Lecture 9

Context-based arithmetic coding ~~- Universal coding with LZ77~~

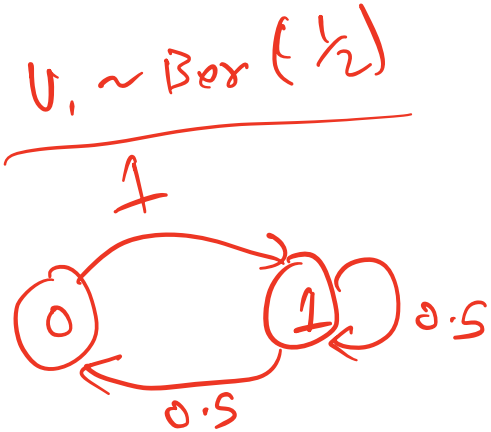
Recap

- Markov chains and stationary processes
- Conditional entropy $H(U|V) \triangleq E\left[\log \frac{1}{P(U|V)}\right] = \sum_{v\in\mathcal{V}} P(v) \sum_{u\in\mathcal{U}} H(U|V=v)$
- Entropy rate
  $H(\mathbf{U}) = \lim_{n\to\infty} H(U_{n+1}|U_1, U_2, \ldots, U_n) = \lim_{n\to\infty} \frac{H(U_1,U_2,\ldots,U_n)}{n}$

$H(U,V) = H(V) + H(U|V)$
$H(U|V) \le H(U)$

Quiz - Q1 Entropy for Markov Chain

$$U_1 = Ber(0.5)$$
$$P(U_{i+1} = 1 | U_i = 0) = 1$$
$$P(U_{i+1} = 0 | U_i = 0) = 0$$
$$P(U_{i+1} = 1 | U_i = 1) = 0.5$$
$$P(U_{i+1} = 0 | U_i = 1) = 0.5$$



1. What is $H(U_1)$?

$H(U_1) = 1$ bit

Quiz - Q1 Entropy for Markov Chain

$$U_1 = Ber(0.5)$$
$$P(U_{i+1} = 1 | U_i = 0) = 1$$
$$P(U_{i+1} = 0 | U_i = 0) = 0$$
$$P(U_{i+1} = 1 | U_i = 1) = 0.5$$
$$P(U_{i+1} = 0 | U_i = 1) = 0.5$$

2. What is $H(U_2)$?

$H(Ber(1/4)) = 0.81$ bit.

$U_1 \sim Ber(1/2)$

$P(U_2=0) = P(U_1=0)P(U_2=0|U_1=0) + P(U_1=1)P(U_2=0|U_1=1)$

$= 1/4$

$P(U_2=1) = 1 - 1/4 = 3/4$

Quiz - Q1 Entropy for Markov Chain

$U_1 = Ber(0.5)$
$P(U_{i+1} = 1 | U_i = 0) = 1$
$P(U_{i+1} = 0 | U_i = 0) = 0$
$P(U_{i+1} = 1 | U_i = 1) = 0.5$
$P(U_{i+1} = 0 | U_i = 1) = 0.5$

3. What is $H(U_2|U_1)$?

$P(U_1=0) \underbrace{H(U_2|U_1=0)}_{0} + P(U_1=1) \underbrace{H(U_2|U_1=1)}_{1} = \frac{1}{2}$

($U_1=0 \Rightarrow U_2$ is deter.)

($U_1=1 \Rightarrow U_2$ is $Ber(1/2)$)

Quiz - Q1 Entropy for Markov Chain

$U_1 = Ber(0.5)$
$P(U_{i+1} = 1 | U_i = 0) = 1$
$P(U_{i+1} = 0 | U_i = 0) = 0$
$P(U_{i+1} = 1 | U_i = 1) = 0.5$
$P(U_{i+1} = 0 | U_i = 1) = 0.5$

4. Is this process stationary?

No because $H(U_2) \neq H(U_1)$

Quiz - Q1 Entropy for Markov Chain

Change initial distribution to make Markov chain stationary.

$$U_1 = Ber(2/3)$$
$$P(U_{i+1} = 1 | U_i = 0) = 1$$
$$P(U_{i+1} = 0 | U_i = 0) = 0$$
$$P(U_{i+1} = 1 | U_i = 1) = 0.5$$
$$P(U_{i+1} = 0 | U_i = 1) = 0.5$$

5. Calculate the entropy rate $H(\mathbf{U})$ of this stationary Markov source.

How to achieve the entropy rate?

Let's start with a first-order Markov source

Recall entropy rate

$$H(\mathbf{U}) = \lim_{n\to\infty} \frac{H(U_1,U_2,\ldots,U_n)}{n} = \lim_{n\to\infty} H(U_{n+1}|U_1,U_2,\ldots,U_n)$$

For a first-order Markov source this is simply

$$H(\mathbf{U}) = \lim_{n\to\infty} \frac{H(U_1,U_2,\ldots,U_n)}{n} = H(U_2|U_1)$$

Suggests two ways:

1. Coding in bigger and bigger blocks (to achieve $\lim_{n\to\infty} \frac{H(U_1,U_2,\ldots,U_n)}{n}$)
2. Coding incrementally (to achieve $H(U_2|U_1)$)

Working with known 1st order Markov source

Idea 1: Use Huffman on blocks of length n.

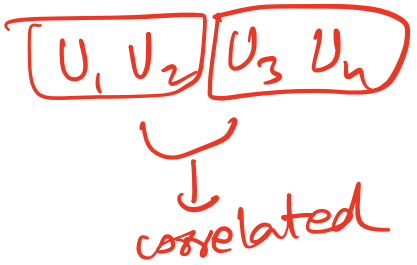
- Usual concerns: big block size, complexity, etc.
- For non-iid sources, working on independent symbols is just plain suboptimal even discounting the effects of non-dyadic distributions.

Exercise: Compute $H(U_1)$ and $H(U_1, U_2)$ for

$$U_1 \sim Unif(\{0, 1, 2\})$$

$$U_{i+1} = (U_i + Z_i) \bmod 3$$

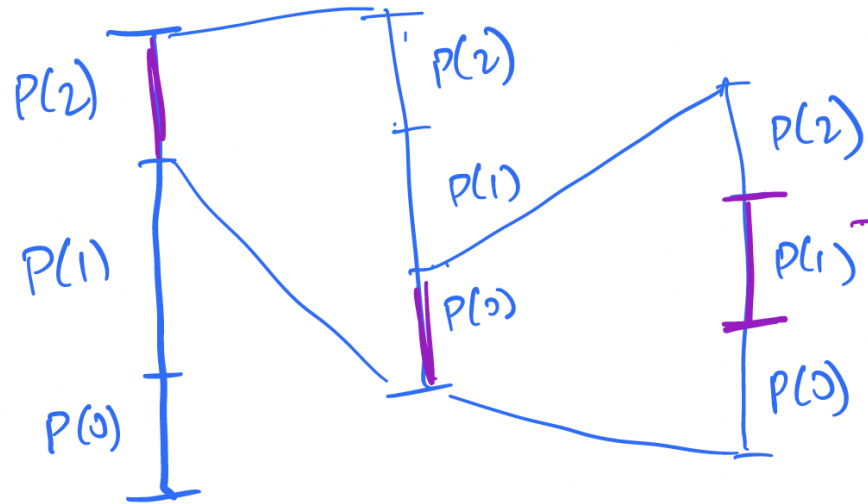
$$Z_i \sim Ber\left(\frac{1}{2}\right)$$



and compare to $H(\mathbf{U})$.

