Reducing storage size of large quantitative data using a combined lossy compression method
An application to an IoT dataset

Nguyen, Linh Diep

Award date:
2022

Department of Mathematics and Computer Science

Reducing storage size of large quantitative data using a combined lossy compression method - An application to an IoT dataset

Bachelor End Project Report

Linh Diep Nguyen

Supervisor:
Erik Quaeghebeur

June - 2022

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

In this digitalized world, the amount of data produced, stored, and processed everyday is massive, and it keeps soaring and transforming human life. According to Statista Research Department in 2022, the total amount of data generated was predicted to grow constantly, passing 64.2 zettabytes in 2020 and to more than 180 zettabytes up to 2025 [51]. As estimated, 90% of the world data was created over the last two years [39]. With explosive technological advancements, an increasing number of electronic devices including smart phones, computers, sensors, the Internet and digital communications has been used extensively in human life. This has led to the vast data generated from all kinds of sources. The amount of data and the generated frequency are so intensive that they are usually referred to as Big Data [25].

As a result, the installed infrastructure of storage capacity also needs to grow to meet the demanding storage of the big data volume generated. However, the storage capacity could not increase as much as the volume of data generated, which leads to a serious shortage of the demanded storage capacity. In the year 2013, only 33% of the digital data generated could be stored with the available storage capacity, and it could just hold less than 15% of the data generated by 2020 [30]. According to a calculation, a minimum of over 6 ZBs of critical data, which was approximately double all the data generated in 2013, was expected to create the data-capacity gap in 2020 [13]. Concerning this, reducing the data needed to be stored can lead to much less demand for storage capacity and other related resources required to maintain the database infrastructure. As regards, the size of a dataset is important in terms of storage and transmission. In particular, it has an impact on storage

and transmission costs, as well as the usability of dataset, and how long it will be maintained.

Quantitative study is a popular field that concentrates on quantifying the data collection and analysis [15]. This is often concerned with quantitative data, which are numerical values, such as statistics, percentages, etc [26]. In many fields of natural science, such as physics, chemistry, engineering, there are also a lot of quantitative data generated for measurements. This suggests that focusing on quantitative data to reduce the dataset size might lead to large space savings.

Therefore, the general topic in this research project is to reduce the storage size of large quantitative data using some concepts of significant information.

1.2 Research Topic

Given the explosive growth of total data needed to be stored and transmitted, it requires a large number of bytes to represent these data in digital form. For instance, a single second of video would take more than 20 megabytes, or 160 megabits to represent digitally without compression [49]. Typically, the process of reducing the data size is often achieved by data compression. Data compression is a technique to encode data in a compact representation by detecting and utilizing a data representation that uses fewer bits than the original representation [38]. This process is useful since it minimizes the amount of required data storage and transmission resources, especially for big data streams in text, image, speech, audio and video data.

In particular, data compression can either be lossless or lossy. Lossless compression results in no loss of information and thus, the original data can be precisely reconstructed from the compressed data. Such compression is necessary when there can be no differences between the recovered and original data. As an example, compressing text data must result in identical meanings after the reconstruction since a small variation in the text can cause inconsistent interpretations of the meaning. In contrast, lossy compression allows some loss of information, but the compressed data cannot be recovered completely. In return, this type of compression leads to much larger compression rates compared to lossless compression [49].

Nevertheless, the data compression and decompression operations require computational power. As a result, designing data compression processes

needs to take into consideration the optimal balance between various aspects, such as the degree of compression, the amount of distortion from lossy compression processes, and the computing power consumption [52].

In quantitative studies, standard (floating-point) number formats are commonly used to store the attibutes. However, there are many more digits than necessary in many cases. Too many digits may overcomplicate the problem, make the data less intuitive, and mislead the data users. This will also lead to wasteful resources for storing and processing those data. As an example, storing the p-value of 0.53741219349382582 does not give more informative insights than rounding to less decimal digits, e.g. $p = 0.54$. In this example, storing a lot of decimal digits will confuse the interpretation of the result in the hypothesis testing. Concerning this, there are some rules and guides regarding rounding off decimal digits, which introduces a process of lossy data compression [18]. Consequently, lossy compression, which involves dropping insignificant digits, can lead to a large reduction in the size of a quantitative dataset.

Besides, the storage format of data file can largely affect the storage size. A file format represents the structure of a file that specifies how information is encoded to store in a computer system. There can be many different ways to store the data in a file. For instance, tabular data can be stored in a CSV (Comma-separated values) file, TSV (Tab-separated values) file, or db (databank) file format. Since different saving formats have different structure of storing data, each storage format will lead to different storage size. In addition, storage format is critical concerning efficiency, particularly with respect to saving time, loading time, processing speed and transferring time. However, there is not a universal standard file storage format that can efficiently store all different types of information. Alternatively, it should be considered explicitly for each case on which formats to store data.

To sum up, there are different ways to store and compress data from the current literature [47]. However, it is not clear how this is combined to decrease the storage size of large quantitative data in a practical setting. Therefore, this research will focus on reducing storage size of a large quantitative dataset using a combined lossy compression method.

1.3 Research Questions

As discussed above, it is not necessary to store all the excessive precision of the numerical records when considering a quantitative dataset. Next to that, the data storage format can also significantly affect the storage size, loading and processing time. Thus, the storage size of large quantitative data can be reduced by investigating various methods of lossy compression considering the decimal length and data storage format. In consequence, the study goal is to introduce a combined lossy compression technique for large quantitative data. Specifically, this research attempts to address the following research questions:

1. To what extent does dropping decimal digits affect the data storage size?

2. How much can data storage size be reduced by transforming the file storage format?

3. How much can data storage size be reduced by using data compression?

Chapter 2

Literature Review

As discussed in chapter 1, data compression is an efficient technique to utilize a compact data presentation by removing repetitive and redundant data elements. Consequently, this helps to minimize the storage capacity, reduce database cost and speed up data transmission process. In this chapter, the fundamental compression coding schemes that are relevant for quantitative data will be reviewed for lossless compression and lossy compression. The discussed compression techniques are presented in the following sections.

