Authorship Identification of Source Codes

Chunxia Zhang[1], Sen Wang[1], Jiayu Wu[1], Zhendong Niu[2]

[1] School of Software, Beijing Institute of Technology, Beijing,China
[2] School of Computer Science, Beijing Institute of Technology, Beijing,China
{cxzhang,wangyuwangsen,2220160656,zniu}@bit.edu.cn

Abstract. Source code authorship identification is an issue of authorship identification from documents, and it is to identify authors of source codes or programs based on source code examples of programmers. The main applications of authorship identification of source codes include software intellectual property infringement, malicious code detection and software maintenance and update. This paper proposes an approach of constructing author profiles of programmers based on a logic model of continuous word-level n-gram and discrete word-level n-gram, and a multi-level context model about operations, loops, arrays and methods. Further, we employ the technique of sequential minimal optimization for support vector machine training to identify authorship of source codes. The advantage of author profiles in this paper can discover explicit and implicit personal programming preference patterns of and between keywords, identifiers, operators, statements, methods and classes. Experimental results on programs from two open source websites demonstrate that our approach achieves a high accuracy and outperforms the baseline methods.

Keywords: Authorship identification, source code, software forensics, discrete word-level n-gram, sequential minimal optimization

1 Introduction

Source code authorship identification is to identify authors of source codes or programs based on code examples of a given set of candidate programmers[1–6]. Authorship identification of source codes is an issue of authorship identification which aims to determine authors of texts such as literatures, articles, essays, emails, blogs, source codes and online forum messages[7, 8]. In addition, authorship identification of source codes is an important task in the field of software forensics, and this field is to analyze software source codes or executable codes in order for distinguishing authors of softwares or personalities of those authors[9, 10].

The wide applications of authorship identification technique of source codes include software intellectual property infringement, malicious code detection, and software maintenance and update[1, 3, 10–14]. First, software intellectual property infringement contains copyright or patent infringement. Authorship

identification of source codes can be used to settle ownership disputes of unauthorised source codes[1, 10, 12–14]. Second, malicious code detection is to detect computer viruses, computer worms, spyware and adware and so on[1, 3, 12, 14]. Authorship identification can help to decide authors or developers of malicious codes. Third, authorship identification of source codes can be utilized to identify authors of previous programs or program modules and track the authorship of program variations in the process of software maintenance and update[1, 12].

However, it is infeasible and time-consuming to recognize authors of source codes in a manual way[13, 15]. Hence, this paper focuses on how to identify authors of source codes or programs. The differences between authorship identification of source codes and authorship identification of natural language texts are given as follows[14, 16–18]:(1) a natural language is a kind of open and complicated language. However, a programming language in which source codes are written is a type of formal and restrictive language. (2) The flexibility of source codes is mainly embodied in the aspects of layout, style, structure and logic of programs. And features about layout, style, structure and logic of source codes usually rely on personal experiences and habits of programmers.

The major challenges of authorship identification of source codes include (1) how to extract features which are independent of functions or purposes of source codes, and specific names of identifiers such as variables and methods; (2) how to extract features which are relatively steady across different programs of the same programmer and in the evolving process of programming characteristics of developers[6, 18]; (3) how to extract features which can highlight distinctivenesses among different programmers[13, 15].

In this paper, we propose an approach of constructing author profiles of programmers based on a logic source code-oriented model of continuous word-level n-gram and discrete word-level n-gram, and a multi-level context model about operations, loops, arrays and methods. Further, we employ the technique of sequential minimal optimization(SMO) for support vector machine(SVM) training to identify authorship of source codes. Experimental results on programs from two open source websites demonstrate that our approach achieves a high accuracy and outperforms the baseline methods.

The principal contributions of this work are given as follows. (1) An approach of building author profiles of programmers based on a logic source code-oriented model of continuous word-level n-gram and discrete word-level n-gram, and a multi-level context model about operations, loops, arrays and methods is proposed in this paper. The author profiles capture explicit and implicit personal programming characteristics of different programmers on the using of keywords, identifers, operators, statements, methods and classes, and programming collation patterns between those different granularities of components of programming languages. Moreover, extracted features within author profiles are purpose-independent, and are not restricted by specific user-defined names of identifiers such as variables, methods, classes and interfaces. (2) This paper provides a promising technique to fulfill the task of source code authorship identification. Experimental results on two open source websites show that the identification

performance based on our author profiles of programmers by using SMO outperforms two baseline methods, is better than those of present features in related works and those of decision tree and random forest.

The rest of the paper is organized as follows. Section 2 introduces related works about authorship identification of source codes. Section 3 presents our source code authorship identification algorithm. Experimental results are given in section 4. Section 5 concludes the paper and discusses future works.

