Comparison of Traditional Compression Methods of Different Datasets for Cloud Storage Systems

Klen, Deni

Master's thesis / Diplomski rad

2023

Degree Grantor / Ustanova koja je dodijelila akademski / stručni stupanj: **University of Rijeka, Faculty of Engineering / Sveučilište u Rijeci, Tehnički fakultet**

Permanent link / Trajna poveznica: https://urn.nsk.hr/urn:nbn:hr:190:611761

SVEUČILIŠTE U RIJECI
TEHNIČKI FAKULTET
Diplomski studij računarstva

Diplomski rad

Usporedba metoda kompresije različitih skupova podataka za sustave pohrane u oblaku / Comparison of Compression Methods of Different Datasets for Cloud Storage Systems

Rijeka, srpanj 2023.

Deni Klen
0069085590

SVEUČILIŠTE U RIJECI
TEHNIČKI FAKULTET
Diplomski studij računarstva

Diplomski rad

Usporedba metoda kompresije različitih skupova podataka za sustave pohrane u oblaku / Comparison of Compression Methods of Different Datasets for Cloud Storage Systems

Mentor: izv. prof. dr. sc. Jonatan Lerga

Rijeka, srpanj 2023.

Deni Klen
0069085590

SVEUČILIŠTE U RIJECI
TEHNIČKI FAKULTET
POVJERENSTVO ZA DIPLOMSKE ISPITE

Rijeka, 20. ožujka 2023.

Zavod: **Zavod za računarstvo**
Predmet: **Kodiranje i kriptografija**
Grana: **2.09.04 umjetna inteligencija**

ZADATAK ZA DIPLOMSKI RAD

Pristupnik: **Deni Klen (0069085590)**
Studij: Sveučilišni diplomski studij računarstva
Modul: Programsko inženjerstvo

Zadatak: **Usporedba tradicionalnih metoda kompresije različitih skupova podataka za sustave pohrane u oblaku / Comparison of Traditional Compression Methods of Different Datasets for Cloud Storage Systems**

Opis zadatka:

Cilj projekta je usporediti metode kompresije poput Lempel-Ziv-Welch (LZW), Huffmanovog i aritmetičkog kodiranja i primijeniti ih na različite velike skupove podataka. Potrebno je provesti validaciju temeljem različitih metrika poput vremena izvođenja i gubitka informacija u podacima ako se podaci prije i nakon kompresije razlikuju. Nadalje, potrebno je razviti model strojnog učenja temeljen na modelu auto enkodera.

Rad mora biti napisan prema Uputama za pisanje diplomskih / završnih radova koje su objavljene na mrežnim stranicama studija.

Zadatak uručen pristupniku: 20. ožujka 2023.

Mentor:

Izv. prof. dr. sc. Jonatan Lerga

Predsjednik povjerenstva za diplomski ispit:

Prof. dr. sc. Miroslav Joler

Izjava o samostalnoj izradi rada

Izjavljujem da sam samostalno izradio ovaj rad.

Rijeka, srpanj 2023.

_____

Deni Klen

Zahvala

Zahvaljujem svojem mentoru izv.prof.dr.sc. Jonatanu Lergi na prenesenom znanju, podršci, mentorstvu te poticanju samostalnog rada tijekom implementacije i izrade diplomskog rada. Također, zahvaljujem svojoj obitelji i prijateljima te kolegama i kolegicama na podršci, razumijevanju te strpljenju tijekom studiranja.

Usporedba tradicionalnih metoda kompresije različitih skupova podataka za susatave pohrane u oblaku