Recap - arithmetic coding for iid probability model

$X_1, X_2 \ldots \sim$ iid X



interval length $= P(2) \times P(0) \times P(1)$

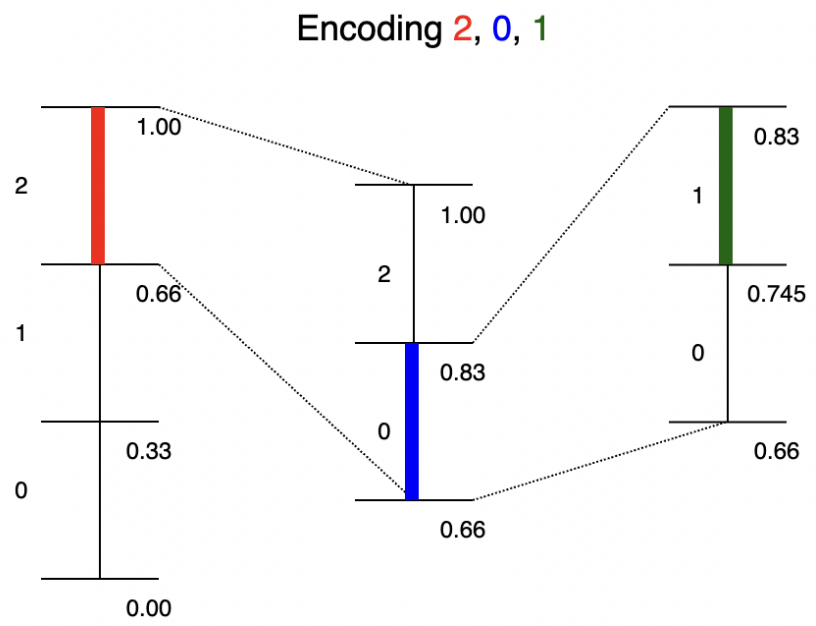
$$\text{Code length} \approx \log_2 \frac{1}{\text{interval length}}$$

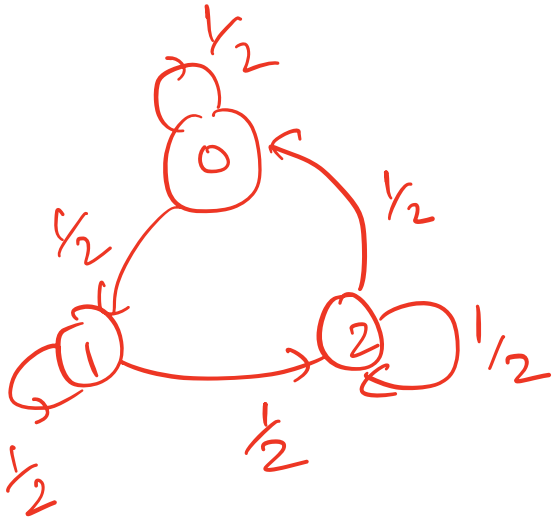
$$\approx \log_2 \frac{1}{P(x_1)P(x_2)\cdots P(x_n)}$$

$$E[\text{code length}] \approx nH(X)$$

n-block

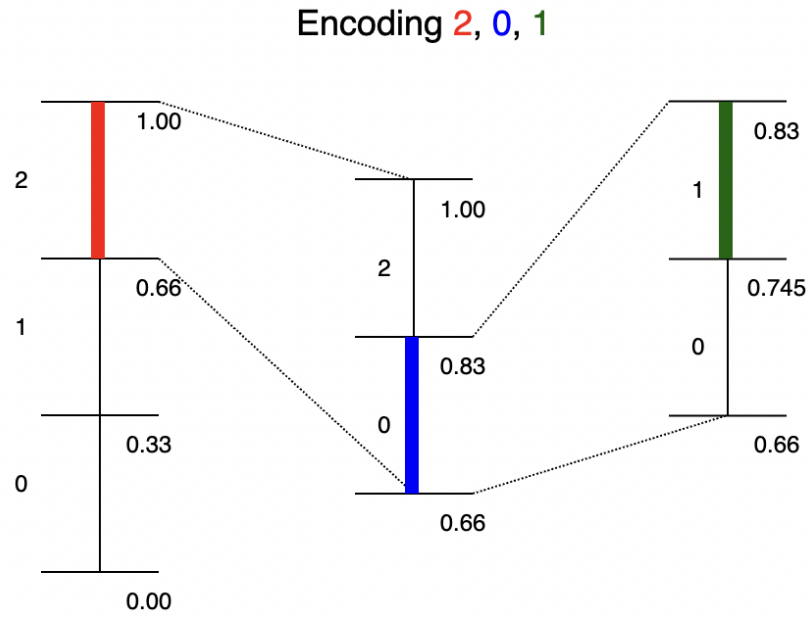
Working with known 1st order Markov source





Question: Can you explain the general idea?

Working with known 1st order Markov source



Question: Can you explain the general idea?

Answer: At every step, split interval by $P(-|u_{i-1})$ [more generally by $P(-|\text{entire past})$].

Arithmetic coding for known 1st order Markov source

Length of interval after encoding $u_1, u_2, u_3, \ldots, u_n =$
$P(u_1)P(u_2|u_1)\ldots P(u_n|u_{n-1})$

Bits for encoding ~ $\log_2 \frac{1}{P(u_1)P(u_2|u_1)\ldots P(u_n|u_{n-1})}$

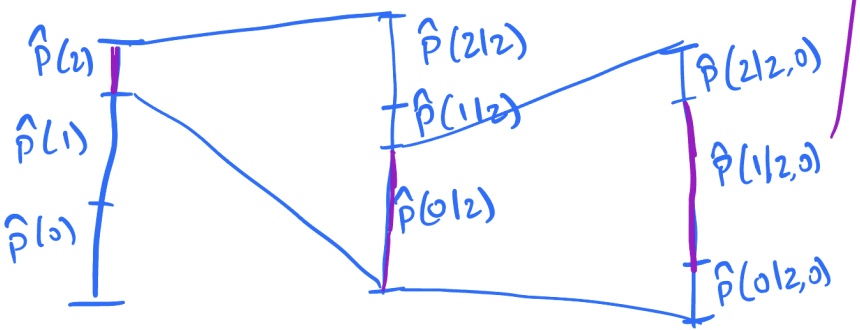
Expected bits per symbol

$$
\begin{aligned}
&\sim \frac{1}{n} E\left[\log_2 \frac{1}{P(U_1)P(U_2|U_1)\ldots P(U_n|U_{n-1})}\right] \\
&= \frac{1}{n} E\left[\log_2 \frac{1}{P(U_1)}\right] + \frac{1}{n}\sum_{i=2}^{n} E\left[\log_2 \frac{1}{P(U_i|U_{i-1})}\right] \\
&= \frac{1}{n} H(U_1) + \frac{n-1}{n} H(U_2|U_1) \\
&\sim H(U_2|U_1)
\end{aligned}
$$

Arithmetic coding for general probability model

Data: $X_1, X_2, \ldots$

Model: $\hat{P}(x_i \mid x_1, \ldots, x_{i-1})$



interval length

$\hat{P}(2) \times \hat{P}(0|2)\,\hat{P}(1|2,0)$

Code length

$\approx \log_2 \frac{1}{\text{interval length}}$

$= \log_2 \frac{1}{\hat{P}(x_1)\,\hat{P}(x_2|x_1)\cdots}$

Better Predictor $\Rightarrow$ Higher prob. for observed sequence $\Rightarrow$ larger interval $\Rightarrow$ Smaller code length

So as long as we can estimate the probability distribution of the next symbol given some context, we can use arithmetic coding to encode the data.

