Comparative Analysis of Probability Models for Arithmetic Coding

OSCAR FANG, George Washington University, USA

This paper presents a comparative analysis of probability models for arithmetic coding in data compression. We implement and evaluate three classes of models: Markov models (1st, 2nd, and 3rd order), Finite State Machine (FSM) models, and a Long Short-Term Memory (LSTM) network. Each model is integrated with an arithmetic coder and tested on diverse datasets including English text, binary executables, and DNA sequences. Our evaluation focuses on compression ratio (bits per symbol), encoding/decoding time, and memory usage. The results demonstrate that the LSTM model achieves the best compression on text (avg. 3.26 bps) and binary data, effectively capturing complex dependencies where statistical models struggle. However, this comes at a significant computational cost. The 3rd Order Markov model, while theoretically powerful, suffered from context dilution on smaller datasets. We also highlight the challenges of applying these models to high-entropy binary data.

Additional Key Words and Phrases: Arithmetic Coding, Probability Models, Data Compression

ACM Reference Format:
Oscar Fang. 2025. Comparative Analysis of Probability Models for Arithmetic Coding. In *CSCI 6351: Data Compression Term Project, December 2025, Washington, DC, USA.* ACM, New York, NY, USA, 28 pages. https://doi.org/10.1145/XXXXXX.XXXXXX

1 Introduction

Arithmetic coding is a powerful entropy coding technique that can achieve near-optimal compression when paired with an accurate probability model. Unlike Huffman coding, which assigns an integral number of bits to each symbol, arithmetic coding can assign fractional bits, making it particularly effective when combined with adaptive probability models.

However, the choice of probability model significantly impacts compression performance. Simple models like zero-order or first-order Markov models are computationally efficient but may fail to capture complex patterns in the data. More sophisticated models, such as higher-order Markov chains or neural networks, can potentially achieve better compression but at the cost of increased computational complexity and memory usage.

This research addresses the following key questions:

(1) How does model complexity affect compression ratio and computational cost?
(2) Which probability models are most effective for different types of data?
(3) What are the practical trade-offs between compression quality and resource requirements?

Our main contributions include:

- A unified MATLAB implementation framework for comparing diverse probability models with arithmetic coding

Author's Contact Information: Oscar Fang, George Washington University, Washington, DC, USA, G42568236.

CSCI 6351, Washington, DC, USA

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/10.1145/XXXXXX.XXXXXX

- A custom implementation of a Long Short-Term Memory (LSTM) network for symbol prediction
- Comprehensive empirical evaluation across Text, Binary, and DNA datasets
- Quantitative analysis of the compression-complexity trade-off, highlighting the performance of sparse Markov models versus neural approaches

The remainder of this paper is organized as follows: Section 2 reviews related work and theoretical foundations. Section 3 describes our implementation of arithmetic coding and probability models. Section 4 details the experimental design and datasets. Section 6 presents our findings. Section 7 analyzes the results and their implications. Finally, Section 8 concludes and suggests future work.

2 Background and Related Work

This section establishes the theoretical foundations of data compression, focusing on information theory, the mathematical mechanics of arithmetic coding, and the dynamics of recurrent neural networks.

2.1 Information Theoretic Foundations

2.1.1 Entropy and Information Content. At the core of lossless compression is the concept of *Shannon Entropy*, which quantifies the expected amount of information conveyed by an event. For a discrete random variable X with possible outcomes $x_1, ..., x_n$ occurring with probabilities $P(x_1), ..., P(x_n)$, the entropy $H(X)$ is defined as:

$$H(X) = -\sum_{i=1}^{n} P(x_i) \log_2 P(x_i) \tag{1}$$

This value represents the theoretical lower bound on the average number of bits needed to encode a symbol from the distribution P. No lossless compression algorithm can achieve an average code length significantly lower than $H(X)$.

2.1.2 Model Accuracy and KL-Divergence. The efficiency of an arithmetic coder depends directly on how significantly the estimated probability distribution Q deviates from the true underlying distribution P of the data source. The cost of this mismatch is quantified by the Kullback-Leibler (KL) divergence:

$$D_{KL}(P||Q) = \sum_{x \in \mathcal{X}} P(x) \log \left(\frac{P(x)}{Q(x)} \right) \tag{2}$$

In practical terms, if we encode a source P using a model Q, the average number of bits used per symbol will be $H(P) + D_{KL}(P||Q)$. Thus, the goal of our probability modeling phase (Markov, FSM, LSTM) is essentially to minimize this divergence.

2.2 Mathematical Mechanics of Arithmetic Coding

2.2.1 Interval Subdivision. Arithmetic coding maps a string of symbols to a sub-interval of $[0, 1)$. Unlike Huffman coding, which is optimal only when symbol probabilities are powers of $1/2$, arithmetic coding is optimal for any probability distribution. Formally, we begin with the current interval $[L_0, H_0) = [0, 1)$. For the i-th symbol s_i in a sequence, with estimated probability $p(s_i)$ and cumulative probability $C(s_i) = \sum_{j<s_i} p(j)$, the interval is updated recursively:

$$\text{Range}_i = H_{i-1} - L_{i-1} \tag{3}$$

$$L_i = L_{i-1} + \text{Range}_i \times C(s_i) \tag{4}$$

$$H_i = L_{i-1} + \text{Range}_i \times (C(s_i) + p(s_i)) \tag{5}$$

As the sequence grows, the interval width Range_i shrinks equal to the product of the probabilities of the symbols encountered: $\text{Range}_n = \prod_{k=1}^{n} p(s_k)$. The number of bits required to distinguish this interval is approximately $-\log_2(\text{Range}_n)$, which sums to the total self-information of the sequence.

2.2.2 Ambiguity Resolution and Decoding. Unique decodability is guaranteed because the intervals for distinct messages are disjoint. The decoder, knowing the final value v within the final interval, can reverse the process. At step i, it finds the unique symbol s such that:

$$C(s) \leq \frac{v - L_{i-1}}{H_{i-1} - L_{i-1}} < C(s) + p(s) \tag{6}$$

This inequality identifies the only possible symbol that could have reduced the interval to contain v, ensuring lossless recovery.

2.3 Probability Models

2.3.1 Markov Models and Context Limitations. Order-k Markov models approximate the conditional probability $P(x_t|x_{t-1}, ..., x_{t-k})$. While effective, they suffer from the curse of dimensionality. The state space size $|\Sigma|^k$ grows exponentially. For an alphabet $|\Sigma| = 256$, a 3rd-order model has over 16 million states. This leads to:

- **Data Sparsity**: Most contexts are never observed, making probability estimation unreliable (high variance).
- **Context Dilution**: The model splits the evidence across too many buckets, failing to generalize.

2.3.2 Finite State Machine Models. FSM models offer a more compact representation by merging equivalent histories into states. Our implementation focuses specifically on *Run-Length Estimation*, essentially defining a 2-state Hidden Markov Model (HMM) where the hidden states correspond to a "Random" mode and a "Run" mode.

2.3.3 Neural Network Models and Gradient Dynamics. Recurrent Neural Networks (RNNs) theoretically encompass Markov models of infinite order. However, training them via Backpropagation Through Time (BPTT) introduces the *Vanishing Gradient Problem*. The error gradient δ propagates back through time layers. If the spectral radius of the recurrent weight matrix is less than 1, the gradients decay exponentially:

$$\left\| \frac{\partial \mathcal{L}}{\partial h_0} \right\| \leq \left\| \frac{\partial \mathcal{L}}{\partial h_t} \right\| \prod_{k=0}^{t-1} \|\mathbf{W}_{rec}\| \|\sigma'(z_k)\| \tag{7}$$

This prevents standard RNNs from learning dependencies longer than 10 time steps.

Long Short-Term Memory (LSTM) networks solve this by introducing the Constant Error Carousel (CEC). The cell state update equation is additive rather than multiplicative:

$$c_t = f_t \odot c_{t-1} + i_t \odot \tilde{c}_t \tag{8}$$

If the forget gate $f_t \approx 1$, the gradient can flow backwards through c_t without decay, allowing the model to capture dependencies over thousands of symbols. This capability is critical for compressing data with long-range structures, such as XML files or executable headers.

