Information Theory and Data Compression thanks to the Lempel-Ziv Code

1 Introduction

Lempel and Ziv have invented two data compression codes, often referred to as LZ77 and LZ78 because of their publication dates [1, 2]. Here, we only discuss LZ78 which we simply call "Lempel-Ziv code".

The Lempel-Ziv code is a code that offers the possibility to compress a source. Its advantages are:

- The compression is universal, that means the probability distribution of the source symbols does not need to be known in advance.
- Asymptotically, the code is optimal, *i.e.*, the number of bits tends to the entropy.
- The algorithm is easy to implement.

2 Principle

The source sequence (or string) is divided into minimal substrings which did not occur before. For example, `1011010100010` is divided into $\mathtt{1}, \mathtt{0}, \mathtt{11}, \mathtt{01}, \mathtt{010}, \mathtt{00}, \mathtt{10}$. Each time one wants to extract a substring, one has to take into account the previous substrings and needs a sufficient amount of bits in order not to recreate a substring that has already occurred before. For `010` in our example `0` exists already, `01` as well, and `010` is the minimal substring that has not occurred before. We remark that each substring has as prefix a previous substring — if this was not the case, the extracted substring would not be minimal.

Once the sequence is divided, the Lempel-Ziv code transforms each substring in a couple that contains:

- the position of the longest previous substring which is the prefix of the current substring and
- the added character that makes the current substring unique.

For instance, the substrings $\mathtt{1}, \mathtt{0}, \mathtt{11}, \mathtt{01}, \mathtt{010}, \mathtt{00}, \mathtt{10}$ are encoded as $(0, \mathtt{1})$, $(0, \mathtt{0})$, $(1, \mathtt{1})$, $(2, \mathtt{1})$, $(4, \mathtt{0})$, $(2, \mathtt{0})$, $(1, \mathtt{0})$. By convention the substrings are numbered from left to right, starting by 1. The substring with position 0 is the empty substring.

In other words, the Lempel-Ziv code transforms a sequence of elements of an alphabet $\mathcal{A}$ in a new sequence of elements belonging to $\mathbf{Z} \times \mathcal{A}$. In the example mentioned above the alphabet is binary.

As discussed before, the number of the position also needs to be encoded (here in binary or more generally in an alphabet $\mathcal{A}$). The longer the source sequence is, the more bits we need to encode the position. Fortunately, one can verify that asymptotically the number of bits per symbol tends to the entropy of the source.

We remark finally, that the fundamental disadvantage of Lempel-Ziv is the demand for a possibly big memory in order to stock all the past substrings. Asymptotically, this memory is not bounded.

3 Implementation

In order to encode with Lempel-Ziv, one needs to pre-code as described in the following. One assumes to have a dictionary that contains all past substrings, where the entries are enumerated.

1. One initializes the dictionary by putting the empty substring at position 0.
2. One extracts a substring of $n + 1$ bits[1] of the sequence that one wants to encode. This substring has to be the shortest substring that is not present in the dictionary, where the first n bits correspond to an existing entry of the dictionary.
3. At the output one generates a code that represents the number of the entry and the $(n + 1)$-th bit of the substring, which was obtained in step 2.
4. One adds the extracted substring to the dictionary. The number of the entry corresponds to the first empty entry[2].
5. One repeats steps 2 to 4 until the entire source sequence is encoded.

We remark:

- In order to properly treat the end of the sequence, we add a special character at the end.
- The decoding can be directly deduced from the encoding. It suffices if the decoder updates his dictionary in the same way as the encoder.

[1] We speak for simplicity of bits. In general, it is easy to generalize the method for another alphabet.
[2] We remark that here the number of the entry in the dictionary is equal to the number of appearance of the substring, in other words the position of the substring

In our example with the sequence `1011010100010`, the algorithm works as follows:

Substring	Output	Dictionary
		`dico[0] = ""`
`1`	$(0,1)$	`dico[1] = "1"`
`0`	$(0,0)$	`dico[2] = "0"`
`11`	$(1,1)$	`dico[3] = "11"`
`01`	$(2,1)$	`dico[4] = "01"`
`010`	$(4,0)$	`dico[5] = "010"`
`00`	$(2,0)$	`dico[6] = "00"`
`10`	$(1,0)$	`dico[7] = "10"`

4 In theory...

For detailed theoretical explanations, it may help to consult [3]. We state here the final result, which proves the asymptotic optimality of the Lempel-Ziv code. For a stationary and ergodic source $\{X_i\}_{-\infty}^{+\infty}$ one can prove that

$$\limsup_{n\to\infty} \frac{1}{n} l(X_1, X_2, \dots X_n) = H(\mathcal{X}) \quad \text{with probability 1,} \tag{1}$$

where $l(X_1, X_2, \dots X_n)$ is the length of a codeword of the Lempel-Ziv code associated to $X_1 \dots X_n$ where $H(\mathcal{X})$ is the entropy of the source.

5 In practice...

In practice, the variant LZW thanks to Welch [4] is often applied. A reference is for instance `compress` under Unix, or the image format GIF. For more details, one can consult [5].

References

[1] J. Ziv A. Lempel. A universal algorithm for sequential data compression. *IEEE Transaction on Information Theory*, 23(3), 1977.

[2] J. Ziv A. Lempel. Compression of individual sequences via variable-rate coding. *IEEE Transaction on Information Theory*, 24(5), 1978.

[3] Thomas M. Cover and Joy A. Thomas. *Elements of Information Theory.* Wiley, 1991.

[4] Terry Welch. A technique for high-performance data compression. *IEEE Computer*, 17(6), 1984.

[5] Mark Nelson *The Data Compression Book.* M&T Publishing, 1992.