2 Related Works

The authorship identification techniques of source codes include ranking approaches and machine classifier approaches[14, 17]. Further, the latter kind of approaches can be classified into three types of methods: statistical analysis, machine learning, and similarity calculation methods[14, 17].

The representative works which used ranking approaches consist of works of Burrows et al.[13, 14, 19–22] and Frantzeskou et al.[16]. Burrows et al.[13, 14, 19–22] adopted an information retrieval technique to deal with the task of authorship identification of source codes. First, they converted programs into token sequences with operator tokens, keyword tokens, and white space tokens. Second, transformed token sequences into a token-level n-gram representation. Third, built an index of all programs and queried the index for a test program with unknown authors. Fourth, selected the authors of top-ranked programs as the authors of the test program. Frantzeskou et al.[16, 23] applied a method of Source Code Author Profiles with byte-level n-gram features to solve the problem of source code authorship identification. Here, an author profile is composed of the most frequent n-grams in training data of that author. To a test program with unknown authorship, they first computed the number of common n-grams between the test program and each author profile, and then chose the author with the most common n-grams as the author of that test program.

Krsul et al.[24], Ding et al.[2] and MacDonell et al.[12] utilized statistical analysis methods to recognize authorship of source codes. Krsul et al.[14, 24] extracted metrics which contain programming layout metrics, style metrics and structure metrics, and used a multiple discriminant analysis method to identify authorship of source codes. MacDonell et al.[12] calculated metrics about whitespace features, some specific character-level features and some keyword features, and employed a multiple discriminant analysis approach to address the issue of source code authorship identification. Furthermore, Ding et al.[2] adopted an canonical discriminant analysis technique to judge authorship of source codes.

The main works of machine learning approaches include the works of Stephen-MacDonell et al.[12], Kothari et al.[1] and Elenbogen et al.[25]. MacDonell et al.[12] used feed-forward neural networks and case-based reasoning to identify authorship of source codes. Kothari et al.[1] extracted programming style metrics and character-level n-gram features, and utilized the Bayes classifier and the voting feature intervals to determine authorship of source codes. In addition, Elenbogen et al.[25] computed style metrics including number of lines of

code, number of comments, average length of variable names, number of variables, and adopted an decision tree model to decide authorship of source codes.

Lange et al.[3] and Shevertalov et al.[26] employed the similarity calculation method to identify authorship of source codes. The two works computed distances between a program with unknown authors and programs of known authors, and used the nearest-neighbor method to judge authorship of codes.

3 An Authorship Identification Approach of Source Codes

We first give the definition of source code authorship identification.

Definition 1. Given a set P of programmers and their source code examples, the task of *source code authorship identification* is to determine which programmer in the set P wrote a program with unknown authors.

3.1 Overview of Our Approach

Our approach of authorship identification of source codes includes two phases: author profiles construction and author identification, as shown in Fig.1. The phase of author profiles construction is composed of four steps: (1) programming layout feature extraction, (2) programming style feature extraction, (3) programming structure feature extraction, and (4) programming logic feature extraction based on a continuous word-level n-gram model, a discrete word-level n-gram model and a character-level n-gram model. Furthermore, a method based on the sequential minimal optimization for SVM training is used to identify authorship of source codes or programs.

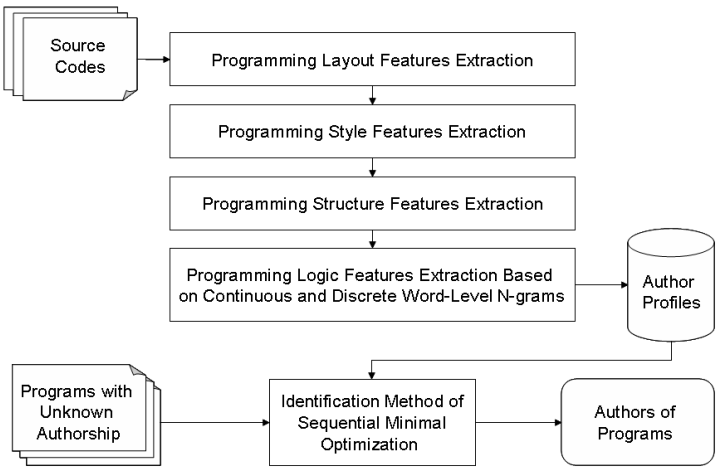

Fig. 1. The Process of Our Authorship Identification Approach of Source Codes.

3.2 Feature Extraction

Author profiles of programmers play an important role in the issue of source code authorship identification. Author profiles should have relative stability within a programmer and highlight distinctivenesses among programmers.

