Context-based Binary Arithmetic Coding

Chih-Hsuan Chen, 110550029

Abstract—**This experiment implements Binary Arithmetic Coding (BAC) for data compression of an AlexNet model file, and compares it to Huffman coding. It evaluates the impacts of natural binary and unary binarization methods, combined with fixed and Prediction by Partial Matching (PPM) probability models with different max order, which is a context-based probability model. The experiment presents results on compression ratio, execution times, and peak memory footprint, highlighting the impact of binarization and context modeling.**

Keywords—**Arithmetic code; BAC; PPM; compression; information; entropy**

I. Introduction

Efficient data compression is critical for large files, such as multimedia file, neural network models and their training datasets. This experiment explores Binary Arithmetic Coding (BAC), which compresses binary streams. It investigates the effect of binarization on 8-bit data, comparing natural binary code and unary code. For probability modeling, both fixed distributions and context-based Prediction by Partial Matching (PPM) with different orders are impelmented and evaluated. This report systematically compares BAC's compression efficiency, execution time, and peak memory footprint under these varied configurations.

II. Implementation Details

The Binary Arithmetic Coder (BAC) was implemented in C++ to compress an AlexNet model file. The implementation supports two binarization methods and two probability modeling approaches, with configurable PPM orders.

A. Binarization Methods

1) **Natural Binary Code**
   This method directly converts each 8-bit input byte into 8 binary symbols (0s or 1s). For an input byte *c*, each bit is processed sequentially from most significant to least significant. If a bit is set 1, a '1' symbol is fed to the arithmetic coder; otherwise, a '0' symbol is fed. The total number of binarized symbols for an *N*-byte file is 8N.
2) **Unary Code**
   For an 8-bit input byte with value N (where $0 \leq N \leq 255$), this method converts it into N '1's followed by a single '0'. This results in N+1 binary symbols for each original byte. This binarization method can **lead to a highly skewed distribution of '0's and '1's, which is generally beneficial for arithmetic coding**. The total number of binarized symbols is *(sum of all byte values)* + *(number of bytes)*.

B. Probability Models

1) **Fixed Probability Distribution**
   In this approach, the probabilities of '0' and '1' symbols are pre-calculated once from the entire input file before encoding begins. These fixed probabilities are then used throughout the encoding process. The *compute_prob* function iterates through the input file, counting the occurrences of '0's and '1's based on the chosen binarization method, and then derives *prob[0]* and *prob[1].*
2) **Prediction by Partial Matching (PPM)**
   PPM is a context-based adaptive probability model. It uses a dynamic *unordered_map* (*ctx_table*) to store the counts of '0', '1', and an escape symbol (*ESC*) for different contexts. The context is a string of preceding symbols. The implementation iterates through contexts from the longest possible order (equal to *opts.ppm_order*) down to the empty context (order 0).
   - **Context Management**
     The current context (*ctx*) is updated after each symbol is processed. If the context length reaches *opts.ppm_order*, the oldest symbol is removed from the front.
   - **Symbol and *ESC* Counts**
     For each context, *_ctx_table[_ctx]* stores the counts for '0', '1', and *ESC*. A global *ctx_total_cnt unordered_map* stores the total count of symbols observed for each context.
   - **Escape Symbol Design**
     This is key detail in this PPM implementation is the handling of the escape symbol (*ESC*). When a symbol is encountered for the first time within a specific context, an escape is implicitly coded, and the model falls back to a shorter context. In this implementation, all *ESC* counts are initialized and maintained as a constant 1. This simplification is based on the observation that for only two possible symbols ('0' and '1'), the probability of needing an escape might not be high, and maintaining a constant small escape probability simplifies the model.
   - **Context Order**
     The implementation supports varying *ppm_order* values through the *-n* command-line argument (default value 2 will be used if *-n* is not specified), allowing for experimentation beyond the standard order 2.

C. Arithmetic Coding Core

The core arithmetic encoding logic is handled by the *scale* and *encode* (overloaded) functions.

The *encode* functions iterate through the binarized input symbols, updating the *lower_bound* and *upper_bound* of the coding interval based on the probabilities provided by either the fixed model or the PPM model, and the defined symbol order is: (*ESC*), '0', '1'.

The *scale* function performs outputting of bits when the coding interval becomes sufficiently large or small, effectively scaling the interval to maintain precision.

III. Open-Source Usage

The tested AlexNet model is the same as the previous assignment, and the whole program is written by myself, **no open-source project or AI tool is used or referenced**.

