FAST LEMPEL-ZIV (LZ'78) COMPLEXITY ESTIMATION USING CODEBOOK HASHING

Harman Jot, Rupinder Kaur
M.Tech, Department of Electronics and Communication,
Punjabi University, Patiala, Punjab, India

I. INTRODUCTION

Compression is useful as it helps reduce resources usage, such as storage space or transmission capacity. But we must decompress the compressed data before use. This overhead processing imposes extra computational costs. For instance, a compression scheme for video may require expensive hardware for the video to be decompressed fast enough to be viewed as it is being decompressed, [1] and the option to decompress the video in full before watching it may be inconvenient or require additional storage. The design of data compression schemes involves trade-offs among various factors, including the degree of compression, the amount of distortion introduced and the computational resources required to compress and un-compress the data.[2] An order over some alphabet typically exhibits some regularities, what is essential to think of compression. For distinctive English texts we can spot that the most regular letters are e, t, a, and the least regular letters are q, z. We can also discover such words as the, of, to regularly. Often also longer remains of the text reappearance, probably even the whole sentences. We can use these stuffs in some way, and the succeeding sections elaborate this matter.

II. SOURCE CODING

Source coding or data compression is a process of efficiently converting the output of either an analog or digital source into a sequence of binary digits. Most data shows patterns and is subject to certain constraints. This is true for text, as well as for images, sound and video. [3]

III. DICTIONARY CODING

Dictionary coding techniques rely upon the [4] observation that there are correlations between parts of data (recurring patterns). The basic idea is to replace those repetitions by (shorter) references to a "dictionary" containing the original.[5]

1.3.1 Static Dictionary

The simplest forms of dictionary coding use a static dictionary. Such a dictionary may contain frequently occurring phrases of arbitrary length, di-grams (two-letter combinations) or n-grams. This kind of dictionary can easily be built upon an existing coding such as ASCII by using previously unused codewords or extending the length of the codewords to accommodate the dictionary entries [4].

1.3.2 Semi-Adaptive Dictionary

The aforementioned problems can be avoided by using a semi-adaptive encoder. This class of encoders creates a dictionary custom-tailored for the message to be compressed. Unfortunately, this makes it necessary to transmit/store the dictionary together with the data. [4]

1.3.3 Adaptive Dictionary

The ***Lempel Ziv*** algorithms belong to this third category of dictionary coders. The dictionary is being built in a single pass, while at the same time also encoding the data. As we will see, it is not necessary to explicitly transmit/store the dictionary because the decoder can build up the dictionary in the same way as the encoder while decompressing the data.[6]

IV. COMPRESSION ALGORITHMS

This segment develops a methodical building of binary codes compressing the data of a foundation

- Shannon-Fano Algorithm
- Huffman Algorithm
- The Lempel-Ziv Algorithm

1.4.1 Huffman Algorithm

This algorithm, invented in 1952 by D.A. Huffman, delivers a prefix code whose building can be achieved by a binary tree. Here are the successive steps:[7]

First step

We assemble the source symbols on a row in order of increasing likelihood from left to right.

Second step

Let us denote A and B the two basis symbols of lowest likelihoods P_A and P_B in the list of the source words. We association A and B together with two subdivisions into a node which substitutes A and B with probability task equal to $P_A + P_B$. A and B are removed from the list and substituted by the node. [8]

Third step

We apply the process of the second step until the likelihood assignment is equal to 1. Then, the conforming node is the root of the binary tree.

1.4.2 The Lempel-Ziv Algorithm

1.4.2.1 Principle

The LZ'78 is a dictionary-based compression algorithm that maintains an explicit dictionary. The codewords output by the algorithm consist of two elements: an index referring to the longest matching dictionary entry and the first non-matching symbol.[9][10]

In addition to outputting the codeword for storage/transmission, the algorithm also adds the index and symbol pair to the dictionary. When a symbol that not yet in the dictionary is encountered, the codeword has the index value 0 and it is added to the dictionary as well. With this method, the algorithm gradually builds up a dictionary.[11]

1.4.2.2 Algorithm

```
w := NIL;
while (there is input)
{
        K := next symbol from input;
        if (wK exists in the dictionary)
        {
                w := wK;
        }
        else
        {
                output (index(w), K);
                add wK to the dictionary;
                w := NIL;
        }
}
```

This simplified pseudo-code version of the algorithm does not prevent the dictionary from growing forever. There are various solutions to limit dictionary size, the easiest being to stop adding entries and continue like a static dictionary coder or to throw the dictionary away and start from scratch after a certain number of entries has been reached. [11]

V. IMPLEMENTATION

Despite the problems discussed above, LZ78 is among the more available universal complexity estimators. However, complexity estimation using LZ78 usually quantities to performing the entire compression procedure and comparing inverse density ratios as a measure of complexity. In fact, the simple Lempel Ziv partition covers enough data to estimate complexity without execution the entire compression encoding procedure. Central to the LZ78 algorithm is the partitioning scheme familiarized by Ziv and Lempel. The LZ78 algorithm partitions a string into prefixes that it hasn't seen before, forming a codebook that will (given a long adequate string with enough repetition) enable long strings to be encoded with small indexes. Consider an example to illustrate how this algorithm works: LZ partitioning of the string is,