The features constituting author profiles can be divided into four categories: programming layout features, programming style features, programming structure features and programming logic features. Table 1 gives the sets of programming features which are used in our approach and other related works, while table 2 demonstrates the sets of programming features which are proposed in our approach. Table 1 and table 2 show the category, ID, feature focus and feature description. Here, the feature focus means the programming topic or entity which the feature is concerned with. In this paper, lines of a program are classified into four kinds of lines: code lines, comment lines, blank lines, and hybrid lines of code and comment. A hybrid line of code and comment means a line which contains a piece of code and a piece of comment. Moreover, non-comment lines of a program contain code lines and blank lines.

Programming Layout Feature Extraction . We will present how to extract programming layout features in this subsection.

Definition 2. *Programming layout features* are features which reflect the layout of a program or the arrangement of source codes and comments of a program.

The feature focuses of the programming layout features in table 1 and table 2 constitute a set {import, if statement, format of operator, format of loop, leading whitespaces of lines, percentage of blank lines}. For example, the feature focus of the feature fl_1 in table 2 is "import" which denotes the action of importing package. And fl_1 means whether there is at least a blank line between the last line includes the keyword "import" and the next line which contains codes.

The reason that features fl_1, fl_2 and fl_3 in table 2 are proposed in our approach is given as follows. The goal of fl_1 is to describe the layout characteristic between lines of importing packages and other code lines. Furthermore, fl_2 and fl_3 are to represent the arrangement of whitespaces within for loop statements. Actually, those three features are independent of purposes of programs and exhibit preferences of programmers to the layout about import and for loops. The set of programming layout features in our experiments includes layout features in table 1 and table 2, i.e., $\{fl_1, fl_2, fl_3, tl_1, tl_2, tl_3, tl_4\}$.

Programming Style Feature Extraction . This subsection will discuss programming style feature extraction from source codes.

Definition 3. *Programming style features* are ones which embody preferences of programmers to stylistic characteristics of programming such as variable names and variable lengths.

We will explain why the style features fs_1, fs_2, ..., fs_{10} in table 2 are designed in our approach. (1) fs_1, fs_2, fs_3 and fs_7 are to reflect personal traits

Table 1. Programming Features Used in Our Approach and Other Works

Category	ID	Feature Focus	Feature Description
Layout	tl_1	format of operator	Average number of whitespaces adjacent to operators
	tl_2	if statement	Is the frequency of left braces on the end of lines including "if(condition)" is greater than that of left braces on the next lines of lines including "if(condition)"
	tl_3	percentage of blank lines	Percentage of blank lines to all lines of a program
	tl_4	leading whitespaces of lines	Average number of leading whitespaces per line
Style	ts_1	number of comment lines	Is the number of hybrid lines is greater than that of comment lines
	ts_2	percentage of comments	Percentage of comment lines and hybrid lines to non-comment lines
	ts_3	variable length	Average number of characters per variable
	ts_4	frequency of for/while loop	Comparing the frequency of the keyword "for" with that of "while"
	ts_5	frequency of if/switch	Comparing the frequency of the keyword "if" with that of "switch"
	ts_6	percentage of static global variables	Percentage of the number of static global variables to non-comment lines of a program
	ts_7– ts_9	public, private, protected	Respective percentages of the keywords "public", "private" and "protected"
	ts_{10}	kinds of operators	The kinds of occurred operators
	ts_{11}	percentage of operators	Percentage of the number of occurred operators to non-comment lines
	ts_{12}– ts_{15}	percentage of methods	Respective percentages of four kinds of methods including int, char, String and void methods
	ts_{16}	percentage of methods	Percentage of methods to non-comment lines
	ts_{17}– ts_{19}	variable names	Do variable names include numbers, uppercase letters and lowercase letters, respectively
	ts_{20}	percentage of variables	Percentage of the number of variables to non-comment lines
	ts_{21}	number of variables	Sum of the numbers of all variables
	ts_{22}	go to	Is the statement "go to" used in a program
Structure	tr_1	average length of lines	Average number of characters per line
Logic	tg_1	character-level n-gram	Frequency of character-level n-grams

