Example of LZ76 Estimation of Complexity

Glen Langdon

The LZ76 algorithm appears in the following paper, followed by the paper’s abstract:

Abstract – A new approach to the problem of evaluating the complexity (“randomness”) of finite sequences is presented. The proposed complexity measure is related to the number of steps in a self-delimiting production process by which a given sequence is presumed to be generated. It is further related to the number of distinct substrings and the rate of their occurrence along the sequence. The derived properties of the proposed measure are discussed and motivated in conjunction with other well-established criteria.

Since the senior author for the 1976 paper is Lempel, we denote the algorithm in this paper as LZ76. For the papers written by the same co-authors in 1977 (sliding window) and 1978 (parsing algorithm), the senior author was Ziv. Thus, the two later papers are also denoted as ZL77 and ZL78. Nelson follows the tradition of calling the respective algorithms LZ77 and LZ78.

What is “randomness”? The LZ76 paper explores the randomness of sequences. One expects a high degree of randomness results from a zero-order Markov process where the symbols found in the finite sequence $S$ are equally likely. The opposite case is a high degree of predictability, such as a two-symbol alphabet with one symbol (1) having probability 1 and the other symbol (0) with probability 0. A finite sequence $S$ of all 1s is highly predictable (the absence of randomness).

The problem addressed in the LZ76 paper concerns a measure of complexity for data sequences.
- **Basic idea**: parse the test sequence into phrases of symbols from a given alphabet $\mathcal{A}$.

- A phrase is a null symbol, an individual symbol, or a sequence of symbols of $\mathcal{A}$.

- The parsing algorithm proceeds as follows:

  1. *The dictionary of phrases is initialized to the null symbol as the only member.* The null symbol phrase has index 0. Each next phrase added to the dictionary is assigned the next higher index from that of the previously added phrase.

  2. Phrases are built recursively, adding each next symbol from the data sequence $\mathcal{S}$ to a subsequence. The growing subsequence is compared with phrases in the dictionary. It is the next symbol following the longest match found in the dictionary that creates the new phrase. The last symbol of the new phrase is the innovative symbol.

  3. The recursive search stops when the subsequence just built forms a new phrase.

  4. The first phrase so recognized is assigned index 1, since the initial dictionary already contains the null phrase as index 0.

  5. The first phrase of the sequence will have the null phrase of index 0, and the first symbol of $\mathcal{S}$ as the innovative symbol.

  6. The phrase corresponding to index 1 is the single symbol that begins sequence $\mathcal{S}$.

  7. *After creating each new phrase for the dictionary, that new phrase is added to the dictionary and assigned the next available index.*

  8. The phrase recognition recursion inserts an imaginary comma after the recognized phrase, and begins again the phrase recognition phase with the next unread symbol of $\mathcal{S}$.

At first, the number of commas per symbol is typically one comma per symbol, an early indication the sequence is random. *However*, if there are intersymbol dependencies in the sequence, then certain symbols will have a high probability of following certain given symbols. Another valid
notion of *an intersymbol dependency* is that certain symbols will have a low probability of following certain given symbols.

The most “random” sequence, in the sense of the LZ76 complexity measure, would be a sequence of symbols generated by a zero-order information source of alphabet \( \mathcal{A} \) with the *uniform* distribution. *Think about it:* all sequences of same length are equally likely with the uniform distribution. Everything is balanced and there are no popular symbols and no unpopular symbols.

Given a zero-order uniform distribution, the LZ76 phrase recognition and comma-insertion algorithm most likely grows all possible single-symbol phrases. Next, we see most of all possible two-symbol phrases created. Then, we see all three-symbol phrases generated, etc, because there is no “skewed” distribution generating a subset of relatively long sequences comprised mostly of popular symbols. The longer sequences occur from the occurrence of more popular symbols, while the relatively short sequences occur with mostly less probable symbols.

