APPLYING MACHINE LEARNING TO IDENTIFY OPTIMAL FILE COMPRESSION METHODS

BY

PITAWAT CHAIVUTINUN

AN INDEPENDENT STUDY SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE DEGREE OF MASTER OF ENGINEERING (ARTIFICIAL INTELLIGENCE AND INTERNET OF THINGS)
SIRINDHORN INTERNATIONAL INSTITUTE OF TECHNOLOGY
THAMMASAT UNIVERSITY
ACADEMIC YEAR 2025

THAMMASAT UNIVERSITY
SIRINDHORN INTERNATIONAL INSTITUTE OF TECHNOLOGY

INDEPENDENT STUDY

BY

PITAWAT CHAIVUTINUN

ENTITLED

APPLYING MACHINE LEARNING TO IDENTIFY OPTIMAL FILE COMPRESSION METHODS

was approved as partial fulfillment of the requirements for
the degree of Master of Engineering (Artificial Intelligence and Internet of Things)

on November 17, 2025

Member and Advisor	[illegible]
	(Associate Professor Cholwich Nattee, D.Eng.)
Member and Co-Advisor	[illegible]
	(Associate Professor Nirattaya Khamsemanan, Ph.D.)
Member	[illegible]
	(Assistant Professor Seksan Laitrakun, Ph.D.)
Director	[illegible]
	(Associate Professor Kriengsak Panuwatwanich, Ph.D.)

Independent Study Title	APPLYING MACHINE LEARNING TO IDENTIFY OPTIMAL FILE COMPRESSION METHODS
Author	Pitawat Chaivutinun
Degree	Master of Engineering (Artificial Intelligence and Internet of Things)
Faculty/University	Sirindhorn International Institute of Technology/ Thammasat University
Advisor	Associate Professor Cholwich Nattee, D.Eng.
Co-Advisor	Associate Professor Nirattaya Khamsemanan, Ph.D.
Academic Years	2025

ABSTRACT

This independent study addresses the inefficiency of using a single lossless compression algorithm for diverse file types, a common practice in financial reporting and other domains. We propose an adaptive framework that uses supervised machine learning to predict the most suitable compression method for each file. A dataset of approximately 120,000 real-world files (including text, tabular, and semi-structured formats) was created. Each file was compressed using six major algorithms (Zstd, LZ4, Brotli, LZMA, Bzip2, and zlib) to determine the "ground-truth" best method based on the lowest compression ratio achieved within a 30-second time limit. We extracted an initial set of 15 structural features for each file. A Sequential Feature Selection (SFS) technique was then employed to identify the most predictive subset of features. The final model predicts the optimal algorithm, achieving compression ratios close to the empirical optimum without the high cost of an exhaustive search. This model can be embedded into existing data pipelines to automatically reduce storage costs and data transfer times with minimal added latency.

Keywords: Lossless compression, machine learning, algorithm selection, feature engineering, file storage optimization, data transfer efficiency

ACKNOWLEDGEMENTS

First and foremost, I would like to express my deepest gratitude to my advisor, Assoc. Prof. Cholwich Nattee, and my co-advisor, Assoc. Prof. Nirattaya Khamsemanan, for their invaluable guidance, encouragement, and continuous support throughout the course of my Master's Degree study and research. Their insightful advice, constructive feedback, and profound knowledge have been the driving force behind the successful completion of this thesis. I am truly honored to have had the opportunity to learn under their supervision, and I am deeply thankful for their patience and inspiration that have shaped both my research and academic growth.

I would also like to express my appreciation to my committee members for their time, thoughtful comments, and valuable suggestions that helped improve the quality of this work.

My sincere thanks go to all colleagues and friends at the Artificial Intelligence and Internet of Things (AI & IoT) program at Sirindhorn International Institute of Technology (SIIT), Thammasat University, for their encouragement, technical discussions, and friendship that made my graduate life both productive and enjoyable.

Finally, I would like to extend my heartfelt gratitude to my family for their unconditional love, patience, and endless support. Their encouragement and belief in me have been my greatest motivation throughout my academic journey.

Pitawat Chaivutinun

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS/ABBREVIATIONS

Symbols/Abbreviations	**Terms**
ASCII	American Standard Code for Information Interchange
ML	Machine Learning

CHAPTER 1
INTRODUCTION

Nowadays, data compression is playing an increasingly important role in various technologies. It reduces storage costs, lowers data transfer, and optimizes resource utilization for many enterprises. Yang, Qin and Hu (2023) state that traditional data compression algorithms work well with specific types of data, but this can cause problems with heterogeneous datasets and complicated file structures. This research aims to explore the potential of machine learning to find the most effective compression method for a given file based on its data structure. By applying machine learning models, we can achieve higher compression ratios with acceptable compression speeds, resulting in efficient resource management compared to traditional methods.

1.1 Data Compression

Data compression is the process of encoding data into a file, producing an output with fewer bits than the original file. Data compression reduces the size of data by identifying and eliminating redundancy and irrelevancy. Compression techniques are essential for various applications. Reducing file size leads to reduced storage space, increasing data transfer speeds, lowering bandwidth usage, and enhancing resource utilization (Fitriya, Purboyo and Prasasti, 2017).

Traditional data compression algorithms can be categorized into lossless and lossy compression. Lossless compression algorithms, such as Huffman coding and Lempel-Ziv variations, ensure that, after decompression, the reconstructed data is identical to the original. Unlike lossless compression, lossy compression algorithms, such as JPEG for images, achieve higher compression ratios but discard some data, so the reconstructed data may not match the original quality.

1.2 Machine Learning with Data Compression

Machine learning is a field of artificial intelligence (AI) that focuses on enabling computer systems to learn from data without explicit programming. It involves the development of algorithms that allow computers to identify patterns, make predictions, and improve their performance over time based on the input data. Machine learning has gradually become a powerful tool for solving various modern problems, including data compression.

In this research, machine learning can be used to learn patterns and entropy of data in files to improve compression efficiency. Machine learning can analyze file structures, identify relevant features, and predict the most effective data compression algorithm for a given file. Burtchell & Burtscher (2024) demonstrated that in traditional compression algorithms, single compression algorithms might not fit diverse data. It might have a good compression ratio with most types of files, but it is impossible for all types of files. Then, the approach of using machine learning has the potential to outperform using single data compression algorithms, especially for heterogeneous datasets that diverse data characteristics.

1.3 Problem Statement

Traditional lossless compression algorithms often do not perform consistently on different kinds of data. Since the structure of files varies a lot, no single algorithm always gives the best result. This study aims to solve the problem of selecting algorithms by using machine learning to predict the most effective compression for each file, based on its structural features.

1.4 Motivation

The motivation for this research stems from the limitations of using one static compression method for all files. This approach often leads to suboptimal compression ratios and inefficient system performance, especially when working with diverse datasets. By applying machine learning, we aim to develop a more adaptive and intelligent compression system that selects the optimal algorithm dynamically, based on file structure.

1.5 Objectives

The objective of this research is to develop machine learning models for predicting the best compression algorithms for specific files in terms of compressed size and compression time, based on their structural features.

1.6 Expected Outcomes

The expected outcome of this research is a machine learning model that can intelligently predict the most effective compression algorithm, in terms of compressed size and compression time, for a given file based on its structural features.

1.7 Scope and Limitation

The scope of this study focuses only on lossless compression algorithms. The dataset for training will mostly come from financial data. The algorithms within scope are Zstd, LZ4, Brotli, LZMA (XZ), Bzip2, and Deflate (ZIP, Gzip). We do not focus on lossy compressions or any data that do not belong to financial type.

1.8 Structure of the Independent Study

This independent study contains 6 chapters.

Chapter 1: Introduction

This chapter introduces data compression techniques, challenges of traditional data compression algorithms, and the possibility of applying machine learning in data compression including current problems, the aim and the scope of the research.

Chapter 2: Review Of Literature

This chapter discusses the fundamental tools and techniques of the research. This chapter

also includes the review and validation of the related previous works.

Chapter 3: Methodology

This chapter describes the dataset creation, feature extraction, model development, and evaluation process for predicting optimal compression algorithms.

Chapter 4: Experimental Results

Presents the results of the compression testing and machine learning model performance. Includes performance metrics, model comparisons, and analysis of feature selection outcomes.

Chapter 5: Conclusion and Future Direction

This chapter is the summarization of the research on applying machine learning to identify optimal file compression methods and discusses future directions for improving adaptive data compression.

CHAPTER 2
REVIEW OF LITERATURE

This chapter explains studies related to data compression and reviews previous work concerning machine learning approaches for identifying optimal file compression algorithms. This chapter also explains the background knowledge, techniques and tools that are used in this research.

2.1 Data Compression

Data compression is the technique used to reduce file size by encoding information efficiently. It is commonly used for optimizing data storage and improving data transmission. Compression methods are usually divided into two categories: lossless compression and lossy compression. Lossless methods ensure decompress data the same as original data, mostly used in scenarios where data integrity is crucial, such as executable programs, textual information, or archival systems. Lossless compression algorithms that are currently popular include Huffman coding, DEFLATE (zlib), bzip2, Lempel-Ziv-Markov chain Algorithm (LZMA), Zstandard (Zstd), LZ4, and Brotli.

2.2 Lossless Compression Algorithms

This study utilizes multiple well-established lossless compression algorithms. The following subsections provide detailed descriptions of these algorithms, including their strengths and potential limitations.

2.2.1 zlib (DEFLATE)

The DEFLATE algorithm used in zlib combines the LZ77 algorithm and Huffman coding. LZ77 does compression by replacing repeated data sequences with pointers to previously seen sequences, which greatly helps reduce redundancy. Huffman coding is then applied for entropy encoding, to compress the data even more.

Due to synergy between LZ77 and Huffman coding, zlib achieves a good balance of speed, memory usage, and compression ratio. It was also widely adopted in web protocols (HTTP compression, PNG image format, etc.) for its reliability and performance (Deutsch, 1996).

2.2.1.1 LZ77 Dictionary Encoding

LZ77 (Ziv-Lempel 1977) compresses data by replacing repeated occurrences of substrings with references to a single copy of that substring existing earlier in the uncompressed data. Formally, suppose we have an input sequence of bytes:

$$S = (s_1, s_2, s_3, \dots, s_n), \tag{2.1}$$

where each s_i is a symbol (often a byte). Instead of storing a repeated substring $(s_k, \dots, s_{k+m-1})$ in full, LZ77 stores a triple (d, l, c):

d : The distance backward from the current position (how far back to copy).

l : The length of the matched substring.

c : The next literal character that follows the matched substring (for the next symbol).

