

2008

# An automata based authorship identification system

Shangxuan Zhang  
*San Jose State University*

Follow this and additional works at: [https://scholarworks.sjsu.edu/etd\\_theses](https://scholarworks.sjsu.edu/etd_theses)

---

## Recommended Citation

Zhang, Shangxuan, "An automata based authorship identification system" (2008). *Master's Theses*. 3516.  
DOI: <https://doi.org/10.31979/etd.fewh-t45k>  
[https://scholarworks.sjsu.edu/etd\\_theses/3516](https://scholarworks.sjsu.edu/etd_theses/3516)

This Thesis is brought to you for free and open access by the Master's Theses and Graduate Research at SJSU ScholarWorks. It has been accepted for inclusion in Master's Theses by an authorized administrator of SJSU ScholarWorks. For more information, please contact [scholarworks@sjsu.edu](mailto:scholarworks@sjsu.edu).

AN AUTOMATA BASED  
AUTHORSHIP IDENTIFICATION SYSTEM

A Thesis

Presented to

The Faculty of the Department of Computer Science  
San Jose State University

In Partial Fulfillment  
of the Requirements for the Degree  
Master of Science

by

Shangxuan Zhang

August 2008

UMI Number: 1459705

### INFORMATION TO USERS

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleed-through, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

**UMI**<sup>®</sup>

---

UMI Microform 1459705

Copyright 2008 by ProQuest LLC.

All rights reserved. This microform edition is protected against unauthorized copying under Title 17, United States Code.

ProQuest LLC  
789 E. Eisenhower Parkway  
PO Box 1346  
Ann Arbor, MI 48106-1346

© 2008

Shangxuan Zhang

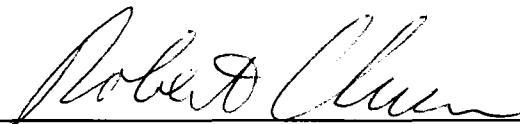
ALL RIGHTS RESERVED

APPROVED FOR THE DEPARTMENT OF COMPUTER SCIENCE



---

Dr. Tsau Young Lin



---

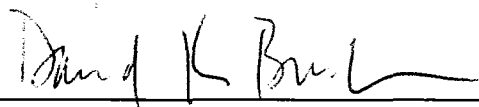
Dr. Robert Chun



---

Dr. Howard(Ching-Tien) Ho  
IBM ALMADEN RESEARCH CENTER

APPROVED FOR THE UNIVERSITY



# ABSTRACT

## AN AUTOMATA BASED AUTHORSHIP IDENTIFICATION SYSTEM

by Shangxuan Zhang

This thesis gives a design and implementation for an authorship identification system based on automata modeling. The writing samples of an author were collected to build a tree and use the ALERGIA algorithm to merge all the compatible states of the tree in order to get a stochastic finite automaton. This automaton represents the writing style of the author. We can use this automaton to test whether an anonymous writing piece belongs to this author.

## ACKNOWLEDGMENTS

Foremost, I would like to thank my advisor Tsau Young Lin for his invaluable insight and inspiring guidance, without which I would have lost my direction and never come to the end of my research.

Moreover, I offer my deepest appreciation to Dr. Robert Chun and Dr. Howard Ho for participating in my thesis committee.

My special thanks go to Dr. Cay Horstmann for his help in the past two years. Without his email, I would not know my application material wasn't delivered correctly and who I should talk with about this issue.

I have been very lucky to have many supportive and loving family members. The person who deserves my gratitude the most is my husband, Baosen Wu. I thank my parents, Chengji Zhang and Tingting Hong, for giving me the freedom to explore my interests.

Finally, I would like to express my sincere thanks to the Department of Computer Science at San Jose State University. Without two years study here, I might not be able to be admitted to

Applied Mathematics and Statistics Department of Stony Brook  
University for 2008 fall and get Wise2008 Fellowship from Cornell  
University.

I made many friends and earned a bright future from SJSU.

Thank you all!



## TABLE OF CONTENTS

1. INTRODUCTION .....	1
2. STOCHASTIC FINITE AUTOMATA .....	4
3. ALERGIA ALGORITHM .....	7
4. AUTOMATA BASED MODELING .....	11
5. IMPLEMENTATION .....	18
6. USAGE OF SOFTWARE .....	29
7. RESULTS .....	37
8. CONCLUSION .....	42
REFERENCES .....	43
APPENDIX A: STOP WORD LIST .....	46
APPENDIX B: DETAILS OF HOW THE FUNCTIONS IN THE PROGRAM WORK .....	49
APPENDIX C: TEST ENVIRONMENT AND PERFORMANCE .....	56

## LIST OF TABLES

Table 1. STATISTIC DATA OF THE PTA.....	16
Table 2. THE TABLE OF TESTING WITHOUT MERGING.....	39
Table 3. THE RESULT OF TESTING WITH $A=0.7$ .....	40

# LIST OF FIGURES

FIGURE 1. FIRST EXAMPLE OF PTA ..... 14

FIGURE 2. EXAMPLE OF PTA ..... 15

FIGURE 3. SFA RESULTED FROM MERGING ..... 16

FIGURE 4. CLASS VIEW OF THE PROGRAM ..... 18

FIGURE 5. THE INTERFACE OF THE PROGRAM ..... 29

FIGURE 6. THE TAB OF SETTING ..... 30

FIGURE 7. THE RESULT AFTER TRAINING ..... 32

FIGURE 8. THE RESULT AFTER TEST ..... 34

FIGURE 9. THE FILE AUTOMATON.TXT ..... 35

FIGURE 10. AUTOMATON FROM EXAMPLE ..... 38

FIGURE 11. SAMPLE .TXT ..... 49

FIGURE 12. TEST.TXT ..... 50

FIGURE 13.  $A=0.9$  ..... 51

FIGURE 14.  $A=0.8$  ..... 52

FIGURE 15.  $A=0.4$  ..... 52

FIGURE 16.  $A=0.3$  ..... 53

FIGURE 17. AUTOMATON FOR  $A=0.9$ ..... 54

FIGURE 18. AUTOMATON FOR  $A=0.8$ ..... 54

FIGURE 19. AUTOMATON FOR  $A=0.4$ ..... 54

FIGURE 20. AUTOMATON FOR  $A=0.3$ ..... 55

# 1. INTRODUCTION

Based on the Kolmogorov complexity  $K(x)$  for binary string  $x$ , in 1993, Lin proposed to use the opposite of randomness as the concept of patterns [1], namely, a sequence  $x$  has pattern if  $K(x) \ll \text{length}(x)$ . Obviously, one can conclude that a sequence is said to have pattern if and only if there exists a constant subsequence (Lin stated it for infinite sequences). This could be viewed as the foundation of frequent item sets (high frequency patterns). In [2], Lin ported the idea to numerical world. In [3], the idea was ported to the world of finite automata, in which the automata were used to detecting (learning the patterns) the sequences of system calls in program. Here we switch the applications from the intrusion detection system to authorship identification system, in which we use automata to detect the string of stop words in a book.

It is well-known that every author has some particular writing style, depending on his or her gender, age, experience, knowledge, etc. To illustrate, some people name a few statistic writing characters: average word length, average sentence length in words, word frequency, etc. Given an anonymous writing piece and possible

authors with their writing samples, theoretically, one can investigate these writing characters and identify the author of this writing piece [4].

Life is easy if that is the whole story. In practice, we don't have a complete set of quantities to characterize the writing style. Even if such a set exists, it must be too huge to incorporate into a program. On the other hand, it seems not possible to describe the writing style only by using these statistic quantities. There are some hidden relations between the contexts. Hidden Markov model has been used widely to reveal these relations.

The aim of this paper is to study authorship identification through function words based on the theory of automaton. Function words have long ago been used to identify the writing style. Recently, some interesting work has been done along this direction.

This work is inspired by the work of P.Baliga and T.Y.Lin on the virus intrusion detection system [3]. More precisely, we collect writing samples of a prescribed author. From each sample, we keep the function words for each sentence and wipe out all other information. These sequences of function words are actually the realization of a

hidden automaton. Our goal is to use this data and machine learning technique to figure out this automaton, which is our representation of the normal writing pattern of the author.

For any other writing sample, our program will test the structure of function words sentence by sentence. We record the proportion of sentences which pass the test. The higher the proportion, the more likely this sample belongs to the author. It is recommended to combine this result with other classical methods of authorship identification to get a more accurate result.

The content of this paper is organized as follows. In section 2, we review stochastic finite automata. In section 3, we describe the ALERGIA algorithm which is used to build an automaton from sample data. In section 4, we handle the data of writing samples, and describe the application of the algorithm to our specific problem. In section 5 we give a briefly description of the implementation of the program. In section 6 we introduce the main feature of the software. Finally in section 7 we present partial results of the running of our program.

## 2. STOCHASTIC FINITE AUTOMATA

In this section we shall review the notion of finite automata and its variation stochastic finite automata [5-11]. In this paper, we shall limit ourselves to deterministic automata. In later sections, we are primarily interested in stochastic finite automata. The basic ingredients are same except the extra information of transition probability.