Deni Klen

1 Sažetak

Kako se naša ovisnost o elektroničkim medijima eksponencijalno povećava svake godine s napredovanjem digitalnog doba, tako se značajno povećava i potreba za pohranjivanjem golemih količina podataka u digitalnom dobu kojem živimo. Stvaranjem, obrađivanjem i razmjenjivanjem sve veće količine podataka, rješenja za pohranu postaju sve važnija. Ovo ne vrijedi samo za pohranu osobnih datoteka, već i za tvrtke i organizacije koje trebaju pohraniti velike količine podataka za razne svrhe, poput analize, istraživanja i razvoja. Stoga se u ovom radu posvećujemo usporedbama metoda kompresije. Kompresija je proces smanjivanja veličine podataka. Glavni cilj je postići što veći omjer kompresije, dok se istovremeno zadržavaju bitne informacije podataka. Algoritmi kompresije mogu se podijeliti u dvije vrste: (1) bez gubitaka i (2) s gubitkom. Iako su njihova imena opisna, kod kompresije bez gubitaka podataka izlazni podaci moraju biti isti kao i originalni. Kompresija s gubitcima, s druge strane, uklanja nepotrebne bitove, smanjujući veličinu datoteke. Komprimirana datoteka ne može se pretvoriti natrag u izvornu datoteku, zbog gubitka dijela informacija tijekom kompresije. Uspoređena su tri načina kompresije bez gubitaka te je implementiran autoenkoder kao način kompresije s gubitcima. Unutar metodologije pojedinačno smo objasnili svaku od metoda kompresije i njihovu implementaciju: (1) Huffmanova metoda temelji se na frekvenciji ulaznih parametara te postiže efektivnu kompresiju zamjenom originalnih znakova s novo generiranim kodovima. Znakovi češćeg ponavljanja zamijenjeni su kraćim kodom dok su rjeđe ponavljajući znakovi reprezentirani dužim kodom. (2) Lempel-Ziv-Welch radi temeljem izgradnje rječnika u koji se iterativno dodaju znakovi. Kompresija se ostvaruje smanjenjem duljine koda za česte uzorke znakova. (3) Aritmetičko kodiranje temelji se na izračunu matematičke vjerojatnosti za kodiranje ulaznih simbola, svaki se simbol zamjenjuje brojčanom vrijednošću koja predstavlja vjerojatnost pojavljivanja tog simbola. Također, opisan je i autoenkoder kao vrsta neuronske mreže koji se sastoji od dva glavna dijela enkodera i dekodera. Enkoder je odgovoran za smanjivanje dimenzije ulaznih podataka, dok je dekoder odgovoran za vraćanje dimenzije podataka što bliže

originalnom. Autoenkoderi se mogu primjenjivati u različite svrhe, poput rekonstrukcije podataka, izvlačenje značajki te generiranje novih podataka. Unutar ovog rada autoenkoder je korišten kao kompresijski autoenkoder. U sljedećem poglavlju opisani su korišteni setovi podataka te način evaluiranja dobivenih rezultata. Setovi su dohvaćeni sa stranice Kaggle, te su ručno evaluirani i pripremljeni kako bi testiranje obuhvatilo što veće razlike unutar ispitivanja. Evaluacija je odrađena ručno usporedbom vremena potrebnih za kompresiju analiziranih podataka, dekompresiju i potrebno vrijeme učenja i usporedbom omjera postotka kompresije. Slijedi prikaz rezultata gdje je moguće vidjeti i zaključiti sljedeće. Svaka metoda kompresije ima svoje prednosti i nedostatke ovisno o setu podataka i načinu korištenja. Kod kompresije malih setova, najbolji rezultat ostvaruje aritmetičko kodiranje s visokim postotkom kompresije od 70 posto i prihvatljivim troškom računanja. Ako je potrebna velika brzina izvođenja, ostale dvije metode pokazale su se kao bolje rješenje. LZW algoritam pokazao se kao najbrže rješenje s nešto lošijom kompresijom, ali je stoga primjenjiv u kompresiji u realnom vremenu, dok se Huffmanov algoritam pokazao kao nisko troškovni, s dobrom kompresijom, odlično primjenjiv u datotekama poput ZIP ili GZIP kompresije. Kod autoenkodera je prikazano da rezultati uvelike ovise o količini treniranja i načinu implementacije autoenkodera. Prilikom korištenja više slojeva ili veće gustoće slojeva rezultati će sadržavati manje gubitke i obrnuto. Zaključno, prikazano je da izbor algoritma uvelike ovisi o zahtjevima aplikacije. Budući rad mogao bi uključivati istraživanje algoritama čak i za veće skupove podataka, hardverske implementacije ovih algoritama ili implementacije za različite slučajeve upotrebe.