2.1 Lossless Compression Techniques

2.1.1 Huffman coding

In 1952, David Huffman proposed a fundamental coding technique that can efficiently compress data in various formats [32]. Huffman coding is an entropy based compression technique. The basic idea is that it encodes the input symbols so that the length of the encoding is based on the occurrence frequency of the corresponding symbol [54]. In consequence, the algorithm outputs a variable-length code table to represent a source character. The key property of this algorithm is that it uses a specific scheme to choose the representation for each symbol, which results in an optimal prefix code. This minimum-redundancy prefix code is uniquely decoded and consists of two parts, the first one is to construct a Huffman tree from the input sequence, and the second one is to traverse the tree to encode the symbols.

There have been extensive studies on the Huffman coding theory's analysis, implementation and developments with a wide range of applications [29, 24, 56, 10]. As regards, other different variations are available, namely

Length-constrained Huffman coding, Canonical Huffman coding, Adaptive Huffman coding, Rice code, Tunstall code and Golomb code [34]. Some other famous compression methods also use a Huffman code as a critical step, such as s ZIP, ZSTD, BLOSC, JPEG, MPEG, MP3, SZ and MGARD. For some compression applications, Huffman coding is preferred because of its simplicity, speed and effectiveness [14]. Although the algorithm was invented many years ago, it is still a major development of efficient data representation method [40].

2.1.2 Dictionary-based coding

Dictionary-based encoding do not use a statistical model, or variable-length codes as for Huffman encoding. Instead, it assigns each variable-length string of symbols as a single token that forms an index to a phrase dictionary [48]. The input can belong to the group of frequently occurring patterns, which are stored in the dictionary, or infrequently occurring patterns, which will be encoded less efficiently [49]. The dictionary contains strings of symbols, with two available types of static (permanent) or dynamic (adaptive). The static type determines a complete set of strings before encoding and does not allow modifications during the procedure, while the dynamic type allows both insertion and deletion of strings as new input is read. More popular techniques have the dictionary content changing during the coding procedure depending on the preliminary state.

Among these techniques, Lempel–Ziv algorithm (LZ) is well-know for lossless compression, due to its applications for different file formats. Two versions of this algorithm were introduced by Lempel and Ziv in 1977 and 1978, which are LZ77 [60] and LZ78 [61]. LZ77 uses a circular buffer as a "sliding window" to iterate sequentially through the input and search for matches in the past N-byte substring as dictionary entries. On the other hand, LZ78 uses a more specific dictionary structure that appends strings in the encoding process. After that, in 1984, Terry Welch introduced an enhanced compression technique of the previous LZ77 and LZ78, namely Lempel–Ziv-Welch (LZW) [57]. The algorithm builds a dynamic dictionary and index for encoding and decoding data during the compression process, similar to LZ78. However, the dictionary is initialized as single-character strings representing all possible symbols. LZ77 is applied for GZIP and ZIP, while LZW is applied for Unix file compression utility compress, PNG and GIF image format.

2.2 Lossy Compression Techniques

Lossy compression techniques represent data using approximations and partial data deletion, which allows some loss of information. This leads to a higher compression ratios than lossless compression, but the original data cannot be reconstructed exactly [49]. For example, data that are not sensitive to human perception are discarded. Thus, it is most popular for compressing multimedia, including sound, images, or videos [59].

Most of the techniques for image data are derived directly for scientific data, including waveform transformation, coefficient dominance, and vector quantization [27, 35]. Since these methods are usually created and optimized concerning human perception, lossy compression algorithms may be suitable for visualization, but there might be not sufficient control for the compression errors [16]. As a consequence, new lossy compression techniques for scientific data are introduced, built on data prediction (SZ, FPZIP [20, 53, 36]), block transforms (ZFP [37]), or multi-grid approach (MGARD [1, 2, 3, 4]).

Although lossy compression can save up more storage space, the discarded information is not retrievable after the compression [33]. Thus, the data file is limited to further changes to avoid any more information lost. As regards, lossy compression should not be implemented for the same data file many times, otherwise it will generate inherent errors in the data. In consequence, this can cause potential limitations in the data transfer process, especially when the data are accessed and modified by various users. Next to that, lossy compression sometimes requires more expensive computational power [17]. Another disadvantage is the decreasing precision of the data with higher ratio of compression [43]. Although high compression ratio is desirable when comparing different compression algorithms, this may cause great issues to the quality of the quantitative data in this research. Besides, it is usually difficult for the users to control the errors caused by approximation and prediction of the lossy compression techniques.

Considering the trade-off between the storage reduction and data precision for quantitative data, this research aims to introduce a combined lossy compression method that the users can control the loss of information in the data.

Chapter 3

Methodology

3.1 Context Understanding

Recently, the Internet of Things (IoT) has become an emerging phenomenon. It refers to a system of physical devices that are integrated into the information network, which can connect, communicate, interact, query their state and any associated information, as well as offer services through the Internet or other communications networks [28]. Through this Internet-based information architecture, everyday objects, such as household devices as well as more advanced computation and information services, will be able to communicate and interact [45]. With the increasing development of technology, the market demand for IoT application devices has increased greatly. There are a wide range of smart technology devices that consumers can use to monitor and automate their homes and surrounded environments [5]. The IoT networks can have various applications in different fields, including consumer, commercial, industrial, and infrastructure spaces [44].

However, every "object" in the IoT can create a large amount of data with different types of information. It is estimated that the total amount of data produced by IoT systems has passed one zettabyte over the last few years [55]. Considering these characteristics, this type of data, captured by sensor or Radio Frequency Identification system in IoT networks, has been described as a kind of "Big Data" [12]. Hence, all these generated data can cause technical issues and challenges on the data management and processing.

Therefore, the objective of this research is to reduce the data storage size considering data generated from an IoT system.