The bits used to encode u_n is simply $\log_2 \frac{1}{P(u_n|past)}$

Good model
$P(u_n|past) \approx 1$
$\log_2 \frac{1}{1} \approx 0$ bits

Higher the probability of the actually observed symbol, lower the bits you pay!

Bad model
$P(u_n|past) \approx 0$
$\log_2 \frac{1}{\epsilon} \approx$ large

Predicting the next token with Llama

```
>>> predict_next_token("than")
Token: x, Probability: 18.6%
Token: e, Probability: 8.5%
Token: , Probability: 5.2%
Token: the, Probability: 5.2%
Token: king, Probability: 4.3%
```

Predicting the next token with Llama

```
>>> predict_next_token("louder than")
Token: words, Probability: 30.4%
Token: love, Probability: 11.9%
Token: a, Probability: 11.2%
Token: the, Probability: 5.8%
Token: bombs, Probability: 4.7%
```

Predicting the next token with Llama

```
>>> predict_next_token("speak louder than")
Token: words, Probability: 47.8%
Token: money, Probability: 7.8%
Token: a, Probability: 4.7%
Token: the, Probability: 3.2%
Token: actions, Probability: 2.5%
```

if words is the true next symbol

$\log_2 \frac{1}{1/2} = 1$ bit

Predicting the next token with Llama

```
>>> predict_next_token("Actions speak louder than")
Token: words, Probability: 96.5%
Token: the, Probability: 0.2%
Token: a, Probability: 0.1%
Token: any, Probability: 0.1%
Token: Words, Probability: 0.1%
```

Predicting the next token with Llama

```
>>> predict_next_token("Stanford's data compression")
Token: research, Probability: 9.0%
Token: group, Probability: 7.5%
Token: and, Probability: 5.6%
Token: library, Probability: 5.3%
Token: team, Probability: 4.1%
```

Predicting the next token with Llama

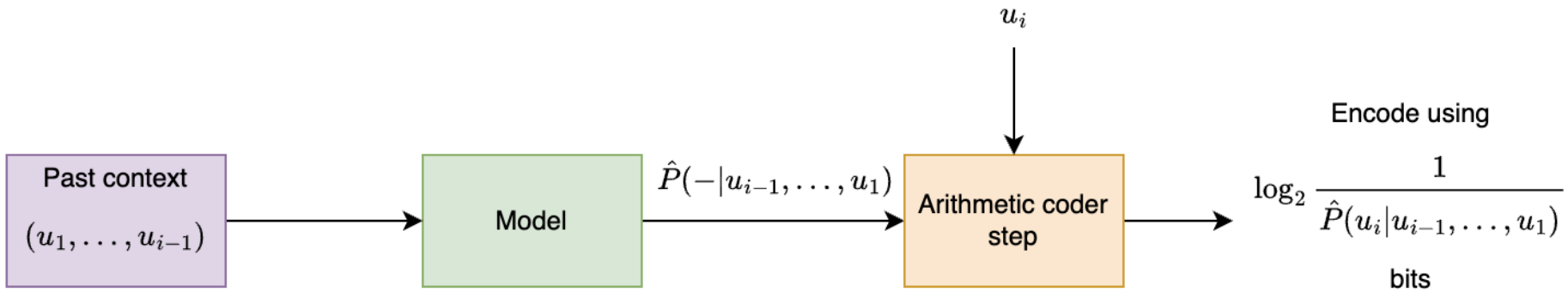
```
>>> predict_next_token("Enroling in Stanford's data compression")
Token: course, Probability: 56.6%
Token: class, Probability: 10.1%
Token: program, Probability: 4.8%
Token: courses, Probability: 4.5%
Token: and, Probability: 3.0%
```

For a kth order model, the previous k symbols are sufficient to predict the next symbol.

In general, the more past context you can use, the better the prediction.

Before we look at some specific prediction models, let's look at the general framework for context-based arithmetic coding.

Context-based arithmetic coding

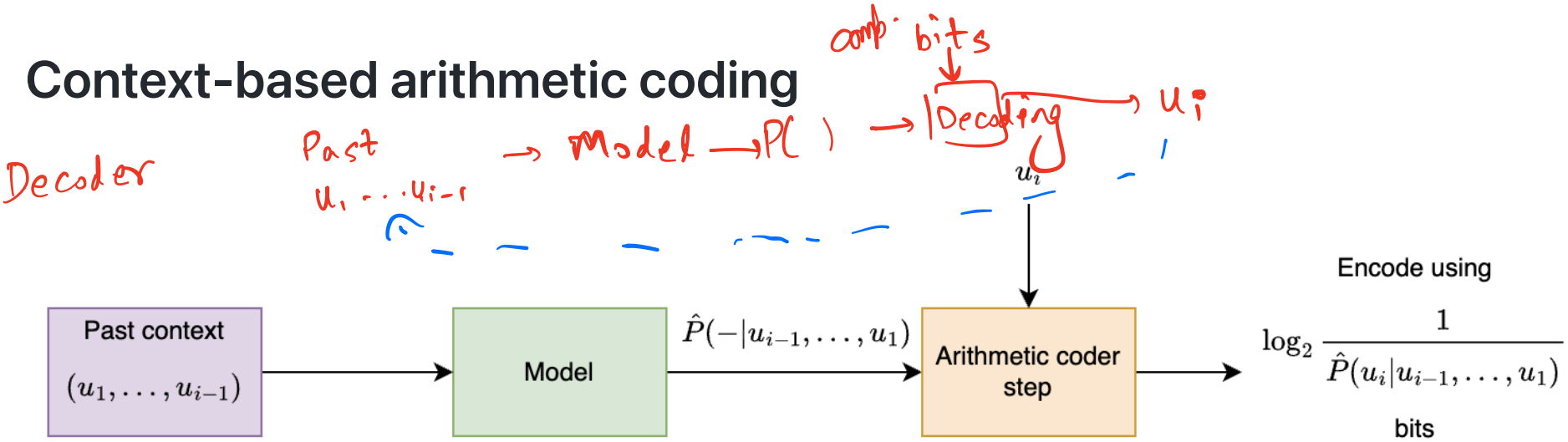


Total bits for encoding:

$$\sum_{i=1}^{n} \log_2 \frac{1}{\hat{P}(u_i|u_1, \ldots, u_{i-1})}$$

Question: How would the decoding work?

