Future Generation Computer Systems 95 (2019) 104–115

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

Code authorship identification using convolutional neural networks

Mohammed Abuhamad [a], Ji-su Rhim [a], Tamer AbuHmed [a,*], Sana Ullah [b], Sanggil Kang [a], DaeHun Nyang [a]

[a] Computer Engineering Department, INHA University, Incheon, South Korea
[b] Gyeongsang National University, Jinju, South Korea

HIGHLIGHTS

- We proposed three CNN-based code authorship identification systems.
- We explained various source code representations and our feature learning technique.
- We then fed the code representations into a CNN-based code authorship model.
- Large-scale code authorship process of different programming languages is conducted.
- Our technique identified a large number of programmers (1,600) with 99.5% accuracy.

ARTICLE INFO

Article history:
Received 30 June 2018
Received in revised form 17 October 2018
Accepted 15 December 2018
Available online 24 December 2018

Keywords:
Code authorship identification
Program features privacy
Convolutional neural network
Deep learning identification
Software forensics and security

ABSTRACT

Although source code authorship identification creates a privacy threat for many open source contributors, it is an important topic for the forensics field and enables many successful forensic applications, including ghostwriting detection, copyright dispute settlements, and other code analysis applications. This work proposes a convolutional neural network (CNN) based code authorship identification system. Our proposed system exploits term frequency-inverse document frequency, word embedding modeling, and feature learning techniques for code representation. This representation is then fed into a CNN-based code authorship identification model to identify the code's author. Evaluation results from using our approach on data from Google Code Jam demonstrate an identification accuracy of up to 99.4% with 150 candidate programmers, and 96.2% with 1,600 programmers. The evaluation of our approach also shows high accuracy for programmers identification over real-world code samples from 1987 public repositories on GitHub with 95% accuracy for 745 C programmers and 97% for the C++ programmers. These results indicate that the proposed approaches are not language-specific techniques and can identify programmers of different programming languages.

1. Introduction

Recently, the code authorship identification task has gained increased attention in the research community [1] due to its importance in software forensics. Code authorship identification is the process of identifying programmers based on their distinctive programming styles. Style is based on various factors, such as the programmer's preferences in the way to write code, naming of the variables, programming proficiency and experience, and the thinking process to solve any programming task. All of these factors help to extract specific features from a given piece of a programmer's code to enable the authorship identification process by assigning each piece to the programmer who wrote it. Thus, the advancements in this field could assist in several aspects of software forensics, such as software authorship disputes [2], code integrity investigations [3], code plagiarism detection [4], and copyright infringement [5]. Moreover, code authorship identification can be used to identify programmers of malicious code.

The success of code authorship identification depends on effective features extraction process that captures the distinctive characteristics of programmers' coding styles. This process is challenging, since the "coding style" of a programmer could change when working in environments or when following certain software engineering paradigms [6]. Being able to extract such features would enable accurate code authorship identification by assigning programmers to the input source code samples. This work investigates the capabilities of a convolutional neural network (CNN) to solve the code authorship identification problem. CNN shows a remarkable ability to extract robust features by applying layers

* Corresponding author.
E-mail addresses: abuhamad@seclab.inha.ac.kr (M. Abuhamad), rhimjisu@inha.edu (J.-s. Rhim), tamer@inha.ac.kr (T. AbuHmed), sanajcs@gmail.com (S. Ullah), sgkang@inha.ac.kr (S. Kang), nyang@inha.ac.kr (D. Nyang).

https://doi.org/10.1016/j.future.2018.12.038
0167-739X/

of convolving filters to local features [7]. These robust features incredibly help in the classification process. Several works [8,9] have shown that CNN capabilities are not limited to computer vision applications, since CNN models have shown interesting results in natural language processing (NLP) tasks. Applying a CNN to NLP tasks is well-explored in the literature [8]. CNNs have achieved remarkable results in several NLP tasks, such as sentence modeling [10], search query retrieval [11], and semantic parsing [12], among others [13]. However, to the best of our knowledge, no previous work has applied a CNN to structured codes for the purpose of code authorship attribution. This work explores the applicability of CNNs to identify programmers based on code samples. In this work, we present different variations of CNN models trained using different code representations. Moreover, we explore the adequacy of different source code representations, as well as different CNN-based deep learning architectures in the performance of the code authorship identification task.

Summary of Contributions. We summarize the main contributions of this work in multiple directions as follows.

- We designed feature learning and extraction techniques using CNN-based deep learning architectures. In this phase, complete or incomplete source code is given as input to a CNN-based deep learning architecture to generate high-quality and distinctive code authorship attributes that allow effective code authorship identification.
- Prior to the process of feeding the CNN with source code, we represent source code samples using two well-known approaches for textual data analysis [14–16]. The first approach is to apply term frequency-inverse document frequency (TF-IDF), which reflects how important a word is to a document in a collection or corpus. Thus, our approach does not require any prior information on specific programming languages. As a result, our approach is more resilient to language specifics and to the number of code files available per author. The second approach is to apply dense representations of the source code using word embedding representation, which allows words with similar meaning in the corpus to have a similar representation.
- We conducted a large-scale code authorship identification process with three different programming languages (C++, Java, and Python) and demonstrated that our technique can handle identification of a large number of programmers (1600) while maintaining high accuracy (99.50%).

Organization. The remainder of the paper is structured as follows. We review the related work in Section 2. We introduce the theoretical background required for understanding our work in Section 3. In Section 4 we present our CNN-based approaches to source code authorship identification. We proceed with a detailed overview of the experimental results from our approaches in Section 5. Finally, we provide our conclusion in Section 7.

2. Related work

Authorship attribution at the document level began as early as the late of 19th century with the first attempts to quantify author writing style. However, it was only recently that researchers started to investigate authorship attribution for software writers.

In the literature of software authorship attribution, most of the works can be broadly divided into either single or multi-author identification [17,18], for each sample of code. For both approaches, the code sample can be source code written using a programming language or a binary code generated from a compilation process.

Unlike binary code identification techniques, which assume only executable machine code is available [19,20,17], we review techniques related to source code authorship identification that assume the source code of the program is available, and that each code sample is written by a single programmer.

Most of the literature on code authorship identification attempted to solve the problem in two main steps: feature extraction and authorship identification. The feature extraction step is considered the labor-intensive part, where manually handcrafted features are extracted from code samples. These features are collected based on the fact that programmers each have their own quantitative and qualitative features to represent each programmer's coding style. Examples are layout (spacing, indentation, and boarding characters, etc.), style (variable naming, choice of statements, comments, etc.) and environment (computer platform, programming language, compiler, text editor, etc.). Following that, most previous work on code authorship identification adopted either statistical analysis techniques, machine learning classification approaches, or ranking approaches that are based on similarity measurements in order to do authorship identification for code samples [21]. In particular, Krsul et al. were among the first to explore the possibility of identifying the author of a program by examining handcrafted programming style characteristics [22]. In their work, they adopted a statistical analysis technique called multivariate discriminant analysis to classify code samples of 29 programmers. Burrows [14] and Frantzeskou et al. [23] extracted n-grams from source codes to establish profiles of authors' styles in order to rank the most likely author of any given source code. Burrows and Frantzeskou approach did not achieve high accuracy as the number of authors increased. Thus, Burrows et al. [24] extended the work on code authorship identification by including more handcrafted stylometric features to the n-grams' representation of the code, and slightly improved the earlier results. Frantzeskou and colleagues [5] introduced a different approach based on byte-level n-gram profiles in order to represent a source code author's style. Recent work done by Caliskan-Islam et al. [1] proposed a technique to identify the author of source code using manually crafted features from the code's abstract syntax tree. Their work modeled source code authorship attribution as a machine learning problem, and fed the classifier with features extracted from the abstract syntax tree to classify the code's author. Using deep learning techniques, Abuhamad et al. [25] adopted recurrent neural network to extract robust authorship attributions from code samples represented as one-step sequences using TF-IDF. Using the deep authorship attributions to construct a random forest classifier, the authors demonstrated scalable capabilities for their approach. Table 1 summarizes most of the related work for code authorship identification in terms of the number of authors, achieved identification accuracy, and the programming languages used for programmer identification. Despite the interesting results previous works demonstrated, several shortcomings are still to be handled. First, most related works utilize handcrafted feature set using extensive feature extraction process that generates sparse vector representation of code authorship attribution. This requires further feature evaluation and selection process. Even though the selected features might enable high identification accuracy in a small suspect set, handling a large-scale set of programmers (i.e. thousands of programmers) can be a struggle. We propose a CNN approach to generate compact and robust code authorship attribution that enable large-scale code authorship identification. Second, most related works use feature extraction process that is language-dependent in a sense that extracted features perform better for one language than for another. In this work, we evaluate our approach in three programming languages and demonstrate its effectiveness in handling different languages.