3.2 Dataset

This research considers the IoT dataset for Intrusion Detection Systems. Specifically, it is called the Malicious BotNet dataset BotNet_IoT, which contained all of the IoT device data files gathered during the detection of IoT Botnet attacks on a cyber security system [7, 9]. This dataset is publicly accessible on Kaggle [6]. The data are captured from nine IoT devices using Wireshark software in a local network. The network uses a central switch with 23 statistical features that are obtained from a PCAP (Packet Capture) file used for analyzing networks. The PCAP file contains data packets from the network, which is used to manage the network characteristics, control network traffic and check network status.

Dataset	Features	
	Type	Names
BotNet_IoT	Temporal-statistical/time-related	MI_dir_L#_weight, MI_dir_L#_mean, MI_dir_L#_variance, H_L#_weight, H_L#_mean, H_L#_variance, HH_L#_weight, HH_L#_mean, HH_L#_std, HH_L#_magnitude, HH_L#_radius, HH_L#_covariance, HH_L#_pcc, HH_jit_L#_weight, HH_jit_L#_mean, HH_jit_L#_variance, HpHp_L#_weight, HpHp_L#_mean, HpHp_L#_std, HpHp_L#_magnitude, HpHp_L#_radius, HpHp_L#_covariance, HpHp_L#_pccAll

Figure 3.1: BotNet_IoT dataset features

The packet count, jitter, size of outbound packets, combined size of outbound and incoming packets were extracted from the PCAP file and some statistical measures were calculated for these features, creating 23 total features. These

features are shown in Figure 3.2. The seven statistical measures are mean, variance, count, magnitude, radius, covariance, correlation coefficient. This was calculated over a time window of 10 seconds using a decay factor of 0.1 [8]. The 10 seconds time window was chosen to reduce the redundancy of the orginal data [7]. There are four additional categorical features, which are DeviceName, Attack, AttacksubType and label. Thus, there are 27 features in total.

In the raw data file, the ID column is deleted but the duplicate records weren't removed. The data are organized in a CSV file. CSV stands for comma-separated values, which is a text file format with each value separated by a comma. In the file, every data record is described by a line and each record has one or more fields separated by commas. Typically, CSV files represents tabular data in plain text with the same number of fields on each line, which stores all attribute types as text values, such as numbers, texts, dates, etc.

3.3 Detailed Method

3.3.1 Decimal Digits

The decimal places and significant digits of a number determine its precision. The number of decimal places refers to the "number of digits to the right of the decimal point", whereas the number of significant digits refers to the "total number of digits excluding the decimal point and all leading and trailing zeros". "Significant" here mostly means to exclude the meaningless zeros in the floating-point number, such as the 0s in 3.4500000 that might present in the data. Rounding is usually performed to achieve a value that can enhance the report and interpretation of data compared to the original data, as well as to avoid publishing a computed number, measurement, or estimate in an inaccurately precise manner.

Several rounding guidelines regarding decimal places and significant digits have been published in numerous publications. For instance, the European Association of Science Editors suggested a useful general recommendation for formatting regular tables: "numbers should be given in 2–3 effective digits" [41]. On the other hand, the APA Style suggested to round to two decimal places [11]. Some other rounding rules suggest a combination of the both decimal places and significant digits, which round the decimal length to guarantee that the standard deviation has two significant digits [31]. Cole

(2015) suggested some general principles to round summary statistics [18]. For numbers in tables, Ehrenberg suggested a rounding rule to facilitate mental arithmetic: "The general rule is to round to two significant or effective digits, where 'significant' or 'effective' here means digits which vary in that kind of data" [21]. It was indicated that effect sizes should have two or three significant digits, while measures of variability should have one or two significant digits. Besides, it is important to note that any intermediate steps of calculation should be executed to full precision, and rounding is performed only at the last step. This is done to prevent any errors due to approximation and ensure that the results are as exact as possible.

Nevertheless, the suitable number of significant figures largely depends on the purpose of the numerical data to be used and the context of it. Thus, it depends on the context and industry recommendations, together with data statistics to evaluate and detect insignificant digits in a certain numerical dataset and assess its application.

In this part, the purpose is to evaluate how the data storage size can be affected by dropping decimal digits, as stated in research question 1. Thus, the data will be rounded to different decimal length to assess on the storage space. The general rounding rule is applied when implementing the rounding: round the digit up if that digit is followed by 5, 6, 7, 8, or 9; otherwise, do not change the rounding digit.

3.3.2 Storage Format

To answer the second research question, the second approach is to transform the storage format of the current dataset to other suitable formats with different representation so that it can reduce the data size. A data file is usually stored under a text file format or CSV format (comma-separated values file). This is a form of plain text files that store data under string representation with a comma delimiter. It has been used widely for a long time and supported on all software systems without any further installation. However, CSV files are not optimized for storage space and performance since it does not support complex format types. Thus, it is not an ideal format when storing a large amount of data since it can result in large storage space and slow extracting and processing of data.

On the other hand, there are some other data file formats that can store data more efficiently. Binary file formats work directly with binary number representations, which are faster when loading into memory and save up more

space [46]. Thus, this method section will consider some open source binary file formats that are applicable to the research dataset, namely Pickle, HDF and Parquet. After that, the effect on the storage size will be assessed and compared.

The first alternative binary file format is Pickle, a Python object storage format to serialize and deserialize a Python object structure into a binary file. Specifically, it transforms a Python object into a byte stream to store it to the disk, maintains information and transfer data over the network. Pickle is convenient to use, especially with Python since it is a native format of Python for object serialization. The format stores all of the necessary information to reconstruct the serialized object correctly on another machine with a different architecture. This can largely reduce the data storage size, loading and processing time.

Another alternative is to store data in HDF file format. HDF stands for Hierarchical Data Format, an open source file format to store large, complex, heterogeneous data. This file format stores data similar to a file directory structure with different files, which is supported by many software platforms and programming languages, such as C/C++, Lua, Python and Matlab. Specifically, each data object in an HDF file is represented by a series of bits mapping to a single value from a certain set of values. These are predefined representation to store information such as the data type, the size of data, its dimensions and location in the file. This is a flexible and powerful file format since it can represent complex data relations and dependencies. There are two distinct varieties of HDF with different designs, but this research will use the current version HDF5, which improves upon the HDF4 library and applications to modern systems.