Whenever the compressor detects a sequence already seen, it emits (d, l, c) rather than the actual substring. Mathematically, if $s_j = s_{j-d}$ for a match of length l, the LZ77 compressor outputs:

$$\left(d, l, s_{j+l}\right) \tag{2.2}$$

These triple references what was already encountered, thus reducing redundancy.

2.2.1.2 Huffman Entropy Coding

After LZ77, Huffman coding is applied to encode the tokens (distances, lengths, and literal characters) using shorter bit patterns for more frequent symbols and longer bit patterns for less frequent ones. If $p(x_i)$ is the probability of symbol x_i, Huffman coding aims to produce a code of length approximately $-\log_2 p(x_i)$ bits for each symbol. The Huffman algorithm ensures that no code is a prefix of another (prefix-free property), facilitating optimal or near-optimal entropy encoding under certain assumptions.

2.2.2 bzip2

The bzip2 algorithm uses the Burrows-Wheeler Transform (BWT), Move-to-Front (MTF) transformation, Run-length Encoding (RLE), and Huffman encoding. BWT rearranges data in format that makes compression better, and Huffman encoding compresses the data very effectively.

bzip2 has better compression ratio than DEFLATE, but it has higher computational overhead with slower compression speeds, which makes it less suitable for real-time applications or environments with limited computational resources (Mahoney, 2012).

2.2.2.1 Burrows–Wheeler Transform (BWT)

The BWT rearranges a block of input data S into a form that is more amenable to compression by grouping repeated characters. Given a block S of length n, the BWT outputs a transformed block $BWT(S)$. Briefly:

a. Construct all n rotations of S.
b. Sort these rotations lexicographically.
c. The last column of this sorted matrix is taken as $BWT(S)$, along with the index of the original string in the sorted list for use in decompression.

2.2.2.2 Move-to-Front (MTF) Coding

After BWT, repeated patterns tend to cluster. MTF takes advantage of this by maintaining a list of symbols and moving any symbol to the front upon its use. If L is our symbol list and s is the symbol encountered, the MTF-encoded output is the index of s in L at the time it appears:

$$\mathrm{MTF}(s, L) = \mathrm{indexOf}(s, L) \tag{2.3}$$

Then s is moved to the front of L. Common symbols thus often produce small index values, which are more easily compressed.

2.2.2.3 Run-Length Encoding (RLE)

For consecutive repeated symbols in the MTF output, bzip2 applies an RLE step to compress runs of the same value.

2.2.2.4 Huffman Coding

Finally, bzip2 encodes the output using Huffman coding, assigning variable-length bit patterns to symbol frequencies.

2.2.3 LZMA

The Lempel-Ziv-Markov chain Algorithm (LZMA) is a dictionary-based method that combines range encoding to achieve very high compression ratios. The dictionary mechanism is similar in spirit to LZ77, but LZMA refines how matches are searched and encoded.

LZMA takes advantage of past data to encode repeating patterns very efficiently. Although LZMA gives great compression, it has very high computational usage, so it is not practical for time-critical situations or environment that have limited resources (Mahoney, 2012).

2.2.3.1 Dictionary Match Search

Like LZ77, LZMA maintains a sliding dictionary window of recent data. If a substring $(s_k, \dots, s_{k+m-1})$ reappears, it references that substring. However, it employs more sophisticated match-finding techniques (e.g., binary tree or hash chain) to speed up searching for repeats.

2.2.3.2 Range Encoding

LZMA replaces Huffman coding with range encoding, which represents probabilities by continuously refining a range $[low, high)$. Symbols with higher probability occupy a larger portion of the range, allowing more efficient representation of data. If $p(s_i)$ is the probability of symbol s_i, the encoder narrows the interval based on $p(s_i)$. Mathematically, for a symbol s_i,

$$\begin{aligned} \text{range}_{new} &= \text{range} \times p(s_i), \\ \text{low}_{new} &= \text{low} + \textstyle\sum_{j<i} \left(\text{range} \times p(s_j)\right) \end{aligned} \tag{2.4}$$

This process continues iteratively for each symbol

2.2.4 Zstandard (Zstd)

Zstandard (Zstd) algorithm is a modern lossless compression algorithm that was developed by Facebook. It is using dictionary compression techniques combined with entropy encoding methods, offering a flexible balance between speed and compression ratio by adjusting compression levels.

a. Dictionary Builder: At higher levels, Zstd can learn an optimal dictionary for a specific dataset, improving compression ratio for small or homogeneous data.
b. Entropy Encoding: Zstd uses FSE (Finite State Entropy) or Huff0, both of which compress symbols based on their statistical frequencies in a single pass. If the probability of a symbol s_i is $p(s_i)$, the code lengths approach $-\log_2 p(s_i)$, like Huffman.
c. Adjustable Levels: Zstd supports numerous compression levels, allowing users to trade off speed for higher ratio or vice versa.

Zstd is popular among big data and cloud storage due to its compression speed and compression ratio (Collet & Kucherawy, 2019).

2.2.5 LZ4

The LZ4 algorithm focuses on speed rather than maximum compression ratio. It is applying LZ77 algorithm with optimized way to do very fast compression and decompression. It uses an LZ77-style dictionary approach but aggressively optimized for real-time operation:

a. It maintains a hash table of recent data blocks; upon detecting a match, it replaces a substring with a back-reference (d, l).
b. The compressed stream is minimal in overhead, focusing on making both compression and decompression extremely fast.

LZ4 is used a lot in situations where compress and decompress time is more important than the highest compress ratio (Bartik, Ubik and Kubalík, 2015).

2.2.6 Brotli

Brotli algorithm was developed by Google mainly for web and text compression. It combines LZ77-style dictionary compression, Huffman coding, and second-order context modeling to optimize text and web content compression.

a. Window-Based Dictionary: Brotli maintains a sliding window for repeated pattern detection, referencing repeated substrings similarly to LZ77.
b. Second-Order Context Modeling: Brotli attempts to predict upcoming symbols by using context from previously decoded symbols.
c. Huffman: Once repeated sequences are identified, Brotli employs a Huffman-based algorithm to assign variable-length codes to repeated patterns.

Its strength is handling repetitive text data efficiently, which is very common in web technologies particularly for text-based data, such as HTML, CSS, and JavaScript, where repeated substrings are abundant (Cover & Hart, 1967).

2.3 Machine Learning Approaches for Data Compression

Recent research indicates that using machine learning can greatly simplify the difficulty with data compression by adjusting the algorithm selection automatically to the characteristics of the file, including entropy, byte distributions, and file metadata. There is a remarkable advancement from the static heuristic-based selections in these adaptive data driven methods.

2.3.1 Decision Tree

Decision Tree is a supervised machine learning algorithm used for classifying and regression. It works by splitting the data into smaller subsets again and again based on feature value. Each split is chosen using metrics such as Gini impurity or information gain, which show how good the split makes the data different. Decision Tree is very easy to understand but often overfit when dataset is complex, unless controlled with techniques like pruning or ensemble methods such as Random Forest and XGBoost.

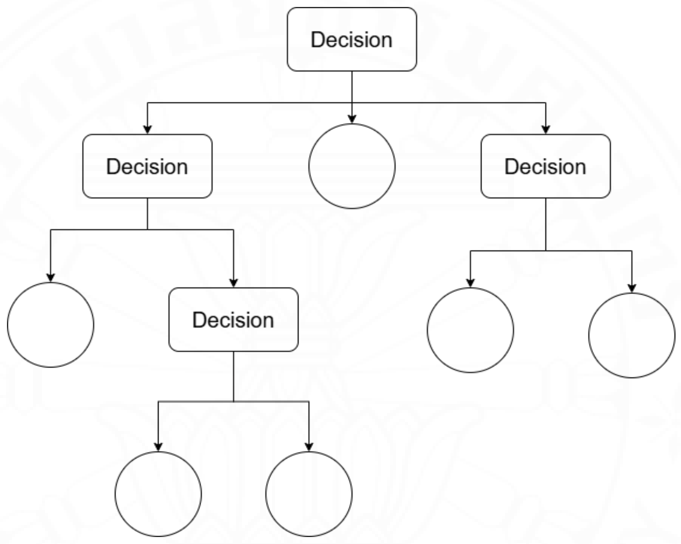

Figure 2.1 Decision Tree

2.3.2 Random Forest

Random Forest is made of many decision trees that run in parallel, using random samples of data. It combines decisions from each tree using majority vote, improving predictive accuracy and reducing variance. Random Forest works well even when data are noisy or have missing values. It also gives feature importance, show which input is most useful for prediction. But it can be slow if it has too many trees or a big dataset (Biau & Scornet, 2016).

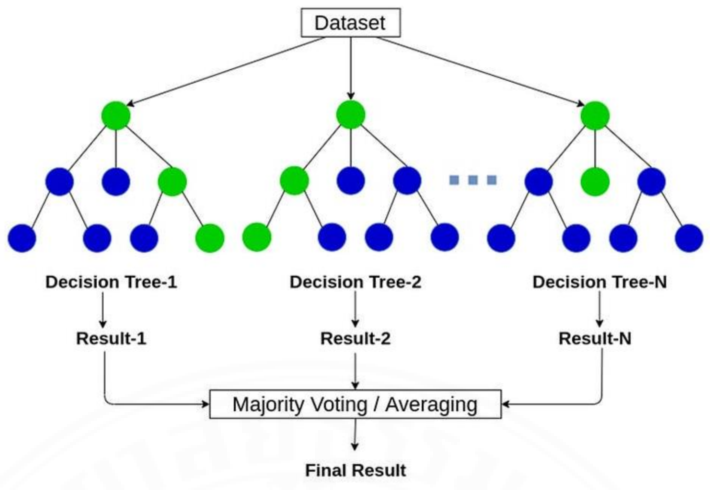

Figure 2.2 Random Forest

2.3.3 XGBoost

Extreme Gradient Boosting (XGBoost) is a very strong ensemble learning algorithm that uses gradient-boosted decision trees. It builds decision tree step by step, reducing residual errors in each iteration. XGBoost is known for its efficiency, scalability, and accuracy, especially on structured datasets. It can handle large and complex feature interactions (Chen & Guestrin, 2016).

2.3.3.1 Ensemble of Decision Trees

XGBoost iteratively adds new trees to reduce the error of prior trees.

2.3.3.2 Objective Function

If y is the real target and $\hat{y}_i$ is the prediction at the i-th iteration, XGBoost updates the model by:

$$\hat{y}_{i+1} = \hat{y}_i + \eta \cdot f_i(\mathbf{x}), \tag{2.5}$$

where f_i is a newly added decision tree, and η is the learning rate. The final prediction is the sum of all trees' outputs.

