ISSN 0554-6397
Stručni članak
(*Professional paper*)

Deni Klen
E-mail: dklen@riteh.hr
Jonatan Lerga
E-mail: jlerga@riteh.hr
University of Rijeka, Faculty of Engineering, Vukovarska 58, 51000 Rijeka, Croatia
Irena Petrijevčanin
E-mail: ipetrijevcanin@uniri.hr
Center for Artificial Intelligence and Cybersecurity, University of Rijeka, R. Matejcic 2, 51000 Rijeka, Croatia

Txt and Tif File Compression Using Lzw, Huffman, and Arithmetic Coding

Abstract

The paper compares compression methods such as Lempel-Ziv-Welch (LZW), Huffman, and arithmetic coding applied to different large text and image datasets. Comparison is done based on metrics such as execution time and compression ratio. LZW produced results of about 30 % median compression ratio for all text records and a median of about 70 % for image records. In addition, Huffman coding produced a compression rate of about 40 % median for text data and a median of about 55 % for image data. Finally, arithmetic coding yielded results of about 70 % median for text compression and about 55 % median for image data compression. The time required was lowest for LZW, followed by Huffman, and worst for arithmetic coding.

Keywords: coding, LZW, Huffman, arithmetic coding, compression

1.1. Introduction

As our dependence on electronic media increases significantly each year with the advancement of the digital age, so does the need to store these vast amounts of data. Storage solutions become increasingly important as more data is created, processed, and exchanged digitally. This is true for personal file storage and businesses and organizations that need to store large amounts of data for various purposes such as analysis, research, and development [2]. Therefore, investing in efficient data compression techniques is crucial to keep up with the ever-increasing demand for data

processing and transfer. So, what exactly is data compression? Data compression is an encoding technique used to transfer data from one representation to another, resulting in a reduction in file or data size. In other words, the number of bits needed to represent and store the data is smaller. However, from an information-theory point-of-view, the main goal is to minimize the amount of data to be transmitted or stored [3]. Compression algorithms can be divided into two types: (1) lossless and (2) lossy. Although their names are self-explanatory, no data loss occurs with lossless data compression. The compression is performed by the file with a smaller number of bits without losing any information. On the other hand, lossy compression removes the unnecessary data, reducing the size of the file [4]. The compressed file cannot be converted back to the original file but rather to its approximation because some information is lost during compression [5]. This paper briefly explains the operation of well-known lossless compression algorithms such as Lempel-Ziv-Welch (LZW), Huffman [6], and arithmetic coding. Then, they are compared based on their compression ratio, and compression and decompression speed. The rest of the paper is structured as follows: Section II discusses traditional and modern compression algorithms. Section III explains and collects information about the database used and compares the algorithms based on their performance on the database. Section V concludes the paper with a summary and suggestions for future work.

1.2. Material and methods

In this section, we briefly review selected compression algorithms used in our analysis.

1.2.1. Huffman compression

Huffman coding is a traditional compression algorithm based on the frequency of characters, such that the character with the highest frequency gets the shortest binary code. The algorithm was first proposed by David A. Huffman in 1952 and has since become a staple for data compression applications [7]. It is a widely used and implemented algorithm because it is fast, requires little computational power, and provides good to very good compression ratios.

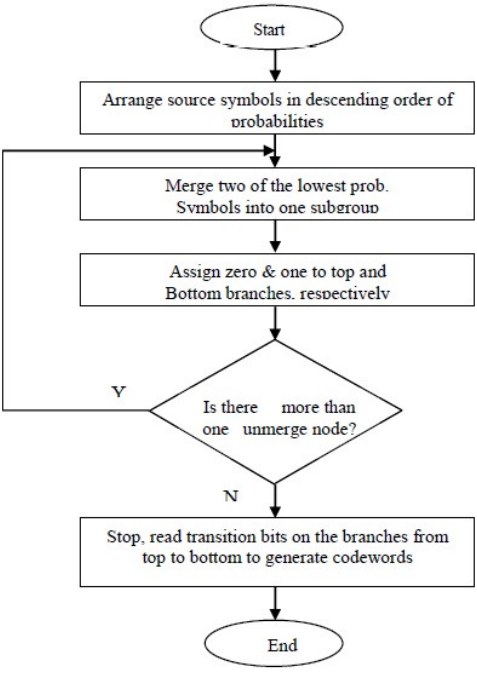

Fig. 1. Flowchart of Huffman algorithm taken from [8]

