



# Lecture 8

## Compression beyond iid data

# Recap

- Huffman, Arithmetic, ANS
- We know how to achieve the entropy in a computationally efficient manner.

```
$ cat sherlock.txt
```

```
...
```

```
In mere size and strength it was a terrible creature which was  
lying stretched before us. It was not a pure bloodhound and it  
was not a pure mastiff; but it appeared to be a combination of  
the two—gaunt, savage, and as large as a small lioness. Even now  
in the stillness of death, the huge jaws seemed to be dripping  
with a bluish flame and the small, deep-set, cruel eyes were  
ringed with fire. I placed my hand upon the glowing muzzle, and  
as I held them up my own fingers smouldered and gleamed in the  
darkness.
```

```
“Phosphorus,” I said.
```

```
“A cunning preparation of it,” said Holmes, sniffing at the dead
```

```
...
```

Let's try and compress this 387 KB book.

```
>>> from core.data_block import DataBlock
>>>
>>> with open("sherlock.txt") as f:
>>>     data = f.read()
>>>
>>> print(DataBlock(data).get_entropy()*len(data)/8, "bytes")

199833 bytes
```

```
$ gzip < sherlock.txt | wc -c
134718
```

```
$ bzip2 < sherlock.txt | wc -c
99679
```

What's up? What are we missing here? Any suggestions?

1. Data is not iid.
2. Maybe the entire file doesn't have the same distribution (think concatenating an English novel with a Hindi novel).

In the next few lectures, we will discuss methods to compress real-life data, attempting to handle non-iid data whose distribution we do not know a priori.

## Beyond iid data

- text
- images
- video
- tables
- basically anything in real life



## Probability recap

Recall for  $U^n = (U_1, \dots, U_n)$ :

for iid

$$P(U^n) = \prod_{i=1}^n P(U_i)$$

in general

$$P(U^n) = \prod_{i=1}^n P(U_i | U^{i-1}) = \prod_{i=1}^n P(U_i | U_1, \dots, U_{i-1})$$

## Stochastic process (aka random process)

Given alphabet  $\mathcal{U}$ , a *stochastic process*  $(U_1, U_2, \dots)$  can have arbitrary dependence across the elements and is characterized by:

$P((U_1, U_2, \dots, U_n) = (u_1, u_2, \dots, u_n))$  for  $n = 1, 2, \dots$  and  $(u_1, u_2, \dots, u_n) \in \mathcal{U}^n$ .

Way too general to be of much use.

# Stationary stochastic process

## Definition: Stationary Process

A stationary process is a stochastic process that is time-invariant, i.e., the probability distribution doesn't change with time (here time refers to the index in the sequence).

More precisely, we have

$$P(U_1 = u_1, U_2 = u_2, \dots, U_n = u_n) = P(U_{l+1} = u_1, U_{l+2} = u_2, \dots, U_{l+n} = u_n)$$

for every  $n$ , every shift  $l$  and all  $(u_1, u_2, \dots, u_n) \in \mathcal{U}^n$ .

- Mean, variance etc. do not change with  $n$ .
- Can still have arbitrary time dependence.