Table 2. Programming Features Proposed in Our Approach

Cate -gory	ID	Feature Focus	Feature focus proposed in our work	Metric proposed in our work	Feature Description
Lay -out	fl_1	import	√	√	Is there at least a blank line between the last line includes the keyword "import" and the next line which contains codes
	fl_2	format of for loop	√	√	Percentage of whitespaces between left and right parentheses of the form "for(...)" to all "for" loops
	fl_3	format of for loop	√	√	Is there at least a whitespace between left parentheses and right parentheses of the form "for(...)"
Style	fs_1	percentage of comments	×	√	Percentage of comment lines and hybrid lines to all lines of a program
	fs_2	percentage of keywords	×	√	Percentage of the number of keywords to non-comment lines of a program
	fs_3	percentage of loops	×	√	Percentage of loops of "for", "while" and "do-while" to non-comment lines
	fs_4	definition of loop variable	√	√	Percentage of loop variables in lines which include the keyword "for" and definitions of those variables to all "for" loops
	fs_5	two-dimen -sional array	√	√	Is a two-dimensional array used in a program
	fs_6	array subscript	√	√	Is a computational formula used within array subscripts in a program
	fs_7	addition operation	√	√	Percentage of the form like " i+=j" to total of the forms like " i=i+j " and " i+=j"
	fs_8	return statement	√	√	Does "return 0;" occur in a program
	fs_9	percentage of import	√	√	Percentage of lines which include the keyword "import" to all lines of a program
	fs_{10}	frequency of class and interface	×	√	Is the frequency of interfaces greater than that of classes
Struc -ture	fr_1	percentage of defined methods	×	√	Percentage of defined methods to non-comment lines
	fr_2	average length of comments	×	√	Average length of comment lines and hybrid lines
	fr_3	average length of methods	×	√	Average number of lines per method
Logic	fg_1	word-level n-gram	√	√	Frequency of word-level continuous n-grams of a program
	fg_2	word-level n-gram	√	√	Frequency of word-level discrete n-grams of a program

of programmers about the using of comments, keywords, loop statements, and statements of add operation. In particular, the feature fs_4 means that whether a developer frequently defines loop variables in for loop statements. (2) The essence of features fs_6 and fs_5 is to reflect the facts about how to define array subscripts for programmers and whether a two-dimensional array is used in their programs. (3) The feature fs_8 is to depict the habit that a developer write a statement "return 0;" on the end of a void method. (4) The aim of the features fs_9 and fs_{10} is to capture the characteristics of using frequencies of import, interface and class for programmers. In our experiments, we used a set of programming style features which is composed of style features in table 1 and table 2, i.e., $\{fs_1, fs_2, ..., fs_9, fs_{10}, ts_1, ts_2, ..., ts_{21}, ts_{22}\}$.

Programming Structure Feature Extraction .We will address how to extract programming structure features in this subsection.

Definition 4. *Programming structure features* are features which reflect the structure characteristics of programs such as average length of methods, which are usually associated with programmers' experiences.

The cause that the features fr_1, fr_2 and fr_3 are proposed in our method lie in that those three features embody the preferences of developers to the application of methods and comments. The set of programming structure features in our experiments contains fr_1, fr_2, fr_3 and tr_1 in table 1 and table 2.

Logic Feature Extraction Based on Continuous and Discrete Word-Level N-Gram Models . Fig.2 demonstrates a continuous word-level n-gram model of source codes and a discrete word-level n-gram model of source codes[27].

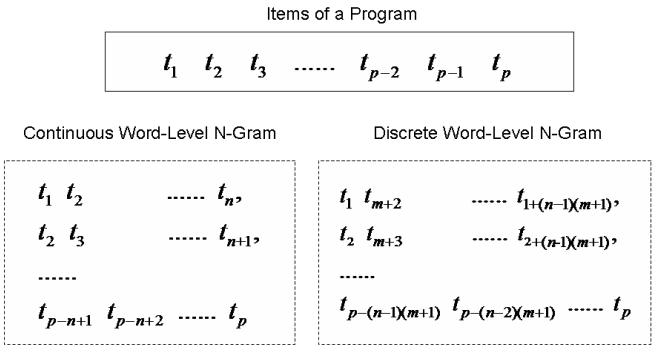

Fig. 2. Continuous and Discrete Word-Level N-Gram Models of Source Codes.

Definition 5. *A continuous word-level n-gram model of source codes* is a contiguous sequence of n items from a program. Here, items of programs are defined as any strings which are separated by whitespaces. Items can be keywords, operators, use-defined identifiers, punctuations and statements and so on. For a

sequence $t_1\ t_2$ ..... $t_{p-1}\ t_p$ of items of a program, we can obtain sequences of items shown in fig.2 according to that model.