The Huffman coding algorithm goes through the data and creates a frequency table. It then recursively removes the last symbol from the table and merges it with the second most frequent symbol until only one node remains. At the end of the recursion, a tree is available from which an adaptive Huffman code can be generated for the given data set. Once the tree is built, starting with the root, one branch is assigned a 0, and the other branch is assigned a 1. As the tree is traversed, each symbol is assigned a code value that is replaced during encoding or decoding. Also, to produce compressed output, each symbol in the original code is replaced with the corresponding value from the tree. To decompress the Huffman encoding, each Huffman symbol must be replaced with the original symbol, as shown in Figure 1.

A. Implementation:

To implement and compress or decompress files with the Huffman algorithm library *dahuffman* [9] is needed to install it, we use the command

```
pip install dahuffman
```

Furthermore, implementing it is as follows for encoding, decoding, and training sets.

```
from dahuffman import HuffmanCodec
codec = HuffmanCodec.fromdata \ ( trainfile )
encodeddata = codec.encode ( inputfile )
decodeddata = codec.decode ( encodedfile )
```

1.2.2. Lempel-Ziv-Welch compression

The algorithm Lempel–Ziv–Welch, hence the name, was developed in 1984 by A. Lempel, J. Ziv, and T. Welch. The algorithm is widely used, especially for GIF, PDF, and TIFF formats. It uses a code table to represent a sequence of repeating bytes, allowing good compression rates to be achieved without data loss. Although compression rates can generally be high, the effectiveness depends heavily on the characteristics of the data to be compressed [10].

One of the advantages of LZW is its simplicity of implementation. In addition, its performance is not hardware heavy, which makes it a popular choice for various use cases. However, it should be noted that LZW is not the most efficient compression method for short and diverse data. Nevertheless, it is widely used due to its versatility, ease of implementation, and lossless data compression.

The algorithm works with an empty dictionary that is traversed from left to right in the original data to find sequences that are not in it. If the sequence is not in the dictionary, it is added with a representative code. To compress the file, we traverse the data and replace the longest repeating sequence with its code. Decompressing the file is done by searching backward for the codes in the dictionary.

A. Implementation:

To implement and compress or decompress files with the Lempel–Ziv–Welch algorithm library *lzwpython* [11] is needed. Implementation is as follows

```
import lzw
encodeddata = lzw.compress( f )
decodeddata = lzw.decompress( encodedfile )
```

Inside the compress function is also dictionary creation, so a separate function call is not needed.

1.2.3. Arithmetic coding

Arithmetic coding is a lossless data compression technique that was first introduced in 1976 and has become very popular due to its high compression ratios, lossless

capabilities, and wide range of applications. Due to its adaptability, it can result in more compact compressed data compared to other algorithms. Arithmetic coding is well suited for small data sets but requires more computational effort than the previously mentioned methods. As with Huffman coding, arithmetic coding requires the probability distribution of the input symbols to be known in advance [12]. The basic idea of arithmetic coding is to divide the unit interval into subintervals, each of which represents a particular letter. The smaller the subinterval, the more bits are needed to distinguish it from other subintervals. The idea of replacing each input symbol with a codeword is bypassed. Instead, a stream of input symbols is encoded with a single fraction, a number between 0 and 1, as compressed output [13].

A. Implementation:

To implement and compress or decompress files with the arithmetic coding algorithm, the library *arithmetic-compressor* [14] is needed. Implementation is as follows

```
from arithmeticcompressor import AECompressor
from arithmeticcompressor.models import MultiPPM,StaticModel
model = MultiPPM( allchars , models = 3)
codec_ae = AECompressor ( model )
compressed = coder.compress ( data )
decoded = coder.decompress ( data , originallength )
```

1.2.4. Evaluation

For the evaluation, which will be discussed later, we wrote our own methods that go through the entire folder we want to evaluate and take each file and run it through all the methods. Each method then has its own timer that records the time for the encoding and decoding process. This time does not include the time it takes to read or write. Once all that is done, we need a function to compare original files and compressed files. This is accomplished with the Python library *os*[15].

