Learning Macros with an Enhanced LZ78 Algorithm

Forrest Elliott and **Manfred Huber**
Department of Computer Science and Engineering
The University of Texas at Arlington, Arlington, Texas 76019-0015
{elliott, huber}@cse.uta.edu

Abstract

One application of the Lempel-Ziv LZ78 algorithm, other than compression, is learning repeating sequences in a data stream One shortcoming of the algorithm though is its slow learning rate. In this paper we enhance the algorithm for improved performance from a learning perspective and apply it to the learning of user macros in a computer desktop environment. Once a macro is learned it can be predicted and offered back at opportune times. With the enhanced algorithm, it is possible for a macro to be learned in as few as two exposures to a sequence.

Introduction

The term commonly associated with automating user actions is creating a "macro" for those actions. Just as one would operate a tape recorder, a macro is recorded and then played back. Consider a scenario where the recorder is never shut off and an algorithm is put in place to learn the user's macro and the macro's start and end points autonomously. The type of algorithm needed to accomplish this goal is one capable of learning sequences of actions. The Lempel-Ziv LZ78 algorithm (Ziv and Lempel 1978), with its dictionary tree, is such an algorithm.

Once the sequence and its demarcation points are learned, the macro can be offered back to the user when the start point is encountered. The actions in the macro can be displayed to the user as a series of small pictures in a manner similar to a video editing application. For example, the list of actions to print a document might look like: "file → print → number of copies → 5 → ok". The user may then play the offered macro.

This paper presents an approach to learning macros without resorting to a brute force search of the user's action history. A brute force search is capable of finding any two matching sequence exposures but the overhead of such a search is high. In particular, the overhead complexity is at least proportional to the length of the user's history. With the LZ78 algorithm, the brute force search through the entire history that has to be performed after every user action is replaced with a dictionary sub tree search of only the last few user actions.

A significant problem when using the LZ78 algorithm though is that the algorithm learns slowly. Many exposures of a sequence are required for a sequence to be represented in the algorithm's dictionary. Typically, the minimum number of sequence exposures required to learn a macro is proportional to the length of the macro. This paper proposes a learning rate enhancement to the LZ78 algorithm. With the enhancement, a repeating sequence can be learned in as few as two exposures to the sequence.

The approach in this paper draws upon preexisting methods and extends them by introducing a learning rate enhancement to the LZ78 algorithm and applying the modified version to learning macros. Furthermore, it develops a novel technique to autonomously learn start and end points for the macros.

Related Work

A landmark contribution to sequential prediction models was provided by M. Feder (Feder et al. 1992). Feder showed that predictability can be described in terms of compressibility. In particular he shows that the Lempel-Ziv LZ78 incremental parsing algorithm in effect becomes a sequence predictor in the long term.

The LZ78 algorithm has been extended many times and in various ways by others. One contribution is the "LeZi-update" method by A. Bhattacharya (Bhattacharya and Das 2002). The LeZi-update method attempts to solve the limitations of location-based mobility tracking by designing a path-based mobility tracking system which learns user location paths. In this variation the algorithm was modified so that a trie graph is formed whereby all low Markov "orders" are represented in the graph. An important precept in this variant is that the Lempel-Ziv algorithm can be applied to a variety of technology areas. Especially important are those areas that are able to benefit from learning.

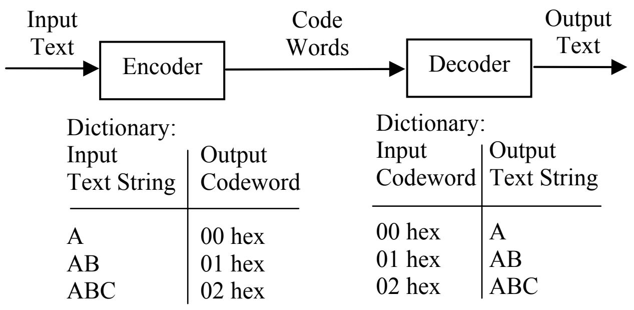
Another LZ78 variation is the "Active LeZi" algorithm by K. Gopalratnam (Gopalratnam and Cook 2003). This contribution is an enhancement to Bhattacharya's LeZi-update method. The enhancement is to limit the depth of the trie graph to the length of the longest phrase seen with classical LZ78 parsing. In this way trie graphs may be constructed with fewer nodes without loss of predictive accuracy.