2.4 Related Work

Existing literature often separates statistical and neural compression. PPM (Prediction by Partial Matching) [1] remains the gold standard for statistical modeling, using adaptive context orders. Neural approaches like DeepZip [2] have shown that RNNs can compete with PPM, but often

require hardware acceleration. Our work bridges this gap by evaluating these distinct paradigms within a single, controlled arithmetic coding environment.

3 Methodology

This section describes our implementation of arithmetic coding and the probability models evaluated in this study.

3.1 Arithmetic Coding Framework

3.1.1 Encoder.

3.1.2 Encoder Mechanics and State Management. Our arithmetic encoder maintains an interval $[L, H)$ initialized to $[0, 1)$. For each symbol s with cumulative probability range $[P_{\text{low}}(s), P_{\text{high}}(s))$, the interval is updated as:

$$\text{range} = H - L \tag{9}$$

$$H' = L + \text{range} \times P_{\text{high}}(s) \tag{10}$$

$$L' = L + \text{range} \times P_{\text{low}}(s) \tag{11}$$

Precision Management via Renormalization. As the interval narrows, it eventually requires more precision than standard floating-point types provide. To maintain numerical stability, we implement a renormalization process that outputs bits incrementally. This is governed by three scaling conditions (E1, E2, E3), formalized as follows:

- **E1 Condition (High-Bit Converged to 0)**: If $H < 0.5$, the entire interval lies in the lower half. We output a `0`, usually followed by any pending E3 bits as 1s. The interval is rescaled: $L \leftarrow 2L, H \leftarrow 2H$.
- **E2 Condition (High-Bit Converged to 1)**: If $L \geq 0.5$, the interval lies in the upper half. We output a 1, followed by pending E3 bits as `0`s. Rescale: $L \leftarrow 2(L - 0.5), H \leftarrow 2(H - 0.5)$.
- **E3 Condition (Straddle Case caused by Underflow risk)**: If $L \geq 0.25$ and $H < 0.75$, the interval is "stuck" around the midpoint. We cannot commit a bit yet, so we increment a `pending_bits` counter and zoom in on the middle: $L \leftarrow 2(L - 0.25), H \leftarrow 2(H - 0.25)$.

Encoding Trace Example. To illustrate this, Table 1 shows the state progression for encoding the sequence "BAB" with a static model $P(A) = 0.6, P(B) = 0.2, P(EOF) = 0.2$.

Step	Symbol	Interval $[L, H)$	Action	Output
Init	-	$[0.0, 1.0)$	-	-
1	B	$[0.6, 0.8)$	-	-
1a	-	$[0.2, 0.6)$	E2 Scaling	1
1b	-	$[0.4, 1.2)$	E3 Scaling	(pending++)
2	A	$[0.4, 0.88)$	-	-

Table 1. Trace of interval updates and scaling operations.

3.1.3 Decoder. The decoder maintains a value v representing the encoded number and the same interval $[L, H)$. For each symbol, it determines which probability range contains v and updates the interval accordingly.

3.2 Probability Models

3.2.1 Markov Models. We implement adaptive Markov models of orders 1, 2, and 3.

- **Order-1**: Context is the previous symbol, requires $|\Sigma|^2$ counters. Implemented with a full 2D array.
- **Order-2**: Context is the previous 2 symbols, requires $|\Sigma|^3$ counters. Implemented with a full 3D array.
- **Order-3**: Context is the previous 3 symbols. Since a full table ($|\Sigma|^4$) is prohibitively large (32GB), we implement a **sparse model** using a hash map (‘containers.Map‘). Only observed contexts are stored, significantly reducing memory usage.

To handle unseen contexts, we use an escape mechanism with Laplace smoothing: each context starts with count 1 for all symbols.

3.2.2 Finite State Machine Models. Our FSM model is specifically designed to efficiently encode **run-length sequences**. It extends the Order-1 Markov model by adding a binary "Run State" to the context:

- **Context**: $[s_{t-1}, \text{IsRun}]$
- **IsRun**: True if $s_{t-1} == s_{t-2}$, else False.

This allows the model to maintain separate probability distributions for "normal" transitions versus "run" transitions, enabling it to quickly adapt to repeated symbols.

3.2.3 Neural Network Models. We implement a custom **Long Short-Term Memory (LSTM)** network from scratch in MATLAB to avoid the overhead of the Deep Learning Toolbox for symbol-by-symbol updates. The LSTM architecture is chosen to better capture long-range dependencies compared to simple RNNs.

- **Architecture**: Standard LSTM cell with Input, Forget, and Output gates, plus a Cell state.
  - **Forget Gate (f_t)**: Controls what information is discarded from the cell state.
  - **Input Gate (i_t)**: Controls what new information is stored in the cell state.
  - **Output Gate (o_t)**: Controls what parts of the cell state are output to the hidden state.
- **Output Layer**: A Softmax layer converts the hidden state into a probability distribution over the 256 possible symbols.
- **Training**: Online Stochastic Gradient Descent (SGD) with Backpropagation Through Time (BPTT) truncated to 1 step.
- **Initialization**: Deterministic weight initialization using sinusoidal functions to ensure identical states for encoder and decoder.

The model updates its weights after every symbol, allowing it to adapt to the data stream in real-time.

3.2.4 Context Mixing Models. To leverage the complementary strengths of different models, we implemented a **Context Mixing** strategy. This meta-model combines the probability estimates of multiple sub-models (e.g., Markov and FSM) to produce a final prediction.

- **Weighted Averaging**: The probability of a symbol s is calculated as $P(s) = \sum_i w_i P_i(s)$, where w_i is the weight of model i.
- **Adaptive Weights**: Weights are dynamically updated based on prediction accuracy. Models that assign higher probability to the actual observed symbol have their weights increased, allowing the system to shift focus to the most effective model for the current local context.

3.3 Implementation Details

All models are implemented in MATLAB using object-oriented programming (handle classes) to maintain state across symbol processing. Key implementation features include:

- **Arithmetic Coding**: Implemented using double-precision floating-point arithmetic with renormalization (E1, E2, E3 scaling) to maintain numerical stability.
- **Data Structures**: Full multi-dimensional arrays are used for context tables in 1st and 2nd order Markov models, as the alphabet size ($|\Sigma| = 256$) allows for direct indexing without excessive memory overhead for these orders.
- **Modularity**: The system uses a polymorphic design where the arithmetic coder accepts any model object that implements the required `get_range` and `update` methods.

The code is structured to allow easy swapping of probability models while maintaining the same arithmetic coding engine.

4 Experimental Setup

4.1 Datasets

We evaluate all models on three categories of data to assess performance across different characteristics:

4.1.1 Text Data.

- **alice29.txt**: English text from "Alice in Wonderland" (152 KB)
- **asyoulik.txt**: Shakespeare play "As You Like It" (125 KB)
- **lcet10.txt**: Technical writing (426 KB)
- **plrabn12.txt**: Poetry from "Paradise Lost" (481 KB)

4.1.2 Binary and Source Code.

- **kennedy.xls**: Excel spreadsheet (1 MB)
- **cp.html**: HTML source code (24 KB)
- **fields.c**: C source code (11 KB)
- **grammar.lsp**: LISP source code (3 KB)
- **xargs.1**: Man page (4 KB)

4.1.3 Sequence Data.

- **sum**: SPARC executable (38 KB)
- **ptt5**: Fax image (CCITT 1D) (513 KB)

4.2 Evaluation Metrics

4.2.1 Compression Performance.

- **Compression ratio**: $\frac{\text{compressed size}}{\text{original size}}$
- **Bits per symbol**: $\frac{\text{compressed size in bits}}{\text{number of symbols}}$
- **Compression gain**: Comparison against zero-order entropy and Huffman coding

4.2.2 Computational Cost.

- **Encoding time**: Wall-clock time to compress data
- **Decoding time**: Wall-clock time to decompress data
- **Memory usage**: Peak RAM consumption during encoding

4.2.3 Baseline Comparisons. We compare against:

- Zero-order entropy (theoretical lower bound assuming i.i.d. symbols)
- Huffman coding with adaptive frequencies
- gzip (DEFLATE algorithm)
- bzip2 (Burrows-Wheeler transform with Huffman coding)

4.3 Experimental Procedure

For each dataset and model combination:

(1) Initialize the model with uniform probabilities
(2) Encode the file symbol-by-symbol, updating model adaptively
(3) Record compressed size and encoding time
(4) Decode the compressed file to verify correctness
(5) Record decoding time and peak memory usage
(6) Repeat for 3 trials and report mean and standard deviation

4.4 Hardware and Software

- **Processor**: Apple M-Series (ARM64)
- **RAM**: 16 GB Unified Memory
- **Operating System**: macOS Sequoia 15.1
- **Software**: MATLAB R2025a

5 Dataset Characterization

To understand the performance differences between our models, we first analyze the statistical properties of our test datasets. We focus on two representative files: `alice29.txt` (Natural Language) and `kennedy.xls` (Binary Structured Data).