101101001001101001001111010010011000 10

Performed by injecting commas each time a sub-string that has not yet been recognized is seen. The following partition results: [12]

1,0,11,01,00,10,011,010,0100,111,01001,001,100,010

The nodes noticeable in black of the five level tree are nodes confined in the LZ78 partition of the example string. Nodes that are not occupied in designate code words or phrases that are not controlled in the LZ78 partition. Each node or phrase happens precisely once in the string with the exclusion of the last phrase which may be a recurrence of a beforehand seen node. Good compression (low complexity estimation) outcomes when the LZ78 partition contains a deep, sparse tree, while poor compression (high complexity estimation) results from strings that are less deep and extra completely occupied at each level Maximum compression of LZ78 is attained if all code words are children of the same branch, for example, the string:

1101011011101101011001011000 partitioned as 1,10,101,1011,10110,101100,1011000 will be extremely compressed by LZ78. However, the following string will not be compressed by LZ78.

10101101001001011101110000001= 1,0,10,11,01,00,100, 101, 110, 111, 000, 001

Since the presentation of LZ78 will be determined by the partition, by absorbed exclusively on the tree partition features of the algorithm we can attain better efficiency when using LZ78 to approximation complexity. The vital metric is the number of phrases in the partition. The minimum number of sub-strings (commas) in an LZ78 partition is the number M such that each sub string is one bit longer than the previous sub string[12]:

$$\sum_{m=1}^{M} m = L = \frac{M(M-1)}{2} = \frac{M^2}{2} = \frac{M}{2}$$

Solving this quadratic calculation and taking the positive solution for M we have:

$$M^2 + M - 2L = 0 \Longrightarrow M = \frac{-1 + \sqrt{1 + 8L}}{2}$$

For strings of any considerable length, the constant terms become insignificant and a good estimation of the lower bound results from ignoring the preservative constant terms:

$$M \geq \sqrt{2L}$$

Since we know the negligible number of phrases a string of length L can have, we can standardize the number of phrases in the LZ78 partition based on this minimum for use in crucial a normalized complexity estimator. We define a the metric C as an estimator of complexity using the LZ78 partition given a string of length L bits and an LZ78 partition of M phrases:

$$C = \frac{M}{\sqrt{2L}}$$

This metric allows use of the LZ78 partitioning algorithm to estimate complexity, regularized by length, providing an estimator similar to compression ratio, but without the necessity for the overhead to actually complete the LZ78 compression.[12]

VI. RESULTS

Simulations were done using MATLAB. The following menu is generated using MATLAB.

```
--------------------------------
####### LZ'78 Algorithm #######
--------------------------------
[1] Encode user-defined message.
[2] Encode pre-defined msg #1.
[3] Encode pre-defined msg #2.
[4] Encode pre-defined msg #3.
[5] Decode a sequence.
[6] Exit.
--------------------------------
################################
--------------------------------
Enter your option:
```

The user needs to select an option to continue. The first option will encode a user-defined message using LZ'78 encoding. The fifth option decodes an already encoded sequence. The second, third and fourth options encode pre-defined messages of length 1 lakh, 2 lakh and 3 lakh respectively. These options are required as it is very difficult to enter large sequences of data for testing. The last option (sixth) is used to terminate the Matlab program.

The MATLAB following program illustrates the encoding of a user-defined message – "Hello World!"

```
--------------------------------
####### LZ'78 Algorithm #######
--------------------------------
[1] Encode user-defined message.
[2] Encode pre-defined msg #1.
[3] Encode pre-defined msg #2.
[4] Encode pre-defined msg #3.
[5] Decode a sequence.
[6] Exit.
--------------------------------
################################
--------------------------------
Enter your option: 1
--------------------------------
################################
--------------------------------
```

```
Enter the message to encode: Hello World!
Do you want to use Hashing? (Y/N) [Y]: Y
Do you want to find Complexity? (Y/N) [Y]: Y

Text Message (12 characters):-
Hello World!

Binary Message (96 characters):-
010010000110010101101100011011000110111100100000
010101110110111101110010011011000110010000100001

Encoded Message (137 characters):-
000101010001100101011100111000110010011111010010
101011010111001010010000001001001100100111011100
101010001000101001001011101111100000111011
```

```
Total Phrases in the Code Book = 29

Time taken to Encode = 00:00:00.049 (0.049489 seconds)

Length of input Message (Total characters) = 12
Length of Message after binary conversion = 96
Length of Message after LZ'78 Encoding = 137

Binary Message Complexity = 16
Binary Message Normalised Complexity = 1.0975

Encoded Message Complexity = 21
Encoded Message Normalised Complexity = 1.088

Compression Ratio -> 142.71 %
```

You can see above that we used hashing as the option "Do you want to use Hashing? (Y/N) [Y]:" was set to "Y" or Yes. The message "Hello World!" consists of twelve characters (including SPACE) which were first converted to binary using UTF8 encoding and then the binary sequence was encoded using LZ'78 (explained earlier) using hashing. Code book size was 29. This means there were in total 29 unique phrases in the message to be encoded. Time taken for encoding was 0.049489seconds.