The last binary file format to consider is Parquet. This is a file format under the Apache Hadoop license that supports fast processing for large complex data. Parquet is a standard storage format supported by many different systems for data analytics, including SQLite, Apache Spark, Pig and Python. Unlike CSV, Parquet organizes data in columns of a specific data type, rather than in rows. It consists of row groups, header and footer, with each row group storing data of the same columns. Since it uses column-based storage, the file storage is much smaller compared to row-based storage in CSV. Parquet stores all similar data types of the same column together, and then compression can be applied to each column. By doing this, the compression is more efficient. To use Parquet, the fastparquet module in Python should be installed. This structure is optimized for fast query performance and low

I/O that minimizes the data needed to be scanned.

3.3.3 Data Compression

To further reduce the storage space and data transmission resources, the data are encoded using various compression algorithms. The goal of this method section is to evaluate the research question 3 about assessing storage size reduction for different compression techniques. The compression techniques considered in this section are lossless compression techniques that are widely used to compress or decompress a single file. These compression can be performed according to the selected operation mode.

Among various standard compression algorithms, zlib library is commonly used within various fields of applications due to its open source code [23]. The output format is compatible with GNU ZIP (or GZIP). The first version 0.1 was published in 1992 by Jean-loup Gailly and Mark Adler to replace the previous compression in Unix systems [22]. GZIP compresses data using the Deflate algorithm, which combines LZ77 algorithm and an adopted Huffman encoding. It first compresses the data by LZ77 [60] and then encodes the output with Huffman coding [32]. Fundamentally, the compression is based on statistics, which assigns sequences of high probability with short bit string and vice versa. Thus, the compression efficiency depends on the size of the original data and the spread of common substrings. GZIP is widely used and can be applied for all the file formats concerning in the previous section, namely CSV, Pickle, HDF, Parquet.

The next compression format to be considered is BZIP2 (BZ2). This is chosen because of its intensive computation power, popular use, public source code and relevant daily application [42]. The first public release version 0.15 of BZ2 was published by Julian Seward in 1996 [50]. It uses several layers of various compression algorithms to compress data, and applies the reverse order to decompress. In essence, a sequence of duplicated symbols is replaced by the first four characters and a repeat length of that symbol. As an example, the sequence AAAAAABBBBCCCD is encoded with AAAA\2BBBB\0CCCD, in which \2 and \0 correspondingly represent byte values 2 and 0. The compressed sequences in BZ2 can be independently decompressed in parallel, which is efficient in big data applications. Unlike GZIP, BZ2 compression will only be applied to CSV and Pickle since the internal compression algorithms of HDF and Parquet do not support BZ2.

The last compression format considered is XZ, which supports various for-

mats. This is a new data compression technique for general purposes, similar to the previous two mentioned compression. XZ belongs to the XZ Utils set of compressors, which was released by Lasse Collin in 2011 [19]. The compression uses the Lempel–Ziv coding combined with arithmetic coding to compress the data. While the compression is new, it can result in more compact compression [58]. Since the internal compression algorithms of HDF and Parquet do not support XZ, the compression will only be applied to CSV and Pickle.

3.3.4 Combined Lossy Compression

As discussed in Chapter 1 and 2, the goal of this research is to introduce a combined lossy compression method that the users can control the loss of information in the data. This can be achieved by compressing on the decimal length, changing file format and applying a lossless compression.

Figure 3.2: A Combined Lossy Compression

In this way, the users can control the precision of the quantitative data by deciding on the decimal length. Furthermore, there are no information loss when the compressed file is transferred and accessed by various users. This will help to maintain the quality of the quantitative data.

3.3.5 Measures of Performance

To evaluate the effect of transforming the file format, the various file format will be written and compared based on the file storage size, time to write the file and time to read and process data into Python. The storage size is measured in megabytes (MB) and the time is measured in seconds. This measures the encoding process of the data without saving index of the Pandas dataframe.

To measure the efficiency of different compression formats, the evaluation is based on comparing the compressed size, amount of compression, compression time, and decompression time. In particular, the compression time

is measured as the time for compressing the data in Python to a compression format. Similarly, the decompression time is calculated as the time for extracting the compressed file to dataframe in Python.

The most common measurement of the efficiency of a compression algorithm is the compression ratio. It is calculated as the ratio of the total storage size of uncompressed data and compressed data. The calculation is represented by the following formula:

$$Compression\ Ratio = \frac{Storage\ Size\ Before\ Compression}{Storage\ Size\ After\ Compression}$$

The compression ratio indicates the average number of storage units needed to store the compressed data.

Next to that, another measurement is used, which is the space saving. The space saving is interpreted as the decrease in storage space relative to the uncompressed storage space, which is calculated as follows:

$$Space\ Saving = 1 - \frac{Compressed\ Storage\ Size}{Uncompressed\ Storage\ Size}$$

The space saving describes the percentile of space saving saved by using compression. For example, the space saving of 0.8 implies that 80% of the original storage size is saved with compression.

Chapter 4

Implementation

4.1 System

The implementation in this research is implemented entirely using Python on Google Colaboratory, which is similar to Jupyter Notebook but developed by Google. It is an open source web IDE for Python based on cloud technology. Each Collab notebook provides the following computation power:

- **CPU:** Intel(R) Xeon(R) Processor with two cores 2.20 GHz, 12 GB RAM and 25 GB Disk space
- **GPU:** NVIDIA Tesla K80 with two cores 2.2 GHz, 12 GB RAM, 358GB Disk space

4.2 Data Preprocessing

The original dataset is the BoTNeTIoT-L01-v2 file. Initially, this dataset has a storage size of 1.56 GB, containing 7,062,606 rows and 27 attributes. After loading the dataset in Python, the data are checked and preprocessed to prepare for further processing.

To ensure that the data are valid, the first step is to remove any missing values in the data. After checking all entries, there are no empty values found and thus there is no need to remove any data from any rows or columns.