Comparison of Compression Methods on Different Data Sets for Cloud Storage Systems

Deni Klen*, Jonatan Lerga*†
* Department of Computer Engineering, Faculty of Engineering, University of Rijeka, Rijeka, Croatia
† Center for Artificial Intelligence and Cybersecurity, University of Rijeka, Rijeka, Croatia
jlerga@riteh.hr

***Abstract*—The paper provides a comparison of compression methods such as Lempel-Ziv-Welch, Huffman, and arithmetic coding applied to different large datasets. Validation is done based on various metrics such as execution time, compression ratio, and information loss in the data when the data differs before and after compression. A machine learning model is also developed based on an autoencoder model. LZW produced results of about 30 percent median compression ratio for all text records and a median of 70 percent for image records. In addition, Huffman coding produced a compression rate of about 40 percent median for text data and a median of 55 percent for image data. Finally, arithmetic coding yielded results of 70 percent median for text compression and 55 percent median for image data compression. The time required was lowest for LZW, followed by Huffman, and worst for arithmetic coding. In addition, autoencoders depend heavily on the encoder and decoder settings, with the best results obtained with a loss of up to 0.0093 per 100 pixels for 8000000 density level setup and 0.094 for 300000 density levels setup.**

I. 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. As more and more data is created, processed and exchanged digitally, storage solutions become increasingly important. This is not only true for personal file storage, but also for businesses and organizations that need to store large amounts of data for various purposes such as analysis, research and development[2]. Therefore, it is crucial to invest in efficient data compression techniques 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 reduced. However, from an information-theoretic 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 more or less 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. Lossy compression, on the other hand, 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 of the information is lost during compression [5]. In this paper, we briefly explain the operation of some well-known lossless compression algorithms such as Lempel-Ziv-Welch, Huffman [6] and arithmetic coding. Then, they are compared, based on their compression ratio, compression and decompression speed, and the number of errors and mistakes. Moreover, a machine learning model based on auto-encoders is developed to use decided algorithm to compress the data with best compression results and as low as possible loss of data while maintaining low cost. The rest of the paper is organized 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.

II. METHODOLOGY

In this section, we briefly review selected compression algorithms used in our analysis and explain our autoencoder model.

A. 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. Although the Huffman algorithm provides good compression rates in most cases, each symbol must be represented with at least one binary code, so it requires more space than some other algorithms.

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, the 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

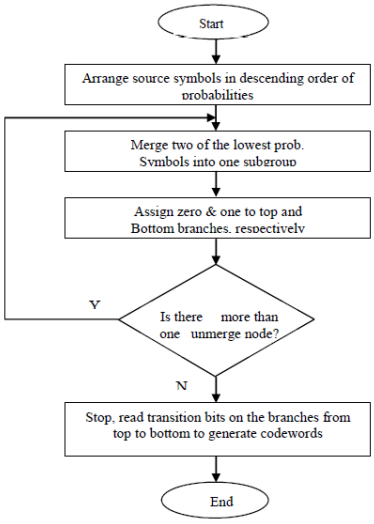

Fig. 1. Flowchart of Huffman algorithm taken from [8]

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.

1) 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 followed for encoding decoding, and training set

```
from dahuffman import HuffmanCodec
codec = HuffmanCodec.from_data \
    (train_file)
encoded_data = codec.encode(input_file)
decoded_data = codec.decode(encoded_file)
```

B. 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 or 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 backwards for the codes in the dictionary.

1) Implementation: To implement and compress or decompress files with the Lempel–Ziv–Welch algorithm library *lzw-python* [11] is needed. Implementation is as followed

```
import lzw
encoded_data = lzw.compress(f)
decoded_data = lzw.decompress
    (encoded_file)
```

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

C. 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].

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

```
from arithmetic_compressor
    import AECompressor
from arithmetic_compressor.models
    import MultiPPM, StaticModel
model = MultiPPM(all_chars, models=3)
codec_ae = AECompressor(model)
compressed = coder.compress(data)
decoded = coder.decompress(data,
    original_length)
```