Another form of sequence prediction is learning by example for the purpose of imitation. P. Sandanayake

(Sandanayake and Cook 2002) presents a system where an imitating agent learns a controlling agent's actions in the Wumpus World game. In his work the interaction of a player agent with the Wumpus World game is monitored. The monitoring allows the agent's play strategy to be extracted. Once extracted, the agent's policy can then be compared to the policy learned by the monitoring algorithm. This imitation and comparison is the same sort of comparison used in the "Turing Test" (Russell and Norvig 1995). The monitoring algorithm should be able to mimic the game agent. Likewise, when learning macros, macros offered to the user should mimic the user.

Lempel – Ziv Algorithm

The Lempel-Ziv 1978 (LZ78) algorithm is a lossless, adaptive dictionary, compression scheme. The technique exactly reproduces the original data after encoding and decoding. In an adaptive dictionary compression scheme, text is translated into dictionary entries and vice versa on the fly.



Dictionary:

Input Text String	Output Codeword
A	00 hex
AB	01 hex
ABC	02 hex

Dictionary:

Input Codeword	Output Text String
00 hex	A
01 hex	AB
02 hex	ABC

Figure 1. LZ78 Dictionary Compression Scheme.

Example

Consider the input text sequence: "AAB". Initially there are no dictionary entries in the encoder. When the first character "A" is input to the encoder the empty dictionary causes the first dictionary entry: A → 00 hex. The character "A" is forwarded to the decoder as a code word. When the next character "A" is input to the encoder the existing dictionary entry is found and 00 hex is marked as the current context. When the next character "B" is input, the dictionary entry AB → 01 hex is created. Then 00 hex and the character B, the first non matching symbol, are forwarded as code words to the decoder. Repeating this basic process for other sequences, a series of code words will take on the following repeating doublet format:

(dictionary index) (non matching char), (…)(…), …

Consider an input text string of "AABABC". The encoder's output sequence becomes:

(null) (A), (00 hex) (B), (01 hex) (C)

Encoder parsing is often shown on example input strings for notational convenience. For this example string the encode parsing is: A, AB, ABC. The code words and dictionary entries for the string "AABABC" are shown in Figure 1. By inverting the process it can be seen that a corresponding dictionary can be constructed at the decoder. Each codeword received at the decoder causes a new dictionary entry on its end.

If the LZ78 dictionary is viewed as a tree the graph of the example sequence would be as shown in Figure 2.

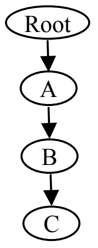


Figure 2. LZ78 Dictionary for Sequence "AABABC".

Another example of an LZ78 dictionary tree is the one shown in Figure 3. This tree is formed when encoding the sequence "ABCABBCCD".

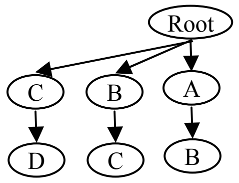


Figure 3. LZ78 Dictionary for Sequence "ABCABBCCD".

Enhanced Lempel – Ziv

One disadvantage of the LZ78 algorithm, and of any finite-context method, is that it converges slowly when used for prediction. The LZ78 algorithm is a good sequence predictor only after having been exposed to a long input sample.

Three Enhancement Rules

The LZ78 enhancement presented here is a learning rate enhancement. Although LZ78 is a greedy algorithm for learning, the changes presented here increase the greed characteristic even more. Basically, the goal is to minimize the number of occurrences required to learn a repeated sequence.