Context-based arithmetic coding



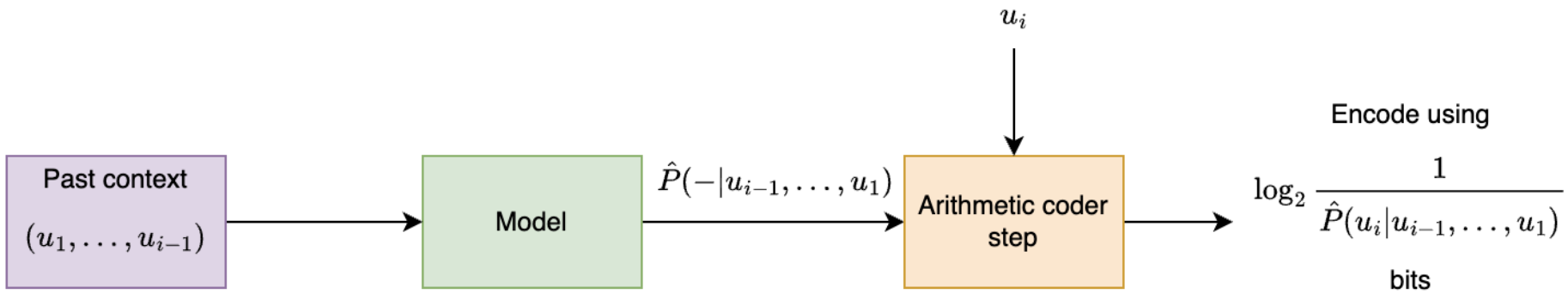
Total bits for encoding:

$$\sum_{i=1}^{n} \log_2 \frac{1}{\hat{P}(u_i|u_1, \ldots, u_{i-1})}$$

Question: How would the decoding work?

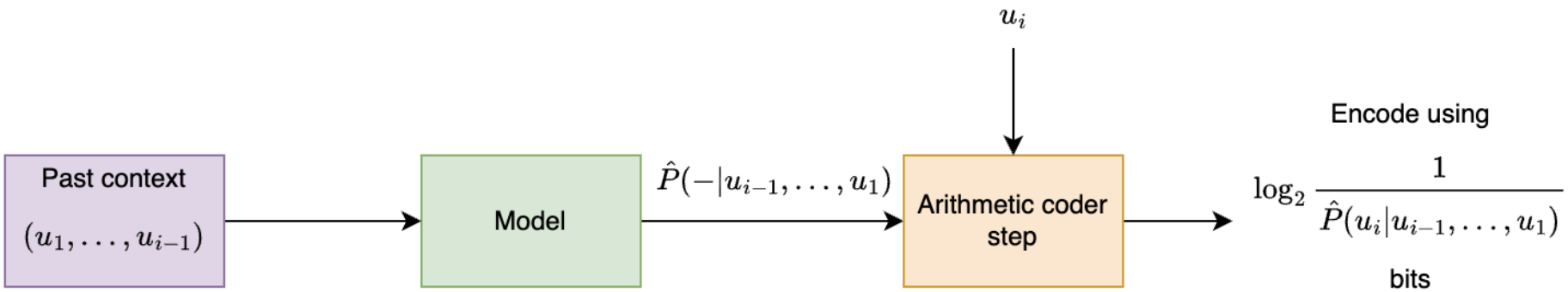
Answer: Decoder uses same model, at step i it has access to $u_1, \ldots, u_{i-1}$ already decoded and so can generate the $\hat{P}$ for the arithmetic coding step!

Context-based arithmetic coding



Question: I don't already have a model. What should I do?

Context-based arithmetic coding



Question: I don't already have a model? What should I do?

Option 1: Two pass: first build ("train") model from data, then encode using it.

Option 2: Adaptive: build ("train") model from data as we see it (more on this shortly).

Two-pass vs. adaptive

Two-pass approach

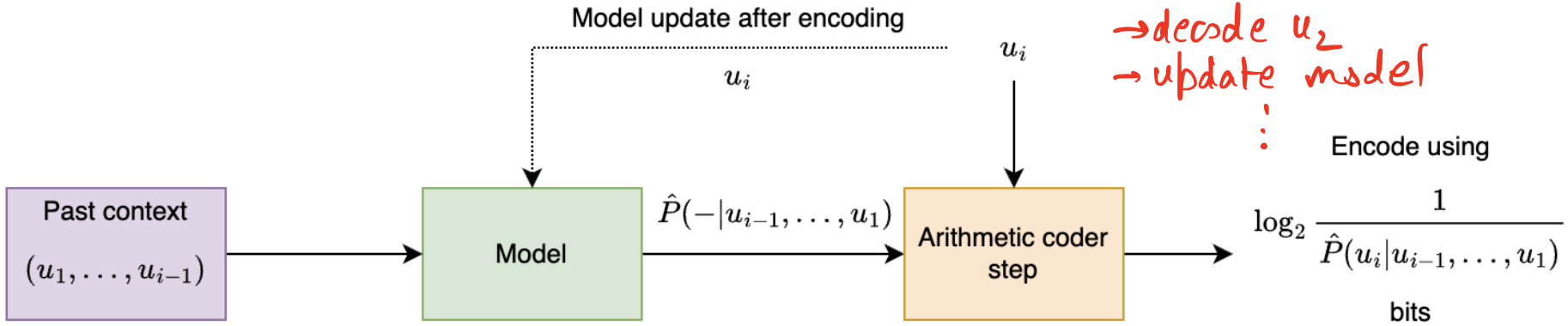
✅ learn model from entire data, leading to potentially better compression
✅ more suited for parallelization
❌ need to store model in compressed file
❌ need two passes over data, not suitable for streaming
❌ might not work well with changing statistics

Adaptive approach

✅ no need to store the model
✅ suitable for streaming
❌ adaptively learning model leads to inefficiency for initial samples
✅ works pretty well in practice!

Adaptive context-based arithmetic coding

Decoder
→ decode u_1
→ update model
→ decode u_2
→ update model
⋮

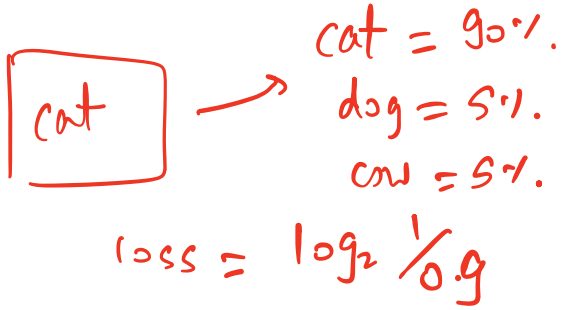


⚠️ Important for encoder and decoder to share exactly the same model state at every step (including at initialization).

⚠️ Don't go about updating model with u_i before you perform the encoding for u_i.