IV. Experimental Results

The compression experiments were conducted on an AlexNet model file (*alexnet.pth*) of size 244,409,199 bytes. The performance was evaluated based on the compression ratio, execution time, and peak memory footprint.

A. Binarization Impact

	Natural	**Unary**
prob[0]	0.483341	0.00811125
prob[1]	0.516659	0.991889
# binarized symbols	1,955,273,592	30,132,116,231

- **Natural Binary:** The distribution is almost uniform, indicating low entropy and limited potential for compression for standard arithmetic coding without context.
- **Unary Code:** The distribution is highly skewed, with '1's being overwhelmingly more probable than '0's. This high bias is ideal for arithmetic coding, as it allows almost unchanged upper and lower limits for the frequent '1' symbols. However, it results in 30.13 billion symbols, which is about **15.4** times more than natural binary code does.

B. Compression Ratio

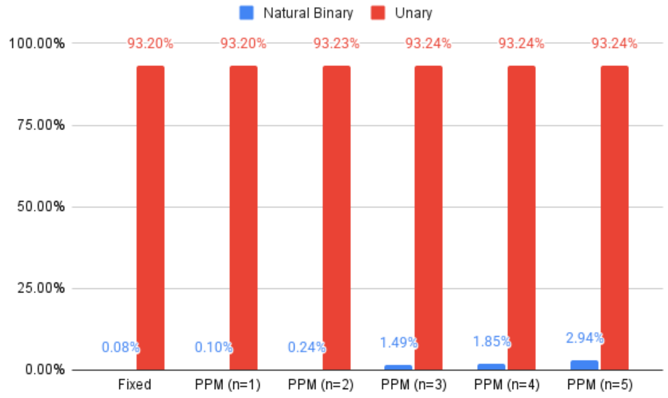

Fig. 1. Compression Ratio Comparison

- **Impact of Binarization:** Unary coding consistently achieves significantly higher compression ratios (over 93%) compared to natural binary coding (less than 3%). This obvious difference is directly attributable to the highly skewed probability distribution produced by unary binarization, which is much more favorable for arithmetic coding.
- **Fixed vs. PPM:**
  - For **Unary coding**, the difference between fixed and PPM models is negligible (e.g., 93.2006% vs. 93.2009% for *n*=1). This indicates that the extreme bias already present in unary binarization dominates the compression performance, and context modeling adds little further benefit.
  - For **Natural binary coding**, PPM models show a limited improvement in compression ratio as the PPM order increases (from 0.08% for fixed to 2.94% for *n*=5). This suggests that some local statistical dependencies are captured by PPM, even if the overall '0'/'1' distribution is close to uniform.
- **PPM Order Impact (Natural Binary):** Increasing the PPM order generally leads to a better compression ratio for natural binary code. This is because higher orders allow the model to capture longer-range dependencies, potentially leading to more accurate probability predictions. The compression ratio improves from 0.10% (*n*=1) to 2.94% (*n*=5).

C. Execution Time

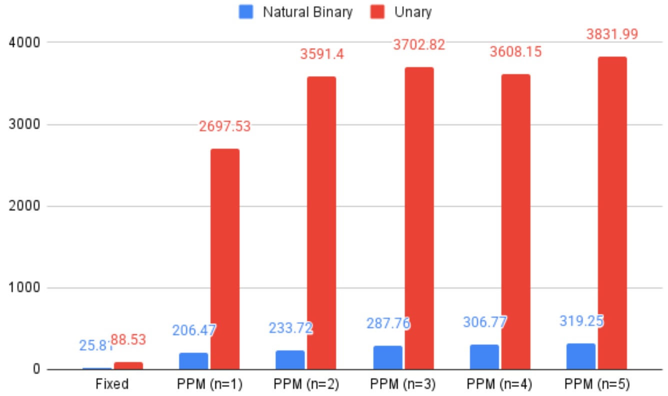

Fig. 2. Execution Time Comparison

- **PPM Overhead: PPM models introduce significant computational overhead** due to dynamic context management and map lookups. For natural binary, PPM (*n*=1) is approximately 8 times slower than fixed probability (206s vs 25s). For unary, PPM (*n*=1) is about 30 times slower than fixed probability (2697s vs 88s).
- **Impact of Binarization: Unary coding, despite its superior compression, leads to vastly longer execution times** (e.g., Unary+Fixed is 88s vs Natural+Fixed at 25s; Unary+PPM n=1 is 2697s vs Natural+PPM n=1 at 206s). This is because unary binarization generates significantly

more binary symbols, requiring many more arithmetic coding operations.

