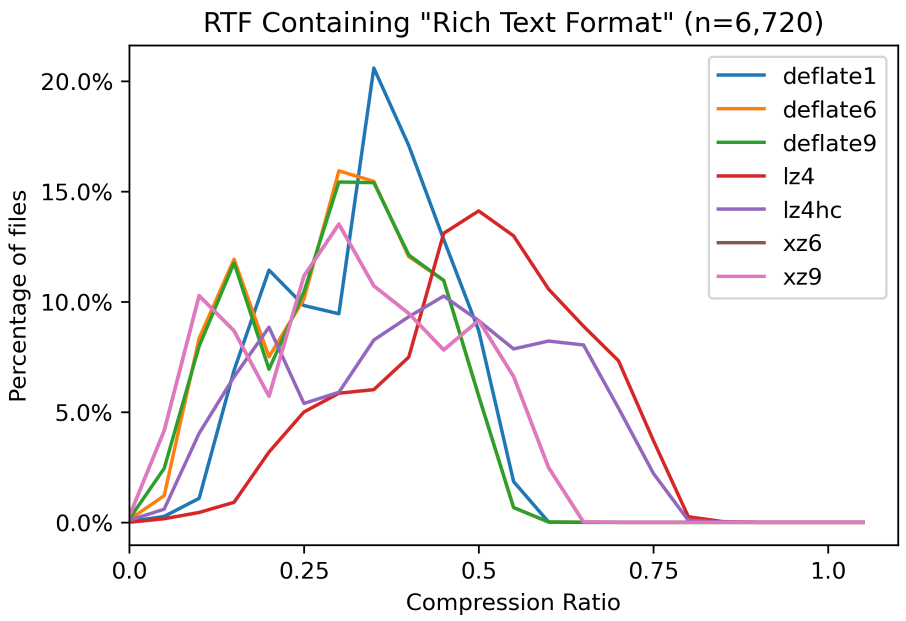

Figure 4.23: Histogram for RTF

Table 4.21: SVG

Algorithm	Mean CR	Median CR	Mode	SD	A
Deflate 1	0.388	0.394	0.35	0.107	140.737, $p < 0.01$
Deflate 6	0.364	0.370	0.40	0.111	82.533, $p < 0.01$
Deflate 9	0.364	0.369	0.40	0.112	85.992, $p < 0.01$
LZ4	0.556	0.566	0.55	0.163	95.158, $p < 0.01$
LZ4HC	0.518	0.527	0.50	0.167	56.495, $p < 0.01$
XZ 6	0.367	0.371	0.30	0.126	31.905, $p < 0.01$
XZ 9	0.367	0.371	0.30	0.126	31.905, $p < 0.01$

4.2.16 SVG

SVG is an abbreviation for Scalable Vector Graphics, an Internet-based static or animated graphic specified in XML [56]. Unlike raster graphics, which are 2D arrays of pixel values, SVGs store descriptions of a graphic as a series of lines, colors, shades, angles, etc. Due to their XML-based structure and simplified nature, we expect SVGs

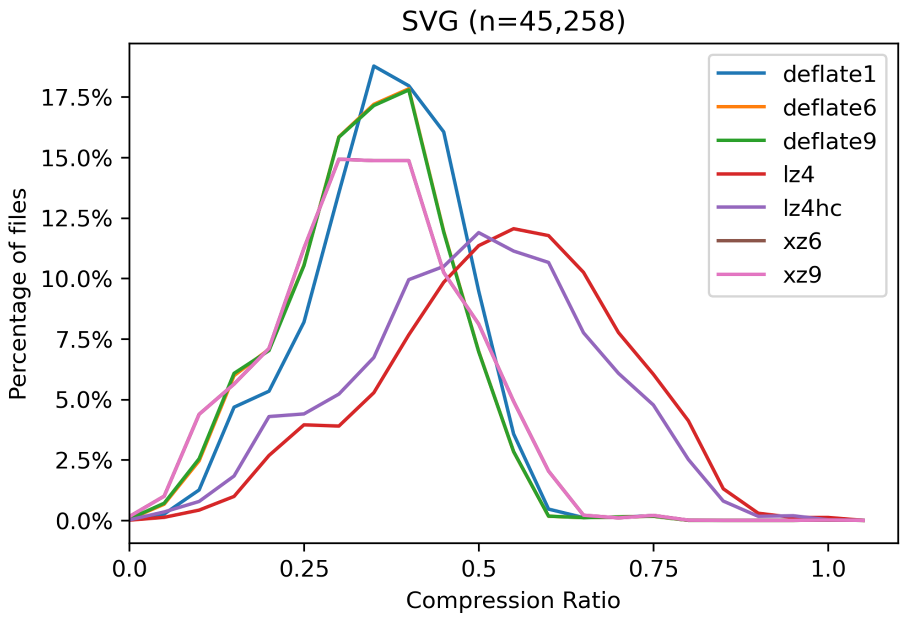

Figure 4.24: Histogram for SVG

to compress well.

SVG (Table 4.21 and Figure 4.24) was the 9th most common extension by ranked vote and the 21st overall. The average file size was 13 KB, and the largest size was 13 MB. LZ4 and LZ4HC performed particularly badly compared to the other five. Unlike the XML-based HTML files, we don't see a cluster of files near 1. Instead, these lines are relatively normal with the exception of the small spike between 0.1 and 0.25. For Deflate and XZ, nearly all files have a CR less than 0.75.

4.2.17 TTF

TTF, or TrueType Font, is a font format used by Mac, Linux, and Windows [57]. TrueType uses a "hinting language" to help a rasterizer appropriately display the font at all sizes and resolutions, improving readability and visual appeal [58]. TTF files contain bytecode to be executed by the rasterizer. TTF files are not compressed; WOFF and WOFF2 are container formats designed to wrap and compress TTF [59].

Table 4.22: TTF

Algorithm	Mean CR	Median CR	Mode	SD	A
Deflate 1	0.582	0.581	0.55	0.102	98.335, $p < 0.01$
Deflate 6	0.552	0.550	0.50	0.107	100.738, $p < 0.01$
Deflate 9	0.551	0.550	0.50	0.108	100.223, $p < 0.01$
LZ4	0.709	0.718	0.70	0.107	80.868, $p < 0.01$
LZ4HC	0.637	0.641	0.60	0.114	55.854, $p < 0.01$
XZ 6	0.471	0.467	0.45	0.123	76.770, $p < 0.01$
XZ 9	0.471	0.467	0.45	0.123	76.751, $p < 0.01$

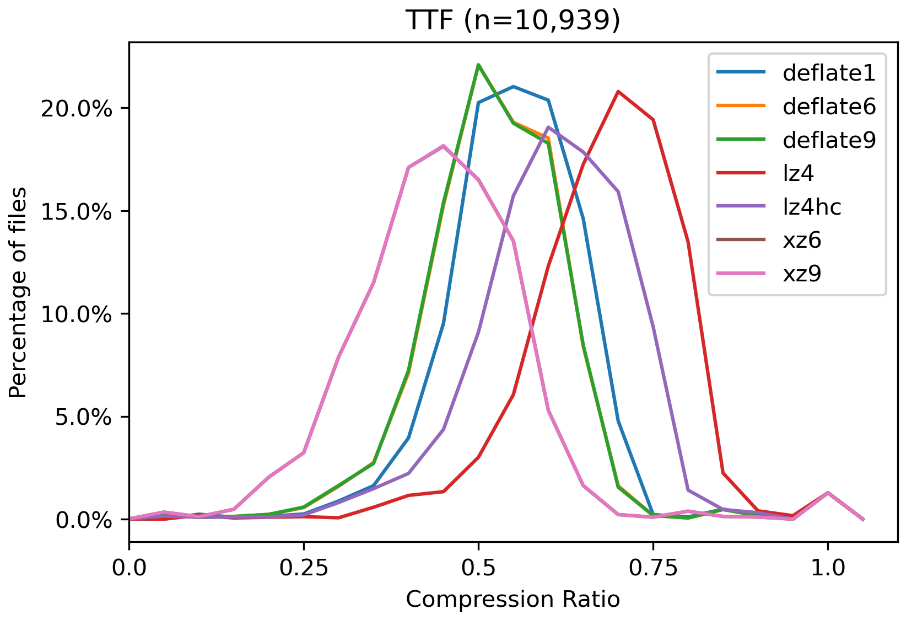

Figure 4.25: Histogram for TTF

We expect TTFs, like any bytecode files, to compress well.

TTF (Table 4.22 and Figure 4.25) was the 15th most common by ranked vote and 66th overall. TTF had an average file size of 402 KB and a largest file size of 48 MB. The TTF lines appear somewhat normal, though the small cluster of files near 1 break from the pattern. The average file size for files with a CR greater than 0.9

Table 4.23: TXT

Algorithm	Mean CR	Median CR	Mode	SD	A
Deflate 1	0.362	0.366	0.40	0.145	61.467, $p < 0.01$
Deflate 6	0.333	0.328	0.30	0.149	53.981, $p < 0.01$
Deflate 9	0.332	0.327	0.30	0.150	53.114, $p < 0.01$
LZ4	0.504	0.508	0.55	0.202	32.825, $p < 0.01$
LZ4HC	0.452	0.441	0.40	0.206	68.612, $p < 0.01$
XZ 6	0.333	0.321	0.30	0.168	97.042, $p < 0.01$
XZ 9	0.333	0.321	0.30	0.168	97.098, $p < 0.01$

was 787 KB, so small file size was not likely a contributor. A great number of these files contained the substring "EmojiRegularVersion." Of all files whose metadata contained this substring, the smallest CR in the XZ 9 table was 0.47, which would be placed in the 0.5 bin during the histogram binning process. This means that all Emoji font files' CRs were larger than the mean, median, and mode. It makes sense that a highly detailed font such as Emoji would have less redundancy. It could be that the cluster near 1 is largely custom fonts.

4.2.18 TXT

TXT is generally associated with unformatted text files [32], which is a very broad category of files that contain mainly ASCII or or other printable characters. Though we can expect English text to compress fairly well, it would be naïve to assume the contents are even close to English. The variety of possible formats will likely create a multimodal distribution.

TXT (Table 4.23 and Figure 4.26) was the 7th most common extension by ranked vote and 29th overall. The average file size was 222 KB, and the largest was 891 MB.

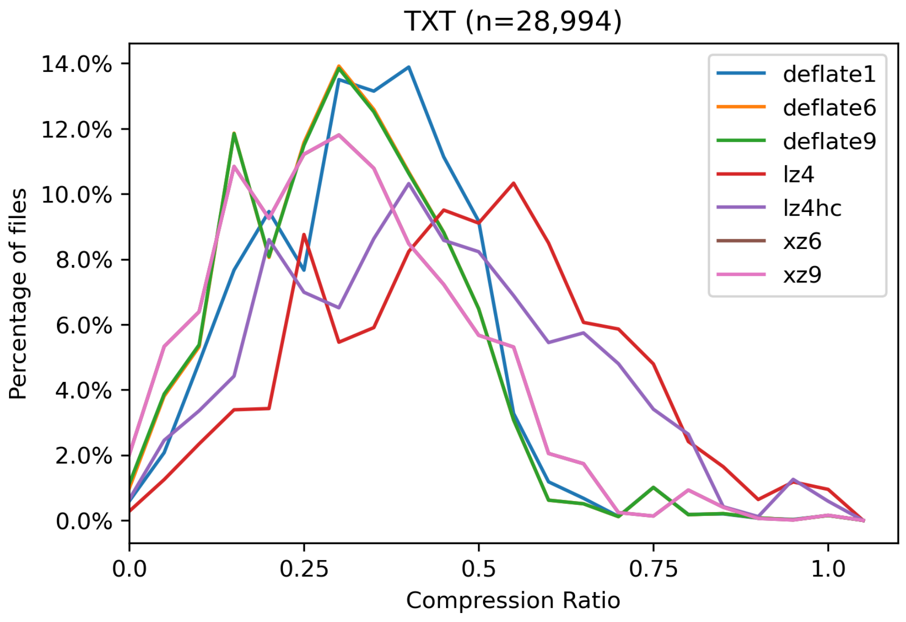

Figure 4.26: Histogram for TXT

The average file size was relatively low at 210 KB, but the largest file reached 869 MB. For all compressors, the mean, median, and mode were very close. The largest distance from the mean to the mode was 0.03, a tie between Deflate 6 & 9. LZ4 had the largest distance from the median to the mode at 0.042. Except for LZ4HC and LZ4, all compressors had a sharp decline in CRs after about 0.6.

The cluster of files with CRs larger than 0.9 is likely due to files breaking with the standard for the TXT extension. The average file size for this cluster was about 300 KB, so it is unlikely that small files were the culprit. Some metadata from this cluster included "Targa image data", "gzip compressed data", and "OpenSSH RSA public key." This perfectly exemplifies a critical flaw in the extension approach: extensions make no guarantee about the data they contain. In the RSA public key case, one could argue that .txt does not break from the convention, because RSA public keys are usually stored in a text representation; however, we expect human language and encryption keys to have dramatically different entropy.

Table 4.24: WAV

Algorithm	Mean CR	Median CR	Mode	SD	A
Deflate 1	0.739	0.767	0.90	0.180	52.502, $p < 0.01$
Deflate 6	0.730	0.759	0.90	0.188	55.727, $p < 0.01$
Deflate 9	0.730	0.759	0.90	0.188	55.937, $p < 0.01$
LZ4	0.889	0.947	0.95	0.155	319.042, $p < 0.01$
LZ4HC	0.841	0.898	0.95	0.179	200.610, $p < 0.01$
XZ 6	0.629	0.628	0.45	0.209	28.310, $p < 0.01$
XZ 9	0.629	0.628	0.45	0.209	28.269, $p < 0.01$

TXT files were largely compressible, with most files compressed to at least 75% of their original size. The file contents appear to have varied significantly. Though most metadata suggested ASCII, UTF-8, UTF-16, or ISO-8859 characters, some files appeared to contain public keys, git commits, or other incompressible data. The category "text files" may be too broad to make any estimations of compressibility.

This extension shows how different the utility of the mean CR can be based on extension. The lines for Deflate and XZ show that the system can confidently assume a TXT file will compress below 0.75. It cannot make the same strong assumption for LZ4. Even though LZ4 will be much faster than Deflate, there is a non-trivial chance that LZ4 will waste time reducing a file by only 10% of its original size.

4.2.19 WAV

WAV refers to Waveform Audio File Format, which is generally uncompressed audio, though may be compressed (all analog-to-digital audio conversion incurs some amount of information loss, but we will explain in Section 4.2.19). WAV supports both mono and stereo sound, variable sample rates, and multiple encoding formats,

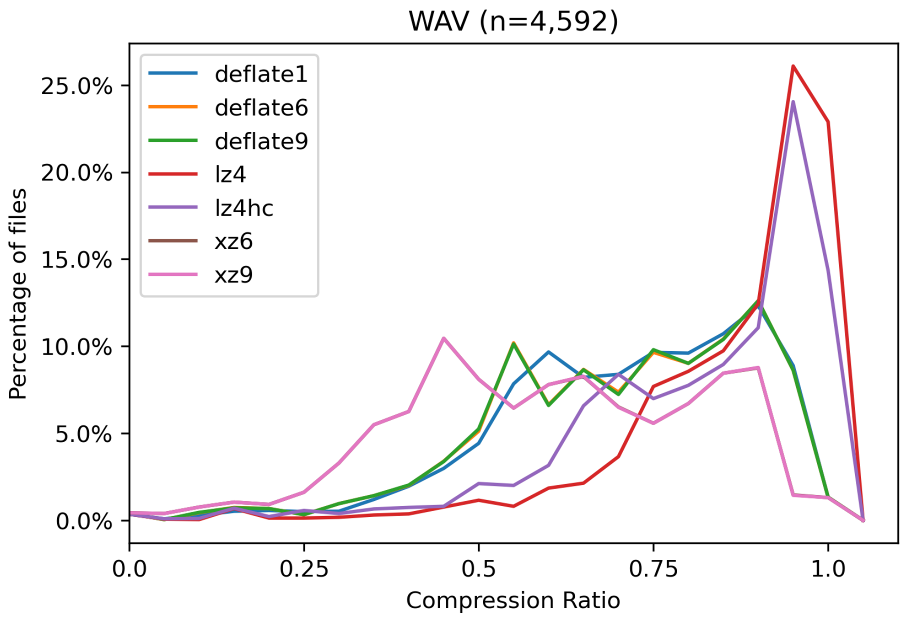

Figure 4.27: Histogram for WAV

including Pulse Code Modulation (PCM), Adaptive Differential Pulse Code Modulation (ADPCM), A-law, and μ-law [60]. We expect the flexibility of encoding to produce a multimodal distribution.

