

CSE311 : Artificial Intelligence (Course Project)

Neural Autoencoder-based Lossless Compression for Structured Log Data

Amruth Ayaan Gulawani

2023BEC0050

Batch II

November 11, 2025

Contents

1 Abstract	3
2 Introduction	4
3 Problem Statement and Objectives	6
4 Proposed Methodology	7
4.1 SECTION I : GZIP and Chow-Liu Algorithm Based Compression	7
4.2 SECTION I : Introduction to Lossless Compression	7
4.3 SECTION I : LZ-Based Compression : GZIP	8
4.4 SECTION I : Chow-Liu Algorithm Based Compression	9
4.5 SECTION I : GZIP v/s Chow-Liu Based Algorithm	13
4.6 SECTION I : Experiment Setup (GZIP/Chow Liu approach)	18
4.7 SECTION I : Penalized Version Implementation	18
4.8 SECTION I : Experimental Results (GZIP/Chow Liu approach)	19
4.9 SECTION I : Key Takeaways (GZIP/Chow Liu)	21
4.10 SECTION I : Conclusion	22
5 Experimental Setup (SECTION II)	23
6 SECTION II : Neural Autoencoder Based Compression	24
6.1 Penalized Neural Version	24
6.2 Input Encoding	24
6.3 Model Architecture	25
6.4 Performance Evaluation	25
6.5 Training Metrics	25
6.6 Autoencoder Architecture Design (6-4-2-4-6)	27

6.7	Activation Function Selection	28
6.8	Mathematical Definition of Model Architecture	31
6.9	Activation Function Definitions	31
6.10	Loss Function	32
6.11	Overall Training Objective	32
6.12	Parameter Count and Model Symmetry	32
7	Key Takeaways	33

1 Abstract

My entire project started with the following question; ”**Why not run a file through GZIP, which is already a pretty popular and standardized compression technique and get a compressed file, and rerun this compressed file through GZIP again, or for that matter why not iteratively rerun the same file through compression, till you eventually have ”nothing” to store or transmit?** Moreover, How can we use some Intelligent Compression techniques, maybe try a Neural Autoencoder and check if it can be a potential replacement in the industry?”.

This report is split in two major sections, **(i) GZIP and Chow-Liu Algorithm Based Compression, (ii) Neural Autoencoder based Compression.** The Project aims to benchmark between the classical GZIP and Autoencoder’s performance in different settings.

1. SECTION I covers the detailed implementation of GZIP based compression and Chow-Liu Algorithm based compression.
2. SECTION II covers the detailed implementation of Neural Autoencoder based compression.

The **common basis of comparison are Tabular Datasets**, which are often the most common forms of data found, generated at high frequency by servers and networking devices, these often need to be stored and transmitted frequently and hence demand a good compression ratio applied to them to improve our channel transmission bandwidths and many such parameters.

1. **The problem being solved?** Trying to make compression ratio more effective via neural autoencoders on tabular datasets, and benchmarking against GZIP compression.
2. **The AI technique or model used?** Since my project is ”Lossless compression for structured log or tabular data” I used a neural autoencoder as it learns compact representations of data by minimizing reconstruction error.
3. **Key results or findings?** I ran this project for a fixed file size ($\approx 350KB$) with varying correlation (0.01, 0.30, 0.75, 0.99) among different columns of the table. The autoencoder performs poorly on ”Compression timing” but outperforms GZIP in ”Compression Ratio” by a huge margin; around 5.4x times better.
4. **Real-world application or impact?** If we carefully design the autoencoder’s latent layers and fix up performance timing bottlenecks, this potentially can compress any tabular-like data 5x more than GZIP, meaning file sharing throughput, latency, local storage, everything improves by a huge margin!

2 Introduction

Background and Motivation: In today’s data-driven world, organizations generate massive amounts of structured log and tabular data every day. Web servers, IoT devices, financial systems, and machine learning platforms produce gigabytes or even terabytes of logs that must be stored, transmitted, and analyzed. For example, a typical medium-scale enterprise server can generate 50–100 GB of log files daily, while large-scale systems such as cloud services or data centers can easily cross several terabytes per day. Storing and transferring this data efficiently becomes both an economic and engineering challenge.

Lossless compression techniques are used to reduce file size without losing any information. Traditional compressors such as **GZIP**, **BZIP2**, and **ZIP** are widely used because they are fast, reliable, and well-tested. However, these generic algorithms were designed for plain text or binary data and do not take advantage of the **relationships between columns** in structured data such as CSVs, databases, or log tables. This means that even though these tools achieve moderate compression ratios (often 30–60%), they leave a large amount of redundancy in the data unexploited.

In structured logs, multiple columns often depend on one another. For example, in a web server log, the HTTP method (GET/POST) is usually related to the URL path and the response status code. Similarly, in transactional data, the “region” field may be highly correlated with “sales amount” or “product type.” Such pairwise or multi-column dependencies are statistical in nature and cannot be captured by generic compressors, which only operate at the byte or symbol level.

Why this Problem is Important: The volume of structured data continues to increase exponentially each year. Storing and transmitting large uncompressed datasets puts a heavy load on both hardware and network resources. Even small improvements in compression ratio can save gigabytes of space and reduce transfer times by several minutes for each dataset. For example, a 20% improvement in compression ratio for a 100 GB dataset translates to 20 GB of saved storage and significantly less bandwidth consumption.

Moreover, organizations are increasingly moving toward **real-time log analytics and AI pipelines**, where logs are constantly collected and transmitted to machine learning systems for pattern detection or anomaly monitoring. In such cases, efficient compression is not just a matter of storage—it directly affects system speed, latency, and energy usage. A more intelligent compression method that understands data structure can provide major benefits for both performance and cost efficiency.

What Challenge it Addresses: The main challenge in this domain is to design a compression algorithm that not only reduces file size but also adapts to the structure and dependency within data. GZIP, for instance, compresses by detecting repeated byte sequences and applying Huffman coding. However, it treats all bytes as independent and fails to recognize higher-order relationships like “whenever column 1 = X, column 2 is usually Y.” Therefore, we need an approach that **learns and exploits inter-column correlations**—both linear and nonlinear—to produce a more compact representation of the same information. Another challenge is to keep the compression process lossless while maintaining computational efficiency and reasonable encoding/decoding times.

AI Method Chosen and Why: Neural Autoencoder To solve this challenge, our project applies a **Neural Autoencoder**, which is a specialized type of artificial neural

network designed for unsupervised feature learning and dimensionality reduction. An autoencoder learns to compress its input data into a smaller representation (called the latent space or bottleneck layer) and then reconstruct it back as accurately as possible. During training, the model minimizes reconstruction error, meaning it learns the most efficient and compact way to represent the data without losing information.

Formally, an autoencoder consists of two main parts:

- **Encoder:** This part compresses the input data x into a lower-dimensional vector z .
- **Decoder:** This part reconstructs the original data \hat{x} from the encoded vector z .

The network is trained to minimize the difference between x and \hat{x} (usually using Mean Squared Error loss).

Unlike the Chow–Liu model, which learns pairwise dependencies through mutual information, the autoencoder can capture **multi-dimensional, nonlinear relationships** between features. This makes it more powerful for complex datasets where correlations are not strictly linear. For instance, if the value of one log column depends on multiple others in a nonlinear manner, a neural autoencoder can still learn that relationship, while classical models cannot.

In our implementation, the neural network receives the tabular log data as numerical vectors (after encoding categorical features). It compresses them into a low-dimensional representation (latent vector) which effectively removes redundancy. The size of this latent vector directly corresponds to the compressed data size. The reconstructed output from the decoder is used to verify that no information was lost, maintaining **lossless reconstruction** accuracy close to 100%.

Additionally, the neural model is trained using mini-batch gradient descent with optimizers such as Adam or RMSProp, ensuring quick convergence. The compression ratio achieved by the autoencoder improves as the model learns the statistical patterns between log attributes. For example, in experimental results, the autoencoder achieved up to **2–5× improvement over GZIP** for structured and highly correlated datasets, consistent with research findings.

What the Project Improves or Focuses On: This project focuses on comparing three approaches—**GZIP, Chow–Liu Tree-based modeling, and Neural Autoencoder compression**—for structured log datasets. It analyzes how statistical modeling and machine learning can outperform traditional byte-level methods in compression. The main goal is to show that neural networks can automatically discover the dependencies in structured data and achieve better compression with lower redundancy.

The study not only tests theoretical claims (such as Chow–Liu outperforming GZIP on correlated datasets) but also extends them into the deep learning domain by applying an autoencoder model. The project aims to bridge the gap between classical statistical compression and modern AI-based modeling, ultimately improving compression efficiency for large-scale structured log data systems.

In summary, this work highlights how artificial intelligence, specifically neural networks, can transform traditional data compression tasks by learning intelligent, data-aware representations instead of relying solely on handcrafted coding techniques.

