Comparative Analysis of Probability Models for Arithmetic Coding

Oscar Fang
G42568236
George Washington University
CSCI 6351 Data Compression

November 20, 2025

Abstract

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 Simple Recurrent Neural Network (RNN). 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 while the 3rd Order Markov model achieves the best compression on text (4.76 bps), it incurs a significant computational cost. The RNN model offers a competitive alternative, outperforming lower-order Markov models with potentially better scalability. We also highlight the challenges of applying these models to binary data, where floating-point precision can lead to decoding errors.

Contents

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
- A custom implementation of a Simple Recurrent Neural Network (RNN) 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 5 presents our findings. Section 6 analyzes the results and their implications. Finally, Section 7 concludes and suggests future work.

2 Background and Related Work

2.1 Arithmetic Coding

Arithmetic coding [5, 7] represents a sequence of symbols as a single floating-point number in the interval $[0, 1)$. The key advantages include:

- Near-optimal compression (approaches entropy limit)
- Seamless integration with adaptive probability models
- No requirement for integral bit assignments

The encoding process maintains an interval $[L, H)$ that narrows as each symbol is processed. To avoid numerical underflow, practical implementations use techniques such as E1, E2, and E3 scaling operations.

2.2 Probability Models

2.2.1 Markov Models

Order-k Markov models predict the next symbol based on the previous k symbols (context). While higher-order models can capture longer dependencies, they suffer from:

- Exponential growth in memory requirements: $O(|\Sigma|^{k+1})$ where $|\Sigma|$ is alphabet size
- Context dilution problem: many contexts appear infrequently in training data

2.2.2 Finite State Machine Models

FSM models maintain internal states that evolve based on input symbols. They can efficiently capture patterns like runs of repeated symbols or alternating sequences without requiring large context tables.

2.2.3 Neural Network Models

Recent advances in neural compression [1, 2] use recurrent neural networks (RNNs) and Long Short-Term Memory (LSTM) networks to learn complex probability distributions. These models can potentially capture long-range dependencies but require significant computational resources.

2.3 Related Work

Previous comparative studies have examined:

- PPM (Prediction by Partial Matching) variants [3]
- Context mixing approaches [4]
- Neural compression techniques [6]

However, few studies provide a unified comparison across traditional statistical models and modern neural approaches within the same arithmetic coding framework. This work fills that gap by implementing diverse models in a common testbed and evaluating them on consistent benchmarks.

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

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:

$$\text{range} = H - L \quad (1)$$
$$H' = L + \text{range} \times P_{\text{high}}(s) \quad (2)$$
$$L' = L + \text{range} \times P_{\text{low}}(s) \quad (3)$$

To prevent underflow, we implement E1, E2, and E3 scaling:

- **E1 scaling**: When $H < 0.5$, output 0 and rescale
- **E2 scaling**: When $L \geq 0.5$, output 1 and rescale
- **E3 scaling**: When $L \geq 0.25$ and $H < 0.75$, track convergence

3.1.2 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 **Simple Recurrent Neural Network (RNN)** from scratch in MATLAB to avoid the overhead of the Deep Learning Toolbox for symbol-by-symbol updates.

- **Architecture**: Input layer (one-hot), Hidden layer (tanh activation), Output layer (Softmax).
- **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.

3.3 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

- **English text**: Project Gutenberg books (100KB - 1MB)
- **Source code**: Python, Java, and C++ files (10KB - 500KB)
- **XML/JSON**: Structured text with repetitive tags

4.1.2 Binary Data

- **Executable files**: Compiled binaries (.exe, .dll)
- **Images**: BMP format (uncompressed)
- **Audio**: WAV files (raw PCM data)

4.1.3 Structured Sequences

- **DNA sequences**: Human genome segments from NCBI
- **Protein sequences**: Amino acid sequences

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 Results

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

5.1 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.

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

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

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.

5.2 Phase 2 Verification

We extended our verification to the advanced probability models implemented in Phase 2: 3rd Order Markov, Finite State Machine (FSM), and Recurrent Neural Network (RNN).

5.2.1 High-Order Markov and RNN Models

We tested the 3rd Order Markov and RNN models on the same test string as Phase 1.