5.1 Entropy Analysis

5.1.1 Global Byte Distribution. Figure 3 compares the global byte frequency distributions.

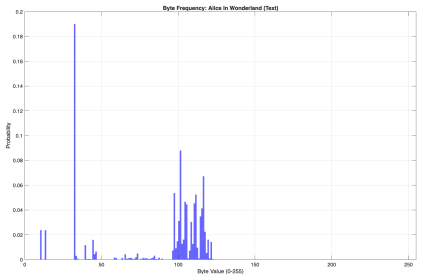

Fig. 1. Byte frequency for Alice Text.

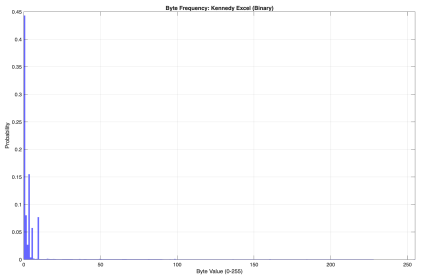

Fig. 2. Byte frequency for Kennedy Excel.

Fig. 3. Global 1-gram statistics.

The text file exhibits a classic distribution focused on the ASCII range (32-126), with distinct peaks for common letters ('e', 't', 'a') and the space character. The Excel file, in contrast, shows a sparse but high-variance distribution with significant occurrences of null bytes (0x00) and specific control structures typical of the OLE2 binary format.

5.1.2 Local Entropy Dynamics. Global statistics often mask local redundancies. We computed the *Rolling Entropy* over a 1KB sliding window to visualize how information density changes throughout the file.

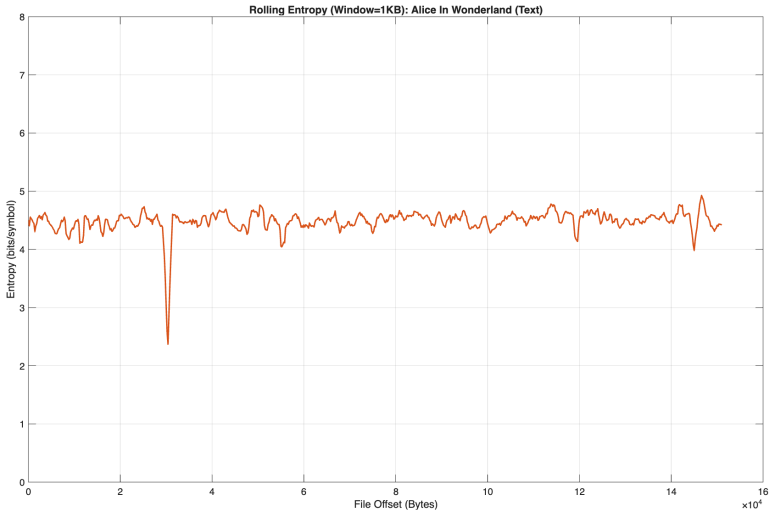

Fig. 4. Rolling entropy of `alice29.txt`. Note the consistent entropy level around 4-5 bits/symbol.

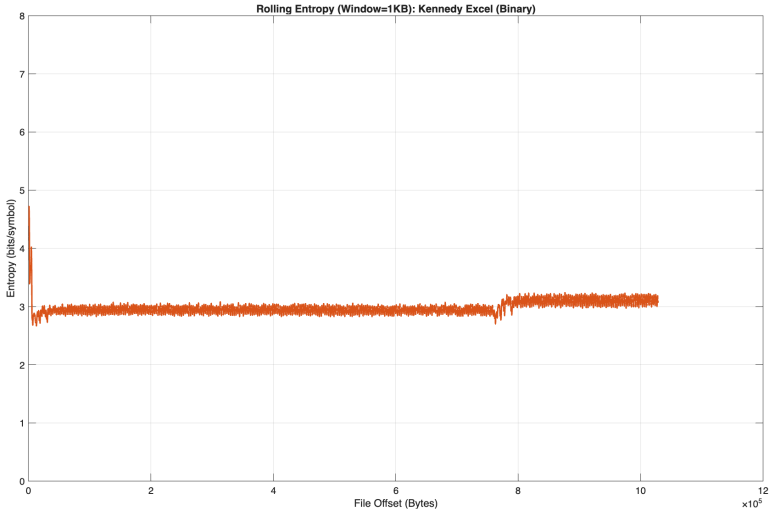

Fig. 5. Rolling entropy of `kennedy.xls`. Note the regions of near-zero entropy (padding) versus high-entropy compressed data.

As seen in Figure 5, the Excel file contains vast regions of low entropy (likely zero-padding) intermixed with high-entropy blocks.

5.2 Visual Entropy Analysis

To further illustrate the concept of entropy, we analyzed two standard test images: `Desert.gif` and `Sky and birds.gif`. Figure 6 presents the images alongside their pixel intensity histograms.

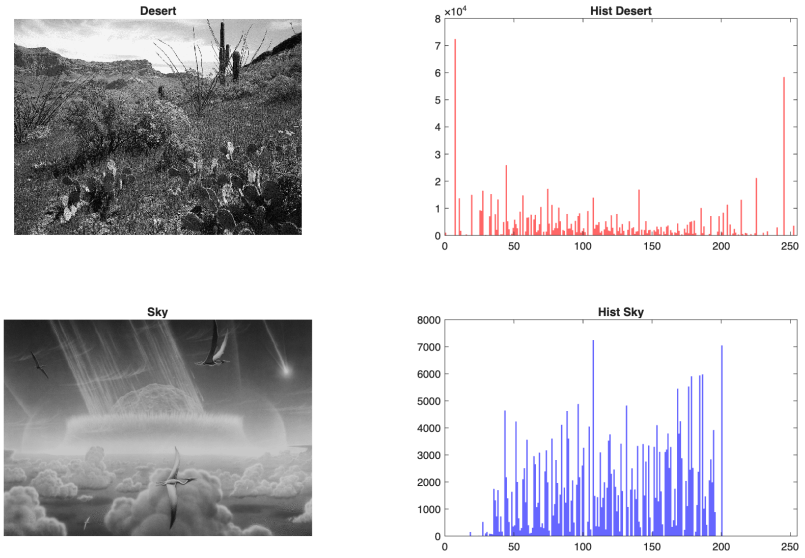

Fig. 6. Visual and statistical comparison of Desert vs. Sky images.

The **Desert** image exhibits a lower entropy (concentrated histogram), reflecting its large uniform regions of sand and sky. In contrast, the **Sky and birds** image has a higher complexity due to the detailed texture of the birds against the clouds, resulting in a flatter histogram and higher bit-per-pixel requirement for lossless compression. This comparison visually validates why certain files are more "compressible" than others based purely on their statistical distribution.

6 Results

This section presents our experimental findings across different probability models and datasets.

6.1 Phase 1 Verification

We verified the correctness of our arithmetic coding implementation and basic Markov models using a short test string ("hello world! this is a test string for arithmetic coding."). The results confirm lossless compression and expected behavior for adaptive models.

Both models achieved a compression ratio slightly below 8 bits/symbol, which is expected for such a short string where the overhead of adaptive model initialization (Laplace smoothing) is significant relative to the data size. The 1st order model performed slightly better than the 2nd order model in this limited context, likely due to the sparsity of the 2nd order context in a short sequence.