Another body of work that is related to our work is where CNN models are applied in NLP identification tasks. These works are

Table 1
A summary of the related work.

Reference	#Authors	Accuracy	Programming languages
Ivan Krsul [22]	29	73.00%	C
MacDonell et al. [34]	7	88.00%	C, C++
Ding and Samadzadeh [35]	46	62.70%	Java
Frantzeskou et al. [16]	30	96.90%	C++
Lange and Mancoridis [36]	20	55.00%	Java
Elenbogen et al. [37]	12	74.70%	C++
Steven Burrows et al. [6]	10	76.78%	C
Steven Burrows et al. [21]	100	80.37%	C, C++
Caliskan-Islam et al. [1]	1600	92.83%	C++, Python
Alsulami et al. [38]	70	88.86%	C++
Meng et al. [17]	284	65.00%	C, C++
Dauber et. al. [18]	106	99.00%	C++

growing at an accelerated pace due to the remarkable results of CNN models. The CNN has achieved good performance on various datasets mostly for topic categorization and sentiment analysis tasks [26]. Adopting the same architecture achieved good results on a text categorization task by adding an additional layer for semantic clustering [27]. A different CNN architecture was proposed for sentence classification for the purpose of sentiment analysis [10]. Johnson et al. introduced an interesting CNN architecture that does not require pre-trained word representations [28], such as Word2Vec [29] or Stanford University's Global Vectors for Word Representation GloVe [30]. The architecture applies convolving filters directly to one-hot vectors as the initial representation. The work also introduced some enhancements to the initial representation of input data in order to reduce the number of network parameters. An extension of the work was proposed that uses unsupervised "region embedding" [31]. Another interesting application of CNN models was presented in the relation classification task [32] and [33]. We find the application of CNN models to NLP tasks prompts an encouraging direction for our research. Applying a CNN to a given task requires choosing many hyperparameters and strategies, including input representation, the model architecture, and the training procedure. We explain our approach in details in Section 4.

3. Theoretical background

In this section, we introduce the methods used for code representation and convolution neural networks. For the readability purpose, we have summarized the used notations in Table 2.

3.1. Data representation

In most NLP tasks, textual data (i.e., sentences or documents) are represented as a matrix where the rows are the representation of tokens, words or characters. These rows, i.e., vector representations of words, are also called word embeddings. Word embeddings are usually low-dimensional representations learned from a given corpus using a frequency-based approach or prediction-based approach. The frequency-based approaches include methods based on TF-IDF, co-occurrence representation, and variations of both. The prediction-based approaches include methods such as Word2Vec [29], GloVe [30], or word embeddings learned as part of a deep learning model. The method for learning word embeddings as part of the deep learning model starts by representing words or terms by one-hot vectors that highlight the index of a given term/word in the entire corpus vocabulary. Using the one-hot vectors as input, the deep learning models learn the embeddings for terms and documents as part of their training process.

Using the TF-IDF scheme simplifies the model architecture since code files are represented as one-step fixed-sized sequences calculated in a separate stage before the training process. Alternatively, using word embedding scheme as an integral part of the training process generates a compact and optimized representation that preserve the order of used terms in the code files. This makes word embeddings more desirable to capture the code features when representing short pieces of textual contents, i.e. short pieces of code. This work provides some insights for using different approaches for code representation (i.e., word embeddings and TF-IDF) considering their impact on the performance of the proposed code authorship identification system.

Table 2
Summary of the notations.

Notation	Definition
t	Term
d	Document
$\mathcal{D}$	Corpus of documents
$\mathfrak{D}$	Corpus vocabulary
$\mathrm{TF}(t, d)$	Term frequency of t in d
$\mathrm{DF}(t, \mathcal{D})$	Number of documents with t in $\mathcal{D}$
$\mathrm{IDF}(t, \mathcal{D})$	Inverse document frequency for t in $\mathcal{D}$
$\mathrm{TF\text{-}IDF}(t, d, \mathcal{D})$	TF-IDF scores of d for all t in $\mathcal{D}$
$\cup$	Feature selection operation
x	Model input
L_{WE}	Word embedding layer
W	Model weights of a given layer
b	Model bias
f	Activation function
$pool, max$	Max-pool operation
c	Feature map of given layer
$conv_i$	Feature map of the ith CNN layer
s	Pooled feature map
$\oplus$	Concatenation operation
$\odot$	Matrix multiplication operation
θ	Model parameters
$loss(\theta, D)$	The softmax cross-entropy loss given θ for dataset D
$Reg(\theta)$	$L2$ regularization
λ	Regularization strength

3.1.1. Word embedding representation

The natural way to apply word embedding would be by using Word2Vec or GloVe since there are available tools (and even pre-trained models) to generate accurate word embeddings. However, handling code samples is different from handling natural language due to the inherent inflexibility of the written code expressions established by the syntax rules of the compilers. Thus, we use a variation of one NPL model [13] that was adopted in other works [39,26], for text data representation. The main purpose of using that model's variation is to learn word embeddings that most relate to the task at hand [13]. The words or terms are fed into the model as indices taken from a finite corpus vocabulary, $\mathfrak{D}$, where they are mapped with feature vectors by the first layer of the network using a lookup table operation. For a given task, relevant representation is learned via backpropagation.

More formally, $\forall\, t \in \mathfrak{D}$, there exists an internal d-dimensional representation extracted from the word embedding layer L_{WE}. For a given t:

$$L_{WE}(t) = \langle W \rangle_t^1$$

, where $W \in \mathbb{R}^{d \times |\mathfrak{D}|}$ is the L_{WE} parameters to be learned, $\langle W \rangle_t^1$ is the ith row $\in \mathbb{R}^d$ of W and d is the dimension of word representations.

A sequence of words, say a piece of code, with n terms can be represented by extracting the representation of every term in the sequence to form a representation matrix

$$L_{WE}([t]_1^n) = (\langle W \rangle_{t_1}^1 \quad \langle W \rangle_{t_2}^1 \quad \dots \quad \langle W \rangle_{t_n}^1).$$

This matrix can be the input representation for the following layers of the model. The weights W of L_{WE} are then included in the training process so that the generated word representations are optimized base on the overall evaluation of the model.

3.1.2. TF-IDF representation

The TF-IDF method is a common tool for representing textual data in NLP and data-mining tasks. Using TF-IDF as an initial representation for code files is motivated by its wide ranging applications on processing textual data. Frequency-based representation for terms and *n-grams* is commonly used in information retrieval, and has been adopted elsewhere for code authorship identification [14–16]. TF-IDF features describe an author's preferences for using certain terms, or his/her preference for specific commands, data types, and libraries.

TF-IDF is a weighting scheme that provides representations for documents based on the weights of terms. These weights reflect the importance of terms to a certain document in a corpus, which can be expressed by the frequency of terms in the document normalized by terms popularity in the corpus. It is straightforward to consider term frequency as indication for term importance in a certain document. However, overcoming the effects of popular terms in a corpus dictates normalizing term frequency by term popularity (i.e., the proportion of documents in which it appears). In TF-IDF, a term t in a file d of a corpus $\mathcal{D}$ is assigned a weight using the formula

$$\text{TF-IDF}(t, d, \mathcal{D}) = \text{TF}(t, d) \times \text{IDF}(t, \mathcal{D}),$$

where $\text{TF}(t, d)$ is the term frequency (TF) of t in d and

$$\text{IDF}(t, \mathcal{D}) = \log(|\mathcal{D}|/\text{DF}(t, \mathcal{D})) + 1,$$

where $|\mathcal{D}|$ is the number of documents in $\mathcal{D}$ and $\text{DF}(t, \mathcal{D})$ is the number of documents containing the term t.

Intuitively, programmers develop tendency to use methods and variable naming strategies to make their code recognizable for future maintenance. This intuition is emphasized in our preliminary experiment, which shows that the distribution of TF-IDF values of used terms is distinguishable among programmers. Moreover, we observed that the TF-IDF scores of reserved keywords from the used language are among the top extracted features. For example, some programmers used the 'cout' command, a command for printing out a message in C++, more frequent than others when solving programming problems Google Code Jam. Thus, the frequent use of specific commands can be considered as part of the coding style that indicates a programmer's preference in using the language to achieve the programming goal.