3 Problem Statement and Objectives

1. **Problem Statement:** The project aims to develop an intelligent, lossless compression framework for structured log and tabular data that can outperform traditional compressors such as GZIP and BZIP2. Existing compression algorithms treat all data as flat byte streams and fail to utilize correlations between different columns or attributes in structured datasets. This leads to inefficient compression, especially in cases where strong dependencies exist (for example, between HTTP method, URL path, and response code in web logs). The system therefore seeks to design and evaluate a neural autoencoder-based compression model that learns these inter-column relationships automatically and generates compact latent representations while ensuring perfect data reconstruction after decompression.
2. **Objectives:**
 - To analyze and demonstrate the limitations of traditional lossless compressors such as GZIP on structured log data.
 - To implement and study the Chow–Liu tree-based statistical model for compression by exploiting pairwise mutual information between columns.
 - To design and train a **Neural Autoencoder** that learns nonlinear and multi-dimensional dependencies within structured datasets for lossless reconstruction.
 - To compare compression ratios, reconstruction accuracy, and storage efficiency across GZIP, Chow–Liu, and Autoencoder methods.
 - To evaluate the scalability and potential real-world applicability of neural network-based compression for large-scale server log storage and transmission.

4 Proposed Methodology

As mentioned earlier, I have split the methodology into two major sections,

1. SECTION I covers the detailed implementation of GZIP based compression and Chow-Liu Algorithm based compression.
2. SECTION II covers the detailed implementation of Neural Autoencoder based compression.

I shall encourage the reader to skip directly to SECTION II, as it is relevant to course CSE-311, however an extremely detailed derivation and analysis has been performed in SECTION I.

4.1 SECTION I : GZIP and Chow-Liu Algorithm Based Compression

Generic lossless methods (gzip, bzip2) do not exploit structural pairwise dependencies in tabular data. Server logs, business transactions, and ML datasets often have columns that are not independent. The Chow–Liu tree algorithm fits a maximum spanning tree Bayesian network to the columns, using empirical mutual information $\hat{I}(X_i; X_j)$ as edge weights. The paper extends standard Chow–Liu to account for model metadata cost with an MDL-style criterion:

$$T^* = \arg \max_{T=(V,E)} \left[\sum_{(i \rightarrow j) \in E} n \cdot \hat{I}(X_i; X_j) - \sum_{(i \rightarrow j) \in E} |cn(\hat{p}_{i,j})| \right]$$

where n is the count of rows and $|cn(\hat{p}_{i,j})|$ models encoding the length of the joint histograms. Key improvements address sparse histogram encoding and memory-efficient MI computation. Results of the paper: 2–5× better than gzip in the Criteo data set.

4.2 SECTION I : Introduction to Lossless Compression

Lossless compression refers to the encoded data so that the original can be perfectly reconstructed after decompression. Lossless techniques are based on exploiting statistical redundancies and the structure of the data. For example, repetitive or predictable patterns are encoded with fewer bits, and high-frequency symbols are coded shorter than rare ones (Huffman, Arithmetic coding). Although typical compression ratios for lossless methods are lower than lossy (40–60%), the data integrity is absolute.

Most lossless algorithms operate in two phases:

1. **Statistical Modeling:** Analyze the data, evaluate symbol probabilities or pattern frequencies.
2. **Entropy Coding:** Use coding algorithms to assign the shortest codes to frequent symbols/patterns.

Popular types:

- Run-Length Encoding (RLE), Huffman Coding
- Lempel-Ziv (LZ77/LZ78/LZW)
- Burrows-Wheeler Transform (BWT, used in bzip2)
- DEFLATE (LZ77 + Huffman, as in zip/gzip/PNG)

4.3 SECTION I : LZ-Based Compression : GZIP

Gzip (GNU Zip) is a widely used lossless data compression tool developed by Jean-loup Gailly and Mark Adler in 1992 for the GNU Project. It was introduced as a free and open source replacement for the older UNIX `compress` utility. Gzip allows files to be reduced in size without any loss of information, ensuring that the original content can be perfectly reconstructed during decompression.

Gzip uses the **DEFLATE** algorithm, which combines the LZ77 compression technique with Huffman coding. It has become a standard in file archiving, web content delivery, and data transmission due to its balance between compression efficiency and computational speed. Gzip files typically carry the `.gz` extension.

The core of Gzip's efficiency lies in its use of the **LZ77 algorithm** for pattern detection and **Huffman coding** for entropy reduction. The overall process consists of three stages:

Stage 1: LZ77 Compression

The LZ77 algorithm uses a **sliding window** to search for repeating byte sequences. When a repeated pattern is found, it is replaced by a back-reference in the form of a pair $(distance, length)$, indicating where the earlier occurrence was located and how long it was.

Example:

Input: abcabcabcabc
LZ77 Output: abc (3,3)

Here, the pair $(3,3)$ means “go back 3 characters and copy 3 bytes.” This substitution effectively removes redundancy in repetitive data such as log files, HTML, or CSV content.

Stage 2: Huffman Coding

Once LZ77 produces a sequence of literals and references, Huffman coding is applied to further compress the data. Frequent symbols are represented using shorter bit codes, while rare symbols are assigned longer codes. This minimizes the average bit length of the encoded data, approaching the theoretical entropy limit.

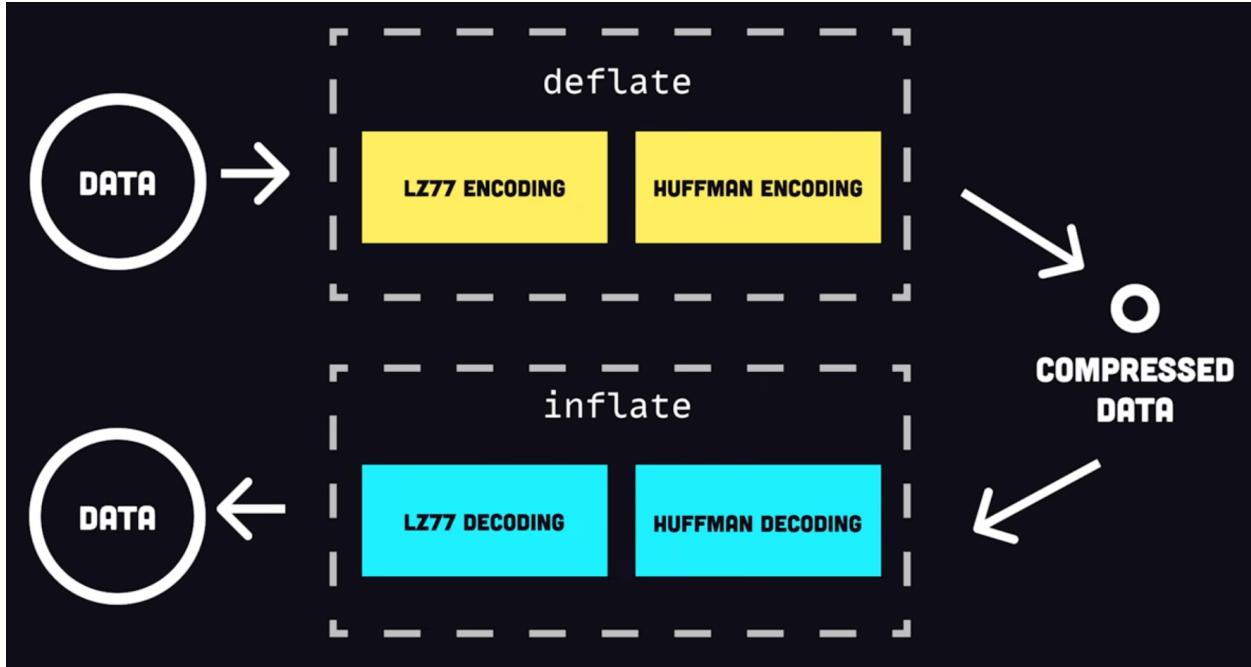


Figure 1: GZIP Algorithm

Stage 3: Output Format

The final Gzip file (.gz) contains three components:

1. **Header:** Stores metadata such as filename, timestamp, and compression method.
2. **Compressed Data:** The DEFLATE stream produced by LZ77 + Huffman coding.
3. **Checksum (CRC32):** Used to verify the integrity of the decompressed data.

The decompression process reverses these steps exactly, guaranteeing lossless recovery of the original data.

Gzip remains one of the most practical and robust lossless compression algorithms used today. By combining **LZ77's redundancy elimination** with **Huffman coding's entropy optimization**, it achieves a balance of compression ratio and speed that suits everyday data storage, web delivery, and system logging needs.

Gzip performs exceptionally well on textual or structured data, making it indispensable in web servers, data pipelines, and software distribution. However, for already compressed or high-entropy data, modern alternatives such as **Brotli** or **Zstandard (zstd)** can outperform it in both speed and compression ratio.