## Examples

IID sequences: e.g., sequence of fair iid coin tosses

## Examples: Stationary time-invariant Markov processes

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

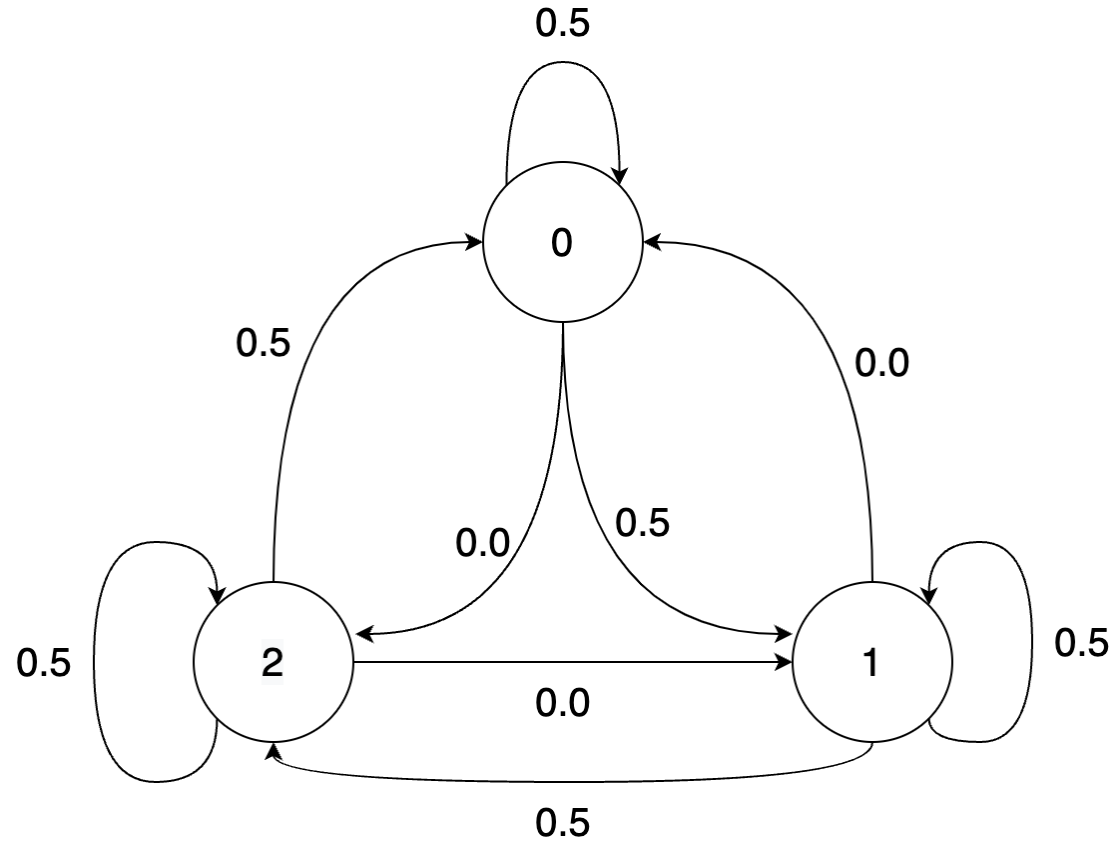
$$U_{i+1} = (U_i + Z_i) \bmod 3$$

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

Transition matrix

$U_{i+1} \backslash U_i$	0	1	2
0	0.5	0.5	0.0
1	0.0	0.5	0.5
2	0.5	0.5	0.0

# Examples: Stationary time-invariant Markov processes



**Question:** Can you convert this to an iid sequence?

All the iid compression work still useful!

## $k$ th order Markov source

### Definition: $k$ th order Markov source

A  $k$ th order Markov source is defined by the condition

$$P(U_n | U_{n-1} U_{n-2} \dots) = P(U_n | U_{n-1} U_{n-2} \dots U_{n-k})$$

for every  $n$ . In words, the conditional probability of  $U_n$  given the entire past depends only on the past  $k$  symbols.

Most practical stationary sources can be approximated well with a finite memory  $k$ th order Markov source with higher values of  $k$  typically providing a better approximation (with diminishing returns).

## Non-example

Arrival times for buses at a bus stop:  $U_1, U_2, U_3, U_4, \dots$

4:16 pm, 4:28 pm, 4:46 pm, 5:02 pm

**Question 1:** Is this stationary?

**Question 2:** Can you convert this to a stationary (in fact iid) process?



# Information-theoretic quantities for non-iid random variables

# Conditional entropy

The conditional entropy of  $U$  given  $V$  is defined as

$$H(U|V) \triangleq E \left[ \log \frac{1}{P(U|V)} \right]$$

Can also write this as

$$\begin{aligned} H(U|V) &= \sum_{u \in \mathcal{U}, v \in \mathcal{V}} P(u, v) \log \frac{1}{P(u|v)} \\ &= \sum_{v \in \mathcal{V}} P(v) \sum_{u \in \mathcal{U}} P(u|v) \log \frac{1}{P(u|v)} \\ &= \sum_{v \in \mathcal{V}} P(v) H(U|V = v) \end{aligned}$$

## Properties of conditional entropy

1. Conditioning reduces entropy:  $H(U|V) \leq H(U)$  with equality iff  $U$  and  $V$  are independent.

## Properties of conditional entropy

1. Conditioning reduces entropy:  $H(U|V) \leq H(U)$  with equality iff  $U$  and  $V$  are independent.
2. Chain rule of entropy:

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

## Properties of conditional entropy

1. Conditioning reduces entropy:  $H(U|V) \leq H(U)$  with equality iff  $U$  and  $V$  are independent.
2. Chain rule of entropy:

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

3. Joint entropy vs. sum of entropies:

$$H(U, V) \leq H(U) + H(V)$$

with equality holding iff  $U$  and  $V$  are independent.

# Properties of conditional entropy

1. Conditioning reduces entropy:  $H(U|V) \leq H(U)$  with equality iff  $U$  and  $V$  are independent.

2. Chain rule of entropy:

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

3. Joint entropy vs. sum of entropies:

$$H(U, V) \leq H(U) + H(V)$$

with equality holding iff  $U$  and  $V$  are independent.

Can generalize to conditioning  $U_{n+1}$  on  $(U_1, U_2, \dots, U_n)$ :

$$H(U_{n+1}|U_1, U_2, \dots, U_n)$$

# Entropy rate

Before we look at examples, let's think about how we can generalize entropy for stationary processes. Some desired criteria:

- works for arbitrarily long dependency so  $H(U_{n+1} | U_1, U_2, \dots, U_n)$  for any finite  $n$  won't do
- has *operational* meaning in compression just like entropy
- is well-defined for any stationary process

# Entropy rate

Not only one, but two equivalent ways of defining it!