3.2. Convolutional neural networks

The convolutional neural network is a category of neural network that requires minimal preprocessing efforts. The CNN was inspired by the biological process that occurs in the animal visual cortex, where cortical neurons are restricted to handle responses from separate regions of the visual field. CNN adopts similar concept in the sense of using convolving filters (i.e., weights or neurons) to handle local regions within the visual field, usually called as receptive fields. Filters of different receptive fields partially overlap to cover the entire input data presented as the visual field. Through an optimization process, filters are optimized to transform the data in a way that enables performing a certain task, e.g., image classification, object localization or image description. Initially, the CNN was designed to process visual imagery. However, several applications have shown the applicability of the CNN to natural language processing tasks [26,27,10]. The structure of the CNN follows the traditional structure of neural networks, where inputs are mapped to outputs through nonlinear operations involving a large number of parameters within a number of hidden layers. The main alteration on the traditional structure is that the CNN includes a number of convolutional layers as part of the operation. Convolutional layers aim to extract features from local regions within the input by using convolving filters. Each filter is applied to regions across the input dimensions by computing the dot products of the filter and the region contents at all positions. The output of the convolution process is usually referred to as feature map c, which will be passed to the following layer by applying a nonlinear function. A pooling process is usually applied after single or multiple convolutional layers to aggregate information within regions in the feature map. For a region, R, another feature map, s, can be generated as: $s_j = pool(f(c_i)) \quad \forall i \in R_j$, where *pool* is a function that selects one element out of the entries of the region, R_j. Among different choices, max-pooling is commonly used, where the maximum value is selected. The resultant aggregated feature map, s, is more robust and invariant to changes in the input data. The convolution and pooling processes can be repeated a number of times by including more convolutional layers. Using multiple convolutional layers enables generating features that are less sensitive to local transformations in the input [40].

4. CNN-based code authorship identification systems

In this section, we introduce our approaches for the code authorship identification problem. Since this work adopts a close-world assumption, only programmers included in the dataset can be identified. This work assume that a programmer develops a distinct coding style preserved in her/his code samples. This coding style can be recognized and identified given sufficient number of samples. All the proposed solutions have two main phases: code presentation and CNN architecture. We briefly explain those two phases in the following and in more details in the subsequent sections.

Code Presentation. In the first phase, we start by adopting several methods for optimizing the representation of documents, such as eliminating stop words, normalizing representations, and removing indistinct features. Then, a straightforward mechanism is used to represent source code files based on a weighting scheme commonly used in information retrieval. In our implementation, we examined two methods namely, word embedding and TF-IDF.

CNN Architecture. Following the first phase, this phase contains a CNN based architecture to learn and refine more distinctive features of authorship from less distinctive ones. In the last layer of the architecture, a softmax classifier is used to classify authors.

4.1. CNN architecture with word embedding approach

Let a code sample, x, be represented as an input sequence of terms, where $t_i \in \mathbb{R}^d$ is the term representation of the ith term in the sequence; then, the input to the CNN model becomes $x = x_{t_1:t_n} = t_1 \oplus t_2 \oplus \cdots \oplus t_n$, where the operation $\oplus$ is a concatenation process. Since code samples have different lengths, sequences are padded to a predefined length, n.

The input code sample is then subjected to a convolution process by applying a filter, $w \in \mathbb{R}^{h \times d}$, to h terms. For the h terms, the convolution process generates a feature map, $c_i = f(w \odot x_{i:i+h-1} + b)$, where $\odot$ is a matrix multiplication operation, $b \in \mathbb{R}$ is a bias, and f is a nonlinear function. Sliding over the entire sequence of terms included in a code sample, $(x_{1:h}, x_{2:h+1}, \ldots, x_{n-h+1:n})$, a feature map, $c = c_1, c_2, \ldots, c_{n-h+1}$ is generated where $c \in \mathbb{R}^{n-h+1}$.

Following the convolution process, a max-pooling process is applied to generate another feature map $s = s_1, s_2, \ldots, s_{n-h+1}$, where, $s_j = max(c_i)$. Applying the max-pooling operation is equivalent to selecting the most important feature that has the highest value within the window of the pooling process. This process is used for one CNN layer with one filter size and can be repeated for multiple CNN layers with different filter sizes.

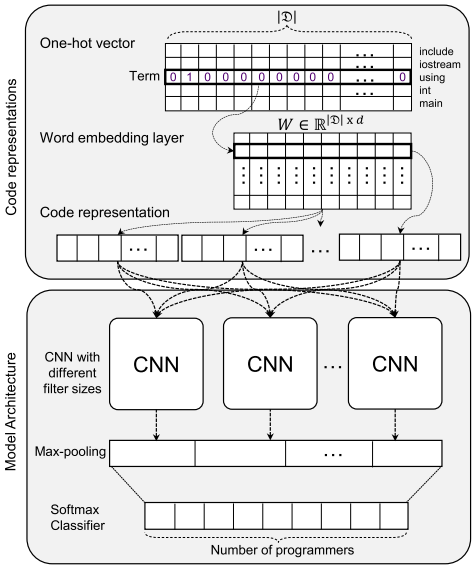


Fig. 1. C-CNN architecture with word embedding layer. The CNN model include multiple convolutional layers with different filter sizes.

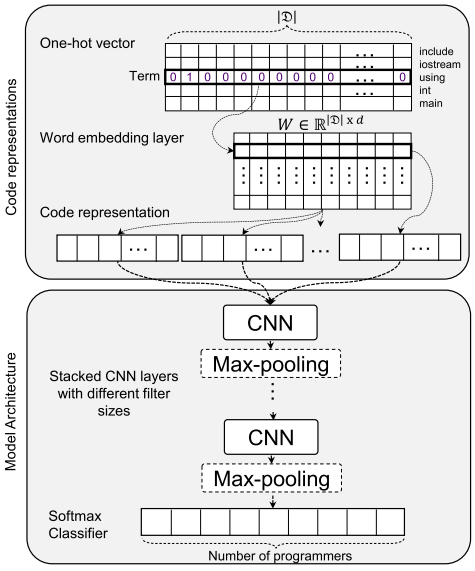


Fig. 2. S-CNN architecture with word embedding layer. The CNN model stacks multiple convolutional layers with different filter sizes.

Concatenated CNN Architecture (C-CNN). The resultant feature maps from different CNN layers can be concatenated as in Fig. 1. The concatenated model architecture adopted in this work is a variant of an earlier model [13]. We refer to this model architecture as a concatenated CNN with word embedding, and it can be expressed as

$$pool(conv_1(x)) \oplus pool(conv_2(x)) \oplus \cdots \oplus pool(conv_n(x))$$

where $conv_i$ is the feature map of the ith CNN layer.

Stacked CNN Architecture (S-CNN). Feature maps produced by CNN layers can be stacked on top of each other as seen in Fig. 2. We refer to this model architecture as stacked CNN with word embedding:

$$pool\Big(conv_n \ldots (pool(conv_1(x))) \ldots\Big)$$

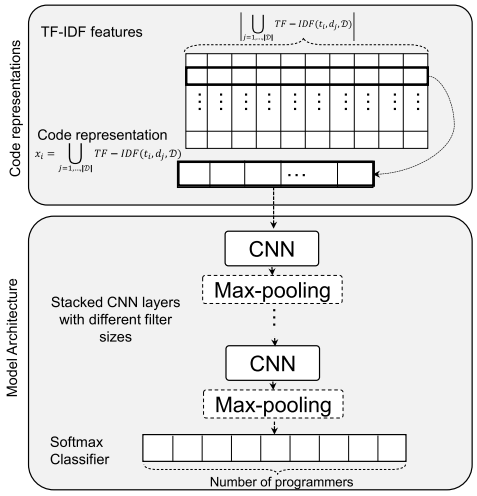


Fig. 3. S-CNN architecture with TFIDF representation. The CNN model stacks multiple convolutional layers with different filter sizes.

In both architectures, the final feature maps are connected to a fully-connected softmax layer that signifies the probability distribution over the authors of the code samples.

Using either approach, concatenated or stacked CNN layers have shown remarkable results. Stacked CNN layers have been used widely in the field of computer vision [40]. However, concatenated CNN layers have been successfully applied to several NLP tasks [13,26,39]. In this work, we explore both model architectures.