⚠️ Try not to provide 0 probability to any symbol.

✗ Encoder
- updates model w/ u_1
→ encodes u_1



Compression and prediction

Cross-entropy loss for prediction (classes $\mathcal{C}$, predicted probabilities $\hat{P}$, ground truth class: y):

$$\sum_{c \in \mathcal{C}} \mathbf{1}_{y_i = c} \log_2 \frac{1}{\hat{P}(c | y_1, \ldots, y_{i-1})}$$

Loss incurred when ground truth is y_i is $\log_2 \frac{1}{\hat{P}(y_i | y_1, \ldots, y_{i-1})}$

Exactly matches the number of bits used for encoding with arithmetic coding!

Compression and prediction

- Good prediction => Good compression
- Compression = having a good model for the data
- Need not always explicitly model the data
- Possible to use rANS instead of arithmetic coding in some settings

$\ell = \log_2 \frac{1}{p}$

$p = 2^{-\ell}$

compressor encodes $x_1, \ldots x_n$ in ℓ bits

then $\hat{p}(x_1, \ldots x_n) = 2^{-\ell}$

Compression and prediction

- Each compressor induces a predictor!
- Recall relation between code length and induced probability model $p \sim 2^{-l}$
- Generalizes to prediction setting
- Explicitly obtaining the prediction probabilities easier with some compressors than others

Prediction models used for compression

kth order adaptive arithmetic coding

- Start with a frequency of 1 for each symbol in the $(k+1)$th order alphabet (to avoid zero probabilities)
- As you see symbols, update the frequency counts
- At each step you have a probability distribution over the alphabet induced by the counts

Remember to update the counts with a symbol after you encode a symbol!

Example: if you saw BANA in past followed by N 90% of times and by L 10% of times, then predict N with probability 0.9 and L with probability 0.1 given a context of BANA.

Example: 1st order adaptive arithmetic coding

→ 2nd order statistics

Data: 101011

Initial frequencies/counts:

$c(0,0) = 1$
$c(0,1) = 1$
$c(1,0) = 1$
$c(1,1) = 1$

Assume past is padded with 0s

Example: 1st order adaptive arithmetic coding

Data: 0101011

Current symbol: 1

Previous symbol: 0 (padding)

Predicted probability: $P(1|0) = \frac{c(0,1)}{c(0,0)+c(0,1)} = \frac{1}{2}$

Counts:

$c(0,0) = 1$

$c(0,1) = 1 \to 2$

$c(1,0) = 1$

$c(1,1) = 1$

0, 1 → 9 times

0, 0 → 1 time

$P(1|0) = \frac{9}{10}$

Example: 1st order adaptive arithmetic coding

Data: 1**0**1011

Current symbol: 0

Previous symbol: 1

Predicted probability: $P(0|1) = \frac{c(1,0)}{c(1,0)+c(1,1)} = \frac{1}{2}$

Counts:

$c(0,0) = 1$

$c(0,1) = 2$

$c(1,0) = 1 \to 2$

$c(1,1) = 1$

Example: 1st order adaptive arithmetic coding

Data: 101011

Current symbol: 1

Previous symbol: 0

Predicted probability: $P(1|0) = \frac{c(0,1)}{c(0,1)+c(1,1)} = \frac{2}{3}$

Counts:

$c(0,0) = 1$

$c(0,1) = 2 \rightarrow 3$

$c(1,0) = 2$

$c(1,1) = 1$

Example: 1st order adaptive arithmetic coding

Data: 101**0**11

Current symbol: 0

Previous symbol: 1

Predicted probability: $P(0|1) = \frac{c(1,0)}{c(1,0)+c(1,1)} = \frac{2}{3}$

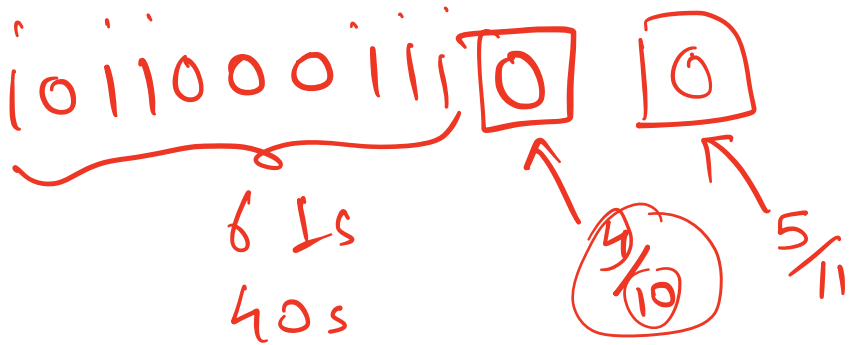
Counts:

$c(0,0) = 1$

$c(0,1) = 3$

$c(1,0) = 2 \rightarrow 3$

$c(1,1) = 1$



Observations

- Over time we learn the empirical distribution of the data
- Initially start off with uniform distribution - can change prior to enforce some prior knowledge [both encoder and decoder need to know!] & nothing 0
- You can do this for $k = 0$ (iid data with unknown distribution)!

kth order adaptive arithmetic coding (AAC)

```
    def freqs_current(self):
        """Calculate the current freqs. We use the past k symbols to pick out
        the corresponding frequencies for the (k+1)th.
        """
            freqs_given_context = np.ravel(self.freqs_kplus1_tuple[tuple(self.past_k)])
```

```
    def update_model(self, s):
        """function to update the probability model. This basically involves update the count
        for the most recently seen (k+1) tuple.

        Args:
            s (Symbol): the next symbol
        """
        # updates the model based on the new symbol
        # index self.freqs_kplus1_tuple using (past_k, s) [need to map s to index]
        self.freqs_kplus1_tuple[(*self.past_k, s)] += 1

        self.past_k = self.past_k[1:] + [s]
```

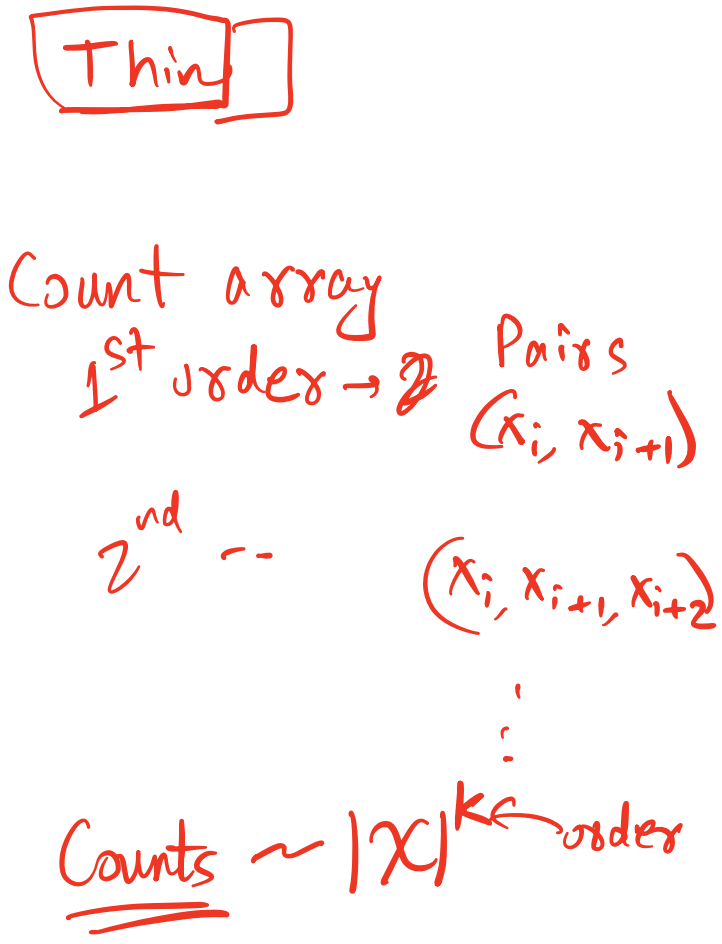
kth order adaptive arithmetic coding (AAC)