## Entropy rate

$$H_1(\mathbf{U}) = \lim_{n \rightarrow \infty} H(U_{n+1} | U_1, U_2, \dots, U_n)$$

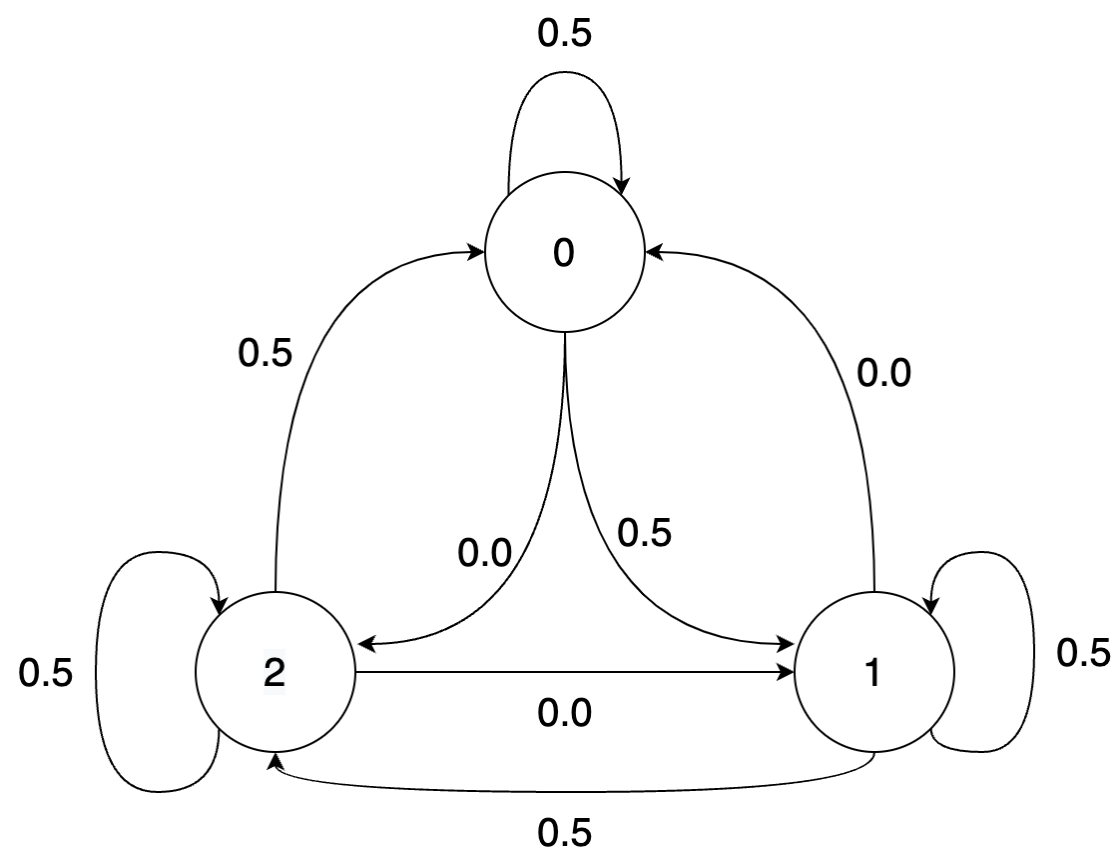
$$H_2(\mathbf{U}) = \lim_{n \rightarrow \infty} \frac{H(U_1, U_2, \dots, U_n)}{n}$$

### C&T Thm 4.2.1

For a stationary stochastic process, the two limits above are equal. We represent the limit as  $H(\mathbf{U})$  (entropy rate of the process, also denoted as  $H(\mathcal{U})$ ).

# Examples

- Fair coin toss
- Markov example



# Example: entropy rate of English text

- Models (estimate probabilities from `text`):

(a) 0th-order Markov chain (iid):

$$H(\mathcal{X}) \approx 4.76 \quad \text{bits per letter}$$

(b) 1st order Markov chain:

$$H(\mathcal{X}) \approx 4.03 \quad \text{bits per letter}$$

(c) 4th order Markov chain:

$$H(\mathcal{X}) \approx 2.8 \quad \text{bits per letter}$$

- Estimate by asking people to guess the next letter until they get it correct. The *order* of their guesses reflects their estimate of the *order* of their conditional probabilities for the next letter. (Shannon 1952).

$$H(\mathcal{X}) \approx 1.3 \quad \text{bits per letter}$$

Source: [http://reeves.ee.duke.edu/information\\_theory/lecture4-Entropy\\_Rates.pdf](http://reeves.ee.duke.edu/information_theory/lecture4-Entropy_Rates.pdf)

## AEP again!

Shannon–McMillan–Breiman theorem

$$-\frac{1}{n} \log_2 P(U_1, U_2, \dots, U_n) \rightarrow H(\mathbf{U}) \text{ a.s.}$$

under technical conditions (ergodicity).

**Takeaway:** entropy rate is the best compression you can hope to achieve.

## How to achieve the entropy rate?

- Today: we start small, try to achieve  $k$ th order entropy  $H(U_{k+1}|U_1, \dots, U_k)$ .
- Next week: achieving entropy rate for arbitrary stationary distributions (in theory) and a really performant scheme (in practice). 🤖

## Working with known 1st order Markov source

Suppose we know  $P(U_2|U_1)$ .

How would you go about compressing a block of length  $n$  using

$$E [\log_2(P(U_1, \dots, U_n))] \approx nH(U_2|U_1)$$

bits?