As mentioned before, the code authorship identification process can be formulated as a classification problem, where authors are classified based on their distinctive authorship attributes. The accuracy of this classification process relies on what is distinctive in the data representation (features or attributes) of each author. To examine the effectiveness of the proposed architectures for code authorship identification based on the data representation of each author's data, we visualized five authors' data at the word embedding level and the output of the last layer of the C-CNN and S-CNN architectures using the principal components analysis (PCA). PCA is a statistical tool that is widely used as a visualization technique that reflects the difference in observations of multidimensional data for the purpose of simplifying further analysis [41,42]. Fig. 6 shows PCA visualizations of C++ code samples from five programmers, with nine samples each at the initial code presentation level and the last layer of the CNN architecture. In Fig. 6(a) and 6(c), code files are presented with the initial word embedding features, which are insufficient to draw a decision boundary for all programmers. In Fig. 6(b) and 6(d), however, the deep representation have increased the margin for decision boundary so distinguishing programmers has become easier. This visualization of the representation space (word embedding features and deep representation) illustrates the quality of code representations obtained by the deep learning technique. We have noticed that the learned word embeddings using the C-CNN architecture form more distinctive representation in comparison to the one obtained using the word embeddings of the S-CNN architecture. The change in the quality of word embeddings using different architectures is due to the end-to-end learning process for the whole architecture as word representations are optimized as part of the overall architecture training process. This can be seen in Fig. 6(c), where the file representations are very close and less distinctive, compared to the word embedding representations generated by the C-CNN structure illustrated in Fig. 6(a).

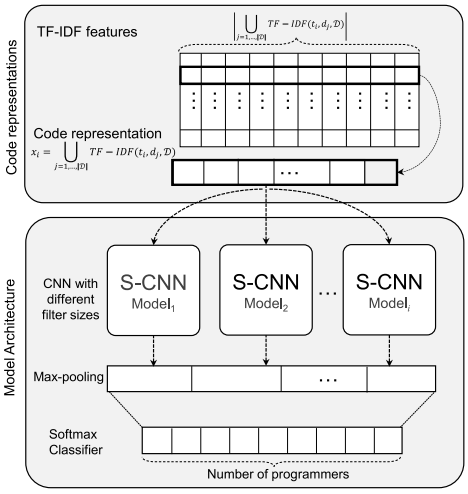


Fig. 4. Concatenated S-CNN architecture (*i*S-CNN) with TF-IDF representation. The *i*S-CNN model concatenates *i* number of S-CNN models.

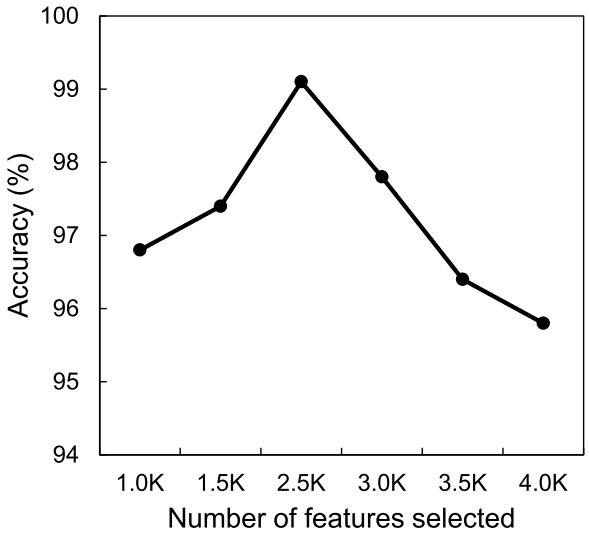


Fig. 5. Accuracy as the number of features of TF-IDF increased.

4.2. CNN architecture with TF-IDF approach

The second approach uses TF-IDF representation as the input representation of code samples as described in Section 3. TF-IDF has been used for several NLP and data mining tasks. However, handling programming codes requires several prepossessing steps to eliminate stop words and remove unnecessary code characters. Moreover, TF-IDF features that could attributed to authors are not limited to unigrams, rather than n-grams. Previous work [1] shows that bi-grams and tri-grams are important in identifying programmers. This work uses the TF-IDF representation of uni-grams, bi-grams, and tri-grams to represent code samples.

Let code sample x be represented as

$$[\text{TF-IDF}(t_1, x, \mathcal{D}), \text{TF-IDF}(t_2, x, \mathcal{D}), \ldots, \text{TF-IDF}(t_n, x, \mathcal{D})],$$

where n is the total number of terms in the corpus $\mathcal{D}$; then, extracting TF-IDF features could result in very sparse and high-dimensional vector representations due to the large existence of unique uni-grams, bi-grams, and tri-grams in the entire corpus. Most of these features are not relevant to our task of identifying programmers. We apply a straightforward univariate feature selection method to select the top-k features. The TF-IDF representation for code sample x_i become

$$x_i = \bigcup_{j=1,\ldots,|\mathcal{D}|} \text{TF-IDF}(t_i, d_j, \mathcal{D}),$$

where $\cup$ is a feature selection operator.

To illustrate the impact of choosing different top-k TF-IDF features on the accuracy of code authorship identification, we created a dataset of 1000 C++ programmers. Fig. 5 demonstrates the impact of choosing different top-k TF-IDF features on the accuracy of authorship identification. The figure shows that the accuracy of our model increases up to some value for the number of features (i.e., reaches the peak when we select the top 2500 features for each code sample); then, it decays quickly. The accuracy, even with the smallest number of features, is relatively high.

To examine the effectiveness of the proposed architectures for code authorship identification on the data representation of each author's data, we visualized five authors' data at the word embedding level and the output of the last layer of the S-CNN architecture using PCA. In Fig. 6(e) code files are presented with the initial TF-IDF features, which are insufficient to draw a decision boundary for all programmers. In Fig. 6(f), however, the deep representation have increased the margin for the decision boundary, so distinguishing programmers has become easier. This visualization of the representation space (TF-IDF features and deep representations) illustrates the quality of code authorship representation obtained using the deep learning technique.

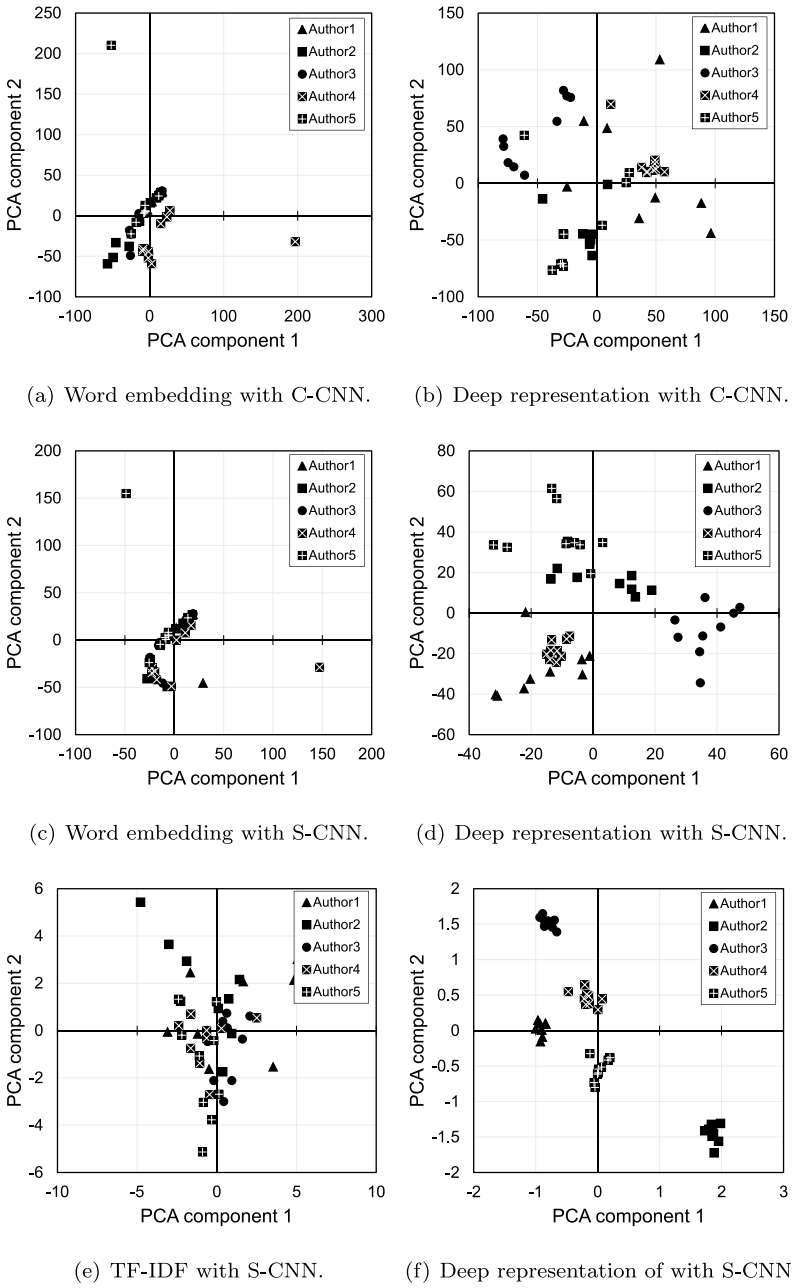
Unlike the word embedding approach, TF-IDF produces fixed-size representation for code samples. Moreover, TF-IDF representation neglect the order of terms in a code sample. Thus, a convolution process would highlight the importance of these input features, $x_i \in \mathbb{R}^k$, by data transformation using a filter, $w \in \mathbb{R}^d$, and generating a feature map, $c \in \mathbb{R}^k$, with zero-padding. We apply a max-pooling operation to extract the most important features. In this architecture, we stack multiple CNN layers on top of each other, where the output of the last layer is connected to a fully-connected softmax layer. This model architecture is illustrated in Fig. 3.