WAV (Table 4.24 and Figure 4.27) was the 28th most common extension by ranked vote and 48th overall. The average file size was 801 KB, and the largest size was 891 MB. LZ4 and LZ4HC had the worst performance, with mean, median, and mode fairly tight about 0.95. XZ 6 and XZ 9 had virtually identical results and achieved CRs for most files lower than 0.9.

The WAV data clearly suggest a multimodal distribution. The several dips and peaks in the XZ compressors' lines show that there are probably multiple underlying distributions. XZ's mode at 0.45 is only slightly larger than the mode near 0.9, and so using 0.45 would incur significant penalties for many marginally compressible files. Deflate suffers from the opposite problem: the modes for all three levels are at 0.9, but Deflate 1 has a mode near 0.6, and Deflate 6 & 9 have a mode near 0.55. Using the mean CR, these three compressors would likely miss some opportunities to save time and energy.

The largest noticeable change from the original distribution happens in Figure 4.28 when we plot 8-bit PCM mono audio samples. Here, the lines shift left, suggesting that this cluster of files is uncompressed or at least more compressible. The 16-bit PCM mono files, shown in Figure 4.29, did not follow this same trend. The 8-bit PCM stereo files, shown in Figure 4.30, seemed about as compressible as the 8-bit PCM mono, but there were only 33 files in this sample.

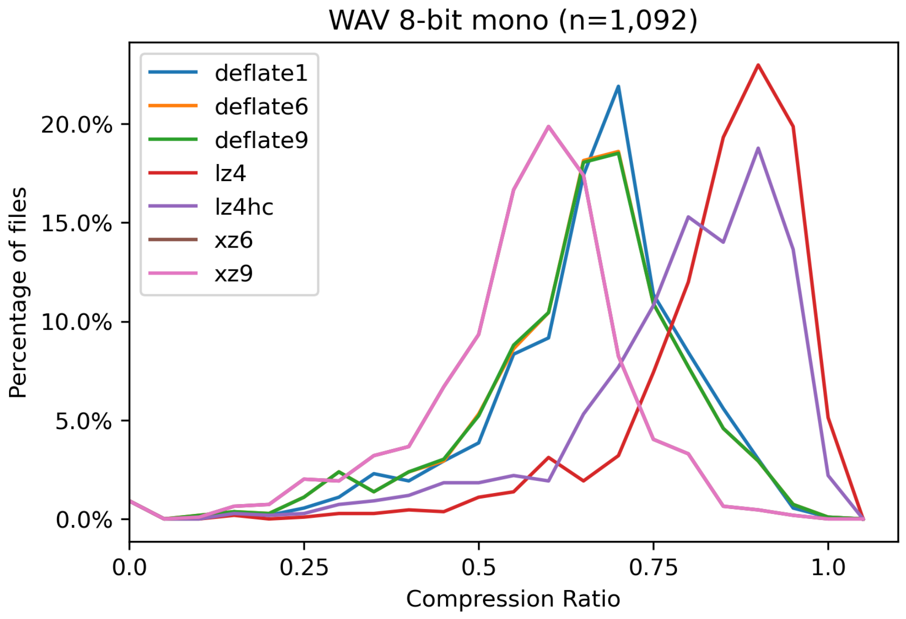

Figure 4.28: WAV audio, 8-bit mono

To understand the difference in compressibility between 8- and 16-bit files, we need to briefly cover how WAV audio is encoded.

WAV Quantization

Converting a continuous signal into discrete steps will always lose some amount of information; however, the number of steps and step size determine information

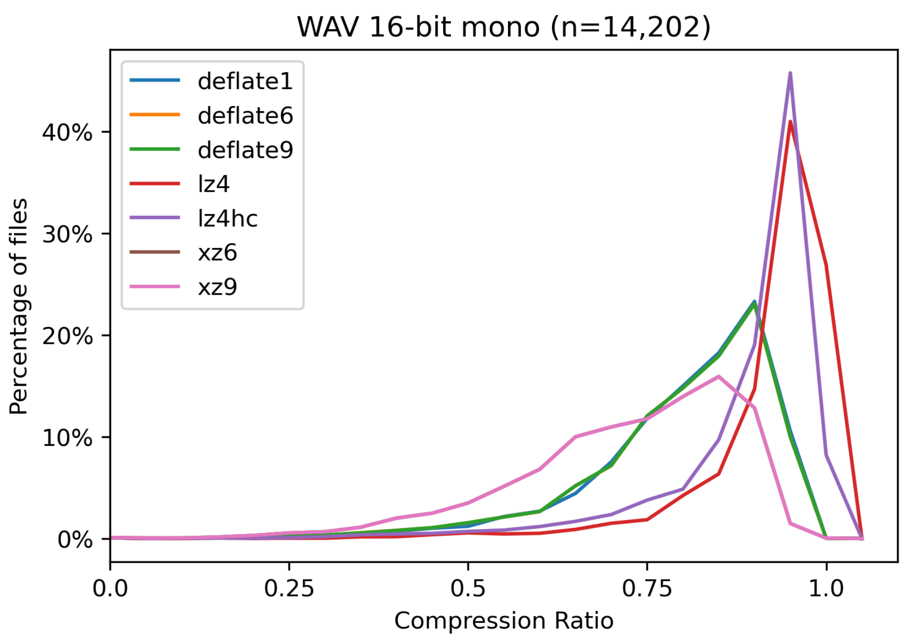

Figure 4.29: WAV audio, 16-bit mono

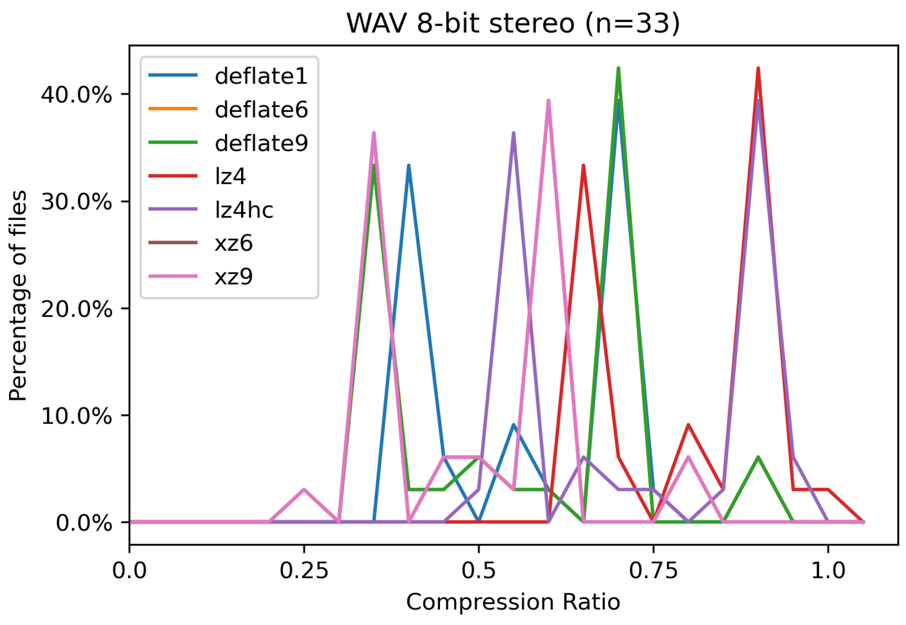

Figure 4.30: WAV audio, 8-bit stereo

loss. PCM encodes each audio sample individually using constant-size steps. The bit resolution, in this case typically 8-bit or 16-bit, determines how many steps the audio will have. Since the step size will not change, PCM-encoded data is considered "uncompressed."

This means the compressibility difference between 8- and 16-bit PCM files is likely due to the number of steps. 8-bit samples allow for 256 possible pitches sounds, which essentially is a 256-character alphabet. The 16-bit files can have $2^{16} = 65,536$ possible sounds. The larger alphabet is likely breaking up some patterns.

ADPCM does not encode each sample alone, but rather it encodes the difference between the previous sample and the current sample [61]. Additionally, it changes the step sizes dynamically and reduces necessary number of steps, which requires fewer bits to encode a step, but results in some information loss. Thus, ADPCM-encoded data is considered "compressed." A-law and μ-law are also compressed encodings, but we did not have enough of these files to analyze them.

At least 90% of WAV files were encoded with PCM or ADPCM[3]. As expected, separating PCM from ADPCM-encoded files showed a large difference in compressibility. There are some clear differences between the PCM data, shown in 4.31, and the ADPCM data, shown in 4.32. Figure 4.31 shows many PCM files are compressible below the mode, with some at a CR of 0.25. Figure 4.32 shows ADPCM files are much more closely packed near 1. There were no discernible differences between ADPCM mono and stereo, and so we did not include their charts.

4.3 Comparison among File Formats

We chose to compare the performance of compressed formats to see how efficient compression is for each format relative to others. If all formats have very efficient compression, then we assume their mean compression ratios should be somewhat

[3]90% includes only files for which we had metadata and could therefore determine the encoding.

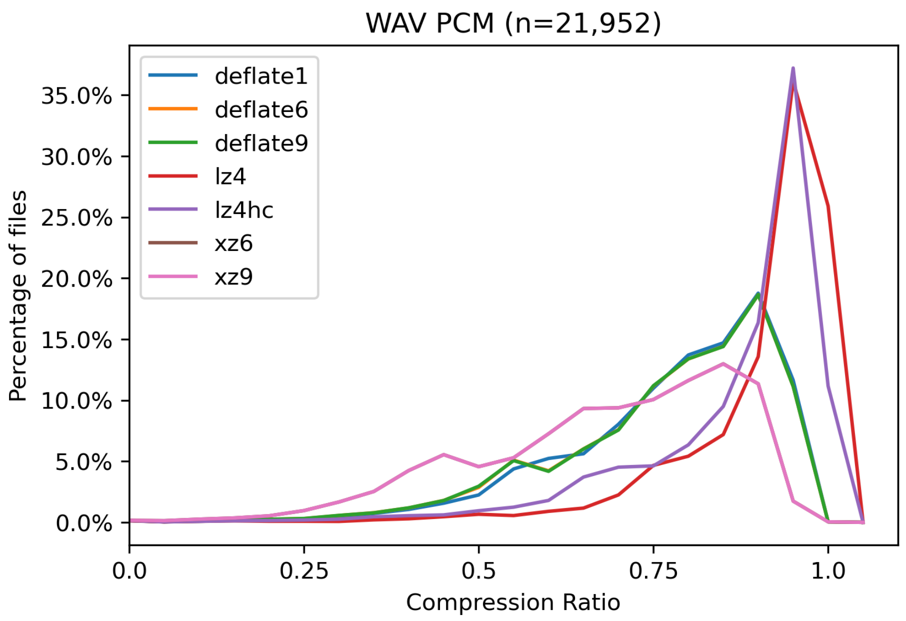

Figure 4.31: WAV audio, PCM samples

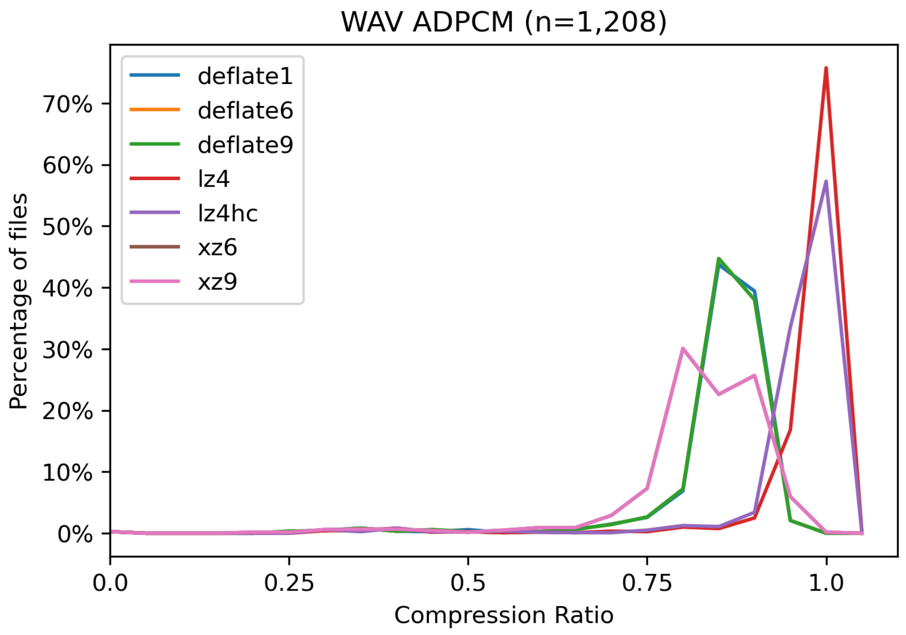

Figure 4.32: WAV audio, ADPCM

equal and their histograms should have very similar shapes.

4.3.1 Compressed File Formats

The extensions we analyzed are DOCX, GIF, GZ, JPG, MP3, PNG, and ZIP. These formats are *always* compressed, unlike the extensions PDF and WAV, which are only *usually* compressed. To make the histograms easy to distinguish, we charted only the Deflate 6 results. We chose Deflate 6 because being the default compression level for both zip and gzip programs (see Section 2.2.1), it is the most common compressor of the seven we used.

Table 4.25: Compressed Format Performance with Deflate 6

Ext.	Mean	Median	Mode
docx	0.865	0.871	0.80
gif	0.892	0.981	0.95
gz	1.000	1.003	1.00
jpg	0.954	0.989	0.95
mp3	0.922	0.949	0.95
png	0.937	0.999	1.00
zip	0.891	0.984	0.95

Overall, the lines all followed a very similar shape. DOCX was the exception, but Figure 4.8 showed that many DOCX files would compress somewhat well. Of these extensions, DOCX is the only one expected to be mostly textual data.

The least surprising result was GZ performance. Though some gzip files may have been compressed with a level lower than 6, we do not expect gzip to compress its own output any further. It is also not surprising that Deflate 6 poorly compressed JPG and PNG files, since JPG is lossy compressed and PNG uses a version of Deflate (see Section 4.2.14). We did expect some GIFs to compress fairly well for the reasons outlined in Section 4.2.6.

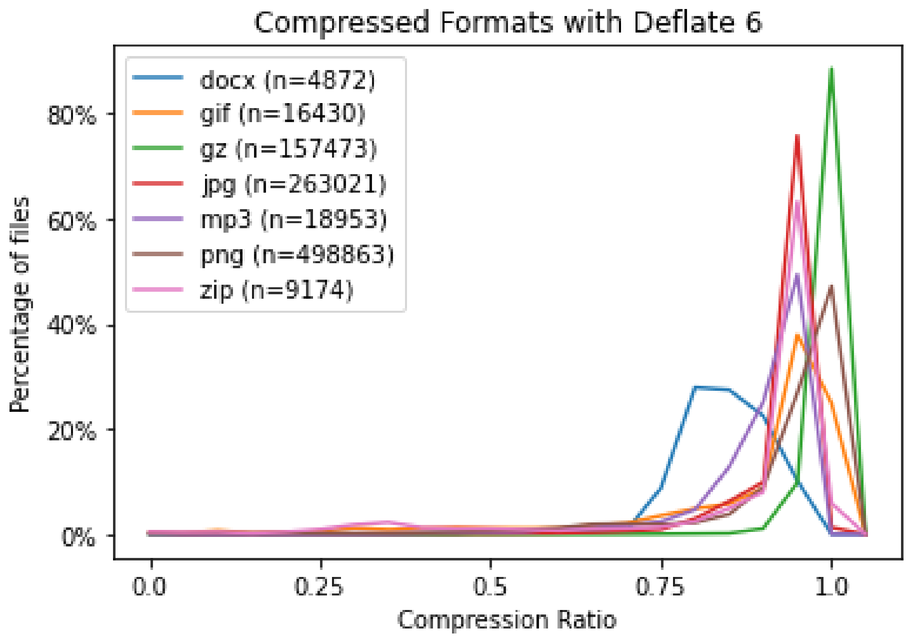

Figure 4.33: Compressed Format Performance with Deflate 6

It is interesting that the mean CR for ZIP is somewhat low, especially relative to GZ. The primary difference between zip and gzip is that zip is an archive format, while gzip requires a separate application (like tar) to archive. This means that zip compresses files individually, while gzip compresses one single file. Maybe zip is not detecting patterns across files, but gzip, compressing an entire archive at once, can detect these patterns. We can see that there is a small cluster of files with a CR near 0.35, and so this likely has some effect on the mean. There is something strange about the zip files, however. A total of 857 files had a CR less than 0.5, and 79 files had a CR less than 0.1. Of the 50 files with the lowest CRs, two were labeled "data" (the lowest CR was 0.001 and was labeled "data"), three were labeled "AppleDouble encoded Macintosh file," and one was labeled "Mozilla archive omni.ja." Nevertheless, only eight files with a CR less than 0.5 had metadata that did not include the string "Zip archive data"[4].

[4]This number does not include files with no metadata. Some Windows machines did not have a

It is somewhat unexpected that a number of MP3s were further compressed. Figure 4.33 shows that many MP3s compressed better than other formats. Like GIF, JPG, and ZIP, the mode was at 0.95, but the mean was 0.922.

