IEEE Access
Multidisciplinary | Rapid Review | Open Access Journal

Received August 27, 2020, accepted October 2, 2020, date of publication October 30, 2020, date of current version November 11, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3034932

Language and Obfuscation Oblivious Source Code Authorship Attribution

SARIM ZAFAR, MUHAMMAD USMAN SARWAR, SAEED SALEM, (Member, IEEE), AND MUHAMMAD ZUBAIR MALIK

Department of Computer Science, North Dakota State University, Fargo, ND 58105, USA

Corresponding author: Muhammad Usman Sarwar (muhammad.sarwar@ndsu.edu)

This work was supported in part by the North Dakota Established Program to Stimulate Competitive Research (ND EPSCoR) under Grant FAR0032069.

ABSTRACT Source Code Authorship Attribution can answer many interesting questions such as: Who wrote the malicious source code? Is the source code plagiarized, and does it infringe on copyright? Source Code Authorship Attribution is done by observing distinctive patterns of style in a source code whose author is unknown and comparing them with patterns learned from known authors' source codes. In this paper, we present an efficient approach to learn a novel representation using deep metric learning. The existing state of the art approaches tokenize the source code and work on the keyword level, limiting the elements of style they can consider. Our approach uses the raw character stream of source code. It can examine keywords and different stylistic features such as variable naming conventions or using tabs vs. spaces, enabling us to learn a richer representation than other keyword-based approaches. Our approach uses a character-level Convolutional Neural Network (CNN). We train the CNN to map the input character stream to a dense vector, mapping the source codes authored by the same author close to each other. In contrast, source codes written by different programmers are mapped farther apart in the embedding space. We then feed these source code vectors into the K-nearest neighbor (KNN) classifier that uses Manhattan-distance to perform authorship attribution. We validated our approach on Google Code Jam (GCJ) dataset across three different programming languages. We prepare our large-scale dataset in such a way that it does not induce type-I error. Our approach is more scalable and efficient than existing methods. We were able to achieve an accuracy of **84.94%** across **20,458** authors, which is more than twice the scale of any previous study under a much more challenging setting.

INDEX TERMS Software engineering, natural language processing, artificial neural networks.

I. INTRODUCTION

Source code often contains distinctive patterns that represent a programmer's[1] style of writing code. The source code authorship attribution aims to extract these patterns from the source code and identify the author. Source code authorship attribution has primarily relied on feature engineering, where unique features are associated with each author, such as variable naming conventions, use of *for*, or *while* loop. However, extracting such features is time-consuming and challenging. Even for a single author, coding style varies across different programming languages due to language-specific conventions and constraints. Further, with continuous learning and increased programming expertise, programmers' styles keep on evolving.

Source code authorship attribution has numerous applications in the information security domain, such as identifying malicious source code authors, plagiarism detection [1], and resolving copyrights infringement [2]. Despite its application in numerous fields, source code authorship identification can also be a privacy risk for the programmers who do not want to reveal their true identities, such as contributors to open-source projects and activists [3].

Numerous studies [4]–[6] have been conducted to address the source code authorship attribution problem. The approaches usually focus on sophisticated graph-based features such as abstract syntax trees, program dependency graphs, and machine learning algorithms. However, most of the previous studies' features limit the applicability to usually one language because of the highly specialized/hand-crafted

The associate editor coordinating the review of this manuscript and approving it for publication was Hiram Ponce.

[1]We use the term author and programmer interchangeably

nature of the feature engineering step. For instance, features extracted for authorship attribution in the Python language cannot identify authors of C++ or Java language source code. There are a few studies [3], [7] that do address this issue by utilizing language-agnostic features. However, they work on the keyword level, thus limiting the elements of style they can consider. Also, existing studies use relatively small scale datasets, where the number of authors ranges from a few hundred to a few thousand authors. To address the issues mentioned above, we present an efficient character-based approach to learn a novel deep representation that maps the source code to a fixed-length vector. We enforce the deep representations to behave in such a way that enables us to use a lazy learning classifier on top of them.