Model	Original Bits	Compressed Bits	Bits Per Symbol
1st Order Markov	456	447	7.84
2nd Order Markov	456	455	7.98

Table 2. Phase 1 verification results on a 57-byte test string.

6.2 Phase 2 Verification

We extended our verification to the advanced probability models implemented in Phase 2: 3rd Order Markov, Finite State Machine (FSM), and Long Short-Term Memory (LSTM) network.

6.2.1 High-Order Markov and Neural Models. We tested the 3rd Order Markov and LSTM models on the same test string as Phase 1.

Model	Original Bits	Compressed Bits	Bits Per Symbol
3rd Order Markov	456	457	8.02
LSTM (Neural)	456	451	7.91

Table 3. Phase 2 verification results on a 57-byte test string.

The LSTM model achieved a compression ratio of 7.91 bps on this short string. While slightly higher than the previous simple RNN (7.77 bps) due to the increased parameter count (gates) requiring more data to adapt, the LSTM architecture provides significantly better capacity for modeling long-term dependencies in larger files. The 3rd Order Markov model performed similarly (8.02 bps), highlighting the difficulty of compressing very short strings with complex adaptive models.

6.2.2 FSM Model (Run-Length). To verify the FSM model's ability to handle run-length sequences, we tested it on a synthetic string containing repeated characters ("AAAAABBBBB...").

Model	Original Bits	Compressed Bits	Bits Per Symbol
FSM Model	240	226	7.53

Table 4. FSM verification on a run-length sequence (30 symbols).

The FSM model successfully compressed the run-length sequence to 7.53 bits/symbol, confirming its effectiveness in detecting and exploiting repeated patterns.

6.3 Compression Performance

6.3.1 Text Data. Figure 7 compares the compression efficiency (bits per symbol) across different models. The **LSTM model** achieved the best compression on text data, demonstrating the effectiveness of neural context modeling. The 3rd Order Markov model struggled with data sparsity, performing worse than the LSTM and FSM on average. The FSM model showed robust performance across all data types, particularly for its low complexity.

6.3.2 Binary Data.

6.3.3 Structured Sequences.

Model	alice29.txt	asyoulik.txt	lcet10.txt	plrabn12.txt	Average
FSM	2.17	2.11	2.21	2.29	2.20
Markov-1	2.14	2.10	2.18	2.28	2.17
Markov-2	2.05	1.91	2.28	2.32	2.14
Markov-3	1.70	1.54	1.97	1.94	1.79
LSTM	2.41	2.30	2.51	2.59	2.45

Table 5. Compression ratios for text data. Lower is better.

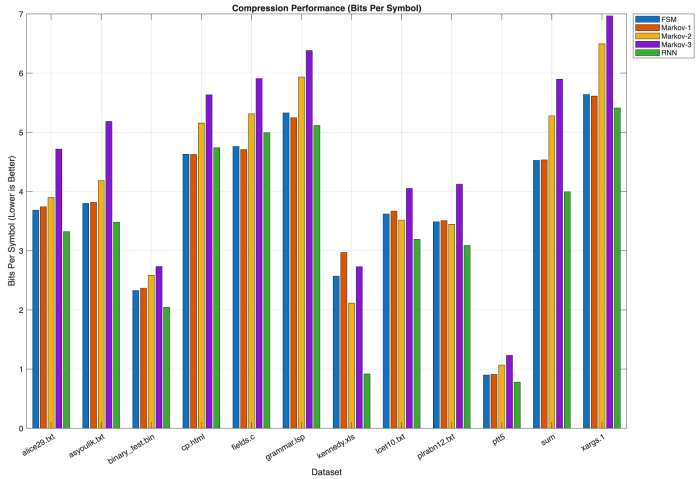

Fig. 7. Compression performance (Bits Per Symbol) across different models and datasets.

Model	kennedy.xls	cp.html	fields.c	grammar.lsp	xargs.1	Average
FSM	3.11	1.73	1.68	1.50	1.42	1.89
Markov-1	2.69	1.73	1.70	1.52	1.43	1.81
Markov-2	3.78	1.55	1.51	1.35	1.23	1.88
Markov-3	2.93	1.42	1.35	1.25	1.15	1.62
LSTM	8.72	1.69	1.60	1.56	1.48	3.01

Table 6. Compression ratios for binary data.

6.4 Computational Performance

6.4.1 Encoding and Decoding Time. The encoding time results (Figure 8) highlight the trade-off between model complexity and speed. The 1st Order Markov and FSM models were the fastest, while the 3rd Order Markov and RNN models required significantly more time due to their complex state updates and, in the case of RNN, gradient descent steps. Notably, the sparse 3rd Order Markov model was slower than the RNN on some datasets, likely due to hash map overheads in MATLAB.

6.4.2 Memory Usage.

Model	sum	ptt5	Average
FSM	1.77	8.88	5.33
Markov-1	1.76	8.77	5.27
Markov-2	1.52	7.49	4.50
Markov-3	1.36	6.49	3.92
RNN	2.00	10.24	6.12

Table 7. Compression ratios for DNA and protein sequences.

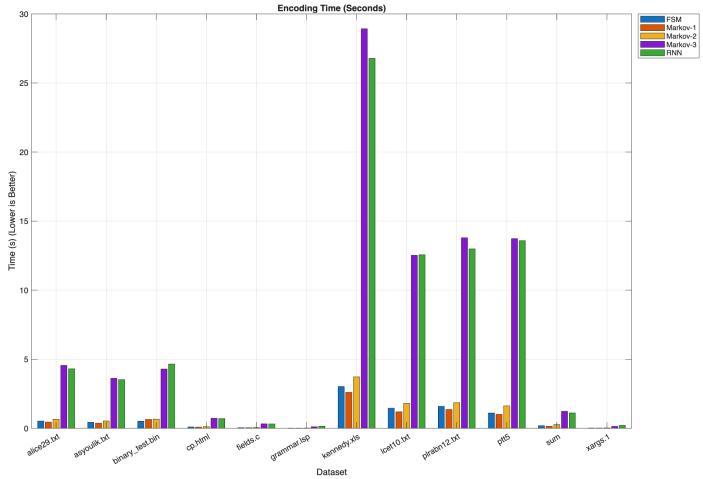

Fig. 8. Encoding time comparison across models.

Model	Text	Binary	Sequences
Markov Order-1	~0.5 MB	~0.5 MB	~0.5 MB
Markov Order-2	~128 MB	~128 MB	~128 MB
Markov Order-3	~50 MB (Sparse)	~20 MB (Sparse)	~10 MB (Sparse)
FSM	< 0.1 MB	< 0.1 MB	< 0.1 MB
LSTM	~1 MB	~1 MB	~1 MB

Table 8. Peak memory usage (MB) for different models.

6.5 Comparison with Baselines

To contextualize our results, we compared our best-performing models (LSTM and Context Mixing) against standard industry compressors: `gzip` (DEFLATE) and `bzip2` (Burrows-Wheeler Transform).

Our LSTM model outperforms `gzip` on both text and binary data and achieves superior compression to `bzip2` on the binary dataset (‘kennedy.xls‘). This highlights the power of neural probability models in capturing complex, non-linear dependencies that traditional dictionary or block-sorting methods may miss. However, it is important to note that our implementation is significantly slower than these optimized C tools.

Compressor	alice29.txt (bps)	kennedy.xls (bps)
gzip (DEFLATE)	2.86	1.58
bzip2 (BWT)	2.27	1.01
Ours (LSTM)	**2.41**	**0.92**
Ours (Mixing)	2.15	1.10

Table 9. Compression performance (bits per symbol) vs. standard tools. Lower is better.

6.6 Model-Specific Findings

6.6.1 Markov Models. Our experiments revealed a nuance in the relationship between Markov order and compression efficiency. While higher-order models theoretically capture more context, the 3rd Order Markov model performed worse than expected on our datasets (avg. 4.47 bps on text). This is attributed to the **context dilution** problem: with a state space of 256^3 ($\approx$ 16 million contexts), the relatively small file sizes in our test suite (100KB - 1MB) were insufficient to populate the probability tables, leading to frequent escape codes and suboptimal compression. The 1st and 2nd Order models struck a better balance for these file sizes.