D. Autoencoders

Autoencoders are a special type of neural network that are trained to transfer given inputs to their outputs without explicit supervision. As such, they have found applications in areas such as image recognition, natural language processing, and anomaly detection [15]. In addition, autoencoding is a data compression algorithm in which the compression and

decompression functions are learned automatically, are lossy, and are data specific. Data specific means that they can only work with the data for which they have been trained. Lossy means that the output is degraded compared to the original, and automatically learned means that it is easy to train them. Autoencoders have the advantage of being able to learn data representations that capture the most important features and patterns of the input data. This enables efficient data compression and reconstruction, making autoencoders a valuable tool for tasks such as data dimensionality reduction, data denoising, and generating synthetic data [16]. Their ability to adapt to the specific characteristics of the data on which they are trained also makes them highly flexible and applicable to a wide range of domains and problem types. The architecture of an autoencoder consists of the original input, the encoder, the compressed representation, the decoder, and the reconstructed input[17].

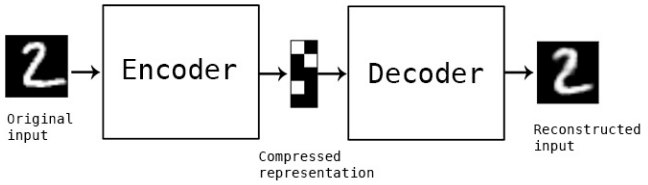

Fig. 2. Autoencoder shema

Encoder and decoder are parametric functions (usually neural networks) that can be optimized to work best in our case. The encoder's job is to accept the original data, which may have two or more dimensions, and create a single 1D vector. The encoder progressively reduces the dimensionality of the input by passing the data through a series of layers. The decoder receives the encoded representation and is tasked with reconstructing the original image with the highest possible quality from a single 1D vector using the data learned from the compressed representation. During training, the autoencoder tries to minimize the reconstruction error. To achieve this, the autoencoder learns important features of the data.

1) Implementation: Our autoencoder will be based on the Keras library [17]. For successful creation of an encoder and decoder, we need to have the same output size as the input size, so based on the database prepared for the encoder, the shape size will be set to 4096 (64x64) and 4 layers: a dense layer with 500 neurons, LeakyReLU [1], another dense layer with 4 neurons and another LeakyReLU layer.

```
x = tensorflow.keras.layers.
    Input(shape=(4096),
        name="encoder_input")

encoder_dense_layer1 = tensorflow.keras.
    layers.Dense(units=500,
        name="encoder_dense_1")(x)

encoder_activ_layer1 = tensorflow.keras.
    layers.LeakyReLU(name=
        "encoder_leakyrelu_1")
    (encoder_dense_layer1)

encoder_dense_layer2 = tensorflow.keras.
    layers.Dense(units=4,
        name="encoder_dense_2")
    (encoder_activ_layer1)
encoder_output = tensorflow.keras.
    layers.LeakyReLU
    (name="encoder_output")
    (encoder_dense_layer2)
```

The decoder must be similar to the encoder but reversed. So first we need to take the 4-element layer and convert it to the 500-element layer and then to the full size. The finished code looks like this.

```
decoder_input = tensorflow.keras.layers.
    Input(shape=(4),
        name="decoder_input")

decoder_dense_layer1 = tensorflow.keras.
    layers.Dense(units=500,
        name="decoder_dense_1")
    (decoder_input)
decoder_activ_layer1 = tensorflow.keras.
    layers.LeakyReLU(
        name="decoder_leakyrelu_1")
    (decoder_dense_layer1)

decoder_dense_layer2 = tensorflow.keras.
    layers.Dense(units=4096,
        name="decoder_dense_2")
    (decoder_activ_layer1)
decoder_output = tensorflow.keras.
    layers.LeakyReLU(
        name="decoder_output")
    (decoder_dense_layer2)
```

When both the decoder and encoder are set and ready, we need to implement them inside the autoencoder.

```
ae_input = tensorflow.keras.layers.
    Input(shape=(4096), name="AE_input")
ae_encoder_output = encoder(ae_input)
ae_decoder_output = decoder
    (ae_encoder_output)
```

Training and prediction are performed on the data sets discussed in the next chapter, and the evaluation of the results in the chapter after next.