4.3.2 Source Code Formats

Table 4.26: Source Code Files with Deflate 6

Ext.	Mean	Median	Mode
c	0.316	0.302	0.25
cpp	0.318	0.313	0.25
h	0.343	0.346	0.35
java	0.344	0.338	0.25
js	0.358	0.357	0.40
py	0.304	0.297	0.25

We chose to compare results from source code files C (C language), CPP (C++), H (C/C++ header files), JAVA (Java), JS (JavaScript, see Section 4.2.10), and PY (Python) files using the Deflate 6 compressor. Even though these are separate languages, we expect source code to compress similarly, specifically because they all follow common patterns. C & CPP files may have the most similar compression, since they are such closely related languages. PY has the most dissimilar structure, though it will likely be close enough to compress like the other extensions.

Surprisingly, PY and C appear to compress more similarly than C and CPP, at least according to the shape of the lines in Figure 4.34. Table 4.26 shows that these three source code files have near identical means — a difference of 0.003 would mean C compressed an average of 0.3% more than CPP and PY.

method to return results of the `file` command.

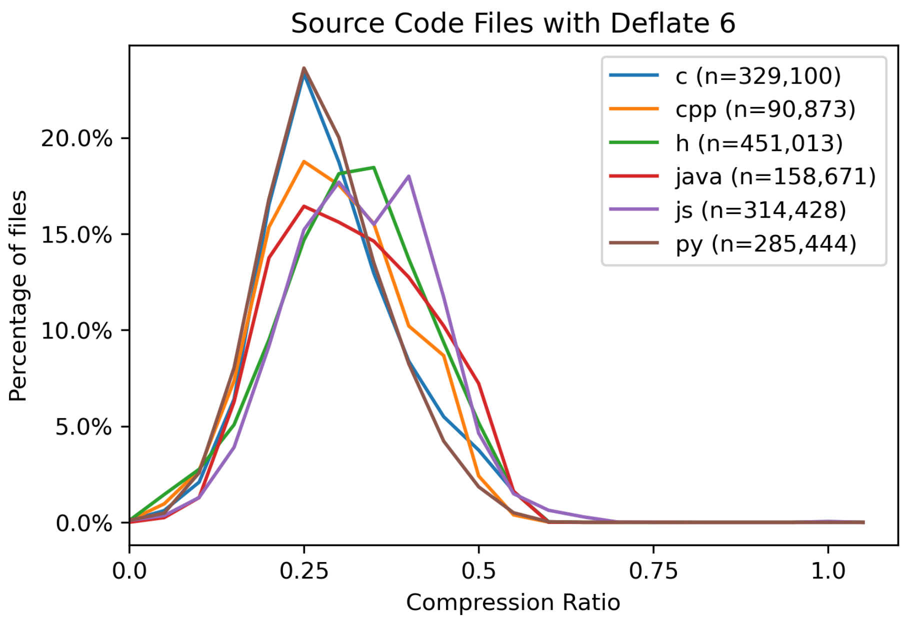

Figure 4.34: Source Code Files with Deflate 6

All means were larger than the modes for every extension. The means and modes also had very little difference, the range being 0.017 to 0.068. JAVA's relatively poor performance could be related to the average file size: JAVA had an average size of 11 KB, all other extensions had 15 to 31 KB.

One reason we did not do a deeper dive into all of these extensions was the average file size. Even JS, having an average of 31 KB, is fairly small when considering whether to compress. Compressing a single file requires some amount of overhead regardless of file size. When the file being compressed is 2 MB, the overhead is trivial, but when the file is 5 KB, overhead should be considered. As we discuss in Section 5.2, an AC system will doubtfully try to compress any file smaller than the filesystem's page size. In total, 83,025 of 158,671 JAVA files (52.3%) were smaller than 4 KB, the default page size for NTFS and ext4 filesystems. This means that even though a JAVA file usually compressed to less than half its original size, compression usually provided little to no space savings.

5 Discussion

In this chapter, we make some higher-level observations about the results in the previous section.

5.1 Normality

We tested a total of 18 extensions for normality. Our results strongly suggest that no file extension has a normal distribution of compressibility for the three algorithms and seven total compressors we used. This is supported by the agreement of three tests for normality — Kolmogorov-Smirnov, Anderson-Darling, and Chi-Squared — all suggesting we should reject the null hypothesis that these distributions are normal ($p < 0.01$ for AD, $p < 0.001$ for both KS and Chi-Squared).

There were several extensions for which not all normality tests agreed. 22 extension-compressor pairs did not have significant evidence to reject the null hypothesis for one of the three tests performed. Of these 22, 16 passed the KS test and 6 passed the Chi-Squared test (there was no overlap). This means that for all of these extensions, two of three tests still returned not normal. While these results may warrant more testing, we do not consider it evidence of normality. All raw results are in Table A.1.

Some extensions appeared to follow a unimodal, somewhat bell-shaped curve even though they were quantitatively not normal. The departure from normality may be explained by some anomalous files that according to their metadata, may contain data other than what is associated with their extension. For example, JS (4.16) appeared to have curves closest to normal. The XZ compressors along with Deflate 6 & 9 appear somewhat bimodal for JS, but LZ4 looks very close to the characteristic bell curve. We built the JS distributions with 57,724 files. Of these, 466 were “data” according the metadata we gathered, while we would expect “ASCII,” “UTF-8,” or “text” to be in that metadata.

Perhaps the LZ4 curve was not normal due to the cluster of the files with CRs near 1, which appears as a slight bump; however, a back-of-the-envelope test shows that the bump near 1 might not be the culprit. There are 137 files in the LZ4 data with a CR greater than 0.9 and 96 files with a CR less than 0.128, which are three standard deviations from the mean. That means 99.60% of files were within three standard deviations, which is very close to the expected 99.73%. Additionally, there were 87 files with a CR greater than 0.9665 (3.5-σ events), leaving 98.49% of files within 3.5 standard deviations, again, not so far off from the expected 99.95% (3.5 standard deviations less than the mean is a negative number).

The larger factor, then, must be the slightly misshapen center of the curve. LZ4 must have a second mode slightly right of 0.45, just as the other curves appear to have a rightward mode. The difference between the mean and the mode for LZ4 is 0.065, which shows there is some skew. Since the mean is slightly larger than the mode, the small bias towards larger CRs might have very little effect on overall performance, since in theory the AC system would have slightly better compression performance than the mean suggests.

5.2 File Sizes

Our experiment looked at all files between 1 KB and 1 GB. While we would have liked to learn how files larger than 1 GB would compress, this upper limit was created to prevent Comprestimator from using too much memory and/or swap on participants' computers. We do not know how many files this limit excluded. At the opposite end of the spectrum, the 1 KB file size could also be adjusted. We chose 1 KB as the lower limit considering that virtually no system would have a page size smaller than 1 KB, and so compressing such small files would not save any space on disk. Additionally, network packets are commonly about 1 KB. Many page sizes, however, are 4 KB or larger. If future work finds that file size plays a large role in the mean CR for a particular extension, then any AC system would need to use a distribution that excludes all files below its page size or whatever the size of the smallest container would be.

5.3 Mean as an Estimator

Our results show that the utility of using the mean as a compressibility estimator varies from extension to extension. The mean is an especially bad estimator for extensions that have very flat distributions, like BIN (4.2.2), or extensions that are multimodal with the modes very far apart, like DLL (4.2.3), EXE (4.2.5), and WAV (4.2.19). Especially for DLL and EXE, the using the mean would be an utter disaster, because their distributions show that the file will almost certainly have a CR much higher or much lower than the mean. The best example is XZ, which has a mean CR at 0.55, but the modes are clearly at 0.3 and 1.

The mean is also an inaccurate prediction when a small number of highly compressible files shifts the mean below the mode. The best examples are GIF (4.2.6) and PDF (4.2.13). With GIF, a few files with CRs between 0.25 and 0.5 shift the mean CR for XZ 6 & 9 0.07 lower than the mode, which is at 0.95. For LZ4HC, the mean is 0.09 lower than the mode. The distributions for PDF also have a large mode at 0.95, but the long tail to the left skews the mean 0.14 lower for XZ. Even when we separate PDFs by version, the means are universally skewed to the left. Compression decisions based on the mean in distributions like this are more likely to result in CRs closer to the mode, which has worse compression.

There is one more problem with the mean: using it assumes that the files that are really more compressible will cancel out the effect of files that are less compressible, and that this will be worthwhile in the long run. In reality, this may not be true. The unexpected benefit from a more-compressible file may not be equal to the penalty of one less compressible. The imbalance between benefit and penalty might even differ between machines. This would imply that each individual machine would need to measure the average penalty and average benefit in order to calibrate what the estimated compressibility should be for each extension.

Lastly, the mean changes potentially dramatically depending on the compressor. In general, LZ4 and LZ4HC had worse compression performance, but how worse they were was variable. For JS (Figure 4.16), LZ4 had CRs up to 0.25 higher than Deflate. LZ4 and LZ4HC's significantly worse performance also appeared in the results for JSON (Figure 4.17), SVG (Figure 4.24), TXT (Figure 4.26), and especially WAV (figure 4.27). Though LZ4 is a very fast compressor, it greatly underperforms Deflate and XZ.

5.4 Mode as an Estimator

For the majority of extensions, the (highest) mode was larger than the mean CR. The difference may be as small as 0.001, as shown in the table for TTF 4.22, or as large as 0.454, as shown in the table for DLL 4.6. Since no results for an extension-compressor pair appeared to be from one single distribution (no set of sample means followed a normal distribution), it is very difficult to choose one mode value that would cover all sub-distributions. In general, the mode would be a very cautious estimator. It is typically better to avoid wasting resources by compressing the incompressible than missing opportunities to compress, but using the mode represents a strategy that is "pessimism over accuracy."

Specifically for pre-compressed files, the mode may be an acceptable estimate, because virtually all modes for pre-compressed files were 0.95 or 1.00, both discouraging attempts to compress. Particularly for JPG (Figure 4.15), the mode covered almost 70% of files compressed with Deflate 6 or 9. It appears that for the majority of extensions indicating a pre-compressed file, it is indeed a bad decision to attempt compression.

5.5 File Extension as a Predictor of File Type

One concern with using file extensions to predict compressibility is that the file extension does not guarantee file format or type. According to our results, we should all share this concern. First, the most common extension in our data was no extension at all! This "null extension" does not communicate useful information. It is a severe problem that the most common extension is one of the least helpful.

Also, many files appeared not to conform with their extension's conventional format. The best example is RTF (see Figure 4.22) where almost one in three files was not Rich Text. 10 of 24 participants each contributed between 147 to 154 of these files. Such a consistent number suggests they may be system files. Fortunately, the metadata was extremely helpful in removing the incompressible offenders. Metadata also revealed distinguishing characteristics of WAV files, where the substrings "PCM" and "ADPCM" helped separate compressible and incompressible files.

JPG had a number of offenders, including “gzip compressed data,” “CSV text,” “ASCII text, with CRLF line terminators,” “Python script,” “PE32 executable (DLL),” and “XML 1.0 document.” Most importantly, these files were from participants who presumably were not trying to game their own systems.

These examples show that people designing AC systems to use file extensions should account for the problem of inaccurate extensions in some way.

5.6 Future Work

This experiment creates many opportunities for future work with the current dataset. Most obviously, we analyzed only 18 extensions. It would especially be interesting to analyze extensions like MP4, MOV, WMV, and other extensions associated with video. As noted in Appendix B, we skipped a number of system-dependent files. Since these files are extremely common, they may be worth investigating. The dataset also includes a number of metrics that we have not yet used, like the byte-count, entropy, and compression time for each file.

We have evidence that for at least some extensions, the metadata from the Unix `file` command better predicts compressibility than the file extension alone. Since there is an overwhelming number of extension-metadata pairs, data mining could potentially find some combinations that reliably predict compression performance. Another machine learning project could find file properties that predict to which underlying distribution a file belongs. Many other questions could be answered through machine learning or data mining, such as: How does file size correlate with compressibility? Does a user’s average C source file compressibility reliably predict other source code file compressibility? Does their WAV file compressibility somehow predict their MP3 compressibility?

Though we know the theoretical maximum compression is the file's entropy, it would be interesting to compare the entropy to the observed CRs to see if entropy predicts real-world compressibility, not just theoretical. Similarly, it would be worthwhile to compare the bytecount to observed CRs. Bytecount has so far been used in network traffic compression [25] and local filesystem compression [62], and comparing it to CRs and entropy from our dataset could give more information on its utility for future projects.

5.6.1 Participant Diversity

It would be helpful to expand our list of participants to reach more people who are not part of the computer science world. It would also help to include a wider range of participant ages and include people from different geographic regions and who speak languages other than English. Building a more-diverse dataset may significantly change some results. It would be extremely helpful for future AC development to know whether geographic region or primary language will affect compression performance.

5.6.2 Time Estimation

Estimated time to compress is a very important metric for making compression decisions. Although we collected information on compression time, we did not include it in our analysis. Understanding the interaction of runtime, compressors, and extensions would compliment this work well, especially if the runtime and CR can be paired to estimate a byte reduction per second for a particular file.

5.6.3 Implementation in Real-World AC

The largest question for work was whether file extensions can be of any use for AC systems. The best test would be a real-world implementation. An implementation could show us how useful both time and compressibility estimates are when based on mean compressibility. This could also show how much time is saved by making a quicker decision. Even if compression takes longer than expected, the reduction in decision time will offset at least some of the time penalty of compression. If the file extension is the only estimator, decision time will be reduced for all files, both for those that compressed faster than their estimate and those slower. If the extension is the sole estimator for only a subset of files, decision time will slightly increase for other files, though the system should experience a net reduction. AC systems for smaller machines could especially benefit from this type of implementation.

6 Conclusion

In this thesis we provided strong evidence refuting the assumed claim that file compressibility follows a normal distribution. Using a tool we built to collect compressibility information of over 12 million files, we estimated the probability distributions of file compressibility for 18 different extensions across seven different file compressors. From these results, we concluded that not only are these distributions not normal, but the CR distribution for files with a shared extension likely come from multimodal distributions — that is, multiple underlying distributions.

These results sow doubt that the mean compression ratio is good predictor of file compressibility for a majority of file extensions, though it may be reliable enough for some extensions to warrant use in a larger adaptive compression system. Knowing the distributions of compressibility for file extensions motivates future work to improve compressibility prediction, such as by using a combination of the mean and the mode or other statistical properties, or finding readily accessible file metadata that can be used for identifying to which sub-distribution a file belongs.

A Normality Test Results

The following is the raw results for the top 100 extensions by ranked vote, plus DOCX. The Kolmogorov-Smirnov statistic (KS), Chi-Squared statistic (χ^2), and Anderson-Darling statistic (A) all have $p < .01$ unless otherwise marked. One asterisk (*) indicates $p < .05$, ** indicates $p < .1$, and *** indicates $p > .1$.