# 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

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

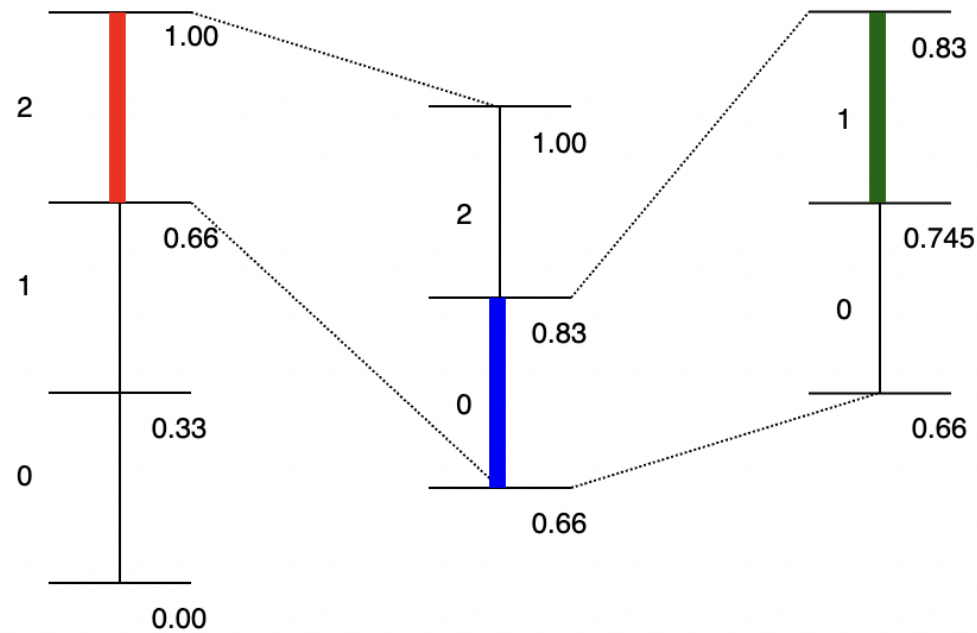
$$U_{i+1} = (U_i + Z_i) \bmod 3$$

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

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

# Working with known 1st order Markov source

Encoding 2, 0, 1

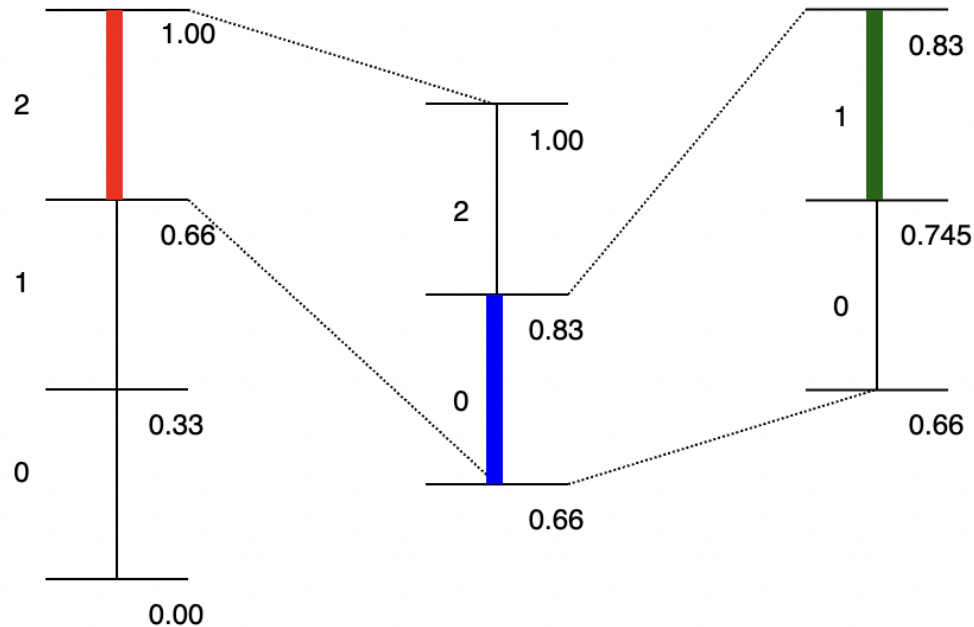


**Question:** Can you explain the general idea?



# Working with known 1st order Markov source

Encoding 2, 0, 1



**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

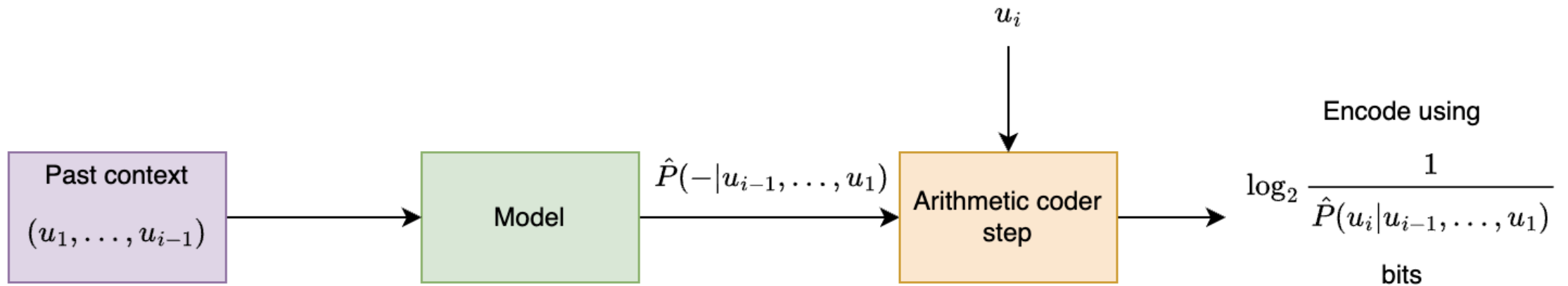