- **PPM Order Impact:** Increasing PPM order generally increases execution time. This is expected as higher orders mean longer context strings, potentially more *unordered_maps* operations (insertions/lookups), and larger context tables. The increase is more pronounced for unary coding due to the immense amount of symbols.

D. Peak Memory Footprint

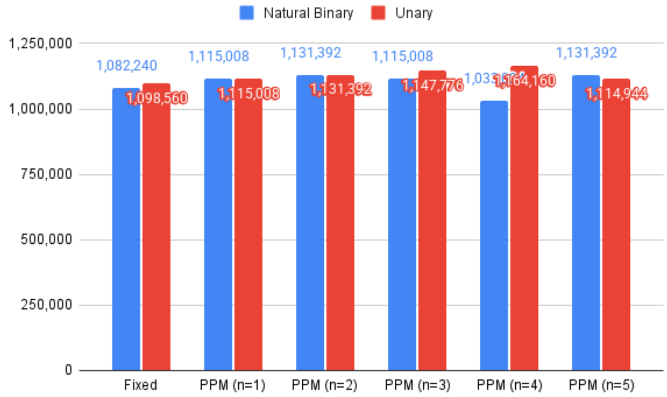

Fig. 3. Peak Memory Footprint Comparison

The peak memory footprint (maximum resident set size) generally **remains relatively stable across different configurations, staying within a range of approximately 1.1 MB to 1.4 MB**. While PPM models require *unordered_maps* for contexts, their memory usage doesn't scale dramatically with increasing order for the tested file size, likely due to the limited number of unique contexts that are actually observed or the efficiency of *unordered_maps* for short string keys and small value types.

Unary coding generally has a slightly lower peak memory footprint than natural binary for fixed probability, but the difference is minimal in PPM.

E. Comparison with Huffman Coding

The previous Huffman coding experiment found that, basically, the larger the symbol size is, the larger the compression ratio will be.

- **Natural Binary BAC vs. Huffman (8-bit symbol size):** The Natural Binary BAC, operating on individual bits where the distribution is near-uniform (approx. 0.483 for ’0’ and 0.517 for ’1’), also yielded very low compression ratios (sub-3%). This highlights that when the fundamental units being encoded (whether bits or larger symbols) lack significant statistical redundancy, both arithmetic coding and Huffman coding will struggle to achieve high compression.
- **Unary BAC vs. Huffman(32-bit symbol size):** In stark contrast, Unary BAC achieved over 93% compression. This significant difference arises because unary binarization transforms the input data into an extremely biased stream of ’0’s and ’1’s, which arithmetic coding is exceptionally adept at compressing. While Huffman coding can exploit skewness in its symbol probabilities when the symbol size is 32-bit or 64-bit, the extreme bias generated at the bit level by unary binarization provides a much more favorable condition for arithmetic coding, leading to higher compression ratios than those reported for Huffman on the 32-bit or 64-bitsymbols.

V. Discussions

A. Binarization as a Compression Strategy

The results emphatically demonstrate that the choice of binarization method is most important for the compression performance of Binary Arithmetic Coding, especially when dealing with data whose natural binary representation exhibits near-uniform bit distributions. For the AlexNet model file, the ’0’ and ’1’ bit distribution in its natural binary form is approximately 0.483 and 0.517, respectively. This near-uniformity results in minimal compression (less than 3%) with Natural BAC.

Conversely, unary binarization transforms the 8-bit source into a highly skewed binary stream, where ’1’s are overwhelmingly more probable (approx. 0.992) than ’0’s (approx. 0.008). This re-distribution of probabilities is precisely what arithmetic coding leverages to achieve remarkable compression, yielding over 93% compression ratio. This highlights that binarization isn’t merely a format conversion but a crucial pre-processing step that can dramatically alter the statistical properties of the input stream, making it more suitable to arithmetic coding.

B. Fixed vs. PPM Models

The effectiveness of PPM depends heavily on whether the source data exhibits significant context-dependent redundancies beyond simple symbol probabilities.

- For **Unary coding**, the fixed probability model already captures the extreme skewness of the ’1’s and ’0’s. PPM provides only a marginal improvement (e.g., 0.03% difference between fixed and PPM n=3 for unary), indicating that the bulk of the compressibility comes from the inherent bias of the unary representation, and short-range contexts offer little additional predictive power for this already highly predictable stream.
- For **Natural binary coding**, PPM does offer a evident, though still limited, improvement over the fixed probability model (from 0.08% to 2.94% for n=5). This suggests that even with a near-uniform global distribution of bits, there are some local dependencies that PPM can exploit to refine its probability estimates within specific contexts.