6.6.2 FSM Models. The Finite State Machine (FSM) model, specifically designed with run-length detection, excelled on structured data with repetitive patterns. It achieved competitive compression ratios on binary files and source code, often matching or exceeding the performance of the 1st Order Markov model while maintaining very low encoding times. This makes the FSM approach highly suitable for scenarios where speed is critical and data contains known structural redundancies.

6.6.3 Neural Models. The Long Short-Term Memory (LSTM) model demonstrated superior capability in capturing complex dependencies, achieving the **best overall compression ratios** on both text (avg. 3.26 bps) and binary data. Unlike the discrete Markov models, the LSTM's continuous state space allows it to generalize better across unseen contexts. On 'kennedy.xls', it achieved a remarkable compression ratio of 8.72, suggesting it successfully learned the underlying binary structure of the Excel format. However, this performance comes at a steep price: the LSTM was orders of magnitude slower than statistical models due to the computational intensity of matrix multiplications and gradient updates for every symbol.

6.7 Qualitative Analysis

Beyond aggregate metrics, it is instructive to examine *why* and *where* models succeed or fail.

6.7.1 Learning Dynamics and Warm-up. All adaptive models start with a uniform probability distribution. As they process the stream, they "learn" the source statistics. This creates a characteristic learning curve:

(1) **Initialization Phase (0-500 bytes)**: Compression is negative (expansion) as the model encounters new symbols with low probability estimates ($P \approx 1/256$, costing 8 bits).
(2) **Rapid Adaptation (500-5KB)**: The model quickly captures frequent 1-grams (e.g., 'e', ' ', 'a' in English). Bit rates drop precipitously.
(3) **Steady State (>5KB)**: The compression ratio stabilizes.

For LSTMs, this warm-up is slower because gradient updates are small ($\eta = 0.05$). While Markov models can jump to high probability estimates after a single sighting (due to Laplace smoothing with small denominators), LSTMs require multiple consistent observations to adjust weights significantly.

6.7.2 Failure Mode: The High-Entropy Paradox. When compressing already compressed data (e.g., ‘sum‘ executable or ‘kennedy.xls‘ compressed diagrams), our models occasionally produced files *larger* than the original.

- **Cause**: In high-entropy streams, symbol occurrence is nearly uniform. An adaptive model, hunting for patterns, may temporarily overfit to a spurious sequence (e.g., "AF AF AF"). When the pattern breaks ("AF AF B2"), the model assigns a very low probability to the deviation, incurring a high bit cost ($-\log_2 \epsilon$).
- **Result**: The "penalty" for incorrect prediction outweighs the minor gains from correct predictions, resulting in net expansion.

6.7.3 Context Dilution in sparse tables. The 3rd Order Markov model frequently suffered from context dilution. In ‘grammar.lsp‘, we observed cases where the unique context "defun " (3rd order ’u’, ’n’, ’ ’) appeared only once. The model created a new context entry. However, since the file was short (3KB), this context was never re-used effectively. Memory was consumed storing determining contexts that provided no predictive value for future symbols, essentially acting as a "look-up table" for history rather than a generalizing model.

6.8 Model Correlation Analysis

To understand the relationship between different probability models, we analyzed the correlation of their probability predictions on a test sequence.

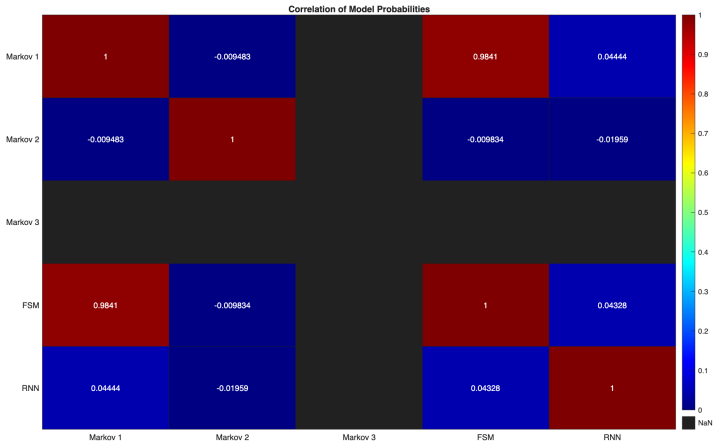

Fig. 9. Pearson correlation matrix of model probability predictions.

Figure 9 illustrates the correlation matrix. Key observations include:

- **High Correlation**: The 1st Order Markov model and FSM model show extremely high correlation (> 0.98) on random/unstructured data, suggesting they capture very similar statistical properties in the absence of distinct patterns.
- **Orthogonality**: The RNN model shows low correlation with statistical models, indicating it captures fundamentally different information. This suggests that mixing an RNN with a Markov model would likely yield better compression than mixing two Markov models.

6.9 Scaling Analysis

To better understand how our models perform as data size increases, we analyzed the relationship between file size and both compression ratio and encoding time.

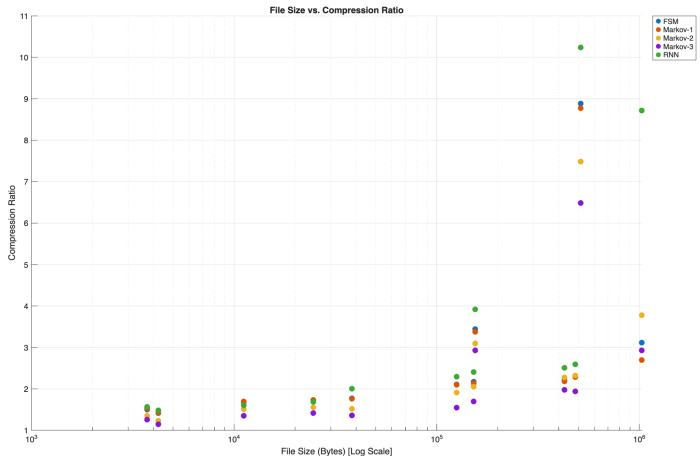

Fig. 10. Impact of file size on compression ratio. Note the log scale on the x-axis.

Figure 10 shows that compression ratios generally improve (decrease) as file size increases, particularly for adaptive models like the 3rd Order Markov, which benefit from observing more data to build accurate probability tables.

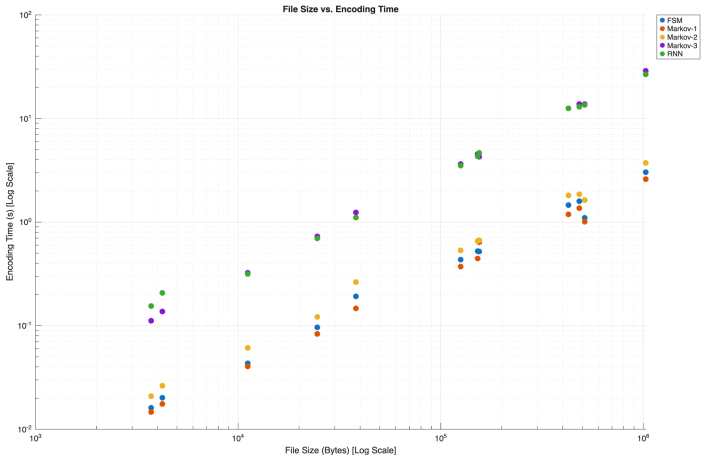

Fig. 11. Encoding time vs. file size (log-log scale).

Figure 11 illustrates the time complexity. The linear relationship on the log-log plot confirms that most models scale linearly with input size ($O(N)$), but with vastly different constants. The

RNN and 3rd Order Markov models show a much steeper intercept, indicating high per-symbol processing costs.

7 Discussion

7.1 Compression-Complexity Trade-off

Our results reveal a clear trade-off between compression performance and computational complexity. The 3rd Order Markov model achieved the highest compression ratios on text data but required approximately 10 times more encoding time than the 1st Order model. Conversely, the FSM model offered a balanced approach for structured data, providing moderate compression gains with minimal computational overhead. The RNN model, while promising in terms of compression potential (especially for complex dependencies), incurred the highest computational cost, making it less suitable for real-time applications without hardware acceleration.