2.3.3.3 Regularization

XGBoost includes regularization terms on tree complexity to prevent overfitting and encourage generalization.

2.3.4 Support Vector Machine (SVM)

Support Vector Machines classify data by finding hyperplanes that maximize separation margin between classes. It uses kernel functions to capture nonlinear relationships in data. However, SVM can require substantial computational resources when working with big datasets, making it not the best option for real-time compression predictions on large-scale data (Guido, Ferrisi, Lofaro and Conforti, 2024).

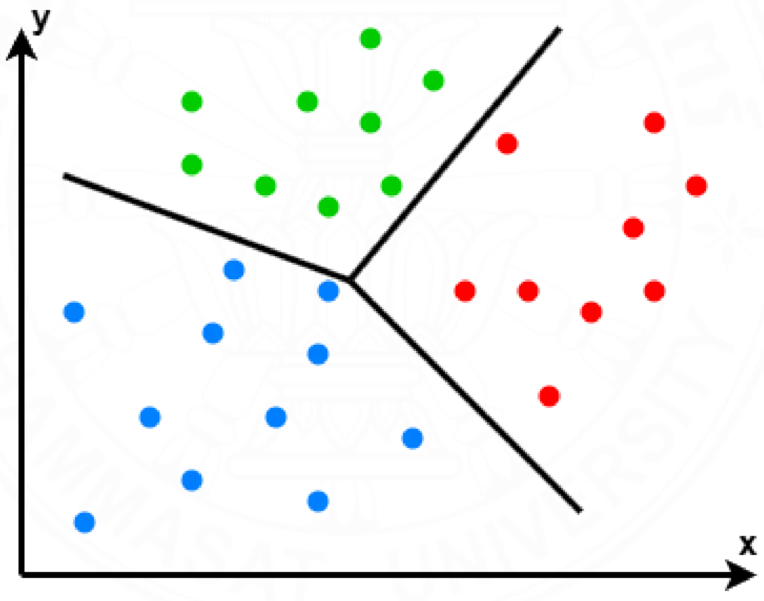

Figure 2.3 Support Vector Machine

2.3.5 Logistic Regression

Logistic Regression model probabilistic outcomes using linear combination of input features, which makes it interpretable and fast to compute. However, because it assumes data is linearly separable, it is not as useful when dealing with high-dimensional or complex datasets that are typical in compression selection tasks.

Let $\mathbf{x} \in \mathbb{R}^d$ be an input vector (file features), then the model predicts:

$$\hat{p} = \sigma(\mathbf{w}^T\mathbf{x} + b), \text{ where } \sigma(z) = \frac{1}{1+e^{-z}} \tag{2.6}$$

Although straightforward and easily interpretable, logistic regression struggles to capture non-linear relationships unless extended with polynomial or other feature transformations (James, Witten, Hastie and Tibshirani, 2021).

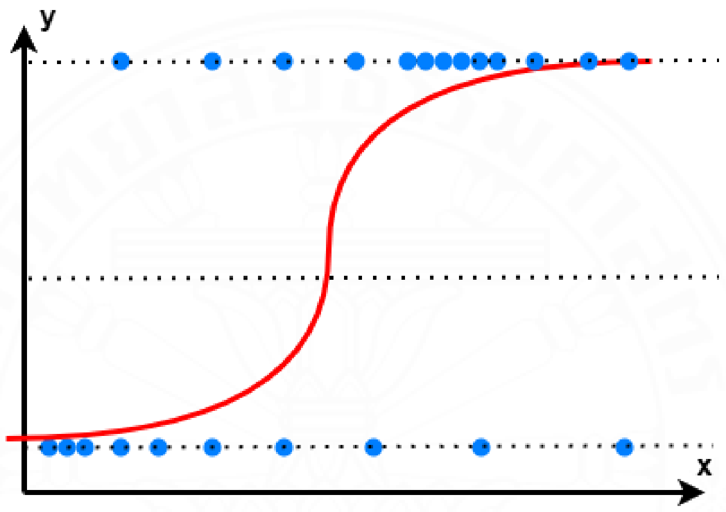

Figure 2.4 Logistic Regression

2.3.6 K-Nearest Neighbors (KNN)

K-Nearest Neighbors (KNN) makes predictions by looking at the k closest training examples in feature space. The class is determined by majority vote among these neighbors. For a query point **x**:

$$\hat{y}(\mathbf{x}) = \text{majority}\{\, y_i \mid \mathbf{x}_i \in \mathcal{N}_k(\mathbf{x})\}, \tag{2.7}$$

where $\mathcal{N}_k(\mathbf{x})$ is the set of k closest points to **x** under a distance metric (often Euclidean distance).

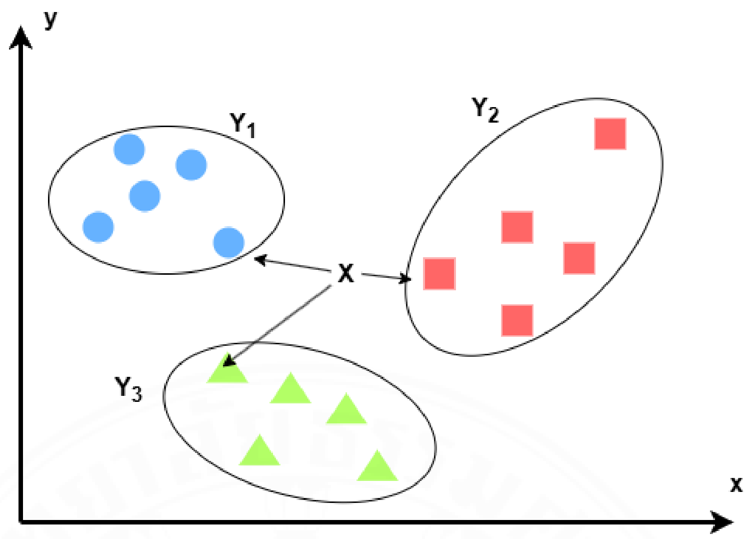

Figure 2.5 K-Nearest Neighbors

It is simple and intuitive, but its prediction speed becomes but its prediction speed degrades with large datasets, making it less practical for real-time or large-scale compression tasks (Cover & Hart, 1967).

2.4 Model Comparison

This section provides a comparative overview of both compression and machine learning algorithms used in the study.

Table 2.1 Comparison of Lossless Compression Algorithms

Algorithm	Compression Ratio	Compression Speed	Decompression Speed	Computational Cost
zlib	Medium	High	High	Low
bzip2	High	Medium	Medium	Medium
LZMA	Very High	Low	Medium	High
Zstd	High	High	High	Medium
LZ4	Low	Very High	Very High	Very Low
Brotli	High	Medium	High	Medium

The compression choice depends heavily on application-specific factors such as speed and compression efficiency.

Table 2.2 Comparison of Machine Learning Algorithms

Algorithm	Accuracy	Interpretability	Computational Cost	Training Speed
XGBoost	Very High	Medium	High	Medium
Random Forest	High	Medium	Medium	Medium
Support Vector Machine	High	Low	High	Low
Logistic Regression	Medium	High	Low	High
Decision Trees	Medium	High	Low	High
K-Nearest Neighbors	Medium	Medium	Medium	High

2.5 Related Works

This research reviewed 3 related works that use machine learning techniques to select compression algorithms. The following are the descriptions of their works.

2.5.1 Using Machine Learning to Predict Effective Compression Algorithms for Heterogeneous Datasets (Burtchell and Burtscher, 2024)

Burtchell and Burtscher (2024) proposed MLcomp, a method that uses a Random Forest classifier to automatically predict optimal compression algorithms for heterogeneous datasets. Their model uses the compression ratio from short preliminary runs on representative file as input features for predicting the most effective compression algorithms among numerous possible combinations. They reported achieving approximate 98% of compression performance obtain by exhaustive evaluating all possible algorithms. Despite these results, MLcomp methodology only focus on optimizing compression ratio, neglecting computational runtime or resource usage considerations during compression prediction. This independent study clearly addresses this limitation by including both compression effectiveness and computational runtime into predictive criteria.

2.5.2 Adaptive Compression Algorithm Selection Using LSTM Network in Column-oriented Database (Jin et al., 2019)

Jin et al. (2019) has developed adaptive methods using a Long Short-Term Memory (LSTM) neural network models for predicting the optimal compression algorithm specifically tailored to column-oriented databases. Their methodology involves training LSTM network on sequences of raw data byte extract from database columns, enabling models to capture intrinsic data patterns and predicting the most efficient compression algorithm for each data block. Their result showed prediction accuracy around 64% in training set and approximate 55% in heterogeneous testing datasets. Despite moderate predictive performance, the LSTM-driven adaptive selection consistently yielded better compression outcome compared to fixed heuristic method typically employed in database systems. Their study highlights both the potential and challenges of applying ML-based selection approach, particularly emphasizing complexity and computational overhead associated with deep learning method for real-time predictions scenario.

Unlike this deep learning approach for databases, this independent study focuses on a broader range of heterogeneous file types and uses computationally lighter machine learning models to ensure minimal prediction latency.

2.5.3 Compression Selection for Columnar Data using Machine-Learning (Larsen and Persson, 2023)

Larsen and Persson (2023) have introduced a machine learning-driven framework that uses XGBoost algorithm for automatically selecting most cost-effective compression algorithm and encoding combination specific tailored columnar database. Their research utilizes carefully design cost functions that integrate three critical factors: compression ratio, compression time and decompression time. This enables the system to optimize compression not just data size but explicitly balance storage efficiency and processing overhead. Using extensive feature engineering based on real-world IoT telemetry data store in ClickHouse database, their model achieves impressive predictive accuracy approximately 99% on their test dataset, with around 90% accuracy when predicting compression strategies for unseen data columns. Furthermore, deployment of their machine learning recommendations significant enhanced system performance, achieved roughly 95% increase in compression speed and nearly 60% improvement in decompression speed. However, this improvement came at the expense of storage efficiency, resulting in about 66% reduction in compression ratio compared highest possible compression scenario. Their study highlights strength and potential trade-off involved when applying machine learning models for adaptive compression decisions, particularly emphasizing computational efficiency and feature relevance maintaining high predictive accuracy in column-oriented storage environment.

While their work balances compression ratio with both compression and decompression time, this study prioritizes achieving the maximum compression ratio (lowest size) within a strict upload time budget (30 seconds), a constraint more relevant to file transfer and storage pipelines.

CHAPTER 3
METHODOLOGY

This chapter details the experimental framework for predicting the optimal compression algorithm for a given file. The process spans data collection, feature extraction, compression benchmarking, labeling of best algorithms per file, feature selection, and machine learning model training. Each step is described in sequence, with a particular focus on the features extracted from files and the rationale behind them.

