The LZW Coding Scheme as an Example of Sequence Prediction by Data Compression

Mathias Winther Madsen

January 28, 2016

Contents

1 Introduction

When an agent walks around in the world, it generates a sequence of data,

$$x_1, x_2, x_3, x_4, \ldots$$

We would like the agent to learn complex behaviors from such a sample path. In particular, we would like the agent to learn how to execute long-term plans whose payoffs lie far into the future.

We've been talking about approaching this problem as a data compression problem: if the agent learns to recognize certain sub-plans, perhaps it can use those as a stepping stone for learning longer and more complex super-plans, and gradually build up a collection of complex, reliable behaviors. This behavioral description corresponds to the epistemic process of picking up on long-distance dependencies in the data in order to make more reliable predictions.

What are our options for defining such a structure-sensitive agent? In this note, I describe one classic compression method that exploits structure in a data strem to achieve compression, and I will indicate how this might be relevant to our learning problem. To clarify our thinking, I also make a point of translating back and forth between data compression and probabilistic inference, explaining why they're a really just reformulations of the same problem.

2 Lempel–Ziv–Welch Coding

Lempel-Ziv-Welch (LZW) coding is a lossless data compression method that reduce a file's size by recognizing repeated patterns. It maps sequences of letters to sequences of integers, which are then encoded as bitstrings. The method was patented in the 1980s, but the patent ran out in 2004.

2.1 General Idea

An LZW encoder reads a text in whole "words," but restricts itself to words it has already learned. It learns new words by concatenating the last word it read with the first character of the next word. The encoder is initialized with a phrasebook which contains at least the entire relevant alphabet.

As an example, suppose an encoder initialized with the phrasebook [T, H, E, A]. It then encounters the text

T, H, A, T, H, A, T, T, H, A, T, H, E, H, A, T, E, ...

On scanning through this text, the encoder goes through the following operations, with the asterisk denoting the current position in the file:

t	Remainging text	Familiar	Unfamiliar
0	*THATHATTHATHEHATE...	T	TH
1	T*HATHATTHATHEHATE...	H	HA
2	TH*ATHATTHATHEHATE...	A	AT
3	THA*THATTHATHEHATE...	TH	THA
4	THATH*ATTHATHEHATE...	AT	ATT
5	THATHAT*THATHEHATE...	THA	THAT
6	THATHATTHA*THEHATE...	TH	THE
7	THATHATTHATH*EHATE...	E	EH
8	THATHATTHATHE*HATE...	HA	HAT
⋮	⋮	⋮	⋮

In each of these steps, the encoder outputs a codeword representing the longest familiar word that can be shaved off the begnning of the file. It also augments its phrasebook with the shortest unfamiliar word is can shave off the beginning of the file. In other words, it reads characters from the left until it encounters an unfamiliar word, and then it learns that new word. When the phrasebook is large, the encoder generally reads faster.

The code for a phrase is its index in the phrasebook. This means that if the size of the phrasebook was t_0 at time $t = 0$, and if a certain phrase was first encountered at time t, then the phrase while be stored on line $t_0 + t$ of the phrasebook. This number will then be the code for that phrase.

Since the phrasebook learns exactly one phrase per time step, the phrasebook also contains $t_0 + t$ phrases at time t. Hence, at time t, the encoder always outputs an integer between 0 and $t_0 + t$, since these are the codewords available.

2.2 Behavior

Returning to the example above, the file

$$\texttt{T}, \texttt{H}, \texttt{A}, \texttt{T}, \texttt{H}, \texttt{A}, \texttt{T}, \texttt{T}, \texttt{H}, \texttt{A}, \texttt{T}, \texttt{H}, \texttt{E}, \texttt{H}, \texttt{A}, \texttt{T}, \texttt{E}, \ldots$$