E. Other methods

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

[1]leakyReLU - Leaky version of a Rectified Linear Unit. It allows a small gradient when the unit is not active

the encoding and decoding process. This time does not include the time it takes to read or write. We have also separated out the time needed for training so that we can include that time as well. Once all that is done, we need a function to compare original files and compressed files. This is accomplished with the Python library *os*[18].

III. CASE STUDY

In this section, we will discuss the datasets used in this work and evaluate the thoughts and processes.

A. Data Sets

For successful implementation and evaluation of the results, we need to prepare data sets [19]. We decided to divide the datasets into two main groups: (1) compression datasets, (2) autoencoder datasets. 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. The second large dataset is used for training and evaluating auto-encoders. This dataset is divided into three groups. Training, testing, and validation datasets. The training dataset accounts for 80 % of the total dataset, while the test and validation datasets each contain 10 %. All datasets were found and downloaded from *kaggle* [20]. The second dataset we used is the MINST dataset created by Keras [21], which contains 60000 28x28 grayscale images divided into training and test groups. We then processed them manually and selected the most suitable options for our work.

B. 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. To evaluate the work of auto-encoders, we want to save time for training an epoch and evaluate the difference between the pixels in the original image and the final image. The evaluation is done by changing the learning parameters of the autoencoder and the layers that form the encoder and decoder. Also, the number of epochs is changed. We also want to evaluate the loss of the predicted data from epoch to epoch. The results of the evaluated work will be discussed in the next chapters.

TABLE I
SPECIFICATION OF THE PLATFORM THAT HAS BEEN USED FOR THE EXECUTION OF THE PROGRAMS

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

IV. RESULTS

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

A. .txt

First of all, 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. Last but not least, 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:

1) 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 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 II. The time cost, on the other hand, 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 III.

TABLE II
COMPRESSION RATES DATASET 1

Original size	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

TABLE III
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

2) 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 that contain 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 IV, 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 size, as shown in table V, 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.

TABLE IV
COMPRESSION RATES DATASET 2

Original size	LZW	Huffman	Arithmetic
77505	30.99	42.68	71.76
113457	30.17	42.85	71.83
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

3) 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 VI table, but the time consumption here is worth mentioning. As we can see with arithmetic coding, the time almost doubles for each increase in file size, while the other two methods don't have such a steep curve, as shown in table VII.

TABLE V
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	0.323	250.621
322587	0.312	0.419	422.772

TABLE VI
COMPRESSION RATES DATASET 3

Original size	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 VII
COMBINED TIME NEEDED FOR COMPRESSION AND DECOMPRESSION DATASET 3

Original size	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

B. .tif

As for the image datasets, the patterns were indeed highly dependent on the pixel groups of the particular 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 case of this dataset. The compression ratios are given individually since we do not have a general pattern, while for the temporal results Lempel–Ziv–Welch is 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.

1) First dataset: The first dataset consisted of colour 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 VIII and IX 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 better time consumption. It is also worth noting that the compression rate did not change dramatically over this period, but we could see

TABLE VIII
COMPRESSION RATES IMAGE DATASET 1

Original size	LZW	Huffman	Arithmetic
6356928	90.18	91.22	50.45
7733184	88.03	89.29	50.35
8847296	86.27	87.74	50.30
9633729	85.07	86.64	50.25
10354688	83.94	85.63	50.22
11042816	82.87	84.68	50.21

a pattern where the compression decreases slightly as the file size increases, as shown in Table IX.

TABLE IX
COMBINED TIME NEEDED FOR COMPRESSION AND DECOMPRESSION IMAGE DATASET 2

Original size	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	648.519
9633729	13.354	19.494	709.357
10354688	14.381	20.957	757.654
11042816	15.450	22.986	816.332

2) 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 not as 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 X. Considering the time required to compress and

TABLE X
COMPRESSION RATES IMAGE DATASET 2

Original size	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

decompress 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 XI.