However, there are 621,659 duplicated rows, which need to be removed from the raw dataset. To ensure that the duplicated records is correctly removed, the original data are loaded in a dataframe with all columns converted as type string, similar to how the data are stored in the original CSV file. Then, all

the duplicated rows are removed and the new dataset is written back to a new CSV file without the index column from Pandas dataframe.

The data preprocessing results in a new CSV file BoTNeTIoT with the storage size of 1.48 GB, containing 6,440,947 rows and 27 attributes. This is the complete data to implement the discussed methods in the following sections. After that, the final results will be presented in the Chapter 5.

4.3 Implementation of Method

All storage measurements are obtained in MegaBytes (MB) by divide the obtained storage size by 1024^2 (since Python measures in Bytes).

4.3.1 Decimal Digits

In this section, data are rounded to different decimal lengths to assess on the storage size. This process includes the following steps:

1. The numeric part of the data is rounded to 1 to 6 decimal digits using round() function, the last four columns (categorical variables) remain the same.
2. The rounding dataframe is written to CSV file using Pandas function to_csv() with the option index=False.
3. Measure the storage space after rounding to different decimal lengths using command os.stat(file name).st_size.
4. Calculate the compression ratio and space saving.
5. Fit a linear model using LinearRegression model from sklearn.linear_model to obtain the coefficients.

4.3.2 Storage Format

In this section, the data is written to different file formats to measure the storage size reduction. Package h5py should be imported and fastparquet is required to be installed before running the following process:

1. Data is saved to different file formats for CSV, Pickle, HDF, Parquet using the corresponding functions: to_csv(), to_pickle(), to_hdf(), to_parquet() with the option index=False and compression='None' and measure the executed time.

2. Use os.path.getsize() method to check the storage size of the various file formats.

3. Compute the compression ratio and space saving.

4. Import the data back to Python using the following commands: read_csv(), read_pickle(), read_hdf(), read_parquet() and measure the executed time.

The implementation in step 1 and step 4 are measured using the command %timeit in IPython. This is used to time a particular code command after executing each line. The command %timeit will execute each code 5 times and the best timing will be saved in a result dataframe. .

4.3.3 Data Compression

In this section, the discussed compression methods are implemented to measure the storage size reduction. The procedure is shown below:

1. The data compression is performed with different file formats for CSV, Pickle, HDF, Parquet using the corresponding functions: to_csv(), to_pickle() with the option index=False and compression='gzip', 'bz2', 'xz'; and to_hdf(), to_parquet() with the option index=False and compression='gzip'. The executed time is also measured.

2. Use os.path.getsize() method to check the storage size of the various compression formats.

3. Compute the compression ratio and space saving.

4. Import the data back to Python using the following commands: read_csv(), read_pickle(), read_hdf(), read_parquet() and measure the executed time.

The implementation in step 1 and step 4 are measured using the command %timeit in IPython by executing each code 5 times and saving the best timing in a result dataframe. .

Chapter 5

Results

5.1 Decimal Digits

When looking at how the decimal length can affect the storage size of the CSV file, the data is rounded off to various decimal places. After the implementation in section 4.3.1, the achieved results can be seen in Table 5.1.

Decimal Length	Storage Size (MB)	Compression Ratio	Space Saving
6	1307.91	1.16	0.14
5	1249.51	1.22	0.18
4	1188.07	1.28	0.22
3	1127.63	1.35	0.26
2	1065.86	1.43	0.30
1	1012.16	1.50	0.34

Table 5.1: Storage size, Compression ratio and Space saving of different decimal length in CSV

The storage size of the CSV files shows a slight decreasing trend after removing each decimal digit. At the least decimal length of 1 digit, the storage size of the CSV file is around 1012.16 MB, which is approximately two third of the original storage size (1522.46 MB). In addition, it has the highest compression ratio of around 1.5 and the highest storage saving of around 0.34, which means that around 34% of the original storage space is saved by compressing the original data to 1 decimal length. Besides, every extra removed decimal digit approximately reduces the storage space by 50 MB and saves around 4% of the storage space on average.
To further understand the relationship between the number of decimal places

and the storage size, a linear regression model is fitted. The number of decimal places is used as an independent variable and the storage size is fitted as a dependent variable. The model is visualized in Figure 5.1.

Figure 5.1: Linear regression model on the storage size

After fitting the linear regression model, the obtained coefficient is approximately 59.7 and the intercept at 0 is around 949.5. This suggests that increasing one decimal number will result in an increase of around 59.7 MB in the storage size and the storage size without any decimal digits is around 949.5 MB.

5.2 Storage Format

After storing data in different file formats, the storage size, the write time and read time are compared in Table 5.2.

Storage Format	Storage Size (MB)	Write Time	Read Time
CSV	1673.92	173.47	36.15
Pickle	1255.72	6.27	3.19
HDF	1303.06	7.23	6.94
Parquet	752.32	7.83	3.54

Table 5.2: Storage size, Write time, Read time, Compression Ratio and Space saving of different storage formats without compression

Noticeably, the Parquet file format has significantly reduced the storage size to only 752.32 MB, which is smaller than one half of the CSV file (around 1673.92 MB). The Pickle and HDF file formats result in similar storage spaces, of around 1255.72 MB and 1303.06 MB respectively.

Looking at the time to construct the file format and the time to load the data to Python, it can be seen that the time to write and read Pickle file format are the fastest, for around 6 seconds and 3 second correspondingly. This might be explained because Pickle is a native format of Python for object serialization. Thus, it is the most supported file format over other file formats when using Python.

Besides, the read time of HDF and Parquet are quite similar, for around 7.5 seconds, while the read time of HDF file almost doubles the read time of Parquet file. On the other hand, the CSV file requires much longer time to process, for around 173.47 seconds to constructing the file and around 36.15 seconds to import data back in Python.

5.3 Data Compression

In this section, different compression formats are implemented regarding the storage size. Table 5.3 presents the comparison between the compression for different file formats.