Figure 3.1 shows a conceptual flow diagram representing the methodological pipeline from data gathering to machine learning-based decision-making:

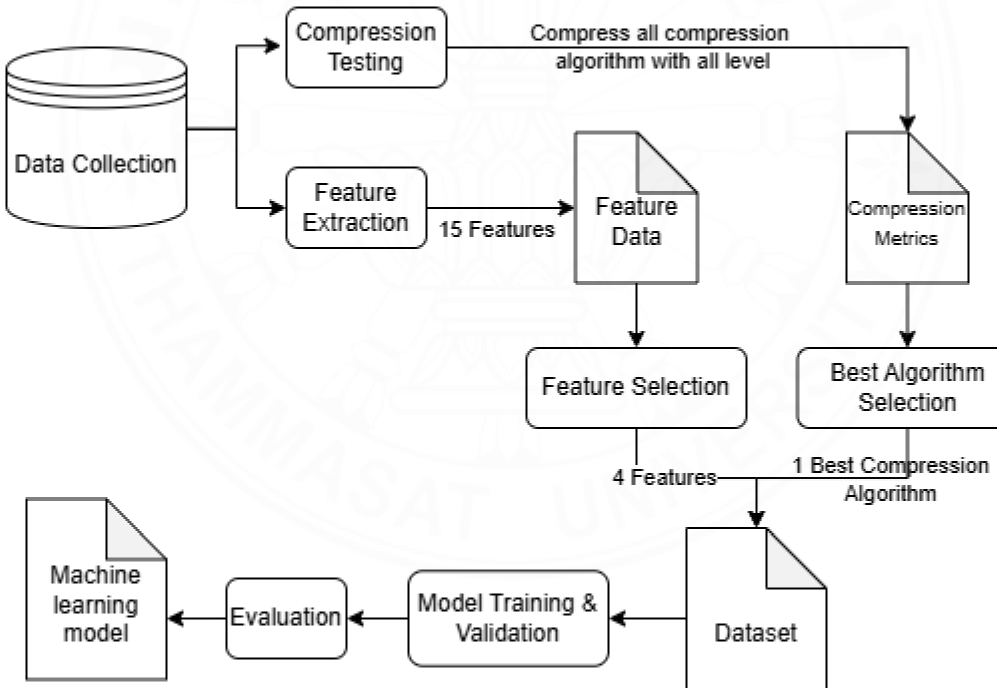

Figure 3.1 Overview of the methodology

Figure 3.1 illustrates the complete methodology pipeline. The process begins with Data Collection. From this data, two parallel processes are initiated: Feature Extraction, which identifies 15 initial features to create "Feature Data", and

Compression Testing, which runs all compression algorithms on the files to generate "Compression Metrics".

These metrics are then used for Best Algorithm Selection to determine the single best algorithm for each file. This best algorithm label is combined with the "Feature Data" and passed to Feature Selection, which narrows the features down to the 4 most predictive ones. This creates the final Dataset, which is used for Model Training & Validation. Finally, the trained model undergoes Evaluation to produce the deployable Machine learning model.

3.1 Data Collection

A large dataset of 120,263 files, denoted as set $\mathscr{F} = \{f_1, f_2, \ldots, f_n\}$ with n = 120,263 was compiled to capture diverse real-world data. These files cover a broad range of types, including plain text, structured documents, images, and other common formats. Each file underwent an integrity check to ensure it was not corrupted or incomplete, as corrupted data could bias the results. Basic metadata recorded for each file included the file name, file size ($|f_i|$), file extension, and file type (a coarse category label). These metadata fields are summarized in Table 3.1 below, which presents a clear overview of the information collected during data preparation.

Table 3.1 Summary of basic file metadata collected during data preparation

Metadata	Description
File name	The original name of the file (used for identification).
File size	The size of the file in bytes, denoted as $\lvert f_i \rvert$.
File extension	The file’s extension (e.g., .txt, .jpg) indicating format.
File type	A coarse category of the file (e.g., Text, Image, Archive) based on its format or content.

By assembling a wide variety of file types and sizes (ranging from ~1 KB up to 2 GB), the study ensures that the subsequent analysis reflects realistic and heterogeneous scenarios. This diversity is important because compression effectiveness can vary greatly with file structure and content.

3.2 Feature Extraction

For every file $f_i \in \mathcal{F}$, a feature vector x_i was generated to capture key characteristics that might influence compressibility. Formally, each file is transformed into $x_i = (x_{i1}, x_{i2}, \ldots, x_{id})$, where d is the total number of extracted features. In total, 15 numeric features were extracted from each file. These features encompass basic metadata, statistical properties of the byte content, and measures of redundancy or structure in the file. Before modeling, all feature values were scaled to a consistent range because their magnitudes differ. The description of all extracted features is shown below:

3.2.1 File Size (N)

The total number of bytes in the file. This is a basic attribute given by the length of the byte sequence. This feature simply captures the file's size.

3.2.2 Entropy

The Shannon entropy of the file's byte-value distribution, measuring the randomness or unpredictability of bytes. We compute this by treating the file as a sequence of symbols (0–255) and calculating the entropy of their frequency distribution. Let p_i be the probability of byte value i (estimated as the frequency of i divided by N). The entropy is then:

$$H = -\sum_{i=0}^{255} p_i log_2 p_i \tag{3.1}$$

where the sum is taken over all byte values that occur in the file. A higher H (up to 8 bits for 256 uniform symbols) indicates more uniform and random byte content, whereas lower values indicate more structured or repetitive content.

3.2.3 Chi-Square (χ^2)

A chi-square goodness-of-fit statistic comparing the file's byte frequency distribution to a uniform distribution. It is defined as:

$$\chi^2 = \sum_{i=0}^{255} \frac{(count_i - N/256)^2}{N/256} \tag{3.2}$$

where $count_i$ is the observed frequency of byte value i and $N/256$ is the expected frequency for a uniform distribution (with N total bytes and 256 possible values). A larger χ^2 indicates the byte frequencies deviate more from uniform. Therefore, certain byte values appear often than expected by chance.

3.2.4 Byte Variance

The statistical variance of the byte values interpreted as numerical 0–255. This feature measures the spread of byte values around their mean. If $b = \frac{1}{N}\sum_{i=1}^{N} b_i$is the mean byte value, the variance is:

$$\sigma^2 = \frac{1}{N}\sum_{i=1}^{N}\left(b_i - \bar{b}\right)^2 \tag{3.3}$$

Higher variance means the byte values are more widely distributed across the 0–255 range, whereas low variance means the bytes cluster around a certain value.

3.2.5 Byte Kurtosis

This feature measures how “peaky/heavy-tailed” the file’s byte-value distribution is. If ByteKurtosis is high, this indicates a few byte values dominate (often more compressible). If it is low, bytes are spread more evenly (usually less obvious redundancy). The formula can be explained as:

$$\text{ByteKurtosis} = \frac{\frac{1}{N}\sum_{j=1}^{N}\left(b_j - \mu\right)^4}{\left(\frac{1}{N}\sum_{j=1}^{N}\left(b_j - \mu\right)^2\right)^2} \tag{3.4}$$

where:

N = number of bytes in the file

b_j = value of the j-th byte (0–255)

$\mu = \frac{1}{N}\sum_{j=1}^{N} b_j$ (mean byte value)

3.2.6 Byte Standard Deviation

The standard deviation of byte values, defined as the square root of the byte variance. It is given by:

$$\sigma = \sqrt{\frac{1}{N}\sum_{i=1}^{N}\left(b_i - \bar{b}\right)^2} \qquad (3.5)$$

This provides the dispersion of byte values in the same units as the byte values themselves.

3.2.7 Longest Repeated Byte Sequence

The length of the longest run of identical bytes in the file (also referred to in code as the "LongestRepeatedSubstring"). This feature captures the longest consecutive sequence of the same byte value. Formally, if we define a *run* as a maximal substring of the form $b_i = b_{i+1} = \cdots = b_{i+\ell-1}$, then this feature is the maximum length ℓ over all such runs in the file:

$$L_{max} = max\,\{\ell \mid \exists\, i\ such\ that\ b_i = b_{i+1} = \cdots = b_{i+\ell-1}\} \qquad (3.6)$$

A larger value indicates that some byte is repeated many times in a row (e.g. a long sequence of zeros).

3.2.8 Average Repeat Length

The average length of repeated byte runs in the file. We compute the lengths of all consecutive byte runs and then take the average of those lengths that are greater than 1. Let $R_1, R_2, \ldots, R_k$ be the lengths of all runs of identical bytes (with $R_j \geq 2$ for each, i.e. we consider only runs of length at least 2). Then the feature is

$$L_{avg} = \frac{1}{k}\sum_{j=1}^{k} R_j \qquad (3.7)$$

In case there is at least one repeated run ($k > 0$). If the file contains no consecutive repeated bytes (i.e. all runs are of length 1), we define this feature as 0. A higher L_{avg} means that on average, repeating sequences tend to be longer.

3.2.9 Dictionary Fit

The number of unique byte values present in the file. This essentially is |{unique bytes} | and reflects how large a "dictionary" an algorithm would need to encode the file's content. A smaller unique byte set (for instance, a file that contains only 10 distinct byte values) often compresses better than a file using the full 0–255 range of bytes.

3.2.10 ASCII Ratio

The proportion of bytes in the file that fall within the ASCII printable character range (byte values 32 through 126 inclusive). This feature gauges how "text-like" the file is. It is computed as:

$$AsciiRatio = \frac{\#\{i{:}32 \leq b_i \leq 126\}}{N} \tag{3.8}$$

the count of bytes in the ASCII printable range divided by the file size N. The ratio approaches 1 for plain text files (comprised mostly of readable characters) and 0 for data with mostly non-printable bytes (such as compressed or encrypted files).

3.2.11 File Type

A binary indicator derived from content, used to roughly distinguish text from binary files. The FileType will be set to 1 if the AsciiRatio > 0.8 (meaning the file is likely text-heavy) and 0 otherwise. This feature provides the model with a simple categorical flag about the file's nature.

3.2.12 Average Line Length

The average number of characters per line when interpreting the file as text. To calculate this, the file's bytes are decoded as UTF-8 text (ignoring decoding errors), split into lines on newline characters, and the lengths of these lines are averaged. If $L_1, L_2, \dots, L_m$ are the lines obtained, then it is defined as:

$$AvgLineLength = \frac{1}{m}\sum_{j=1}^{m}\left|L_j\right| \tag{3.9}$$

provided $m > 0$. (If the file cannot be decoded into any lines, we define this value as 0.) This feature is meaningful for text files, indicating typical line length, and is 0 for files that are not interpretable as text.