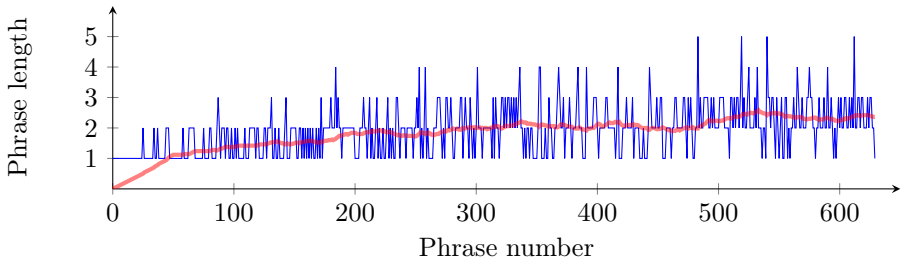
will be encoded as

$$0, 1, 3, 4, 6, 7, 4, 2, 5, \ldots$$

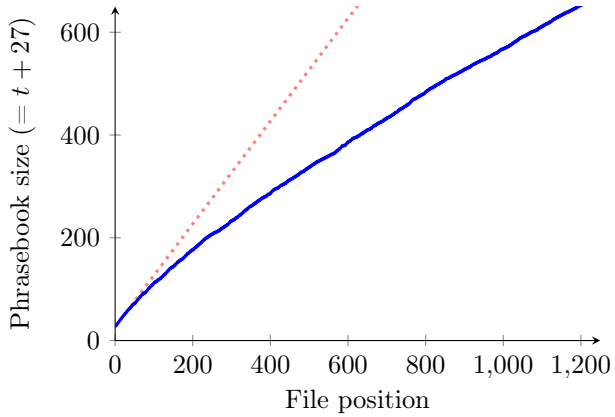
if the initial phrasebook was $[\texttt{T}, \texttt{H}, \texttt{E}, \texttt{A}]$. This corresponds to reading the file as if it were "chunked" into the phrases

$$\texttt{T}, \texttt{H}, \texttt{A}, \texttt{TH}, \texttt{AT}, \texttt{THA}, \texttt{TH}, \texttt{E}, \texttt{HA}, \ldots$$

These phrases tend to get longer as the encoder reads more text and thus learns more phrases. Here is a graph of how the phrase length grows over time as one particular 27-character English text is encoded:



The same information can be visualized by plotting the size of the phrasebook at various points in the file:



This plot shows a mildly decreasing growth rate, indicating that the encoder recognizes more and more phrases as it scans through the file. The encoder thus spends more of its time reading through familiar phrases and less of its time halting because it bumps into an unfamiliar phrase.

2.3 Implementation

In Python, LZW encoding can be implemented as follows:

```
def text_to_integers(text, alphabet):

    # initialize by copying:
    phrasebook = alphabet[:]
    register, tape = "", text

    while tape:

        # while nothing new happens, move forward:
        while tape and (register + tape[0] in phrasebook):
            register, tape = register + tape[0], tape[1:]

        # then encode the contents of the buffer:
        yield phrasebook.index(register)

        # learn a new word:
        if tape:
            phrasebook.append(register + tape[0])

        # flush the buffer:
        register = ""
```

The integer sequences returned by this encoder are uniquely decodable: the decoder can observe its own output and thus learn the same phrases as the encoder, in the same order.

(The only slight complication is that the "lookahead character" at the end of the most recently learned phrase is only revealed one time step into the future. However, by adding incomplete entries to the phrasebook and then retroctively completing them, the decoder can keep up with the encoder just in time to reconstruct the input message.)

The following Python function implements an LZW decoder:

```
def integers_to_text(integers, alphabet):

    # initialize by copying:
    phrasebook = alphabet[:]
    tape = integers[:]

    # decode the first item:
    n = tape.pop(0)
    word = phrasebook[n]
    yield word

    # add an incomplete entry to the phrasebook:
    phrasebook.append(word)

    while tape:

        n = tape.pop(0)

        # if you're already familiar with this phrase:
        if n < len(phrasebook) - 1:
            word = phrasebook[n]
            phrasebook[-1] += word[0]

        # if the phrase is still awaiting completion:
        else:
            phrasebook[n] += phrasebook[n][0]
            word = phrasebook[n]

        # add an incomplete entry to the phrasebook:
        phrasebook.append(word)

        # decode the completed part of it:
        yield word
```

As mentioned above, the integer sequences produced by LZW coding are typically mapped to bitstrings in a uniquely decodable way. The conventional way of doing this is to read and write the bitstream in blocks of $w = \lceil \log(t_0 + t) \rceil$ bits after the end of time step t.