TABLE XI
COMBINED TIME NEEDED FOR COMPRESSION AND DECOMPRESSION IMAGE DATASET 2

Original size	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

3) Thrid dataset: The third dataset was the largest among them and contained 10 grayscale files created from the original grayscale image, but with different pixels. 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 lasty Huffman with about 54 percent, as can be seen in the table XII. Looking at time, LZW has the best time consumption of all three algorithms, followed by Huffman and lastly arithmetic coding, as seen in table XIII.

TABLE XII
COMPRESSION RATES IMAGE DATASET 3

Original file size	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 XIII
COMBINED TIME NEEDED FOR COMPRESSION AND DECOMPRESSION IMAGE DATASET 3

Original file size	**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
149334502	63.832	161.969	9753.392
149334502	65.085	165.141	13631.715
149334502	70.668	168.742	9803.332
149334502	74.851	177.857	9694.130
149334502	66.087	162.031	10059.035

C. Autoencoder

Autoencoder compression was evaluated with two datasets: (1) the MNIST Keras dataset and (2) our own prepared dataset containing the 64 x 64 color images. Both datasets were divided into a training dataset and a test dataset. Then, the performance of the coders and decoders was trained and evaluated. The coding model consisted of four layers whose density, quality and speed of compression were evaluated. When a higher density was used, the results were closer to the original. It can also be seen that when 200 and 2 density layers were used, there were 157402 trainable parameters compared to the 402520 trainable parameters in Figure 3 for parameters 500 and 20. The decoder is a reciprocal representation of the encoder model and consists of four equal layers, but in reverse order, so that we can convert a low-dimensional output into a high-dimensional one, where the density layer must be the same as the encoding layer. According to the two cases above,

```
Model: "encoder_model"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 encoder_input (InputLayer)  [(None, 784)]             0

 encoder_dense_1 (Dense)     (None, 500)               392500

 encoder_leakyrelu_1 (LeakyR  (None, 500)              0
 eLU)

 encoder_dense_2 (Dense)     (None, 20)                10020

 encoder_output (LeakyReLU)  (None, 20)                0

=================================================================
Total params: 402,520
Trainable params: 402,520
Non-trainable params: 0
```