Table A.1: Normal Test Results

Ext.	Compressor	KS	χ^2	A	Mean	Median	Mode	SD
(none)	Deflate 1	0.11	5.1e+05	14478.21	0.612	0.577	1.0	0.277
	Deflate 6	0.11	8.7e+05	14908.49	0.597	0.563	1.0	0.287
	Deflate 9	0.11	8.8e+05	14850.18	0.597	0.563	1.0	0.288
	LZ4	0.11	4.4e+04	10776.76	0.696	0.721	1.0	0.249
	LZ4HC	0.10	9.7e+04	11256.43	0.669	0.695	1.0	0.265
	XZ 6	0.10	8.1e+05	13441.19	0.584	0.567	1.0	0.296
	XZ 9	0.10	8.1e+05	13439.87	0.584	0.567	1.0	0.296
png	Deflate 1	0.31	1.6e+05	40396.16	0.937	0.999	1.0	0.151
	Deflate 6	0.31	1.6e+05	40758.18	0.936	0.999	1.0	0.153
	Deflate 9	0.31	1.6e+05	40765.34	0.936	0.999	1.0	0.153
	LZ4	0.34	1.8e+05	45919.61	0.948	1.003	1.0	0.143
	LZ4HC	0.33	1.7e+05	42701.27	0.941	1.0	1.0	0.147
	XZ 6	0.27	1.5e+05	34029.47	0.937	1.0	1.0	0.159
	XZ 9	0.27	1.5e+05	34029.49	0.937	1.0	1.0	0.159

Ext.	Compressor	KS	χ^2	A	Mean	Median	Mode	SD
js	Deflate 1	0.01	4.7e+03	26.77	0.374	0.371	0.35	0.096
	Deflate 6	0.02	4.4e+03	44.74	0.345	0.34	0.3	0.103
	Deflate 9	0.02	4.3e+03	43.29	0.344	0.34	0.3	0.104
	LZ4	0.01	6.2e+01	5.53	0.515	0.513	0.45	0.129
	LZ4HC	0.02	2.6e+02	46.28	0.466	0.46	0.4	0.143
	XZ 6	0.04	1.9e+03	121.82	0.355	0.346	0.3	0.121
	XZ 9	0.04	1.9e+03	121.82	0.355	0.346	0.3	0.121
html	Deflate 1	0.06	1e+04	378.41	0.33	0.325	0.25	0.101
	Deflate 6	0.07	1.2e+04	706.27	0.303	0.294	0.2	0.105
	Deflate 9	0.07	1.2e+04	711.84	0.303	0.294	0.2	0.106
	LZ4	0.05	1.1e+03	396.10	0.444	0.438	0.35	0.136
	LZ4HC	0.07	2.5e+03	761.32	0.405	0.392	0.3	0.142
	XZ 6	0.08	9.8e+03	1020.90	0.308	0.293	0.2	0.118
	XZ 9	0.08	9.8e+03	1020.90	0.308	0.293	0.2	0.118
xml	Deflate 1	0.05	3.5e+03	500.55	0.325	0.307	0.25	0.152
	Deflate 6	0.06	3.8e+03	557.60	0.306	0.287	0.25	0.157
	Deflate 9	0.06	3.7e+03	544.07	0.306	0.287	0.25	0.157
	LZ4	0.06	2.9e+03	532.27	0.428	0.402	0.35	0.198
	LZ4HC	0.06	3.2e+03	589.74	0.399	0.372	0.35	0.204
	XZ 6	0.06	2.9e+03	558.96	0.316	0.295	0.25	0.17
	XZ 9	0.06	2.9e+03	558.90	0.316	0.295	0.25	0.17

Ext.	Compressor	KS	χ^2	A	Mean	Median	Mode	SD
jpg	Deflate 1	0.30	3.8e+04	6040.40	0.942	0.982	0.95	0.111
	Deflate 6	0.30	3.7e+04	5996.98	0.94	0.981	0.95	0.113
	Deflate 9	0.30	3.7e+04	5999.50	0.94	0.981	0.95	0.114
	LZ4	0.31	3.9e+04	6082.69	0.95	0.989	0.95	0.108
	LZ4HC	0.30	3.8e+04	5831.24	0.944	0.985	0.95	0.111
	XZ 6	0.29	3.5e+04	5698.47	0.937	0.982	0.95	0.119
	XZ 9	0.29	3.5e+04	5698.34	0.937	0.982	0.95	0.119
txt	Deflate 1	0.04	7.3e+02	61.47	0.362	0.366	0.4	0.145
	Deflate 6	0.04	1.5e+03	53.98	0.333	0.328	0.3	0.149
	Deflate 9	0.04	1.4e+03	53.11	0.332	0.327	0.3	0.15
	LZ4	0.04	4e+02	32.82	0.504	0.508	0.55	0.202
	LZ4HC	0.04	7.4e+02	68.61	0.452	0.441	0.4	0.206
	XZ 6	0.03	1.4e+03	97.04	0.333	0.321	0.3	0.168
	XZ 9	0.03	1.4e+03	97.10	0.333	0.321	0.3	0.168
json	Deflate 1	0.08	3.3e+02	380.76	0.35	0.373	0.4	0.121
	Deflate 6	0.07	1.1e+02	318.11	0.322	0.343	0.4	0.124
	Deflate 9	0.07	1e+02	325.30	0.321	0.342	0.4	0.125
	LZ4	0.09	9.8e+02	500.48	0.477	0.515	0.6	0.165
	LZ4HC	0.09	5.8e+02	491.17	0.429	0.467	0.55	0.168
	XZ 6	0.07	9.7e+01	364.15	0.332	0.353	0.4	0.141
	XZ 9	0.07	9.7e+01	364.17	0.332	0.353	0.4	0.141

Ext.	Compressor	KS	χ^2	A	Mean	Median	Mode	SD
svg	Deflate 1	0.04	4.4e+02	140.74	0.388	0.394	0.35	0.107
	Deflate 6	0.03	1.4e+02	82.53	0.364	0.37	0.4	0.111
	Deflate 9	0.03	1.6e+02	85.99	0.364	0.369	0.4	0.112
	LZ4	0.03	8.2e+02	95.16	0.556	0.566	0.55	0.163
	LZ4HC	0.02	6.8e+02	56.50	0.518	0.527	0.5	0.167
	XZ 6	0.02	1.9e+02	31.91	0.367	0.371	0.3	0.126
	XZ 9	0.02	1.9e+02	31.91	0.367	0.371	0.3	0.126
py	Deflate 1	0.03	6.5e+02	95.03	0.337	0.334	0.3	0.082
	Deflate 6	0.04	1.8e+03	235.74	0.304	0.297	0.25	0.087
	Deflate 9	0.04	1.8e+03	232.13	0.304	0.296	0.25	0.088
	LZ4	0.02	4.5e+02	69.16	0.468	0.464	0.45	0.116
	LZ4HC	0.04	2e+03	266.03	0.413	0.401	0.35	0.126
	XZ 6	0.06	3.4e+03	500.94	0.313	0.299	0.25	0.105
	XZ 9	0.06	3.4e+03	500.93	0.313	0.299	0.25	0.105
bin	Deflate 1	0.16	1.3e+05	860.55	0.576	0.489	1.0	0.33
	Deflate 6	0.16	1.2e+05	941.25	0.561	0.463	1.0	0.339
	Deflate 9	0.16	1.2e+05	942.09	0.56	0.462	1.0	0.34
	LZ4	0.17	3.6e+04	681.39	0.643	0.623	1.0	0.311
	LZ4HC	0.17	1.6e+05	761.43	0.608	0.557	1.0	0.324
	XZ 6	0.16	1.2e+05	1024.49	0.532	0.416	1.0	0.357
	XZ 9	0.16	1.2e+05	1023.72	0.532	0.416	1.0	0.357

Ext.	Compressor	KS	χ^2	A	Mean	Median	Mode	SD
pyc	Deflate 1	0.03	5.2e+03	174.21	0.486	0.49	0.45	0.094
	Deflate 6	0.02	2.5e+03	104.31	0.459	0.463	0.45	0.101
	Deflate 9	0.02	2.5e+03	103.76	0.458	0.462	0.45	0.102
	LZ4	0.03	5.3e+03	295.41	0.642	0.652	0.65	0.12
	LZ4HC	0.03	2.7e+03	196.83	0.591	0.599	0.6	0.135
	XZ 6	0.01	2.3e+02	40.81	0.444	0.443	0.4	0.119
	XZ 9	0.01	2.3e+02	40.81	0.444	0.443	0.4	0.119
h	Deflate 1	0.02	1.3e+03	122.45	0.356	0.362	0.35	0.104
	Deflate 6	0.01	6.4e+02	43.23	0.332	0.335	0.3	0.109
	Deflate 9	0.01	5.9e+02	46.09	0.331	0.334	0.3	0.11
	LZ4	0.02	9.6e+02	121.50	0.49	0.497	0.5	0.142
	LZ4HC	0.01	6.4e+02	43.21	0.45	0.454	0.45	0.151
	XZ 6	0.01	1.5e+02	26.67	0.343	0.343	0.35	0.127
	XZ 9	0.01	1.5e+02	26.67	0.343	0.343	0.35	0.127
dll	Deflate 1	0.21	1.6e+05	23910.96	0.604	0.499	1.0	0.275
	Deflate 6	0.21	2.4e+05	24748.28	0.585	0.471	1.0	0.287
	Deflate 9	0.21	2.5e+05	24729.36	0.584	0.469	1.0	0.288
	LZ4	0.19	1.4e+04	15878.54	0.678	0.614	1.0	0.237
	LZ4HC	0.20	7.5e+04	20155.42	0.633	0.549	0.95	0.259
	XZ 6	0.21	5.3e+05	27016.87	0.547	0.409	1.0	0.314
	XZ 9	0.21	5.3e+05	27015.30	0.546	0.409	1.0	0.314

Ext.	Compressor	KS	χ^2	A	Mean	Median	Mode	SD
ttf	Deflate 1	0.06	1.3e+03	98.33	0.582	0.581	0.55	0.102
	Deflate 6	0.06	1.5e+03	100.74	0.552	0.55	0.5	0.107
	Deflate 9	0.06	1.5e+03	100.22	0.551	0.55	0.5	0.108
	LZ4	0.06	1.7e+03	80.87	0.709	0.718	0.7	0.107
	LZ4HC	0.05	7.6e+02	55.85	0.637	0.641	0.6	0.114
	XZ 6	0.06	2.1e+03	76.77	0.471	0.467	0.45	0.123
	XZ 9	0.06	2.1e+03	76.75	0.471	0.467	0.45	0.123
dat	Deflate 1	0.09	1.5e+03	290.21	0.441	0.386	0.2	0.3
	Deflate 6	0.10	1.5e+03	364.02	0.42	0.343	0.05	0.306
	Deflate 9	0.10	1.5e+03	366.13	0.418	0.34	0.05	0.307
	LZ4	0.08	2.9e+04	214.56	0.536	0.506	1.0	0.316
	LZ4HC	0.09	1.1e+04	254.52	0.488	0.43	0.95	0.318
	XZ 6	0.13	1.3e+03	512.97	0.379	0.276	0.1	0.309
	XZ 9	0.13	1.3e+03	512.53	0.379	0.276	0.1	0.309
gz	Deflate 1	0.39	1.2e+05	14377.65	1.0	1.004	1.0	0.03
	Deflate 6	0.39	1.2e+05	14376.99	1.0	1.004	1.0	0.03
	Deflate 9	0.39	1.2e+05	14381.96	1.0	1.004	1.0	0.03
	LZ4	0.46	1.3e+05	19089.81	1.002	1.004	1.0	0.027
	LZ4HC	0.43	1.3e+05	17329.81	1.0	1.004	1.0	0.028
	XZ 6	0.30	1.1e+05	7328.88	1.011	1.014	1.0	0.034
	XZ 9	0.30	1.1e+05	7329.84	1.011	1.014	1.0	0.034

Ext.	Compressor	KS	χ^2	A	Mean	Median	Mode	SD
mo	Deflate 1	0.06	6.7e+02	111.64	0.444	0.433	0.4	0.075
	Deflate 6	0.05	4.3e+02	72.51	0.417	0.406	0.35	0.086
	Deflate 9	0.05	3.8e+02	67.24	0.417	0.406	0.35	0.087
	LZ4	0.04	3e+01	36.00	0.602	0.592	0.55	0.103
	LZ4HC	0.04	3.6e+02	41.39	0.55	0.54	0.45	0.124
	XZ 6	0.08	1.4e+03	259.08	0.359	0.34	0.3	0.101
	XZ 9	0.08	1.4e+03	259.08	0.359	0.34	0.3	0.101
pm	Deflate 1	0.06	1.5e+03	82.10	0.397	0.399	0.35	0.088
	Deflate 6	0.03	2.7e+02	19.01	0.365	0.365	0.35	0.095
	Deflate 9	0.03	3e+02	19.86	0.365	0.364	0.35	0.096
	LZ4	0.05	1.2e+03	61.25	0.544	0.547	0.5	0.122
	LZ4HC	0.02	1e+02	9.69	0.491	0.489	0.45	0.133
	XZ 6	0.02	2.9e+01	17.55	0.368	0.362	0.3	0.115
	XZ 9	0.02	2.9e+01	17.55	0.368	0.362	0.3	0.115
exe	Deflate 1	0.22	3.1e+04	2426.41	0.621	0.513	1.0	0.268
	Deflate 6	0.22	4.7e+04	2521.37	0.603	0.487	1.0	0.279
	Deflate 9	0.22	4.7e+04	2520.53	0.602	0.486	1.0	0.279
	LZ4	0.19	2.1e+03	1409.94	0.699	0.64	1.0	0.226
	LZ4HC	0.20	1.1e+04	1849.59	0.653	0.574	0.55	0.25
	XZ 6	0.24	1.1e+05	2765.05	0.564	0.427	1.0	0.306
	XZ 9	0.24	1.1e+05	2762.86	0.564	0.427	1.0	0.306

Ext.	Compressor	KS	χ^2	A	Mean	Median	Mode	SD
cat	Deflate 1	0.27	2.1e+04	15258.62	0.568	0.615	0.6	0.094
	Deflate 6	0.27	2.1e+04	15269.43	0.559	0.607	0.6	0.095
	Deflate 9	0.27	2.1e+04	15328.95	0.558	0.607	0.6	0.096
	LZ4	0.27	2.1e+04	15062.12	0.591	0.636	0.6	0.09
	LZ4HC	0.28	2.2e+04	15394.34	0.579	0.625	0.6	0.093
	XZ 6	0.28	2.2e+04	15394.17	0.542	0.592	0.55	0.097
	XZ 9	0.28	2.2e+04	15394.17	0.542	0.592	0.55	0.097
manifest	Deflate 1	0.22	1.1e+04	4942.36	0.449	0.384	1.0	0.323
	Deflate 6	0.23	1.1e+04	5216.29	0.437	0.366	1.0	0.328
	Deflate 9	0.22	1.1e+04	5204.21	0.437	0.366	1.0	0.328
	LZ4	0.15	1.8e+04	2280.05	0.515	0.496	1.0	0.313
	LZ4HC	0.15	1.7e+04	2495.42	0.5	0.475	1.0	0.317
	XZ 6	0.18	1.1e+04	4341.84	0.453	0.392	1.0	0.333
	XZ 9	0.18	1.1e+04	4341.84	0.453	0.392	1.0	0.333
pdf	Deflate 1	0.18	2.9e+03	765.02	0.835	0.903	0.95	0.178
	Deflate 6	0.17	2.7e+03	747.39	0.826	0.897	0.95	0.186
	Deflate 9	0.17	2.7e+03	745.06	0.825	0.896	0.95	0.186
	LZ4	0.18	3.2e+03	707.20	0.856	0.915	0.95	0.16
	LZ4HC	0.17	2.9e+03	684.82	0.839	0.901	0.95	0.171
	XZ 6	0.17	2.4e+03	656.09	0.813	0.884	0.95	0.198
	XZ 9	0.17	2.4e+03	655.63	0.813	0.884	0.95	0.198

Ext.	Compressor	KS	χ^2	A	Mean	Median	Mode	SD
so	Deflate 1	0.07	3.3e+02	87.93	0.332	0.339	0.3	0.12
	Deflate 6	0.07	6.5e+02	87.74	0.309	0.315	0.3	0.117
	Deflate 9	0.07	6.8e+02	86.98	0.308	0.314	0.3	0.116
	LZ4	0.08	2e+02	90.58	0.43	0.44	0.4	0.161
	LZ4HC	0.08	1.9e+02	96.04	0.378	0.388	0.35	0.145
	XZ 6	0.07	1.7e+02	84.68	0.255	0.263	0.25	0.099
	XZ 9	0.07	1.7e+02	84.61	0.255	0.263	0.25	0.099
mum	Deflate 1	0.14	1.7e+04	6270.84	0.33	0.361	0.4	0.092
	Deflate 6	0.14	1.7e+04	5974.25	0.317	0.343	0.35	0.091
	Deflate 9	0.14	1.7e+04	5997.30	0.317	0.343	0.35	0.092
	LZ4	0.15	1.7e+04	6219.92	0.429	0.465	0.5	0.117
	LZ4HC	0.15	1.7e+04	5942.03	0.415	0.45	0.5	0.117
	XZ 6	0.14	1.5e+04	5303.47	0.349	0.381	0.4	0.108
	XZ 9	0.14	1.5e+04	5303.47	0.349	0.381	0.4	0.108
wav	Deflate 1	0.07	5.1e+02	52.50	0.739	0.767	0.9	0.18
	Deflate 6	0.07	4.6e+02	55.73	0.73	0.759	0.9	0.188
	Deflate 9	0.07	4.6e+02	55.94	0.73	0.759	0.9	0.188
	LZ4	0.23	2.3e+03	319.04	0.889	0.947	0.95	0.155
	LZ4HC	0.18	1.4e+03	200.61	0.841	0.898	0.95	0.179
	XZ 6	0.06	1.8e+02	28.31	0.629	0.628	0.45	0.209
	XZ 9	0.06	1.8e+02	28.27	0.629	0.628	0.45	0.209