In this paper, we present a CNN based approach for learning meaningful code representations. We use this approach in conjunction with K-nearest neighbors (KNN) Classifier to perform authorship attribution of the source code files. We train the CNN using lifted structured loss function [8] on character level embedding vectors. The lifted structured loss function forces the network to confine source codes originating from the same author ''close'' to each other and keep the source codes of different authors separated from each other in the embedding space. Then, we extract embedding vectors from CNN and feed them to the KNN to classify the authors. The primary advantage of utilizing a nearest neighbor classifier is its lazy learning capability and its ability to incorporate new samples without the need to retrain, which was not possible with the approach presented by Abuhamad *et al.* [3], [7]. Our study attempts to answers the following research questions. (i) Is our CNN-based approach capable of extracting language-agnostic features that will help to identify authors across different programming languages? (ii) Can our approach perform authorship attribution on obfuscated source codes? (iii) Can our approach identify authors under simulated real-world settings where known source code could be obfuscated or original? (iv) How does our approach perform in the open-world setting?

To answer the aforementioned questions, we conducted a series of experiments using source code files extracted from the Google Code Jam (GCJ) competition. We only considered the three most popular programming languages in the GCJ competition: C++, Java, and Python. We present our analysis of these three individual programming languages and the combination of all three languages (mix-language scenario) under three different settings. We also explored the effects of limited source code files (F) per author and conducted the experiments with F=5, F=7, and F=9 source code files per author, where the KNN uses F-1 source code files for training and one source code file for evaluation.

First, we conducted a large scale authorship attribution on the original source-code files extracted from GCJ. We demonstrate that our proposed framework can identify authors writing in individual programming languages and even the authors who write in multiple programming languages. With nine source code files per author, we were able to achieve 91.67% accuracy on 12,498 programmers writing code in multiple programming languages.

Second, we analyzed the effect of obfuscation of the source code authorship identification. For this, we obfuscated our source code files using off-the-shelf source code obfuscators. We demonstrate that our proposed approach is resilient to the source code obfuscation and can identify authors even from the obfuscated source codes while maintaining good accuracy. However, we observed some degradation in accuracy compared to the results achieved using original source-code files, which is expected as the sole purpose of obfuscation is to hide the stylistic features of an author. With nine files per author, we achieved 83.41% accuracy for 14,100 authors whose source-codes were written in multiple languages.

Third, we mimic a real-world scenario by combining obfuscated and original source-code files. We demonstrate that our framework can identify a large number of authors with high accuracy, even in our simulated real-world setting. With nine source code files per author, we achieved an accuracy of 84.94% for 20,458 programmers writing code in multiple languages.

Finally, we conducted an initial experiment to address our problem's open-world aspect, where the author could be someone outside the dataset. We used a distance threshold to differentiate between the known and unknown (out of the 'world') authors.

This is the first study to identify the authors on such a large scale and under such diverse settings to the best of our knowledge. Previously, Abuhamad *et al.* [3] conducted the largest experiment using 8,903 C++ programmers with seven files per author, as shown in Table 1. Our study scales to 23,850 programmers, which are more than twice the number of programmers considered by Abuhamad *et al.* [3].

The following is a summary of our contributions:

1) We proposed a CNN based framework, where we train the CNN using lifted structured loss function.
2) We conducted various authorship attribution experiments on the Google Code Jam dataset under diverse settings, i.e., single-language, multi-language, original source-code, obfuscated, and simulated real-world settings.
3) We also conducted an initial experiment to address the open-world scenario of our problem.
4) We demonstrated that our proposed framework is efficient and can handle a large number of programmers while maintaining high accuracy and efficiency.

Organization. The paper is structured as follows. We discuss our motivation to solve the author attribution problem in section II. In section III, we discuss the relevant previous studies. In section IV, we provide the theoretical background required to understand our study. In section V, we discuss our CNN based framework for source code authorship attribution. We discuss the different experiments used to evaluate our proposed approach in section VI. In section VII and section VIII, we discuss the limitations and potential future

TABLE 1. Comparison between our work and previous works across number of authors, languages, and performance.