Fig. 3. Autoencoder model

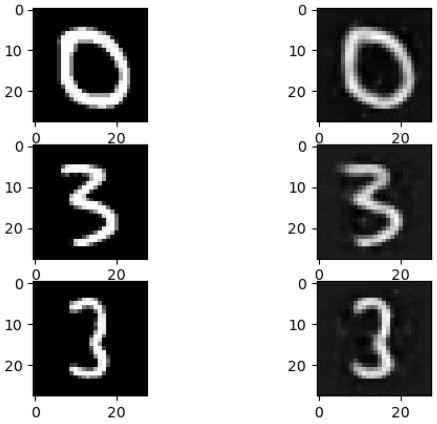

Fig. 4. Autoencoder results for higher density

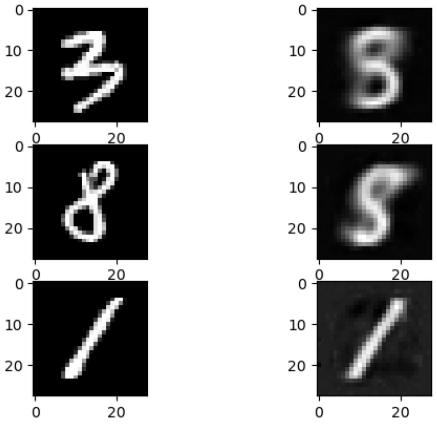

Fig. 5. Autoencoder results for lower density

the decoder has 158184 transferable parameters for the first case and 403284 trainable parameters for the second case. Merging the encoder and decoder results in an autoencoder with just over 300000 trainable parameters for the first case and over 800000 parameters for the second case. We proceed with training the data sets. To train the autoencoder, an epoch number is needed. The higher the epoch number, the lower the image loss. An epoch number of 10 leads to completely unpredictable results because not enough training was done, while a higher epoch number leads to results that are closer to the original image. Now for the numbers. For density layers 500 and 20, we evaluated the results of 3 epochs: (1) epoch 30, where the final prediction was a loss of 0.0102 and an image difference of 0.0093; (2) epoch 100, where the initial loss was 0.0285, the final loss was 0.089, and the actual loss was 0.0083; (3) epoch 250, where the initial loss was predicted to be 0.0293. Examination of the results shows that the loss prediction for the layer slows down from 200 epochs and decreases only by 0.0001 from 200 to 250 epochs. Although the best compression and decompression results are obtained with this high number of epochs, as can be seen in Figure 4. The training time for each epoch at this setting is 3 to 4 seconds, and the time needed to compress and decompress the entire image set is around 2 seconds per image. Below we tested with density layers 200 and 2 with the same 3 epochs and the results were as expected, so we will focus only on epoch 250 to describe it in detail. The compression loss was 5 times greater than with the higher density layers, as you can see in the image 5.

For our own dataset, the results were similar. Although the start and target planes were 64x64 and 4096 pixels respectively, we were able to use an even higher density. This is exactly what we were aiming for. The learning step from one epoch to the next was about 0.0003, and the time required for one epoch was about 10 seconds. The actual compression loss was about 0.094 and the calculated one was about 0.087. In summary, the preference of our autoencoder depends heavily on what data and what types of layers and algorithms are used to encode and decode the data. More layers do not always mean a better result. Although compression of about 3 to 4 percent with autoencoders is not as valuable as other types of compression, it can be useful and perhaps improved in the future.

V. 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 to implement an autoencoder method and evaluate its results. After analyzing the compression rates and speed, including: (1) training time, if necessary, (2) encoding time, and (3) decoding time. Each of the algorithms has its advantages and disadvantages and we will discuss each of them in separate sections.

a) text & images: Arithmetic coding has been shown to give the best result for both text and image datasets for different sizes as shown in pictures 6 and 7, but this performance is not without cost. For some large files, arithmetic

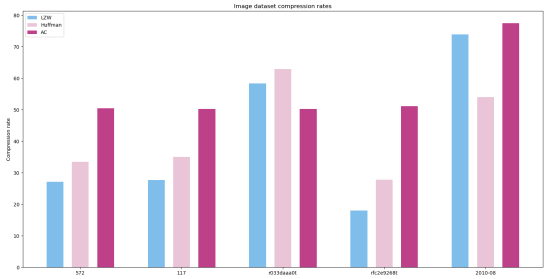

Fig. 6. Compression rate for images

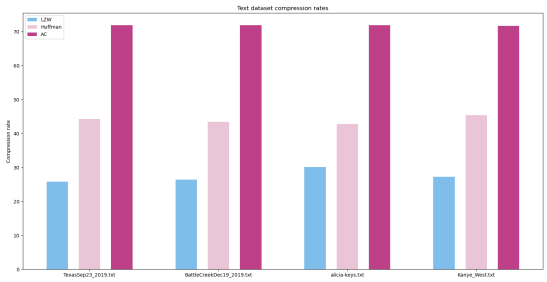

Fig. 7. Compression rate for text

coding takes up to 100 times longer than 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. This means that the high compression ratio of the

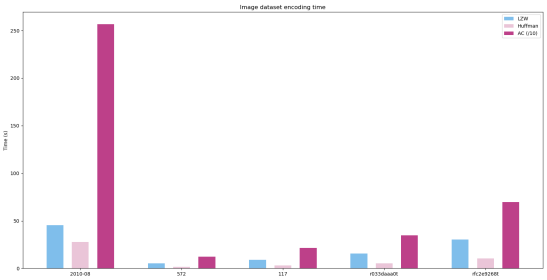

Fig. 8. Encoding times for images

arithmetic algorithm is not free. 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 images 8 and 9. This makes it the best overall candidate and is therefore mostly used in compression programs such as 7zip or in dictionary-based compression. Last but not least, we have Lempel–Ziv–Welch

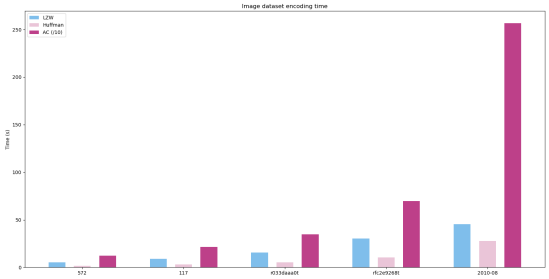

Fig. 9. Decoding times for images

with its high speed shown at pictures 10 and 11 and sufficient compression ratios that make it suitable for compression where data input speed is important. LZW is used, for example, in fax transmission where the speed of encoding and decoding is important while saving space in the transmission bandwidth.