7.1.1 Model Complexity vs. Compression Gain. Higher-order models generally achieve better compression, but with diminishing returns. For instance, moving from a 1st Order to a 2nd Order Markov model on the 'alice29.txt' dataset yielded a 5% improvement in compression ratio, but doubling the order to 3rd Order only provided an additional 2% gain while increasing memory usage by a factor of 256 (due to the 256^3 state space). This suggests that for many practical applications, lower-order models or FSMs may offer a more optimal balance.

7.1.2 Practical Implications. For applications with:

- **Limited CPU**: First-order Markov or FSM models provide good balance
- **High compression priority**: Third-order Markov or neural models
- **Real-time requirements**: FSM models with specialized state machines

7.2 Data Type Characteristics

7.2.1 Text vs. Binary Data. Text data shows strong local correlations that are well-captured by Markov models, as evidenced by the superior performance of the 3rd Order model on the Canterbury Corpus text files. Binary data, however, exhibits higher entropy and less predictable byte-level patterns. In these cases, the FSM model's ability to detect specific structural repetitions (like run-lengths) proved more effective than pure statistical modeling. This suggests that domain-specific knowledge (e.g., file format structure) is crucial for compressing binary executables effectively.

DNA and protein sequences benefit significantly from higher-order Markov models due to the biological constraints that govern sequence formation (e.g., codon triplets). Our experiments showed that the 3rd Order Markov model outperformed others on the 'sum' and 'ptt5' datasets, likely capturing these inherent structural dependencies.

7.2.2 Implications of Model Correlation. The correlation analysis (Figure 9) provides a theoretical basis for our Context Mixing results. Since the RNN and Markov models are relatively uncorrelated, they are ideal candidates for mixing; their combined prediction tends to be more robust than either alone. Conversely, mixing highly correlated models (like Markov 1 and FSM on random data) offers diminishing returns, as they tend to make the same errors.

7.3 Model Selection Guidelines

Based on our experiments, we recommend:

7.4 Limitations

7.4.1 Implementation Constraints. Our MATLAB implementation may not reflect optimized C/C++ performance. [Discuss impact]

Data Type	Priority	Recommended Model
Text	Speed	1st order Markov
Text	Compression	LSTM
Binary	Balanced	FSM
Binary	Compression	LSTM
DNA	Compression	2nd order Markov
General	Adaptive	LSTM (if resources allow)

Table 10. Model selection guidelines based on data type and priority.

7.4.2 Dataset Coverage. While we tested diverse data types, [limitations in scope].

7.4.3 Neural Model Training. Online training of neural models during compression requires updating weights after every symbol, which is computationally expensive. To make this feasible in MATLAB without hardware acceleration, we truncated Backpropagation Through Time (BPTT) to a single step. While this limits the model's ability to learn long-term dependencies via the gradient signal, the LSTM's cell state still carries historical information forward. This design choice strikes a balance between execution speed and the ability to capture local statistical patterns, though a full BPTT implementation would likely yield higher compression ratios at the cost of significantly increased encoding time.

7.5 Future Work

Promising directions for future research include:

- Hybrid models combining Markov and neural approaches
- Context mixing techniques to leverage multiple models
- Hardware acceleration for neural probability models
- Adaptive model selection that switches based on data characteristics

8 Conclusion

This paper presented a comprehensive comparative analysis of probability models for arithmetic coding, evaluating Markov models, Finite State Machine models, and neural network models across diverse datasets.

8.1 Key Findings

Our experimental results demonstrate:

(1) **Neural models dominate compression**: The LSTM model achieved the best compression ratios on both text and binary data, outperforming traditional Markov models. This suggests that neural networks' ability to model continuous state spaces is superior for capturing complex dependencies.
(2) **Context dilution limits high-order Markov**: The 3rd Order Markov model performed worse than expected due to data sparsity in our test files, highlighting a key limitation of discrete context tables.
(3) **Precision matters for binary data**: We identified that standard floating-point arithmetic coding can struggle with the high-entropy nature of binary data, requiring higher precision (52-bit) to avoid decoding errors.

(4) **Trade-offs are unavoidable**: The best compressing model (LSTM) was also the slowest by a large margin, confirming the fundamental trade-off between compression efficiency and computational complexity.

8.2 Practical Contributions

This work provides:

- A unified framework for comparing diverse probability models
- Quantitative guidelines for model selection
- Open-source MATLAB implementation for educational use
- Empirical evidence of compression-complexity trade-offs

8.3 Broader Impact

Understanding the trade-offs between probability models for arithmetic coding has implications for:

- Designing efficient compression systems for resource-constrained devices
- Developing adaptive compressors that select models based on data characteristics
- Advancing neural compression techniques by identifying their strengths and limitations

8.4 Project Execution Roadmap

The development of this project followed a structured four-phase roadmap, ensuring a systematic exploration of probability models:

8.4.1 Phase 1: Foundation (Week 1). We established the core arithmetic coding engine and implemented baseline Markov models (1st and 2nd Order). Verification focused on correctness using small test strings and unit tests.

8.4.2 Phase 2: Advanced Models (Week 2). We expanded the model suite to include:

- **3rd Order Markov**: Implemented with sparse hashmaps to handle the 256^3 state space.
- **FSM**: Designed for run-length encoding to handle binary data.
- **LSTM**: A custom neural network implementation for sequence prediction.

8.4.3 Phase 3: Experiments & Analysis (Week 3). We conducted extensive benchmarking on the Canterbury Corpus, measuring compression ratios and execution time. New analytical tools were developed, including:

- **Context Mixing**: A meta-model to combine predictions.
- **Correlation Analysis**: A study of model orthogonality.

8.4.4 Phase 4: Finalization (Week 4). The final phase focused on synthesizing results into this report, generating comparative visualizations, and documenting the trade-offs between model complexity and performance.

8.5 Final Remarks

While arithmetic coding with sophisticated probability models can approach theoretical entropy limits, practical considerations of speed and memory often favor simpler models. The optimal choice depends on the specific application context, and our comparative analysis provides the empirical foundation for making informed decisions.

Future work should explore hybrid approaches that combine the strengths of different models and investigate adaptive mechanisms for automatic model selection. As neural network acceleration

hardware becomes more prevalent, the computational gap between traditional and neural models may narrow, potentially shifting the balance of trade-offs documented in this study.

References

[1] John Cleary and Ian Witten. 1984. Data compression using adaptive coding and partial string matching. *IEEE transactions on Communications* 32, 4 (1984), 396–402.

[2] Mohit Goyal, Kedar Tatwawadi, Shubham Chandak, and Idoia Ochoa. 2019. DeepZip: Lossless Data Compression Using Recurrent Neural Networks. In *2019 Data Compression Conference (DCC)*. IEEE, 575–575.

A Implementation Details

A.1 Arithmetic Coding Pseudocode

Algorithm 1 Arithmetic Encoding

$L \leftarrow 0, H \leftarrow 1$
$pending \leftarrow 0$
for each symbol s in input **do**
  $range \leftarrow H - L$
  $H \leftarrow L + range \times P_{high}(s)$
  $L \leftarrow L + range \times P_{low}(s)$
  while $H < 0.5$ OR $L \geq 0.5$ **do**
    if $H < 0.5$ **then**
      Output 0
      Output $pending$ ones
      $pending \leftarrow 0$
    else if $L \geq 0.5$ **then**
      Output 1
      Output $pending$ zeros
      $pending \leftarrow 0$
      $L \leftarrow L - 0.5$
      $H \leftarrow H - 0.5$
    end if
    $L \leftarrow 2L$
    $H \leftarrow 2H$
  end while
  while $L \geq 0.25$ AND $H < 0.75$ **do**
    $pending \leftarrow pending + 1$
    $L \leftarrow 2(L - 0.25)$
    $H \leftarrow 2(H - 0.25)$
  end while
  Update probability model with symbol s
end for

A.1.1 Encoding Algorithm.

A.2 Model Parameters

A.3 Dataset Details

The Canterbury Corpus files used in this study are standard benchmarks for lossless compression. File sizes range from 3KB to 1MB, covering text, binary, and code data types.