Reference	No. of Authors	Languages	Accuracy (%)	Approach
Pellin [14]	2	Java	88.47%	SVM with tree kernel
MacDonell et al. [15]	7	C++	81.10%	Machine Learning (FNN). Statistical analysis (MDA)
MacDonell et al. [15]	2	C++	88.00%	Machine learning (case-based reasoning)
Frantzeskou et al. [16]	8	C++	100.00%	Rank similarity using KNN
Burrows et al. [17]	10	C	76.78%	Information retrieval using mean reciprocal ranking
Elenbogen & Seliya [18]	12	C++	74.70%	Statistical analysis using decision tree
Lange & Mancoridis [19]	20	Java	55.00%	Rank similarity measurements (nearest neighbor)
Krsul & Spafford [20]	29	C	73.00%	Statistical analysis (discriminant analysis)
Frantzeskou et al. [16]	30	C++	96.90%	Rank similarity measurements
Yang et al. [21]	40	Java	91.10 %	NN with particle swarm optimization
Ding & Samadzadeh [22]	46	Java	62.70%	Statistical analysis using canonical discriminant
Burrows et al. [12]	100	C, C++	79.90%	Support Vector Machines
Burrows et al. [12]	100	C, C++	80.37%	Random Forest
Caliskan-Islam et al. [5]	229	Python	53.91%	Random Forest
Meng et al. [23]	284	C, C++ Binaries	65.00%	Combined model with Conditional Random Field (CRF)
Farhan Ullah et al. [6]	1000	C#	98%	PDG with Neural Network
Farhan Ullah et al. [6]	1000	C++	100%	PDG with Neural Network
Farhan Ullah et al. [6]	1000	Java	98%	PDG with Neural Network
Abuhamad et al. [7]	1000	Java	95.8%	Stacked Convolutional Neural Network
Abuhamad et al. [7]	1500	Python	94.6%	Stacked Convolutional Neural Network
Abuhamad et al. [7]	1600	C++	96.2%	Stacked Convolutional Neural Network
Caliskan-Islam et al. [5]	1,600	C++	92.83%	Random Forest
Abuhamad et al. [3]	566	C	94.80%	RNN along with Random Forest
Abuhamad et al. [3]	1,952	Java	97.24%	RNN along with Random Forest
Abuhamad et al. [3]	3,458	Python	96.20%	RNN along with Random Forest
Abuhamad et al. [3]	8,903	C++	92.30%	RNN along with Random Forest
Our Approach	2355	Java (Original Source-code Setting)	91.66%	CNN with Similarity Metric
Our Approach	3911	Java (Simulated Real-world Setting)	79.36%	CNN with Similarity Metric
Our Approach	3567	Python (Original Source-code Setting)	92.18%	CNN with Similarity Metric
Our Approach	6910	Python (Simulated Real-world Setting)	84.79%	CNN with Similarity Metric
Our Approach	6238	C++ (Original Source-code Setting)	94.63%	CNN with Similarity Metric
Our Approach	10280	C++ (Simulated Real-world Setting)	90.98%	CNN with Similarity Metric
Our Approach	12498	mix-scenario (Original Source-code Setting)	91.67%	CNN with Similarity Metric
Our Approach	20,458	mix-scenario (Simulated Real-world Setting)	84.94%	CNN with Similarity Metric

avenues for our work respectively. Finally, we conclude our paper in section IX.

II. MOTIVATION

Source code authorship attribution is a widely studied research topic in the information security domain. Effective authorship attribution can help in many research applications such as plagiarism detection, software forensics, copyrights investigation, and authorship verification. These applications of the authorship attribution motivate our study. Some of the applications are described in detail as follows:

Plagiarism Detection: Plagiarism detection is one of the most studied research problems in recent years. Plagiarism is defined as the presentation of work or an idea of another person as your own. Assume we have a source code S and a set of potential source codes from which S could be copied. In this case, source code authorship identification can be helpful in detecting whose source code was copied. Numerous tools are available for plagiarism detection such as Measure of Software Similarity (MOSS) [9], Sherlock [10], Plague [11]. However, with the incorporation of authorship identification, we can considerably improve the accuracy of plagiarism tools [12].