3.2.13 Unique Bytes

The number of distinct byte values in the file. This is effectively the same as DictionaryFit. It was extracted as a separate feature but duplicates the information of DictionaryFit. A lower UniqueBytes count means the file's content is composed of a limited alphabet of bytes, which can be advantageous for certain compression algorithms.

3.2.14 N-gram Redundancy

An approximate measure of repeated byte patterns, using 2-byte sequences as a default. This is computed by sampling pairs of consecutive bytes (2-grams) throughout the file and finding the most frequent 2-byte sequence. The feature value is the frequency of that most common 2-byte pattern divided by the total number of sampled pairs. A higher NgramRedundancy means a particular byte pair occurs very often relative to file length, indicating repetitive structure that could be exploited by compression.

3.2.15 Proxy Compression Ratio

A simple proxy for the file's compressibility, defined as the ratio of the original size to a hypothetical slightly larger size. We calculate it as:

$$ProxyCompressRatio = \frac{N}{N+1} \tag{3.10}$$

This formula yields a value very close to 1 for any non-trivial file (e.g. 0.999 for $N = 999$). In practice, this proxy does not depend on content and varies only with N (smaller files get slightly lower values). It was used as a placeholder approximation of compressibility – a higher value (closer to 1) would intuitively correspond to files that are not easily compressible, though here it is essentially always near 1 except for very small files.

3.3 Compression Testing

In the next phase, each file f_i was subjected to compression by a set of candidate algorithms to observe compression performance. We selected six widely used lossless compression algorithms: zlib, Bzip2, LZMA, Zstandard (Zstd), LZ4, and Brotli. Each algorithm was applied to every file at multiple compression levels or settings (for example, level 1 through 9 for those that support levels, or fast vs. slow modes). This exhaustive benchmarking yields empirical data on how well each method compresses each file. For a given file f_i and compression algorithm instance a_j (where j might represent a specific algorithm at a certain compression level), we recorded two primary metrics:

- Time_{ij}: the compression time in seconds for algorithm a_j on file f_i. This measures how long the algorithm took to compress the file (since some algorithms trade speed for ratio).
- Size_{ij}: the resulting compressed file size in bytes when using a_j on f_i.

From these we derive the compression ratio ρ_{ij} defined as:

$$\rho_{ij} = \frac{\text{Size}_{ij}}{|f_i|} \tag{3.11}$$

The compressed size divided by the original file size. A ratio $\rho < 1.0$ indicates that compression was effective (the file became smaller), whereas $\rho = 1.0$ means no size reduction, and $\rho > 1.0$ would mean the output is actually larger, this indicates compression failed to reduce size, which can happen with already compressed or very random data. Along with ratio, the raw compressed size and time are important for evaluating trade-offs. We carried out this compression testing for all files across all chosen algorithms/levels, producing a comprehensive set C = $\{(f_i, a_j, \text{Time}_{ij}, \rho_{ij})$ for $1 \leq i \leq n$, $1 \leq j \leq m\}$ where m is the total number of algorithm configurations tested. This data allowed analysis of how different algorithms perform on the same file and highlighted the variation in outcomes. For example, some algorithms (like LZ4) are very fast but may not compress as tightly, yielding higher ρ (closer to 1), while others (like Brotli) produce very low ρ (smaller size) but at the cost of longer Time.

It was observed that there is a clear trade-off: methods like LZ4 or Zstd in fast mode execute in fractions of a second but sometimes produce larger outputs, whereas methods like Brotli or LZMA at max settings yield the smallest sizes but can take significantly longer. Understanding these trade-offs was essential for defining what optimal means in context and ensuring our automatic selection does not choose an impractical solution, i.e., one that saves only a few bytes at the cost of an extremely long runtime.

3.4 Best Algorithm Selection

After gathering compression results, we needed to determine, for each file, which algorithm was considered the *best* (ground truth optimal) under practical constraints. A rule-based selection procedure was applied to each file's results to choose its optimal algorithm a_i*:

a. Time Constraint: Any algorithm run that took more than 30 seconds on a given file was disqualified. Formally, for each f_i, we discarded all a_j such that $\text{Time}_{ij} > 30$ seconds. The 30-second threshold was chosen based on practical system considerations. In our real-world use case, users often upload up to 5–6 files simultaneously at most, and compression is followed by an encryption step. To maintain responsiveness, the total additional time introduced by the compression stage should not exceed 3 minutes (180 seconds). Therefore, we decided that each individual file must not require more than 30 seconds of processing. By enforcing this cutoff, we ensure that extremely slow algorithms are not labeled as optimal, even if they achieve slightly better compression ratios, since their runtime would be impractical in an operational setting.

b. Outlier Exclusion: We also removed extreme outlier runs in terms of compression time. For each file, we examined the distribution of compression times {Time_{ij}} across algorithms and flagged any that were abnormally high compared to the others. Specifically, we used the interquartile range (IQR) rule: any algorithm whose Time fell above Q3 + 3(Q3–Q1) was excluded. This guards against algorithms that, while not exceeding 30s outright, are still disproportionately slow outliers for that file. The intuition is that if one algorithm takes much longer than the rest on the same data (perhaps due to some pathological case or inefficiency), it is not a practical choice even if our fixed threshold did not catch it. Removing such outliers yields a set of feasible algorithms A^*_i for each file f_i.

c. Effective Compression Only: We ensure that the algorithm really achieves compression. Any result where the compressed size was larger than the original ($\rho_{ij} \geq 1.0$) is disregarded. In other words, we only consider algorithms that produce $\rho < 1$ for that file. This avoids ever labeling a method as best if it did even compress the data. After this step, for each file we have a filtered set of viable algorithm options that ran within time limits and produced a smaller output.

After applying (a), (b), and (c), each file f_i has a subset A_i of algorithms that passed all criteria. From this subset, we select the algorithm with the lowest compression ratio ρ:

$$a_i^* = \arg\min_{a_j \in A_i} \rho_{ij} \tag{3.12}$$

In summary, a_i is the algorithm that achieved the highest compression (greatest size reduction) on file f_i. Ties are rare but if they occur, one could choose the faster algorithm among the tie, though in our case the continuous nature of ratio usually yields a unique minimum. Each file is thus assigned a single "optimal" algorithm label a_i. This algorithm is considered the ground-truth best choice for that file in the context of our study. It represents the ideal outcome we want a predictive model to achieve. It is worth noting that the best algorithm here is defined purely by compression ratio after filtering out unrealistic options. This implies we favor maximum compression as long as it is within the time budget. This selection method yielded a mapping from each file to its optimal compression algorithm. This mapping becomes the target variable for the machine learning stage.

3.5 Feature Selection

Initially, we considered all 15 features described in Section 3.2 for use in the model. However, not all features provide unique or useful information; some may be redundant or contribute very little to predictive accuracy. Using too many features can also risk overfitting and slow down model training and operation. Therefore, a Sequential Feature Selection (SFS) procedure was employed to reduce the feature set to the most informative subset. Sequential Feature Selection is a greedy algorithm that builds a feature subset step by step:

a. Initialization: Start with no features selected (an empty set S = Ø).
b. Iterative Addition: Iteratively add one feature at a time, choosing the feature that, when combined with the currently selected set S, yields the highest improvement in model performance (typically measured by validation accuracy in our case). That is, in each round, we pick the feature that most boosts the predictive power alongside those already chosen.
c. Stopping Criterion: Continue adding features until adding any remaining feature does not appreciably improve performance, or until a predefined number of features is reached.

Formally, given a universal feature set $\{1,2,\dots,d\}$, SFS attempts to identify:

$$S^* = \arg \max_{S \subseteq \{1,2,\dots,d\}} \Phi\,(S) \tag{3.13}$$

where $\Phi(S)$ is the performance metric for a model built using the feature indices in S. Commonly, Φ is the average classification accuracy across a validation set. By pruning away less informative features, SFS not only reduces the risk of overfitting but also improves computational efficiency, both during training and at inference time. This step is crucial because large-scale datasets might originally have included dozens of potential feature dimensions, many of which offer minimal incremental benefit.

Through this process, we found that the first few features added contributed the most to accuracy, and additional features after a point gave negligible gains. Ultimately, four features were selected: FileSize, AvgLineLength, AsciiRatio, ByteKurtosis.

3.6 Final Dataset

After the feature selection process, the dataset was reduced into a more compact and structured form which is used for model training. Each record in this final dataset corresponds to a single file and contains the essential information needed for supervised learning. The attributes kept are:

a. Filename: the identifier of the file, which allows traceability but not used as predictive input.
b. FileSize: numerical value representing the total bytes in the file.
c. AvgLineLength: the average number of characters per line, capturing structure of text data.
d. AsciiRatio: proportion of printable ASCII characters in the file, reflecting whether the content is mainly textual or binary.
e. ByteKurtosis: the kurtosis of byte distribution, measuring if a few values dominate or bytes are evenly spread.
f. Optimal Algorithm Label: categorical value indicating the best compression algorithm chosen for that file under the criteria explained in Section 3.4.

Thus, each row in the dataset can be represented as d_i = {Filename, FileSize, AvgLineLength, AsciiRatio, ByteKurtosis, BestAlgo} where d_i is the i-th record and BestAlgo is the ground-truth class label for supervised training.

The features were normalized before training to ensure that large values such as file size did not dominate the learning process. Standardization places them on a comparable scale. The final dataset therefore represents a balanced and concise summary: the most informative four numerical features plus the assigned algorithm label. This structure provides enough discriminatory power to the model while avoiding redundant or noisy dimensions.

In summary, the final dataset is both simple and expressive. It captures the essential factors influencing compression performance in four numerical descriptors, and it pairs them with the optimal algorithm outcome. This dataset is the foundation for the machine learning phase that follows in Section 3.7, where classifiers are trained to map features to algorithm labels. It ensures that the training focuses only on meaningful information and avoids unnecessary complexity, leading to more efficient and accurate prediction.

3.7 Machine Learning Model Training

With each file now represented by a feature vector (using the reduced feature set S) and a known optimal algorithm label a_i, we set up a supervised learning task. This is a multi-class classification problem: the model must learn to map a file's features to the correct compression algorithm. There are six possible algorithm classes in our case (Zlib, Bzip2, LZMA, Zstd, LZ4, Brotli). We prepared the final dataset $D = \{(x_i(S), y_i) \mid 1 \leq i \leq n\}$, where $x_i(S)$ is the feature vector of file f_i restricted to the selected feature subset and y_i is the class label (the index of the optimal algorithm for file f_i).