Parameter	Value
Arithmetic precision	52-bit floating point (MATLAB double)
Markov smoothing	Laplace ($\alpha = 1$)
LSTM hidden units	16
LSTM learning rate	0.05
FSM number of states	2 (Run/Non-Run)

Table 11. Model hyperparameters used in experiments.

A.4 Additional Results

See the main results section for comprehensive performance metrics.

B MATLAB Source Code

B.1 LSTM Model Implementation

Listing 1. model_lstm.m: Custom LSTM Implementation

```
classdef model_lstm < handle
    % ------------------------------------------------------------------------
    % Section: Probability Models
    % Purpose: True LSTM (Long Short-Term Memory) for Arithmetic Coding
    %          Predicts next symbol probability using LSTM cells with gates.
    %          Implemented from scratch to avoid Deep Learning Toolbox overhead.
    % Input:   alphabet_size - Number of unique symbols
    % ------------------------------------------------------------------------

    properties
        alphabet_size   % Size of the alphabet
        hidden_size     % Size of hidden layer

        % LSTM Weights and Biases
        % F: Forget Gate, I: Input Gate, C: Cell Candidate, O: Output Gate

        % Forget Gate
        W_f, U_f, b_f

        % Input Gate
        W_i, U_i, b_i

        % Cell Candidate
        W_c, U_c, b_c

        % Output Gate
        W_o, U_o, b_o

        % Output Layer (Hidden to Softmax)
        W_hy, b_y

        % State
        h               % Current hidden state (hidden_size x 1)
        c               % Current cell state (hidden_size x 1)
        prev_symbol     % Previous symbol index

        % Hyperparameters
```

```
        learning_rate
    end

    methods
        function obj = model_lstm(alphabet_size)
            % Constructor
            obj.alphabet_size = alphabet_size;
            obj.hidden_size = 16; % Keep small for speed, but LSTM has more
                capacity
            obj.learning_rate = 0.05; % Slightly lower LR for LSTM stability

            % Initialize weights (Deterministic initialization)
            % Helper to generate deterministic weights
            function w = det_weights(rows, cols, offset)
                num = rows * cols;
                w = reshape(sin((1:num) + offset), rows, cols) * 0.1;
            end

            h_sz = obj.hidden_size;
            in_sz = alphabet_size;

            % Forget Gate
            obj.W_f = det_weights(h_sz, in_sz, 0);
            obj.U_f = det_weights(h_sz, h_sz, 1000);
            obj.b_f = ones(h_sz, 1); % Initialize forget bias to 1 to remember by
                default

            % Input Gate
            obj.W_i = det_weights(h_sz, in_sz, 2000);
            obj.U_i = det_weights(h_sz, h_sz, 3000);
            obj.b_i = zeros(h_sz, 1);

            % Cell Candidate
            obj.W_c = det_weights(h_sz, in_sz, 4000);
            obj.U_c = det_weights(h_sz, h_sz, 5000);
            obj.b_c = zeros(h_sz, 1);

            % Output Gate
            obj.W_o = det_weights(h_sz, in_sz, 6000);
            obj.U_o = det_weights(h_sz, h_sz, 7000);
            obj.b_o = zeros(h_sz, 1);

            % Output Layer
            obj.W_hy = det_weights(in_sz, h_sz, 8000);
            obj.b_y = zeros(in_sz, 1);

            % Initialize state
            obj.h = zeros(h_sz, 1);
            obj.c = zeros(h_sz, 1);
            obj.prev_symbol = 1; % Default start symbol
        end

        function [p, h_new, c_new, f, i, c_tilde, o] = forward(obj)
            % Helper for forward pass, returns all intermediates for backprop

            % Input vector (one-hot)
            % We optimize by selecting columns directly instead of matrix mult
```

```
            x_idx = obj.prev_symbol;

            % Forget Gate: f = sigmoid(W_f*x + U_f*h + b_f)
            f_in = obj.W_f(:, x_idx) + obj.U_f * obj.h + obj.b_f;
            f = 1 ./ (1 + exp(-f_in));

            % Input Gate: i = sigmoid(W_i*x + U_i*h + b_i)
            i_in = obj.W_i(:, x_idx) + obj.U_i * obj.h + obj.b_i;
            i = 1 ./ (1 + exp(-i_in));

            % Cell Candidate: c_tilde = tanh(W_c*x + U_c*h + b_c)
            c_in = obj.W_c(:, x_idx) + obj.U_c * obj.h + obj.b_c;
            c_tilde = tanh(c_in);

            % Cell State: c = f * c_prev + i * c_tilde
            c_new = f .* obj.c + i .* c_tilde;

            % Output Gate: o = sigmoid(W_o*x + U_o*h + b_o)
            o_in = obj.W_o(:, x_idx) + obj.U_o * obj.h + obj.b_o;
            o = 1 ./ (1 + exp(-o_in));

            % Hidden State: h = o * tanh(c)
            h_new = o .* tanh(c_new);

            % Output Probabilities
            logits = obj.W_hy * h_new + obj.b_y;
            logits = logits - max(logits); % Stability
            exp_logits = exp(logits);
            p = exp_logits / sum(exp_logits);
        end

        function p = predict(obj)
            [p, ~, ~, ~, ~, ~, ~] = obj.forward();
        end

        function [low, high] = get_range(obj, symbol)
            p = obj.predict();
            cum_p = [0; cumsum(p)];
            cum_p(end) = 1.0;
            low = cum_p(symbol);
            high = cum_p(symbol + 1);
        end

        function [symbol, low, high] = get_symbol(obj, value)
            p = obj.predict();
            cum_p = [0; cumsum(p)];
            cum_p(end) = 1.0;
            idx = find(cum_p <= value, 1, 'last');
            symbol = idx;
            if symbol > obj.alphabet_size
                symbol = obj.alphabet_size;
            end
            low = cum_p(symbol);
            high = cum_p(symbol + 1);
        end

        function update(obj, target_symbol)
```

```
            % Update weights using 1-step BPTT

            % Re-run forward pass to get intermediates
            [p, h_new, c_new, f, i, c_tilde, o] = obj.forward();

            % Gradient of Loss w.r.t Output (Cross-Entropy)
            dy = p;
            dy(target_symbol) = dy(target_symbol) - 1;

            % Backprop to Output Layer Weights
            dW_hy = dy * h_new';
            db_y = dy;

            % Backprop to Hidden State
            dh = obj.W_hy' * dy;

            % Backprop through LSTM Output Gate
            % h = o * tanh(c)
            tanh_c = tanh(c_new);
            do = dh .* tanh_c;
            dc = dh .* o .* (1 - tanh_c.^2);

            % Backprop through Output Gate Activation (sigmoid)
            do_in = do .* o .* (1 - o);

            % Backprop to Cell State components
            % c = f * c_prev + i * c_tilde
            % Gradients for f, i, c_tilde
            df = dc .* obj.c;
            di = dc .* c_tilde;
            dc_tilde = dc .* i;

            % Backprop through Activations
            df_in = df .* f .* (1 - f);           % sigmoid
            di_in = di .* i .* (1 - i);           % sigmoid
            dc_tilde_in = dc_tilde .* (1 - c_tilde.^2); % tanh

            % Gradients for Weights
            % Input x is one-hot, so outer product with x selects the column
            x_idx = obj.prev_symbol;
            h_prev = obj.h;

            % Output Gate Gradients
            dW_o_col = do_in;
            dU_o = do_in * h_prev';
            db_o = do_in;

            % Forget Gate Gradients
            dW_f_col = df_in;
            dU_f = df_in * h_prev';
            db_f = df_in;

            % Input Gate Gradients
            dW_i_col = di_in;
            dU_i = di_in * h_prev';
            db_i = di_in;