Copyrights Investigation: In the absence of any contract, plagiarism of the source code may lead to copyright infringement. With a practical authorship attribution approach, one can determine the original author of the source code, resolving such disputes.

Software Forensics: Software forensics is the sub-field of computer forensics that deals with software applications and source codes. Source codes can be reviewed for evidence of activity, function, intention, and evidence of the software's author [13]. Software forensics is divided into four distinct research areas: author identification, author discrimination, author characterization, and author intent determination [13]. Effective author identification is a core component of software forensics and can play a pivotal role in software forensics.

III. RELATED WORK

Numerous studies have been conducted to explore software authorship attribution. These studies use machine learning models on different features, such as lexical features, syntactical features, semantic features, textual features, and graph-based features. Table 1 shows a summary of the related work, along with the comparison across four factors: number of authors, programming languages, accuracy, and approach.

Abuhamad *et al.* [3] proposed a CNN based source code authorship framework. Source codes are initially represented using TF-IDF and word embedding based vectors. These code representations are further fed into CNN to learn deep representations. These deep representations and authorship

labels are then used to train a random forest classifier to extract source code authorship. Results were evaluated using Google Code Jam (GCJ) and Github dataset. They reported accuracy of 99.4% for 150 programmers and 96.2% for 1600 programmers. They claim that their approach can scale to several programmers and across various programming languages.

Abuhamad *et al.* [7] proposed a recurrent neural network (RNN) based approach to classify source code authors. Source code files are first encoded in TF-IDF vectors and fed into an RNN model to generate the deep feature vectors. Further, the deep representations are fed into a random forest classifier to classify the source code authors. Results were evaluated using the GCJ and Github dataset. They achieved an accuracy of 96% for 1600 authors in the GCJ dataset and 94.38% for 745 authors in the Github dataset. Also, they reached an accuracy of 93.42% for 120 authors on obfuscated source code files. However, they created their large-scale dataset by assuming that if the same username occurs across different years of the code jam dataset, they must be the same author. While making this assumption, they can induce both false positive and false negative errors in the experiment.

Ullah *et al.* [6] used a program dependence graph (PDG) along with the deep learning model to identify the authors from the source code of different languages. First, the PDG is used to extract the control flow and data variation features from source code files. Then, TF-IDF based representations of PDG features are fed into a neural network to identify the source code author. They conducted the experimentation on a GCJ dataset of three programming languages: C++, Java, C#. These experiments scale to 1000 programmers.

Caliskan-Islam *et al.* [5] utilized machine learning models to de-anonymize source code authors. The extensive feature extraction process for programmer code stylometry involves code parsing. The stylometry feature set is a representation of the coding style and is derived from abstract syntax trees. Syntactic features for code stylometry are extracted using a fuzzy parser to generate an abstract syntax tree. The feature set was composed of a comprehensive set of around 120,000 layout-based, lexical, and syntactic features. They achieved an accuracy of 94% and 98% over 1600 and 250 programmers, respectively, on the GCJ dataset.

Dauber *et al.* [4] analyzed the source code files extracted from the open-source version control systems. They provided an extension of Caliskan-Islam *et al.* [5] work, which performs stylistic authorship identification on the source code sample with high accuracy. Firstly, they ensemble the output probabilities of sample source code files of the same author for the same classifier, which results in improved classification. Further, they were able to link several samples to the same programmer. Further, they used calibration-curves to prove the quality of authorship attribution for a given source code sample.

Burrow *et al.* [24] utilized n-gram features of the source code files to perform authorship attribution. Their work was inspired by the success of using n-gram based features in text authorship identification [12]. Results were reported on the tokenized representations of C language source code files for 100 authors.