Recall the construction of the dictionary tree as input text is revealed. A tree branch is followed until a non-matching character occurs. At this time a new leaf is appended to the end of the current branch or context. As stipulated by the LZ78 algorithm, the current context pointer is reset to the root when the leaf is added. It is this resetting the context pointer to the root which causes a loss of sequence or context information. To address the issue, the following augmenting rules are proposed.

1. Add a "next node" pointer that represents the next character in the original sequence.

2. When about to add a new leaf to an existing branch – check the next node pointer. If the character that the next node pointer points to matches the input character, then duplicate the next node where the new leaf would normally be added.
3. When a duplicate leaf is appended (as outlined in 2), let the current context node pointer continue on from the duplicate leaf instead of resetting the pointer to the root.

As a result of rule three, and given the right situation, the enhanced LZ78 algorithm will add a leaf node without resetting the current context pointer to the root. The context information is not lost and, with the duplication of nodes provided by rule two, the learning rate is improved.

Next Node Pointer

Recall the LZ78 dictionary tree structure. Each pointer from parent node to child node represents exactly one piece of context information. In the sequence "ABC", for example, one piece of context information would be that when A occurs, B follows. Another piece would be that when B occurs, C follows. The LZ78 dictionary tree representation is a pointer from node A to node B and a pointer from node B to node C.

From a context perspective, the worst case tree construction scenario is when context information is discarded. This situation always occurs when a particular input symbol is occurring for the first time. An example is the sequence "ABC" and is shown in Figure 4. Each added node causes the current context pointer to reset to the root. When the pointer is reset, symbol sequence information (the context) is lost.

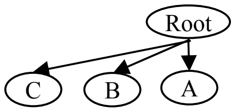


Figure 4. Loss of Context Information.

On the other hand, no loss of context information occurs when the character sequence AB is already in the dictionary tree, the current context is B, and character C is received next. In this case character C is merely appended onto character B as shown in Figure 2 (page 2). This is the best case scenario for dictionary tree construction as the context of C is retained.

The "next node pointer" concept is to create a situation whereby the dictionary tree construction can progress from Figure 4 to Figure 2 on the second exposure to an identical sequence. That is, when given the sequence "ABCABC" the dictionary tree branch of Figure 2 results.

Consider Figure 4 again but with the addition of next node pointers. Next node pointers represent the sequence ordering in the original string. One next node pointer is from A to B and another next node pointer is from B to C. The sequence information within ABC is lost in Figure 4 and is completely intact in Figure 5.

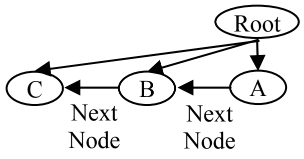


Figure 5. Next Node Pointers for Sequence "ABC".

Next Node Duplication

Now consider what happens when a second ABC sequence arrives. When the second A arrives ("ABCA"), as in the standard LZ78 case, the current context pointer will be set to point to node A. But when the second B arrives ("ABCAB"), the second new rule is followed. By examining the next node pointer from A to B we see that the sequence "AB" is occurring for a second time and thus node B can be duplicated as a leaf of A. The tree of "ABCAB", using next node duplication, is shown in Figure 6.

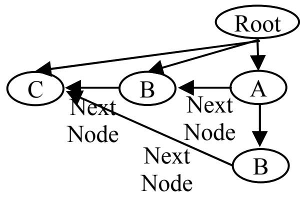


Figure 6. Sequence "ABCAB".

It is important to note that with the addition of B above the root child node B was *duplicated* and appended to A rather than creating an entirely new B node and appending it onto A. This difference is important because the duplicated B node maintains the next node information about what sequence has occurred in the past. Namely, in the past node C occurred after B.

So far (excluding next node pointers) the tree construction for this example text is no different then what would be generated using the normal LZ78 algorithm. A difference will be seen when the next and final symbol is revealed.

Continue When Duplicating Nodes

For the third rule consider the dictionary of Figure 6. In the normal LZ78 algorithm the added new node of B would cause the current context pointer to reset to the root. But this is not necessary here. The next node pointer has accurately identified that sequence "AB" has been seen twice. Therefore the current context pointer is set to the duplicated node B (instead of the root) as outlined in rule three.