A deterministic finite automaton (DFA) is a 5-tuple

$$(Q, A, \delta, q_0, F),$$

where  $Q = \{q_0, q_1, \dots, q_n\}$  is its set of states,  $A$  its input symbols,  $\delta$  its transition function that takes a state and an input symbol as arguments and return a state,  $q_0$  its start state, and  $F$  its set of accepting states.

One simplest nontrivial DFA is an on/off switch. This device has two states: "on" and "off." The user can press the button to switch one state to another state. For general purpose, one can assign "off" as start state and "on" as accepting state.

In reality, a lot of phenomena are actually random. It motivates the following generalization of deterministic finite automata to stochastic finite automata.

A stochastic finite automata (SFA) consists of a DFA  $(Q, A, \delta, q_0, F)$ , and a set  $P$  of probability matrices  $p_{ij}(a)$  for each symbol  $a$  in  $A$ . Each  $p_{ij}(a)$  gives the probability of the  $a$  transition from the state  $q_i$  to state  $q_j$  led by the symbol  $a$ . We let  $p_{if}$  be the probability that the string end at state  $q_i$ . Then we have the following constraint:

$$p_{if} + \sum_{q_j \in Q} \sum_{a \in A} p_{ij}(a) = 1.$$

Intuitively, it means that for each state  $q_i$ , the sum of the probabilities end at  $q_i$  and the probabilities start at  $q_i$  should equal to one.

Let  $A^*$  be the set of all strings on  $A$ . For each string  $w$ , one can define the probability  $p(w)$  inductively as usual. The language generated by the automaton is defined as:



$$L = \{w \in \mathcal{A}^* : p(w) \neq 0\}.$$

A stochastic regular language (SRL) is defined to be the language generated by an SFA. Two SRLs are said to be equivalent if they contain the same set of strings with the same corresponding probabilities, that is,

$$L_1 \equiv L_2 \Leftrightarrow p_1(w) = p_2(w), \forall w \in \mathcal{A}^*,$$

where  $L_1$  and  $L_2$  are two SRLs, and  $p_i(w)$  is the probability of the transition led by  $w$  in language  $L_i$ .

### 3. ALERGIA ALGORITHM

In this section we recall the ALERGIA algorithm to deal with the following problem: Given a fixed SFA, there will be a SRL defined by this SFA. Now suppose the structure of this SFA is not informed, instead a large random subset of strings is given as the SRL generated by this SFA. The goal is to reconstruct the SFA from this given set of strings. For details of the method in this section, please see [6].

Now we describe the approach to solve this problem. First of all, It is to build a tree from these data. This tree is called a prefix tree adapter (PTA). Each node of the PTA represents a state. For each node of the tree, we assign the frequency of transition led by each symbol. Next, each node of the PTA is compared pairwise. The equivalence of nodes is defined. According to this equivalence, the nodes are classified and merged with the equivalent nodes of the PTA. At the end, the frequencies are recalculated and we can conclude a SFA which is an approximation of the original SFA.

Let us start with the definition of PTA. Now suppose the set of sample data is  $S = \{s_1, s_2, \dots, s_m\}$ . We describe the PTA inductively.

For each string  $s_k = a_1 a_2 \dots a_k$ , we begin with the initial node  $q_0$ . Suppose there is a transition from  $q_0$  to one of its child node  $q_i$  led by  $a_1$ , we follow this transition and move to the node  $q_i$ . Otherwise, we add a new node to this tree, the transition from  $q_0$  to this new node is thus led by  $a_1$ . Either way, we move to a new node, now we look at symbol  $a_2$  and continue this process. In the end, we reach a node that accepting this string. One example of this procedure is given in the next section.

When we run through all the sample data, we can assign the frequency of appearance of each symbol as a transition between nodes, and the number of strings entering each node, the number of string accepting by each node. We denote by  $n_i$  the number of strings arriving at node  $q_i$ ,  $f_i(a)$  the number of strings following transition  $\delta_i(a)$  and  $f_i(\#)$  the number of strings ending at node  $q_i$ . Obviously,  $f_i(a)/n_i$  and  $f_i(\#)/n_i$  gives estimate of the probabilities  $p_i(a)$  and  $p_{if}$  respectively.

After we obtain the PTA, we introduce the notion of equivalence between two nodes. Two nodes are said to be equivalent if for all symbols 'a' belongs to A, "the associated transition probabilities from the nodes are equal; the termination probabilities for the nodes are

equal; and the destination nodes of the two transitions for each symbol are equivalent according to a recursive application of the same criteria." In symbols, we have

$$q_i \equiv q_j \Rightarrow \forall a \in A, \text{ we have } p_i(a) = p_j(a) \text{ and } \delta_i(a) \equiv \delta_j(a).$$

In the application of this notion, since we seldom have two equal frequencies by statistic fluctuations in experimental results, the equivalence of two nodes must also be accepted within a confidence range. To this end, we call two nodes are compatible if they are equivalent within some pre-described confidence range.

Since for a Bernoulli variable with probability  $p$  and frequency  $f$  out of  $n$  tries, the confidence range is given by the Hoeffding bound as follows:

$$\left| p - \frac{f}{n} \right| < \sqrt{\frac{1}{2n} \log \frac{2}{\alpha}} \text{ with probability larger than } (1 - \alpha).$$

When the two estimated probabilities differ more than the sum of the confidence ranges, the ALERGIA algorithm will reject equivalence.

$$\left| \frac{f}{n} - \frac{f'}{n'} \right| > \sqrt{\frac{1}{2} \log \frac{2}{\alpha}} \left( \frac{1}{\sqrt{n}} + \frac{1}{\sqrt{n'}} \right).$$

Finally, when two nodes are merged, we should recalculate their frequencies and node numbers in order to ensure that the SFA remains deterministic and order-preserving.

## 4. AUTOMATA BASED MODELING

In this section we shall describe how to model the authorship identification problem using automata.

Our authorship identification approach utilizes function words based automata modeling. In this approach, the first step is to choose an author and collect as many writing samples as possible for use as training data sets that are representative of standard writing style for this author. In the sequel, we shall use the following paragraph as writing sample to illustrate the idea. This piece is cited from the beginning of *Harry Potter and the Prisoner of Azkaban*.

*"Harry Potter was a highly unusual boy in many ways. For one thing, he hated the summer holidays more than any other time of year. For another, he really wanted to do his homework but was forced to do it in secret, in the dead of night. And he also happened to be a wizard."*

After choosing the sample, we fix the basic unit of training data, which can be one sentence, one paragraph or one whole article, then

cut all writing samples into the predetermined units. In this paper, we use one sentence as a unit. The result is finer if the unit is made bigger. However, the running time is longer if we choose larger unit and we need more sample data to keep the number of units large enough to use the ALERGIA algorithms effectively.

In our example, we have four sentences. So we get four units in the sample data. For each unit in the sample, we keep the function words and remove all the other content words. This can be done by choosing a predetermined function words list. We compare each word in the unit according and if the word matches a word in the list, we keep it. Applying this to the example, we obtain the following four sequences:

*was a in many*

*for one he the more than any other of*

*for another he to do his but was to do it in in the of*

*and he also to be a*

Now since the number of function words is around several hundred, to build a tractable automaton, this number is still large as

the alphabet of an automaton. The next step is to replace each function word with its part of speech. Usually, we have the following classes of function words: adverb, auxiliary verb, pronoun, preposition, conjunction, interjection and number.

In the following, we use the digits 0, 1, 2, 3, 4 to represent adverb, auxiliary verb, preposition/conjunction, pronoun and number respectively. This way, we greatly simplify the data of each unit into a sequence of numbers. As an example, we obtain the following sequence of digits.

1 3 2 3

2 4 3 3 3 2 3 3 2

2 3 3 2 1 3 2 1 2 1 3 2 2 3 2

2 3 0 2 1 3

Now from this data we follow the method described in the previous section, we can build the following PTA (Fig. 1).



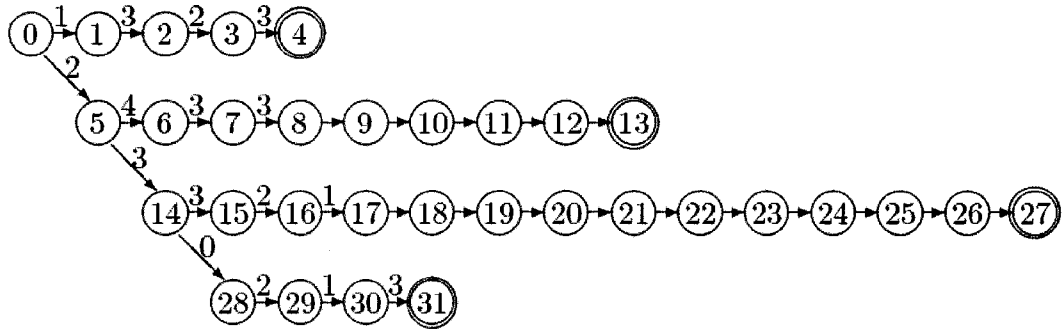


FIGURE 1. FIRST EXAMPLE OF PTA

One can calculate the frequency for the transition from each node to its children by virtue of the data recorded in the PTA.