Frantzeskou *et al.* [16] presented an approach called Source Code Author Profiles (SCAP). It utilizes the byte level n-grams and a similarity measure to predict the author of the source code. Experiments were conducted on Java and C++ programming languages with the number of authors ranging from 6 to 30 programmers. With 30 java authors, they were able to achieve 96.9% accuracy while they were able to achieve 100% accuracy for 6 C++ programmers.

Code authorship identification of the source code binaries has also been studied in the past. One such study was conducted by Caliskan *et al.* [25], where abstract syntax trees based author identification approach was used to extract authors from binaries. First, syntactical features are extracted using an abstract syntax tree. Further, these features are fed to the random forest classifier to yield authors of the binaries. The approach was evaluated using the GCJ dataset and reported an accuracy of 96% for 100 programmers and 86% accuracy for 600 programmers.

Meng *et al.* [23] proposed a framework to identify multiple authors in a binary file. They exploit the features that capture the programming style at the programming block level. The feature set includes the instruction features, control flow features, data flow features, and context features. Further, these features are fed into a joint classification model trained with Conditional Random Field (CRF). The study was conducted on three open-source projects: Apache HTTP server, the Dyninst binary analysis, instrumentation tool suite, and GCC. The dataset contains 147 binaries of C language and 22 binaries of C++ language. They were able to achieve 65% accuracy on 284 authors.

Despite the exciting results demonstrated by previous work, there are numerous limitations. First, most of the earlier studies' features limit the applicability to usually one language because of the feature engineering step's hand-crafted nature. Because of that, the features extracted for one particular language cannot be used to identify authors in other languages. However, Abuhamad *et al.* [3], [7] do address this issue by extracting language-oblivious features. However, they work on the keyword level, which limits the elements of style. Also, existing works use relatively small scale datasets, where the number of authors ranges from a few hundred to a few thousand authors.

Numerous loss functions have been presented in previous studies to solve few-shot learning problems, including contrastive loss [26], triplet loss [27], Quadruplet loss [28], and lifted structured loss [8]. These loss functions have been shown to perform well in few-shot learning settings. The contrastive loss function was proposed to minimize the embedding distance between positive pairs and maximize the distance between the negative pairs. Triplet loss function aims to pull an anchor point closer to the positive point and increase the distance between the anchor and negative points by a fixed margin. Quadruplet loss was proposed to address the

limitations of triplet loss. Chen *et al.* [28] argue that a model trained using triplet loss function would still show a relatively large intra-class variation. They proposed to address this problem by adding an extra penalty in the loss function, which forces the two negatives in the quadruplet to maximize the distance between them. For the lifted structured loss function, the idea is to extract as much information from each batch as you can. This is done by considering all possible negative for each anchor and positive in the batch. So the loss function will force to push away all the possible negative while bringing the anchor and positive closer together. We used the lifted structured loss to train our CNN.

IV. BACKGROUND

A. DEEP REPRESENTATION LEARNING

Code authorship attribution is often formulated as a classification problem. However, the authorship identification problem largely depends on each author's unique features, such as variable naming conventions, use of *for*, or *while* loop. Representation learning as a means to automate rich and distinctive feature engineering has gained increasing attention in the machine learning community and has become a field in itself. Representation learning has been previously used in multiple applications such as source code authorship attribution [3], human motion classification [29], and person re-identification task [30]. This study used representation learning, where we train CNN using lifted structured loss function to extract meaningful feature vectors.

Convolutional Neural Network: CNN is a type of neural network used for solving visual (image, video) tasks. However, it has also been used to solve different natural language processing tasks such as authorship attribution [31], Sentiment analysis [32], and document summarization [32]. CNN architecture has three main components, i.e., convolutional layer, pooling layer, and dense layer. We will briefly explain these three components below.

The convolutional layer is responsible for extracting features from the input vector by convolving it with filters. Each filter within a convolution is applied to the input. We do so by computing the dot product between the filter and the input. Filters are usually randomly initialized from a pre-determined distribution. The convolution outputs the features maps, which are then passed to the pooling layer.