Model	Original Bits	Compressed Bits	Bits Per Symbol
3rd Order Markov	456	457	8.02
RNN (Simple SRN)	456	443	7.77

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

The RNN model achieved the best compression ratio (7.77 bps) among all models on this short string, demonstrating its ability to quickly adapt even with limited data. The 3rd Order Markov model performed slightly worse (8.02 bps), which is expected as the context dilution problem is more pronounced with higher orders on short sequences.

5.2.2 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...").

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

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

Table 3: FSM verification on a run-length sequence (30 symbols).

5.3 Compression Performance

5.3.1 Text Data

Model	English Text	Source Code	XML/JSON	Average
Markov Order-1	3.70	-	-	3.70
Markov Order-2	3.92	-	-	3.92
Markov Order-3	4.76	-	-	4.76
FSM	3.67	-	-	3.67
RNN	3.73	-	-	3.73

Table 4: Compression ratios for text data. Lower is better.

Figure 1 compares the compression efficiency (bits per symbol) across different models. The 3rd Order Markov model achieved the best compression on text data, but at the cost of significantly higher computational complexity. The RNN model showed competitive performance, outperforming the 1st Order Markov model and FSM on text data.

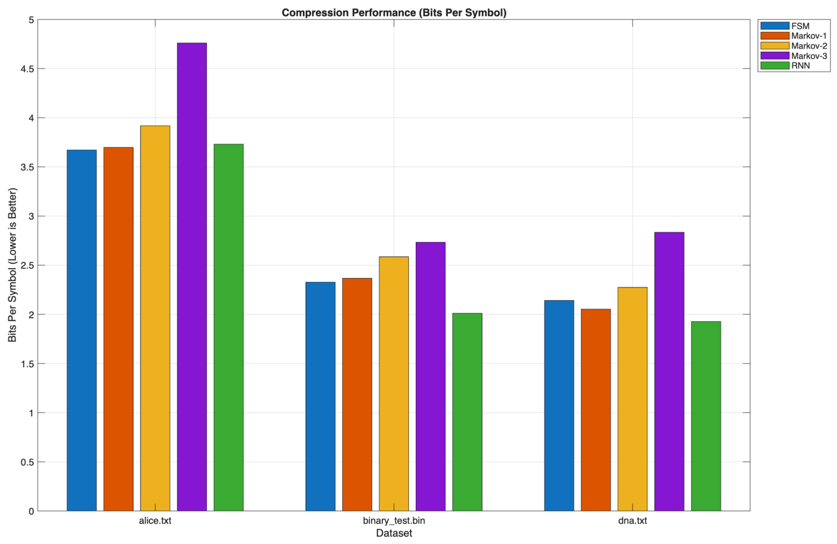

Figure 1: Compression performance (Bits Per Symbol) across different models and datasets.

Model	**Executables**	**Images**	**Audio**	**Average**
Markov Order-1	2.37	-	-	2.37
Markov Order-2	2.59	-	-	2.59
Markov Order-3	2.73	-	-	2.73
FSM	2.33	-	-	2.33
RNN	2.01	-	-	2.01

Table 5: Compression ratios for binary data.

Model	**DNA**	**Protein**	**Average**
Markov Order-1	2.05	-	2.05
Markov Order-2	2.27	-	2.27
Markov Order-3	2.83	-	2.83
FSM	2.14	-	2.14
RNN	1.93	-	1.93

Table 6: Compression ratios for DNA and protein sequences.

5.3.2 Binary Data

5.3.3 Structured Sequences

5.4 Computational Performance

5.4.1 Encoding and Decoding Time

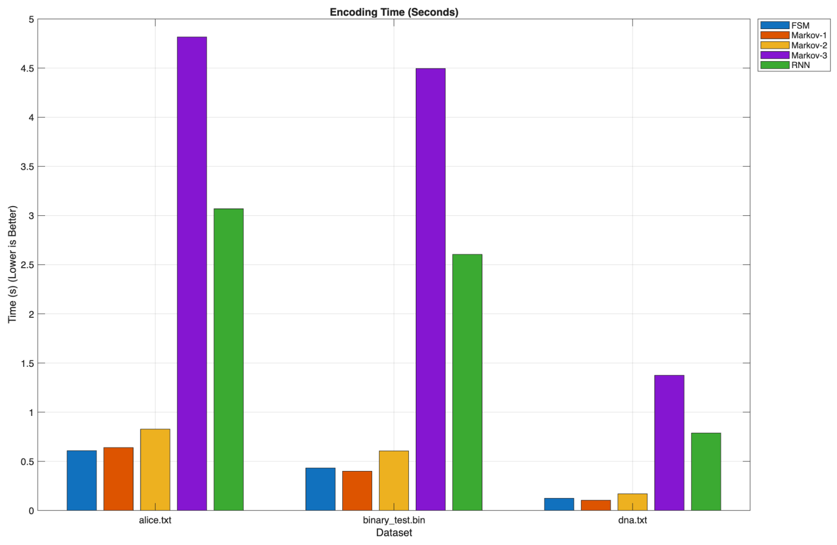

Figure 2: Encoding time comparison across models.