Ext.	Compressor	KS	χ^2	A	Mean	Median	Mode	SD
css	Deflate 1	0.04	1.2e+03	28.10	0.307	0.3	0.25	0.116
	Deflate 6	0.04	1.3e+03	27.56	0.278	0.269	0.25	0.122
	Deflate 9	0.04	1.2e+03	26.84	0.277	0.268	0.25	0.123
	LZ4	0.03	1.9e+02	18.74	0.421	0.414	0.4	0.157
	LZ4HC	0.04	3.3e+02	19.64	0.373	0.363	0.35	0.166
	XZ 6	0.04	6.8e+02	19.29	0.287	0.279	0.25	0.138
	XZ 9	0.04	6.8e+02	19.29	0.287	0.279	0.25	0.138
md	Deflate 1	0.04	1.4e+02	10.13	0.43	0.433	0.4	0.088
	Deflate 6	0.03*	4.7e+01	3.70	0.406	0.41	0.4	0.094
	Deflate 9	0.03*	5.1e+01	3.94	0.406	0.409	0.4	0.094
	LZ4	0.04	9e+01	6.76	0.597	0.599	0.55	0.129
	LZ4HC	0.02**	3.3e+01	2.81	0.56	0.565	0.55	0.137
	XZ 6	0.02**	1.3e+01*	2.13	0.428	0.429	0.45	0.115
	XZ 9	0.02**	1.3e+01*	2.13	0.428	0.429	0.45	0.115
mui	Deflate 1	0.34	1.1e+04	6830.24	0.426	0.341	0.35	0.263
	Deflate 6	0.34	1.1e+04	6578.22	0.399	0.312	0.3	0.276
	Deflate 9	0.34	1.1e+04	6580.18	0.397	0.311	0.3	0.277
	LZ4	0.27	8.3e+03	4426.25	0.514	0.457	0.45	0.225
	LZ4HC	0.27	8.4e+03	4241.59	0.457	0.398	0.4	0.252
	XZ 6	0.33	1.1e+04	6522.88	0.375	0.281	0.3	0.291
	XZ 9	0.33	1.1e+04	6522.88	0.375	0.281	0.3	0.291

Ext.	Compressor	KS	χ^2	A	Mean	Median	Mode	SD
java	Deflate 1	0.04	2.9e+03	197.15	0.372	0.366	0.3	0.096
	Deflate 6	0.04	5.2e+03	249.81	0.346	0.34	0.25	0.104
	Deflate 9	0.04	5.2e+03	246.13	0.345	0.34	0.25	0.105
	LZ4	0.03	3.2e+03	182.96	0.514	0.508	0.45	0.136
	LZ4HC	0.04	5.9e+03	238.91	0.474	0.468	0.4	0.149
	XZ 6	0.05	6.1e+03	293.04	0.365	0.357	0.25	0.125
	XZ 9	0.05	6.1e+03	293.04	0.365	0.357	0.25	0.125
hpp	Deflate 1	0.03	3.2e+02	30.75	0.341	0.348	0.35	0.111
	Deflate 6	0.03	3.2e+02	27.34	0.319	0.325	0.3	0.116
	Deflate 9	0.03	3.2e+02	27.90	0.318	0.325	0.3	0.116
	LZ4	0.03	2.8e+02	26.04	0.467	0.476	0.4	0.154
	LZ4HC	0.03	2.8e+02	21.21	0.432	0.438	0.4	0.161
	XZ 6	0.04	4.2e+02	28.75	0.336	0.343	0.35	0.131
	XZ 9	0.04	4.2e+02	28.75	0.336	0.343	0.35	0.131
jar	Deflate 1	0.27	1.5e+03	810.48	0.784	0.885	0.9	0.227
	Deflate 6	0.27	1.5e+03	817.55	0.772	0.879	0.9	0.238
	Deflate 9	0.27	1.5e+03	817.19	0.772	0.879	0.9	0.238
	LZ4	0.25	1.8e+03	706.47	0.804	0.892	0.9	0.2
	LZ4HC	0.26	1.6e+03	748.17	0.781	0.878	0.9	0.221
	XZ 6	0.27	1.5e+03	840.14	0.755	0.871	0.85	0.26
	XZ 9	0.27	1.5e+03	839.54	0.755	0.871	0.85	0.26

Ext.	Compressor	KS	χ^2	A	Mean	Median	Mode	SD
gif	Deflate 1	0.26	3.3e+03	746.17	0.903	0.978	0.95	0.165
	Deflate 6	0.26	3.3e+03	739.44	0.901	0.977	0.95	0.167
	Deflate 9	0.26	3.3e+03	739.61	0.901	0.977	0.95	0.167
	LZ4	0.28	3.6e+03	856.34	0.913	0.984	1.0	0.163
	LZ4HC	0.28	3.4e+03	819.54	0.909	0.981	1.0	0.166
	XZ 6	0.20	2.6e+03	531.53	0.88	0.954	0.95	0.173
	XZ 9	0.20	2.6e+03	531.53	0.88	0.954	0.95	0.173
pl	Deflate 1	0.18	1.6e+03	315.68	0.396	0.453	0.45	0.129
	Deflate 6	0.17	2.1e+03	291.80	0.367	0.425	0.45	0.131
	Deflate 9	0.17	2e+03	292.71	0.367	0.425	0.45	0.132
	LZ4	0.19	1.8e+04	439.12	0.611	0.679	0.8	0.228
	LZ4HC	0.18	2.7e+05	407.11	0.57	0.636	0.75	0.237
	XZ 6	0.08	7.1e+02	70.33	0.312	0.293	0.2	0.128
	XZ 9	0.08	7.1e+02	70.33	0.312	0.293	0.2	0.128
vim	Deflate 1	0.05	8.9e+02	61.32	0.368	0.374	0.35	0.092
	Deflate 6	0.04	3.5e+02	28.94	0.344	0.347	0.35	0.096
	Deflate 9	0.04	3.5e+02	29.62	0.343	0.347	0.35	0.096
	LZ4	0.05	9.6e+02	63.50	0.51	0.518	0.5	0.127
	LZ4HC	0.04	2.9e+02	25.88	0.471	0.477	0.45	0.134
	XZ 6	0.03	6.3*	17.59	0.356	0.352	0.3	0.113
	XZ 9	0.03	6.3*	17.59	0.356	0.352	0.3	0.113

Ext.	Compressor	KS	χ^2	A	Mean	Median	Mode	SD
c	Deflate 1	0.03	4.7e+01	9.59	0.352	0.348	0.3	0.09
	Deflate 6	0.04	1.3e+02	17.64	0.321	0.314	0.25	0.098
	Deflate 9	0.04	1.2e+02	16.82	0.32	0.314	0.25	0.098
	LZ4	0.02*	4e+01	6.81	0.485	0.479	0.45	0.126
	LZ4HC	0.04	1.6e+02	17.51	0.431	0.424	0.35	0.14
	XZ 6	0.05	2.1e+02	33.15	0.325	0.314	0.25	0.116
	XZ 9	0.05	2.1e+02	33.15	0.325	0.314	0.25	0.116
log	Deflate 1	0.10	1.5e+03	130.27	0.197	0.177	0.15	0.126
	Deflate 6	0.11	1.8e+03	171.95	0.177	0.155	0.05	0.125
	Deflate 9	0.11	1.8e+03	175.65	0.175	0.154	0.05	0.125
	LZ4	0.08	6.8e+02	90.60	0.256	0.234	0.1	0.157
	LZ4HC	0.10	9.8e+02	146.04	0.224	0.201	0.05	0.154
	XZ 6	0.10	1.6e+03	212.30	0.165	0.138	0.05	0.13
	XZ 9	0.10	1.6e+03	212.32	0.165	0.138	0.05	0.13
lua	Deflate 1	0.09	1.9e+03	76.60	0.378	0.378	0.45	0.127
	Deflate 6	0.09	2e+03	73.28	0.35	0.348	0.45	0.133
	Deflate 9	0.09	2e+03	72.51	0.349	0.348	0.45	0.133
	LZ4	0.06	8.2e+01	44.33	0.519	0.532	0.6	0.15
	LZ4HC	0.06	6.7e+01	45.87	0.468	0.48	0.6	0.159
	XZ 6	0.07	2e+03	56.54	0.346	0.349	0.4	0.137
	XZ 9	0.07	2e+03	56.54	0.346	0.349	0.4	0.137

Ext.	Compressor	KS	χ^2	A	Mean	Median	Mode	SD
tfm	Deflate 1	0.09	7.4e+03	527.98	0.473	0.478	0.65	0.18
	Deflate 6	0.09	9.5e+03	510.67	0.462	0.465	0.65	0.183
	Deflate 9	0.09	9.7e+03	507.76	0.461	0.463	0.65	0.183
	LZ4	0.10	4.3e+03	618.21	0.638	0.655	0.85	0.234
	LZ4HC	0.10	7.3e+03	618.87	0.614	0.614	0.85	0.236
	XZ 6	0.09	4e+04	638.35	0.398	0.414	0.55	0.174
	XZ 9	0.09	4e+04	638.35	0.398	0.414	0.55	0.174
pak	Deflate 1	0.34	3e+03	940.27	0.429	0.39	0.35	0.179
	Deflate 6	0.31	2.8e+03	806.45	0.378	0.336	0.3	0.199
	Deflate 9	0.31	2.7e+03	779.06	0.376	0.336	0.3	0.2
	LZ4	0.24	1.7e+03	419.68	0.533	0.517	0.5	0.164
	LZ4HC	0.25	1.8e+03	475.27	0.444	0.416	0.4	0.194
	XZ 6	0.33	3e+03	943.74	0.31	0.258	0.25	0.216
	XZ 9	0.33	3e+03	945.30	0.31	0.258	0.25	0.215
sys	Deflate 1	0.26	6.1e+03	437.35	0.655	0.541	1.0	0.255
	Deflate 6	0.27	1.7e+04	456.42	0.635	0.511	1.0	0.268
	Deflate 9	0.27	1.7e+04	456.67	0.635	0.51	1.0	0.268
	LZ4	0.21	1.5e+02	253.38	0.734	0.677	1.0	0.21
	LZ4HC	0.23	2.1e+03	370.41	0.684	0.59	0.55	0.237
	XZ 6	0.27	5.1e+04	485.88	0.597	0.45	1.0	0.297
	XZ 9	0.27	5.1e+04	485.87	0.597	0.45	1.0	0.297

Ext.	Compressor	KS	χ^2	A	Mean	Median	Mode	SD
pri	Deflate 1	0.43	2e+03	2212.39	0.828	1.001	1.0	0.277
	Deflate 6	0.43	2e+03	2215.32	0.82	1.0	1.0	0.288
	Deflate 9	0.43	2e+03	2214.36	0.82	1.0	1.0	0.289
	LZ4	0.42	2.4e+03	2094.70	0.873	1.004	1.0	0.221
	LZ4HC	0.42	2.2e+03	2086.48	0.858	1.002	1.0	0.24
	XZ 6	0.42	2.2e+03	2134.34	0.811	1.004	1.0	0.313
	XZ 9	0.42	2.2e+03	2134.34	0.811	1.004	1.0	0.313
cpp	Deflate 1	0.03*	1.6e+02	2.96	0.34	0.334	0.3	0.093
	Deflate 6	0.04	1.6e+02	5.61	0.314	0.305	0.25	0.099
	Deflate 9	0.04	1.6e+02	5.51	0.313	0.304	0.25	0.1
	LZ4	0.03*	1.4e+01	2.62	0.466	0.457	0.4	0.127
	LZ4HC	0.04	4.3e+01	6.39	0.422	0.41	0.35	0.138
	XZ 6	0.05	1e+02	9.71	0.328	0.313	0.25	0.117
	XZ 9	0.05	1e+02	9.71	0.328	0.313	0.25	0.117
inf	Deflate 1	0.07	1.5e+03	95.70	0.274	0.273	0.3	0.151
	Deflate 6	0.08	1.8e+03	116.85	0.249	0.243	0.05	0.153
	Deflate 9	0.08	1.7e+03	116.10	0.248	0.241	0.05	0.154
	LZ4	0.09	7.5e+02	103.29	0.383	0.389	0.45	0.199
	LZ4HC	0.09	5.6e+02	126.81	0.339	0.337	0.05	0.202
	XZ 6	0.09	1.8e+03	173.71	0.239	0.219	0.05	0.168
	XZ 9	0.09	1.8e+03	173.71	0.239	0.219	0.05	0.168

Ext.	Compressor	KS	χ^2	A	Mean	Median	Mode	SD
final	Deflate 1	0.21	9.3e+02	591.13	0.883	0.875	0.85	0.062
	Deflate 6	0.21	9.8e+02	568.74	0.882	0.874	0.85	0.064
	Deflate 9	0.21	9.8e+02	568.65	0.882	0.874	0.85	0.064
	LZ4	0.34	1.6e+04	2119.75	0.995	1.003	1.0	0.026
	LZ4HC	0.27	1.1e+04	1270.98	0.983	0.998	0.95	0.035
	XZ 6	0.13	9.2e+02	270.01	0.872	0.869	0.85	0.077
	XZ 9	0.13	9.2e+02	270.01	0.872	0.869	0.85	0.077
etl	Deflate 1	0.14	8.7e+03	298.26	0.104	0.101	0.1	0.066
	Deflate 6	0.14	9.1e+03	343.40	0.094	0.088	0.05	0.064
	Deflate 9	0.14	9.2e+03	353.30	0.093	0.085	0.05	0.064
	LZ4	0.13	7.8e+03	242.02	0.124	0.118	0.1	0.079
	LZ4HC	0.15	8.3e+03	288.46	0.111	0.099	0.05	0.075
	XZ 6	0.16	9.4e+03	375.65	0.084	0.08	0.05	0.062
	XZ 9	0.16	9.4e+03	375.62	0.084	0.08	0.05	0.062
tga	Deflate 1	0.12	2.1e+03	301.65	0.222	0.178	0.05	0.183
	Deflate 6	0.14	2.5e+03	381.02	0.199	0.157	0.0	0.182
	Deflate 9	0.14	2.5e+03	384.12	0.197	0.155	0.0	0.182
	LZ4	0.10	1.2e+03	206.39	0.307	0.256	0.15	0.235
	LZ4HC	0.14	2e+03	340.32	0.249	0.191	0.0	0.222
	XZ 6	0.13	2.6e+03	316.51	0.182	0.152	0.0	0.16
	XZ 9	0.13	2.6e+03	316.51	0.182	0.152	0.0	0.16

Ext.	Compressor	KS	χ^2	A	Mean	Median	Mode	SD
rtf	Deflate 1	0.26	1.8e+04	334.73	0.568	0.412	1.0	0.341
	Deflate 6	0.26	1.8e+04	332.70	0.542	0.384	1.0	0.362
	Deflate 9	0.26	1.8e+04	329.22	0.541	0.384	1.0	0.363
	LZ4	0.25	2.4e+04	199.24	0.649	0.579	1.0	0.298
	LZ4HC	0.25	1.9e+04	218.45	0.603	0.533	1.0	0.336
	XZ 6	0.25	1.8e+04	284.21	0.544	0.401	1.0	0.373
	XZ 9	0.25	1.8e+04	284.21	0.544	0.401	1.0	0.373
1	Deflate 1	0.14	1e+03	126.90	0.441	0.437	0.4	0.124
	Deflate 6	0.12	1.1e+03	104.88	0.412	0.407	0.4	0.13
	Deflate 9	0.12	1.1e+03	104.17	0.411	0.407	0.4	0.131
	LZ4	0.08	2.1e+02	39.40	0.588	0.596	0.6	0.139
	LZ4HC	0.05	9.9e+01	15.96	0.534	0.537	0.55	0.151
	XZ 6	0.09	7.4e+02	46.00	0.399	0.39	0.4	0.147
	XZ 9	0.09	7.4e+02	45.93	0.399	0.39	0.4	0.147
rb	Deflate 1	0.04	1e+02	19.34	0.373	0.372	0.35	0.078
	Deflate 6	0.04	1.6e+02	15.68	0.343	0.34	0.3	0.084
	Deflate 9	0.04	1.5e+02	15.03	0.343	0.339	0.3	0.085
	LZ4	0.03	4.3e+01	9.93	0.523	0.522	0.5	0.108
	LZ4HC	0.03	1.2e+02	10.60	0.469	0.464	0.4	0.12
	XZ 6	0.04	1.2e+02	15.05	0.361	0.353	0.3	0.101
	XZ 9	0.04	1.2e+02	15.05	0.361	0.353	0.3	0.101