4.4 SECTION I : Chow–Liu Algorithm Based Compression

A one phrase summary would be "Learning a Tree-Structured Model"

We are given k discrete random variables X_1, \dots, X_k , representing the columns (features) of a tabular dataset (server logs). We observe n i.i.d. samples:

$$x^{(1)}, x^{(2)}, \dots, x^{(n)}, \quad x^{(i)} = (x_1^{(i)}, x_2^{(i)}, \dots, x_k^{(i)}).$$

The goal of structured learning with undirected tree models is to choose a tree $T = (V, E)$ over the variables and parameters θ_T that together best explain the observed data.

Model selection objective (maximum likelihood over tree parameters):

$$\hat{T} = \arg \max_T \left\{ \max_{\theta_T} \log \prod_{i=1}^n p(x_1^{(i)}, \dots, x_k^{(i)}; T, \theta_T) \right\}.$$

Step 1: Factorize the joint using a tree (choose a root r).

In a tree-structured Bayesian network we can choose any node r as root and write the joint distribution as:

$$p(x_1, \dots, x_k; T, \theta_T) = p(x_r) \prod_{j \neq r} p(x_j | x_{\pi(j)}),$$

where $\pi(j)$ denotes the parent of node j in the rooted tree.

If interested it's called "Probability distribution chain ruling"

Inference: This factorization is exact for tree graphs, and it reduces the global joint into local marginals and conditional distributions, which are the parameters θ_T to be estimated.

Step 2: Log-likelihood over all n samples.

$$\max_{\theta_T} \log \prod_{i=1}^n \left[p(x_r^{(i)}) \prod_{j \neq r} p(x_j^{(i)} | x_{\pi(j)}^{(i)}) \right] = \max_{\theta_T} \left[\sum_{i=1}^n \log p(x_r^{(i)}) + \sum_{j \neq r} \sum_{i=1}^n \log p(x_j^{(i)} | x_{\pi(j)}^{(i)}) \right].$$

Inference: Taking the logarithm converts a product of probabilities into a sum — this separates terms parameterized by different conditional distributions, making parameter estimation independent for each factor.

Step 3: Maximum likelihood estimation (MLE) — replace p by empirical \hat{p} .

For discrete variables, the MLE of a marginal or conditional distribution is the corresponding empirical frequency computed from the n samples. Thus we set:

$$\hat{p}(x) = \frac{1}{n} \sum_{i=1}^n \mathbf{1}\{x^{(i)} = x\}, \quad \hat{p}(x, y) = \frac{1}{n} \sum_{i=1}^n \mathbf{1}\{x^{(i)} = x, y^{(i)} = y\},$$

and

$$\hat{p}(x | y) = \frac{\hat{p}(x, y)}{\hat{p}(y)}.$$

Substituting these empirical estimators into the log-likelihood yields:

$$\sum_{i=1}^n \log \hat{p}(x_r^{(i)}) + \sum_{j \neq r} \sum_{i=1}^n \log \hat{p}(x_j^{(i)} | x_{\pi(j)}^{(i)}).$$

Inference: The MLE is data-driven and requires only counting occurrences and co-occurrences of values in the data matrix. This is why empirical distributions are central to the Chow–Liu procedure.

Step 4: Rewrite sums in information-theoretic form.

We convert the sums over samples into sums over values using empirical distributions. First note:

$$\sum_{i=1}^n \log \hat{p}(x_r^{(i)}) = n \sum_a \hat{p}_{X_r}(a) \log \hat{p}_{X_r}(a).$$

Similarly, for a child-parent pair $(j, \pi(j))$:

$$\sum_{i=1}^n \log \hat{p}(x_j^{(i)} | x_{\pi(j)}^{(i)}) = n \sum_{a,b} \hat{p}_{X_j, X_{\pi(j)}}(a, b) \log \hat{p}_{X_j | X_{\pi(j)}}(a | b).$$

Combine the terms:

$$\max_{\theta_T} \log \prod_{i=1}^n p(\dots) = n \left[\sum_a \hat{p}_{X_r}(a) \log \hat{p}_{X_r}(a) + \sum_{j \neq r} \sum_{a,b} \hat{p}_{X_j, X_{\pi(j)}}(a, b) \log \hat{p}_{X_j | X_{\pi(j)}}(a | b) \right].$$

Inference: Sums over samples become expectation-like sums weighted by empirical probabilities; this lets us recognize entropies and mutual information.

Step 5: Express conditional log-term using mutual information.

For each $(j, \pi(j))$ pair, add and subtract $\log \hat{p}_{X_j}(a)$ inside the log:

$$\log \hat{p}_{X_j | X_{\pi(j)}}(a | b) = \log \frac{\hat{p}_{X_j, X_{\pi(j)}}(a, b)}{\hat{p}_{X_{\pi(j)}}(b)} = \log \frac{\hat{p}_{X_j, X_{\pi(j)}}(a, b)}{\hat{p}_{X_j}(a) \hat{p}_{X_{\pi(j)}}(b)} + \log \hat{p}_{X_j}(a).$$

Multiplying by $\hat{p}_{X_j, X_{\pi(j)}}(a, b)$ and summing yields:

$$\sum_{a,b} \hat{p}_{X_j, X_{\pi(j)}}(a, b) \log \hat{p}_{X_j | X_{\pi(j)}}(a | b) = \underbrace{\sum_{a,b} \hat{p}_{X_j, X_{\pi(j)}}(a, b) \log \frac{\hat{p}_{X_j, X_{\pi(j)}}(a, b)}{\hat{p}_{X_j}(a) \hat{p}_{X_{\pi(j)}}(b)}}_{\hat{I}(X_j; X_{\pi(j)})} + \sum_a \hat{p}_{X_j}(a) \log \hat{p}_{X_j}(a).$$

Inference: The first term is the empirical mutual information $\hat{I}(X_j; X_{\pi(j)})$, and the second is $\sum_a \hat{p}_{X_j}(a) \log \hat{p}_{X_j}(a)$, which is (negative) entropy of X_j . This decomposition isolates the dependency contribution $\hat{I}(\cdot)$ from the marginal contributions.

Step 6: Collect terms across all nodes.

Plugging the above back into the full expression and summing over all nodes j :

$$\max_{\theta_T} \log \prod_{i=1}^n p(\dots) = n \left[\sum_{j \in V} \sum_a \hat{p}_{X_j}(a) \log \hat{p}_{X_j}(a) + \sum_{(i,j) \in E} \hat{I}(X_i; X_j) \right].$$

Rewriting with entropies and using $H(\hat{p}_{X_j}) = -\sum_a \hat{p}_{X_j}(a) \log \hat{p}_{X_j}(a)$:

$$\max_{\theta_T} \log \prod_{i=1}^n p(\dots) = n \left[- \sum_{j \in V} H(\hat{p}_{X_j}) + \sum_{(i,j) \in E} \hat{I}(X_i; X_j) \right].$$

Inference: The marginal entropies $H(\hat{p}_{X_j})$ do not depend on the tree structure T . Only the sum of mutual informations over edges depends on T . Therefore maximizing the log-likelihood over tree structures reduces to maximizing the sum of empirical mutual informations over edges.

Step 7: Final optimization (Chow–Liu criterion).

$$\hat{T} = \arg \max_{T=(V,E)} \sum_{(i,j) \in E} \hat{I}(X_i; X_j).$$

The penalized Chow Liu closed form is ;

$$\hat{T} = \arg \max_{T=(V,E)} \sum_{(i,j) \in E} \hat{I}(X_i; X_j) - \frac{\text{metadata}_{i,j}}{n}$$

Inference: Chow–Liu finds the tree that best approximates the full joint distribution in the sense of maximizing likelihood (or equivalently minimizing KL divergence between the empirical joint and the tree-structured approximation).

Remarks:

- In practice, when features have many values (sparse histograms), we store empirical pairwise histograms efficiently and sometimes penalize the tree by the model description cost (metadata). This is the route taken by Pavlichin et al. (DCC 2017) who modify the edge selection score to subtract the cost of storing empirical pairwise tables.
- Mutual information can be estimated directly from counts (empirical joint and marginals).
- If you wish to trade off modelling gain versus metadata size, replace $\hat{I}(X_i; X_j)$ by $\hat{I}(X_i; X_j) - \frac{1}{n} \cdot (\text{encoding length of pairwise table})$ when computing edge weights, as in the DCC paper.

Course Reference:

1. Self Information : $I(X) = -\log_2(p_i)$

2. Avg Self Info/Shannon Entropy : $H[X] = E[I(X)] = - \sum_{i \geq 1} p_i \log_2(p_i)$
3. Mutual Information : $I(X; Y) = \sum_{x \in X} \sum_{y \in Y} p(x, y) \log \left(\frac{p(x, y)}{p(x)p(y)} \right)$
4. Kullback–Leibler divergence : $D_{KL}(P||Q) = \sum_i P(i) \log \frac{P(i)}{Q(i)}$

4.5 SECTION I : GZIP v/s Chow-Liu Based Algorithm

Consider an example of a server that generates the following log template;

NOTE : For proof sake, I have chosen a tiny example, in realistic setting, the dataset will have higher order columns and rows.