For instance, to the following program (1), we can get sequences of 3 items, as shown in (2), based on the continuous word-level 3-gram model.

$$public \sqcup void \sqcup setUseGradient(\ \sqcup boolean \sqcup useGradientValue \sqcup\ ) \quad (1)$$

$$\begin{array}{ll} (1) & public\quad void\quad setUseGradient(\\ (2) & void\quad setUseGradient(\quad boolean \\ (3) & setUseGradient(\quad boolean\quad useGradientValue \\ (4) & boolean\quad useGradientValue\quad) \end{array} \quad (2)$$

Definition 6. For a sequence $t_1\ t_2$ ..... $t_{p-1}\ t_p$ of items of a program, *a discrete word-level n-gram model with m skip items* means a sequence of n items, like $t_k\ t_{k+m+1}$ ..... t_p ($1 \leq k \leq p-(n\text{-}1)(m+1)$), shown in fig.2.

As an illustration, to the program (1), we can acquire sequences of two items with one skip item, as shown in (3), according to the discrete word-level 2-gram model with one skip item.

$$\begin{array}{ll} (1) & public\quad setUseGradient(\\ (2) & void\quad boolean \\ (3) & setUseGradient(\quad useGradientValue \\ (4) & boolean\quad) \end{array} \quad (3)$$

We first extract a set of the top-k most frequent continuous word-level n-gram sequences, a set of the top-k discrete word-level n-gram sequences, and a set of character-level n-gram sequences from programs in the corpus. The feature fg_1, fg_2 and tg_1 of a program are occurring frequencies of sequences in those three sets, respectively.

The reason that we proposed features based on the continuous and discrete word-level n-gram models are analyzed as follows. (1) Extracted features based on the continuous and discrete word-level n-gram models reflect the using preferences of keywords, identifiers, operators and statements. (2) The continuous word-level n-gram model capture implicit programming patterns of programmers, that is, the collocation patterns between keywords and user-defined identifiers, between operators and identifiers. Here, user-defined identifers include variable names, methods names and class names and so on. However, it is difficult for the character-level n-gram model to achieve this goal. (3) Features founded on the discrete word-level n-gram model discover underlying collocation patterns of developers between keywords themselves, between user-defined identifies themselves, between keywords and operators. (4) Features based on the continuous and discrete word-level n-gram models are easy to be extracted from source codes, and they do not need any preprocessing. Therefore, those features based on those two models are irrelevant to the specific purposes of programs, can embody unconscious, relatively stable and personal programming characteristics of programmers.

3.3 An Authorship Identification Algorithm

Programs in the corpus have been represented as feature vectors after the process of feature extraction. Further, the problem of source code author identification was transformed into a multi-class classification problem in this paper. The classifier of sequential minimal optimization(SMO) for SVM training[28–30] was employed to determine authorship of source codes.

The sequential minimal optimization is designed to solve the quadratic programming problem, that is, the Lagrangian dual problem of SVM, as shown in (4)[28–30]:

$$\begin{aligned} &\max_{\alpha} \sum_{i=1}^{n} \alpha_i - \tfrac{1}{2} \sum_{i=1}^{n} \sum_{j=1}^{n} y_i y_j K(x_i, x_j) \alpha_i \alpha_j, \\ &s.t. \quad 0 \le \alpha \le \mathrm{C}, \quad for\ i = 1, 2, ..., n, \\ &\sum_{i=1}^{n} y_i \alpha_i = 0, \end{aligned} \tag{4}$$

where x_i is the input vector of a training source code example, y_i is a class label for x_i, n is the number of training examples, α_i are Lagrange multipliers, C is an hyperparameter, and $K(x_i, x_j)$ is the kernel function.

The advantages of the SMO approach are that it reduces the computation time and obtains a better scaling characteristic than the SVM training method[28–30]. Now we give our identification algorithm in Algorithm 1.

Algorithm 1 *Authorship identification from source codes*

Input: The source codes C of anonymous authors.
Output: the author of each program in C.

1: **for** $c_i \in C$, i=1, 2, ..., n **do**
2: Build the layout feature set $\{fl_1, fl_2, fl_3, tl_1, tl_2, tl_3, tl_4\}$.
3: Extract the style feature set $\{fs_1, fs_2, ..., fs_{10}, ts_1, ts_2, ..., ts_{22}\}$.
4: Construct the structure feature set $\{fr_1, fr_2, fr_3, tr_1\}$.
5: Extract the logic feature set $\{fg_1, fg_2, tg_1\}$.
6: Build the feature vector v of c_i.
7: **end for**
8: Employ the sequential minimal optimization(SMO) to classify the authorship of each program.