1.3. Case study

This section will describe the datasets used in this work and evaluate the thoughts and processes.

1.3.1. Data Sets

For successful implementation and evaluation of the results, we need to prepare data sets [16]. Compression datasets are used for comparison and evaluation of lossless

compression methods. Compression methods are usually applied to different file types, but for this work, we chose to apply and evaluate them to text and image files (more specifically, for the *.txt* and *.tif* formats). These formats were chosen because they contain unformatted objects with no special styling, compression methods, formatting, etc. Both datasets are divided into three groups. The text files are divided by context type. The first dataset contains 35 speeches by Donald Trump, the more versatile dataset in which words and characters are not often repeated. The second dataset contains song lyrics, in which words and characters are simpler and are often repeated. It contains 49 artists and all song lyrics from their music bibliography. The last dataset is a large file of movie scripts and books from which the movies were transcribed. Again, each file in this group is very space intensive, ranging from 15 MB to 100 MB. The image dataset is also divided into three groups by size: small - up to 15 MB, medium - from 15 to 100 MB, and large data over 100 MB. The first two datasets are based on a size score and the third on a context score, where each image is a copy of the same original grayscale image but has repeating pixels at different locations in the image, so we can see how pixel placement affects time and compression ratios.

1.3.2. Evaluation

The evaluation of our work includes the computation and implementation of all algorithms, as well as analysis of the results and drawing a conclusion from them. All implemented algorithms were executed on a platform whose specification is shown in Table 1. The results of the executed compression methods are stored in separate Excel and CSV tables, which are divided into data sets. To evaluate the lossless compression methods, the following data are stored: the original file size, the training or learning time of the encoder, the encoding time, the decoding time, and the compression ratio for each method. The compression ratio is the percentage difference of the file change between two files.

Table 1: specification of the platform that has been used for the execution of the programs.

RAM	DDR4 - 16GB
Processor	Ryzen 7 5800H
Number of cores of processor	16
Processor clock speed	3.2GHz
Operating system	Windows 11 Pro - 64 bit

1.4. Rezultati/Results

In this section, we present the raw results, divided and explained based on the input databases and show some selected results from each database that best describe the progress.

1.4.1. .txt

First, by looking through all the records, we can find and extract recurring patterns. Lempel–Ziv–Welch algorithm had a compression reduction of about 24 to 35 percent and a time rate that slowly increased with file size. In addition, the Huffman algorithm's compression size results were slightly better, ranging from 40 to 50 percent in reduced size of compressed data, but the combined time was about twice as long as the LZW algorithm. Finally, the arithmetic algorithm showed the best compression rates in the 70 to 75 percent range but with a much larger time overhead than the previous two algorithms. Now let us take a closer look at the individual data sets.

A. First dataset:

In the first data set, the results follow the pattern described earlier. The best compression ratio is obtained with the arithmetic coding and the best time with the Lempel–Ziv–Welch method. If we take a closer look at the

Table 2: compression rates dataset 1

Original size (B)	LZW (%)	Huffman (%)	Arithmetic (%)
14616	25.81	44.27	71.87
34486	26.98	43.86	71.99
49162	26.34	43.48	71.84
64560	26.47	43.20	71.78
78192	26.20	43.77	72.04
95916	26.47	43.39	71.88

results, we can see that the compression methods have almost the same efficiency for the same method over the whole data, as shown in table 2. On the other hand, the time cost is slightly larger than it may be expected on the arithmetic side. For the first two methods, Huffman and LZW, it increases slowly, while the time required for arithmetic coding increases in a much steeper curve, as shown in Table 3.

Table 3: combined time needed for compression and decompression dataset 1.