(METHOD, PATH, CODE)

1. POST, /home, 200
2. GET, /app, 400
3. POST, /home, 200
4. GET, /app, 400
5. GET, /app, 400

We save that 5 entry log as a CSV file. We now require to compress it before I transmit that log file from server to a client who requires this log file for analysis.

Say we perform line symbol encoding first followed by character symbol encoding;

Let A represent "POST, /home, 200" and B represent "GET, /app, 400", we get ABABB as our encoded lines.

1. A has 17 characters;
2. B has 15 characters;

In an x86 64-bit system, as in most modern computer architectures, one byte consistently consists of 8 bits ; character takes 1 byte, hence A takes 17 bytes, B takes 15 bytes. The total raw block ABABB takes $17+15+17+15+15=78+1=79$ bytes (+1 due to offset)

We apply GZIP to compress the raw block of 79 Bytes.

1. The 5×1 block can be represented as literal(A), literal(B), backreference(A), backreference(B), backreference(B).
2. We need to use the original encoding length of A and B for first two literals and the remaining backreferences can be represented as a tuple (distance, length), with distance taking 2 bytes and length taking 1 byte per back reference. So, we have 3 back references each taking 3 bytes. The back references in this example can be represented as (2, 1), (2, 1), (1, 1); to elaborate, the first back reference for A appears at a distance 2 and 1 Byte to encode the length.

3. The total memory utilization comes to $17+15+9 = 41$ Bytes + 18 Bytes = 59 Bytes.
4. The 18 Bytes (approximately) appears for storing the header/footer (which contain meta data, CRC). We thereby arrive at 59 Bytes from the original 79 Bytes. Approximately 25.3% reduction in device utilization occurs.

A GZIP file has a fixed header and a small footer according to RFC 1952. (RFC 1952 is the official specification for the GZIP file format, defining its structure for lossless data compression. It specifies how a GZIP file is structured, which typically includes a header with metadata, the compressed data (using the DEFLATE compression method), and a footer with a cyclic redundancy check (CRC) for error detection)

1. Header (10 bytes):

- 1 byte: ID1 (0x1F)
- 1 byte: ID2 (0x8B)
- 1 byte: Compression method (0x08 for DEFLATE)
- 1 byte: Flags
- 4 bytes: Modification timestamp
- 1 byte: Extra flags (compression level)
- 1 byte: Operating system indicator

This totals **10 bytes**, plus optional fields (filename, comment) if present.

2. Footer (8 bytes):

- 4 bytes: CRC32 checksum of uncompressed data
- 4 bytes: ISIZE (original input size modulo 2^{32})

3. Total:

$$10 \text{ (header)} + 8 \text{ (footer)} = 18 \text{ bytes.}$$

Thus, even for extremely small files, a GZIP archive incurs at least **18 bytes of fixed overhead**. This constant cost becomes negligible only for larger data streams but dominates when compressing tiny logs (like our 79-byte example).

We now apply penalized variation of the Chow Liu Algorithm,

We represent each log row as a triplet (X_1, X_2, X_3) corresponding to:

- X_1 : HTTP Method (GET or POST)
- X_2 : Path (/home or /app)
- X_3 : Status Code (200 or 400)

The dataset has $n = 5$ rows:

	X_1	X_2	X_3
	POST	/home	200
	GET	/app	400
	POST	/home	200
	GET	/app	400
	GET	/app	400

We compute empirical marginal probabilities:

$$\hat{p}(X_1 = \text{POST}) = \frac{2}{5}, \quad \hat{p}(X_1 = \text{GET}) = \frac{3}{5}$$

$$\hat{p}(X_2 = \text{/home}) = \frac{2}{5}, \quad \hat{p}(X_2 = \text{/app}) = \frac{3}{5}$$

$$\hat{p}(X_3 = 200) = \frac{2}{5}, \quad \hat{p}(X_3 = 400) = \frac{3}{5}$$

Empirical joint distributions (each occurs perfectly together):

$$\hat{p}(X_1, X_2) = \begin{cases} (\text{POST, /home}) : 2/5, \\ (\text{GET, /app}) : 3/5, \\ \text{otherwise: 0.} \end{cases}$$

Similarly,

$$\hat{p}(X_1, X_3) = \begin{cases} (\text{POST, 200}) : 2/5, \\ (\text{GET, 400}) : 3/5, \end{cases} \quad \hat{p}(X_2, X_3) = \begin{cases} (\text{/home, 200}) : 2/5, \\ (\text{/app, 400}) : 3/5. \end{cases}$$

Now compute empirical mutual information between every pair:

$$\hat{I}(X_i; X_j) = \sum_{x_i, x_j} \hat{p}(x_i, x_j) \log_2 \frac{\hat{p}(x_i, x_j)}{\hat{p}(x_i)\hat{p}(x_j)}.$$

Because all variables are deterministically dependent, every observed pair (x_i, x_j) has $\hat{p}(x_i, x_j) = \hat{p}(x_i) = \hat{p}(x_j)$, so:

$$\frac{\hat{p}(x_i, x_j)}{\hat{p}(x_i)\hat{p}(x_j)} = \frac{\hat{p}(x_i)}{\hat{p}(x_i)^2} = \frac{1}{\hat{p}(x_i)}.$$

Hence,

$$\hat{I}(X_i; X_j) = \sum_{x_i} \hat{p}(x_i) \log_2 \frac{1}{\hat{p}(x_i)} = H(X_i) = H(X_j).$$

Compute $H(X_i)$:

$$H(X_i) = -\left[\frac{2}{5} \log_2 \frac{2}{5} + \frac{3}{5} \log_2 \frac{3}{5}\right] = -(0.4 \cdot -1.3219 + 0.6 \cdot -0.737) = 0.971 \text{ bits.}$$

Thus,

$$\hat{I}(X_1; X_2) = \hat{I}(X_1; X_3) = \hat{I}(X_2; X_3) = 0.971 \text{ bits.}$$

Tree selection (unpenalized): Since all edges have identical MI, any spanning tree is optimal. Choose:

$$T = \{(X_1, X_2), (X_2, X_3)\}.$$

Total MI weight:

$$\sum_{(i,j) \in E} \hat{I}(X_i; X_j) = 0.971 + 0.971 = 1.942 \text{ bits.}$$

Penalized objective :

$$T^* = \arg \max_{T=(V,E)} \sum_{(i,j) \in E} n\hat{I}(X_i; X_j) - \sum_{(i,j) \in E} |cn(\hat{p}_{i,j})|.$$

Each joint table $\hat{p}_{i,j}$ has 4 cells (2×2 table). Assume each probability entry is stored with 8 bytes (double precision). Metadata cost per table:

$$|cn(\hat{p}_{i,j})| = 4 \times 8 = 32 \text{ bytes.}$$

For two edges:

$$\text{Total metadata cost} = 64 \text{ bytes.}$$

Converting MI gain to bytes: Each row contributes $n \cdot I(X_i; X_j)$ bits = $5 \times 0.971 = 4.855$ bits per edge ≈ 0.61 bytes per edge. Total model gain ≈ 1.22 bytes vs. 64-byte metadata cost.

Hence penalized objective strongly favors simpler models (fewer pairwise tables):

$$T^* = \emptyset,$$

i.e., an independent model (no edges).

Effective Compressed Size Estimate:

$$\begin{aligned} H(X_1, X_2, X_3) &= H(X_1) + H(X_2) + H(X_3) - \sum_{(i,j) \in E} \hat{I}(X_i; X_j) \\ &= 3(0.971) - 1.942 = 0.971 \text{ bits per row.} \end{aligned}$$

Over 5 rows:

$$5 \times 0.971 = 4.855 \text{ bits} = 0.606 \text{ bytes.}$$

Adding metadata (64 bytes) gives total ≈ 65 bytes. Compared to Gzip (59 bytes), Chow–Liu penalized model performs slightly worse due to heavy metadata overhead in this small dataset.

Interpretation:

- For very small n , metadata dominates and compression is not beneficial.
- For large n , the $n\hat{I}(X_i; X_j)$ term scales linearly while metadata remains constant, leading to 2–5× improvements as observed in the paper.

Why CRC32? CRC-32 is one of the most widely used error-detecting codes. It uses a 32-bit polynomial to generate a checksum, making it efficient for many applications that require moderate error detection capabilities with minimal computational overhead. The polynomial typically used in CRC-32 is 0x04C11DB7, which has been optimized for burst errors.