Length of interval after encoding  $u_1, u_2, u_3, \dots, u_n =$   
 $P(u_1)P(u_2|u_1) \dots P(u_n|u_{n-1})$

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

Expected bits per symbol

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

# Context-based arithmetic coding

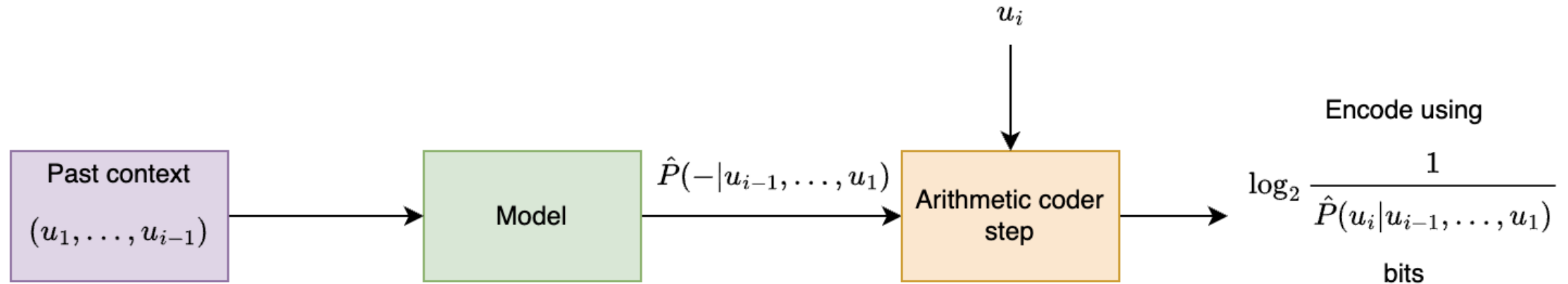


Total bits for encoding:

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

**Question:** How would the decoding work?

# Context-based arithmetic coding



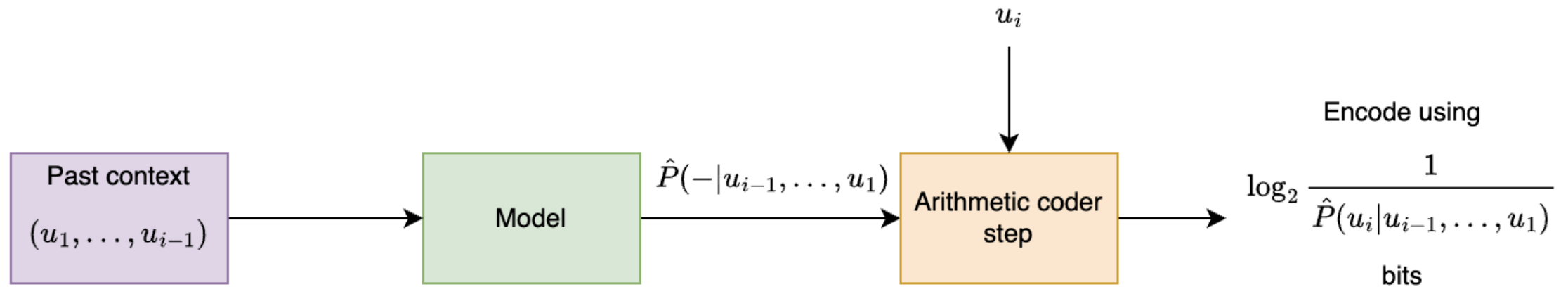
Total bits for encoding:

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

**Question:** How would the decoding work?

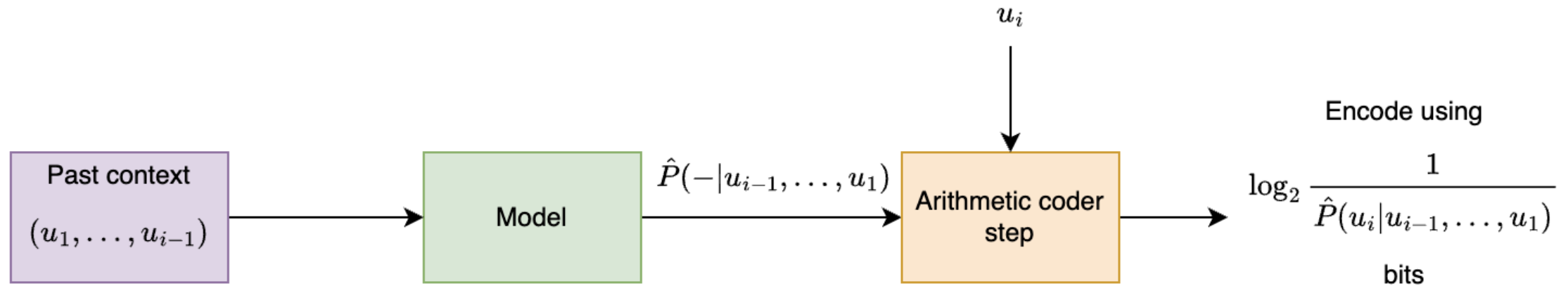
**Answer:** Decoder uses same model, at step  $i$  it has access to  $u_1, \dots, u_{i-1}$  already decoded and so can generate the  $\hat{P}$  for the arithmetic coding step!

## 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

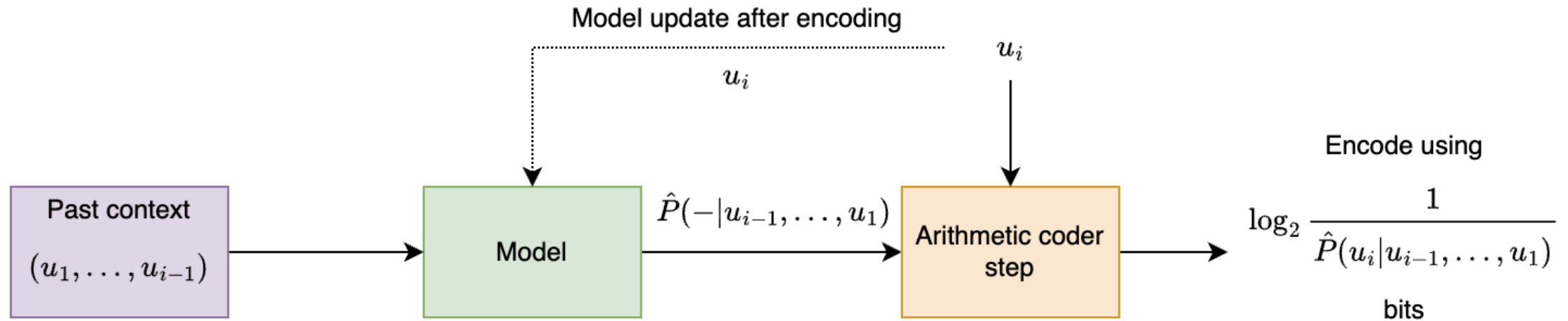
## Two-pass approach

- ✓ 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



- ⚠ 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.



## Compression and prediction

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

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

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

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

# Compression and prediction

- Good prediction  $\Rightarrow$  Good compression
- Compression = having a good model for the data
- Need not always explicitly model the data

# 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

# 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

## $k$ th order adaptive arithmetic coding

```
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]
```

## $k$ th order adaptive arithmetic coding

On `sherlock.txt`:

```
>>> with open("sherlock.txt") as f:
>>>     data = f.read()
>>>
>>> data_block = DataBlock(data)
>>> alphabet = list(data_block.get_alphabet())
>>> model_params = (alphabet, order)
>>> encoder = ArithmeticEncoder(AECPARAMS(), model_params, AdaptiveOrderKFreqModel)
>>> encoded_bitarray = encoder.encode_block(data_block)
```

## $k$ th order adaptive arithmetic coding

Compressor	bits/char
0th order	4.12
1st order	3.34
2nd order	<b>2.85</b>
3rd order	3.09
gzip	2.78
bzip2	<b>2.06</b>



## $k$ th order adaptive arithmetic coding

Compressor	bits/char
0th order	4.12
1st order	3.34
2nd order	<b>2.85</b>
3rd order	3.09
gzip	2.78
bzip2	<b>2.06</b>

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

# $k$ th order adaptive arithmetic coding

## Limitations

- slow, 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

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

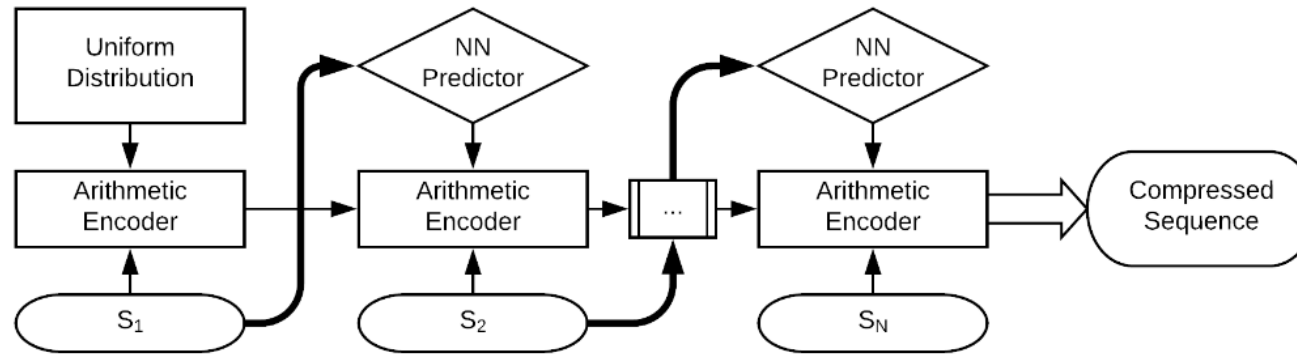
**Note:** Despite their performance limitations, context based models are still employed as the entropy coding stage after suitably preprocessing the data (LZ, BWT, etc.).