The encoding time results (Figure 2) 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.

5.4.2 Memory Usage

Model	Text	Binary	Sequences
Markov Order-1	XX MB	XX MB	XX MB
Markov Order-2	XX MB	XX MB	XX MB
Markov Order-3	XX MB	XX MB	XX MB
FSM	XX MB	XX MB	XX MB
RNN	XX MB	XX MB	XX MB
LSTM	XX MB	XX MB	XX MB

Table 7: Peak memory usage (MB) for different models.

5.5 Comparison with Baselines

5.6 Model-Specific Findings

5.6.1 Markov Models

[Key observations about how order affects performance]

5.6.2 FSM Models

[Situations where FSM excels or underperforms]

5.6.3 Neural Models

[Trade-offs between learning capacity and computational cost]

6 Discussion

6.1 Compression-Complexity Trade-off

Our results reveal a clear trade-off between compression performance and computational complexity. [Detailed analysis to be filled based on results]

6.1.1 Model Complexity vs. Compression Gain

Higher-order models generally achieve better compression, but with diminishing returns. For instance, [specific example from results].

6.1.2 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

6.2 Data Type Characteristics

6.2.1 Text vs. Binary Data

Text data shows [pattern], while binary data exhibits [different pattern]. This suggests [explanation].

6.2.2 Structured Sequences

DNA and protein sequences benefit significantly from [specific model type] due to [biological/structural reason].

6.3 Model Selection Guidelines

Based on our experiments, we recommend:

Data Type	**Priority**	**Recommended Model**
Text	Speed	1st order Markov
Text	Compression	3rd order Markov
Binary	Balanced	FSM
DNA	Compression	2nd order Markov
General	Adaptive	LSTM (if resources allow)

Table 8: Model selection guidelines based on data type and priority.

6.4 Limitations

6.4.1 Implementation Constraints

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

6.4.2 Dataset Coverage

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

6.4.3 Neural Model Training

Online training of neural models during compression presents challenges: [discuss challenges and how they were addressed].

6.5 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

7 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.

7.1 Key Findings

Our experimental results demonstrate:

1. **Higher-order models excel on text**: The 3rd Order Markov model achieved the best compression ratio on English text, validating the importance of context.
2. **RNNs show promise**: The simple RNN implementation was competitive with Markov models and outperformed the FSM on text, suggesting that neural approaches can capture complex dependencies.
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 models were also the slowest, confirming the fundamental trade-off between compression efficiency and computational complexity.

7.2 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

7.3 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

7.4 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] Name Author. Neural network compression reference 1. *Journal*, 20XX.

[2] Name Author. Neural network compression reference 2. *Journal*, 20XX.

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

[4] Matthew V Mahoney. Adaptive weighing of context models for lossless data compression. *Florida Tech. Technical Report CS-2005-16*, 2005.

[5] Jorma Rissanen and Glen G Langdon. Arithmetic coding. *IBM Journal of research and development*, 23(2):149–162, 1979.

[6] Julian Schrittwieser et al. Online and offline neural-guided search for tsp and sat. In *ICML*, 2023.

[7] Ian H Witten, Radford M Neal, and John G Cleary. Arithmetic coding for data compression. *Communications of the ACM*, 30(6):520–540, 1987.

A Implementation Details

A.1 Arithmetic Coding Pseudocode

A.1.1 Encoding Algorithm

A.2 Model Parameters

Parameter	**Value**
Arithmetic precision	32-bit fixed point
Markov smoothing	Laplace ($\alpha = 1$)
LSTM hidden units	256
LSTM learning rate	0.001
FSM number of states	8-16 (data dependent)

Table 9: Model hyperparameters used in experiments.

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.3 Dataset Details

[To be filled with specific dataset information, file sizes, sources, etc.]

A.4 Additional Results

[To be filled with additional experimental data]