To illustrate the method, let's take a look at node 5 in our example. We have totally four strings in sample data, out of which the last 3 strings arrive at node 5. By our notation in section 2, we have  $n_5=3$ , where the subscript 5 represents node 5. Notice that node 5 has two children, one is node 6, and another one is node 14. There is only one string following the transition symbol 4 from node 5 to node 6, thus  $f_5(4)=1$ .

Likewise we have  $f_5(3)=2$  and  $f_5(a)=0$  for  $a \neq 3,4$ . Since a node with a double circle means there is at least one string ending at this node, we know there is no string ending at node 5, and obtain  $f_5(\#)=0$ .

In the above example, we have insufficiently few data, so the frequency is not accurate as the approximation of probabilities. Ideally, when we go through a large set of sample data, we can get a large PTA which approximates the probabilities quite well. From this PTA, one can merge the compatible nodes to get an SFA. We regard this resulting SFA as a representative of the writing style of the author. A string is seemed to be belonged to the same author if it is accepted by this SFA.

As an example, we look at another set of data as sample.

Suppose we have a set of strings:

$\{0,01,01,011,0101,0101,0101,0101,0101,010101,010101\}$ ,

We can build the following PTA (Fig. 2) according to the method described earlier:

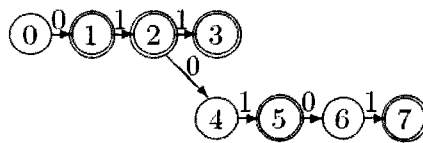


FIGURE 2. EXAMPLE OF PTA

We calculate the values of  $n_i$ ,  $f_i(\#)$  and  $f_i(a)$  for  $a=0,1$  and  $0 \leq i \leq 7$  in the following table (table 1).

TABLE 1. STATISTIC DATA OF THE PTA

Node $i$	0	1	2	3	4	5	6	7
$n_i$	11	11	10	1	7	7	2	2
$f_i(\#)$	0	1	2	1	0	5	0	2
$f_i(0)$	11	0	7	0	0	2	0	0
$f_i(1)$	0	10	1	0	7	0	2	0

It is obvious from the table that node 3 and node 7 are equivalent. If we let  $\alpha=0.7$ , then one can check that node 5 and node 7 (or 3) are compatible because

$$\left| \frac{f_5(\#)}{n_5} - \frac{f_7(\#)}{n_7} \right| = \frac{2}{7} < \sqrt{\frac{1}{2} \log \frac{2}{0.7} \left( \frac{1}{\sqrt{n_5}} + \frac{1}{\sqrt{n_7}} \right)}.$$

$$\left| \frac{f_5(0)}{n_5} - \frac{f_7(0)}{n_7} \right| = \frac{2}{7} < \sqrt{\frac{1}{2} \log \frac{2}{0.7} \left( \frac{1}{\sqrt{n_5}} + \frac{1}{\sqrt{n_7}} \right)}.$$

Similarly, one can verify that node 4 and node 6 are compatible. For other pair of nodes, this inequality does not hold. So we can merge nodes 3, 5, 7 and get the following SFA (Fig. 3):

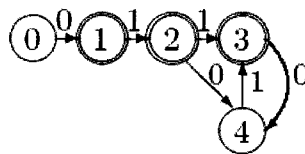


FIGURE 3. SFA RESULTED FROM MERGING

Now for any piece of writing, we form the sequences of digits according to the method mention above. Suppose the number of sequences is  $m$ . For each sequence, we test if it is accepted by the SFA. The number of accepting sequences is denoted by  $m_a$ . Therefore we get a quotient  $m_a/m$  which is called the accepting probability.

For instance, if we have a set of 4 strings  $\{01010101, 0111, 001, 01010\}$  which are all different from our sample strings. Applying our test program, we see that only the first string 01010101 is accepted by this SFA. The accepting probability is then equal 0.25. We remark that the accepting probability depends on the parameter  $\alpha$  in our method. This parameter is used to control the accuracy of our merge process. Sometimes it is possible to merge non-equivalent states when  $\alpha$  is too small.

## 5. IMPLEMENTATION

The code for our program was written by C++. We compile the code on Windows XP, using MFC. We now describe the major structure of the implementation (Fig. 4).

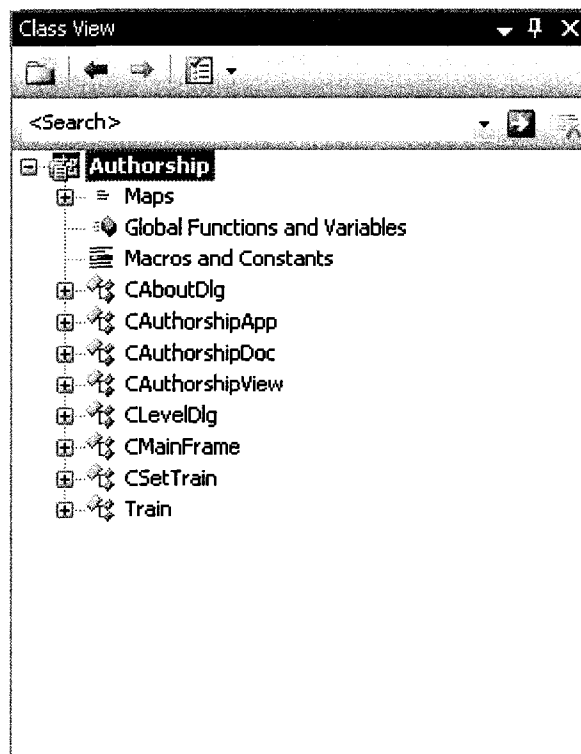


FIGURE 4. CLASS VIEW OF THE PROGRAM

The main class is the following:

```
class Train : public CObject
{
public:
    Train(void);
```

```

public:
    ~Train(void);

private:
    struct node{
        long label;
        long par;
        long num_tdata;
        long num_acpstring;
        bool end;
        bool merged;
        long merge_to;
        long child[WordType];
        long num_appear[WordType];
    };

public:
    static const long StateBound=1000000;//number of state
    static const int WordType=5;//number of stop words
    static const int M=1;//sentence num
    static const int WordLength=100;
    static const int WordNumber=100;
    static const int Exception1=10;

```

```

static const int Exception2=20;

static const int Exception3=30;

enum {Adv,Aux,Prep,Pron,Number};

public:

    long max_state,trCounter;

    node state[StateBound];

    long temp[StateBound];

    long treeEnd[StateBound];

    long count;

    double progress;

    double a;

public:

    long GetFunWord(CString dir,CString in,CString
out_dir,CString out);

    int CreatePTA(CString dir,CString in);

    int Compatible(long node_i, long node_j);

    int Differ(double n_1,double n_2,double f_1,double
f_2);

    long Delta(long i, int t);

    int MergeAll(CString dir);

    int Combine(void);

```

```
int TestAuthor(CString dir, CString name);  
};
```

In this class `Train`, we use `struct node` to store the data of the nodes of the SFA. Precisely,

`label` is a long integer represents the index of the nodes;

`par` is the parent of the node;

`num_tdata` represents the number of all strings pass through this node;

`num_acpstring` is the number of strings that are accepted by this node; if this node is not an accepting state, the value of this variable is zero;

`end` is a bool type variable, it is set to true if the current node is an accepting state, otherwise it is set to false;

`merged` and `merge_to` are used when we merge compatible states;



`child[WordType]` is an array that gives children of the node, for each string we have a corresponding child, the number of children cannot be greater than the number of word types. For each word type, we record the number of string pass through by this string by the variable `num_appear[WordType]`.

The major methods in class `Train` are described as follows:

The first function is

```
long GetFunWord(CString dir,CString in,CString  
out_dir,CString out);
```

The arguments of this function are the input directory of the text file and the output directory of the resulting files. It reads the text file word by word and translates the stop words into its corresponding part of speech which is represented by an integer between 0 and 4; it also ignores all content words. The result is written to a new file consists of numbers. After this step, we abstract the text into a workable integer sequence. Finally, we use -1 to mark the end of each sentence

and the end of the whole text. As a byproduct, we record some statistic data into another text file for possibly later use.

The second function is

```
int CreatePTA(CString dir,CString in);
```

It is the first step to create the SFA. When we get a sequence of stop words, we want to first construct a PTA by virtue of the given sequence. This function starts to create the states of the PTA one by one. The arguments of the function are text file directory and file names. The result of running this function is the assignment of value to the array `state[StateBound]` which stores the nodes of the PTA.

The next few functions

```
int Compatible(long node_i, long node_j);
```

```
int Differ(double n_1,double n_2,double f_1,double f_2);
```

```
long Delta(long i, int t);
```

are easy to understand, they calculate the statistic data of the SFA, these data are used to merge compatible states. We remark that function `Delta` is basically the transition function of the SFA.