3.7.1 Classification Setup

A variety of candidate classification algorithms were explored to find the best predictor for this problem. We evaluated common machine learning models including Decision Tree, Random Forest, Support Vector Machine (SVM), k-Nearest Neighbors (k-NN), Logistic Regression, and XGBoost (Extreme Gradient Boosting). We trained each model on the training dataset and assessed their accuracy in predicting the correct algorithm class.

3.7.2 Hyperparameter Tuning and Validation

To fairly compare models and tune them, we used cross-validation. The dataset was split into k folds. Here we used 5-fold cross-validation in most cases, and model performance was averaged across different splits to ensure it generalizes. We also performed hyperparameter optimization for each model. We searched for the hyperparameter combination that gave the highest validation accuracy. The use of cross-validation provided an estimate of how well each model would perform on unseen data, mitigating overfitting during the tuning process. Formally, if θ represents a set of hyperparameters for a given model, we evaluate an average accuracy:

$$\mathrm{Acc}(\theta) = \frac{1}{k}\sum_{j=1}^{k} \mathrm{Accuracy}\left(\theta; D_j\right) \tag{3.14}$$

where D_j is the j-th fold used as a validation set. We chose the θ that maximize this Acc(θ). Additionally, this process helped decide which type of model is inherently best for our task.

3.7.3 Model Selection

After training and tuning, we found that an XGBoost classifier performed the best in terms of accuracy in predicting the optimal compression algorithm, outshining the other approaches. The XGBoost model was able to reliably learn the relationship between our file features and the best algorithm choice. It achieved the highest cross-validation accuracy, meaning it most often predicted the correct algorithm label for files in the validation folds. This model benefits from the ensemble of trees, capturing non-linear interactions among features. Moreover, XGBoost provides feature importance scores, which aligned with our expectations.

The chosen XGBoost model was then trained on the entire training dataset, using the selected features to finalize it. The model's hyperparameters were tuned for a balance of accuracy and complexity to avoid overfitting. Finally, this model was saved for integration. It will be embedded into a C# application to automatically decide compression algorithms before encryption in a real system. Therefore, we ensure the research outcomes can be applied in practice, compressing files on-the-fly with the learned optimal choices.

3.8 Summary

In summary, the methodology involved gathering a rich dataset of files, extracting a diverse set of features to characterize each file's content and structure, determining the ground-truth best compression algorithm for each file through comprehensive testing, and then training a machine learning model to predict that choice using only the file’s features. Careful feature selection and model tuning were key to achieving high prediction accuracy. The result is an adaptive compression decision system that aims to yield compression ratios close to the optimal achieved by exhaustive search, but much more efficiently by leveraging learning instead of brute-force trial of every algorithm on every file.

CHAPTER 4
EXPERIMENTAL RESULTS

This chapter presents the results of our experiments, evaluating both the compression algorithms performance and the accuracy of the machine learning model. We analyze the data collected in the methodology and demonstrate the benefits of the proposed approach using charts and figures. Key evaluation aspects include the distribution of optimal algorithms, the predictive performance of the model, and comparisons to non-adaptive compression strategies.

4.1 Optimal Algorithm Distribution

First, we examine which algorithms were most often the optimal choice across the dataset. Figure 4.1 shows the frequency of each algorithm being the winning choice for files in the dataset. This is presented as a bar chart, where the x-axis lists the six compression algorithms (aggregating levels for simplicity), and the y-axis shows the number of files for which each algorithm produced the smallest compressed size under constraints. This chart reveals the overall winner distribution. We found that Zstd and LZMA dominate a large portion of files, especially large text-heavy files. Meanwhile, LZ4 and zlib are rarely the best in terms of compression ratio, because we favored speed over ratio in this setup.

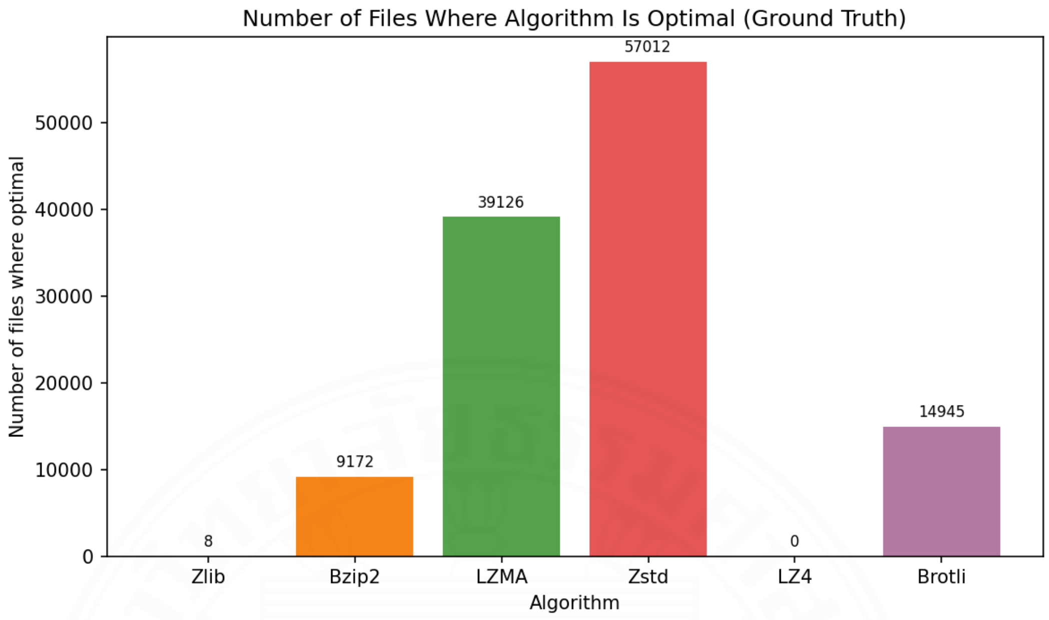

Figure 4.1 Optimal algorithm counts

In addition to overall counts, the detailed breakdown indicates which compression level for each algorithm was most often optimal. In our results, we observed that for algorithms like Zstd and Brotli, higher frequently provided the best compression within 30s for many files. LZMA at a mid-level (around preset 6) also appeared frequently, likely because it balances speed and ratio. On the other hand, algorithms known for speed (LZ4) never appear as the best ratio-wise. They are more likely to win in a time-constrained scenario not focused purely on size. This justifies the need for an intelligent selection: the optimal choice varies considerably from file to file.

4.2 Feature Importance and Selection Analysis

We do analyze how each feature contributes to the model. Results in Figure 4.2 show the incremental benefit of adding features during the feature selection process. In this figure we start with no feature and then add them one-by-one in order that maximize accuracy. The x-axis shows the number of features used, from 0 up to 14, and the y-axis is the classification accuracy we got. The curve in Figure 4.2 rises steeply at first, the first feature added gives a big jump in accuracy, meaning that feature alone carry significant predictive power. In our case the single most informative feature was

FileSize, it already let the model make a decent guess. When the second and third features are added, accuracy improves further. By around four features the gains basically plateau, confirming the top 4–5 features already capture most necessary information. Adding features beyond the fifth does not improve accuracy, the curve flattens in the figure and in some trials even cause small dips because of noise. This analysis validates our choice to focus on a small set of features

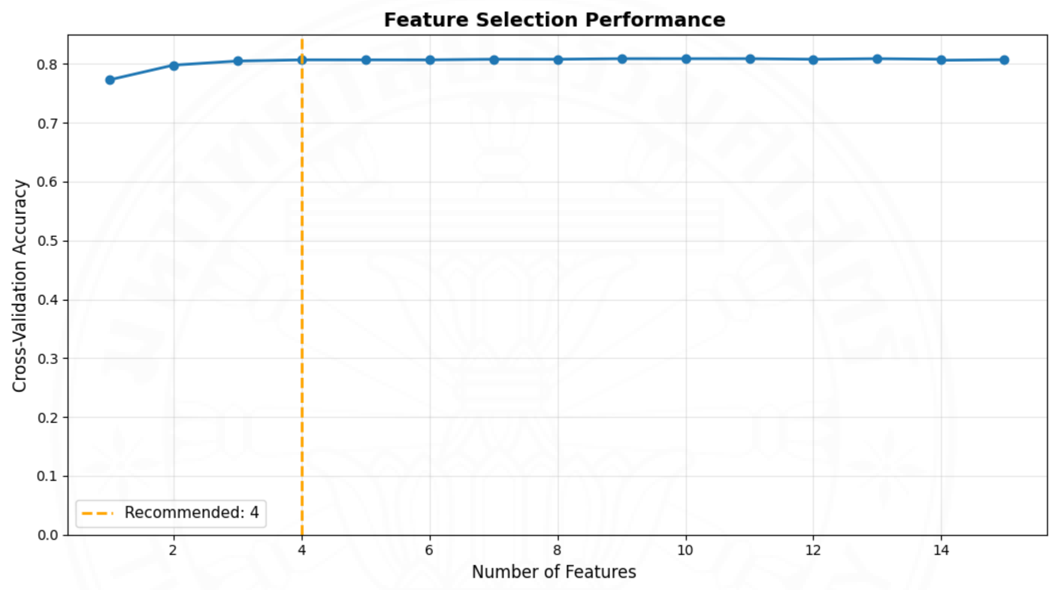

Figure 4.2 Feature Selection Performance

Moreover, the final trained XGBoost model feature importance is shown in Figure 4.3. This figure is a bar chart showing the relative importance of each selected feature as model assesses them. The features in final model are FileSize, AvgLineLength, AsciiRatio and ByteKurtosis. According to the plot, FileSize is most influenced feature, which is expected. File size has strong impact on which algorithm is best. The next most important feature is ByteKurtosis, it helps models to recognize files that have highly skewed byte distribution. AvgLineLength and AsciiRatio also have notable importance. AvgLineLength, even less dominant than FileSize or ByteKurtosis, still contributes by identifying files with many short lines compared to files with continuous stream data. The AsciiRatio helps to differentiate text-heavy file from binary file. The percentages in Figure 4.3 confirm that no single feature dominates the decision completely; the model really uses a combination. FileSize take around 30% importance, and the rest shared by ByteKurtosis, AvgLineLength and AsciiRatio. This balance reliance is good sign that the model considers multiple aspects of file structure, not just based everything on size or entropy. In summary, the feature importance reinforces our understanding: the model focuses on how large the file is and how text-like or repetitive the content is, to predict which compression method is best.