5. Experiment and evaluation

In this section, we present the results of the three proposed architectures of Section 4. For our experiment implementation, we used TensorFlow [43], an open source library for building and training neural networks using data flow graphs. We ran our experiments on a workstation with 24 cores, one GeForce GTX TITAN X graphics processing uni (GPU), and 256 GB of memory. We note that our use of the GeForce GTX TITAN X GPU was purely performance-driven, and the specific platform does not affect the end-results. Upon the release of our scripts and data, our findings can be reproduced in any other experimental settings. First, we provide an introduction to the dataset used for our evaluation. Then, we give an overview of the model architectures we constructed and the hyperparameters for these models. Following that, we present the training data requirements and model training details. Finally, we present the main results of our proposed approaches and how these approaches scale up to identify with high accuracy the authorship of code written by up to 1600 programmers.

5.1. Corpus

Google Code Jam (GCJ) is an international programming competition run by Google since 2008 [44]. At GCJ, programmers from all over the world use several programming languages and development environments to solve programming problems over multiple



(a) Word embedding with C-CNN. (b) Deep representation with C-CNN.

(c) Word embedding with S-CNN. (d) Deep representation with S-CNN.

(e) TF-IDF with S-CNN. (f) Deep representation of with S-CNN.

Fig. 6. The PCA visualization of code samples represented initially by word embedding or TF-IDF and the corresponding deep representation of code authorship attributions from the last layer of different CNN architectures. The visualization shows code sample representations for five programmers with nine samples each.

rounds. Each round of the competition involves writing a program to solve a small number of problems—three to six, within a fixed amount of time. We evaluate our approach on the source code of solutions to programming problems from GCJ.

The most commonly used programming languages at GCJ are C++, Java, and Python, in that order. Each of those languages has a sufficient number of source code samples from each programmer, and thus, we used them for our evaluation. For a large-scale evaluation, we used the contest code from 2008 to 2016, with general statistics, as shown in Table 3. The table shows the number of files per author across the years, with the total number of authors per programming language, and the average file size, or lines of code (LoC). For the evaluation, we created a dataset that includes files from across all the years from 2008 to 2016 in a "cross-years" view, as shown in Table 3.

Table 3
Datasets used in our study with corresponding statistics, including the number of authors with at least a specific number of files across all years.

Competition year	Author files	No. of authors for languages		
		C++	Python	Java
Across years	9	6635	2300	1279
Average lines of code		71.53	44.44	86.70

5.2. Constructed models for experiments

Using a cross-validation process, different model architectures were tested in terms of accuracy and performance. The CNN architecture can be designed using a number of hyperparameters, such as the number of convolutional layers in which each applies a number of filters with size k and strides s over the input data.

Table 4
Model Architectures for the concatenated and stacked CNN structures.

	Word embedding	Convolutional layers	Layers depth	# options
C-CNN	32, 64, 128, 256	[3, 4, 5], [3, 5, 7], [3, 5, 7, 9]	128	12
S-CNN	32, 64, 128, 256	[3, 4, 5], [3, 5, 7], [3, 5, 7, 9]	[32, 64, 128], [32, 64, 128], [32, 64, 128, 256]	12

A convolutional layer is usually followed by a max-pooling layer, as a down-sampling operation using a filter of size k with stride s. After a sequence of convolutional layers, fully-connected layers are used to map the features extracted by the convolutional layers to the output labels. Table 4 describes the different CNN architectures to be tested. The table includes 24 possible model architectures, three for each embedding size and with a total of 12 for each proposed architecture (i.e., C-CNN or S-CNN architectures). We used either three or four layers in each model. For example, for word embedding of size 128, we used three layers with filter sizes 3, 4, and 5, and a depth equal to 128 filters. We decided to use the most common choice of max-pooling in the literature by applying 1×4 filters with a stride of 4, down-sampling only 25% of the input activations.

In the C-CNN models, we fixed all hyperparameters except word embedding size and filter size. In the S-CNN models, we also fixed all hyperparameters except word embedding size and filter size, but we followed layer depths of 32, 64, and 128 in all the three-layer CNN architectures and layer depths of 32, 64, 128, and 256 for the four-layer CNN architecture. Then, we conducted a cross-validation evaluation. We adopted stratified nine-fold cross-validation, where each fold contains one file for each author. Given the filter size and embedding size, the collected results are the average performance of the model on each fold after repeating the nine-fold cross validation five times with different random seeds.

Finally, the reported results are the average of each model, given the word embedding size. For example, the results of the models with convolutional layers [3, 4, 5], [3, 5, 7], and [3, 5, 7, 9] and word embedding size 32 were averaged and then reported in the results Section 5.4.

5.3. Model training

The weights of model $W \in R^{m \times n}$ in a given layer are initialized from a zero-mean Gaussian distribution with standard deviation equals to $\sqrt{2/n}$ [45]. The b biases are initialized to zero. For training our deep learning structure, we used moment estimation algorithm (Adam optimizer) [46]. The Adam optimizer is an efficient stochastic optimization method that only requires first-order gradients with few memory requirements. Using estimations of the first two moments of the gradients, the Adam optimizer assigns different adaptive learning rates for different parameters. The method was inspired by combining the advantages of two popular stochastic optimization methods, AdaGrad [47], which is efficient in handling sparse gradients, and RMSProp [48], which is efficient in on-line and non-stationary settings [46].

To control the training process and to prevent overfitting, two different regularization methods were used, namely: dropout regularization and $L2$ regularization. Dropout regularization [49] randomly and temporally excludes a number of units on the forward pass, and weight updates on the backward pass during the training process. This approach was proven to enable the neural network to reach better generalization capabilities [49]. $L2$ regularization penalizes certain parameter configurations. Given loss function $loss(\theta,\ D)$, where θ is the set of all the model parameters and D is the dataset of length n samples, the regularization loss becomes

$$total_{loss}(\theta,\ D) = loss(\theta, D) + [\lambda \times Reg(\theta)],$$

where λ is a constant that controls the importance of regularization and

$$Reg(\theta) = (\sum_{j=0}^{|\theta|} |\theta_j|^p)^{(1/p)}$$

where p is equal to 1 or 2 (hence the $L1$ and $L2$ nomenclature).

In the training process of the deep learning architecture, we used a mini-batch size ranging from 64 to 128 samples. The idea of using mini-batches is to reduce the variance in gradients of individual observations since observations could be significantly different. Instead of computing the gradient of one observation, the mini-batch approach computes the average gradient for a number of observations at a time. This approach is widely accepted and commonly used in the literature. The training termination mechanism was either to reach $10k$ iterations or to hit an early termination threshold for the loss value. Since the output of the model is programmers' labels, the loss function used to train the model is softmax cross-entropy loss:

$$\mathrm{E}(p_n, \hat{p}_n) = -\sum_{n=1}^{N} p_n \log \hat{p}_n,$$

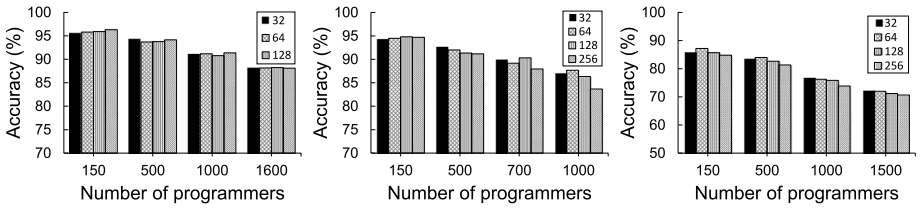
$$where, \hat{p}_n(x) = softmax(x) = e^{x_i} \cdot (\sum_j e^{x_j})^{-1} \quad \forall j$$

The learning process aims to minimize the loss between estimated distribution $\hat{p}_n$ and actual distribution p_n. For the training process, we explored different hyperparameters to conclude with a learning rate of $1e^{-4}$ without reducing the learning rate over time, dropout with a keep-probability of 0.6, and $L2$ regularization strength $\lambda = 1e^{-2}$.