The process of merging is done by functions

```
int MergeAll(CString dir);
```

```
int Combine(void);
```

here `Combine` is a preprocessor for merging, it indices all pairs of nodes needed to be merged, the real merging is done by `MergeAll` which changes the value of children and parents.

We now explain the main idea in these functions.

The following is the source code of the function `Combine()`;

```
int Train::Combine(void)
{
    int l=0;
    for(long i=0;i<trCounter;i++){
```

```

long j=treeEnd[i];

int m=0;

long temp=state[j].par;

while(temp!=0){

    bool pass=false;

    while((temp!=0)&&(!(pass=Compatible(j,temp)))){

        temp=state[temp].par;

    }

    if(pass){

        state[j].merge_to=temp;

        state[j].merged=true;

        if(state[j].end==true){

            state[temp].end=true;

        }

        j=state[j].par;

        temp=state[temp].par;

        m++;

    }

}

if(l<m)

    l=m;

}

for(long i=1;i<=max_state;i++){

    long k=state[i].merge_to;

    if(k!=i){

        while(state[k].merge_to!=k)

            k=state[k].merge_to;

    }

}

```

```

        state[i].merge_to=k;
    }
}

return 0;
}

```

Primarily, this function set the bool value variable *merged* to be true when the corresponding node has been identified to its compatible pairs, although the real merge is not done. The long integer value variable *merged\_to* is the label of compatible node.

The essential part of the source code of the function MergeAll(CString) is the following:

```

int Train::MergeAll(CString dir)
{

    //some deleted code here to deal with file operations

    Combine();

    for(long i=1;i<=max_state;i++){
        long k=state[i].merge_to;
    }
}

```

```

        if(k!=i){
            long p=state[i].par;
            int j=0;//find the transform string 'j' from the
parent p to the child i;
            while(state[p].child[j]!=i)
                j++;
            state[p].child[j]=k;//set the child of p as k instead
of i;

            for(int j=0;j<WordType;j++){
                if(state[i].child[j]>=0){
                    if(state[k].child[j]<0)

state[k].child[j]=state[state[i].child[j]].merge_to;
                    else

state[k].child[j]=state[state[k].child[j]].merge_to;
                }
            }
        }
    }

    /* write the automaton into the output file automaton.txt */
    //the code deleted for brevity

    return 0;
}

```

You may find the process of merging is slightly different from the algorithm described in previous section. The reason is that the current method we used here is quicker than the one in the theoretical part. To deal with a large set of data, we have to sacrifice the relative accuracy of the result to make the program running in more realistic limited time constraint. For different branches in PTA, the states weren't merged since it won't bring out new knowledge by doing it. This automaton is still equivalent to the originally proposed automation since they can accept the same language.

As our result is already good enough to distinguish the authors, we don't have to improve the program to a limited upper level. It doesn't worth waiting for a long time to see a little improved result on quantity level.

## 6. USAGE OF SOFTWARE

In this section we briefly introduce the functions of the software *Authorship*. This program is designed to run in a Windows XP operating system.

After open *Authorship*, you will see a following simple window (Fig. 5).

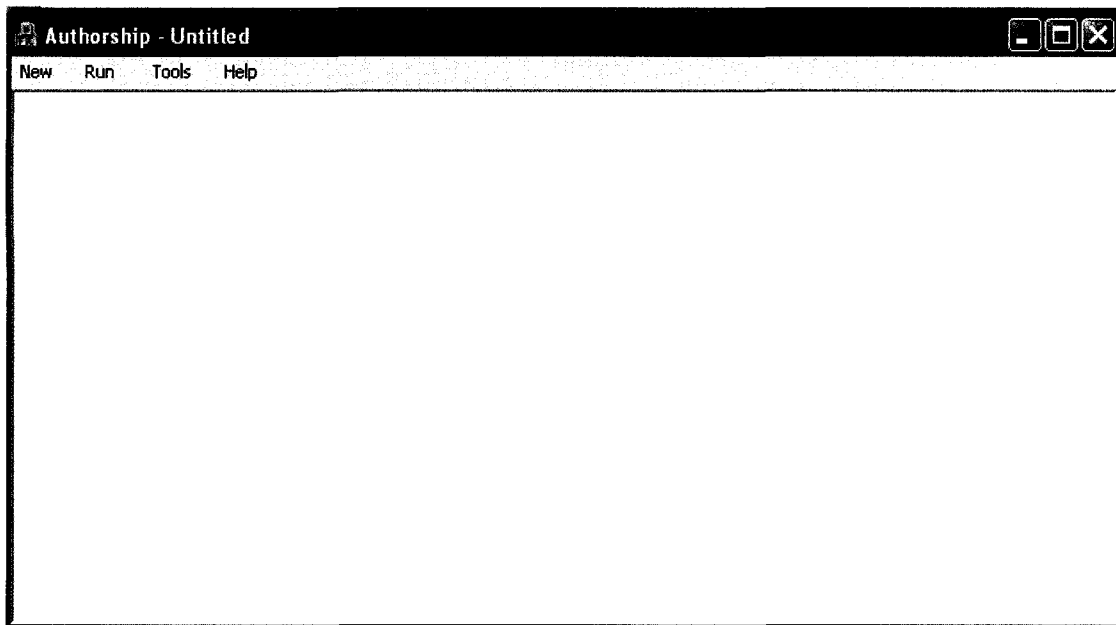


FIGURE 5. THE INTERFACE OF THE PROGRAM

Before running of the program, we need to get familiar with the menu in this window.



The most frequently used menus are Run and Tools. One needs to first open Tools and click the first item Setting to setup parameters needed to run the program. The first important parameter is the confidence level, and other parameters include the directory of data files.

When you click Setting item, you will see a window popped up as shown in the following (Fig. 6).

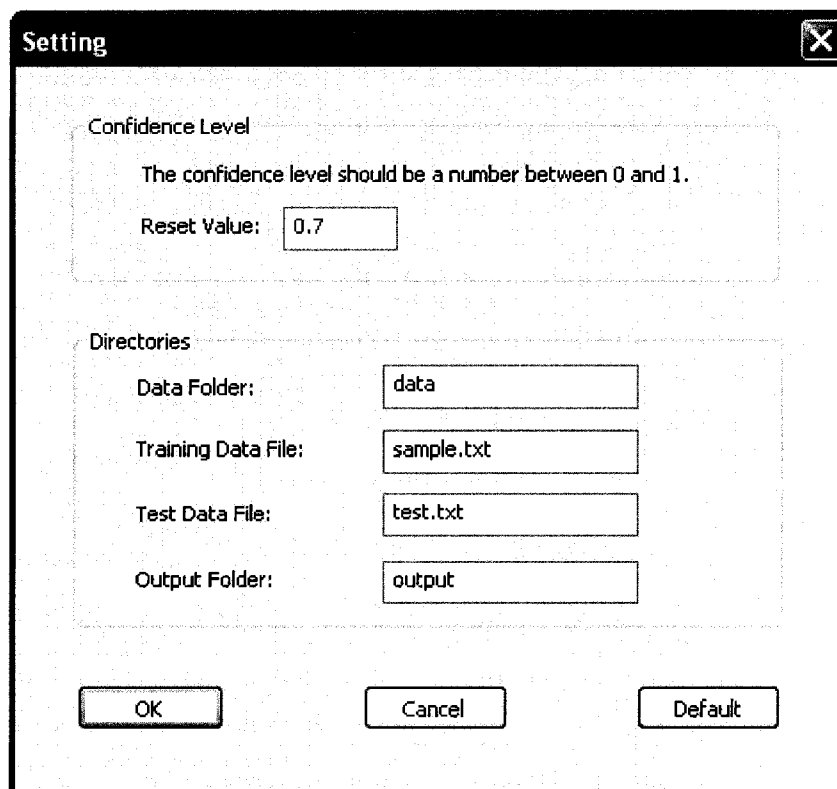


FIGURE 6. THE TAB OF SETTING

The data in this tab are set to default values as above. The Confidence Level is a parameter which controls the degree of merging. This value should be a number between 0 and 1, the smaller of this number; the coarser of the merging process, that is, more states are regarded as compatible and merged. The resulting SFA will accept more language and actually the confidence of authorship will decline.

On the other hand, if this value is big and close to 1, few states are merged and the standard for a language to be accepted is high. In this case, some other writing piece of the same author would probably be rejected in the testing due to the difference in writing style. We need to adjust this parameter appropriately so that it is practically useful and reasonable. For the moment, the author believes that 0.7 is an ad hoc appropriate value.

The second data need to be set are the sample text file directory and file name, and test file directory and name. The default values for these are data/sample.txt and output/test.txt. You can change them by hand. After you set the value, you need to create the corresponding directories and files.

Now it is ready to train the program data, click Run->Train, the windows will show

Training data, please wait...