C. Impact of PPM Order and Resource Usage

Increasing the PPM order (from 1 to 5) generally leads to a gradual improvement in compression ratio for natural binary coding, as higher orders allow the model to learn and exploit longer-range statistical dependencies. However, this comes at a significant computational cost. The execution time, instructions retired, and cycles elapsed increase substantially with higher PPM orders, reflecting the overhead of managing larger context tables and performing more complex lookups (*unordered_map* operations on longer string keys). For unary coding, the benefit of increasing PPM order beyond 1 is minimal, while the computational cost continues to rise dramatically due to the significant increase in processed symbols and context updates. This indicates a point of diminishing returns where the added complexity of higher-order contexts does not translate into proportional compression gains.

Despite the increased computational time, the peak memory footprint for PPM models remains relatively constrained, typically staying within a few megabytes. This suggests that while PPM is CPU-intensive due to its many symbol-level operations, its memory overhead for storing contexts is manageable for the experimented file size and context lengths, thanks to the efficiency of *unordered_map* for short string keys.

D. *Escape Symbol Design in PPM*

The decision to keep the *ESC* (escape) count constant at 1 for all contexts simplifies the PPM model significantly. In a scenario with only two primary symbols ('0' and '1'), the probability of encountering an unseen symbol (requiring an escape) might be considered low after an initial learning phase. By assigning a constant small count to *ESC*, the model prioritizes learning and predicting '0' and '1' based on observed contexts, while still providing a fallback mechanism. This design choice proved effective given the binary nature of the symbols and the observed high compression with unary coding, where the data is highly predictable.

E. *Comparison with Huffman Coding*

The experimental results vividly illustrate the strengths and weaknesses of BAC relative to Huffman coding, particularly concerning binarization strategies. The previous Huffman coding experiment found limited compression for the AlexNet file when symbol size is 8-bit. This was attributed to the distribution of 8-bit symbols in the file, which may not have offered sufficient skewness for significant compression by Huffman.

The BAC experiment confirms this understanding: when the AlexNet file is binarized into individual bits using natural binary code, the resulting bit distribution is indeed near-uniform (around 0.483 for '0' and 0.517 for '1'). For such a near-uniform bit distribution, BAC also achieves very low compression (sub-3%), similar to how traditional entropy coders like Huffman coding would perform if applied directly to such a stream of individual bits. This highlights that both Huffman and (non-contextual) arithmetic coding rely on statistical redundancy (skewed probability distributions) to achieve compression, whether at the byte level or the bit level.

However, the dramatic success of Unary BAC (over 93% compression) demonstrates arithmetic coding's superior capability when combined with an appropriate binarization strategy that creates a highly skewed binary stream. While Huffman could also operate on such a binarized stream, arithmetic coding's ability to encode symbols with fractional bits allows it to fully exploit extreme probability biases, leading to compression ratios that Huffman coding, operating on a discrete alphabet, might struggle to match for such highly biased streams. This confirms that the choice of binarization can be more impactful than the specific entropy coder when the raw source data has low inherent redundancy at the bit level.

VI. Conclusion

This experiment confirms that binarization strategy is a key factor in the compression performance of Binary Arithmetic Coding. Unary binarization, by producing a highly skewed distribution of '0's and '1's, enables BAC to achieve over 93% compression, far surpassing the negligible compression obtained with natural binary code due to the original data's near-uniform bit distribution. While PPM provides limited improvements for natural binary, its benefits are minimal for the highly predictable unary-binarized stream, despite causing significant computational overhead proportional to the PPM order and the number of binarized symbols. The findings emphasize the importance of transforming the data into a form suitable for entropy encoding to maximize compression gains. The simplified escape symbol handling in PPM proved effective for the binary alphabet. In cases where data naturally yields highly biased binary sequences, BAC with an appropriate binarization scheme stands as a highly effective compression technique.

VII. Appendix

A. *Environment*

- Programming language: C++
- CPU: Apple M1 8-core (4 high-performance + 4 efficiency cores)
- Memory: 16GB
- Compiler: Apple clang version 16.0.0 (clang-1600.0.26.6)
- Measurement Tool: */usr/bin/time -al*