When given the final C in the sequence "ABCABC" node duplication occurs again as per rule two. The root child node C is duplicated and appended to node B. The resulting tree is shown in Figure 7.

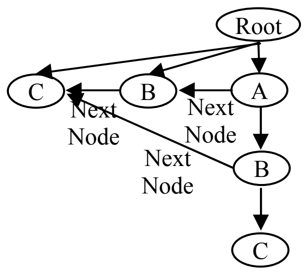


Figure 7. Sequence "ABCABC".

A tree branch similar to Figure 2 has been constructed on the second occurrence of sequence ABC.

Side Effects

One side effect of the three rules is the realization that leaf nodes may be created (1) using the normal LZ78 method criteria or (2) through duplication as described in the enhanced algorithm. In the first case the leaf node will have a next node pointer that points to a root child node since the LZ78 algorithm specifies to reset the context pointer when a leaf is appended. In the second case the leaf will again have the next node pointer that points to a root child node. This is so because the node being duplicated is always a child of the root and children of the root always have a next node pointer which point to another child of the root. Children of the root always have next node pointer which point to another child of the root because when a new root child is created the context pointer is always reset.

Learning

One sequence that constructs Figure 2 would be "AABABC". The sequence parsing is A, AB, ABC. Another sequence that constructs the branch is ABC1ABC2ABC3. The sequence parsing is A, B, C, 1, AB, C2, ABC, 3. Note that at least three exposures are required to construct branch ABC using the standard LZ78 algorithm. As shown in Figure 7, only two exposures of the sequence ABC were required to construct the same branch with the enhanced LZ78 algorithm.

From this comparison it can be seen that the enhanced LZ78 algorithm is capable of learning repeating sequences faster then the standard LZ78 algorithm. This is accomplished by speeding the creation of dictionary structures when sequences repeat. The first exposure of a sequence of unique symbols will always generate child nodes of the root. These child nodes of the root may then be duplicated and appended as entire branches when the sequence repeats.

A best case, or big-Omega, experiment is performed to obtain a lower limit on the learning rate improvement. The experiment consists of ideal repeating sequences. Consider, for example, the construction of the dictionary branch ABC using instances of the sequence ABC. For the standard LZ78 dictionary tree, one sequence which will construct the branch is:

A, B, C, 1, AB, C2, ABC, 3.

Three exposures of ABC were required to construct the branch ABC. For the enhanced LZ78 dictionary tree, one sequence which will construct the branch is:

A, B, C, 1, ABC, 2.

Two exposures of ABC were required to construct the branch. Table 1 shows the results for sequence construction lengths from three to six.

Table 1. Idealized Minimum Exposures vs. Sequence Length.

Sequence Length	**Minimum Exposures For Branch Construction**	
	LZ78	**Enhanced LZ78**
3	3	2
4	4	2
5	5	2
6	6	2

In this idealized setting it is apparent that the best case learning rate for the LZ78 algorithm is proportional to the length of the repeating sequence. For the enhanced LZ78 algorithm, the rate is constant. It is possible to learn a sequence in two exposures. In a non ideal situation, where the next node pointer does not point to the next input symbol, the enhanced LZ78 learning rate degenerates to the standard LZ78 learning rate.

Macro Learning

A macro can be discovered by first quantifying macro like characteristics and then looking for those characteristics in the LZ78 dictionary tree. The start of a macro is easily identified. The start of the macro is the first, or one of the first, symbols in the dictionary tree. The end of the macro requires analysis. The endpoint can be identified by examining the number of children a node has. If the node has zero or one child then it is not the last symbol of a repeating sequence. If the node has two or more children then it is candidate macro endpoint as shown in Figure 8.