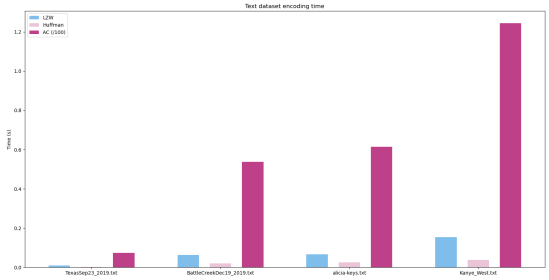

Fig. 10. Encoding times for text

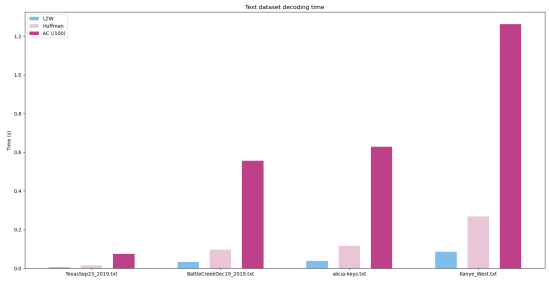

Fig. 11. Decoding times for text

b) Autoencoder: Autoencoders, by their nature, can be used in a variety of domains, such as anomaly detection, noise reduction, image restoration, feature learning, and generative modeling. In this work, however, the focus is on image compression. They are used for lossy image compression. The evaluation of the results shows that they have significantly reduced the dimensionality of the image, which leads to compression. The good execution time and fast learning time make them useful in some cases. In addition, the compu-

tational efficiency of the autoencoder was remarkable as it provided fast encoding and decoding of the images. This makes it a suitable approach for real-time image compression applications where both storage space and processing time are critical factors. Although there are already compression techniques that have a higher compression ratio with less data loss, autoencoders should be used in some other cases for the time being.

VI. Conclusion

In conclusion, this work compared three lossless compression algorithms: (1) Lempel–Ziv–Welch, Huffman coding, and arithmetic coding, and the implementation of a lossy autoencoder whose purpose is data compression. 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. In addition to these traditional compression methods, this study also introduced the concept of autoencoders. Autoencoders are neural network based models that provide a different approach to compression. They are applicable in scenarios where unsupervised learning, dimensionality reduction, and generative modelling are beneficial. 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, hardware implementations of these algorithms, or implementing different use cases of autoencoders.

Acknowledgment

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, and University of Rijeka projects uniri-tehnic-18-17 and uniri-tehnic-18-15.

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/arithmetic-compressor/. [Accessed 24-Jun-2023].

[15] Introduction to autoencoders. — jeremyjordan.me. https://www.jeremyjordan.me/autoencoders/. [Accessed 23-Jun-2023].

[16] Joseph Rocca. Understanding Variational Autoencoders (VAEs) — towardsdatascience.com. https://towardsdatascience.com/understanding-variational-autoencoders-vaes-f70510919f73. [Accessed 11-Jun-2023].

[17] Building Autoencoders in Keras — blog.keras.io. https://blog.keras.io/building-autoencoders-in-keras.html. [Accessed 24-Jun-2023].

[18] os — Miscellaneous operating system interfaces — docs.python.org. https://docs.python.org/3/library/os.html. [Accessed 22-Jun-2023].

[19] Preparing Your Dataset for Machine Learning: 10 Basic Techniques That Make Your Data Better — altexsoft.com. [Accessed 15-Jun-2023].

[20] Kaggle: Your Machine Learning and Data Science Community — kaggle.com. https://www.kaggle.com/. [Accessed 24-Jun-2023].

[21] Keras Team. Keras documentation: MNIST digits classification dataset — keras.io. https://keras.io/api/datasets/mnist/. [Accessed 10-Jun-2023].