5.4. Evaluation results

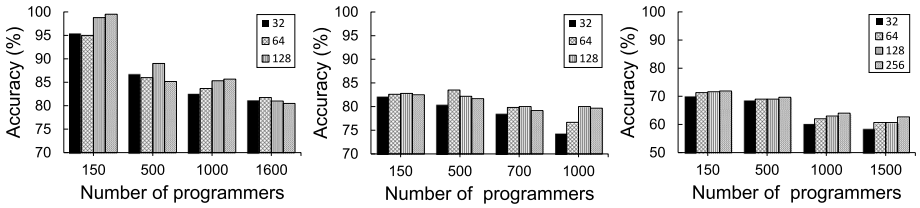
In this section, we present our results for the three proposed approaches. We used the dataset in Table 3. There are three large-scale datasets corresponding to three programming languages, with programmers who have at least nine code files to be selected. The number of code files per author in this experiment was deemed sufficient for extracting distinctive code authorship attribution features [1]. An accuracy metric was used to evaluate our results. Accuracy is defined as the percentage of code files correctly attributed over the total number of tested code files. Using accuracy instead of other evaluation metrics (e.g. precision and/or recall) is good enough, because the classes are balanced in terms of the number of presented files per author in the dataset. As mentioned in Section 5.2, we conducted stratified nine-fold cross-validation evaluation. Given the filter size and embedding size, the collected results are the average performance of the model on each fold after repeating the nine-fold cross validation five times with different random seeds. Moreover, the reported results are the average of each model given the word embedding size.

Experiment 1. In this experiment, we implemented the CNN architecture of Fig. 1. Fig. 7 shows how well our approach using the concatenated CNN structure scales for a large number of programmers, and for the various programming languages. The results report the accuracy for different word embedding sizes in learning code authorship attribution. Each reported accuracy result is the average over different filter size settings (i.e., applying filter sizes 3, 4, and 5 and 3, 5, and 7) when the words embedding size is



(a) C++ authors (b) Java authors (c) Python authors.

Fig. 7. The accuracy of the authorship identification of programmers in three programming languages using C-CNN with word embedding size 32, 64, 128, and 256.



(a) C++ authors (b) Java authors (c) Python authors.

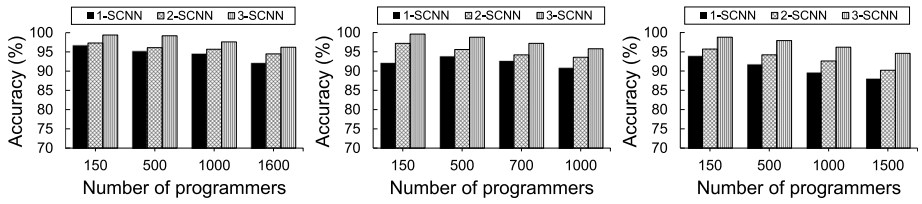
Fig. 8. The accuracy of the authorship identification of programmers in three programming languages using S-CNN with word embedding size 32, 64, 128, and 256.

fixed. Fig. 7(a) shows that the concatenated CNN approach with words embedding of size 256 outperforms models with other options for words embedding and achieved 96.3% accuracy for 150 C++ programmers. As we increased the number of programmers, accuracy reached 88.3% for 1600 programmers. Using the same CNN architecture with the dataset of Java programming in Table 3, similar results were obtained, as illustrated in Fig. 7(b), with 94.8% accuracy when the number of programmers is 150. As we scaled up the experiment to more programmers, we achieved 87.7% accuracy for 1000 programmers. For the Python language dataset, our approach achieved an accuracy of 87.2% for 150 programmers and 72.3% for 1500 programmers, as shown in Fig. 7(c). These results indicate that our deep learning approach is capable of learning deep representation of code authorship attributions to enable large scale authorship identification regardless of the used programming language and with a little influence from word embedding size.

Experiment 2. In this experiment, we implemented the stacked CNN model of Fig. 2 to explore the effect of increasing the depth of the CNN model. Using this approach was motivated by the remarkable results achieved by deep CNN models in the field of computer vision [50]. Giving the model more depth rather than breadth (e.g., a concatenated approach or wider layers), would enable us to investigate the impact of deep networks in our task. Similar to Experiment 1, the reported accuracy results are based on different word embedding sizes, where each reported accuracy is the average over different filter size settings (i.e., applying filter sizes 3, 4, and 5, and 3, 5, and 7) when the word embedding size is fixed. Fig. 8(a) shows that our approach with the stacked CNN architecture and words embedding of size 256 outperforms models with other options for word embedding and achieved 99.5% accuracy for 150 C++ programmers. As we increased the number of programmers, the accuracy reached 81.75% for 1600 programmers. Given the same CNN structure with the dataset of Java language, similar results are obtained, as illustrated in Fig. 8(b) with 82.8% accuracy when the number of programmers is 150. As we scaled the experiment to more programmers, we achieved 80% accuracy for 1000 programmers. For the Python language dataset, our approach achieved an accuracy of 72% for 150 programmers and 62.66% for 1500 programmers, as shown in Fig. 8(c). The results of this experiment show some degradation in accuracy, compared to results achieved in Experiment 1 (e.g., 12% (= 94.8 − 82.8) and 15.2% (= 87.2 − 72) for identifying 150 Java and Python programmers, respectively). This could be due to using an end-to-end training mechanism, which requires a significant amount of data to train. Also, it becomes more difficult for deeper networks to map code representation to the output layer as the number of hyperparameters involved increases. The word embeddings generated from the word embedding layer becomes less influential owing to the distance from the final output of the model. Using different approaches to generate fixed code representation could make it easier for the stacked CNN to achieve better results, and we keep this for future work.

Experiment 3. In this experiment, we implemented the S-CNN model with TF-IDF representation as shown in Fig. 3, which achieved the best results of all our proposed models. The first bar of Fig. 9(a), labeled S-CNN, shows that our approach with a TF-IDF-based S-CNN structure outperforms other proposed architectures, and achieved 96.7% accuracy for 150 C++ programmers. As we increased the number of programmers, the accuracy decreased slightly to reach 92.2% for 1600 programmers. Given the same S-CNN architecture with the dataset of Java language, similar results were obtained, as illustrated in Fig. 9(b) with 92.2% accuracy when the number of programmers is 150. As we scaled the experiment to more programmers, we achieved 90.9% accuracy for 1000 programmers. For the Python language dataset, our approach achieved an accuracy of 93.9% for 150 programmers and 88.1% for 1500 programmers, as shown in Fig. 9(c). Compared to S-CNN approach with word embeddings, S-CNN with TF-IDF representation achieved better identification accuracy (e.g., an accuracy improvement of 9.4% (92.2 − 82.2) and 21.9% (93.9 − 72 for identifying 150 Java and Python programmers). This indicates that our deep learning approach using TF-IDF is capable of learning deep representation of code authorship attributions that enable achieving large-scale authorship identification, regardless of the programming language. Also, these results show the importance of code representation so the CNN can achieve large-scale authorship identification.

Since TF-IDF with S-CNN achieved the best results in the three experiments, we combined the idea of the C-CNN and S-CNN to investigate whether we can get better performance. In Fig. 9(a), we report the results of S-CNN with two and three concatenated S-CNN branches (labeled 2S-CNN and 3S-CNN, respectively), where



(a) C++ authors (b) Java authors (c) Python authors.

Fig. 9. The accuracy of the authorship identification of programmers in three programming languages using *i*S-CNN with TF-IDF, where $i = 1$, 2, and 3.

the whole structure contains either two or three stacked CNN models. The new architecture illustrated in Fig. 4 achieved the best accuracy under 3S-CNN with a depth of three CNN layers. The 3S-CNN architecture achieved 99.4% accuracy for 150 C++ programmers. As we increased the number of programmers, the accuracy decreased slightly to reach 96.2% for 1600 programmers. Given the same CNN architecture for the dataset of the Java language, similar results were obtained, as illustrated in Fig. 9(b) with 99.6% accuracy when the number of programmers is 150. As we scaled the experiment to more programmers, we achieved 95.8% accuracy for 1000 programmers. For the Python language dataset, our approach achieved an accuracy of 98.8% for 150 programmers and 94.6% for 1500 programmers, as shown in Fig. 9(c).