If the information source is a higher-order source with conditioning states that have various popular symbols, then certain symbols will follow certain given substrings with high probability. It is not uncommon in text files that, given a known three- or four-symbol higher-order state, that a majority of symbols never follow that state. Higher-order sources generate sequences that are less random, and thus should have a lower randomness metric. Whatever the type of source, be it high order or zero order with uniform distribution, the metric of “how many commas per symbol” starts to decrease and converge to a value that represents the rate of occurrence of distinct substrings. Each parsed phrase gets a “comma” and the next available Index in Dictionary \( \mathcal{D} \), and each phrase is a distinct substring. The rate is calculated on a per-symbol basis: the average number of commas (phrases) for each symbol seen.

The LZ76 algorithm operates on sequence \( \mathcal{S} \) of arbitrarily large length, since the results are asymptotic as the length of \( \mathcal{S} \) tends to infinity.

Instead, we shall operate on a finite sequence \( \mathcal{S} \) of length \( N \), where \( N \) a finite number equal to or less than a googol. The googol equals \( 10^{100} \). The value \( 10^{100} \) exceeds the number of particles in the known universe. In the following example, \( N \) is 25. We denote sequence \( \mathcal{S}_1^N \) as the sequence
(typical sequence symbol located in symbol $\text{Position } i$ is $s_i$) as beginning with symbol $s_1$, and ending with symbol $s_N$.

Example

Consider the following example sequence of 25 symbols:

\[ a \ b \ c \ a \ c \ b \ c \ a \ b \ c \ a \ c \ b \ c \ a \ b \ c \ a \ b \]

\[ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7\ 8\ 9\ 0\ 1\ 2\ 3\ 4\ 5\]

The following table is an example of the phrase-building or “production” algorithm. The example presents the so-called self-delimiting production process mentioned in the Abstract. The self-delimiting process inserts a comma (“,”) into sequence $S$ after the creation of each new phrase formed by the concatenation of the longest recognized dictionary phrase and the innovative symbol that follows.

\[ a, \ b, \ c, \ ac, \ bc, \ ab, \ ca, \ bca, \ bcab, \ cab, \ cabc, \]

Relative to the 25-symbol example sequence above, the comma-insertion process has the effect of rewriting the sequence $S$ as follows. For the first three symbols, the recognized phrase is the null phrase of Index 0. Phrases $a$, $b$, and $c$ have respective Indices 1, 2, and 3 in $D$. After treating the first three symbols, the next Position in $S$ is 4, and the next available Index is 4, for which the variable $Rate$ takes on value 1.000. Note that the current Position is the number of symbols so far seen, and Index of the last Dictionary entry is the number of phrases formed so far.

In the following Table, the column headed by $Rate$ has a computed entry at the the creation of each new phrase entered into the dictionary and assigned the next Index. The $Rate$ is based on the metric defined in the LZ76 paper, and is computed by the following Equation:

\[
Rate = \frac{\text{Index}}{\text{Position}}
\]

Note that the rate starts at 1.000 symbols per phrase, and after 25 symbols, has decreased to the rate of 0.440 symbols per phrase.
Step-by-Step Creation of Phrases per LZ76

<table>
<thead>
<tr>
<th>Position</th>
<th>Symbol</th>
<th>Add to Dictionary</th>
<th>Index</th>
<th>Recognized as</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a</td>
<td>a</td>
<td>1</td>
<td>null, a</td>
<td>1.000</td>
</tr>
<tr>
<td>2</td>
<td>b</td>
<td>b</td>
<td>2</td>
<td>null, b</td>
<td>1.000</td>
</tr>
<tr>
<td>3</td>
<td>c</td>
<td>c</td>
<td>3</td>
<td>null, c</td>
<td>1.000</td>
</tr>
<tr>
<td>4</td>
<td>a</td>
<td></td>
<td>4</td>
<td>1,c</td>
<td>0.800</td>
</tr>
<tr>
<td>5</td>
<td>c</td>
<td>ac</td>
<td>5</td>
<td>2,c</td>
<td>0.714</td>
</tr>
<tr>
<td>6</td>
<td>b</td>
<td>bc</td>
<td>6</td>
<td>1,b</td>
<td>0.667</td>
</tr>
<tr>
<td>7</td>
<td>c</td>
<td></td>
<td>7</td>
<td>3,a</td>
<td>0.636</td>
</tr>
<tr>
<td>8</td>
<td>a</td>
<td>ca</td>
<td>8</td>
<td>5,a</td>
<td>0.571</td>
</tr>
<tr>
<td>9</td>
<td>b</td>
<td></td>
<td>9</td>
<td>8,b</td>
<td>0.500</td>
</tr>
<tr>
<td>10</td>
<td>c</td>
<td></td>
<td>10</td>
<td>7,b</td>
<td>0.476</td>
</tr>
<tr>
<td>11</td>
<td>a</td>
<td></td>
<td>11</td>
<td>21,c</td>
<td>0.440</td>
</tr>
</tbody>
</table>