4 Experiments

4.1 Experimental Results

The two datasets of source codes in the java programming language are used in our experiments, which come from two source code websites(i.e., github.com,planet-source-code.com), respectively. The first dataset include 8000 programs of eight programmers (1000 programs per person), while the second dataset is composed

Table 3. The Identification Accuracy with DT, RF and SMO on Dataset 1

	B_1	B	F	$F \cup B_1$	$F \cup B$
	Features of the 1st baseline	Features of the 2nd baseline	Our features	Features in B_1 and our features	Features in B and our features
Decision Tree(%)	95.95	96.03	96.45	97.61	97.54
Random Forest(%)	97.74	97.66	98.08	98.00	98.05
SMO(%)	97.68	97.73	98.40	99.03	99.08

Table 4. The Identification Accuracy with DT, RF and SMO on Dataset 2

	B_1	B	F	$F \cup B_1$	$F \cup B$
Decision Tree(%)	71.31	70.52	72.71	73.31	72.51
Random Forest(%)	72.91	74.30	81.27	75.70	74.70
SMO(%)	72.31	75.50	80.28	82.47	83.47

of 502 programs of 53 programmers, and is a imblanced dataset. We utilize the 10-fold cross-validation to evaluate the performance of our approach.

Two baseline methods were implemented on the same two datasets for performance comparison. Those two method are ones which apply the decision tree based on the feature set B_1 and B to identify authorship of source codes, respectively. The formulas in (5) define seven sets $B_1, B_2, B, F_1, F_2, F_3, F$.

$$\begin{aligned} &B_1 = \{tg_1\},\ B_2 = \{tl_1, ..., tl_4, ts_1, ..., ts_{22}, tr_1\} \\ &B\ = B_1 \cup B_2 \\ &F_1 = \{fl_1, fl_2, fl_3, fs_1, ..., fs_{10}, fr_1, fr_2, fr_3\} \\ &F_2 = \{fg_1\},\ F_3 = \{fg_2\} \\ &F\ = F_1 \cup F_2 \cup F_3 \end{aligned} \tag{5}$$

Table 3 and table 4 show the identification accuracy of decision tree(DT), random forest(RF) and sequential minimal optimization(SMO) by using the features B_1, B, F, $F \cup B_1$ and $F \cup B$ on dataset 1 and dataset 2, respectively. B_1 is the feature set of character-level 6-grams. F_2 and F_3 are the feature set of continuous word-level 3-grams and the feature set of discrete word-level 2-grams with 1-skip item. The dimensions of B_1, F_2 and F_3 are all 2000.

The following facts can be seen from table 3 and table 4. (1) With DT, RF and SMO on dataset 1 and dataset 2, the accuracy of F, $F \cup B_1$ and $F \cup B$ is higher than those of B_1 and B. Hence, our proposed features which either are or are not integrated with feature sets B_1 and B of two baseline methods improve the accuracy of B_1 and B. (2) The feature set $F \cup B$ including our

Table 5. The Identification Accuracy on Dataset 1 by Using Single Feature Set

	B_1	B	B_2	F_1	F_2	F_3
	Features of the 1st baseline	Features of the 2nd baseline	Present features in table 1	Our features in F_1	Our features in F_2	Our features in F_3
Decision Tree(%)	95.95	96.03	62.44	54.00	96.60	96.58
Random Forest(%)	97.74	97.66	76.23	65.59	98.15	97.83
SMO(%)	97.68	97.73	54.90	48.04	98.13	98.06

Table 6. The Identification Accuracy on Dataset 2 by Using Single Feature Set

	B_1	B	B_2	F_1	F_2	F_3
Decision Tree(%)	71.31	70.52	51.20	45.82	71.12	69.32
Random Forest(%)	72.91	74.30	71.51	57.57	77.89	77.49
SMO(%)	72.31	75.50	51.79	24.30	74.30	80.28

proposed features and B achieves the highest accuracy(i.e., 99.08% and 83.47%) on dataset 1 and dataset 2 by utilizing the SMO technique. (3) SMO obtains higher accuracy than those of DT and RF on both dataset 1 and dataset 2 on three feature sets B, $F \cup B_1$ and $F \cup B$.

Table 5 and table 6 give the the identification accuracy of DT, RF and SMO on dataset 1 and dataset 2 by using feature sets B_1, B_2, B, F_1, F_2 and F_3, respectively. Experimental results in table 5 and table 6 show that the accuracy of F_2 and F_3 is higher than those of B_1, B_2 and B on dataset 1 by using DT, RF and SMO, while the performance of F_3 is higher than those of B_1, B_2 and B on dataset 2 by applying RF and SMO. Therefore, the fact demonstrate the validity of our proposed feature sets F_2 and F_3. Moreover, A combination of the feature set F_2 with SMO on dataset 1 gets the highest performance in table 5, while a combination of the feature set F_3 with SMO on dataset 2 obtains the highest performance in table 6.