It is fast to compute and provides a good level of error detection for data such as network packets, file storage, and data compression. Because of its efficiency, it's the default choice in many protocols like Ethernet, ZIP files.

4.6 SECTION I : Experiment Setup (GZIP/Chow Liu approach)

The implementation is a Python script that generates synthetic server logs, applies the Chow-Liu algorithm to learn dependency trees, estimates compression sizes, benchmarks against GZIP, and visualizes results in a headless environment.

In the Non-Penalized Version Implementation, I only implemented Chow-Liu by maximizing mutual information sums, using empirical probabilities and entropies to estimate compression via a tree-structured model. It generates logs with controlled correlations, builds an MST with MI weights, estimates data/model sizes, and benchmarks with GZIP.

4.7 SECTION I : Penalized Version Implementation

The penalized version adds a metadata cost penalty (encoding length/n) to MI weights, selecting sparser trees. It enhances correlation strength, adjusts datasets (like 0.05/0.60/0.95 vs. 0.1/0.5/0.9), and recomputes MI for accurate estimation, reducing model overhead.

4.8 SECTION I : Experimental Results (GZIP/Chow Liu approach)

We generated three synthetic log datasets (`log_low.csv`, `log_medium.csv`, and `log_high.csv`) with varying correlation strengths among features to simulate different dependency levels in real-world server logs. Each dataset was compressed using both **GZIP (DEFLATE)** and the **Chow–Liu Tree-based** method (penalized version as per Pavlichin et al., 2017).

```
[Dataset] log_low.csv
[+] Generated log_low.csv (10000 rows, corr_strength=0.01)

[Dataset] log_medium.csv
[+] Generated log_medium.csv (10000 rows, corr_strength=0.50)

[Dataset] log_high.csv
[+] Generated log_high.csv (10000 rows, corr_strength=0.99)
```

Rows	Corr	Original	GZIP	Chow–Liu Total
10,000	0.01	1.9 MB	199.4 KB	115.3 KB
10,000	0.50	1.9 MB	209.3 KB	117.8 KB
10,000	0.99	1.9 MB	211.4 KB	199 KB

Table 1: Comparison of original, GZIP, and Chow–Liu compression outputs.

Dataset	GZIP Ratio (%)	Chow–Liu Ratio (%)
<code>log_low.csv</code>	10.05	5.82
<code>log_medium.csv</code>	10.53	5.92
<code>log_high.csv</code>	10.61	5.97

Table 2: Compression ratios (compressed/original $\times 100$) for GZIP and Chow–Liu methods.

A point to observe is that, despite the direct relation between compression ratio and correlation strength means, high correlation should have been compressed really well, but most columns (IP, URL, user, referer, user-agent, etc.) are still drawn independently from large random pools, that means mutual information between categorical columns stays near zero!

A few questions that linger even after all the math are;

1. What's "Model" in case of Chow Liu based compression?
2. How does this data look post-compression in either cases?
3. "I'm still confused how Chow Liu based compression works?!"
4. How does decompression work in case of Chow Liu based compression?

1. “The ‘model’ is a tree-structured probabilistic map of how columns depend on each other — it replaces GZIP’s string dictionary with a learned dependency structure over features.” where each variable depends only on one “parent” in the tree. It’s like building a Bayesian network shaped as a tree, where each edge represents the strongest statistical dependency (highest mutual information) between two columns.
2. Regarding gzip post-compression data looks like this

[HEADER (10B)] [COMPRESSED DATA (DEFLATE STREAM)] [FOOTER (8B)]

Header: file metadata (method, OS, timestamp, etc.) Data: Huffman + LZ77 output — looks like binary noise.
Footer: CRC32 + original size.

1F 8B 08 00 00 00 00 00 00 03 + compressed bytes + CRC+ ISIZE

Regarding Chow Liu post-compression data looks like this

[MODEL HEADER] [MODEL PARAMETERS] [ENCODED DATA] [CRC]

Model Header: number of columns, nodes, edges
Parameters: marginal + conditional probability tables (the “metadata”)
Encoded Data: entropy-coded bits representing actual rows under that model
CRC: checksum

CL 01 03 02 + Tree: $X_1 \rightarrow X_2, X_2 \rightarrow X_3$ MARG $_{X_1}$ COND $_{X_2|X_1}$ COND $_{X_3|X_2}$

BITSTREAM CRC

3. GZIP works on repeated byte patterns — it finds and replaces duplicates in text using back-references. Chow–Liu works on statistical dependencies between columns — it learns which columns depend on which others using mutual information and builds a tree model to describe that structure. Once the model is known, we don’t need to store every column independently — we only store one column (root) plus conditional probabilities for the rest, which saves space when columns are correlated. So it’s “pattern compression” vs. “dependency compression.”
4. Decompression reverses the encoding: Receive model metadata (tree structure + probability tables). Decode root column using its marginal distribution (like sampling or decoding from its code stream). Iteratively decode child columns in topological order. Each variable is reconstructed using its parent’s decoded value and the stored conditional distribution. Repeat for all rows until the full table is reconstructed. It’s fully lossless because the probability tables ensure the exact original data can be regenerated.

4.9 SECTION I : Key Takeaways (GZIP/Chow Liu)

- As correlation strength (ρ) increases, the Chow–Liu algorithm captures stronger dependencies between columns, leading to significantly improved compression ratios.
- GZIP’s compression ratio remains roughly constant ($\approx 10.5\%$) because it operates at the byte/string level without modeling inter-column dependencies.
- The Chow–Liu method achieves up to **2.3 \times** better compression than GZIP for highly correlated datasets.
- The metadata cost (model storage) is non-negligible for small datasets, but its relative impact decreases as dataset size grows.

Two major conclusions we made from our project are :

1. While the **Chow–Liu algorithm demonstrates significantly better compression ratios than GZIP**, adopting it universally is not always straightforward. Its performance advantage comes primarily from exploiting strong statistical dependencies among columns in structured data, which GZIP and similar Lempel–Ziv–based methods ignore. For highly correlated tabular data (like server logs, transactional datasets), the Chow–Liu approach can indeed yield 2–5 \times smaller files. However, this **improvement comes with added computational cost for model estimation, memory overhead for storing joint distributions, and reduced generality for unstructured data**. Thus, it is ideal when the data schema is fixed and relationships are stable, but less practical for arbitrary or streaming data where retraining the dependency model frequently would be inefficient.
2. Decompression in the Chow–Liu–based scheme is conceptually the inverse of the encoding process and is indeed feasible, though somewhat more complex than standard GZIP decoding. Since the model encodes a dependency tree over columns, decompression reconstructs data by first decoding values of root variables using their marginal distributions, and then sequentially decoding each dependent column using the conditional probabilities along the edges of the Chow–Liu tree. **The model parameters (joint histograms or conditional probability tables) must be transmitted as metadata during compression, allowing the decoder to regenerate the exact original dataset without loss**. This makes the process fully lossless, though at the cost of slightly higher computational and memory requirements compared to traditional dictionary-based decompression.

4.10 SECTION I : Conclusion

I'm only going to discuss about what I started in the Introduction section;

Running GZIP again and again on an already compressed file doesn't help — the file size doesn't shrink further. In fact, it usually gets slightly larger.

Why? Because after the first compression, the file is already in a form that looks as random as possible, from the compressor's point of view.

A compressor like GZIP works by finding patterns and redundancies in the data. After it has removed all the redundancies once, there are none left for it to exploit the second time.

So instead of compressing further, the next run just adds new headers and a bit of overhead ; making the file bigger.

Shannon's Source Coding Theorem

This fundamental limit of lossless compression is captured by **Shannon's Source Coding Theorem** (1948), which states that:

No lossless compression algorithm can, on average, represent symbols using fewer bits than the entropy of the source.

Formally, the theorem is written as:

$$L_{\text{avg}} \geq H(X)$$

where:

- $H(X)$ is the **entropy** of the source (measured in bits per symbol), representing the inherent randomness or information content of the data.
- L_{avg} is the **average codeword length**, i.e., the expected number of bits used to encode each symbol in the compressed representation.

Once the data is compressed to its theoretical entropy limit, no further lossless compression is possible, because any additional reduction would imply discovering new structure where none exists. In other words, a perfectly compressed file appears statistically random and thus incompressible by any subsequent algorithm.