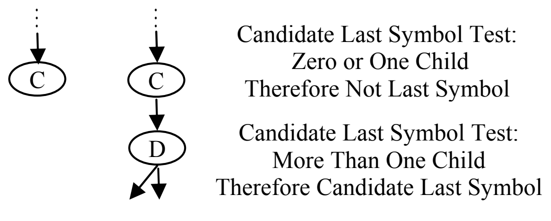

C
C
D
Candidate Last Symbol Test:
Zero or One Child
Therefore Not Last Symbol
Candidate Last Symbol Test:
More Than One Child
Therefore Candidate Last Symbol

Figure 8. Candidate Last Symbol Test.

With this start point and end point understanding, the last few user actions (the user's context) can be searched for in the dictionary tree. If this sequence of actions is found attached to the root, and a sub tree exists, then all branches in the sub tree become candidate macros based on the last symbol test.

Finally, the utility of the candidate macros can be evaluated. A simple utility is to weight each macro based on its visitation probability. Other weighing methods are possible (for more details and evaluation of different utility functions see (Elliott 2004)). Once the macros are weighed, the sequence which has occurred most frequently in the past can be offered back to the user.

Performance Study

The LZ78 and the enhanced LZ78 encoding algorithms were implemented and fed with identical input data. With identical input the approximate amount of compression and the number of dictionary nodes can be directly compared. In other words, the benefit of predicting more macros (provided by the next node pointers) can be compared with the new cost of duplicating nodes. This is the innate trade off of the enhancement. Translating the duplicating node cost into memory and time overheads would be implementation dependant.

Representative Scenario

A "printing" macro test scenario was created in a situational environment likely to occur. One goal of the test was to compare learning rate performance.

The test scenario calls for the user to launch a file from a desktop shortcut and then print the file with a certain consistent set of options. The user closes the file and then works in other applications. The idea of the scenario is that the printed document might, for example, be some sort of report that must be printed for a weekly team meeting. The user works in other applications during the week and only at specific times does the document need to be printed. The macro of interest has a length of ten symbols and is shown in Table 2.

The macro of interest is then embedded within other symbols which represent other application actions. For the macro learning system to succeed, the sequence of interest must be discovered and learned while other user activity is occurring. The entire desktop simulation sequence with the embedded macro subsequence is shown in Table 3. Items marked with "*" are random numbers with the range specified. A new random number is generated each iteration.

Table 2. Macro of Interest.

Symbol	Description
app 1	active window change
app 1 1	file
app 1 2	print
Print	active window change
print 1	properties
print 2	# copies field
print 3	keyboard 5
print 4	ok
app 1	active window change
app 1 3	close document

In this simulation the macro of interest has two prefixing symbols: "desktop" and "desktop 1". One can see from the symbol column of Table 3 that symbol "desktop" prefixes other actions in other application sequences. Note the "z" in rows three, four and five. z takes on a value of one through nine in line three and the value is transferred to lines four and five. In essence, the macro of interest is intermixed with eight other applications.

Table 3. Simulation Sequence.

Symbol	Iterations		
desktop		1 to 4 times *	Macro sequence "exposure" times
desktop 2-30 *			
app 1-9 (=z)			
app z 4-99 *	10 to 20 times *		
app z 3			
desktop			
desktop 1			
[macro of interest]			

Table 4. Predictions for Various Macro Exposures.

		Macro Sequence Exposures													
		1	**3**	**5**	**7**	**9**	**11**	**13**	**15**	**17**	**19**	**21**	**23**	**25**	**27**
Standard LZ	**No Predictions %**	100	100	100	100	100	100	100	96	71	36	16	5	1	0
	Correct Predictions %	0	0	0	0	0	0	0	4	29	64	84	94	99	100
	Number Nodes	34	107	172	233	291	345	397	448	498	547	594	639	685	729
	Compression %	100	97	95	93	91	89	88	86	85	84	84	83	82	82
Enhanced LZ	**No Predictions %**	100	100	60	12	0	0	0	0	0	0	0	0	0	0
	Correct Predictions %	0	0	40	88	99	100	100	100	100	100	100	100	100	100
	Number Nodes	34	111	175	233	289	342	394	444	493	541	588	632	678	722
	Compression %	100	91	85	82	80	79	78	78	77	77	76	76	76	76

The sequence in Table 3 was repeated 400 different times (400 trials). Predictions, nodes, and compression statistics were collected and averaged over the trials. The results are shown in Table 4.