This process may take time, so don't close the window during training. After the completion of training, you will see the following information (Fig. 7):

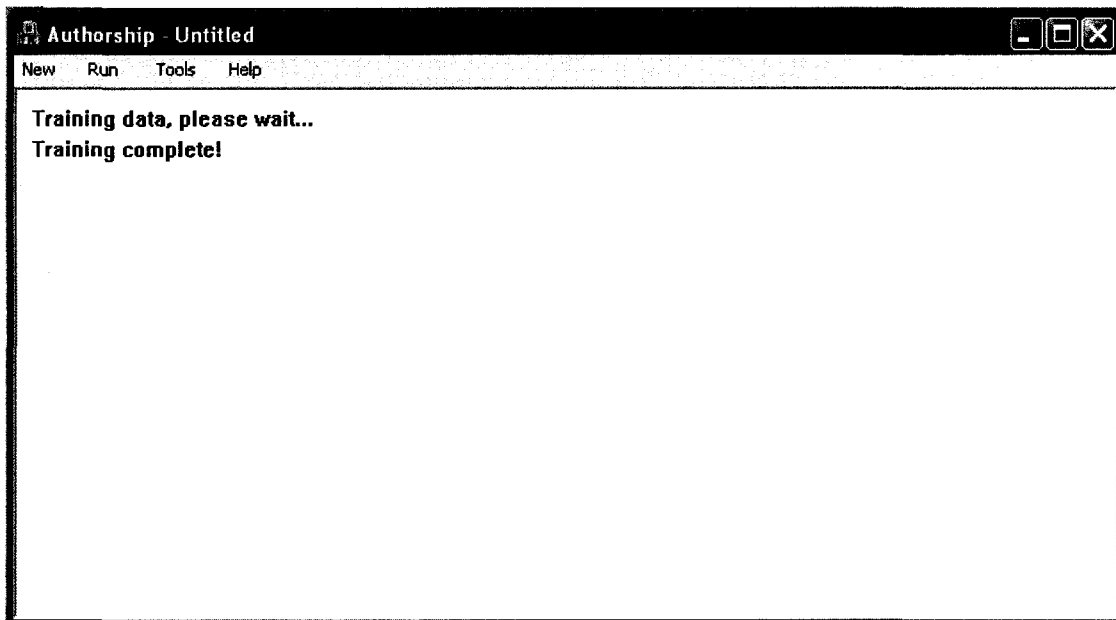


FIGURE 7. THE RESULT AFTER TRAINING

When you see this message, the SFA represented the writing style of the author has been generated. You can then test the writing

piece stored in the text file test.txt (or the file specified by you in the setting tab).

To test the data, simply click Run->Test, this process is relatively not time-costly. After it is done, you will see the result shown on the window. In our example, it reads

The confidence probability is 99 %

It means the test data is written by the same author for a probability of 99% (Fig. 8).

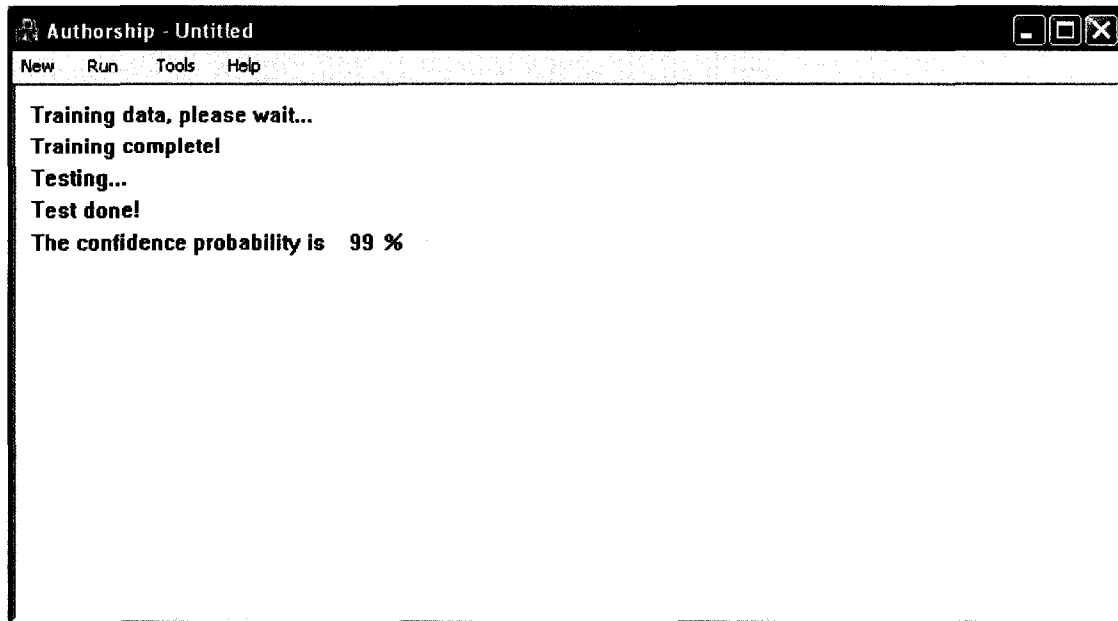


FIGURE 8. THE RESULT AFTER TEST

Some other files are created at the same time when running the program. These files record the intermediate results during the running of the program, or some copy of final results. Some results are actually not used, they are primarily created for reference of the data, or as a backup data for other possible future generalization.

The major files include

sample\_data.txt,

test\_data.txt,

pta

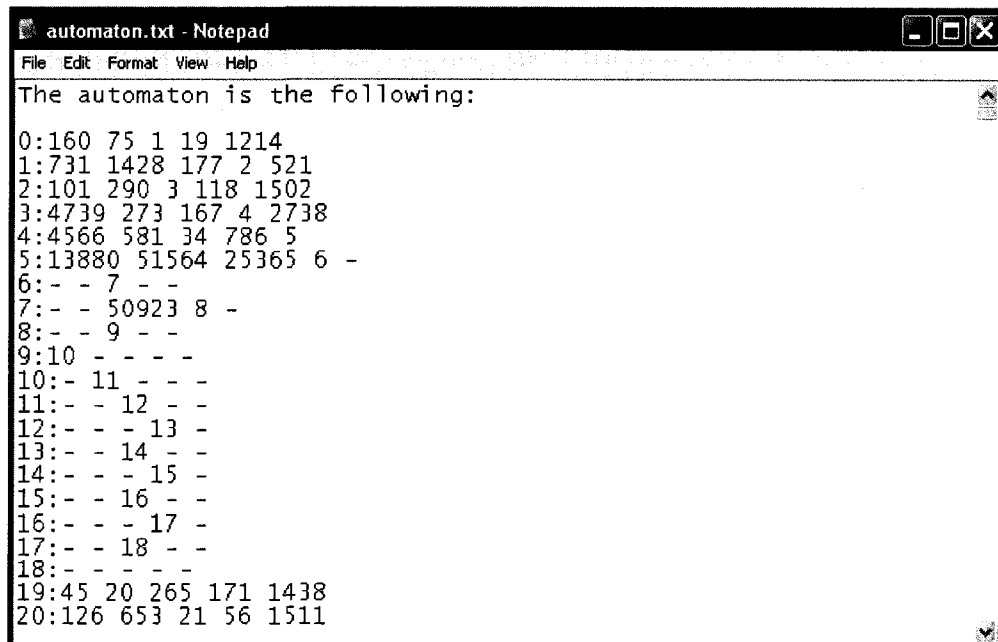
log.txt

automaton.txt

In these files, only pta is not a text file. It is usually opened by WordPad, because it is generally time-costly for notepad to open it, and the format in WordPad is better for browsing it.

Let me give an example here.

In the output directory there is a file named automaton.txt (Fig. 9),



```
automaton.txt - Notepad
File Edit Format View Help
The automaton is the following:
0:160 75 1 19 1214
1:731 1428 177 2 521
2:101 290 3 118 1502
3:4739 273 167 4 2738
4:4566 581 34 786 5
5:13880 51564 25365 6 -
6:- - 7 - -
7:- - 50923 8 -
8:- - 9 - -
9:10 - - - -
10:- 11 - - -
11:- - 12 - -
12:- - - 13 -
13:- - - 14 - -
14:- - - 15 -
15:- - - 16 - -
16:- - - 17 -
17:- - - 18 - -
18:- - - - -
19:45 20 265 171 1438
20:126 653 21 56 1511
```

FIGURE 9. THE FILE AUTOMATON.TXT

It gives the automaton in table format which is the way to store the SFA. To explain it, let's take a look at the last line

```
20: 126 653 21 56 1511
```

It represents the node or state labeled by 20. The first number 126 is a label of the node 126, and it is the first child of node 20, that is, transited by string 0. In the same way, by string 1, node 20 goes to node 653; by string 2, it goes to node 21, etc.

It is easy to guess that the - notation in the table means that the node has no corresponding child for that string. So for instance you will see node 6 has only one child node 7 led to by string 2, because node 7 is in the third position in all five ones (notice that the index for position always start with 0, hence the third one gives string 2).

## 7. RESULTS

In this section we present the results of the running of our program. The author we choose is J.K.Rowling and the writing sample is her book *Harry Potter and the Order of the Phoenix*. The test writings are her other three books:

*Book 1: Harry Potter and the Sorcerer's Stone*

*Book 2: Harry Potter and the Chamber of Secrets*

*Book 3: Harry Potter and the Prisoner of Azkaban*

and one book of Gabriel Garcia Marquez:

*Solitude: One Hundred Years Of Solitude*

In our program, we choose a sentence as a unit. One reason is that we already get good results with this choice. Another reason is that if we choose larger unit, the program will run longer. Since our results are good enough to distinguish authors, we don't bother to waste time to get similar results.