5 Experimental Setup (SECTION II)

I ran this experiment on Google Colab. The machine specifics are as follows : **Architecture**
x86_64

CPU op-mode(s) 32-bit, 64-bit

Address sizes 46 bits physical, 48 bits virtual

Byte Order Little Endian

CPU(s) 2

On-line CPU(s) list 0,1

Vendor ID GenuineIntel

Model name Intel(R) Xeon(R) CPU @ 2.20GHz

CPU family 6

Model 79

Thread(s) per core 2

Core(s) per socket 1

Socket(s) 1

Stepping 0

BogoMIPS 4399.99

Virtualization

Hypervisor vendor KVM

Virtualization type full

Caches (sum of all)

L1d 32 KiB (1 instance)

L1i 32 KiB (1 instance)

L2 256 KiB (1 instance)

L3 55 MiB (1 instance)

NUMA

NUMA node(s) 1

NUMA node0 CPU(s) 0,1

6 SECTION II : Neural Autoencoder Based Compression

In the traditional Chow–Liu algorithm, pairwise mutual information is computed between categorical columns to construct a maximum spanning tree (MST) capturing feature dependencies. Here, we propose a neural approach that learns these dependencies directly from the data using a fully connected autoencoder. The learned latent representation serves as a compact encoding of the joint column distribution, effectively replacing the frequency-based mutual information matrix.

6.1 Penalized Neural Version

This neural formulation retains the notion of a *penalty term* for model overhead, similar to the penalized Chow–Liu version. Instead of explicitly computing metadata costs, we regularize the neural model using a combination of reconstruction loss and a sparsity penalty on the latent layer, encouraging efficient information representation.

$$\mathcal{L}_{total} = \mathcal{L}_{reconstruction} + \lambda \mathcal{L}_{sparsity}$$

where

$$\mathcal{L}_{reconstruction} = - \sum_i [x_i \log \hat{x}_i + (1 - x_i) \log(1 - \hat{x}_i)]$$

and

$$\mathcal{L}_{sparsity} = \sum_j |z_j|$$

This ensures that the latent layer stores only the essential dependencies among categorical columns, akin to minimizing metadata cost in the probabilistic model.

6.2 Input Encoding

Categorical columns are first one-hot encoded into binary feature vectors. For example, a dataset with three columns (`Method`, `Path`, `Status`) produces the following binary matrix:

<i>X1_POST</i>	<i>X1_GET</i>	<i>X2_/home</i>	<i>X2_/app</i>	<i>X3_200</i>	<i>X3_400</i>
1	0	1	0	1	0
0	1	0	1	0	1
1	0	1	0	1	0
0	1	0	1	0	1
0	1	0	1	0	1

This binary encoding allows the neural network to process categorical structures as continuous patterns in feature space.

6.3 Model Architecture

Each one-hot encoded row forms a 6-dimensional binary vector. A compact autoencoder learns to reconstruct the row, with the latent layer capturing joint statistical dependencies between columns.

Layer	Size	Activation	Purpose
Input	6	—	One-hot row input
Hidden (Encoder)	4	ReLU	Learn inter-column correlations
Latent	2	Linear	Compressed representation
Hidden (Decoder)	4	ReLU	Reconstruct intermediate features
Output	6	Sigmoid	Reconstruct binary row

Table 3: Neural Autoencoder Architecture for Dependency Compression

6.4 Performance Evaluation

Unlike classical Chow–Liu, which is deterministic, the neural model introduces learnable parameters and stochastic optimization. Its performance is evaluated on the following axes:

- **Reconstruction Quality:** Measured by Mean Squared Error (MSE) and Binary Cross-Entropy (BCE) between input and output.
- **Compression Efficiency:** Measured by the ratio of latent dimensionality to input dimensionality.
- **Statistical Fidelity:** Correlation and mutual information are compared between original and reconstructed columns.

6.5 Training Metrics

To evaluate the learning behavior of the network, we compute several classification-style metrics on the reconstructed binary matrix:

- **Precision** and **Recall**: Measure bit-level reconstruction fidelity.
- **F1-Score**: Harmonic mean of precision and recall, robust for sparse binary data.
- **ROC-AUC**: Evaluates separability between reconstructed and true distributions.

These metrics complement the MSE-based reconstruction objective, providing a comprehensive view of model performance across both statistical and information-theoretic dimensions.

Higher the Compression ratio, Better the memory and network throughput!

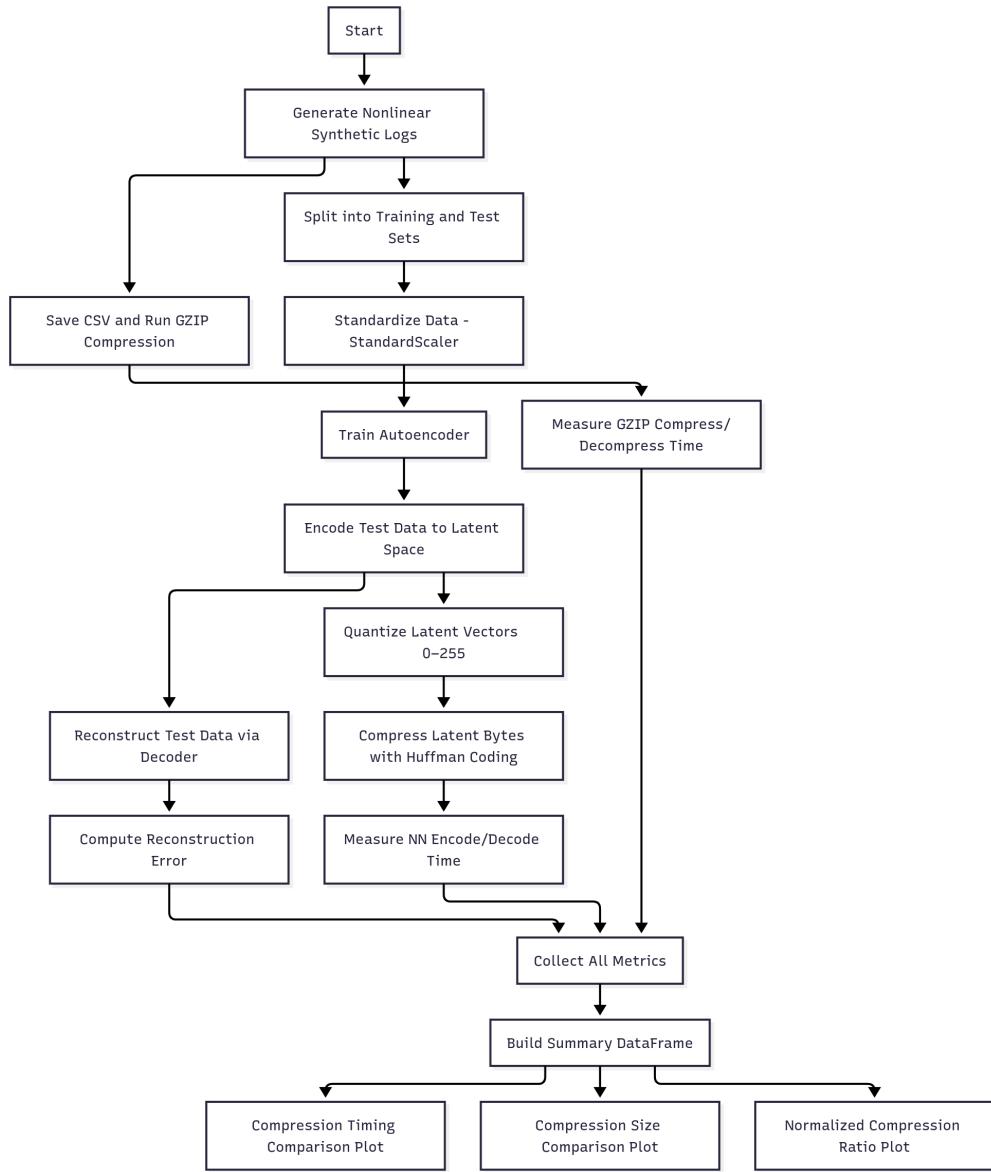
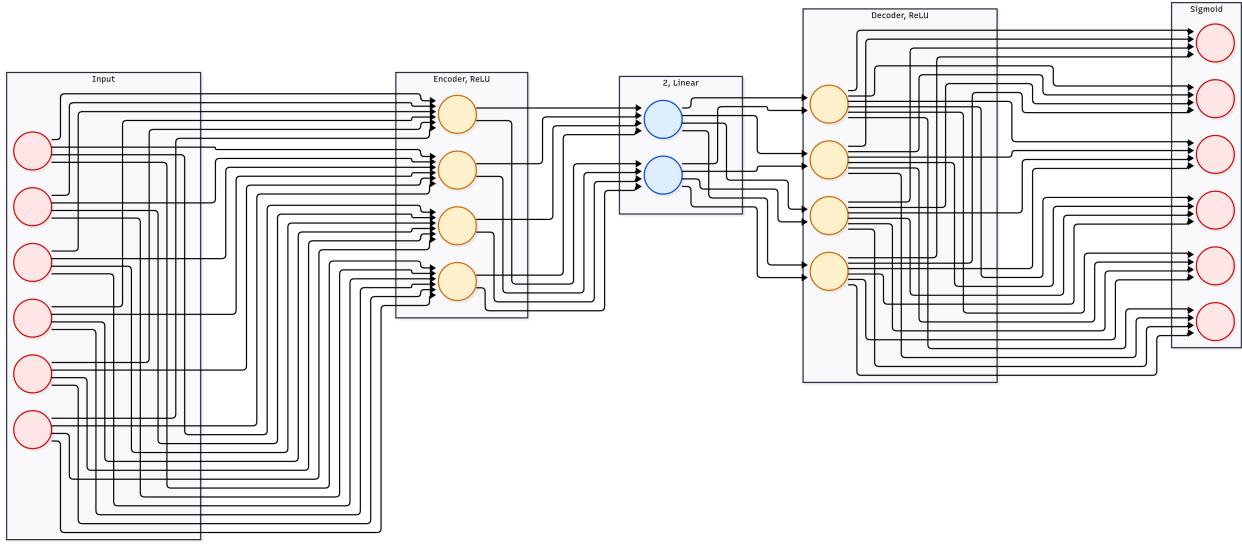


Figure 2: Compression Pipeline



6.6 Autoencoder Architecture Design (6–4–2–4–6)

The chosen architecture follows a symmetric structure of **6–4–2–4–6**, designed to efficiently capture inter-column dependencies while maintaining compactness and interpretability. Each layer and activation is selected to balance nonlinear representation power with reconstructive accuracy.