If the extraction algorithm made an average of zero predictions in the 400 trials then a value of 0 is entered in the "No Predictions" row. If the algorithm made a prediction and the prediction was 100% correct then the row labeled "Correct Predictions" is filled in. The average number of LZ78 dictionary nodes is shown in the "Number Nodes" row. Finally if the algorithm were to be used to output compressed code words then the approximate amount of compression that may be realized is filled in the "Compression %" row.

By comparing the "Correct Predictions" rows in Table 4 it can be seen that the learning rates are different between the algorithms. The macro of interest in this experiment has a length of ten symbols and a context prefix length of two symbols. The macro was learned with 100% effectiveness in 11 exposures using the enhanced LZ78 algorithm and in 27 exposures using the normal LZ78 algorithm. The learning rate in this example has improved by a factor of about 2.5.

Compression

By comparing the "Compression" rows in Table 4 it can be seen that the compression is notable better with the enhanced LZ algorithm with a small number of exposures. The compression improvement appears to taper off as the number of exposures accumulates.

Number of Tree Nodes

Note the two "Number Nodes" rows in Table 4. The number of nodes in the enhanced LZ78 tree is slightly less than the number of nodes in the standard LZ78 tree after seven exposures. This difference can be explained by recalling rule three of the enhanced LZ78 algorithm. Rule three calls for not resetting the current context pointer to the root when duplicating nodes. Resetting the current context pointer creates suffix branches of the sequence of interest with the standard LZ78 algorithm. Not resetting the context pointer reduces the number of suffix branch nodes. For this example, fewer suffix nodes are created and more of the sequence of interest branch has been constructed after seven exposures.

Conclusions

In this paper we explored applying and enhancing the Lempel-Ziv LZ78 algorithm to learn user macros. Incorporating the LZ78 algorithm prevents a brute force search of the user's history. Using the algorithm also simplifies the macro sequence start and end point identification search. By merely noting the LZ78 dictionary structure we saw that the macro demarcation points can be easily identified and that a candidate list of macros is formed.

One problem with the Lempel-Ziv algorithm is its slow learning rate. To improve the learning rate we added a set of three new algorithmic rules. With these rules the enhanced LZ78 algorithm can potentially learn a macro sequence with far fewer sequence exposures. The lower limit is reduced from proportional to the macro length to a constant of two.

Since the LZ algorithm is a one pass algorithm its compression, and thus its predictive performance, is not ideal. But with the simple addition of a next node pointer, the short term predictive capability of the algorithm can be improved.

References

Bhattacharya, A., and Das, S., LeZi-update: An Information-theoretic framework for personal mobility tracking in PCS networks, Wireless Networks, Volume 8, Issue 2/3, Page 121 – 135, March 2002.

Elliott, F., Learning and Identifying Desktop Macros Using an Enhanced LZ78 Algorithm, Technical Report CSE-2004-12, University of Texas at Arlington, 2004.

Feder, M., Merhav, N., and Gutman, M. Universal Prediction of Individual Sequences, IEEE Transactions on Information Theory, Volume 38, Issue 4, Page 1258 – 1270, July 1992.

Gopalratnam, K., and Cook, D., Active Le-Zi: An Incremental Parsing Algorithm for Sequential Prediction, Proceedings of the Florida Artificial Intelligence Research Symposium, 2003.

Russell, S., and Norvig, P., Artificial Intelligence a Modern Approach, Prentice Hall, Page 5 – 6, 1995.

Sandanayake, P., and Cook, D., Imitating Agent Game Strategies Using a Scalable Markov Model, Proceedings of the Fifteenth International Florida Artificial Intelligence Research Society Conference, Page 349 – 353, May 2002.

Ziv, J., and Lempel, A., Compression of Individual Sequences via Variable-rate Coding, IEEE Transactions on Information Theory, Volume 24, Issue 5, Page 530 – 536, September 1978.