For instance, if we start with a phrasebook of $t_0 = 4$, we should write the bitstream in blocks of sizes

$t_0 + t$	4	5	6	7	8	9	10	$\cdots$
$\lceil \log(t_0 + t) \rceil$	2	3	3	3	3	4	4	$\cdots$

The integer sequence

$$0, 1, 3, 4, 6, 7, 4, 2, 5, \ldots$$

would then be translated into the bitstream

$$\texttt{00 001 011 100 0110 0111 0100 0010 0101} \ \ldots$$

This bitsteam is uniquely decodable (also without the spaces shown here).

The fixed-width approach to integer coding ignores the fact that the tth integer in the sequence can in fact not be larger than $\log(t_0 + t)$. This means that some integers are assigned codewords before they are needed, and this inefficiency can waste up to 1 bit of storage space per integer. On the other hand, it is computationally cost-free.

3 LZW as a Random Process

Having now described how LZW coding works, we might want to think about how we could use it, or use a variant of it, to learn patterns from data. However, before we start hacking away, I think it might be a good idea to take a step back and think in a systematic way about the modeling assumptions that a particular encoding scheme is an expression of. This will hopefully give us larger flexibility in terms of how we define new learning methods.

3.1 Codes are Probability Distributions

Any uniquely decodable coding scheme implicitly corresponds to a probabilistic assumption about the world: a code that spends $k(x)$ bits on the event x is the optimal scheme for an environment in which x has point probability

$$p(x) \ = \ 2^{-k(x)}.$$

Reversely, suppose an environment described by the point probabilities $p(X)$. The optimal code for that environment then has codeword lengths

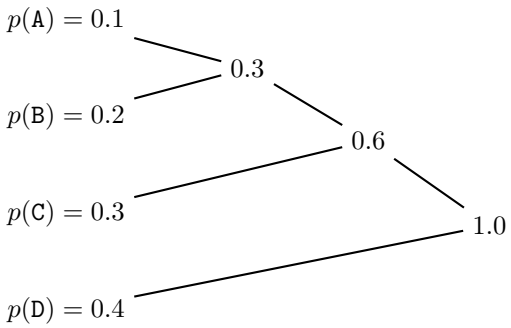
$$k(x) \ = \ \log \frac{1}{p(x)}.$$

Any lossless scheme that achieves high compression of a data stream does so because its implicit assumptions are good approximations to the truth. Consider for instance the following probability distribution:

x	A	B	C	D
$p(x)$	.1	.2	.3	.4

We can design a good code for this distribution by the Huffman algorithm,

which builds a binary tree bottom-up by repeatedly combining the two lowest-probability events:



If we label the branches of this tree with `0`s and `1`s (in an arbitrary way), then the resulting tree defines a code k for the sample space $\{\texttt{A}, \texttt{B}, \texttt{C}, \texttt{D}\}$. That code, in turn, defines an implicit probability distribution $\hat{p} \approx p$:

x	A	B	C	D
$k(x)$	000	001	01	1
$\hat{p}(x)$	1/8	1/8	1/4	1/2

Since $\hat{p} \neq p$, the code is not perfectly adapted to its environment. Instead of consuming the minimum of

$$0.1 \log \frac{1}{0.1} + 0.2 \log \frac{1}{0.2} + 0.3 \log \frac{1}{0.3} + 0.4 \log \frac{1}{0.4} \;\approx\; 1.85$$

bits per character during encoding, it consumes

$$0.1 \log 8 + 0.2 \log 8 + 0.3 \log 4 + 0.4 \log 2 \;=\; 1.9$$

bits per character.

This slight inefficiency would also be visible from the output it produces: for instance, the probability of drawing a `0` if we pluck a random bit out of the output of this encoder is

$$\frac{1}{8} \times 3 + \frac{1}{8} \times 2 + \frac{1}{4} \times 1 + \frac{1}{2} \times 0 \;=\; \frac{5}{8}.$$

Since $5/8 \neq 1/2$, this shows that there is still compressible structure left over even after we encode a stream of characters with this code. If we designed a better code (by buffering more than one character before choosing our output), then the output stream would look more like a unbiased and thus unstructured coin flipping sequence.

We can also get a sense of the "world" that the code thinks it lives in by decoding a completely random coin flipping sequence. For instance, the bitstream

$$\texttt{001 000 001 01 1 01 1 01 1 01 1 1 1 000 000 1 } \ldots$$

(with spaces inserted for readability) is back-translated by k to

$$\texttt{B, A, B, C, D, C, D, C, D, C, D, D, D, B, B, D, } \ldots$$

This stream thus represents a sequence of samples from $\hat{p}$.