Ext.	Compressor	KS	χ^2	A	Mean	Median	Mode	SD
strings	Deflate 1	0.19	1.4e+02	39.66	0.335	0.296	0.25	0.123
	Deflate 6	0.18	1.5e+02	37.04	0.305	0.269	0.25	0.128
	Deflate 9	0.18	1.5e+02	37.15	0.304	0.267	0.25	0.129
	LZ4	0.19	8.8e+01	34.16	0.444	0.403	0.35	0.149
	LZ4HC	0.17	8.1e+01	26.00	0.39	0.358	0.3	0.154
	XZ 6	0.16	1.4e+02	35.34	0.288	0.263	0.25	0.131
	XZ 9	0.16	1.4e+02	35.34	0.288	0.263	0.25	0.131
o	Deflate 1	0.09	4.5e+02	19.93	0.364	0.354	0.3	0.095
	Deflate 6	0.10	5.6e+02	22.11	0.331	0.32	0.3	0.097
	Deflate 9	0.10	5.6e+02	21.81	0.328	0.318	0.3	0.097
	LZ4	0.07	5.8e+01	9.75	0.495	0.491	0.45	0.115
	LZ4HC	0.08	1.2e+02	12.72	0.422	0.413	0.4	0.114
	XZ 6	0.08	6.3e+02	15.17	0.279	0.271	0.25	0.102
	XZ 9	0.08	6.3e+02	15.17	0.279	0.271	0.25	0.102
a	Deflate 1	0.05	7.3e+02	17.44	0.281	0.291	0.3	0.13
	Deflate 6	0.06	1e+03	24.10	0.247	0.25	0.25	0.128
	Deflate 9	0.06	1e+03	24.20	0.243	0.245	0.25	0.129
	LZ4	0.04	1e+02	11.99	0.364	0.378	0.4	0.159
	LZ4HC	0.05	4.6e+02	15.58	0.296	0.298	0.3	0.149
	XZ 6	0.08	1.6e+03	46.18	0.192	0.182	0.1	0.118
	XZ 9	0.08	1.6e+03	45.85	0.192	0.182	0.1	0.118

Ext.	Compressor	KS	χ^2	A	Mean	Median	Mode	SD
plist	Deflate 1	0.06*	3.4e+01	5.77	0.304	0.289	0.35	0.16
	Deflate 6	0.06*	3.8e+01	6.41	0.286	0.268	0.35	0.162
	Deflate 9	0.06*	3.7e+01	6.38	0.284	0.268	0.35	0.163
	LZ4	0.06*	1.8e+01	4.25	0.392	0.382	0.45	0.197
	LZ4HC	0.07*	1.8e+01	5.20	0.362	0.353	0.45	0.198
	XZ 6	0.08	1.5e+01	8.01	0.292	0.281	0.35	0.165
	XZ 9	0.08	1.5e+01	8.01	0.292	0.281	0.35	0.165
tex	Deflate 1	0.03	4.6e+02	12.12	0.377	0.376	0.4	0.117
	Deflate 6	0.03	5e+02	10.28	0.347	0.343	0.35	0.123
	Deflate 9	0.03	5e+02	10.22	0.346	0.343	0.35	0.123
	LZ4	0.02**	1.8e+01	2.22	0.524	0.522	0.45	0.16
	LZ4HC	0.03*	5e+01	6.55	0.473	0.465	0.4	0.168
	XZ 6	0.03	9.7e+01	9.05	0.346	0.337	0.3	0.133
	XZ 9	0.03	9.7e+01	9.05	0.346	0.337	0.3	0.133
0	Deflate 1	0.11	1.6e+02	70.82	0.369	0.394	0.4	0.17
	Deflate 6	0.12	2.7e+02	66.76	0.343	0.361	0.35	0.168
	Deflate 9	0.11	2.8e+02	66.37	0.341	0.359	0.35	0.168
	LZ4	0.11	1.4e+02	65.90	0.478	0.515	0.5	0.203
	LZ4HC	0.11	6.1e+01	67.66	0.415	0.439	0.45	0.193
	XZ 6	0.14	7.9e+02	71.97	0.296	0.298	0.25	0.162
	XZ 9	0.14	7.9e+02	71.91	0.296	0.297	0.25	0.162

Ext.	Compressor	KS	χ^2	A	Mean	Median	Mode	SD
cab	Deflate 1	0.26	3.5e+02	469.06	0.875	0.962	0.95	0.129
	Deflate 6	0.26	3.7e+02	468.67	0.872	0.961	0.95	0.132
	Deflate 9	0.26	3.7e+02	468.59	0.872	0.961	0.95	0.132
	LZ4	0.26	2.6e+02	467.01	0.89	0.974	1.0	0.122
	LZ4HC	0.26	3.8e+02	472.34	0.877	0.962	0.95	0.129
	XZ 6	0.26	2.8e+02	457.74	0.867	0.961	0.95	0.138
	XZ 9	0.26	2.7e+02	456.56	0.867	0.961	0.95	0.138
ko	Deflate 1	0.06	1.5e+02	11.55	0.34	0.343	0.3	0.044
	Deflate 6	0.06	1.9e+02	9.57	0.308	0.309	0.3	0.044
	Deflate 9	0.06	1.9e+02	9.77	0.304	0.305	0.3	0.044
	LZ4	0.10	2.1e+02	24.00	0.443	0.446	0.4	0.051
	LZ4HC	0.07	2.3e+02	15.26	0.37	0.372	0.35	0.049
	XZ 6	0.04**	8.9e+01	4.27	0.261	0.264	0.25	0.044
	XZ 9	0.04**	8.9e+01	4.27	0.261	0.264	0.25	0.044
map	Deflate 1	0.15	3.4e+02	49.01	0.369	0.336	0.25	0.18
	Deflate 6	0.16	3.6e+02	57.33	0.335	0.29	0.2	0.189
	Deflate 9	0.15	3.6e+02	56.71	0.334	0.289	0.2	0.189
	LZ4	0.04*	5.1e+01	5.50	0.49	0.485	0.5	0.181
	LZ4HC	0.07	1.2e+02	15.06	0.422	0.4	0.25	0.189
	XZ 6	0.10	3.5e+02	29.09	0.313	0.278	0.15	0.174
	XZ 9	0.10	3.5e+02	28.93	0.313	0.278	0.15	0.174

Ext.	Compressor	KS	χ^2	A	Mean	Median	Mode	SD
ts	Deflate 1	0.03	1.4e+02	34.65	0.346	0.353	0.35	0.1
	Deflate 6	0.03	9.4e+01	21.74	0.322	0.328	0.3	0.105
	Deflate 9	0.03	9.8e+01	22.46	0.322	0.328	0.3	0.105
	LZ4	0.04	2.9e+02	40.50	0.474	0.485	0.5	0.14
	LZ4HC	0.03	2.8e+02	28.07	0.438	0.448	0.45	0.146
	XZ 6	0.03	3.1e+02	20.17	0.347	0.352	0.35	0.124
	XZ 9	0.03	3.1e+02	20.17	0.347	0.352	0.35	0.124
file	Deflate 1	0.18	1.2e+03	257.63	0.648	0.578	1.0	0.233
	Deflate 6	0.17	1.2e+03	248.67	0.638	0.567	1.0	0.241
	Deflate 9	0.17	1.1e+03	247.93	0.637	0.566	1.0	0.242
	LZ4	0.16	8e+02	246.48	0.701	0.641	1.0	0.201
	LZ4HC	0.16	7.6e+02	237.68	0.682	0.619	1.0	0.214
	XZ 6	0.17	1.3e+03	274.31	0.619	0.535	1.0	0.251
	XZ 9	0.17	1.3e+03	274.31	0.619	0.535	1.0	0.251
res	Deflate 1	0.10	1.1e+03	66.88	0.301	0.278	0.25	0.119
	Deflate 6	0.11	1.2e+03	72.51	0.275	0.251	0.2	0.12
	Deflate 9	0.10	1.2e+03	70.97	0.274	0.25	0.2	0.12
	LZ4	0.11	4e+02	56.33	0.407	0.382	0.35	0.15
	LZ4HC	0.09	6.5e+02	60.33	0.355	0.328	0.3	0.147
	XZ 6	0.07	1.4e+03	38.78	0.272	0.265	0.25	0.108
	XZ 9	0.07	1.4e+03	38.78	0.272	0.265	0.25	0.108

Ext.	Compressor	KS	χ^2	A	Mean	Median	Mode	SD
ogg	Deflate 1	0.21	7.2e+02	127.10	0.956	0.978	0.95	0.054
	Deflate 6	0.20	7.1e+02	126.53	0.953	0.977	0.95	0.056
	Deflate 9	0.20	7.1e+02	126.58	0.953	0.977	0.95	0.056
	LZ4	0.24	9.4e+02	156.42	0.974	0.993	1.0	0.043
	LZ4HC	0.23	8.9e+02	147.18	0.967	0.988	0.95	0.045
	XZ 6	0.20	6.1e+02	118.11	0.949	0.974	0.95	0.061
	XZ 9	0.20	6.1e+02	118.10	0.949	0.974	0.95	0.061
winmd	Deflate 1	0.35	1.5e+04	584.09	0.745	0.987	0.95	0.293
	Deflate 6	0.34	1.5e+04	584.13	0.734	0.985	0.95	0.304
	Deflate 9	0.34	1.5e+04	584.37	0.734	0.985	0.95	0.304
	LZ4	0.33	1.6e+04	543.17	0.807	0.99	0.95	0.224
	LZ4HC	0.33	1.6e+04	545.71	0.785	0.988	0.95	0.247
	XZ 6	0.34	1.5e+04	558.61	0.709	0.985	0.95	0.34
	XZ 9	0.34	1.5e+04	558.61	0.709	0.985	0.95	0.34
go	Deflate 1	0.07	4.8e+01	9.59	0.33	0.35	0.35	0.13
	Deflate 6	0.04*	3.5e+01	4.53	0.304	0.313	0.3	0.129
	Deflate 9	0.04*	3.7e+01	4.59	0.303	0.313	0.3	0.13
	LZ4	0.07	5.1e+01	9.78	0.458	0.487	0.55	0.183
	LZ4HC	0.04*	5.1e+01	4.24	0.414	0.426	0.4	0.182
	XZ 6	0.05*	7.4e+01	4.22	0.308	0.315	0.3	0.152
	XZ 9	0.05*	7.4e+01	4.22	0.308	0.315	0.3	0.152

Ext.	Compressor	KS	χ^2	A	Mean	Median	Mode	SD
ini	Deflate 1	0.12	1.2e+03	94.82	0.364	0.386	0.4	0.156
	Deflate 6	0.12	1.2e+03	91.04	0.341	0.369	0.35	0.162
	Deflate 9	0.12	1.2e+03	90.66	0.34	0.369	0.35	0.163
	LZ4	0.10	3.7e+01	58.40	0.492	0.534	0.55	0.181
	LZ4HC	0.11	3.3e+01	55.69	0.452	0.505	0.5	0.194
	XZ 6	0.09	9.8e+02	66.94	0.34	0.365	0.35	0.172
	XZ 9	0.09	9.8e+02	66.94	0.34	0.365	0.35	0.172
page	Deflate 1	0.03	2.7e+02	23.57	0.431	0.434	0.4	0.068
	Deflate 6	0.02	1.4e+02	11.51	0.412	0.414	0.4	0.07
	Deflate 9	0.02	1.5e+02	11.68	0.412	0.414	0.4	0.071
	LZ4	0.03	4e+02	31.67	0.596	0.603	0.6	0.095
	LZ4HC	0.02	1.8e+02	13.41	0.564	0.569	0.55	0.099
	XZ 6	0.01*	4.1e+01	3.61	0.442	0.442	0.4	0.083
	XZ 9	0.01*	4.1e+01	3.61	0.442	0.442	0.4	0.083
vf	Deflate 1	0.21	2.7e+03	744.72	0.697	0.763	0.75	0.152
	Deflate 6	0.21	2.5e+03	749.78	0.697	0.763	0.75	0.152
	Deflate 9	0.21	2.5e+03	750.37	0.697	0.763	0.75	0.152
	LZ4	0.30	3.5e+03	1371.61	0.893	0.967	0.95	0.164
	LZ4HC	0.33	3.5e+03	1517.85	0.877	0.955	0.95	0.165
	XZ 6	0.20	2.9e+03	684.04	0.644	0.719	0.7	0.183
	XZ 9	0.20	2.9e+03	684.04	0.644	0.719	0.7	0.183

Ext.	Compressor	KS	χ^2	A	Mean	Median	Mode	SD
pod	Deflate 1	0.16	2.9e+02	149.41	0.392	0.416	0.4	0.1
	Deflate 6	0.13	1.7e+02	94.06	0.357	0.373	0.35	0.103
	Deflate 9	0.13	1.7e+02	93.15	0.356	0.373	0.35	0.103
	LZ4	0.14	2.3e+02	112.75	0.538	0.569	0.55	0.137
	LZ4HC	0.10	1.3e+02	43.59	0.478	0.491	0.45	0.145
	XZ 6	0.10	4.4e+01	43.29	0.357	0.364	0.35	0.115
	XZ 9	0.10	4.4e+01	43.29	0.357	0.364	0.35	0.115
sty	Deflate 1	0.07	3.1e+02	18.13	0.36	0.354	0.3	0.094
	Deflate 6	0.07	4.4e+02	21.42	0.331	0.324	0.25	0.101
	Deflate 9	0.06	4.3e+02	21.12	0.331	0.324	0.25	0.102
	LZ4	0.07	3.2e+02	19.03	0.504	0.494	0.45	0.133
	LZ4HC	0.07	4.6e+02	22.92	0.454	0.443	0.6	0.143
	XZ 6	0.07	3.9e+02	22.37	0.341	0.331	0.45	0.119
	XZ 9	0.07	3.9e+02	22.37	0.341	0.331	0.45	0.119
enc	Deflate 1	0.28	6.3e+02	148.59	0.458	0.443	0.45	0.186
	Deflate 6	0.26	6e+02	128.16	0.449	0.43	0.45	0.191
	Deflate 9	0.26	6e+02	127.50	0.448	0.43	0.45	0.191
	LZ4	0.21	4.6e+02	94.55	0.725	0.674	0.95	0.238
	LZ4HC	0.21	1.4e+03	97.24	0.704	0.646	0.95	0.249
	XZ 6	0.26	8.1e+02	176.11	0.401	0.371	0.4	0.201
	XZ 9	0.26	8.1e+02	176.11	0.401	0.371	0.4	0.201

Ext.	Compressor	KS	χ^2	A	Mean	Median	Mode	SD
pfb	Deflate 1	0.24	2.2e+03	352.85	0.917	0.94	0.95	0.11
	Deflate 6	0.24	2.2e+03	339.68	0.914	0.938	0.95	0.112
	Deflate 9	0.24	2.2e+03	340.07	0.914	0.938	0.95	0.112
	LZ4	0.23	2.2e+03	331.13	0.925	0.946	0.95	0.101
	LZ4HC	0.23	2.1e+03	322.20	0.921	0.943	0.95	0.102
	XZ 6	0.23	2.1e+03	322.08	0.908	0.937	0.95	0.126
	XZ 9	0.23	2.1e+03	322.08	0.908	0.937	0.95	0.126
wmf	Deflate 1	0.10	2.2e+03	185.56	0.633	0.652	0.65	0.103
	Deflate 6	0.10	2.1e+03	170.89	0.621	0.64	0.65	0.106
	Deflate 9	0.10	2.1e+03	171.70	0.621	0.639	0.65	0.106
	LZ4	0.10	2.8e+03	216.64	0.822	0.845	0.85	0.121
	LZ4HC	0.10	2.7e+03	207.27	0.806	0.83	0.8	0.126
	XZ 6	0.04	1.5e+02	23.10	0.484	0.482	0.45	0.102
	XZ 9	0.04	1.5e+02	23.10	0.484	0.482	0.45	0.102
pnf	Deflate 1	0.15	9.9e+02	125.71	0.275	0.289	0.25	0.043
	Deflate 6	0.14	1e+03	129.88	0.249	0.262	0.25	0.04
	Deflate 9	0.14	1e+03	130.73	0.248	0.262	0.25	0.04
	LZ4	0.15	9.2e+02	124.24	0.384	0.404	0.4	0.059
	LZ4HC	0.14	8.8e+02	123.92	0.332	0.351	0.35	0.052
	XZ 6	0.18	7.3e+02	199.48	0.204	0.221	0.2	0.046
	XZ 9	0.18	7.3e+02	199.48	0.204	0.221	0.2	0.046

Ext.	Compressor	KS	χ^2	A	Mean	Median	Mode	SD
mof	Deflate 1	0.13	2e+03	136.95	0.254	0.219	0.2	0.152
	Deflate 6	0.12	2e+03	136.24	0.231	0.192	0.1	0.158
	Deflate 9	0.12	2e+03	135.15	0.23	0.192	0.1	0.158
	LZ4	0.10	9e+02	53.78	0.335	0.302	0.25	0.171
	LZ4HC	0.08	9.3e+02	61.24	0.299	0.264	0.2	0.179
	XZ 6	0.11	1.7e+03	114.99	0.231	0.197	0.1	0.171
	XZ 9	0.11	1.7e+03	114.99	0.231	0.197	0.1	0.171
class	Deflate 1	0.06	7.2e+02	48.42	0.475	0.484	0.45	0.067
	Deflate 6	0.06	7.2e+02	44.51	0.456	0.465	0.45	0.069
	Deflate 9	0.06	7.1e+02	43.61	0.456	0.465	0.45	0.069
	LZ4	0.05	4.2e+02	33.11	0.619	0.628	0.6	0.08
	LZ4HC	0.05	3.4e+02	32.92	0.59	0.6	0.6	0.085
	XZ 6	0.07	4.9e+02	68.60	0.441	0.456	0.45	0.082
	XZ 9	0.07	4.9e+02	68.60	0.441	0.456	0.45	0.082
sh	Deflate 1	0.03*	4.9**	3.49	0.406	0.408	0.4	0.099
	Deflate 6	0.04*	5.0**	3.61	0.383	0.384	0.35	0.105
	Deflate 9	0.04*	5.3**	3.66	0.383	0.384	0.35	0.105
	LZ4	0.03**	5.1e+01	3.61	0.553	0.555	0.55	0.136
	LZ4HC	0.04*	6.9e+01	3.89	0.517	0.518	0.5	0.145
	XZ 6	0.04*	4.5e+01	4.26	0.405	0.401	0.35	0.121
	XZ 9	0.04*	4.5e+01	4.26	0.405	0.401	0.35	0.121