A Parse Tree Form for the Dictionary

We consider the null phrase the root of a tree used for parsing phrases. Following each innovative symbol, a leaf bearing the value of that innovative symbol is created. When that innovative symbol (now a leaf) becomes
the last symbol of the recognized phrase, that leaf acquires a child, and becomes an *interior node* of the parse tree.

In the example just described, symbols a, b, and c, are the first three symbols have been seen. At this point, the parse tree consists of the root, and three leaves. The phrase ac is formed next, and a becomes an interior node, and the symbol c is a child node of a, as well as a leaf of the parse tree.

![Parse Tree Diagram]

S = abcabcabcabcabcabcabcabc

**Characterization of the LZ76 Algorithm**

The recursion variables needed in order to implement the LZ76 parsing algorithm, and compute the requisite metric denoted *rate of occurrence of distinct substrings* are:

1. **Dictionary** D: a data structure initialized to the null phrase.

2. A vector of phrase indices, D_k, one phrase index for each unique phrase (distinct substring).

3. Variable **Tot_Symbols** to accumulate the number of symbols seen (to compute the randomness metric).
4. Variable \texttt{Tot\_Phrases} (distinct substrings) to accumulate the number of phrases seen (to compute the randomness metric).

The operations needed by the parsing algorithm are:

1. Search the unprocessed tail of sequence $S$ for the longest matching phrase.
   - If \textit{no match}, it must be an unseen symbol to be encoded as the null phrase (of index 0) followed by the unseen symbol itself.
   - If \textit{match}, encode the index of the longest recognized phrase, followed by the next symbol.

2. In either of the previous two cases, update the Dictionary with the new phrase formed by concatenating the innovative (next) symbol to the end of the longest recognized phrase in the dictionary.

3. Increment \texttt{Tot\_Symbols} by the number of symbols seen this iteration (for computing the randomness metric).

4. Increment the variable \texttt{Tot\_Phrases} (for computing the randomness metric).

Nelson begins Chapter 8, \textit{LZ78 Compression}, as follows: "The LZ77 algorithms have a deficiency that must have troubled their creators, Jacob Ziv and Abraham Lempel. These algorithms use only a small window into the previously seen text, which means they continuously throw away valuable dictionary entries because they slide out of the dictionary".\footnote{The original paper for LZ77 actually provides an asymptotic result whereby the buffer capacity has no size limit.}

Moreover, Nelson set a maximum length of the "match" for his LZ77 algorithm at just 17 bytes. In the actual LZ77 paper, the length of the match was unlimited.

Nelson's point is found under the heading \textbf{Enter LZ78}: "To effectively sidestep these problems, Ziv and Lempel developed a different form of dictionary-based compression."

In reality, the cited shortcomings of LZ77 did not exist at the time LZ78 was written, since there were no published implementations of LZ77 prior to the publication of LZ78. Also, LZ78 is not a completely new algorithm, since LZ78 employs the main parsing algorithm of LZ76. The
major contribution of LZ78, relative to practical compression algorithms,\(^2\) is coding the phrases from the dictionary under the uniform distribution.

**Relationship of LZ76 to LZ78 (ZL78)**

The book by Nelson, *The Data Compression Book*, M&T Books, describes a pseudo-code routine for LZ78 (see page 280) that is actually based on LZ76. From Nelson’s description of LZ78, it is easy to see that the operations on the symbol sequence \(S\) employed by LZ76 to parse phrases and insert commas are exactly the operations Nelson employs on page 280 of his book.