3.2 What Kind of World Does LZW Live In?

To get a sense of the probabilitistic assumptions that LZW coding makes about its environment, we can feed randomly selected integers into its decoder and observe the decoded output.

Consider an LZW encoder initialize to contain the 26 upper-case English letters as well as space. This decoder implicitly expresses a probabilistic assumption about the possible data streams that it might see. In a convoluted way, it defines a probability distribution P over all characters sequences.

We can sample from this distribution by choosing a random sequence from the output stream and feeding this into the LZW decoder. Since our encoder is initialized with an alphabet of 27 characters, we must choose our fake encoded sequence by sampling integers

$$\begin{array}{lcl} Y_1 & \sim & \text{Uniform}\{0, 1, 2, \ldots, 25, 26\} \\ Y_2 & \sim & \text{Uniform}\{0, 1, 2, \ldots, 25, 26, 27\} \\ Y_3 & \sim & \text{Uniform}\{0, 1, 2, \ldots, 25, 26, 27, 28\} \\ Y_4 & \sim & \text{Uniform}\{0, 1, 2, \ldots, 25, 26, 27, 28, 29\} \\ \vdots & \vdots & \vdots \end{array}$$

A typical sample from this process is

$$Y \;=\; 2, 14, 3, 18, 9, 16, 19, 2, 30, 1, 15, 26, 15, 23, 11, 6, 34, 18, 35, 22, 13, 3, 33, \ldots$$

Feeding this integer sequence into the LZW decoder, we get the following decoded output:

```
CODSJQTCSJBP PXLGCSSSJBWNDTCTCSJBPXSSWSSCTCSSCSSWTCTBSDSSW
TCSSBSSSWZLGXSDDTCSSWQTSCSJBPGCNYPXSCSSWDSBISSXLFWNSDSSDDS
SCFW XLSCSXGSSCQTSCTUFW SDSSDUSSTCSJSCSSWDQTSCSSWTSSXVSSCQ
DTCFQTSCUSXLSSWTSSJSSTTCSFQSJPXQTSCUSSUTCNCTQTSCSXLSSJBPCS
GCNCQTSCSDDSJPSDSSDSUZDTSDUJQHKPCTQTCSSZDGSNYSSCTCSSSSTCSS
DDSQTSCSXTCNSJSCQDSUSDDSWNSTCSNCPXSSTGCNTCSSZCSSWDQEHKCSSH
KZDCT PXSBSSCNYSKPSCNSJBPNYSKXLJQDSBYPSCEPCTCSFTCSSZTCPXSC
SSHDTCFKSDDSWSSTRXLSJSCCQYGCNSJPSWNZDCTCNHKXLSSXSCT SJ ...
```

This sequence is a typical sample from the random process that the LZW coding scheme is adapted to. It's not like English or any other language, but it contains many repeated patterns that show a language-like behavior.

By latching onto its own random output as if it were actual data, the LZW can thus teach itself a fictional "language," and the sequence above gives us a sense of what kind of structure it expects to find in that language. The samples we can produce this way are drawn from a random process P which models the LZW encoders beliefs about the world.

3.3 What Can LZW Learn?

Another way of probing the assumptions of the LZW encoder is to first feed it a sequence of actual data generated from English text (so that it learns a bit of English), and then observe what kind of structure it learned to recognize. If the assumptions encoded in P are a good reflection of the truth about the random process Q which (hypothetically) generated the English input, then P should quickly "learn" the structure of Q, in the sense that

$$P(X_{t+1} \,|\, X_1, X_2, \ldots, X_t) \;\approx\; Q(X_{t+1} \,|\, X_1, X_2, \ldots, X_t)$$

after a reasonable amount of data $X_1, X_2, \ldots, X_t \sim Q$ has been fed into P.