The pooling layer reduces the dimensionality of each feature map in order to extract meaningful features. The pooling layer can be of different types: max-pooling, average-pooling, and sum-pooling are some of the examples. For instance, the max-pooling layer extracts the feature map's largest element within a region/window. Similarly, the average-pooling layer takes the average of all the elements in a particular window/region.

A dense layer is a single-layer of neurons that applies a dot product between the neuron weights and the input vector. If it is the neural network's last layer, A softmax activation function is further applied to this layer's output to solve the classification problem. The softmax function returns a probability distribution, which can then classify the input vector into its respective class label.

B. K-NEAREST NEIGHBOUR CLASSIFIER

K-Nearest Neighbour classifier (KNN) is an instance-based lazy learning algorithm that classifies objects based on the feature space's closest training instance. Various distance metrics can be used to determine the closeness between instances, e.g., Manhattan-distance and cosine similarity. KNN is one of the simplest classification algorithms as it does not require any prior knowledge about the distribution of the data. The Nearest Neighbour classifier is the simplest form of KNN when $K = 1$. KNN classification primarily has two stages:

1) Based on distance metric, pick the k nearest instances to a given instance i.
2) Then, based on the class labels of the k nearest instances, the class of instance i is determined.

V. CNN BASED AUTHORSHIP IDENTIFICATION SYSTEM

The authorship identification problem largely depends on the unique features associated with each author. Thus, the feature engineering process can become quite extensive. To enable a distinctive feature extraction process, we present an efficient CNN based approach to learn a novel deep representation that maps the source code to a fixed-length vector. These representations are further fed to the KNN classifier to identify the source code authors. The primary motivation for utilizing the KNN classifier is its lazy-learning ability, which makes our approach more efficient than the ones used by previous studies. We enforce the deep representations to behave in such a way that enables us to use the KNN classifier effectively. To incorporate this property in the embedding vectors, we leverage the lifted structured loss. The lifted structured loss allows us to learn deep representations such that embedding vectors of source codes authored by the same author are closer to each other in the embedding space than source codes written by different authors. Figure 1 shows an overview of our author attribution framework. We briefly highlight the different phases of our approach in this section and explain each phase's details in the subsequent subsections.

A. CHARACTER-LEVEL REPRESENTATION

We use the character-level vectors to represent the source code files, where each character is represented by a unique integer number. A source code S composed of n characters is denoted as $S = \{c_1, c_2, \cdots, c_n\}$. The character-level representation has several advantages over word-level representation. First, the vocabulary is much smaller than the one required by word-level representation. For instance, in the English language, we only need 97 characters, including all punctuation marks, while word-level vocabulary can be tens of thousands of words. Every single word can be formed using the character level representation. Simultaneously, the word-level representation can only naturally handle