On `sherlock.txt`:

```
>>> with open("sherlock.txt") as f:
>>>     data = f.read()
>>>
>>> data_block = DataBlock(data)
>>> alphabet = list(data_block.get_alphabet())
>>> aec_params = AECParams()
>>> encoder = ArithmeticEncoder(aec_params, AdaptiveOrderKFreqModel(alphabet, k, aec_params.MAX_ALLOWED_TOTAL_FREQ))
>>> encoded_bitarray = encoder.encode_block(data_block)
>>> print(len(encoded_bitarray)//8) # convert to bytes
```

kth order adaptive arithmetic coding

Compressor	compressed bits/byte
0th order	4.26
1st order	3.34
2nd order	**2.87**
3rd order	3.10
gzip	2.78
bzip2	**2.05**



kth order adaptive arithmetic coding

Compressor	compressed bits/byte
0th order	4.26
1st order	3.34
2nd order	**2.87**
3rd order	3.10
gzip	2.78
bzip2	**2.05**

Question: Why is order 3 doing worse than order 2?

kth order adaptive arithmetic coding (AAC)

this
This

Limitations

- slow, memory complexity grows exponentially in k
- counts become very sparse for large k, leading to worse performance
- unable to exploit similarities in prediction for *similar* contexts

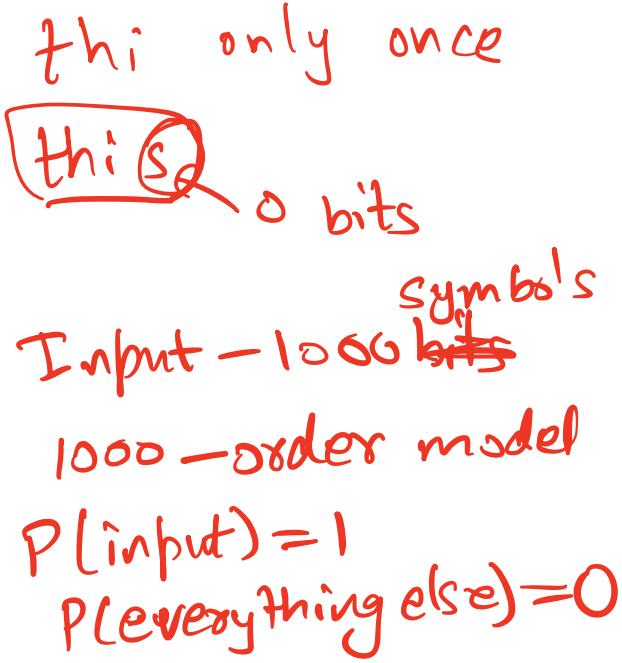
thi
Thi

Some of these can be overcome with smarter modeling as discussed later.

Note: Despite their performance limitations, context based models are still employed as the entropy coding stage after suitably preprocessing the data (LZ, BWT, etc.).

What if we did a two-pass approach?

order	adaptive	empirical conditional entropy
0th order	4.26	4.26
1st order	3.34	3.27
2nd order	2.87	2.44
3rd order	3.10	1.86



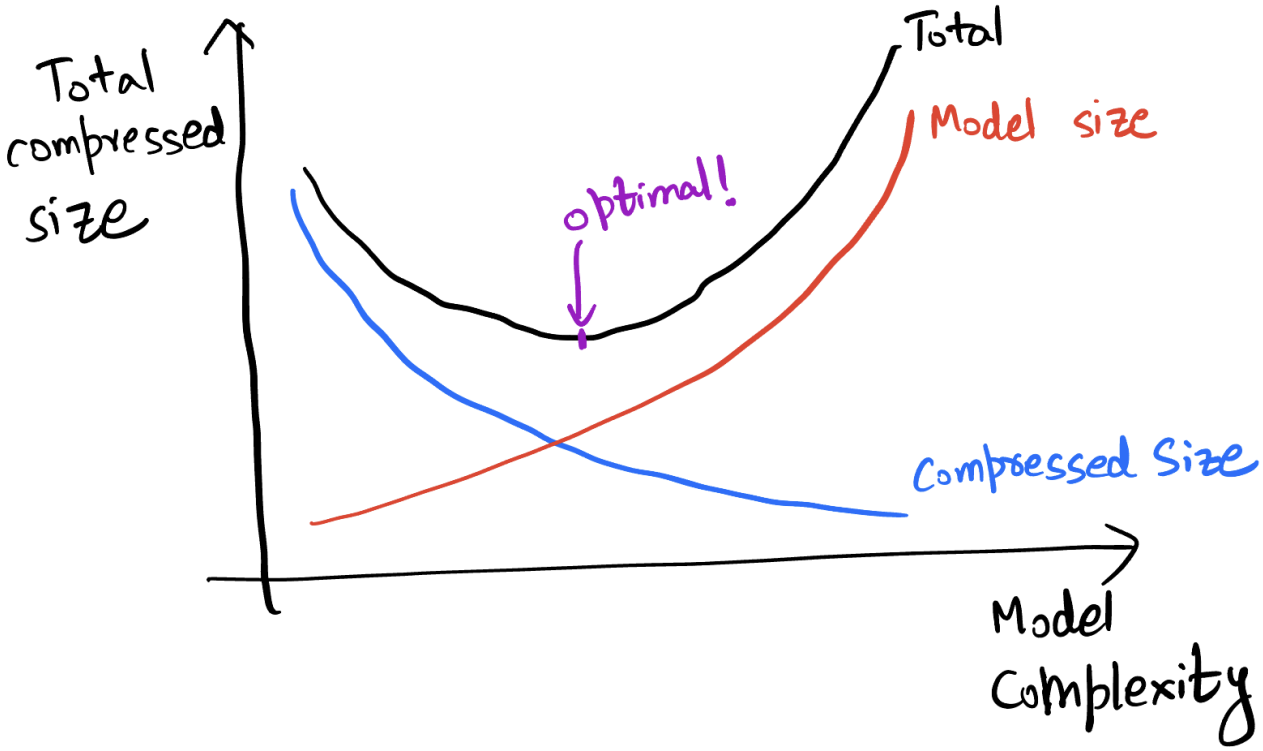
Why is there an increasing gap between adaptive coding performance and empirical entropy as we increase the order?