Format	**Size (MB)**	**Compress Ratio**	**Space Saving**	**Compress Time**	**Decompress Time**
CSV (GZIP)	359.88	4.23	0.76	297.25	44.30
CSV (BZ2)	370.91	4.10	0.76	458.89	150.39
CSV (XZ)	278.50	5.47	0.82	1762.05	71.15
Pickle (GZIP)	413.49	3.68	0.73	112.05	8.48
Pickle (BZ2)	439.33	3.47	0.71	251.57	87.75
Pickle (XZ)	224.25	6.79	0.85	286.94	29.98
HDF (GZIP)	458.77	3.32	0.70	30.60	11.89
Parquet (GZIP)	410.97	3.70	0.73	52.92	5.79

Table 5.3: Storage size (MB), Compression ratio, Space saving, Compression time and Decompression time of different compression formats

From the table, it can be seen that XZ is the most compact compression format, for around 278.50 MB and 224.25 MB for compressing CSV and Pickle file respectively. Overall, XZ has the largest compression ratio of 6.79 and

the largest space saving of 85% for Pickle file. A compression ratio of 6.79 using XZ compression means that the original file storage is 6.79 times higher than the compressed file storage. Besides, GZIP performs a bit better than BZ2 on both the CSV and Pickle file. Surprisingly, GZIP performs best for CSV, worst for HDF and almost equally for Pickle and Parquet.

There is a trade-off between the time efficiency of the compression and the storage size. Specifically, XZ format takes around 1762.05 seconds to compress the CSV file, while the GZIP and BZ2 takes around 297.25 seconds and 458.89 seconds respectively. The compression time for XZ is also the lowest on Pickle file, but it is much better than for CSV. The compression time for HDF and Parquet files are extremely fast, for around 30.60 seconds and 52.92 seconds correspondingly. On the other hand, the decompression time follows a different trend. The decompression time for BZ2 on CSV is the highest, which doubles that for XZ. Noticeably, the decompression time for GZIP on Pickle, HDF and Parquet is especially fast, for only 8.48, 11.89 and 5.79 seconds respectively.

5.4 Combined Lossy Compression

To construct a combined lossy compression technique, the algorithm can be implemented as follows:

1. Round the decimal digits.
2. Store the data in a file format.
3. Compress the file with the best possible compression resulted from the previous section.
4. Possibly perform different combinations and compare the storage size, compression ratio, space saving, compression time and decompression time to select the most suitable.

Specifically, the users can first round the decimal digit to their choice as in the decimal digits parts. After that, the user can decide on which compression to use. For the best compression ratio and saving space, it is recommended to store the data as a compressed Pickle file using XZ as the compressor. This can achieve the most compact compression that does not take too much time like the XZ compression for CSV file. When considering very big data, another recommendation is to use GZIP on HDF or Parquet file format since both perform extremely fast on compressing and extracting the data.

Chapter 6

Discussion

This research used an open-access dataset to answer the research questions about reducing the storage size of quantitative data by various techniques. It is then determined to do so by a combined lossy compression method, including dropping decimal digits, transforming the storage formats and compressing the file using lossless compression. After researching relevant literature, this study implemented some possible methods to store and compress data efficiently. The achieved results show that the storage space is much reduced by doing so.

The first research question is about the effect of decimal length on the storage size. This was achieved by dropping decimal places by each digit until there is only one decimal left. The result shows that there is a linear trend on the storage size by reducing the decimal length. Specifically, each decimal deletion results in an average of around 4% space saving, which corresponds to a decrease of around 59.7 MB approximated by the linear regression model.

Next to that, the second research question is also addressed by transforming the file storage format. For the discussed file formats, Parquet file results in the least storage size, which consequently has the highest space saving. Its storage size is only one half of the original CSV. Besides, the result has shown that all other transforming formats, namely Pickle, HDF and Parquet, are more efficient than the original CSV file, compared on storage size, transforming time and loading time.

Lastly, the third research question is evaluated by performing various compression techniques, combined with different data file formats. The most efficient format is the XZ compression, best when compressing on a Pickle file. The achieved space saving is around 85%, which gives a significant reduc-

tion on the storage size. Considering the other compression formats, there is a trade-off between compression ratio and compression/decompression time. Thus, it also depends on the computational power and CPU resources that are available to determine the compression for other datasets. XZ has the best compression ratio and saves the most disk space, but it also requires longer time to compress and extract data, especially for CSV file. As a result, it could be a good option when there are not much updates that need to frequently re-compress the data. It is recommended to use XZ compression on Parquet file unless compression time is very strict. Otherwise, GZIP on HDF and Parquet provide a fast alternative compression way.

The strength of this method is that the users can control the information loss in the lossy compression process. This is achieved by determining the decimal length to compress. For example, rounding to two decimal digits will result in a storage size of around 1065.86 MB and using the highest compression ratio of 6.79 will reduce the storage space to around 156.97 MB. This is remarkable compared to the original size of 1522.46 MB. From the reviewed literature, it is argued that most lossy compression algorithms usually focus on multimedia data with a clear domain specificity. As regards, there is a lack of transparent lossy compression techniques. Thus, this research has introduced a new combined lossy compression approach that can be applied to other quantitative dataset. Another advantage is that the implementation is clear and can be reproduce easily since all data, coding source and the compression algorithms are publicly available.

On the other hand, a relative weakness of the method is the limited number of the file formats and compression algorithms implemented in this research. Since the scope of the research is limited, it is hard to considered all available compression techniques. Furthermore, there are not many compression methods specifically developed for scientific quantitative data rather than image data. Thus, a possible improvement to the project could be to examine other different combinations of file format and compression algorithms. This could also be a future research to investigate more options for the combined lossy compression approach to further reduce the storage space and facilitate data transmission. Since most compression techniques require data specificity, another possibility is to split the data into two parts, including numeric data and text data. Then, one can apply different compression techniques separately for each part, and recombine when there is a need to update or modify the whole data.