Ext.	Compressor	KS	χ^2	A	Mean	Median	Mode	SD
afm	Deflate 1	0.08	8.9*	28.40	0.303	0.305	0.3	0.077
	Deflate 6	0.09	3.6e+01	25.43	0.261	0.262	0.25	0.072
	Deflate 9	0.10	4.6e+01	26.82	0.258	0.26	0.25	0.072
	LZ4	0.06	3.1***	12.66	0.435	0.434	0.4	0.107
	LZ4HC	0.11	1e+02	26.76	0.36	0.364	0.35	0.095
	XZ 6	0.14	2e+02	39.22	0.219	0.221	0.2	0.079
	XZ 9	0.14	2e+02	39.22	0.219	0.221	0.2	0.079
adml	Deflate 1	0.26	8.6e+02	342.58	0.402	0.318	0.25	0.242
	Deflate 6	0.24	8.4e+02	318.81	0.377	0.294	0.2	0.254
	Deflate 9	0.24	8.4e+02	317.96	0.377	0.294	0.2	0.254
	LZ4	0.14	4.8e+02	154.18	0.504	0.439	0.35	0.219
	LZ4HC	0.13	4.8e+02	148.25	0.465	0.41	0.25	0.237
	XZ 6	0.18	7.2e+02	248.06	0.392	0.306	0.2	0.262
	XZ 9	0.18	7.2e+02	248.06	0.392	0.306	0.2	0.262
zip	Deflate 1	0.28	8.4e+02	281.73	0.887	0.982	0.95	0.2
	Deflate 6	0.28	8.3e+02	281.32	0.882	0.981	0.95	0.207
	Deflate 9	0.28	8.3e+02	281.20	0.882	0.981	0.95	0.207
	LZ4	0.29	9e+02	271.67	0.899	0.991	1.0	0.186
	LZ4HC	0.28	8.5e+02	261.74	0.889	0.985	0.95	0.196
	XZ 6	0.26	7.2e+02	250.29	0.867	0.978	0.95	0.217
	XZ 9	0.26	7.1e+02	246.84	0.866	0.978	0.95	0.217

Ext.	Compressor	KS	χ^2	A	Mean	Median	Mode	SD
qml	Deflate 1	0.10	2.4e+02	58.77	0.345	0.349	0.35	0.055
	Deflate 6	0.10	2.3e+02	49.23	0.318	0.323	0.3	0.059
	Deflate 9	0.10	2.2e+02	49.55	0.318	0.323	0.3	0.06
	LZ4	0.11	2.1e+02	66.53	0.481	0.489	0.5	0.077
	LZ4HC	0.10	1.6e+02	58.71	0.441	0.45	0.45	0.085
	XZ 6	0.08	2.2e+02	35.72	0.333	0.337	0.35	0.07
	XZ 9	0.08	2.2e+02	35.72	0.333	0.337	0.35	0.07
man	Deflate 1	0.09	2.5e+02	42.25	0.354	0.372	0.45	0.124
	Deflate 6	0.08	2.3e+02	40.10	0.339	0.358	0.4	0.125
	Deflate 9	0.08	2.3e+02	40.71	0.338	0.358	0.4	0.126
	LZ4	0.09	2.4e+02	40.79	0.468	0.487	0.55	0.167
	LZ4HC	0.09	2.3e+02	39.96	0.445	0.468	0.55	0.167
	XZ 6	0.08	2.1e+02	36.93	0.356	0.376	0.45	0.139
	XZ 9	0.08	2.1e+02	36.93	0.356	0.376	0.45	0.139
ps1	Deflate 1	0.09	1.3e+03	35.85	0.358	0.361	0.35	0.101
	Deflate 6	0.08	1.1e+03	29.78	0.336	0.339	0.35	0.108
	Deflate 9	0.08	1.1e+03	30.02	0.335	0.339	0.35	0.108
	LZ4	0.08	2.3e+02	17.84	0.474	0.477	0.45	0.121
	LZ4HC	0.06	1.8e+02	15.95	0.439	0.446	0.45	0.13
	XZ 6	0.06	6.4e+02	18.44	0.356	0.369	0.35	0.119
	XZ 9	0.06	6.4e+02	18.44	0.356	0.369	0.35	0.119

Ext.	Compressor	KS	χ^2	A	Mean	Median	Mode	SD
odl	Deflate 1	0.23	4.1e+01	39.90	0.299	0.327	0.3	0.085
	Deflate 6	0.25	4.8e+01	46.56	0.284	0.316	0.3	0.088
	Deflate 9	0.25	4.9e+01	46.47	0.284	0.316	0.3	0.089
	LZ4	0.31	1.1e+02	80.79	0.411	0.471	0.45	0.116
	LZ4HC	0.32	1.1e+02	86.73	0.387	0.451	0.45	0.122
	XZ 6	0.15	3.3e+01	20.24	0.224	0.2	0.25	0.089
	XZ 9	0.15	3.3e+01	20.24	0.224	0.2	0.25	0.089
otf	Deflate 1	0.06	1.4e+01*	8.30	0.649	0.664	0.65	0.121
	Deflate 6	0.06	1.5e+01	8.31	0.633	0.651	0.65	0.128
	Deflate 9	0.06	1.5e+01	8.32	0.632	0.65	0.65	0.128
	LZ4	0.06	1e+02	14.98	0.765	0.779	0.8	0.124
	LZ4HC	0.06	5e+01	12.36	0.712	0.728	0.75	0.134
	XZ 6	0.09	1.2***	13.22	0.554	0.566	0.4	0.131
	XZ 9	0.09	1.3***	13.14	0.554	0.566	0.4	0.131
nib	Deflate 1	0.26	6.4e+01	29.78	0.561	0.652	0.65	0.215
	Deflate 6	0.24	6.3e+01	29.38	0.547	0.638	0.65	0.215
	Deflate 9	0.24	6.3e+01	29.47	0.547	0.638	0.65	0.215
	LZ4	0.23	7e+01	29.40	0.647	0.741	0.8	0.245
	LZ4HC	0.23	6.5e+01	27.85	0.626	0.721	0.75	0.244
	XZ 6	0.20	4.5e+01	19.39	0.487	0.561	0.6	0.199
	XZ 9	0.20	4.5e+01	19.39	0.487	0.561	0.6	0.199

Ext.	Compressor	KS	χ^2	A	Mean	Median	Mode	SD
md5sums	Deflate 1	0.03*	2.3e+02	3.42	0.395	0.396	0.4	0.066
	Deflate 6	0.03**	2.1e+02	3.34	0.381	0.381	0.35	0.067
	Deflate 9	0.02***	1.9e+02	3.20	0.378	0.378	0.35	0.068
	LZ4	0.03*	2.3e+02	4.97	0.589	0.591	0.55	0.101
	LZ4HC	0.03*	2e+02	4.48	0.569	0.572	0.55	0.106
	XZ 6	0.03*	1.1e+01*	3.99	0.391	0.391	0.4	0.094
	XZ 9	0.03*	1.1e+01*	3.99	0.391	0.391	0.4	0.094
cdxml	Deflate 1	0.18	1.1e+03	162.58	0.273	0.203	0.15	0.195
	Deflate 6	0.19	1.1e+03	174.24	0.249	0.175	0.1	0.201
	Deflate 9	0.19	1.1e+03	173.10	0.248	0.173	0.1	0.202
	LZ4	0.17	5.2e+02	109.80	0.343	0.271	0.2	0.217
	LZ4HC	0.18	5.9e+02	131.51	0.312	0.233	0.15	0.222
	XZ 6	0.19	1.1e+03	155.13	0.257	0.181	0.1	0.205
	XZ 9	0.19	1.1e+03	155.13	0.257	0.181	0.1	0.205
pyi	Deflate 1	0.02	1.5e+01	3.35	0.299	0.297	0.25	0.07
	Deflate 6	0.03	3.2e+01	4.37	0.275	0.274	0.25	0.072
	Deflate 9	0.03	3.1e+01	4.26	0.274	0.274	0.25	0.073
	LZ4	0.02*	2.2e+01	3.20	0.416	0.416	0.35	0.096
	LZ4HC	0.02*	2.4e+01	3.21	0.376	0.375	0.3	0.1
	XZ 6	0.03	7.6e+01	8.14	0.298	0.297	0.25	0.087
	XZ 9	0.03	7.6e+01	8.14	0.298	0.297	0.25	0.087

Ext.	Compressor	KS	χ^2	A	Mean	Median	Mode	SD
pyo	Deflate 1	0.03	8.3e+01	5.59	0.404	0.408	0.4	0.077
	Deflate 6	0.02*	2.2e+01	2.01	0.374	0.378	0.35	0.083
	Deflate 9	0.02*	2.3e+01	2.12	0.373	0.377	0.35	0.084
	LZ4	0.03*	5.3e+01	4.54	0.546	0.552	0.55	0.108
	LZ4HC	0.02**	1.5e+01	2.14	0.486	0.485	0.4	0.12
	XZ 6	0.03	4.6e+01	7.66	0.35	0.344	0.3	0.094
	XZ 9	0.03	4.6e+01	7.66	0.35	0.344	0.3	0.094
tcl	Deflate 1	0.09	2.7e+03	96.76	0.364	0.357	0.35	0.106
	Deflate 6	0.08	2.6e+03	88.31	0.333	0.321	0.3	0.113
	Deflate 9	0.08	2.6e+03	86.63	0.333	0.321	0.3	0.113
	LZ4	0.05	4.8e+02	23.69	0.504	0.496	0.45	0.13
	LZ4HC	0.05	5.9e+02	25.26	0.452	0.436	0.4	0.142
	XZ 6	0.08	2e+03	70.23	0.342	0.321	0.3	0.126
	XZ 9	0.08	2e+03	70.23	0.342	0.321	0.3	0.126
ui	Deflate 1	0.14	9.2e+01	20.27	0.169	0.148	0.1	0.076
	Deflate 6	0.15	1e+02	22.99	0.142	0.12	0.05	0.077
	Deflate 9	0.15	9.9e+01	23.05	0.139	0.117	0.05	0.078
	LZ4	0.14	9.2e+01	19.18	0.229	0.2	0.15	0.106
	LZ4HC	0.15	1e+02	21.96	0.185	0.154	0.1	0.107
	XZ 6	0.16	1e+02	24.67	0.147	0.121	0.05	0.089
	XZ 9	0.16	1e+02	24.67	0.147	0.121	0.05	0.089

Ext.	Compressor	KS	χ^2	A	Mean	Median	Mode	SD
p7x	Deflate 1	0.43	1e+05	405.81	0.889	1.004	1.0	0.171
	Deflate 6	0.43	7.5e+04	406.54	0.886	1.004	1.0	0.175
	Deflate 9	0.43	7.7e+04	406.49	0.886	1.004	1.0	0.175
	LZ4	0.43	7.5e+04	405.05	0.894	1.004	1.0	0.162
	LZ4HC	0.43	8e+04	407.49	0.89	1.004	1.0	0.168
	XZ 6	0.42	6.3e+04	402.07	0.89	1.016	1.0	0.186
	XZ 9	0.42	6.3e+04	402.07	0.89	1.016	1.0	0.186
dds	Deflate 1	0.07	2.4e+03	34.82	0.488	0.503	0.75	0.282
	Deflate 6	0.07	2.6e+03	34.25	0.467	0.472	0.0	0.284
	Deflate 9	0.07	2.8e+03	35.49	0.464	0.463	0.0	0.285
	LZ4	0.10	7.4e+03	67.88	0.585	0.625	1.0	0.328
	LZ4HC	0.08	1.1e+04	48.42	0.514	0.524	0.9	0.308
	XZ 6	0.07	6.4e+02	26.18	0.389	0.378	0.0	0.249
	XZ 9	0.07	6.4e+02	26.13	0.389	0.378	0.0	0.248
tiff	Deflate 1	0.11**	3.6e+01	1.96	0.623	0.616	0.95	0.264
	Deflate 6	0.11**	3.6e+01	2.00	0.61	0.595	0.95	0.273
	Deflate 9	0.11**	3.6e+01	2.01	0.609	0.594	0.95	0.274
	LZ4	0.11**	2.6e+01	2.60	0.687	0.726	0.95	0.25
	LZ4HC	0.12*	2.7e+01	2.44	0.654	0.692	0.95	0.265
	XZ 6	0.10***	4.8e+01	1.95	0.575	0.561	0.95	0.281
	XZ 9	0.10***	4.8e+01	1.95	0.575	0.561	0.95	0.281

Ext.	Compressor	KS	χ^2	A	Mean	Median	Mode	SD
list	Deflate 1	0.24	1.2e+03	256.69	0.229	0.191	0.15	0.158
	Deflate 6	0.27	1.2e+03	283.66	0.21	0.173	0.15	0.159
	Deflate 9	0.26	1.2e+03	274.18	0.208	0.171	0.15	0.159
	LZ4	0.20	9.9e+02	191.50	0.31	0.267	0.25	0.169
	LZ4HC	0.20	9.9e+02	187.72	0.278	0.242	0.2	0.161
	XZ 6	0.20	1e+03	171.12	0.223	0.193	0.15	0.157
	XZ 9	0.20	1e+03	171.12	0.223	0.193	0.15	0.157
translation	Deflate 1	0.10	4e+02	128.40	0.143	0.137	0.1	0.03
	Deflate 6	0.13	5.5e+02	184.19	0.114	0.107	0.1	0.029
	Deflate 9	0.13	5.9e+02	191.44	0.11	0.104	0.1	0.029
	LZ4	0.08	2.6e+02	108.68	0.187	0.179	0.15	0.04
	LZ4HC	0.14	5.7e+02	200.84	0.135	0.126	0.1	0.037
	XZ 6	0.15	6.7e+02	228.04	0.101	0.094	0.05	0.029
	XZ 9	0.15	6.7e+02	228.04	0.101	0.094	0.05	0.029
stringsdict	Deflate 1	0.05	4.4e+02	52.43	0.453	0.465	0.5	0.114
	Deflate 6	0.05	5e+02	48.01	0.435	0.446	0.5	0.118
	Deflate 9	0.05	5e+02	48.35	0.435	0.446	0.5	0.118
	LZ4	0.06	5.3e+02	80.58	0.546	0.565	0.6	0.132
	LZ4HC	0.05	6.2e+02	68.42	0.517	0.533	0.6	0.138
	XZ 6	0.05	8.8e+02	60.68	0.422	0.432	0.5	0.124
	XZ 9	0.05	8.8e+02	60.68	0.422	0.432	0.5	0.124

Ext.	Compressor	KS	χ^2	A	Mean	Median	Mode	SD
jsonlz4	Deflate 1	0.24	2.6e+03	256.75	0.846	0.852	0.85	0.021
	Deflate 6	0.24	2.8e+03	271.32	0.844	0.852	0.85	0.023
	Deflate 9	0.24	2.8e+03	271.48	0.844	0.852	0.85	0.023
	LZ4	0.35	4.2e+03	538.91	0.997	1.004	1.0	0.017
	LZ4HC	0.32	3.6e+03	466.77	0.99	1.003	1.0	0.028
	XZ 6	0.24	3.5e+03	278.15	0.811	0.825	0.8	0.041
	XZ 9	0.24	3.5e+03	278.15	0.811	0.825	0.8	0.041
mp3	Deflate 1	0.24	1e+03	171.56	0.923	0.956	0.95	0.107
	Deflate 6	0.23	1e+03	167.21	0.92	0.951	0.95	0.108
	Deflate 9	0.23	1e+03	166.74	0.92	0.951	0.95	0.108
	LZ4	0.26	1e+03	187.14	0.935	0.976	0.95	0.107
	LZ4HC	0.23	1e+03	171.89	0.924	0.959	0.95	0.108
	XZ 6	0.22	9.8e+02	157.78	0.916	0.947	0.95	0.111
	XZ 9	0.22	9.8e+02	157.46	0.916	0.947	0.95	0.111
docx	Deflate 1	0.09	1.4e+02	6.14	0.853	0.849	0.8	0.069
	Deflate 6	0.09	1.3e+02	5.98	0.849	0.845	0.8	0.07
	Deflate 9	0.09	1.3e+02	5.91	0.849	0.844	0.8	0.07
	LZ4	0.10	1.3e+02	5.75	0.862	0.856	0.8	0.07
	LZ4HC	0.09	1.3e+02	5.89	0.853	0.846	0.8	0.071
	XZ 6	0.11	2.6e+02	9.53	0.847	0.843	0.8	0.077
	XZ 9	0.11	2.6e+02	9.53	0.847	0.843	0.8	0.077

B Extensions Not Considered

We chose to skip many file extensions that appeared in the top 50. Some rationale for this decision follows for each extension or group of extensions.