Cost of storing the model!

- As the order increases, knowing the empirical distribution becomes closer to just storing the data itself in the model.
- At the extreme, you just have a single $|data_size|$ long context and the model is just the data itself!
- We need to account for the cost of storing the model.
- In practice, adaptive models are often preferred due to their simplicity and not requiring two passes over the data.

Minimum Description Length (MDL) principle

Minimize sum of model size and compressed size given model



Prediction models used for compression

- kth order adaptive (in SCL): https://github.com/kedartatwawadi/stanford_compression_library/blob/main/scl/compressors/probability_models.py
- Solving the sparse count problem:
  - Context Tree Weighting (CTW)
  - Prediction by Partial Matching (PPM)
- Advanced prediction models:
  - Neural net based: NNCP, Tensorflow-compress, DZip
  - Ensemble methods: CMIX, CMIX talk
- Resources: https://mattmahoney.net/dc/dce.html#Section_4

These are some of the most powerful compressors around, but often too slow for many applications!

DeepZip/NNCP framework

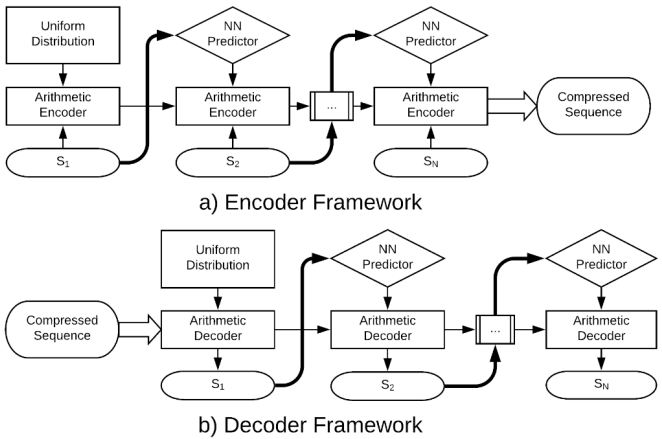
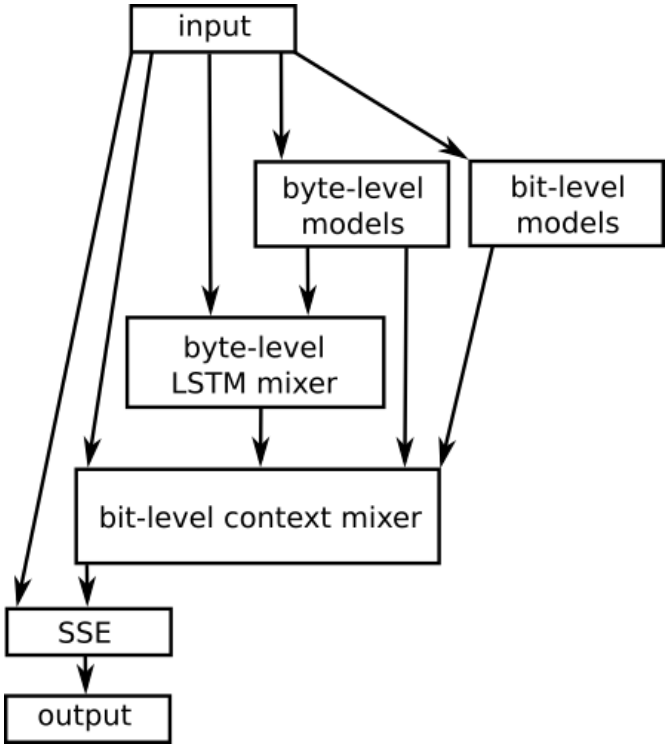


Figure 1: Encoder-Decoder Framework.

DeepZip: Two pass approach, first train model on data, then use it to compress data. Model stored in compressed file.

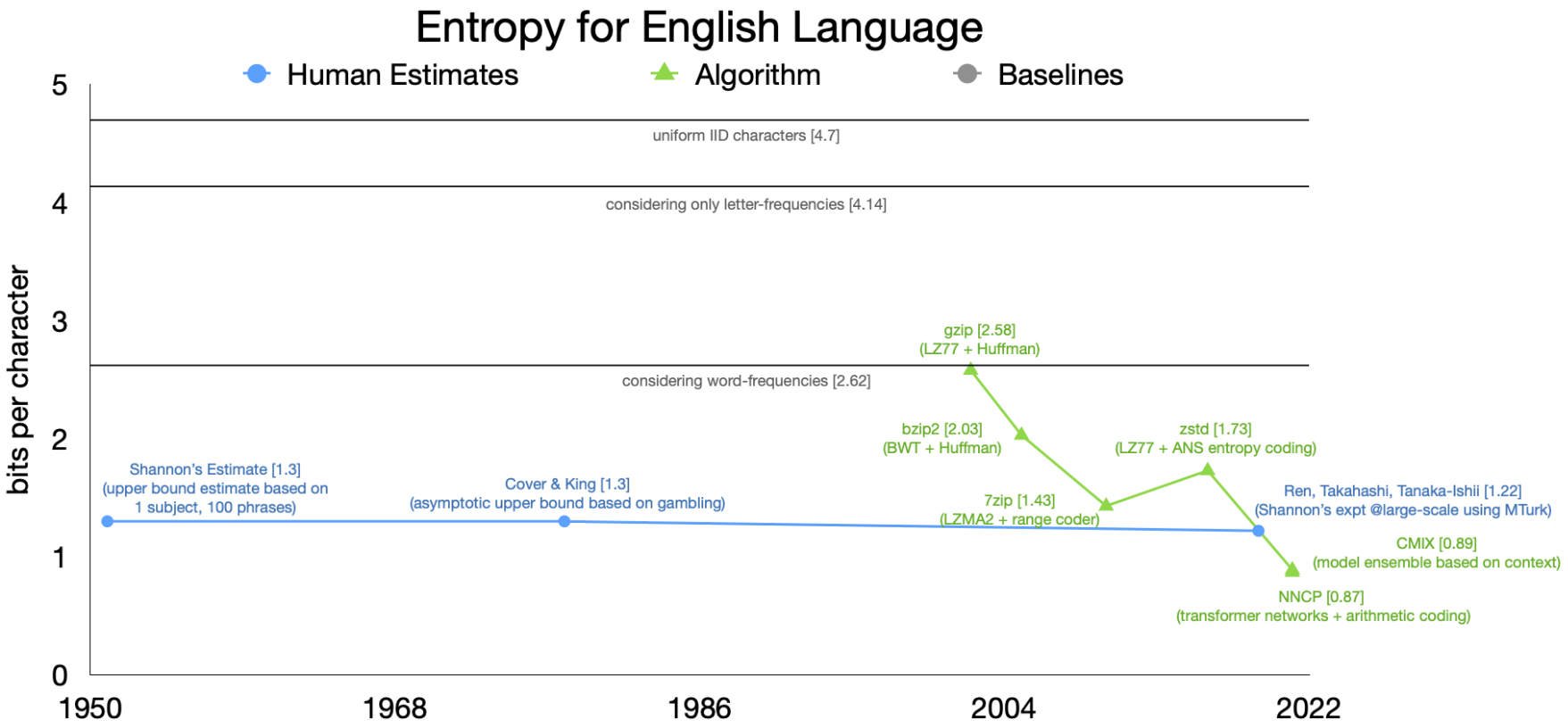
NNCP: Keep updating NN predictor periodically as you see more data.