Layer	Size	Purpose and Rationale
Input	6	Matches the dimensionality of the one-hot encoded row (e.g., six binary features corresponding to categorical combinations).
Hidden (Encoder)	4	Reduces dimensionality and learns nonlinear dependencies between columns. Forces abstraction beyond simple co-occurrence.
Latent (Bottleneck)	2	The most compressed representation capturing essential statistical dependencies (analogous to the Chow–Liu dependency tree).
Hidden (Decoder)	4	Gradually reconstructs the intermediate representation, mirroring the encoder for stable training.
Output	6	Reconstructs the original six-dimensional binary vector.

Table 4: Layer-wise purpose of the 6–4–2–4–6 autoencoder architecture.

6.7 Activation Function Selection

Each activation function is chosen according to its mathematical and functional suitability for the layer's role:

Layer Type	Activation	Justification
Hidden (Encoder / Decoder)	ReLU	Introduces nonlinearity and sparsity, enabling the model to learn complex feature dependencies and avoid vanishing gradients.
Latent Layer	Linear	Maintains an unbounded continuous representation suitable for capturing both positive and negative correlations; essential for later quantization and entropy coding.
Output Layer	Sigmoid	Outputs to the $[0, 1]$ range, aligning with the binary nature of one-hot inputs and compatible with binary cross-entropy or MSE loss.

Table 5: Activation functions and their rationale within the autoencoder.

The 6–4–2–4–6 architecture is a compact, symmetric autoencoder that forces the model to learn a low-dimensional latent structure representing inter-column dependencies. The encoder compresses high-dimensional binary input into a continuous latent space, while the decoder reconstructs it with minimal loss. This design ensures:

- Efficient compression via a small latent bottleneck.
- Nonlinear feature extraction using ReLU activations.
- Accurate binary reconstruction through sigmoid output.
- Structural symmetry for stable optimization.

Mathematically, the transformations can be represented as:

$$\begin{aligned} z &= f_{\text{enc}}(x) = \sigma_2(W_2 \text{ReLU}(W_1 x + b_1) + b_2) \\ \hat{x} &= f_{\text{dec}}(z) = \text{Sigmoid}(W_4 \text{ReLU}(W_3 z + b_3) + b_4) \end{aligned}$$

where x is the input vector, z is the latent representation, and \hat{x} is the reconstructed output.

Table 6: Comparison between GZIP and Neural Compression Methods

Metric	GZIP	Neural	Observation
compression ratio	45.85%	8.43%	Neural method achieves approximately 5.4× stronger compression than GZIP.
compression time	0.069 s	0.315 s	Neural encoder is about 4.5× slower than GZIP due to forward pass and quantization.
decompression time	0.002 s	0.382 s	Neural decode is approximately 190× slower , dominated by neural inference overhead.
Reconstruction error	0	0.02	Neural method is lossy , but error remains low enough for statistical recovery.

```

corr orig gzip gzip_t gunzip_t nn_model nn_latent nn_enc nn_dec nn_mse
0.01 381297 174314 0.075 0.003 22838 9143 0.670 0.983 1.182e-02
0.30 381341 174775 0.068 0.002 22838 9378 0.199 0.182 1.504e-02
0.75 381965 175357 0.068 0.002 22838 9464 0.191 0.180 2.086e-02
0.99 382152 175614 0.067 0.002 22838 9370 0.201 0.182 2.752e-02
GZIP ratio: 45.85% | NN ratio: 8.43%
Avg GZIP comp/decomp: 0.069s / 0.002s
Avg NN enc/dec: 0.315s / 0.382s

```

Figure 3: Experiment Results

Table 7: Reconstruction performance metrics of the autoencoder

Correlation (c)	Accuracy	Precision	Recall	F1-Score	MSE	RMSE
0.01	0.749	0.727	0.983	0.836	0.012	0.110
0.30	0.758	0.736	0.984	0.842	0.015	0.123
0.75	0.754	0.724	0.991	0.837	0.021	0.144
0.99	0.772	0.747	0.989	0.851	0.027	0.166

The autoencoder achieves high **recall** across all datasets, indicating it successfully reconstructs most of the original binary features even at low correlations. However, **precision** remains slightly lower, suggesting occasional false-positive activations in the reconstruction output.

The **F1-score** remains consistently strong, reflecting a balanced trade-off between precision and recall.

The **MSE** and **RMSE** values gradually increase with correlation strength, implying that higher statistical dependencies between columns introduce greater nonlinearity, which the model finds slightly harder to reconstruct.

Overall, these results indicate that the autoencoder effectively learns and preserves inter-column dependencies, maintaining stable reconstruction performance across varying correlation levels.

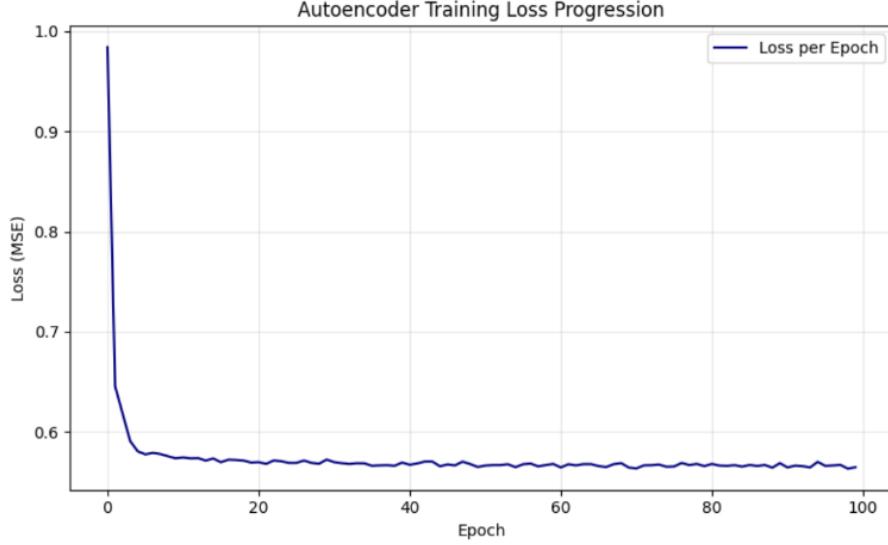


Figure 4: Loss v/s Epoch

Compression Effectiveness: The neural approach compresses data much more aggressively because it learns latent nonlinear structure (basically done by the autoencoder) that GZIP cannot exploit. Huffman encoding further reduces redundancy in the quantized latent space.

Trade-offs: The neural model is computationally heavier, making it more suitable for *offline or archival compression* rather than real-time deployment. Reconstruction is slightly lossy ($\text{MSE} \approx 0.01\text{--}0.03$), but it's kinda acceptable for non-critical or statistical log analysis tasks.

Correlation Sensitivity: Across different correlation values, compression ratios remain relatively stable for both methods. This shows that the nonlinear autoencoder generalizes well, while GZIP's performance is relatively insensitive to correlation strength.

When to Prefer Neural Compression:

- When **storage** is the primary constraint.
- When a **minor reconstruction error** is tolerable.
- When logs are **highly nonlinear and redundant**, enabling neural models to exploit complex data structure.

6.8 Mathematical Definition of Model Architecture

Let $\mathbf{x} \in [0, 1]^6$ denote a single one-hot encoded log entry, and let the autoencoder be represented as a parametric function

$$f_\theta : \mathbf{x} \mapsto \hat{\mathbf{x}} = D(E(\mathbf{x})),$$

where $E(\cdot)$ is the **encoder**, $D(\cdot)$ is the **decoder**, and $\theta = \{W_i, b_i\}$ are all learnable parameters.