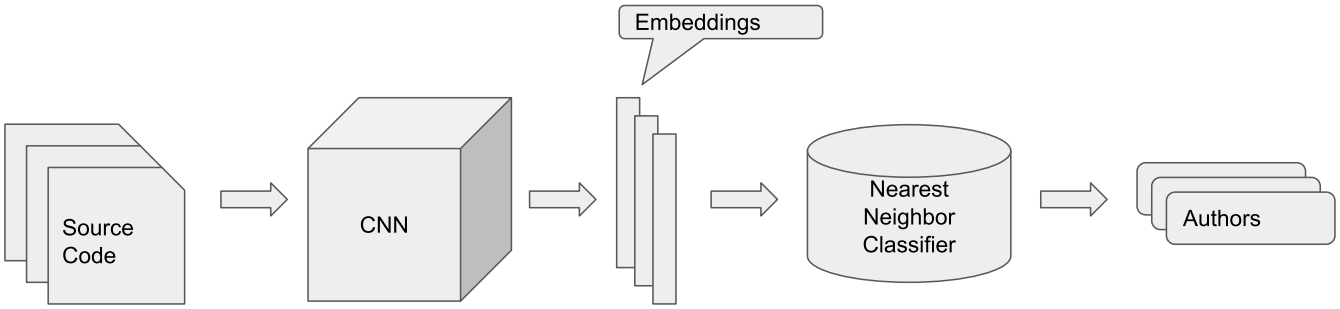


FIGURE 1. Flow diagram depicting the high level illustration of our proposed framework. Source codes are converted into character-level vectors, these vectors are then fed to CNN to learn deep embeddings, and then deep embeddings are further fed into KNN to classify authors.

the words that are part of its vocabulary. This flexibility is one of the critical advantages of character-level embedding over word-level embedding, especially considering that programming languages continuously evolve, and new keywords additions are frequent. However, the disadvantage of using character-level representation is that it results in longer sequences, and the model requires more computation resources to train if used naively.

B. NETWORK ARCHITECTURE

We use a CNN based architecture inspired by Ruder *et al.* [31]. Our network takes a character level vector as an input. Here, each character is represented by an integer, as discussed in subsection V-A. Further, the sequence is passed through an embedding layer such that it maps each integer onto a 128-dimensional embedding vector. After the embedding layer, we use the spatial dropout layer [33] with a 25% dropout and helps the neural network avoid overfitting. We then use four 1-D convolution layers consisting of 512 filters with kernel sizes of 2, 3, 5, and 7. We use he-normal initialization [34] to initialize convolution filter weights. Further, to add non-linearity we applied ReLU activation function [35]. To condense the feature maps and retain the most important features, we use global max-pooling operation over time [36], which outputs four 512 dimensional vectors against each convolution layer. We concatenated these features to form a 2048 dimensional vector and applied a 50% dropout on this vector. We fed this output to a 256-dimensional dense layer and applied the tanh activation function on the dense layer output so that every dimension has an upper and lower bound. We used the glorot-uniform initialization [37] to initialize the weights of the dense layer. The hyper-parameters of our CNN architecture were chosen after various iterations. Figure 2 shows the high-level illustration of our CNN architecture.

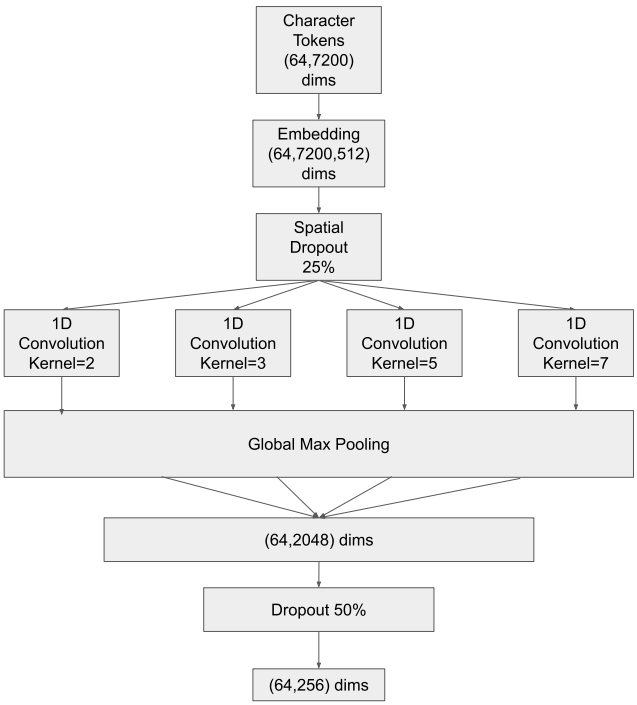


FIGURE 2. A high-level illustration of the character-level convolutional neural network used to extract deep representation of the source code.

C. LOSS FUNCTION

The ability to quantify the pairwise similarities between examples makes the learning problem a lot simpler. Given a similarity function, a classification task can be solved with a simple nearest neighbor classifier [8]. In contrast to conventional classification approaches, metric learning has become increasingly popular due to its ability to learn the general concept of distance metrics and its compatibility with efficient distance-based inference on the learned metric space [8].