As we use one sentence as unit, the patterns we catch all have size smaller than one sentence. Any larger size pattern can be absorbed in the automaton. Now we give an example to illustrate this situation. The following paragraph consists of five sentences:

dabad.caba.baba.cabad.cabacaba.

One pattern is the repeat of string `aba` appeared in every sentence

dabad.caba.baba.cabad.cabacaba.

According to our method, the automaton (Fig. 10) is

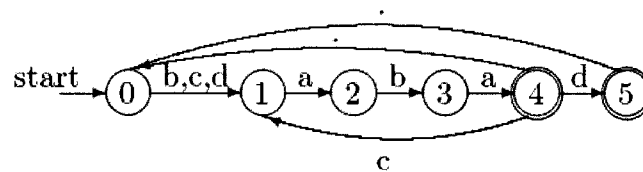


FIGURE 10. AUTOMATON FROM EXAMPLE

Note that there is another larger pattern `abad.caba` across sentences:

dabad.caba.baba.cabad.cabacaba.

This string can be accepted by the previous automaton. If we use two sentences as a unit, we can get a PTA, after merging, we will get the same automaton as above. However, it takes more time using this algorithm. So it is this technical reason we choose one sentence as a unit.

Next, we present our results. First of all, we use the PTA as our SFA, that is, we do not merge the states of the PTA. In this case, the PTA accepts exactly the set of strings of the sample data. The following table (table 2) gives the result:

TABLE 2. THE TABLE OF TESTING WITHOUT MERGING

	# total sentences	# accepted sentences	accepting probability
Book 1	6186	3904	0.631102
Book 2	6360	4007	0.630031
Book 3	8425	5554	0.659228
Solitude	5678	1751	0.308383

In this table, one can find a big gap of the accepting probabilities between the book of same author and the book of different author.

Next we fix the parameter  $\alpha=0.7$ . Then after merging we get an SFA as the writing pattern of the author. The results of accepting probability are given in the following table (table 3).

TABLE 3. THE RESULT OF TESTING WITH  $A=0.7$

	# total sentences	# accepted sentences	accepting probability
Book 1	6186	4285	0.692693
Book 2	6360	4390	0.690252
Book 3	8425	6021	0.714659
Solitude	5678	2079	0.36615

The accepting probabilities in this table are greater than the correspondence probabilities in the table before merging. This is because after merging, the new SFA can accept more strings than the one before merging. These new strings cannot be identified by the sample data.

We remark that if we take the parameter  $\alpha \leq 0.55$  in our program, then a lot of non-equivalent states will merge due to a large error used in the comparison of frequencies. The accepting probability is greater than 0.97 in all four books. This phenomenon does not imply that our method is not effective. It reminds us to pick the parameter appropriately to get the best result. In fact, our first table

of accepting probability obtained from the PTA (before merging) has already shown the difference between Book 1-3 and Solitude.

## 8. CONCLUSION

We believe that there is tremendous potential generalization of this method. For instance, one can change the size of the segment from one sentence to several sentences, or one can use a finer classification of the set of function words instead of part of speech. Even further, one can also include some type of content words into the sample data instead of the set of function words.

Another direction to refine the result is to combine this method with the traditional statistic methods. The author is working on this direction and obtained partial results.

The same method can also be applied to Microarray in biology. It is an interesting direction to work out various details and generalize this method combined with other tools.

## REFERENCES

- [1] T. Y. Lin, "Rough Patterns in Data-Rough Sets and Foundation of Intrusion Detection Systems," Journal of Foundation of Computer Science and Decision Support, Vol.18, No.3-4, pp. 225-241, 1993
  
- [2] T. Y. Lin, "Patterns in Numerical Data: Practical Approximations to Kolmogorov Complexity," SFDGrC, pp. 509-513, 1999
  
- [3] P.Baliga and T.Y.Lin, "Kolmogorov Complexity based Automata Modeling for Intrusion Detection," Proceeding of the 2005 IEEE International Conference on Granular Computing, Beijing, China, pp. 387-392, July 25-27, 2005
  
- [4] J.Grieve, "Quantitative Authorship Attribution: An evaluation of Techniques," Literary and Linguistic Computing, Vol. 22, No. 3, pp. 251-270, 2007
  
- [5] R.C.Carraso and J.Oncina, "Learning stochastic

- regular grammars by means of a state merging method," Proceedings of the 2nd International Colloquium on Grammatical Inference, Lecture Notes in Artificial Intelligence, pp.139-152, 1994
- [6] J.E.Hopcroft, R.Motwani and J.D.Ullman, Introduction to Automata Theory, Language, and Computation, Addison Wesley, 2001
- [7] T.Paek and R.Chandrasekar, "Windows as a Second Language: An Overview of the Jargon Project," Proceedings of the First International Conference on Augmented Cognition,2005
- [8] M.Koppel, S.Argamon and A.Shimoni, "Automatically categorizing written texts by author gender," Literary and Linguistic Computing,Vol.17, No.4, pp. 401-412, 2002
- [9] R.Sekar, M.Bendre, D.Dhurjati and P.Bollineni, "A Fast Automaton-Based Method for Detecting Anomalous Program Behaviors," Proceeding of

2001 IEEE Symposium on Security and  
Privacy, 2001

- [10] M.Young-Lai and F.Tompa, "*Stochastic Grammatical Inference of Text Database Structure*", Machine Learning, pp. 111-137, 2000
- [11] F.Mosteller and D.L.Wallace, Inference and disputed authorship: The Federalist, Reading, Mass, Addison-Wesley, 1964



## APPENDIX A: STOP WORD LIST

The stop word list is important to the program, there are different list online. The one we used in our program is downloaded from

<http://www.marlodge.supanet.com/museum/funcword.html>

To store the data into the program, we defined the following array

```
static const char funword[WordType][WordNumber][WordLength]=
{{"again","ago","almost","already","also","always","anywhere","back","e
lse","even","ever","everywhere","far","hence","here","hither","how","ho
wever","near","nearby","nearly","never","not","now","nowhere","often","
only","quite","rather","sometimes","somewhere","soon","still","then","t
hence","there","therefore","thither","thus","today","tomorrow","too","u
nderneath","very","when","whence","where","whither","why","yes","yester
day","yet"},
{"am","are","aren't","be","been","being","can","can't","could","couldn'
t","did","didn't","do","does","doesn't","doing","done","don't","get","g
```

ets", "getting", "got", "had", "hadn't", "has", "hasn't", "have", "haven't", "having", "he'd", "he'll", "he's", "i'd", "i'll", "i'm", "is", "i've", "isn't", "it's", "may", "might", "must", "mustn't", "ought", "oughtn't", "shall", "shan't", "she'd", "she'll", "she's", "should", "shouldn't", "that's", "they'd", "they'll", "they're", "was", "wasn't", "we'd", "we'll", "were", "we're", "weren't", "we've", "will", "won't", "would", "wouldn't", "you'd", "you'll", "you're", "you've"},

{"about", "above", "after", "along", "although", "among", "and", "around", "as", "at", "before", "below", "beneath", "beside", "between", "beyond", "but", "by", "down", "during", "except", "for", "from", "if", "in", "into", "near", "nor", "of", "off", "on", "or", "out", "over", "round", "since", "so", "than", "that", "though", "through", "till", "to", "towards", "under", "unless", "until", "up", "whereas", "while", "with", "within", "without"}, {"a", "all", "an", "another", "any", "anybody", "anything", "both", "each", "either", "enough", "every", "everybody", "everyone", "everything", "few", "fewer", "he", "her", "hers", "herself", "him", "himself", "his", "i", "it", "its", "itself", "less", "many", "me", "mine", "more", "most", "much", "my", "myself", "neither", "no", "nobody", "none", "noone", "nothing", "other", "others", "our", "ours", "ourselves", "she", "some", "somebody", "someone", "something", "such", "that", "the", "their", "theirs", "them", "themselves", "these", "they", "this", "those", "us", "we", "what", "which", "who", "whom", "whose", "you", "your", "yours", "yourself", "yourselves"},

{"billion", "billionth", "eight", "eighteen", "eighteenth", "eighth", "eightieth", "eighty", "eleven", "eleventh", "fifteen", "fifteenth", "fifth", "fiftieth", "fifty", "first", "five", "fortieth", "forty", "four", "fourteen", "fourteenth", "fourth", "hundred", "hundredth", "last", "million", "millionth", "next", "nine", "nineteen", "nineteenth", "ninetieth", "ninety", "ninth", "once", "one", "second", "seven", "seventeen", "seventeenth", "seventh", "seventieth", "

```
seventy", "six", "sixteen", "sixteenth", "sixth", "sixtieth", "sixty", "ten", "
tenth", "third", "thirteen", "thirteenth", "thirtieth", "thirty", "thousand",
"thousandth", "three", "thrice", "twelfth", "twelve", "twentieth", "twenty", "
twice", "two"}};
```