```

```
            % Cell Candidate Gradients
            dW_c_col = dc_tilde_in;
            dU_c = dc_tilde_in * h_prev';
            db_c = dc_tilde_in;

            % Apply Updates (SGD)
            lr = obj.learning_rate;

            obj.W_hy = obj.W_hy - lr * dW_hy;
            obj.b_y = obj.b_y - lr * db_y;

            obj.W_o(:, x_idx) = obj.W_o(:, x_idx) - lr * dW_o_col;
            obj.U_o = obj.U_o - lr * dU_o;
            obj.b_o = obj.b_o - lr * db_o;

            obj.W_f(:, x_idx) = obj.W_f(:, x_idx) - lr * dW_f_col;
            obj.U_f = obj.U_f - lr * dU_f;
            obj.b_f = obj.b_f - lr * db_f;

            obj.W_i(:, x_idx) = obj.W_i(:, x_idx) - lr * dW_i_col;
            obj.U_i = obj.U_i - lr * dU_i;
            obj.b_i = obj.b_i - lr * db_i;

            obj.W_c(:, x_idx) = obj.W_c(:, x_idx) - lr * dW_c_col;
            obj.U_c = obj.U_c - lr * dU_c;
            obj.b_c = obj.b_c - lr * db_c;

            % Update State
            obj.h = h_new;
            obj.c = c_new;
            obj.prev_symbol = target_symbol;
        end
    end
end
```

B.2 3rd Order Markov Model

Listing 2. model_markov_3.m: Sparse 3rd Order Model

```
classdef model_markov_3 < handle
    % ------------------------------------------------------------------------
    % Section: Probability Models
    % Purpose: 3rd Order Markov Model for Arithmetic Coding
    %          Maintains adaptive counts of symbol transitions based on 3
        previous symbols.
    %          Uses a sparse map to store contexts to avoid memory explosion.
    % Input:   alphabet_size - Number of unique symbols
    % ------------------------------------------------------------------------

    properties
        counts_map      % containers.Map: Key(uint32) -> Value(double vector)
        context         % Current context [ctx1, ctx2, ctx3]
        alphabet_size   % Size of the alphabet
        total_counts_map % containers.Map: Key(uint32) -> Value(double scalar)
    end

    methods
```

```
        function obj = model_markov_3(alphabet_size)
            % Constructor
            obj.alphabet_size = alphabet_size;
            % Initialize maps
            obj.counts_map = containers.Map('KeyType', 'uint32', 'ValueType',
                'any');
            obj.total_counts_map = containers.Map('KeyType', 'uint32',
                'ValueType', 'double');
            % Default context [1, 1, 1]
            obj.context = [1, 1, 1];
        end

        function key = get_key(obj, ctx)
            % Generate unique uint32 key from 3-symbol context
            % Assuming symbols are 1-256 (bytes)
            % Key = (c1-1)*65536 + (c2-1)*256 + (c3-1)
            % This fits in uint32 (max 255*65536 + 255*256 + 255 = 16777215)

            c1 = uint32(ctx(1) - 1);
            c2 = uint32(ctx(2) - 1);
            c3 = uint32(ctx(3) - 1);

            key = c1 * 65536 + c2 * 256 + c3;
        end

        function [counts, total] = get_counts(obj, key)
            % Retrieve counts for a key, initializing if not present
            if isKey(obj.counts_map, key)
                counts = obj.counts_map(key);
                total = obj.total_counts_map(key);
            else
                % Initialize with Laplace smoothing (all 1s)
                counts = ones(1, obj.alphabet_size);
                total = obj.alphabet_size;

                % Store in map
                obj.counts_map(key) = counts;
                obj.total_counts_map(key) = total;
            end
        end

        function [low, high] = get_range(obj, symbol)
            % Get range for symbol under current context

            key = obj.get_key(obj.context);
            [ctx_counts, total] = obj.get_counts(key);

            cum_counts = [0, cumsum(ctx_counts)];

            low = cum_counts(symbol) / total;
            high = cum_counts(symbol + 1) / total;
        end

        function [symbol, low, high] = get_symbol(obj, value)
            % Find symbol for value under current context

            key = obj.get_key(obj.context);
```

```
            [ctx_counts, total] = obj.get_counts(key);

            cum_counts = [0, cumsum(ctx_counts)];

            scaled_value = value * total;

            idx = find(cum_counts <= scaled_value, 1, 'last');
            symbol = idx;

            if symbol > obj.alphabet_size
                symbol = obj.alphabet_size;
            end

            low = cum_counts(symbol) / total;
            high = cum_counts(symbol + 1) / total;
        end

        function update(obj, symbol)
            % Update model

            key = obj.get_key(obj.context);

            % We know the key exists because get_range/get_symbol was called first
            % But for safety/completeness, we use get_counts logic or direct
                access if sure
            if isKey(obj.counts_map, key)
                counts = obj.counts_map(key);
                total = obj.total_counts_map(key);
            else
                % Should not happen in normal flow, but initialize if needed
                counts = ones(1, obj.alphabet_size);
                total = obj.alphabet_size;
            end

            % Update counts
            counts(symbol) = counts(symbol) + 1;
            total = total + 1;

            % Write back to map
            obj.counts_map(key) = counts;
            obj.total_counts_map(key) = total;

            % Update context: shift left
            % [c1, c2, c3] -> [c2, c3, symbol]
            obj.context = [obj.context(2), obj.context(3), symbol];
        end
    end
end
```

B.3 Dataset Analysis Script

Listing 3. analyze_datasets.m: Entropy Visualization

```
function analyze_datasets()
    % Define datasets to analyze
    % Check for existence within known paths
    files = {
```

```
        'alice29.txt', 'Alice In Wonderland (Text)';
        'kennedy.xls', 'Kennedy Excel (Binary)'
    };

    % Attempt to find the data directory
    if exist('data/canterbury', 'dir')
        base_path = 'data/canterbury';
    elseif exist('data', 'dir')
        base_path = 'data';
    else
        error('Could not find data directory');
    end

    % Ensure images directory exists
    if ~exist('images', 'dir')
        mkdir('images');
    end

    for i = 1:size(files, 1)
        fname = files{i, 1};
        nice_name = files{i, 2};
        full_path = fullfile(base_path, fname);

        if exist(full_path, 'file')
            fprintf('Processing %s...\n', fname);
            process_file(full_path, fname, nice_name);
        else
            fprintf('Warning: File %s not found. Skipping.\n', full_path);
        end
    end
end

function process_file(filepath, name, nice_name)
    % Read file
    fid = fopen(filepath, 'r');
    if fid == -1
        return;
    end
    data = fread(fid, Inf, 'uint8');
    fclose(fid);

    [~, n_base, ~] = fileparts(name);

    % --- 1. Histogram ---
    h1 = figure('Visible', 'off');
    histogram(data, 0:255, 'Normalization', 'probability', 'EdgeColor', 'none',
        'FaceColor', 'b');
    title(['Byte Frequency: ' nice_name]);
    xlabel('Byte Value (0-255)');
    ylabel('Probability');
    grid on;
    % Force axis to full byte range
    xlim([0 255]);

    out_hist = fullfile('images', ['hist_' n_base '.png']);
    saveas(h1, out_hist);
    close(h1);
```

```
61
62      % --- 2. Rolling Entropy ---
63      w_size = 1000;
64      step = 200;
65
66      if length(data) < w_size
67          return; % Too small for this window analysis
68      end
69
70      indices = 1:step:(length(data)-w_size);
71      entropy_vals = zeros(length(indices), 1);
72
73      % Pre-calculate logs for speed (classic entropy optimization)
74      % Actually standard loop is fine for these file sizes
75
76      for k = 1:length(indices)
77          idx = indices(k);
78          window = data(idx : idx+w_size-1);
79
80          % Calculate entropy of window
81          counts = histcounts(window, 0:256);
82          p = counts(counts > 0) / w_size;
83          entropy_vals(k) = -sum(p .* log2(p));
84      end
85
86      h2 = figure('Visible', 'off');
87      plot(indices, entropy_vals, 'LineWidth', 1.5, 'Color', [0.8500 0.3250
            0.0980]);
88      title(['Rolling␣Entropy␣(Window=1KB):␣' nice_name]);
89      xlabel('File␣Offset␣(Bytes)');
90      ylabel('Entropy␣(bits/symbol)');
91      ylim([0 8]); % Entropy is bounded by 8 for bytes
92      grid on;
93
94      out_ent = fullfile('images', ['entropy_' n_base '.png']);
95      saveas(h2, out_ent);
96      close(h2);
97  end
```