IMA312 : Compressing Tabular Data via Pairwise Dependencies

Amruth Ayaan Gulawani

October 28, 2025

Contents

1 Introduction

This document is a report containing our analysis on a research paper that claimed to have outperformed GZIP in compression in tabular datasets. We have tested out this claim via demo and a crude mathematical analysis of the algorithm involved.

2 Abstract

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:

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

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.

3 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)

3.1 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.

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**

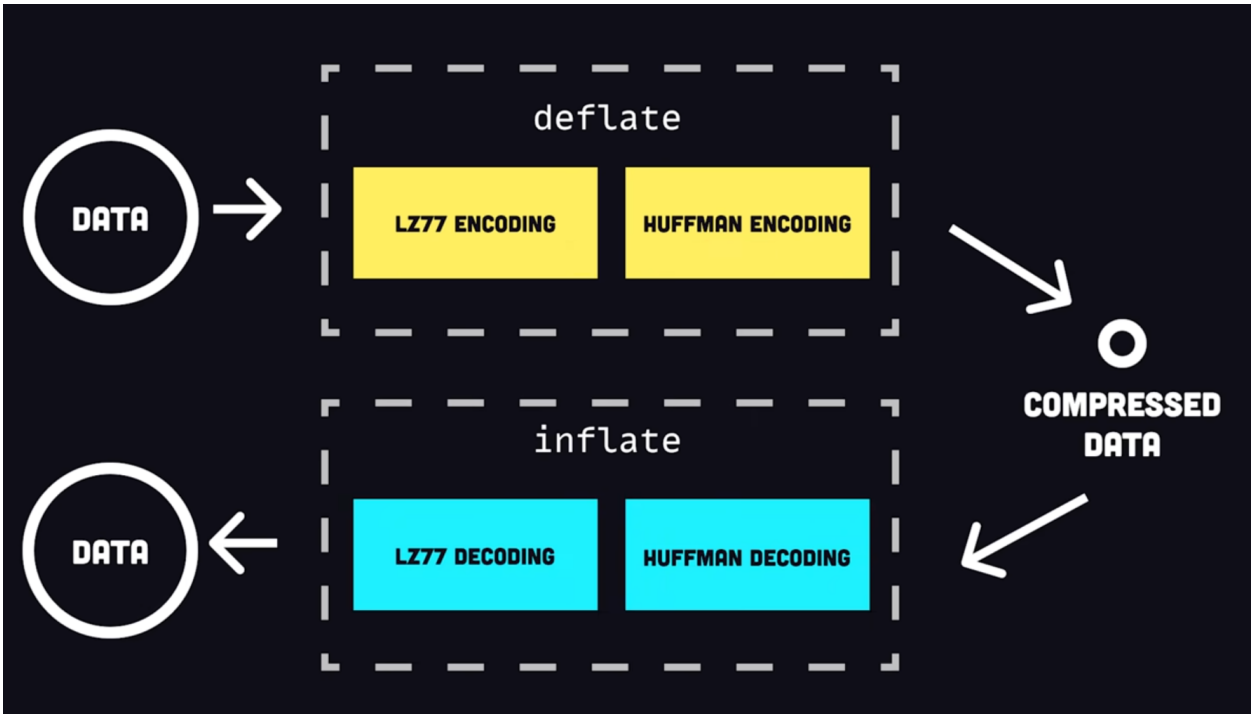

Figure 1: GZIP Algorithm

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.

3.2 Chow–Liu Algorithm Based Compression

A one phrase summary would be "Learning a Tree-Structured Model"
We are given k discrete random variables $X_1, \ldots, X_k$, representing the columns (features) of a tabular dataset (server logs). We observe n i.i.d. samples:

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

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):

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

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:

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

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

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.

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

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:

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

and

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

Substituting these empirical estimators into the log-likelihood yields:

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

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 $(j, \pi(j))$:

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

Combine the terms:

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

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:

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

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

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

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:

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

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

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

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 $\hat{I}(X_i; X_j) - \frac{1}{n} \cdot$ (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 GZIP v/s Chow-Liu Algorithm

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

(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/foot (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:

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

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

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

Empirical joint distributions (each occurs perfectly together):

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

Similarly,

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

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 $H(X_i)$:

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

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 (Pavlichin et al. 2017):

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

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):

$$T^* = \varnothing,$$

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.

5 Experiment Setup

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.

5.1 Non-Penalized Version Implementation

The non-penalized version implements 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.

5.2 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.

6 Experimental Results

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 (%)
`log_low.csv`	10.05	5.82
`log_medium.csv`	10.53	5.92
`log_high.csv`	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?

To address these four significant questions,

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: X1$\rightarrow X2, X2 \rightarrow X3$ MARG_{X1} $\text{COND}_{X2|X1}\text{COND}_{X3|X2}$

   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.

7 Key Takeaways

- 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 $\mathbf{2.3}\times$ 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.

8 Conclusion

This entire project invoked the following question; "Why not run a file through GZIP, which is already a pretty popular and standardized compression technique and get a com- pressed 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?".

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.

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, IMA312-ChowLiuCompression, GitHub repository, Available: https://github.com/NotCleo/IMA312-ChowLiuCompression