Notice that the enumerate type

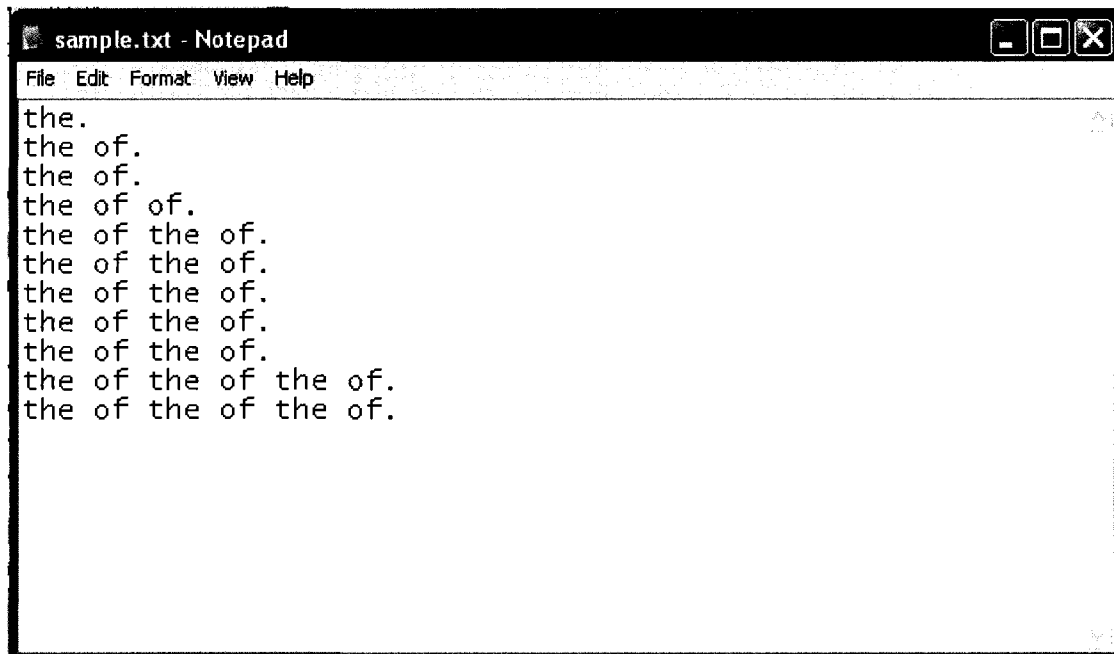
```
enum {Adv, Aux, Prep, Pron, Number}
```

stores the part of speech we are interested in.

## APPENDIX B: DETAILS OF HOW THE FUNCTIONS IN THE PROGRAM WORK

In this appendix we give an illuminating example which shows how the functions in our program work.

The sample text file (Fig. 11) is



```
sample.txt - Notepad
File Edit Format View Help
the.
the of.
the of.
the of of.
the of the of.
the of the of.
the of the of.
the of the of.
the of the of.
the of the of the of.
the of the of the of.
```

FIGURE 11. SAMPLE .TXT

You can think of these as 11 sentences containing the above stop words; we just ignore all content words.

Similarly, we have our test file for some unknown author. We also ignore all content words to avoid interrupting information.

The test file (Fig. 12) is the following:

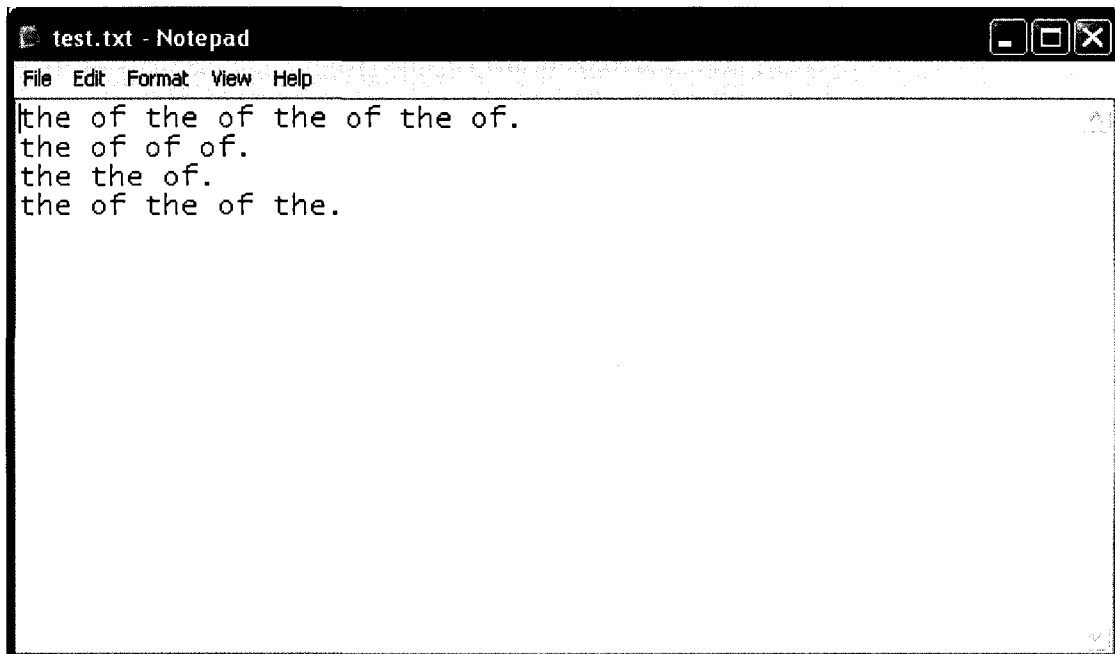


FIGURE 12. TEST.TXT

Notice that all sentences are different from the sentences in sample.txt. We potentially varied each sentence a little bit by adding a repetition, or by deleting a word, or partially repeat some part. We will see how this merging will give rise to new knowledge to identify these new sentences.

The following several figures collect the result of running the program for different parameter  $\alpha$ . You will see the importance of this parameter in the influence of the final result.

We first set  $\alpha=0.9$ , we expect to see a low confidence probability as a result, because the merge standard is high and few states are merged (Figs. 13, 14, 15, 16).