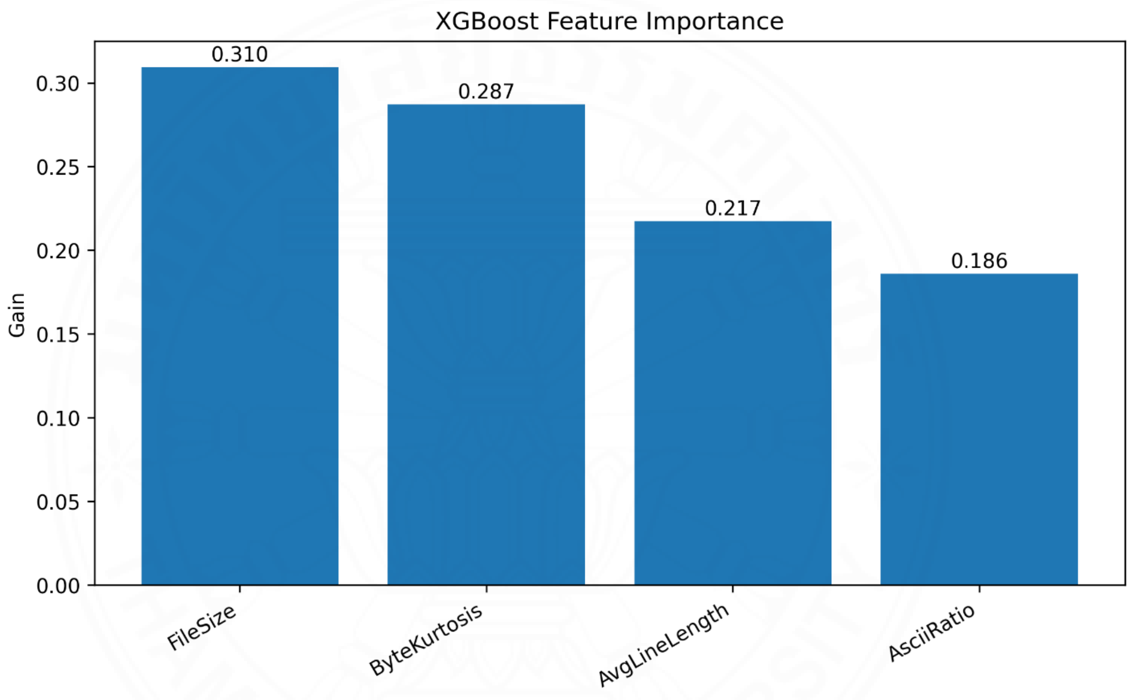

Figure 4.3 Feature Importance

4.3 Hyperparameter Optimization Results

This section reports the best validated performance obtained by each candidate classifier after tuning its hyperparameters. The summary bar chart in Figure 4.4 shows the best cross-validation accuracy achieved by each model:

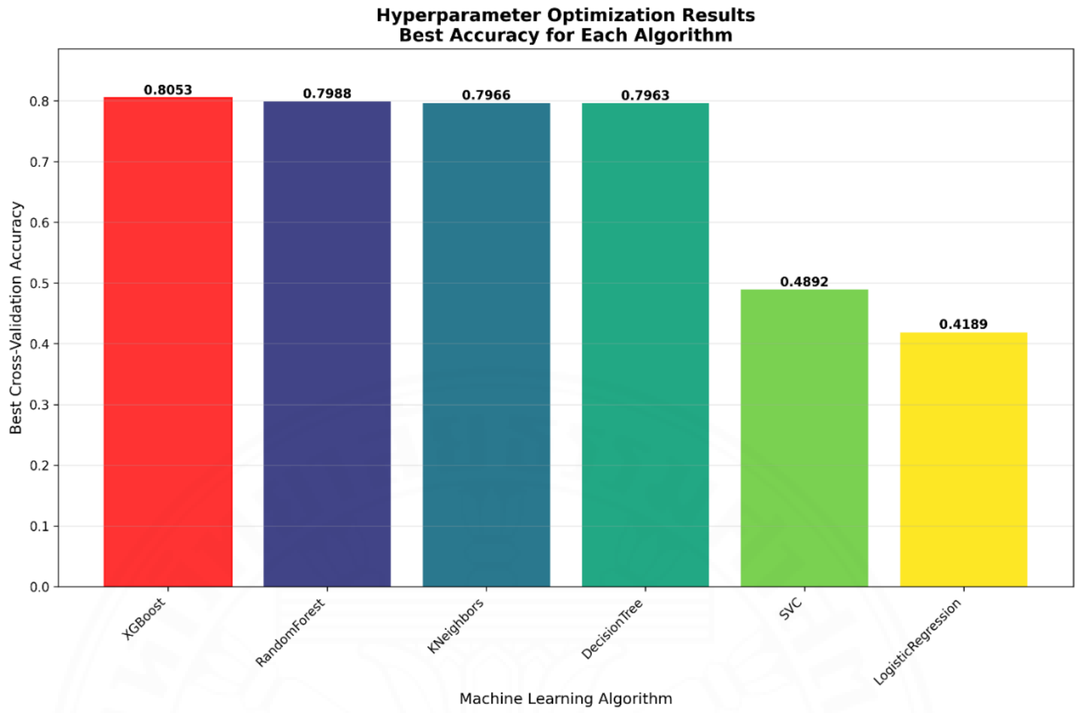

Figure 4.4 Hyperparameter Optimization Results

Figure 4.4 shows that Tree-based learners (XGBoost, Random Forest, Decision Tree) and the instance-based KNN substantially outperform the linear baseline (Logistic Regression) and the tested SVC configuration. The margin is small but steady in favor of XGBoost over the two ensemble/tree competitors and KNN, indicating boosted trees extract a bit more signal from interactions among features. Meanwhile, the poor accuracy of SVC and Logistic suggests the decision boundary in this task is highly non-linear and not well modeled by linear separators; they tend to underfit even when tuned.

Although Random Forest and KNN are close, XGBoost attains the highest validated score at 0.8053 and is selected as the final predictor. This choice also offers practical benefits: built-in feature importance for interpretability, good control of capacity via depth/regularization to avoid overfitting, and efficient inference time suitable for integration into the C# pipeline.

The optimization results demonstrate that boosted decision trees provide the best trade-off for this problem, high accuracy with manageable complexity. Linear and margin-based baselines do not capture the structure of the features well, while XGBoost reliably generalizes and will be used as the core predictive model in the subsequent evaluations.

4.4 Classification Performance (Confusion Matrix)

To evaluate how well the trained model performs in practice, we look at the confusion matrix of its predictions on a test set (or via cross-validation). The matrix is visualized in Figure 4.5, which shows predicted algorithm classes versus actual optimal algorithm classes for a set of files. Each row of the matrix corresponds to the true algorithm (ground truth a_i* for files), and each column corresponds to the algorithm predicted by the model. The diagonal entries (where prediction matches actual) represent correct predictions, while off-diagonals indicate mistakes, with the intensity or number in each cell showing how many files fall into that category.

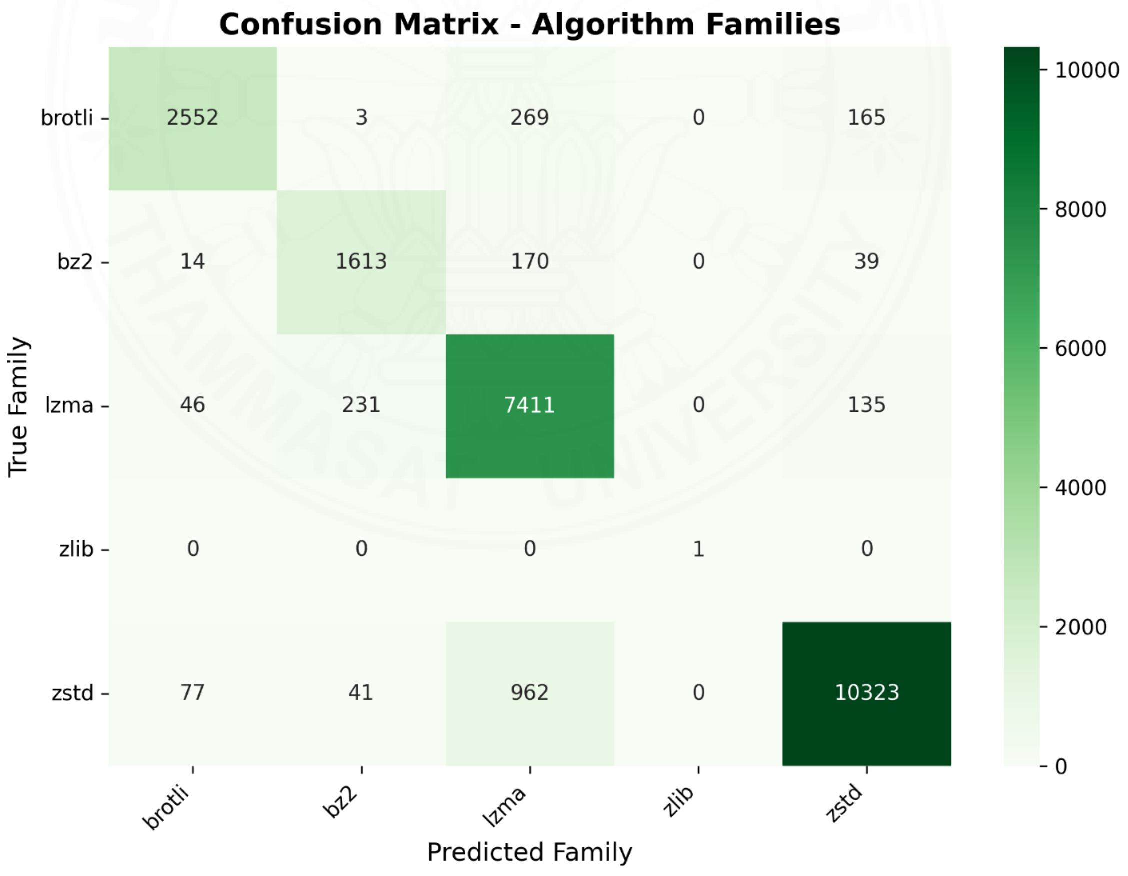

Figure 4.5 Confusion Matrix – Algorithm Families

The confusion matrix reveals that the model achieves a high overall accuracy – the majority of files lie on the diagonal, meaning the model correctly predicts their optimal compression algorithm. The few errors the model makes are mostly between algorithms that have somewhat similar performance profiles or occur in borderline cases. One noticeable pattern is a slight confusion between Brotli and Zstd for some files: these are both modern algorithms that compress well, and a few files that are actually best compressed by Brotli were predicted to use Zstd by the model or vice versa. Another area of confusion occurs between LZMA and Bzip2 on certain files. These two algorithms are somewhat similar in that they aim for high compression at the cost of speed, and for some text-heavy data the model might misidentify which of the two will edge out the other. These off-diagonal entries in the confusion matrix are relatively small compared to the correct predictions, indicating the error rate is low. Importantly, when the model predicts wrong, it usually picks an algorithm that is the second-best for that file rather than something completely unsuitable. Therefore, the impact on compression ratio is minor in those cases. The confusion matrix confirms the model predicts the optimal algorithm in most cases and only occasionally swaps between algorithms that are in the same category of performance for a file. Overall, Figure 4.5 demonstrates strong classification performance, with high true positive rates for each compression algorithm class and misclassification errors that are infrequent and mostly between similar algorithm choices.