Chapter 7

Conclusion

In this research, the goal is to provide a combined lossy compression approach that can reduce the storage size of a large quantitative dataset. To do so, it has been researched how decimal digits can affect the storage size and whether rounding can reduce a lot of data storage. Another approach is to change the file format of the original CSV file to a more efficient binary file format. The last approach is to reduce the storage space and transmission resources by compressing the data on different formats. Based on the results, it is shown that deleting each decimal place will result in a space saving of around 4%. In addition, binary file formats significantly reduce the storage size, with more than a half storage size reduction using Parquet. Furthermore, XZ is the most efficient compression format compared with GZIP and BZ2 on the considered IoT dataset, but it might take longer time to compress/decompress the data. In the end, it is recommended to use XZ compression on a Pickle file to compress the data. Otherwise, it is only recommended to use other compression like GZIP on HDF or Parquet file when there are insufficient computational resources and strict running time limit.

Acknowledgements

First and foremost, I would like to thank my research supervisor, Professor Erik Quaeghebeur. I want to express my sincere thank for his dedicated support and guidance during the researching process. Without his critical feedback, I might not be able to progress on this bachelor thesis.

Besides, I would also want to say thank to my thesis groupmate, Iris Jacobs and Patrick Huijten, for inspiring new ideas about the research topic.

Finally, I would like to thank my family and friends for supporting me during the compilation of this bachelor thesis.

Bibliography

[1] Mark Ainsworth, Ozan Tugluk, Ben Whitney, and Scott Klasky. Multilevel techniques for compression and reduction of scientific data—the univariate case. *Computing and Visualization in Science*, 19(5):65–76, 2018.

[2] Mark Ainsworth, Ozan Tugluk, Ben Whitney, and Scott Klasky. Multilevel techniques for compression and reduction of scientific data - the multivariate case. *SIAM Journal on Scientific Computing*, 41(2):A1278–A1303, 2019.

[3] Mark Ainsworth, Ozan Tugluk, Ben Whitney, and Scott Klasky. Multilevel techniques for compression and reduction of scientific data-quantitative control of accuracy in derived quantities. *SIAM Journal on Scientific Computing*, 41(4):A2146–A2171, 2019.

[4] Mark Ainsworth, Ozan Tugluk, Ben Whitney, and Scott Klasky. Multilevel techniques for compression and reduction of scientific data - the unstructured case. *SIAM Journal on Scientific Computing*, 42(2):A1402–A1427, 2020.

[5] Omar Alfandi, Salam Khanji, Liza Ahmad, and Asad Khattak. A survey on boosting iot security and privacy through blockchain. *Cluster Computing*, 24(1):37–55, 2021.

[6] Alaa Alhowaide. Iot dataset for intrusion detection systems (ids). https://www.kaggle.com/datasets/azalhowaide/iot-dataset-for-intrusion-detection-systems-ids?select=BoTNeTIoT-L01-v2.csv. Accessed: 2022-03-22.

[7] Alaa Alhowaide, Izzat Alsmadi, and Jian Tang. Features quality impact on cyber physical security systems. In *2019 IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON)*, pages 0332–0339. IEEE, 2019.

[8] Alaa Alhowaide, Izzat Alsmadi, and Jian Tang. An ensemble feature selection method for iot ids. In *2020 IEEE 6th International Conference on Dependability in Sensor, Cloud and Big Data Systems and Application (DependSys)*, pages 41–48. IEEE, 2020.

[9] Alaa Alhowaide, Izzat Alsmadi, and Jian Tang. Towards the design of real-time autonomous iot nids. *Cluster Computing*, pages 1–14, 2021.

[10] Rabia Arshad, Adeel Saleem, and Danista Khan. Performance comparison of huffman coding and double huffman coding. In *2016 Sixth International Conference on Innovative Computing Technology (INTECH)*, pages 361–364. IEEE, 2016.

[11] American Psychological Association et al. Publication manual of the american. 2010.

[12] Simon Berkovich and Duoduo Liao. On clusterization of "big data" streams. In *Proceedings of the 3rd International Conference on Computing for Geospatial Research and Applications*, pages 1–6, 2012.

[13] Wasim Ahmad Bhat. Bridging data-capacity gap in big data storage. *Future Generation Computer Systems*, 87:538–548, 2018.

[14] Abraham Bookstein and Shmuel T Klein. Is huffman coding dead? *Computing*, 50(4):279–296, 1993.

[15] Alan Bryman. Social research methods 4th ed, 2012.

[16] Franck Cappello, Sheng Di, Sihuan Li, Xin Liang, Ali Murat Gok, Dingwen Tao, Chun Hong Yoon, Xin-Chuan Wu, Yuri Alexeev, and Frederic T Chong. Use cases of lossy compression for floating-point data in scientific data sets. *The International Journal of High Performance Computing Applications*, 33(6):1201–1220, 2019.

[17] Shubham Chandak, Kedar Tatwawadi, Chengtao Wen, Lingyun Wang, Juan Aparicio Ojea, and Tsachy Weissman. Lfzip: Lossy compression of multivariate floating-point time series data via improved prediction. In *2020 Data Compression Conference (DCC)*, pages 342–351. IEEE, 2020.

[18] TJ Cole. Too many digits: the presentation of numerical data. *Archives of disease in childhood*, 100(7):608–609, 2015.

[19] Lasse Collin. Xz utils. *http://tukaani.org/xz/*, 2011.

[20] Sheng Di and Franck Cappello. Fast error-bounded lossy hpc data compression with sz. In *2016 ieee international parallel and distributed processing symposium (ipdps)*, pages 730–739. IEEE, 2016.

[21] Andrew SC Ehrenberg. Rudiments of numeracy. *Journal of the Royal Statistical Society: Series A (General)*, 140(3):277–297, 1977.

[22] Jean-loup Gailly and Mark Adler. Gnu gzip. *https://www.gzip.org/*, 1992.

[23] Jean-loup Gailly and Mark Adler. Zlib compression library. 2004.

[24] Robert Gallager. Variations on a theme by huffman. *IEEE Transactions on Information Theory*, 24(6):668–674, 1978.

[25] M Ghotkar and P Rokde. Big data: How it is generated and its importance. *IOSR Journal of Computer Engineering*, 1(5), 2016.

[26] Lisa M Given. *The Sage encyclopedia of qualitative research methods*. Sage publications, 2008.