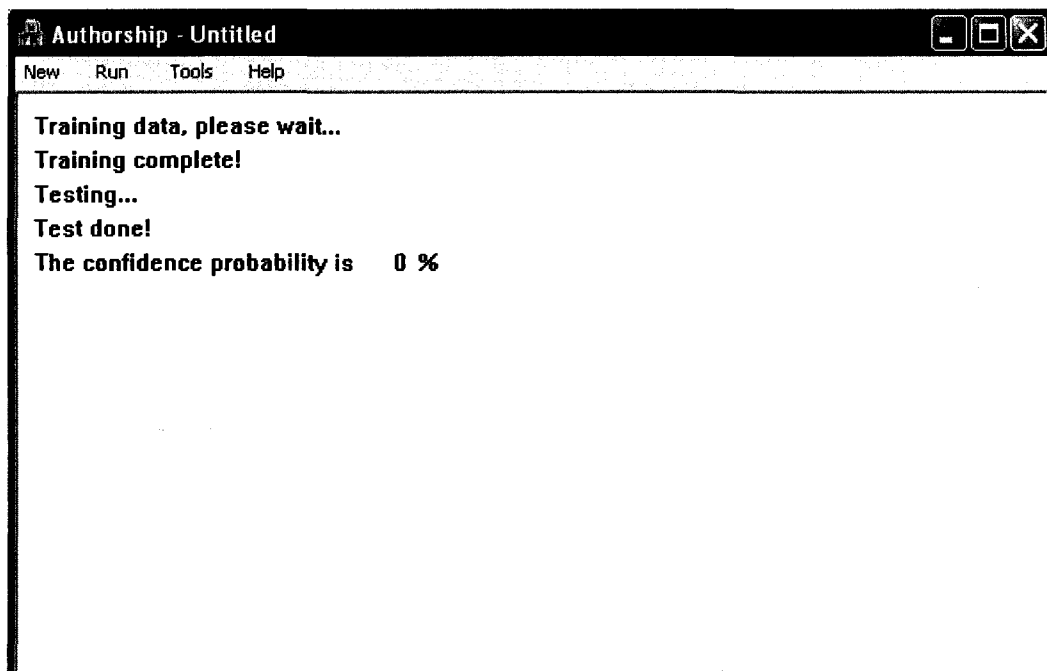


FIGURE 13.  $\alpha=0.9$

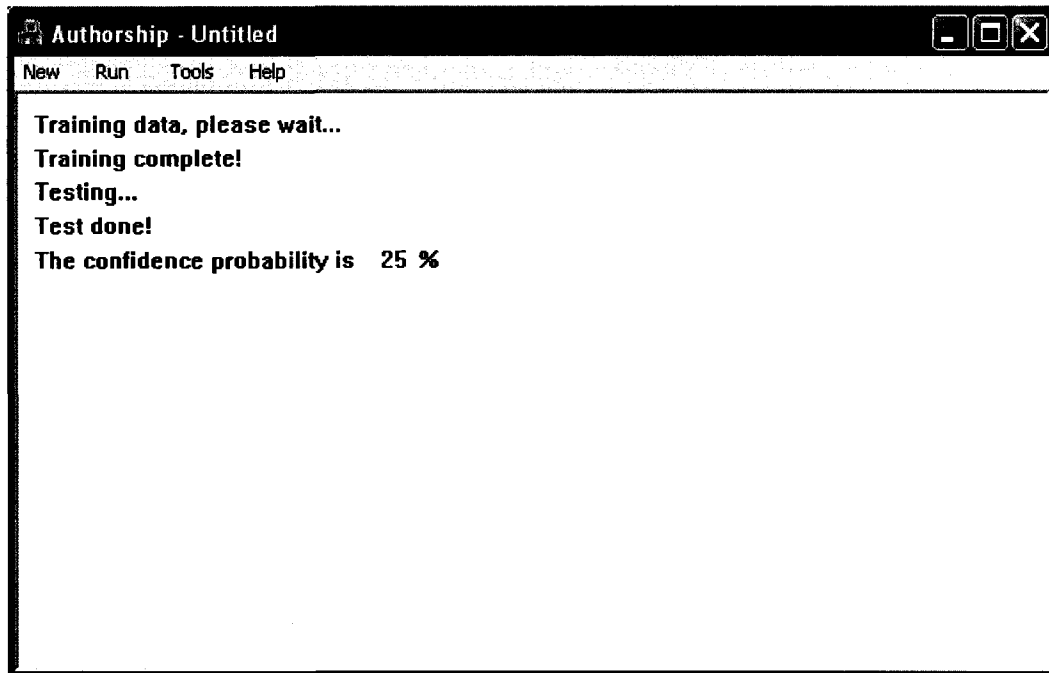


FIGURE 14.  $A=0.8$

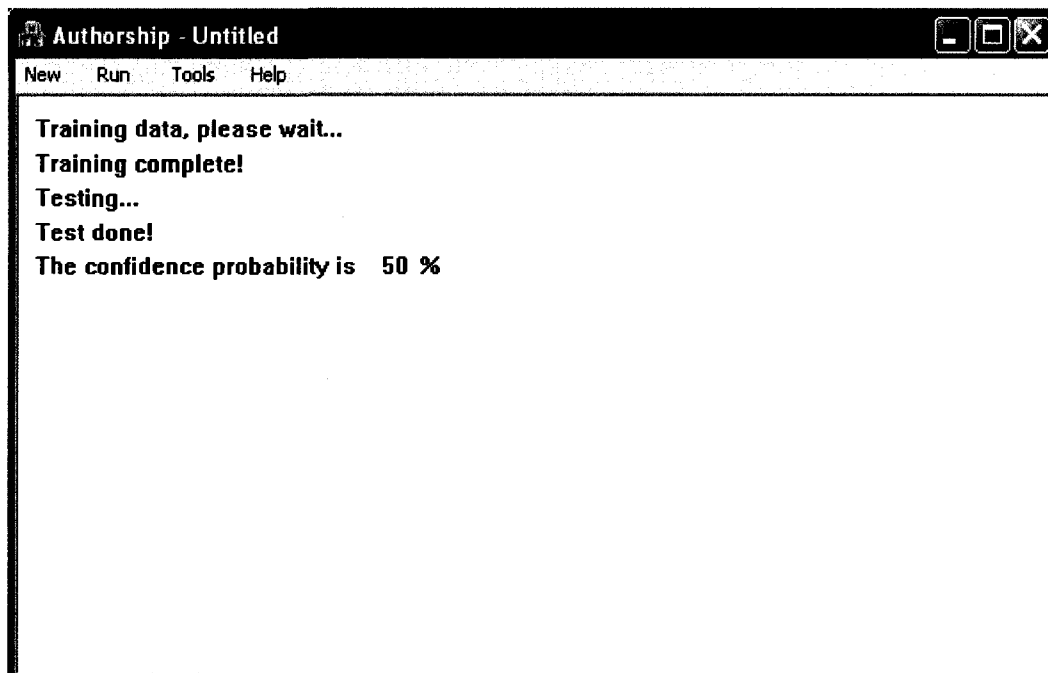


FIGURE 15.  $A=0.4$

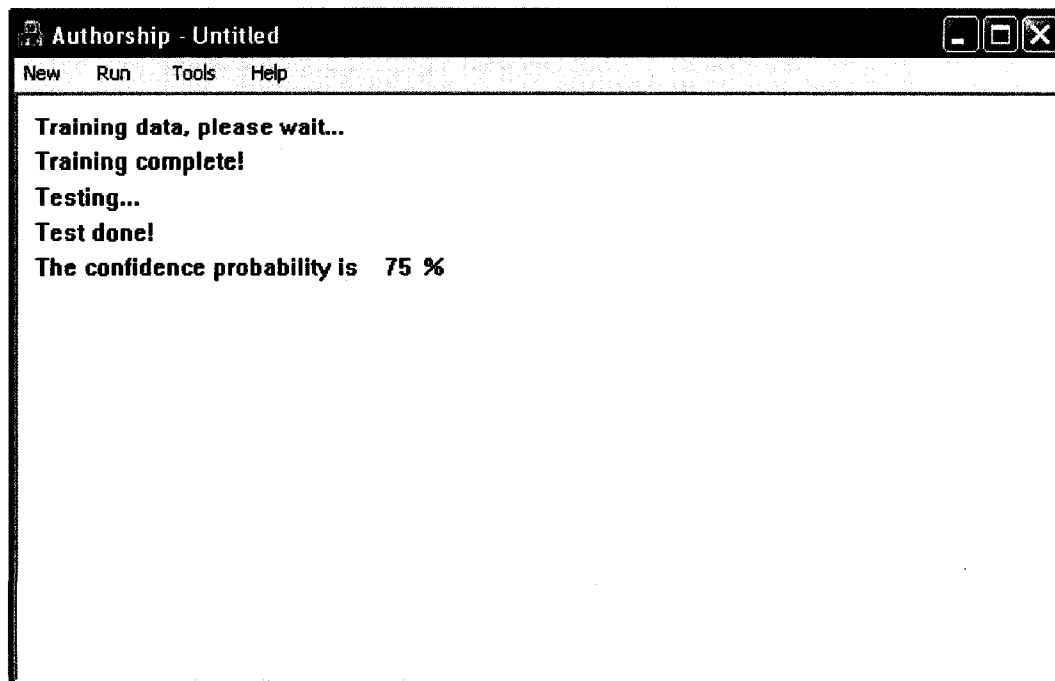


FIGURE 16.  $A=0.3$

To see what is happening, we take a look at the corresponding automaton we get stored in the file `automaton.txt` (Figs. 17, 18, 19, 20) in each setting of parameter  $\alpha$ .



```

0: - - - 1 -
1: - - 2 - -
2: - - 3 4 -
3: - - - - -
4: - - 5 - -
5: - - - 6 -
6: - - 7 - -
7: - - - - -

```

FIGURE 17. AUTOMATON FOR A=0.9

```

0: - - - 1 -
1: - - 2 - -
2: - - 3 4 -
3: - - - - -
4: - - 5 - -
5: - - - 4 -
6: =4
7: =5

```

FIGURE 18. AUTOMATON FOR A=0.8

```

0: - - - 1 -
1: - - 2 - -
2: - - 2 4 -
3: =2
4: - - 5 - -
5: - - - 4 -
6: =4
7: =5

```

FIGURE 19. AUTOMATON FOR A=0.4

```
0:- - - 1 -  
1:- - 2 - -  
2:- - 2 1 -  
3:=2  
4:=1  
5:=2  
6:=1  
7:=1
```

FIGURE 20. AUTOMATON FOR A=0.3

## APPENDIX C: TEST ENVIRONMENT AND PERFORMANCE

The result is gained by running program on:

Window XP professional SP2

2007C4U--- LENOVO THINKPAD T60

Intel CPU CORE DUO T2500 2 GHZ

2.5GB of RAM

Sample file:

<<Harry Potter and the Order of the Phoenix>>

total sentences: 17,214

total function words: 133,867

total words in the articles: 1,223,507

txt file size: 1,500KB

states in the automaton: 57741

total training time: <= 20 seconds

(I got different running time such as 15 or 16 or 18 or 20 seconds. It depends on whether my laptop is responding to other programs.)

The total training time includes reading file, extracting the function words, building the PTA and merging(dominating factor).The training part dominates the time since the testing part is much quicker.

Test Time(using confidence value ,  $\alpha=0.7$ ):

Harry Potter 1(6186 sentences), it takes 2 seconds to test.

Harry Potter 2(6560 sentences), it takes 2 seconds to test.

Harry Potter 3(8425 sentences), it takes 2 seconds to test.

One Hundred Year of Solitude (5678 sentences), it takes 3 seconds to test.

Remark: The running time is not proportional to the number of sentences, but proportional to the number of function words.