4.2 Parameters Analysis

In addition, we evaluate the influence of different dimensions of F_2 and F_3. The dimensions of F_2 and F_3 are set as 1000, 2000, 3000, 4000 and 5000. Fig.3(a),(b) and Fig.4(a),(b) illustrate the accuracy curves of $F \cup B_1$ and $F \cup B$ by using DT, RF and SMO on dataset 1 and dataset 2, respectively. The curves in fig.3 and fig.4 indicate that the accuracy of SMO is higher than that of DT and RF by using $F \cup B_1$ or $F \cup B$ within all five cases on both dataset 1 and dataset 2.

As a whole, the performance with B and our proposed feature set F by using SMO on dataset 1 and dataset 2 achieves the highest accuracy.

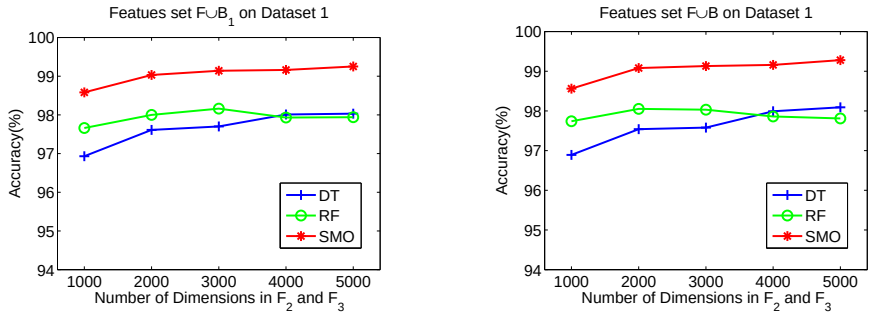

Fig. 3. The Identification Accuracy of Different Feature Dimensions on Dataset 1

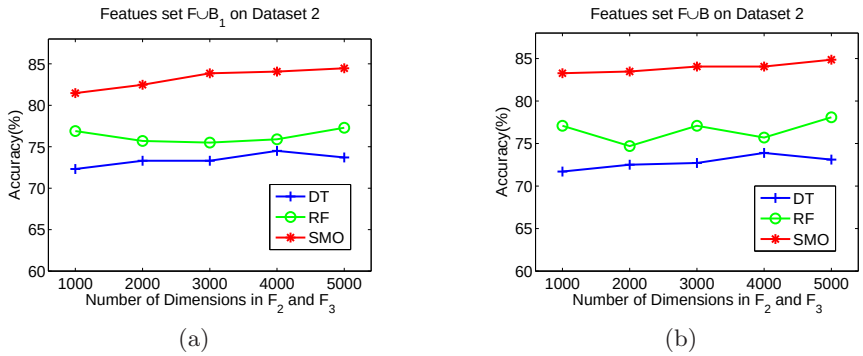

Fig. 4. The Identification Accuracy of Different Feature Dimensions on Dataset 2

5 Conclusion

The task of author identification of source codes is an significant issue in the field of author identification from on-line or off-line texts. Author identification has been widely used in many areas including computer forensics, network public opinion monitoring, intellectual property protection and information security[2, 6]. In this paper, an approach of building author profiles of programmers is proposed, which is based on a logic model of continuous word-level n-grams and discrete word-level n-grams, and a multi-level context model about operations,

loops, arrays and methods. Moreover, the classifier method of sequential minimal optimization for training SVM is applied to identify authors of source codes. The distinguish characteristic of author profiles not only represent personal preferences to using components of programming languages, i.e., keywords, identifers, operators, statements, methods and classes; but also express explicit and implicit personal combinative patterns among those components. Features within author profiles are independent of purposes of programs, are not constrained by specific user-defined names of variables, methods, classes and parameters and so on. Comparative experimental results on programs from two open source websites indicate that our constructed author profiles with SMO significantly increase the performance of source code author identification. In the future, we will introduce feature selection algorithms to the task of author identification of source code.

6 ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China (NO.61672098, NO.61272361).