[27] Jill R Goldschneider. *Lossy compression of scientific data via wavelets and vector quantization*. University of Washington, 1997.

[28] Stephan Haller, Stamatis Karnouskos, and Christoph Schroth. The internet of things in an enterprise context. In *Future internet symposium*, pages 14–28. Springer, 2008.

[29] Reza Hashemian. Memory efficient and high-speed search huffman coding. *IEEE Transactions on communications*, 43(10):2576–2581, 1995.

[30] Martin Hilbert and Priscila López. The world's technological capacity to store, communicate, and compute information. *Science*, 332(6025):60–65, 2011.

[31] WG Hopkins, AM Batterham, DB Pyne, and FM Impellizzeri. Sports medicine update response. *Scandinavian Journal of Medicine & Science in Sports*, 21(6):867–868, 2011.

[32] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[33] Barna Iantovics and Roumen Kountchev. *Advanced Intelligent Computational Technologies and Decision Support Systems*. Springer, 2014.

[34] Uthayakumar Jayasankar, Vengattaraman Thirumal, and Dhavachelvan Ponnurangam. A survey on data compression techniques: From the perspective of data quality, coding schemes, data type and applications. *Journal of King Saud University-Computer and Information Sciences*, 33(2):119–140, 2021.

[35] Shaomeng Li, Nicole Marsaglia, Christoph Garth, Jonathan Woodring, John Clyne, and Hank Childs. Data reduction techniques for simulation, visualization and data analysis. In *Computer graphics forum*, volume 37, pages 422–447. Wiley Online Library, 2018.

[36] Xin Liang, Sheng Di, Dingwen Tao, Sihuan Li, Shaomeng Li, Hanqi Guo, Zizhong Chen, and Franck Cappello. Error-controlled lossy compression optimized for high compression ratios of scientific datasets. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 438–447. IEEE, 2018.

[37] Peter Lindstrom. Fixed-rate compressed floating-point arrays. *IEEE transactions on visualization and computer graphics*, 20(12):2674–2683, 2014.

[38] Omar Adil Mahdi, Mazin Abed Mohammed, and Ahmed Jasim Mohamed. Implementing a novel approach an convert audio compression to text coding via hybrid technique. *International Journal of Computer Science Issues (IJCSI)*, 9(6):53, 2012.

[39] Bernard Marr. How much data do we create every day? the mind-blowing stats everyone should read. *Forbes*, 21:1–5, 2018.

[40] Alistair Moffat. Huffman coding. *ACM Computing Surveys (CSUR)*, 52(4):1–35, 2019.

[41] European Association of Science Editors. Ease guidelines for authors and translators of scientific articles to be published in english. *J Teh Univ Heart Ctr*, 6:206–210, 2011.

[42] Victor Pankratius, Ali Jannesari, and Walter F Tichy. Parallelizing bzip2: A case study in multicore software engineering. *IEEE software*, 26(6):70–77, 2009.

[43] Ranjan Parekh. *Fundamentals of Image, Audio, and Video Processing Using MATLAB® and Fundamentals of Graphics Using MATLAB®: Two Volume Set*. CRC Press, 2022.

[44] Charith Perera, Chi Harold Liu, and Srimal Jayawardena. The emerging internet of things marketplace from an industrial perspective: A survey. *IEEE transactions on emerging topics in computing*, 3(4):585–598, 2015.

[45] Davy Preuveneers and Yolande Berbers. Internet of things: A context-awareness perspective. *The Internet of Things: From RFID to the Next-Generation Pervasive Networked Systems*, pages 287–307, 2008.

[46] Erik Quaeghebeur and Michiel B Zaaijer. How to improve the state of the art in metocean measurement datasets. *Wind Energy Science*, 5(1):285–308, 2020.

[47] David Salomon. *Data compression: the complete reference*. Springer Science & Business Media, 2004.

[48] David Salomon and Giovanni Motta. *Handbook of data compression*. Springer, 2010.

[49] Khalid Sayood. *Introduction to data compression*. Morgan Kaufmann, 2017.

[50] Julian Seward. bzip2 and libbzip2. *http://www.bzip.org/*, 1996.

[51] Statista. Amount of information globally 2010-2025. https://www.statista.com/statistics/871513/worldwide-data-created/, 2022. Accessed: 2022-03-03.

[52] Minaldevi K Tank. Implementation of lempel-ziv algorithm for lossless compression using vhdl. In *Thinkquest 2010*, pages 275–278. Springer, 2011.

[53] Dingwen Tao, Sheng Di, Zizhong Chen, and Franck Cappello. Significantly improving lossy compression for scientific data sets based on multidimensional prediction and error-controlled quantization. In *2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1129–1139. IEEE, 2017.

[54] Jiannan Tian, Cody Rivera, Sheng Di, Jieyang Chen, Xin Liang, Dingwen Tao, and Franck Cappello. Revisiting huffman coding: Toward extreme performance on modern gpu architectures. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 881–891. IEEE, 2021.

[55] Chun-Wei Tsai, Chin-Feng Lai, Ming-Chao Chiang, and Laurence T Yang. Data mining for internet of things: A survey. *IEEE Communications Surveys & Tutorials*, 16(1):77–97, 2013.

[56] Jeffrey Scott Vitter. Design and analysis of dynamic huffman codes. *Journal of the ACM (JACM)*, 34(4):825–845, 1987.

[57] Terry A. Welch. A technique for high-performance data compression. *Computer*, 17(06):8–19, 1984.

[58] Erik S Wright et al. Using decipher v2. 0 to analyze big biological sequence data in r. *R J.*, 8(1):352, 2016.

[59] Peng Zhang. 6-data communications in distributed control system. *Industrial Control Technology*, pages 675–774, 2008.

[60] Jacob Ziv and Abraham Lempel. A universal algorithm for sequential data compression. *IEEE Transactions on information theory*, 23(3):337–343, 1977.

[61] Jacob Ziv and Abraham Lempel. Compression of individual sequences via variable-rate coding. *IEEE transactions on Information Theory*, 24(5):530–536, 1978.