# Prediction models used for compression

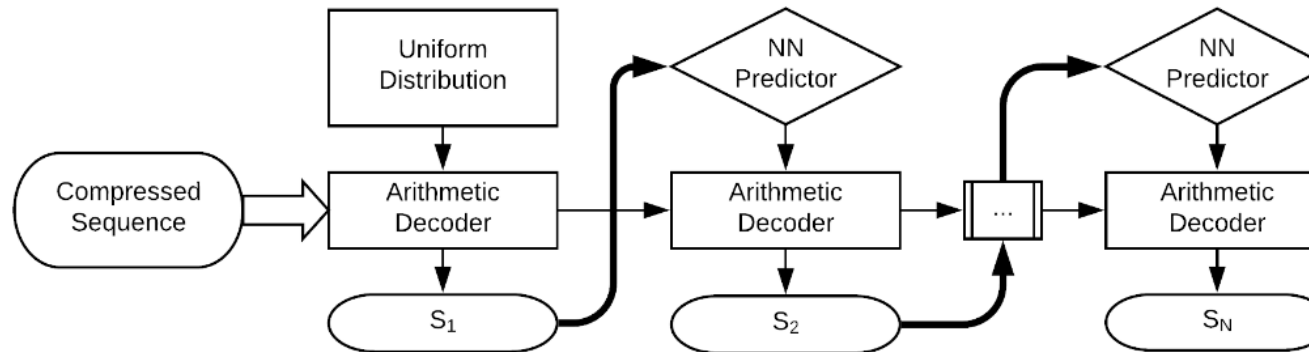
- $k$ th order adaptive (in SCL):  
[https://github.com/kedartatwawadi/stanford\\_compression\\_library/blob/main/compressors/probability\\_models.py](https://github.com/kedartatwawadi/stanford_compression_library/blob/main/compressors/probability_models.py)
- Bit-level models
- Context Tree Weighting (CTW)
- Prediction by Partial Matching (PPM)
- Neural net based: NNCP, Tensorflow-compress, DZip
- Ensemble methods: CMIX

These are some of the most powerful compressors around, but often too slow to use in practice!