Original size (B)	LZW (s)	Huffman (s)	Arithmetic (s)
14616	0.016	0.020	14.806
34486	0.031	0.058	35.335
49162	0.047	0.062	51.567
64560	0.066	0.068	66.736
78192	0.081	0.103	81.177
95916	0.099	0.121	109.430

B. Second dataset:

The results of the second dataset show us that the compression ratio is consistent, but there is one important thing to mention. Files with more word repetitions and a larger number of the same characters have a slightly better compression rate than others. This is the expected result since all algorithms work with repeating characters and patterns within the compression code. For example, in the data shown in Table 4, it can be seen that the penultimate file has better compression than the others, even though it requires more memory. The time required for compression and decompression follows the previous conclusions and grows with file, as shown in table 5, but with the same execution options as described before. As can be seen, larger files with more repetitions in their content require more time than files with the same file size but whose content is more versatile. This difference is particularly evident in the arithmetic encoding of inputs five and six, where file number six takes less time than file five despite its larger size.

C. Third dataset:

On the third data set, we again see that the results are as for the previous groups. The file compression for each of them falls within the range described, as shown in the table 6, but the time consumption here is worth mentioning.

Table 4: compression rates dataset 2

Original size (B)	LZW (%)	Huffman (%)	Arithmetic (%)
77505	30.99	42.68	71.76
113457	30.17	42.85	42.93
143729	30.84	42.93	71.91
170292	22.73	41.63	71.68
210141	28.85	41.96	71.43

257379	27.26	45.34	72.20
322587	25.85	42.67	71.64

Table 5: combined time needed for compression and decompression dataset 2

Original size (B)	LZW (s)	Huffman (s)	Arithmetic (s)
77505	0.068	0.100	78.469
113457	0.105	0.145	124.278
143729	0.133	0.186	159.343
170292	0.163	0.219	192.796
210141	0.202	0.293	252.443
257379	0.240	0.293	250.621
322587	0.312	0.419	422.772

As we can see with arithmetic coding, the time almost doubles for each increase in file size, while the other two methods do not have such a steep curve, as shown in table 7.

Table 6: compression rates dataset 3

Original size (B)	LZW (%)	Huffman (%)	Arithmetic (%)
9675152	24.83	42.34	70.80
23031328	25.00	42.39	70.63
25906340	24.94	45.63	72.50
38342761	24.54	43.51	72.27
45944328	25.17	45.10	72.17

Table 7: combined time needed for compression and decompression dataset 3

Original size (B)	LZW (s)	Huffman (s)	Arithmetic (s)
9675152	9.581	12.867	15127.286
23031328	22.539	29.193	30520.398
25906340	26.838	31.632	38791.480
38342761	38.045	51.684	48729.519
45944328	50.155	58.953	76512.883

1.4.2. .tif

As for the image datasets, the patterns were highly dependent on the pixel groups of the image. For example, images containing only grayscale pixels achieved the best compression results with the arithmetic coding, while the other two datasets achieved the best results with the Huffman compression algorithm. The best results are obtained with arithmetic compression in the case of this dataset. The compression ratios are given individually since we do not have a general pattern, while the temporal results Lempel–Ziv–Welch are far better than the other two algorithms, followed by Huffman, and the worst result for time consumption is arithmetic coding. Let us now consider the individual results.

A. First dataset:

The first dataset consisted of color images up to 15 MB, and the size results were as follows: The best compression rate is achieved with Huffman in the range of 80 to 90 percent, followed by Lempel–Ziv–Welch ranging in the same range but with lower rates, and finally arithmetic coding in the range of 50 to 51 percent. As can be seen from the 8 and 9 tables, the best time for compression and decompression is by far on the side of LZW. Moreover, we can conclude from the results that the best algorithm for this dataset was Lempel–Ziv–Welch, although the overall compression ratio is lower than Huffman's due to more efficient time consumption. It is also worth noting that the compression rate did not change dramatically over this period, but we could see a pattern where the compression decreases slightly as the file size increases, as shown in Table 9.