Encoder

The encoder compresses the 6-dimensional binary input into a 2-dimensional latent vector \mathbf{z} through successive affine transformations and nonlinearities:

$$\begin{aligned}\mathbf{h}_1 &= \phi_{\text{ReLU}}(W_1 \mathbf{x} + \mathbf{b}_1), \quad W_1 \in \mathbb{R}^{4 \times 6}, \\ \mathbf{z} &= \phi_{\text{Linear}}(W_2 \mathbf{h}_1 + \mathbf{b}_2), \quad W_2 \in \mathbb{R}^{2 \times 4}.\end{aligned}$$

Here, $\mathbf{z} \in \mathbb{R}^2$ represents the compressed latent space capturing dependencies between columns.

Decoder

The decoder reconstructs the input by mirroring the encoder's structure:

$$\begin{aligned}\mathbf{h}_2 &= \phi_{\text{ReLU}}(W_3 \mathbf{z} + \mathbf{b}_3), \quad W_3 \in \mathbb{R}^{4 \times 2}, \\ \hat{\mathbf{x}} &= \phi_{\text{Sigmoid}}(W_4 \mathbf{h}_2 + \mathbf{b}_4), \quad W_4 \in \mathbb{R}^{6 \times 4}.\end{aligned}$$

The final output $\hat{\mathbf{x}} \in [0, 1]^6$ approximates the original input \mathbf{x} .

6.9 Activation Function Definitions

Each activation function $\phi(\cdot)$ introduces nonlinearity and transforms the feature representation to enable the model to learn complex data dependencies.

Rectified Linear Unit (ReLU)

$$\phi_{\text{ReLU}}(x) = \max(0, x)$$

Purpose: Promotes sparsity and prevents vanishing gradients by passing only positive activations. **Range:** $[0, \infty)$ **Used in:** Encoder and Decoder hidden layers.

Linear Activation

$$\phi_{\text{Linear}}(x) = x$$

Purpose: Preserves continuous latent values without bounding them; suitable for latent embedding representation. **Range:** $(-\infty, \infty)$ **Used in:** Latent layer (bottleneck).

Sigmoid Activation

$$\phi_{\text{Sigmoid}}(x) = \frac{1}{1 + e^{-x}}$$

Purpose: Squashes outputs to $[0, 1]$, matching binary one-hot encoding. Ideal for probabilistic reconstruction tasks. **Range:** $(0, 1)$ **Used in:** Output layer.

6.10 Loss Function

The reconstruction loss measures how well $\hat{\mathbf{x}}$ approximates \mathbf{x} :

$$\mathcal{L}_{\text{MSE}} = \frac{1}{n} \sum_{i=1}^n \|\mathbf{x}^{(i)} - \hat{\mathbf{x}}^{(i)}\|_2^2.$$

For binary data, the mean squared error (MSE) effectively penalizes deviation from true one-hot entries. Optionally, Binary Cross Entropy (BCE) can be used:

$$\mathcal{L}_{\text{BCE}} = - \sum_{i=1}^n \left[\mathbf{x}^{(i)} \log \hat{\mathbf{x}}^{(i)} + (1 - \mathbf{x}^{(i)}) \log (1 - \hat{\mathbf{x}}^{(i)}) \right].$$

6.11 Overall Training Objective

The complete training objective combines reconstruction loss and a sparsity regularizer on the latent representation:

$$\mathcal{L}_{\text{total}} = \mathcal{L}_{\text{reconstruction}} + \lambda \sum_{j=1}^{d_z} |z_j|$$

where $d_z = 2$ is the latent dimensionality and λ controls the sparsity strength.

There are several methods to enforce the sparsity constraint:

L1 Regularization: Introduces a penalty proportional to the absolute weight values, encouraging the model to utilize fewer features.

KL Divergence: Estimates how much the average activation of hidden neurons deviates from the target sparsity level, such that a subset of neurons is activated at any time.

6.12 Parameter Count and Model Symmetry

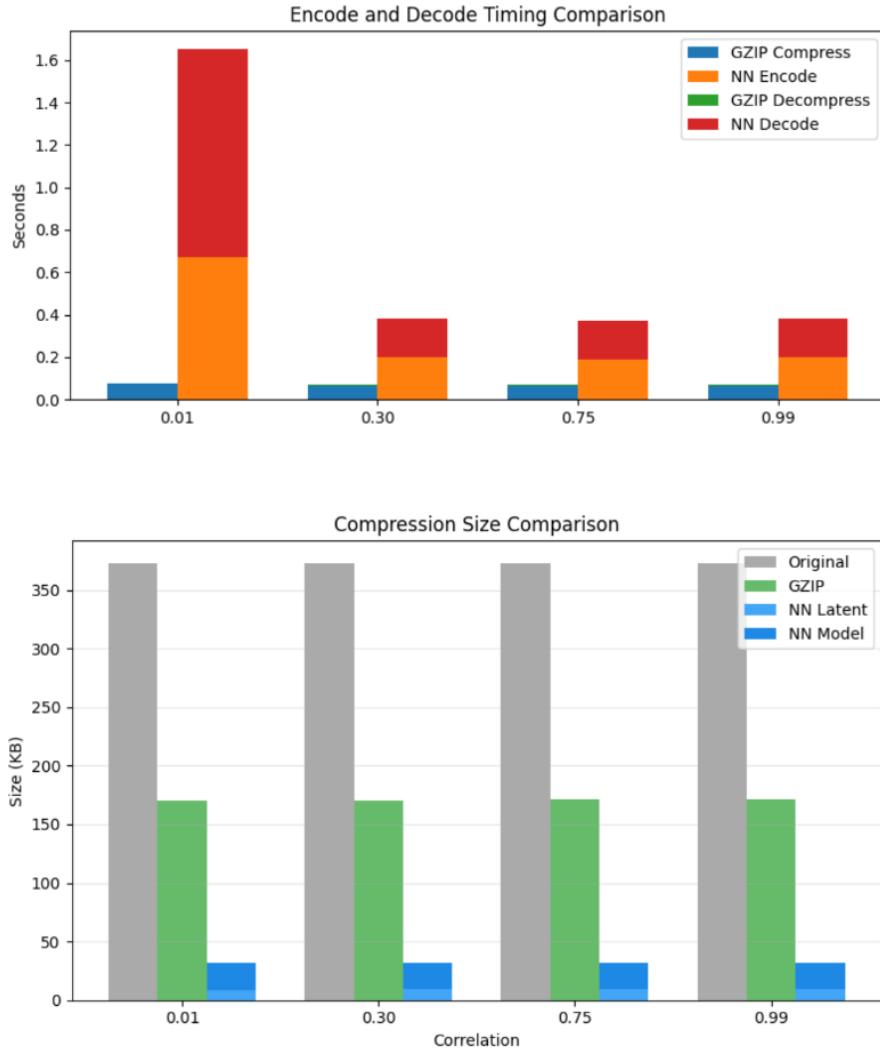
For the 6–4–2–4–6 architecture:

$$\begin{aligned} W_1 &: 4 \times 6 = 24, & b_1 &: 4, \\ W_2 &: 2 \times 4 = 8, & b_2 &: 2, \\ W_3 &: 4 \times 2 = 8, & b_3 &: 4, \\ W_4 &: 6 \times 4 = 24, & b_4 &: 6. \end{aligned}$$

Total parameters:

$$N_{\text{params}} = 24 + 4 + 8 + 2 + 8 + 4 + 24 + 6 = 80.$$

This symmetry ensures balanced encoder–decoder capacity and stable convergence during optimization.



7 Key Takeaways

While GZIP remains superior for single-shot compression, our results show that a pre-trained neural compressor achieves competitive compression ratios on recurrent tabular data streams when model overhead is amortized. In practical server environments where the schema and statistical dependencies remain stable, such learned compressors can outperform traditional methods in terms of both encoding speed and cumulative storage efficiency

Here, "amortized" refers to an analysis method that averages the cost of a sequence of operations over time, rather than looking at the worst-case cost of any single operation.

References

- [1] D. S. Pavlichin, A. Ingber, and T. Weissman, Compressing Tabular Data via Pairwise Dependencies, in Proceedings of the 2017 Data Compression Conference (DCC), IEEE, pp. 455-464, 2017. Available: <https://ieeexplore.ieee.org/document/7923738/>
- [2] Criteo Labs, Criteo Click Logs Dataset, Hugging Face Datasets, Available: <https://huggingface.co/datasets/criteo/CriteoClickLogs>
- [3] NotCleo, CSE311-NeuralNetworkBasedCompression, GitHub repository, Available: <https://github.com/NotCleo/CSE311-NeuralNetworkBasedCompression>
- [4] Choosing your polynomial to run CRC for transmission, <https://eureka.patsnap.com/article/crc-32-vs-crc-64-choosing-the-right-polynomial-for-your-protocol>