References

1. Kothari, J., Shevertalov, M., Stehle, E., et al.: A Probabilistic Approach to Source Code Authorship Identification. In:4th International Conference on Information Technology, pp.243-248(2007)
2. Ding, H., Samadzadeh, M H.: Extraction of Java Program Fingerprints for Software Authorship Identification. Journal of Systems and Software 72(1), 49-57(2004)
3. Lange, R., Mancoridis, S.: Using Code Metric Histograms and Genetic Algorithms to Perform Author Identification for Software Forensics. In: 9th Annual Conference on Genetic and Evolutionary Computation, 2082-2089(2007)
4. Tennyson, M F.: On Improving Authorship Attribution of Source Code. In: International Conference on Digital Forensics and Cyber Crime, 58-65(2012)
5. Gray, A., Sallis, P., MacDonell, S.: Identified: A dictionary-based System for Extracting Source Code Metrics for Software Forensics. In: International Conference on Software Engineering: Education and Practice, 252-259(1998)
6. Zhang, C., Wu, X., Niu, Z., et al.: Authorship Identification from Unstructured Texts. Knowledge-Based Systems, 66, 99-111(2014)
7. Tennyson, M F., Mitropoulos, F J.: A Bayesian Ensemble Classifier for Source Code Authorship Attribution. In: International Conference on Similarity Search and Applications, 265-276(2014)
8. Stamatatos, E.: A Survey of Modern Authorship Attribution Methods. Journal of the American Society for information Science and Technology, 60(3),538-556(2009)
9. Spafford, E H., and Weeber, S A.: Software Forensics: Tracking Code to Its Authors, Computers and Security, 12,585-595(1993)
10. Software Forensics, `http://en.wikipedia.org/wiki/Software_forensics`.
11. Bandara U, Wijayarathna G.: Source Code Author Identification with Unsupervised Feature Learning. Pattern Recognition Letters, 34(3),330-334(2013)

12. MacDonell, S., Gray, M., MacLennan, G., Sallis, P.: Software Forensics for Discriminating between Program Authors Using Case-based Reasoning, Feed-forward Neural Networks and Multiple Discriminant Analysis. In: 6th International Conference on Neural Information Processing, 66-71(1999)
13. Burrows, S., Tahaghoghi, S M M.: Source Code Authorship Attribution Using N-grams. In: 12th Australasian Document Computing Symposium, 32-39(2007)
14. Burrows, S., Uitdenbogerd, A L., Turpin, A.: Comparing Techniques for Authorship Attribution of Source Code. Software: Practice and Experience 44(1),1-32(2014)
15. Bandara, U., Wijayarathna, G.: Deep Neural Networks for Source Code Author Identification. In: International Conference on Neural Information Processing, 368-375(2013)
16. Frantzeskou, G., Stamatatos, E., Gritzalis, S.: Supporting the Cybercrime Investigation Process: Effective Discrimination of Source Code Authors Based on Byte-level Information. In: 2nd International Conference on E-business and Telecommunication Networks, 163-173(2005)
17. Frantzeskou, G., Gritzalis, S., MacDonell, S G.: Source Code Authorship Analysis for Supporting the Cybercrime Investigation Process. In: 1st International Conference on E-business and Telecommunication Networks, 85-92(2004)
18. Krsul, I.: Authorship Analysis: Identifying the Author of a Program. Technical Report TR-94-030, Purdue University (1994)
19. Burrows, S., Tahaghoghi S M M., Zobel,J.: Efficient Plagiarism Detection for Large Code Repositories. Software-Practice and Experience 37(2),151-175(2007)
20. Burrows, S., Uitdenbogerd, A L., Turpin A.: Temporally Robust Software Features for Authorship Attribution. In: 33rd Annual International Computer Software and Applications Conference, 599-606(2009)
21. Burrows, S.: Source Code Authorship Attribution. PhD Thesis. RMIT University, Melbourne, Australia, 2010.
22. Burrows, S., Uitdenbogerd A L., Turpin, A.: Application of Information Retrieval Techniques for Source Code Authorship Attribution. In: 14th International Conference on Database Systems for Advanced Applications, 699-713(2009)
23. Frantzeskou, G., Stamatatos, E., Gritzalis, S., et al.: Identifying Authorship by Byte-level N-grams: the Source Code Author Profile(SCAP) method. International Journal of Digital Evidence 6(1), 1-18(2007)
24. Krsul, I, Spafford, E H.: Authorship Analysis: Identifying the Author of a Program. Computers Security 16(3), 233-257(1997)
25. Elenbogen, B S., Seliya, N.: Detecting Outsourced Student Programming Assignments. Journal of Computing Sciences in Colleges 23(3), 50-57(2008)
26. Shevertalov, M., Kothari, J., Stehle, E., Mancoridis, S.: On the Use of Discretised Source Code Metrics for Author Identification. In: 1st International Symposium on Search Based Software Engineering, 69-78(2009)
27. N-gram, `http://en.wikipedia.org/wiki/N-gram`.
28. Platt, J.: Sequential Minimal Optimization: A Fast Algorithm for Training Support Vector Machines, `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.55.560`(1998)
29. Sequential minimal optimization, `http://en.wikipedia.org/wiki/Sequential_minimal_optimization`.
30. Sequential minimal optimization, `http://blog.csdn.net/yclzh0522/article/details/6900707`.