Compression ratio is 142.71%. Ideally, we want this ratio to be less than 100%. Less ratio implies that the encoded sequence length is less than the original message length (in binary) and that the data is compressed.

$$\text{Compression Ratio} = \frac{Length\ of Encoded\ Binary\ Message}{Length\ of\ Original\ Binary\ Message} \times 100\ \%$$

Similarly, the below program will decode the above generated encoded sequence to get the original message back.

```
--------------------------------
####### LZ'78 Algorithm #######
--------------------------------
[1] Encode user-defined message.
```

```
[2] Encode pre-defined msg #1.
[3] Encode pre-defined msg #2.
[4] Encode pre-defined msg #3.
[5] Decode a sequence.
[6] Exit.
--------------------------------
################################
--------------------------------
Enter your option: 5
--------------------------------
################################
--------------------------------

Enter the sequence to decode:
00010101000110010101110011100011001001111101001010
10101101011100101001000000100100110010011101110010
10101000100010100100101110111110000111011

Decoded Message :-
Hello World!

Time taken to Decode = 00:00:00.022 (0.021591 seconds)

Length of Encoded Sequence = 137
Length of Decoded Message = 12
```

Similarly, the same process was repeated for pre-defined messages and the following results were obtained.

Table 1: Compression Ratio comparison for different message length.

Input Binary Message Length	Encoded Message Length	Code Book Entries	Compression Ratio
801,096	611,609	39,833	76.35 %
1,602,200	1,170,977	72,337	73.09 %
2,402,448	1,706,459	102,086	71.03 %

Table 2: Encoding time comparison with and without Hashing.

Input Message Length (Binary Message)	Time taken to Encode	
	With Hashing	Without Hashing
801,096	0.4150 seconds	45.5558 seconds
1,602,200	1.0525 seconds	157.4849 seconds
2,402,448	1.3753 seconds	319.1074 seconds

We can see from the results obtained in table 1, larger the input message length more is the compression ratio. Also the number of code book entries increase with input message length. With such a huge amount of entries it is virtually impossible for any system to perform a real-time search.

Form the results obtained in table 2, we can see that with increase in the number of codebook entries the encoding time increases from 45 seconds to nearly 320 seconds without hashing. But as explained earlier about fastness of hashing, we can see that the encoding time remains almost unchanged with increase in number of codebook entries. It increases but slowly as compared to the one without hashing. If we can encode this fast with LZ'78, this implies we can find the codebook size swiftly, which in turn is the complexity estimation of a string as explained earlier. Hence, our results.

REFERENCES

[1] C. E. Shannon, "A mathematical theory of communication," *ACM SIGMOBILE Mobile Computing and Communications Review*, vol. 5, no. 1, pp. 3-55, 1948.

[2] M. E. Hellman, "An extension of the Shannon theory approach to cryptography," *Information Theory, IEEE Transactions on*, vol. 23, no. 3, pp. 289-294, 1977.

[3] R. N. Williams, "An extremely fast Ziv-Lempel data compression algorithm," in *Data Compression Conference, 1991. DCC'91.*, 1991.

[4] K. Sayood, Introduction to data compression, Newnes, 2012.

[5] T. Jacob and R. K. Bansal, "On the optimality of Sliding Window Lempel-Ziv algorithm with side information," in *Information Theory and Its Applications, 2008. ISITA 2008. International Symposium on*, 2008.

[6] T. C. Bell, J. G. Cleary and I. H. Witten, Text compression, vol. 348, Prentice Hall Englewood Cliffs, 1990.

[7] D. Kirovski and Z. Landau, "Generalized Lempel--Ziv compression for audio," *Audio, Speech, and Language Processing, IEEE Transactions on*, vol. 15, no. 2, pp. 509-518, 2007.

[8] M. Malyutov, "Recovery of sparse active inputs in general systems: a review," in *Computational Technologies in Electrical and Electronics Engineering (SIBIRCON), 2010 IEEE Region 8 International Conference on*, 2010.

[9] J. Ziv and A. Lempel, "A universal algorithm for sequential data compression," *IEEE Transactions on information theory*, vol. 23, no. 3, pp. 337-343, 1977.

[10] S. Wadhwani, A. Wadhwani, S. Gupta and V. Kumar, "Detection of bearing failure in rotating machine using Adaptive Neuro-fuzzy inference system," in *Power Electronics, Drives and Energy Systems, 2006. PEDES'06. International Conference on*, 2006.

[11] J. Ziv and N. Merhav, "A measure of relative entropy between individual sequences with application to universal classification," *Information Theory, IEEE Transactions on*, vol. 39, no. 4, pp. 1270-1279, 1993.

[12] J. H. G. S. S.C. Evans, "Kolmogorov Complexity Estimation and Analysis," *Information and Decision Technologies,* no. 1, pp. 1-6, October 2002.