Table 8: compression rates image dataset 1

Original size (B)	LZW (%)	Huffman (%)	Arithmetic (%)
6356928	90.18	91.22	50.45
7733184	88.03	89.29	50.35
8847296	86.27	87.74	50.30
10354688	83.94	85.63	50.22
11042816	82.87	84.68	50.21

Table 9: combined time needed for compression and decompression image dataset 1

Original size (B)	LZW (s)	Huffman (s)	Arithmetic (s)
6356928	8.610	13.123	464.758
7733184	10.595	15.122	576.795

8847296	11.929	17.947	647.519
10354688	14.381	20.957	757.654
11042816	15.450	22.986	816.332

B. Second dataset:

The second dataset consisted of color images ranging in size from 15 MB to 40 MB and yielded the following results: The highest compression rates were obtained with Huffman, although less consistently. While Huffman and LZW both ranged from 25 MB to 65 MB, image compression was highly context-dependent, while arithmetic coding had a consistent rate of about 50 percent, as seen in Table 10. Considering the time required to compress and decompress

Table 10: compression rates image dataset 2

Original size (B)	LZW (%)	Huffman (%)	Arithmetic (%)
17945404	58.40	62.88	50.24
18405661	57.32	61.94	50.27
20163380	53.33	58.42	50.39
25500362	40.94	47.59	50.58
35719284	17.58	27.51	51.22

the data, the Lempel–Ziv–Welch algorithm was also best in this case. Although the compression ratio is slightly lower than Huffman's, it is much faster, as can be seen in Table 11.

Table 11: combined time needed for compression and decompression image dataset 2

Original size (B)	LZW (s)	Huffman (s)	Arithmetic (s)
17945404	25.120	36.907	1296.596
18405661	24.639	36.028	1325.421
20163380	28.896	41.286	1445.024
25500362	35.510	52.115	1885.728
35719284	50.906	72.136	2533.473

C. Thrid dataset:

The third dataset was the largest among them and contained ten grayscale files created from the original grayscale image but with different pixel values. The goal of this dataset was to find out if the repetition of pixels could improve the compression

ratio of images. The first image had the most pixel variations and then slowly decreased. Taking this into account, we can see the following results. The best compression rates can be seen for arithmetic coding with about 77 percent, followed by Lempel–Ziv–Welch with compression rates of about 73 percent, and lastly Huffman with about 54 percent, as can be seen in the table 12. Looking at the computation time, LZW has the best time consumption of all three algorithms, followed by Huffman and lastly, arithmetic coding, as seen in table 13.

Table 12: compression rates image dataset 3

Original size (B)	LZW (%)	Huffman (%)	Arithmetic (%)
149334502	73.66	53.56	77.24
149334502	73.60	53.58	77.25
149334502	73.38	53.59	77.25
149334502	73.37	53.60	77.26
149334502	73.47	53.87	77.40
149334502	73.73	53.93	77.43
149334502	73.89	54.06	77.50
149334502	73.91	53.99	77.47
149334502	73.65	53.79	77.37
149334502	73.41	53.71	77.31

Table 13: combined time needed for compression and decompression image dataset 3

Original size (B)	LZW (s)	Huffman (s)	Arithmetic (s)
149334502	74.318	172.287	10178.523
149334502	67.645	172.536	10134.959
149334502	63.539	155.998	12190.427
149334502	65.051	164.515	9316.795
149334502	64.110	162.338	9545.117

1.5. Discussion

In this work, we aimed to test and compare standard lossless compression methods such as Huffman coding, arithmetic coding, and Lempel–Ziv–Welch coding and evaluate their results. After analyzing the compression rates and speed, including (1) training time, if necessary, (2) encoding time, and (3) decoding time. Each algorithm has its advantages and disadvantages, and we will discuss each in the sequel.

