LLM Arithmetic Coder

Liam Wilbur

December 4, 2025

Introduction

Compression algorithms are used everywhere in computing. Whether it be images, storage, or the model weights in machine learning, most modern software depends on effective compression. For my 109 Challenge Project, I made a website that demonstrates an arithmetic coder, a lossless data compression algorithm and form of entropy coding.

Utilizing ideas from the paper *Language Modeling is Compression* and Claude Shannon's theorem that the best possible lossless compression is equal to the entropy of the true distribution, I have created an arithmetic coder which incorporates LLMs as the underlying probability distributions behind the algorithm. This version of the arithmetic coder, which you can try out here, is a more effective compressor (meaning less characters in encoded output) than compressors such as gzip, bzip2, and xz.

1 Arithmetic Coding

To begin, the arithmetic coder works by encoding an input message into a single number between 0 and 1, which can be decoded back to the original message.

The arithmetic coder depends upon an underlying PMF of every possible word, or token, mapped to the probability of that token. Each probability is mapped on a number line from 0 to 1 as a range instead of a discrete value, creating a CDF for every possible first token. For each consecutive token, this process is repeated using the range represented by the previous token as a new number line from 0 to 1, which is subdivided based on the conditional probabilities of all possible next tokens.

To decode, we start with an encoded number between 0 and 1 which can be used to identify which range it falls into on the initial CDF, revealing the corresponding first token. Then, it zooms into that range and divides it using conditional probabilities to find the next token. This continues until the complete message is reconstructed.

1.1 Compression Example

Let's say our alphabet is {A, I, X}. We want to take our input, then encode the input string using arithmetic coding into a binary sequence representing a decimal number between 0 and 1 (i.e. $0.0101010_2 = 0.328125$).

A user types in the input "AIXI." Here are the 4 PMFs necessary for encoding our input:

$$\begin{array}{cccc}
P(A) = 0.45 & P(A|A) = 0.2 & P(A|AI) = 0.2 & P(A|AIX) = 0.6 \\
P(I) = 0.3 & P(I|A) = 0.6 & P(I|AI) = 0.45 & P(I|AIX) = 0.2 \\
P(X) = 0.25 & P(X|A) = 0.2 & P(X|AI) = 0.35 & P(X|AIX) = 0.2
\end{array}$$

1.2 Visualizing Compression

For the input "AIXI," map the PMF of all possible first tokens on a number line from 0 to 1 to create an initial CDF (seen on the left) with ranges representing each possible token: A - $[0, 0.45]$, I - $[0.45, 0.75]$, X - $[0.75, 1]$.