To get a sense of how close P gets to this goal, we can feed it a sequence of integers which is partly encoded English, and partly random. After decoding, this gives a sample of the following form:

```
ON GLANCING OVER MY NOTES OF THE SEVENTY ODD CASES IN WHICH
I HAVE DURING THE LAST EIGHT YEARS STUDIED THE METHODS OF
MY FRIEND SHERLOCK HOLMES I FIND MANY TRAGIC SOME COMIC A
LARGE NUMBER MERELY STRAGHBEDDFR GLM DRLALM INGS OERRLAG HB
WANYTRXSTHBEDLO HLM IS SSTHET WAMY LM IRLAG MHAHS TES O DRED
HLE ER NENTER HL WAMED EES AGHAH OFDDLYS S HLE FIR OCSEED
TRILYIC CA IIC AS O GLLM ILAINGSKONODSBEDS O SES O EE
NOVSHUMUD FIRINDCAGLWHS O OCBELSTHERLAG HO GLL HLEUMUT RE
LIC ASTHE ESRLS IHCHNT FIRID OFD HLERLALAR ME SCKIN RIE S
LI EOFVE TESURTHEOLSTHERYAGDDL OFDSTU OFLYSOD HO OLHELAA S
OFD ERRSTHERYETTHEICURTED FRIEOCSOCSAGDDDLTHE EH SHURT ES
TBEDS SSHAH CHLS HO OUES ABEDSYENUOCSOLSES FVE OFD TRO DSF
MVEN W LETTTHEURARTHIC ALM IRLAG
```

Again, this is not English, but it's closer than before. The randomly generated output stream now contains many more spaces, more actual word segments, and a higher proportion of sequences that are frequent in actual English.

3.4 An LZW Predictor

Consider an agent which wants to predict what will happen one time step into the future. How could such an agent use the LZW coding scheme to make predictions?

As explained above, the LZW code implicitly expresses certain probabilitistic beliefs about the world, expressed in terms of a random process, and we can sample from that random process. We can also sample from the conditional (or posterior) versions of this process so as to assess the probability of various continuations of a text.

Consider for instance the 200-character text snippet

```
ON GLANCING OVER MY NOTES OF THE SEVENTY ODD CASES IN WHICH
I HAVE DURING THE LAST EIGHT YEARS STUDIED THE METHODS OF
MY FRIEND SHERLOCK HOLMES I FIND MANY TRAGIC SOME COMIC A
LARGE NUMBER MERELY STRA ...
```

that is, the observation

$$X_1 = \texttt{O},\ X_2 = \texttt{N},\ \ldots,\ X_{200} = \texttt{A}.$$

How might this text continue? That is, what might X_{201} be?

In order to answer that question, we need to infer what state we left the LZW encoder in after it encodeed the last complete phrase it could recognize. By running the encoding method until it bumps into the string boundary, we can find that this happens at the point indicated by the asterisk:

```
...  A LARGE NUMBER MERELY ST*RA
```

At this point, the encoder still has to decide how to parse the two letters `RA` and whatever mystery sequence that might follow. By sampling a series of phrases that are consistent with this short segment, we can reconstruct the probability distribution over possible continuations, like `STRAR`, `STRANY`, `STRAV`, `STRAS`, etc. By counting how often the next character is $\texttt{A}, \texttt{B}, \texttt{C}, \ldots$, we can compile the following table of frequencies:

N	_	R	S	G	V	E	I	O	T	H	···
.15	.14	.10	.08	.07	.05	.04	.03	.03	.03	.02	···

This is not unreasonable from an informed English-language perspective, since `STRANGE` is a strong canidate for a continuation. An agent which models the world in terms of the random process defined by the LZW decoder can thus come to make quite well-informed decisions given reasonable amounts of data, at least one time step into the future.

4 Perspectives

LZW coding is a universal coding scheme in the sense that will eventually replicate the n-gram frequencies of a stationary random process for any arbitrary but fixed n. However, it makes highly specific assumptions about its world, and these assumptions influence which patterns that it's biased to learn the fastest.

We may not want to make the same assumptions. Some dubious properties of the LZW encoder are, for instance:

1. it only reuses a phrase when that exact same phrase has already been attested in the exact form, that is, it has a zero-tolerance policy for devations from the phrasebook;

2. it only knows right-branching grammatical structures, in the sense that it can grows the phrases from its phrasebook by adding to them from the right;

3. it scans the string in a greedy fashion and does not take future coding costs into account;

4. its phrasebook has no internal structure, and it cannot make any predictions based on the fact that it has found itself in similar but different conditions previously.

All of these assumptions can be challenged and changed. In particular, we might want to think about how to build an LZW-inspired encoder for a Hidden Markov Model, since this will both allow richer grammatical structure and take the noisiness of the agent's information into account.