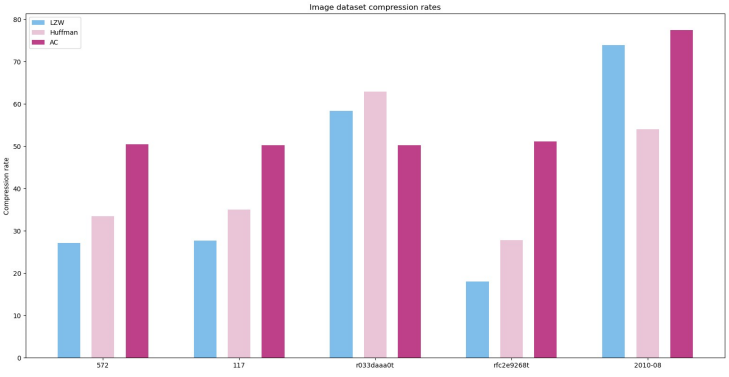

Fig. 2. Compression rate for images

Arithmetic coding has been shown to give the best result for both text and image datasets for different sizes, as shown in figures 2 and 3, but this performance is not without cost. For some large files, arithmetic coding takes up to 100 times longer than the other two algorithms, and for images, it can take even longer, and for that reason, results for arithmetic coding time shown on plots are divided by 100.

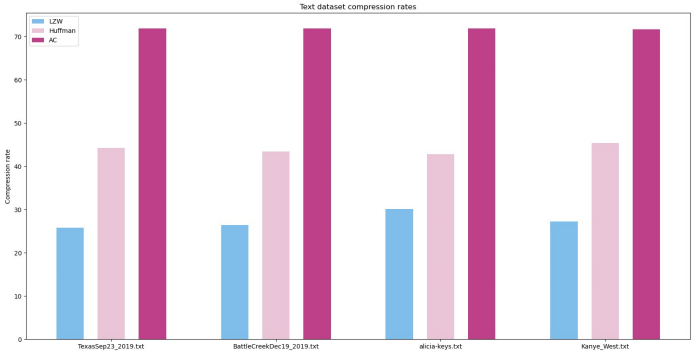

Fig. 3. Compression rate for text

This means that the high compression ratio of the arithmetic algorithm is not free.

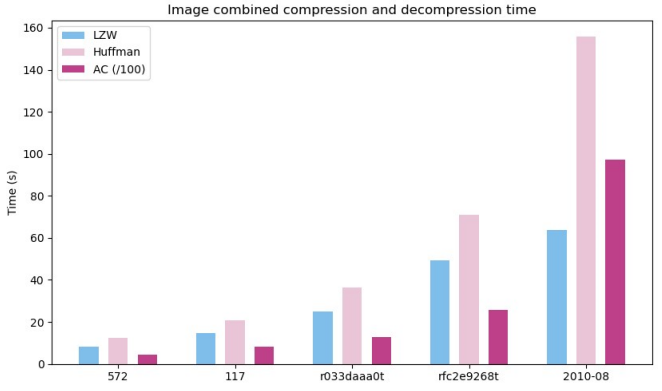

Fig. 4. Combined compression times for images

Considering this, arithmetic coding is well suited for small jobs where compression is more important than execution speed or for jobs running on small microcontrollers where the data is no larger than 100 KB, but compression is still important. Arithmetic coding is followed by Huffman coding, which offers the best of both worlds. It keeps time and processing power low while providing medium to high compression ratios, as shown in image 4. This makes it the best overall candidate and is, therefore, mostly used in compression programs such as 7zip or in dictionary-based compression. Finally, we have Lempel–Ziv–Welch with its high speed shown in figure 5 and sufficient compression ratios that make it suitable for compression where data input speed is important. LZW may be used, for example, in fax transmission, where the speed of encoding and decoding is important while saving space in the transmission bandwidth.