CMIX context mixing



Use neural net to mix the predictions of various context models, according to how well they predicted in the past.

Text compression over the years



LLM based compression - going beyond MDL setting

- Throw your most powerful predictor, a model that would take gigabytes to describe, and use that to compress the data.
- In a traditional setting, the size of the model would be prohibitive to store as part of the compressed file (also painstakingly slow to compress and decompress)
- When is this setting useful?

LLM based compression - going beyond MDL setting

- Throw your most powerful predictor, a model that would take gigabytes to describe, and use that to compress the data.
- In a traditional setting, the size of the model would be prohibitive to store as part of the compressed file (also painstakingly slow to compressi and decompress)
- When is this setting useful?
  - For understanding limits of compressibility/entropy rate estimation
  - When there is a large amount of data of the same type and you can afford to deploy the model separately on each decompression node
  - To demonstrate concepts in a compression course!

LLM perplexity loss

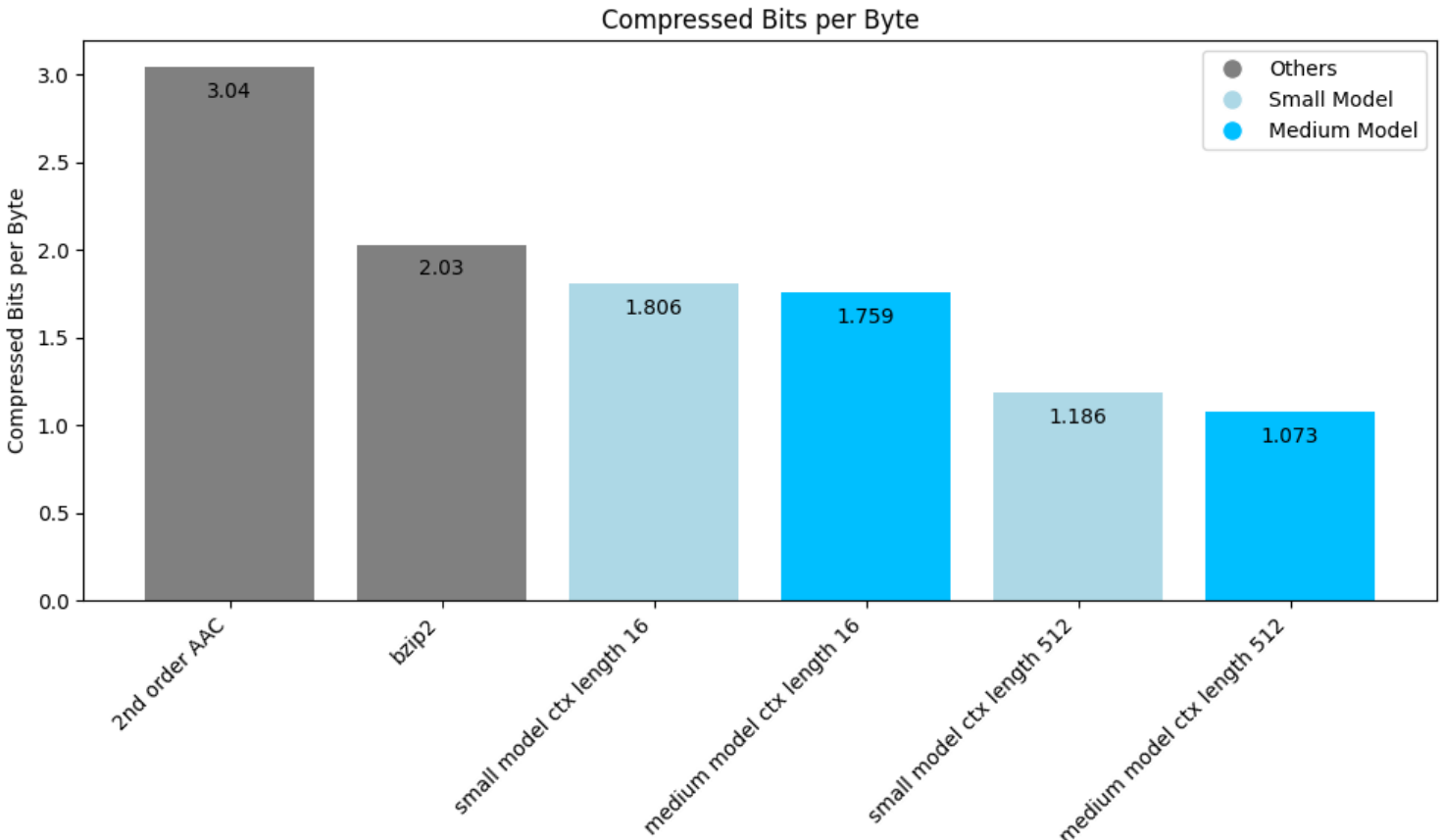
LLMs are trained as predictors, and their loss function is simply the cross-entropy loss (or perplexity = $2^{\text{cross-entropy}}$). Thus they are in-effect being trained for compression!

Let's use ts_zip: Text Compression using Large Language Models to use LLMs for compression!

We use rwkv_169M and rwkv_430M models.

LLM results (note context size is in tokens not characters)

Results on a 2023 novel (848 KB English text).



LLM results

On an ancient Pali text (transcribed in Roman script):

Compressor	compressed bits/byte
2nd order AAC	2.66
gzip	2.11
bzip2	1.71
small LLM model	2.41
medium LLM model	2.19

Why do the LLMs no longer do so well compared to bzip2?

Even more powerful models (credit: Kedar)!

https://gist.github.com/chachachaudhary274/707eeed868167b2e8c30000d747316d9

Llama-13B (4 bit quantized)

Dataset	Context length	compressed bits/byte
2023 short story	10	1.228
2023 short story	50	1.027
2023 short story	512	0.874

Even more powerful models (credit: Kedar)!

https://gist.github.com/chachachaudhary274/707eeed868167b2e8c30000d747316d9

Llama-13B (4 bit quantized)

Dataset	Context length	compressed bits/byte
Sherlock	10	1.433
Sherlock	50	0.542
Sherlock	512	0.200

$\log_2 \frac{1}{p}$

This one is way too good! What's going on?

LLM based compression

- Remarkable results
- Be careful about model-data mismatch (e.g., Pali text) and overfitting to training data (e.g., Sherlock)
- Very slow and compute intensive
  - might become practical with hardware acceleration in future (for some applications)
- Resources:
  - ts_zip: https://bellard.org/ts_server/ts_zip.html
  - DeepMind paper: https://aps.arxiv.org/abs/2309.10668

Next time:- LZ77
+
lossless compression in practice

Thank You!