4.5 Comparison with Baseline Compression

We evaluate the machine learning based compression strategy compared to a traditional baseline compression (standard ZIP) over random 1,200 files to see their performance. The comparison looks at compression effectiveness (compressed size) and compression speed, also if machine learning method can improve both at the same time. For fair analysis, we group test files by size: small (<10 MB), medium (10-300 MB), and large (>300 MB). This helps reveal trends that depend on input scale. The results are summarized in Table 4.1, showing if machine learning -based method is better, worse or similar to baseline in each case.

For small files (<10 MB), the machine learning selector is usually better than baseline in both. Most of the time it produces a bit smaller compressed size than ZIP.

More importantly, it compresses much faster for this small input. The baseline has more overhead on tiny file, machine learning approach often finishes quicker. In fact, for almost every small file test, the model method gives smaller output and also less time. This means on small data the machine learning approach gives consistent benefit in both efficiency and speed.

For medium files (10-300 MB), the result are mixed. The machine learning method still usually gets better compression ratio compared to baseline. But this comes with cost of speed. The model method was slower. In fact, in our test it slowed on all medium files, therefore the conventional compressor finishes faster. There is no case where machine learning approach was better in both size and speed together for medium. It usually gives gain in size reduction but sacrifices speed. For medium-sized input, machine learning methods offer small benefits in size reduction but cannot match baseline time.

For large files (>300 MB), the trade-off is clear. The machine learning method always makes smaller archive than baseline, that is often much smaller for big files, showing strength in effectiveness. But it compresses much slower on these large inputs. The baseline was far faster, while machine learning method took much longer to finish. Therefore, there was no case where machine learning improves metric for large files; it always trades huge increase in time for smaller sizes. This suggests that while the model reduces size substantially for large data, the time cost increases and may be impractical when speed is important.

Overall, the machine learning algorithm selection outperformed baseline in most cases when looking at individual metrics, especially for small files. It got smaller size in majority and also faster compression for most inputs. The baseline only clearly wins in speed for medium and large files, where model overhead is high. Importantly, across all file tests, machine learning approach led to much smaller total compressed data size while the total compression time is about same as baseline. In summary, the machine learning method gives big benefits in compression effectiveness and often improves speed, but the advantage depends on file size, and it may slow down on larger data.

Table 4.1 Performance Comparison of Machine Learning Algorithm Selector vs. Baseline ZIP Compression Over Random 1,200 Files

File Size Group	Compression Size (Output) vs Baseline	Compression Speed vs Baseline	Both Size & Speed Improved?
Small (<10 MB)	Better – output is commonly smaller than baseline Size Save: 17% (7064 KB)	Better – compresses faster in almost all cases Time Save: 95% (434 s)	Yes – both better tin speed and output size
Medium (10 – 300 MB)	Better – output slightly smaller in most cases Size Save: 13% (25 MB)	Worse – compresses slower for virtually all files Time Save: -1360% (-91 s)	No – only better in output size but slightly worse speed
Large (>300 MB)	Better – output significantly smaller for all tested files Size Save: 50% (158.66 MB)	Worse – compresses much slower on large files Time Save: -3173% (-236 s)	No – only better in output size but got worse speed
Total	Size Save: 36%	Time Save: 107 s	Yes – both better

In conclusion, the experiment comparison shows that machine learning base selection systems give strong advantage on compression effectiveness, especially for small input where both size and speed improve. For medium and large files, the model still gives better output size, but runtime becomes slower. However, this trade-off is not big problem for our target case. In real practice, most files submit to system are small, usually less than 10 MB. Since the model works best in this range by making compression faster and output smaller, it fits well with real operational needs. Even though the method loses speed when handling very large files, the design is still very suitable and effective for the actual data profile we face in application.

CHAPTER 5
CONCLUSION AND FUTURE DIRECTION

5.1 Conclusion

Many studies have been conducted to explore ways to improve data compression and reduce file transfer overhead. Traditional methods typically apply one compression algorithm uniformly to all files, which may not be optimal for every file type. Some approaches require manual user effort or complicated content analysis to choose an algorithm, making them impractical. This independent study presents an intelligent system that addresses these limitations by automatically selecting the optimal compression method for each file using machine learning.

This independent study aims to develop an adaptive compression framework that integrates machine learning with feature analysis of files. We assembled a large corpus of real-world files from the SEC database and other sources, covering a variety of file formats. For each file, various structural features (such as file size, entropy, and byte distribution statistics) were extracted, and the best compression algorithm was identified via exhaustive tests under practical time constraints. Then a predictive model was trained to learn the relationship between file features and the optimal algorithm.

The experimental results show that the proposed model can reliably predict the most effective compression algorithm for a given file. The model achieved around 80% accuracy in cross-validation, meaning it correctly selected the best algorithm for most files. As a result, the system often achieved compression ratios close to the theoretical optimum for each file without needing to try all algorithms. Our approach clearly outperforms the single-algorithm baseline: it yields smaller compressed files on average, and for many cases (especially with smaller files) it also compresses faster than using a fixed method. This confirms that an adaptive, file-specific compression strategy can significantly improve efficiency of storage and transmission while keeping processing time within acceptable limits.

However, the benefits vary depending on file characteristics. The results also reveal that for extremely large files, the selected algorithm (often a slower but high-compression method) sometimes leads to longer processing time compared to a fast baseline like standard Zip. This trade-off indicates that while our method maximizes compression, it may sacrifice speed on very large data. Fortunately, even in such cases, the chosen algorithm is usually the second-best alternative and the impact on final size remains beneficial. The study confirms that no single compression algorithm is universally optimal – the best choice differs from file to file, validating the core premise of this research.

5.2 Limitations

Despite the model's successful performance, this study has several limitations. First, the training dataset was composed mostly of financial data, which creates a potential domain bias; the model may not perform as well on files from vastly different fields, such as genomics or multimedia. Second, a clear trade-off was observed for large files, where the model's optimal choice "compresses much slower on these large inputs" to achieve a smaller size, a trade-off that may not be acceptable in all time-sensitive applications. Finally, the scope was limited to six specific lossless compression algorithms; other modern or specialized algorithms were not included in the analysis.

5.3 Future Direction

The Future work will focus on techniques to further improve the system, such as integrating compression speed prediction into the decision-making, which will help optimize both compression ratio and time performance especially for large files. Another enhancement is to expand the feature set or use deeper learning techniques, which might increase prediction accuracy beyond the current 80%. For instance, including content-specific features or training specialized models for certain file categories might help the model differentiate algorithm choices even better. Additionally, evaluating the approach on broader types of data or under different operational constraints (such as streaming data or more stringent time limits) will be valuable. These improvements will help increase the practicality and robustness of the system. Ultimately, the framework introduced by this independent study provides a foundation for intelligent compression in data pipelines, and future developments will aim to make it even more accurate, faster and widely applicable.

REFERENCES

Alakuijala, J., & Szabadka, Z. (2016). Brotli compressed data format (RFC 7932). Internet Engineering Task Force. doi: 10.17487/RFC7932

Bartik, M., Ubik, S., & Kubalík, P. (2015). LZ4 compression algorithm on FPGA. *Proceedings of the 2015 IEEE International Conference on Electronics, Circuits, and Systems (ICECS)* (pp. 179-182). Cairo, Egypt: IEEE. doi: 10.1109/ICECS.2015.7440278

Biau, G., & Scornet, E. (2016). A random forest guided tour. *TEST*, 25(2), 197–227. doi: 10.1007/s11749-016-0481-7

Burtchell, B. A., & Burtscher, M. (2024). Using machine learning to predict effective compression algorithms for heterogeneous datasets. *Proceedings of the 2024 Data Compression Conference (DCC)* (pp. 183-192). Snowbird, UT, USA: IEEE. doi: 10.1109/DCC58796.2024.00026

Chen, T., & Guestrin, C. (2016). XGBoost: A scalable tree boosting system. *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD 2016)* (pp. 785-794). San Francisco, CA, USA: ACM. doi: 10.1145/2939672.2939785

Collet, Y., & Kucherawy, M. (2018). Zstandard compression and the application/zstd media type (RFC 8478). Internet Engineering Task Force. doi: 10.17487/RFC8478

Cover, T. M., & Hart, P. E. (1967). Nearest neighbor pattern classification. *IEEE Transactions on Information Theory, 13*(1), 21-27. doi: 10.1109/TIT.1967.1053964

Deutsch, P. (1996). DEFLATE compressed data format specification version 1.3 (RFC 1951). Internet Engineering Task Force. doi: 10.17487/RFC1951

Fitriya, L. A., Purboyo, T. W., & Prasasti, A. L. (2017). A review of data compression techniques. *International Journal of Applied Engineering Research, 12*(19), 8956-8963.

Guido, R., Ferrisi, S., Lofaro, D., & Conforti, D. (2024). An overview on the advancements of support vector machine models in healthcare applications: A review. *Information, 15*(4), 235. doi: 10.3390/info15040235

James, G., Witten, D., Hastie, T., & Tibshirani, R. (2021). *An introduction to statistical learning: With applications in R* (2nd ed.). New York, NY: Springer.

Jin, Y., Fu, Y., Liu, T., & Dong, L. (2019). Adaptive compression algorithm selection using LSTM network in column-oriented database. *Proceedings of the 2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference (ITNEC)* (pp. 652-656). Chengdu, China: IEEE. doi: 10.1109/ITNEC.2019.8729341

Juelsson Larsen, L., & Persson, D. (2023). *Compression selection for columnar data using machine-learning and feature engineering* (Master's thesis). Malmö, Sweden: Malmö University. Retrieved from https://urn.kb.se/resolve?urn=urn:nbn:se:mau:diva-61266

Mahoney, M. (2012). *Data compression explained*. Round Rock, TX: Dell Inc. Retrieved from https://mattmahoney.net/dc/dce.html

Yang, H., Qin, G., & Hu, Y. (2023). Compression performance analysis of different file formats. *arXiv preprint arXiv:2308.12275*. Retrieved from https://arxiv.org/abs/2308.12275