Experiment 4. Since the results from the 3S-CNN architecture with TF-IDF representation are the best of all the proposed architectures, we examined the 3S-CNN model of Fig. 4 on code samples collected from 1987 public GitHub repositories. The aim of this experiment is to demonstrate the applicability of our approach to real-world applications using real dataset. We collected samples from repositories with one contributor and set C and C++ as the primary language. We filtered all programmers with fewer than five samples to settle with 142 and 745 from C++ and C programmers, respectively. For programmers who had more than 10 samples, we randomly selected 10 samples only. For the ground truth of our dataset, we collected repositories with a single contributor under the assumption that the collected samples were written by the same contributor to the repository. We acknowledge that this assumption is not always valid, because parts of the code samples might have been copied from other sources [18]. Even though this GitHub dataset could include some reused code in the samples, the 3S-CNN architecture with TF-IDF representation achieved 95% accuracy for 745 C programmers and 97% for the C++ programmers. This result shows that our approach is still effective when handling real-world datasets.

6. Limitations

This work introduces large-scale code authorship identification approach using CNN. The proposed approach achieves high identification accuracy in diverse settings using different code representation techniques and CNN architectures. However, there are several unexplored issues that we highlight in the following.

Authorship verification. During the processing and preparing of the GCJ dataset (cross-years dataset in Table 3), we accumulated code samples for authors who used the same username in different-year competitions. Our assumption was that participants used the same information through the competition. We acknowledge that this assumption does not always hold and there could be users who used the same username that had been used by different participant during past competitions. Another authorship assumption was regarding the Github dataset, since we considered all code samples in a repository are written by the contributor. To verify authorship of code samples in both datasets can be achieved by considering a separate prior stage for checking the consistency of code authorship attribution across different samples to insure authorship before using the dataset.

Multiple authors. The experiments of this work show high accuracy of identifying programmers based on their sole code samples. In reality, however, code samples are written by several programmers considering large software projects [17,18]. Future work might explore the possibility of identifying multiple programmers in one code sample.

Mixed-languages. Experimenting with three different programming languages (C++, Java, and Python), our approach shows resilience to language specifics by achieving high accuracy for all used languages. However, all experiments were conducted to include programmers with code samples of one language individually. As programmers might use different languages, exploring the robustness of authorship attribution across different languages is an interesting direction for future work.

Comprehensive code authorship attribution. This work uses code authorship attribution extracted by the CNN model fed with either word embedding or TF-IDF code representation, which are oblivious to some coding style traits such as layout features (i.e., spacing and indentation). Although previous work [1] has demonstrated a positive impact of including layout features, this work has not incorporated these features to simplify the approach and avoid the relatively complicated process of manually crafting these layout features. We leave exploring the effects of including layout features in the accuracy of the proposed approach for future work.

7. Conclusion

Knowing the author of a source code has a significant impact on the advancement of malicious software forensics, copyright dispute resolutions, and detection of software plagiarism. In this work, we presented three code authorship identification structures that utilize word embedding and TF-IDF with CNN-based deep learning process to extract distinctive features for authorship identification. We conducted an empirical study of 1600 programmers from GCJ through the years 2008 to 2016 who used one of three programming languages: C++, Java, or Python, and real-world code samples from 1987 public repositories on GitHub. Our evaluation showed that the proposed approach is robust and scalable, and achieved high accuracy in various settings. Our approach can distinguish up to 1600 C++ programmers with 96.2% accuracy, 1000 Java programmers with 95.8% accuracy, and 1500 python programmers with 94.6% accuracy. In the future, we will explore the effects of copying code samples of other programmers on the performance of our approach and the way to extend our approach for multi-author code authorship identification.

Acknowledgments

This research was supported by the Global Research Lab. (GRL) Program of the National Research Foundation (NRF) funded by

the Ministry of Science, ICT and Future Planning, Korea (NRF-2016K1A1A2912757) and the NRF (Korea) grant (NRF-2016R1D1A1B03934816).

References

[1] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, R. Greenstadt, De-anonymizing programmers via code stylometry, in: Proceedings of the 24th USENIX Conference on Security Symposium, SEC'15, USENIX Association, Berkeley, CA, USA, 2015, pp. 255–270.

[2] L.J. Wilcox, Authorship: The coin of the realm, the source of complaints, JAMA 280 (3) (1998) 216–217.

[3] C.H. Malin, E. Casey, J.M. Aquilina, Malware forensics: Investigating and analyzing malicious code, Syngress, 2008.

[4] S. Burrows, S.M.M. Tahaghoghi, J. Zobel, Efficient plagiarism detection for large code repositories, Softw. - Pract. Exp. 37 (2) (2007) 151–175.

[5] G. Frantzeskou, E. Stamatatos, S. Gritzalis, C.E. Chaski, B.S. Howald, Identifying authorship by byte-level n-grams: The source code author profile (SCAP) method, Int. J. Digit. Evid. 6 (1) (2007) 1–18.

[6] S. Burrows, A.L. Uitdenbogerd, A. Turpin, Temporally robust software features for authorship attribution, in: 2009 33rd Annual IEEE International Computer Software and Applications Conference, vol. 1, 2009, pp. 599–606.

[7] Y. LeCun, L. Bottou, Y. Bengio, P. Haffner, Gradient-based learning applied to document recognition, Proc. IEEE 86 (11) (1998) 2278–2324.

[8] R. Johnson, T. Zhang, Semi-supervised convolutional neural networks for text categorization via region embedding, in: Advances in Neural Information Processing Systems, 2015, pp. 919–927.

[9] S. Lai, L. Xu, K. Liu, J. Zhao, Recurrent convolutional neural networks for text classification, in: AAAI, vol. 333, 2015, pp. 2267–2273.

[10] P. Blunsom, E. Grefenstette, N. Kalchbrenner, A convolutional neural network for modelling sentences, in: Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics, 2014.

[11] Y. Shen, X. He, J. Gao, L. Deng, G. Mesnil, Learning semantic representations using convolutional neural networks for web search, in: Proceedings of the 23rd International Conference on World Wide Web, 2014, pp. 373–374.

[12] W.-t. Yih, X. He, C. Meek, Semantic parsing for single-relation question answering, in: Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 2: Short Papers), vol. 2, 2014, pp. 643–648.

[13] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, P. Kuksa, Natural language processing (almost) from scratch, J. Mach. Learn. Res. 12 (Aug) (2011) 2493–2537.

[14] S. Burrows, S.M.M. Tahaghoghi, Source code authorship attribution using n-grams, in: A. Spink, A. Turpin, M. Wu (Eds.), in: Proceedings of the Twelfth Australasian Document Computing Symposium, ADCS'07, 2007, pp. 32–39.

[15] V. Keselj, F. Peng, N. Cercone, C. Thomas, N-gram-based author profiles for authorship attribution, in: Proceedings of the Conference Pacific Association for Computational Linguistics, PACLING, vol. 3, 2003, pp. 255–264.

[16] G. Frantzeskou, E. Stamatatos, S. Gritzalis, S. Katsikas, Effective identification of source code authors using byte-level information, in: Proceedings of the 28th International Conference on Software Engineering, ICSE '06, New York, NY, USA, 2006, pp. 893–896.

[17] X. Meng, B.P. Miller, K.-S. Jun, Identifying multiple authors in a binary program, in: European Symposium on Research in Computer Security, Springer, Oslo, Norway, 2017, pp. 286–304.

[18] E. Dauber, A. Caliskan, R. Harang, R. Greenstadt, Git blame who?: Stylistic authorship attribution of small, incomplete source code fragments, in: Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings, ACM, 2018, pp. 356–357.

[19] N. Rosenblum, X. Zhu, B. Miller, Who wrote this code? Identifying the authors of program binaries, in: Computer Security–ESORICS 2011, Springer, 2011, pp. 172–189.

[20] A. Caliskan, F. Yamaguchi, E. Dauber, R. Harang, K. Rieck, R. Greenstadt, A. Narayanan, When coding style survives compilation: De-anonymizing programmers from executable binaries, in: Proceedings of the 25th Network and Distributed System Security Symposium, NDSS 2018, Internet Society, San Diego, CA, USA, 2018.

[21] S. Burrows, A.L. Uitdenbogerd, A. Turpin, Comparing techniques for authorship attribution of source code, Softw. - Pract. Exp. 44 (1) (2014) 1–32.

[22] I. Krsul, E.H. Spafford, Refereed paper: Authorship analysis: Identifying the author of a program, Comput. Secur. 16 (3) (1997) 233–257.