**Nelson pseudo-code**

```plaintext
for (; ; ) {
    current_match = 1;  /* In case find_match returns "no match at all" */
    current_length = 0;  /* Same reason as above comment */
    memset( test_string, '\0', MAX_STRING );  /* use for "getc" next chars */
    /* Above "memset" places a null ("\0") into first MAX_STRING positions */
    /* The "real" LZ76 has no upper bound on the phrase length */
    for (; ; ) {
        test_string[ current_length++ ] = getc( input );  /*read next symbol*/
        new_match = find_match ( test_string );  /* returns -1 if fails */
        if ( new_match == -1 )
            break;  /* Exit from "infinite" for loop */
        current_match = new_match;  /* longest phrase recognized */
    }
    output_code( current_match );  /* Write the index of longest phrase */
    output_char( test_string[ current_length - 1 ] );  /* Write innovative */
    add_string_to_dictionary( test_string );  /* Function adds the new */
    /* phrase composed of recognized phrase and innovative symbol to */
    /* the dictionary */
}
```

The pseudo-code function called `find_match(test_string)` performs the parsing algorithm itself. Prior to the execution of `find_match`, value `current_match` is initialized to 1, and `current_length` to 0.

If the outcome of `find_match(test)` is no match, then the returned variable `new_match` is set to -1. The return of -1 indicates the first symbol that was encountered in `test_string` has never been seen before. In this case, the same action as LZ76 is taken: output a null (index number 1) followed by the new character as the innovative symbol.

Following the parsing of the longest phrase from `test_string`, (in this

\(^2\)Theoretical results are also present in [LZ78].
case the null string), the next character in \texttt{test\_string} is the innovative character and the encoder needs to output the next phrase/character pair. The code (index) for the longest phrase is output by instruction \texttt{output\_code( current\_match)}, and then followed by the next symbol in \texttt{test\_string} when instruction \texttt{output\_char(test\_string[current\_length - 1])}.

The Encoding algorithm for LZ78 is easily derived from LZ76 as follows:

1. Retain the parsing algorithm for LZ76.
2. Phrase located at index 0 is the null phrase, 1 is the next index number, which will be assigned to the first symbol of sequence $S$.
3. Following each step that recognises the longest phrase:
   
   (a) Encode the index of that longest recognized phrase.
   (b) Concatenate the symbol value (it has a fixed number of bits)
   (c) Assign the next available index number to the newly formed phrase (result of the concatenation).
   (d) Add the new phrase to the dictionary in the position of its index number.
   (e) Increment the index number of the previous dictionary entry to obtain the next index number.
4. The recognized phrase has an index that is encoded (Note 1).
5. Accumulate the total number of phrases contained in the dictionary (Note 1).
6. Assume each phrase in the dictionary is equally likely (Note 1).
7. Encode the index number of the phrase followed by the symbol itself (Note 1).

\textbf{Note 1}

The accumulated total number of phrases is a binary number with a leading (most significant) 1-bit somewhere. For example, after 32 phrases are in the dictionary, we have a 5-bit number: $2^5 = 10000$. At this
time, all indices become 5-bit numbers. We need 6-bit indices when the phrases total 64: $2^6 = 64$ (decimal) = 100000 (binary). At this point, all phrases are coded by the 6-bit version of their index number. This process continues. After the encoder crosses another power-of-two threshold, the encoded phrase indices all become one bit longer. If a symbol whose index is a decimal 3 (11 binary), its index becomes 000011 which is 6 bits. Clearly the probability model is the uniform distribution: clearly at any point in time, the applicable codeword lengths for all phrases have the same length in bits, and all possible innovative symbols also have the same length in bits.

**Concluding Remarks**

We have shown the strong relationship between the parsing model of the paper LZ76, and the algorithm described in the paper LZ78. A major contribution of LZ78 was the employment of the parsing model into a compression algorithm and the theoretical results on the asymptotic properties of the algorithm.

**References:**