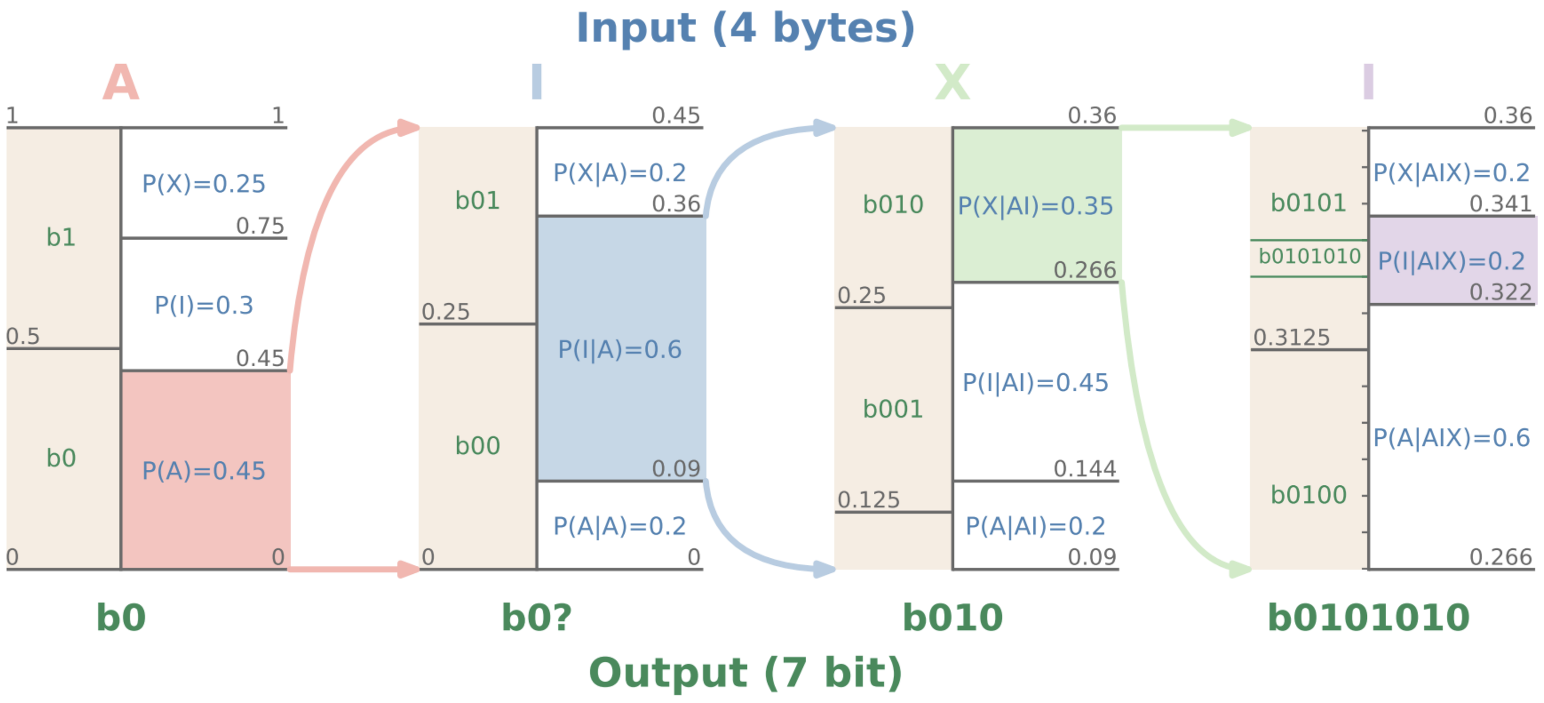

Figure 1: Arithmetic encoding of 'AIXI' (Image from *Language Modeling is Compression*)

Take the first token in our input: "A." We know that our encoded output must be between 0 and 0.45 in decimal, and must start with 0.0 in binary. Then, we treat the range $[0, 0.45]$ as if it was our original number line, now splitting each bound based upon the compound probability of $P(A)$ and $P(x|A)$ for all possible tokens x given that we have the first token "A." For example, the upper limit of the section representing "AA" is $P(A) \cdot P(A|A) = 0.45 \cdot 0.2 = 0.09$. We see the ranges representing each possible second token: A - $[0, 0.09]$, I - $[0.09, 0.36]$, X - $[0.36, 0.45]$.

Take our second token "I." We know that our encoded output must be between 0.09 and 0.36 in decimal and starts with either 0.01 or 0.00 in binary. Then we split up our segment from 0.09 to 0.36 using the PMF of all tokens given the string "AI" and take our third token "X" and continue this process until we have a final CDF where the segment $[0.322, 0.341]$ represents the possible decimal values we can use as encoded outputs for the input "AIXI."

In the case above, we encode "AIXI" as 0.328125, or 0.0101010_2 in binary. On my site, this binary is converted to base64 as a final step.

Then, when we decode our binary code, we start at the first number line for the first token and use the number 0.0101010_2 to determine which range our number is in, representing the token which is the first token in the input message. We then zoom into that section, breaking it up using conditional probabilities as shown in the encoding stage, continuing this process until we retrieve our entire original input, one token at a time.

2 Using Large Language Models

To compress using arithmetic coding, we need a probability distribution of each possible token. For a user's arbitrary input message, this could be any possible combination of tokens.

Luckily, there exists the perfect probability distribution: LLMs. Large language models are able to produce conditional PMFs to determine the probability of any possible token and the probability of each token *given* any other combination of tokens. Using open source models (Llama-3.2-1B is used here), I can retrieve the exact probability of any token given any other series of tokens.

Imagine the number lines explained above, but split up into extremely small segments using a PMF containing every possible token instead of just "A," "I," and "X." Then when we zoom into a particular segment, we split that segment up using the PMF of conditional probabilities we can retrieve from the open source model of every possible token given the existing tokens of the segment we zoomed in on.

Using this process, any input message can be encoded and decoded in the same way as in the previous section.

3 Entropy and Why LLMs Work In Compression

The reason modern LLMs are able to be used in compression and perform better than other compression algorithms is due to a shared goal in the training of the models and what makes a compression algorithm effective.

Claude Shannon's source coding theorem describes that the best possible lossless compression is equal to the entropy of the true distribution. See that the "surprise" of a sequence of tokens $x_{1:n}$ using the true distribution of sequences p is:

$$Surprise(x_{1:n}) = \sum_{i=1}^{n} -log_2 p(x_i|x_{<i})$$

We then see that the entropy of the true distribution p is:

$$H(p) = \sum_{x_{1:n}} p(x_{1:n}) \cdot \sum_{i=1}^{n} -log_2 p(x_i|x_{<i})$$

By the source coding theorem, the entropy of p is equivalent to the length of the encoded output of the perfect compressor. For our LLM however, we use an imperfect distribution $\hat{p}$. The expected length of the encoded output of the compressor is equivalent to the *cross entropy*:

$$H(p, \hat{p}) = \sum_{x_{1:n}} p(x_{1:n}) \cdot \sum_{i=1}^{n} -log_2 \hat{p}(x_i|x_{<i})$$

The closer $\hat{p}$ is to p, the more effective the compression will be. The above equation is equivalent to the log-loss function that LLMs are trained to minimize on their training data distributions, meaning they learn to predict the next token as accurately as possible. This objective simultaneously optimizes the model to be an effective compressor. In our case, a better arithmetic coder uses a probabilistic distribution $\hat{p}$ that is as close as possible to the *true distribution* p of the input text.

4 Conclusion

Billions of dollars have been spent over the past 5 years to train top models with the objective of minimizing the cross-entropy with respect to the true distribution. This is the same objective that drives effective lossless compression algorithms. As these models improve and better approximate the true distribution of natural language, the LLM arithmetic coder gets closer to the theoretical limit of compression.