[23] G. Frantzeskou, E. Stamatatos, S. Gritzalis, S.K. Katsikas, Source code author identification based on n-gram author profiles, in: Artificial Intelligence Applications and Innovations, 3rd IFIP Conference on Artificial Intelligence Applications and Innovations, AIAI, 2006, June 7–9, 2006, Athens, Greece, 2006, pp. 508–515.

[24] S. Burrows, A.L. Uitdenbogerd, A. Turpin, Application of information retrieval techniques for source code authorship attribution, in: Database Systems for Advanced Applications, 14th International Conference, DASFAA 2009, Brisbane, Australia, April 21–23, 2009. Proceedings, 2009, pp. 699–713.

[25] M. Abuhamad, T. AbuHmed, A. Mohaisen, D. Nyang, Large-Scale and language-oblivious code authorship identification, in: Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, CCS '18, ACM, New York, NY, USA, 2018, pp. 101–114.

[26] Y. Kim, Convolutional neural networks for sentence classification, in: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP, Association for Computational Linguistics, 2014, pp. 1746–1751.

[27] P. Wang, J. Xu, B. Xu, C. Liu, H. Zhang, F. Wang, H. Hao, Semantic clustering and convolutional neural network for short text categorization, in: Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 2: Short Papers), vol. 2, 2015, pp. 352–357.

[28] R. Johnson, T. Zhang, Effective use of word order for text categorization with convolutional neural networks, in: Proceedings of the 2015 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, 2015, pp. 103–112.

[29] M. Tomas, K. Chen, G. Corrado, J. Dean, Efficient estimation of word representations in vector space, in: Proceedings of Workshop at International Conference on Learning Representations 2013, 2013.

[30] J. Pennington, R. Socher, C. Manning, Glove: Global vectors for word representation, in: Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing, EMNLP, 2014, pp. 1532–1543.

[31] R. Johnson, T. Zhang, Supervised and semi-supervised text categorization using LSTM for region embeddings, in: Proceedings of the 33rd International Conference on International Conference on Machine Learning - Volume 48, ICML'16, JMLR.org, 2016, pp. 526–534.

[32] T.H. Nguyen, R. Grishman, Relation extraction: Perspective from convolutional neural networks, in: Proceedings of the 1st Workshop on Vector Space Modeling for Natural Language Processing, 2015, pp. 39–48.

[33] D. Zeng, K. Liu, S. Lai, G. Zhou, J. Zhao, Relation classification via convolutional deep neural network, in: Proceedings of COLING 2014, the 25th International Conference on Computational Linguistics: Technical Papers, 2014, pp. 2335–2344.

[34] S.G. Macdonell, A.R. Gray, G. MacLennan, P.J. Sallis, Software forensics for discriminating between program authors using case-based reasoning, feedforward neural networks and multiple discriminant analysis, in: Neural Information Processing, 1999. Proceedings. ICONIP '99. 6th International Conference on, vol. 1, 1999, pp. 66–71.

[35] H. Ding, M.H. Samadzadeh, Extraction of java program fingerprints for software authorship identification, J. Syst. Softw. 72 (1) (2004) 49–57.

[36] R.C. Lange, S. Mancoridis, Using code metric histograms and genetic algorithms to perform author identification for software forensics, in: Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation, GECCO '07, 2007, pp. 2082–2089.

[37] B.S. Elenbogen, N. Seliya, Detecting outsourced student programming assignments, J. Comput. Sci. Coll. 23 (3) (2008) 50–57.

[38] B. Alsulami, E. Dauber, R. Harang, S. Mancoridis, R. Greenstadt, Source code authorship attribution using long short-term memory based networks, in: Computer Security – ESORICS 2017: 22nd European Symposium on Research in Computer Security, Oslo, Norway, September 11–15, 2017, Proceedings, Part I, 2017, pp. 65–82.

[39] Y. Zhang, B.C. Wallace, A sensitivity analysis of (and practitioners' guide to) convolutional neural networks for sentence classification, CoRR abs/1510.03820 (2015).

[40] Y.-L. Boureau, J. Ponce, Y. LeCun, A theoretical analysis of feature pooling in visual recognition, Proceedings of the 27th International Conference on Machine Learning, ICML-10, 2010, pp. 111–118.

[41] A.T. Basilevsky, Statistical Factor Analysis and Related Methods: Theory and Applications, vol. 418, John Wiley & Sons, 2009.

[42] B.S. Everitt, G. Dunn, Applied Multivariate Data Analysis, vol. 2, Wiley Online Library, 2001.

[43] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D.G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, X. Zheng, TensorFlow: A system for large-scale machine learning, in: Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI'16, 2016, pp. 265–283.

[44] Google Code Jam, 2016. https://code.google.com/codejam/. (Accessed on 15/02/2018).

[45] K. He, X. Zhang, S. Ren, J. Sun, Delving deep into rectifiers: Surpassing human-level performance on imagenet classification, in: Proceedings of the IEEE International Conference on Computer Vision, 2015, pp. 1026–1034.

[46] D.P. Kingma, J. Ba, Adam: A method for stochastic optimization, in: International Conference on Learning Representations, ICLR, vol. 5, 2015.

[47] J. Duchi, E. Hazan, Y. Singer, Adaptive subgradient methods for online learning and stochastic optimization, J. Mach. Learn. Res. 12 (Jul) (2011) 2121–2159.
[48] T. Tieleman, G. Hinton, Lecture 6.5-RMSProp: Divide the gradient by a running average of its recent magnitude, COURSERA Neural Netw. Mach. Learn. 4 (2) (2012).
[49] N. Srivastava, G.E. Hinton, A. Krizhevsky, I. Sutskever, R. Salakhutdinov, Dropout: a simple way to prevent neural networks from overfitting, J. Mach. Learn. Res. 15 (1) (2014) 1929–1958.
[50] Y. LeCun, Y. Bengio, G. Hinton, Deep learning, Nature 521 (7553) (2015) 436.

Mohammed Abuhamad received the B.S. degree in computer science from The IUG in 2007, and the M.S. in artificial intelligence from The National University of Malaysia in 2013. He is currently a Ph.D. candidate and active member in the Information Security Research Laboratory (ISRL) at Inha University, South Korea. His research interests lie in software security, authentication, privacy, and deep learning.

Ji-su Rhim received the B.S. degree in computer engineering from Chungnam National University in 2007. He is currently a master degree student and active member at the intelligence mobile Laboratory at Inha university, South Korea. His research interests lie in software security, mobile intelligence, and deep learning.

Tamer Abuhmed received the Ph.D. degree in information and telecommunication engineering from Inha University in 2012. He is currently an Assistant Professor with the Department of Computer Engineering at Inha University, South Korea. His research interests include applied cryptography and information security, network security, Internet security, and machine learning and its application to security and privacy problems.

Sana Ullah received the Ph.D. degree in information and communication engineering from Inha University, Incheon, South Korea, in 2011. From December 2011 to March 2014, he was an Assistant Professor with the College of Computer and Information Science, King Saud University, Riyadh, Saudi Arabia. He is currently an Assistant Professor with the Department of Computer and Software Technology, University of Swat, Mingora, Pakistan. Dr. Ullah is currently an Editor for the Springer Journal of Medical Systems (JOMS), KSII Transaction of Internet and Information Systems, Wiley Security and Communication Network, Journal of Internet Technology, and International Journal of Autonomous and Adaptive Communications Systems. He was a Guest Editor for many top journals including Elsevier's Journal of Information Science, Springer's Journal of Medical Systems, and Springer Journal of Telecommunication Systems. He was also the Co-Chair/TPC member for a number of international conferences including BodyNets, IEEE PIMRC, IEEE Healthcom, IEEE Globecom, and IEEE WCNC.

Sanggil Kang received the M.S. and Ph.D. degrees in Electrical Engineering from Columbia University and Syracuse University, USA in 1995 and 2002, respectively. He is currently an associate Professor in the Department of Computer Engineering at INHA University, Korea. His research interests include Semantic Web, Artificial Intelligence, Multimedia Systems, Inference Systems, etc.

DaeHun Nyang received the B.Eng. degree in electronic engineering from Korea Advanced Institute of Science and Technology in 1994, and the M.S. and Ph.D. degrees in computer science from Yonsei University, South Korea, in 1996 and 2000, respectively. He was a Senior Member of the Engineering Staff with the Electronics and Telecommunications Research Institute, South Korea, from 2000 to 2003. Since 2003, he has been a Full Professor with the Computer Engineering Department, Inha University, South Korea, where he is currently the Founding Director of the Information Security Research Laboratory. His research interests include cryptography and network security, privacy, usable security, biometrics, and their applications to authentication and public key cryptography. He is a member of the Board of Directors and Editorial Board of the Korean Institute of Information Security and Cryptology.