B.1 H, HPP, C, PY, PYC, CPP, JAVA, GO, LUA

Our participants were largely part of the computer science world, and so we saw many source code files rise to the top of the list. We chose to skip deeper dives into these extensions partly because we assumed JS compressibility could generalize to other source code files (and according to Section 4.3.2 this was true), but also because they are almost certainly less common in the general population. Since any internet user commonly interacts with JavaScript behind the scenes, we considered this code ubiquitous enough to warrant analysis. Additionally, these files were very small, which is a problem we address in discussion Section 5.2.

B.2 TFM, TEX, VF

These extensions correspond to TeX. TeX files are probably less common in the general population, and so we did not consider them interesting. TFM and VF files are associated with TeX fonts, and since we analyzed TTF, we did not believe we needed to analyze more font files. Additionally, VF files were very small on average (16 KB), and so they are generally not targets for compression.

B.3 DAT

DAT is associated with "data," which is too broad to make assumptions about content. This extension might mimic the chaos of BIN, and it may be very user-dependent.

B.3.1 STRINGS, NIB, MANIFEST, KO, PM, MUM, MUI, etc.

These extensions are highly system-dependent. STRINGS and NIB are almost exclusive to Mac, KO and PM to Linux, MUM and MUI to Windows. Analyzing these extensions would greatly reduce the number of participants, e.g., we only had two Mac users. Some of these extensions, however, are very common, and so it may be worthwhile to explore them in future work.

C Rsync Compression Skip List

Table C.1: File Extensions that Rsync Does Not Compress

3g2	crypt8	m4p	rpm
3gp	deb	m4v	rzip
gpp	dmg	mkv	s7z
7z	drc	msi	sfx
aac	ear	mov	svgz
ace	gz	mp3	tbz
amr	flac	mp4	tgz
apk	flv	mpeg	tlz
appx	gpg	mpg	txz
appxbundle	iso	mpv	vob
arc	jar	oga	wim
arj	jp2	ogg	wma
asf	jpg	ogv	wmv
avi	jpeg	opus	xz
bz2	lz	pack	z
cab	lzma	png	zip
crypt5	lzo	qt	zst
crypt7	m4a	rar	

References

[1] *IDC's Global DataSphere Forecast Shows Continued Steady Growth in the Creation and Consumption of Data*, en, May 2020. [Online]. Available: https://www.idc.com/getdoc.jsp?containerId=prUS46286020 (visited on 04/12/2021) (cit. on p. 1).

[2] S. Morgan, *The world will store 200 zettabytes of data by 2025*, Jun. 2020. [Online]. Available: https://cybersecurityventures.com/the-world-will-store-200-zettabytes-of-data-by-2025/ (cit. on p. 1).

[3] C. E. Shannon, "A mathematical theory of communication," *Bell System Technical Journal*, vol. 27, no. 4, pp. 623–656, 1948. DOI: 10.1002/j.1538-7305.1948.tb00917.x (cit. on p. 4).

[4] D. Salomon, D. Bryant, and G. Motta, *Data Compression: The Complete Reference*, eng, Fourth Edition. London: Springer London, Limited, 2006, ISBN: 1846286026 (cit. on pp. 5–9, 23, 40, 41, 47).

[5] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," eng, *IEEE transactions on information theory*, vol. 23, no. 3, pp. 337–343, 1977, ISSN: 0018-9448 (cit. on p. 5).

[6] J. Ziv and A. Lempel, "Compression of individual sequences via variable-rate coding," *IEEE Transactions on Information Theory*, vol. 24, no. 5, pp. 530–536, 1978 (cit. on p. 5).

[7] D. A. Huffman, "A method for the construction of minimum-redundancy codes," eng, *Proceedings of the IRE*, vol. 40, no. 9, pp. 1098–1101, 1952, ISSN: 0096-8390 (cit. on p. 6).

[8] I. H. Witten, R. M. Neal, and J. G. Cleary, "Arithmetic encoding for data compression," en, *Communications of the ACM*, vol. 30, no. 6, pp. 520–540, Jun. 1987 (cit. on p. 7).

[9] R. C. Pasco, "Source Coding Algorithms for Fast Data Compression," en, PhD thesis, Stanford University, May 1976 (cit. on p. 7).

[10] P. W. Katz, "String searcher, and compressor using same," US5051745A, Sep. 1991. [Online]. Available: https://patents.google.com/patent/US5051745/en (visited on 10/15/2020) (cit. on p. 7).

[11] L. P. Deutsch, "DEFLATE Compressed Data Format Specification version 1.3," en, RFC Editor, Tech. Rep. 1951, May 1996. [Online]. Available: https://tools.ietf.org/html/rfc1951 (visited on 10/15/2020) (cit. on p. 8).

[12] G. Roelofs and M. Adler, *Zlib Home Site*, Dec. 2017. [Online]. Available: http://zlib.net/ (visited on 10/15/2020) (cit. on p. 8).

[13] T. Boutell, "PNG (Portable Network Graphics) Specification Version 1.0," en, RFC Editor, Tech. Rep. 2083, Mar. 1997. [Online]. Available: https://www.rfc-editor.org/rfc/pdfrfc/rfc2083.txt.pdf (visited on 10/29/2020) (cit. on p. 8).

[14] T. Bienz, R. Cohn, and J. R. Meehan, *Portable Document Format Reference Manual Version 1.2*, Adobe Systems Incorporated, Nov. 1996. [Online]. Available: https://web.archive.org/web/20051103044315/http://www.pdf-tools.com/public/downloads/pdf-reference/pdfreference12.pdf (visited on 10/29/2020) (cit. on p. 8).

[15] J. Storer and T. Szymanski, "Data compression via textual substitution," eng, *Journal of the ACM (JACM)*, vol. 29, no. 4, pp. 928–951, 1982, ISSN: 1557-735X (cit. on p. 8).

[16] Y. Collet, *Lz4/lz4*, original-date: 2014-03-25T15:52:21Z, Aug. 2020. [Online]. Available: https://github.com/lz4/lz4 (visited on 10/19/2020) (cit. on p. 8).

[17] Y. Collet and T. Matsuoka, *LZ4 - Extremely fast compression*, Aug. 2020. [Online]. Available: https://lz4.github.io/lz4/ (visited on 10/19/2020) (cit. on p. 8).

[18] *OpenZFS*. [Online]. Available: https://openzfs.org/wiki/Main_Page (visited on 10/19/2020) (cit. on p. 8).

[19] N. Bunge, "Quality of service improvement in ZFS through compression," Master's thesis, Universität Hamburg, May 2017 (cit. on pp. 9, 11).

[20] A. Gupta, A. Bansal, and V. Khanduja, "Modern lossless compression techniques: Review, comparison and analysis," in *2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT)*, IEEE, Feb. 2017, pp. 1–8. DOI: 10.1109/ICECCT.2017.8117850 (cit. on p. 9).

[21] I. Pavlov, *7z Format*, Apr. 2019. [Online]. Available: https://www.7-zip.org (visited on 10/21/2020) (cit. on pp. 9, 18).

[22] *Openvpn(8) - linux man page* (cit. on p. 11).

[23] D. Asamoah Owusu, "Modeling Outputs of Efficient Compressibility Estimators," Master's thesis, University of Minnesota Duluth, Jun. 2018 (cit. on pp. 11, 14).

[24] W. Culhane, "Statistical Measures as Predictors of Compression Savings," Senior Honors Thesis, Ohio State University, May 2008 (cit. on pp. 11, 16).

[25] P. A. H. Peterson and P. L. Reiher, "Datacomp: Locally independent adaptive compression for real-world systems," in *2016 IEEE 36th International Conference on Distributed Computing Systems (ICDCS)*, Jun. 2016, pp. 211–220. DOI: 10.1109/ICDCS.2016.106 (cit. on pp. 11, 15, 79).

[26] *Rsync(1) - linux man page* (cit. on p. 11).

[27] Y. Xiao, M. Siekkinen, and A. Ylä-Jääski, "Framework for energy-aware lossless compression in mobile services: The case of e-mail," May 2010, pp. 1–6. DOI: 10.1109/ICC.2010.5502590 (cit. on p. 11).

[28] *Mod_deflate - Apache HTTP Server Version 2.4*. [Online]. Available: https://httpd.apache.org/docs/2.4/mod/mod_deflate.html (visited on 05/24/2021) (cit. on p. 11).

[29] F. Ehmke, "Adaptive Compression for the Zettabyte File System," Master's thesis, Universität Hamburg, Hamburg, DE, Feb. 2015 (cit. on pp. 11, 13).

[30] *Using and Running Mirrors*, May 2021. [Online]. Available: https://www.gnu.org/server/mirror.html (visited on 06/16/2021) (cit. on p. 11).

[31] S. Hamminga, *List of compressed file formats for use with Rsync –skip-compress*, en. [Online]. Available: https://gist.github.com/StefanHamminga/2b1734240025f5ee916 (visited on 09/23/2020) (cit. on p. 11).

[32] *Common file name extensions in Windows.* [Online]. Available: https://support.microsoft.com/en-us/windows/common-file-name-extensions-in-windows-da4a4430-8e76-89c5-59f7-1cdbbc75cb01 (visited on 10/28/2020) (cit. on pp. 11, 61).

[33] H. Devarajan, A. Kougkas, and X.-H. Sun, "An Intelligent, Adaptive, and Flexible Data Compression Framework," en, in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, Larnaca, Cyprus: IEEE, May 2019, pp. 82–91, ISBN: 978-1-72810-912-1. DOI: 10.1109/CCGRID.2019.00019. [Online]. Available: https://ieeexplore.ieee.org/document/8752926/ (visited on 09/22/2020) (cit. on p. 13).

[34] T. Titovets, *Nefelim4ag/Entropy_calculation*, original-date: 2017-06-19T13:15:47Z, Jun. 2017. [Online]. Available: https://github.com/Nefelim4ag/Entropy_Calculation (visited on 09/23/2020) (cit. on p. 15).

[35] D. Harnik, R. Kat, D. Sotnikov, A. Traeger, and O. Margalit, "To zip or not to zip: Effective resource usage for real-time compression," in *11th USENIX Conference on File and Storage Technologies (FAST 13)*, San Jose, CA: USENIX Association, Feb. 2013, pp. 229–241, ISBN: 978-1-931971-99-7. [Online]. Available: https://www.usenix.org/conference/fast13/technical-sessions/presentation/harnik (cit. on p. 15).

[36] *Deflater (Java Platform SE 8)*, Jul. 2020. [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/util/zip/Deflater.html (visited on 10/31/2020) (cit. on p. 18).

[37] *Lz4/lz4-java*, original-date: 2012-07-18T17:26:42Z, Feb. 2021. [Online]. Available: https://github.com/lz4/lz4-java (visited on 02/16/2021) (cit. on p. 18).

[38] *XZ utils*, Jan. 11, 2021. [Online]. Available: https://tukaani.org/xz/ (visited on 02/23/2021) (cit. on p. 18).

[39] *1.3.5.16. Kolmogorov-Smirnov Goodness-of-Fit Test*, Jan. 2018. [Online]. Available: https://www.itl.nist.gov/div898/handbook/eda/section3/eda35g.htm (visited on 04/12/2021) (cit. on p. 24).

[40] *1.3.5.14. Anderson-Darling Test*, Jan. 2018. [Online]. Available: https://www.itl.nist.gov/div898/handbook/eda/section3/eda35e.htm (visited on 04/12/2021) (cit. on p. 25).

[41] *1.3.5.15. Chi-Square Goodness-of-Fit Test*, Jan. 2018. [Online]. Available: https://www.itl.nist.gov/div898/handbook/eda/section3/eda35f.htm (visited on 04/12/2021) (cit. on p. 25).

[42] *Binary file Definition*, Feb. 2006. [Online]. Available: http://www.linfo.org/binary_file.html (visited on 05/27/2021) (cit. on p. 32).

[43] Deland-Han, darugeri, helenclu, and simonxjx, *What is a DLL*, en-us, Sep. 2020. [Online]. Available: https://docs.microsoft.com/en-us/troubleshoot/windows-client/deployment/dynamic-link-library (visited on 05/27/2021) (cit. on p. 33).

[44] M. Oberhumer, L. Molnar, and J. Reiser, *UPX: The Ultimate Packer for eXecutables - Homepage.* [Online]. Available: https://upx.github.io/ (visited on 06/14/2021) (cit. on p. 34).

[45] *[ms-docx]: Word extensions to the office open xml (.docx) file format*, v20210216, Rev. 14.1, Microsoft Corporation, Feb. 2021 (cit. on p. 35).

[46] *.EXE File Extension*, May 2021. [Online]. Available: https://fileinfo.com/extension/exe (visited on 05/28/2021) (cit. on p. 39).

[47] "Graphics Interchange Format Version 89a," Compuserve Incorporated, Columbus, Ohio, Tech. Rep., Jul. 1990 (cit. on p. 42).

[48] P. F. Brown, S. A. D. Pietra, V. J. D. Pietra, J. C. Lai, and R. L. Mercer, "An Estimate of an Upper Bound for the Entropy of English," en, *Computational Linguistics*, vol. 18, p. 10, Mar. 1992. [Online]. Available: https://www.cs.cmu.edu/~roni/11761/PreviousYearsHandouts/gauntlet.pdf (cit. on p. 48).

[49] *Introducing JSON*. [Online]. Available: https://www.json.org/json-en.html (visited on 05/28/2021) (cit. on p. 49).

[50] *Vorbis I specification*, Jul. 2020. [Online]. Available: https://www.xiph.org/vorbis/doc/Vorbis_I_spec.html (visited on 05/28/2021) (cit. on p. 51).

[51] I. E. Gonsalves, S. Pfeiffer, and C. Montgomery, "Ogg Media Types," RFC Editor, RFC 5334, Sep. 2008. [Online]. Available: https://www.xiph.org/ogg/doc/rfc5334.txt (visited on 05/28/2021) (cit. on p. 51).

[52] *What is a PDF?* en. [Online]. Available: https://acrobat.adobe.com/us/en/acrobat/about-adobe-pdf.html (visited on 05/31/2021) (cit. on p. 53).

[53] D. Lukan, *PDF file format: Basic structure [updated 2020]*, en-US, Sep. 2020. [Online]. Available: https://resources.infosecinstitute.com/topic/pdf-file-format-basic-structure/ (visited on 05/31/2021) (cit. on p. 53).

[54] G. Roelofs, *Portable Network Graphics: An Open, Extensible Image Format with Lossless Compression*, Aug. 2020. [Online]. Available: http://www.libpng.org/pub/png/ (visited on 05/31/2021) (cit. on p. 54).

[55] *.RTF File Extension*, Apr. 2021. [Online]. Available: https://fileinfo.com/extension/rtf (visited on 06/10/2021) (cit. on p. 56).

[56] M. Murata, S. St.Laurent, and D. Kohn, “XML Media Types,” RFC Editor, RFC 3023, Jan. 2001. [Online]. Available: https://datatracker.ietf.org/doc/html/rfc3023 (visited on 06/01/2021) (cit. on p. 58).

[57] *.TTF File Extension*, May 2016. [Online]. Available: https://fileinfo.com/extension/ttf (visited on 06/11/2021) (cit. on p. 59).

[58] alib-ms, PeterCon, WorkNihar, and mijacobs, *TrueType hinting tutorial - Basic hinting philosophies and TrueType instructions*, en-us, May 2018. [Online]. Available: https://docs.microsoft.com/en-us/typography/truetype/hinting-tutorial/hinting-and-truetype-instructions (visited on 06/14/2021) (cit. on p. 59).

[59] T. Blancpain, *Understanding Web Fonts and Getting the Most Out of Them*, en, Feb. 2018. [Online]. Available: https://css-tricks.com/understanding-web-fonts-getting/ (visited on 06/14/2021) (cit. on p. 59).

[60] *Multimedia Programming Interface and Data Specifications 1.0*, Aug. 1991. [Online]. Available: https://www.aelius.com/njh/wavemetatools/doc/riffmci.pdf (cit. on p. 64).

[61] K. Sayood, *Introduction to Data Compression*, 5th ed. Waltham, MA: Morgan Kaufmann, 2018 (cit. on p. 67).

[62] J. Beaulieu, “Adaptive Filesystem Compression for General Purpose Systems,” Master’s thesis, University of Minnesota Duluth, Jun. 2018 (cit. on p. 79).