# DeepZip framework



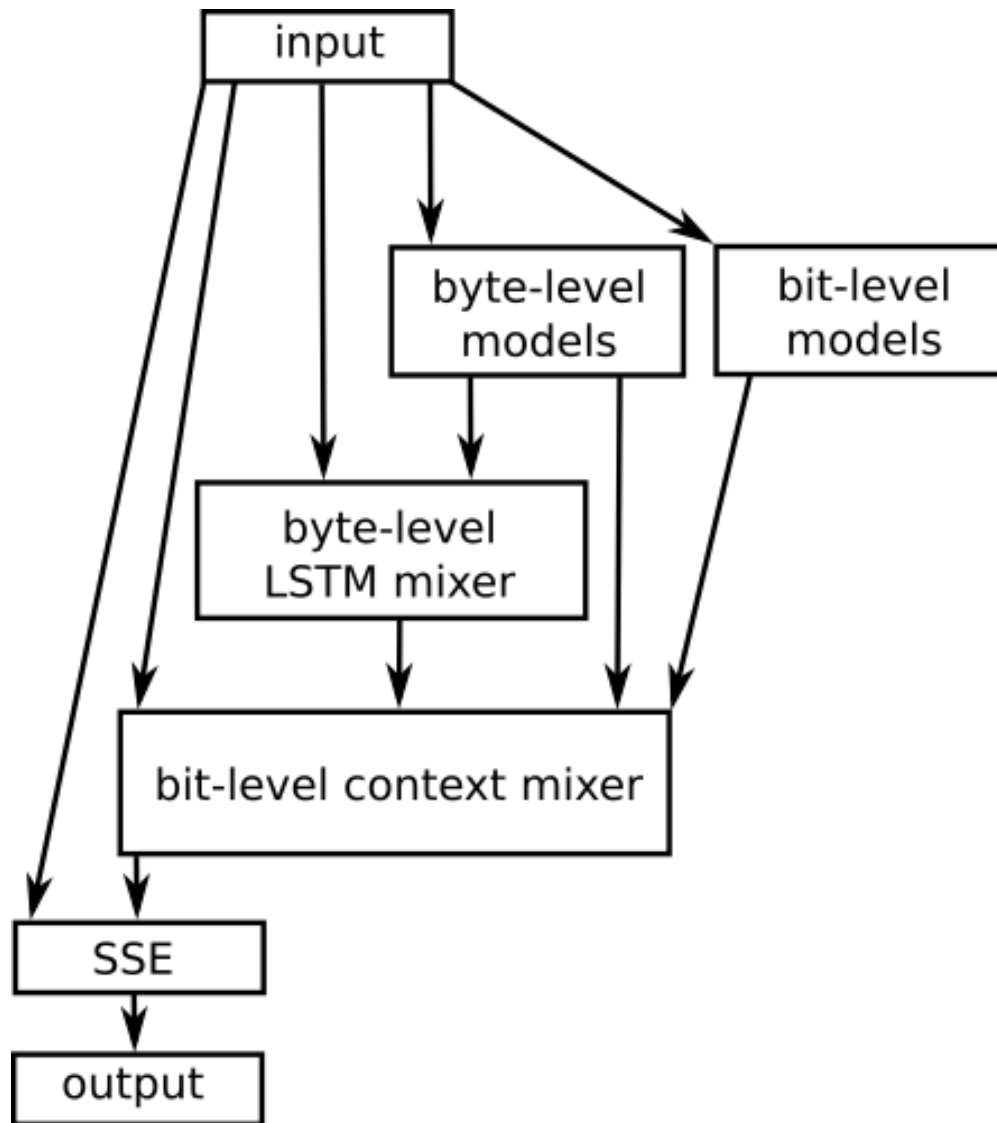
a) Encoder Framework



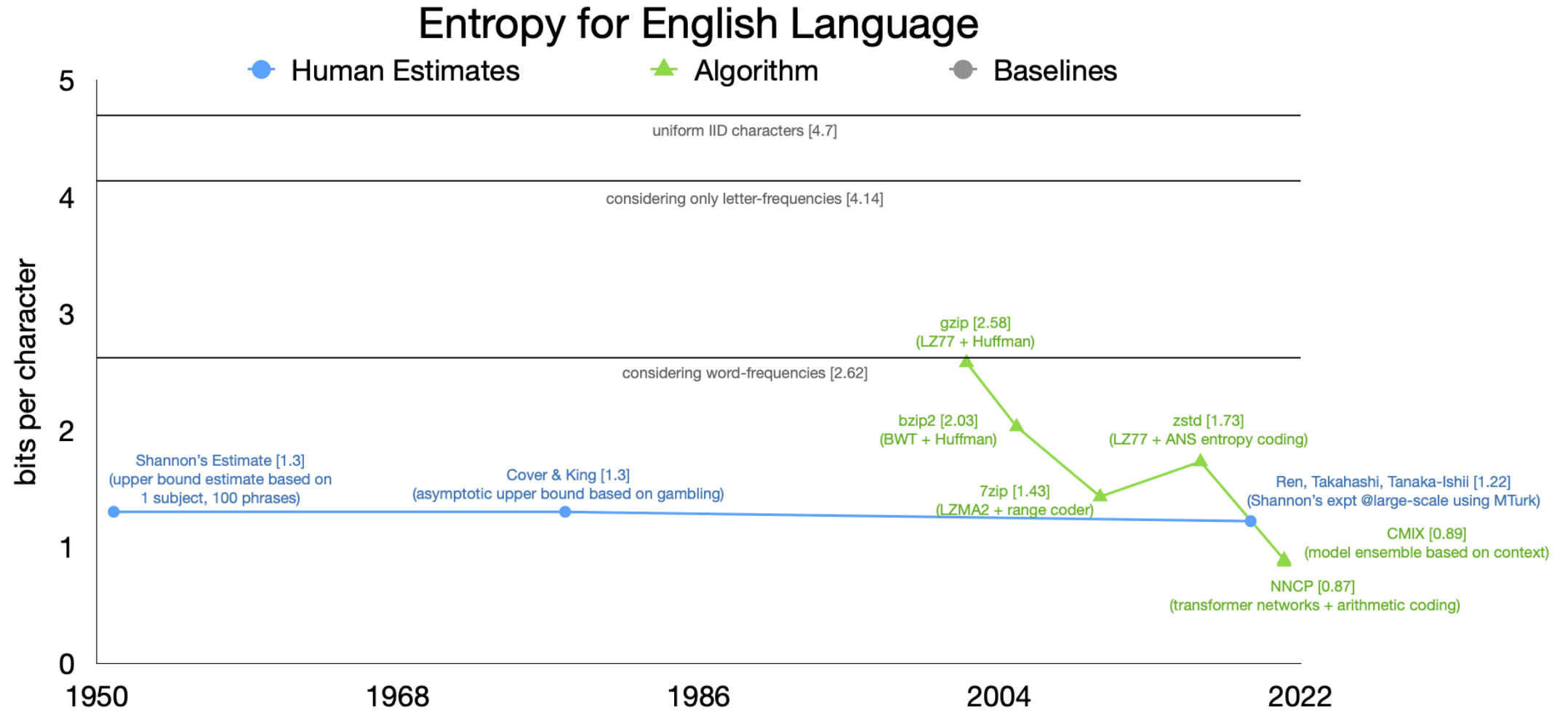
b) Decoder Framework

Figure 1: Encoder-Decoder Framework.

# CMIX context mixing



# Text compression over the years



## Next week

- Lempel-Ziv algorithms - the most widely used algorithms in practice!