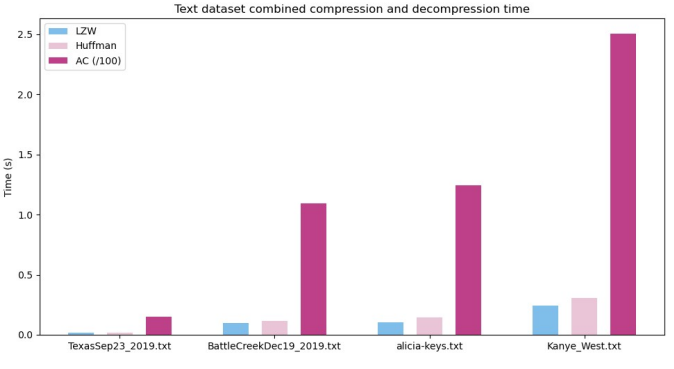

Fig. 5. Combined compression times for text

2. Conclusions

In conclusion, this work compared three lossless compression algorithms: (1) Lempel–Ziv–Welch, Huffman coding, and arithmetic coding. LZW compression provides a good balance between time and compression rates, making it a widely used algorithm. Huffman coding provides good compression rates with sufficient computation time. Arithmetic coding achieves the highest compression rates in most cases but with the slowest times. To add to what has already been said, all three compression methods are commonly used, and none is generally better than the other for all datasets. Each method has proven its superiority in some cases. For real-time data compression, LZW is best, while for compression where time is not an issue, arithmetic coding is the best choice. For the compression of images and other media, Huffman has shown that excellent compression results can be achieved with a slightly longer execution time and slightly higher computational cost, which is why it is used in programs such as ZIP, GZIP, etc. Overall, this work provided insight into the differences in using different compression methods. The choice of algorithm depends heavily on the requirements of the application. Future work could include exploring algorithms even for larger datasets and hardware implementations of these algorithms.

3. Acknowledgements

This work was supported by the EU Horizon 2020 project INNO2MARE (" Strengthening the capacity for excellence of Slovenian and Croatian innovation ecosystems to support the digital and green transitions of maritime regions") under the number 101087348, INTERREG projects Veza2 and AI.com, and University of Rijeka projects uniri-tehnic-18-17, uniri-zip2103-4-22 and uniri-tehnic-18-15.

4. References

1. Modern lossless compression techniques: Review, comparison and analysis — ieeexplore.ieee.org. https://ieeexplore.ieee.org/abstract/document/8117850. [Accessed 30-Jun-2023].
2. Senthil Shanmugasundaram and Robert Lourdusamy. A comparative study of text compression algorithms. International Journal of Wisdom Based Computing, 1(3):68–76, 2011.
3. Debra A. Lelewer and Daniel S. Hirschberg. Data compression. ACM Comput. Surv., 19(3):261–296, sep 1987.
4. SR Kodituwakku and US Amarasinghe. Comparison of lossless data compression algorithms for text data. Indian journal of computer science and engineering, 1(4):416–425, 2010.
5. Apoorv Gupta, Aman Bansal, and Vidhi Khanduja. Modern lossless compression techniques: Review, comparison and analysis. In 2017 Second International Conference on Electrical, Computer and Communication Technologies (ICECCT), pages 1–8, 2017.
6. David Salomon and Giovanni Motta. Data Compression: The Complete Reference. Springer Science & Business Media, 2007.
7. Donald E Knuth. Dynamic huffman coding. Journal of algorithms, 6(2):163–180, 1985.
8. Alistair Moffat. Huffman coding. ACM Computing Surveys (CSUR), 52(4):1–35, 2019.
9. dahuffman — pypi.org. https://pypi.org/project/dahuffman/. [Accessed 30-Jun-2023].

10. Mark R Nelson. Lzw data compression. Dr. Dobb's Journal, 14(10):29–36, 1989.
11. GitHub - joeatwork/python-lzw: LZW compression in pure python — github.com. https://github.com/joeatwork/python-lzw. [Accessed 24-Jun-2023].
12. Jorma Rissanen and Glen G Langdon. Arithmetic coding. IBM Journal of research and development, 23(2):149–162, 1979.
13. Khalid Sayood. Chapter 4 - arithmetic coding. In Khalid Sayood, editor, Introduction to Data Compression (Fifth Edition), The Morgan Kaufmann Series in Multimedia Information and Systems, pages 89–130. Morgan Kaufmann, fifth edition edition, 2018.
14. arithmetic-compressor — pypi.org. https://pypi.org/project/arithmeticcompressor/. [Accessed 24-Jun-2023].
15. os — Miscellaneous operating system interfaces — docs.python.org. https://docs.python.org/3/library/os.html. [Accessed 22-Jun-2023].
16. Preparing Your Dataset for Machine Learning: 10 Basic Techniques That Make Your Data Better — altexsoft.com. [Accessed 15-Jun-2023].