We train our network using the lifted structured loss function [8]. The lifted structured loss function is used for the 256 dimensional embedding layer. The lifted structured loss minimizes the distance between all the similar embedding vectors and maximizes the dissimilar embedding vectors' distances. Equation 1, shows the lifted structured loss function.

$$\tilde{J}_{i,j} = \log\left(\sum_{(i,k)\in\mathcal{N}} \exp\{\alpha - D_{i,k}\} + \sum_{(j,l)\in\mathcal{N}} \exp\{\alpha - D_{j,l}\}\right) + D_{i,j}$$
$$\tilde{J} = \frac{1}{2|\mathcal{P}|} \sum_{(i,j)\in\mathcal{P}} \max\left(0,\ \tilde{J}_{i,j}\right)^2 \quad (1)$$

Here, $\mathcal{P}$ denotes the set of positive pairs in the batch, $\mathcal{N}$ denotes the set of negative pairs in the batch, α is the margin parameter, and $D_{i,k}$ is the distance between sample i and sample k.

D. CNN TRAINING

We train the neural network using Ranger optimizer, a combination of the Radam [38] and the Lookahead [39] optimizers. RAdam stabilizes the training at the start, and LookAhead stabilizes the training and convergence during the rest of the training process. We train the network for 250,000 steps. We utilize the learning rate warm-up, as it has been shown to work well to stabilize the early training process [39]. The learning rate was increased linearly from 0 to $1e{-}3$ in the first 1% of the training process, and then it was decreased linearly from $1e-3$ to $1e-6$ over the rest of the training process.

The dataset's prior distribution reveals that code files are not evenly spread across authors, so random sampling of authors and their respective files for training might induce a bias as authors with more files might not get proper representation. To address this issue, we sample an author based on the prior distribution of source code files such that the authors with more source code files get sampled more in each iteration. Here, we only select two source code files from each author's files and put them in a batch. We do this to maximize the number of unique authors within each batch. We repeat the process while ensuring that no author appears more than twice in a batch. So with this in mind, we selected a batch size of 64, which means that we have precisely 32 authors per batch. This process is called massaging the batch and is usually used in imbalanced class problems.

We train our CNN using TensorFlow,[2] an open-source framework used to built machine learning applications such as neural networks using data-flow graphs. We executed our experiments on a workstation with GeForce GTX 1660 Ti GPU (6GB) and 16GB of RAM. With this specification, our CNN took around 48 hours to train.

E. K-NEAREST NEIGHBOUR CLASSIFIER

After training the CNN, we extract the deep embedding vectors of all source code files. Further, some of these deep embedding vectors are used for the KNN classifier as the training dataset, and the rest are used as a test dataset to evaluate the classifier. If a source code vector V is 'closer' to a vector belonging to a particular author, then the probability of V being authored by the same author is high. To determine this 'closeness', we use Manhattan-distance, as it has been shown to perform well in high dimensional space in a wide variety of tasks [40].

Abuhamad *et al.* [3] utilized the Random Forest classifier, which has a training time complexity of $\mathcal{O}(n^2dt)$, where n is the number of samples, d is the number of dimensions, and t is the number of trees. The prediction time complexity of this classifier is $\mathcal{O}(dt)$. However, a naive KNN has a prediction complexity of $\mathcal{O}(dn)$. We use a KNN implementation that utilizes the ball trees data structure [41] to look up nearest neighbors, which makes our model much more efficient and performs prediction with $\mathcal{O}(d \log n)$ complexity. Also, being a lazy-learning classifier, KNN does not require a training phase. We used scikit-learn[3] to implement KNN classifier, where K=1. We empirically chose K=1 because it gave us the best performance overall.

VI. EVALUATION

This section discusses the steps employed to perform authorship identification experiments and their results under various settings. (A) We introduce the dataset used in the experiments. (B) We discuss how we obfuscated the source code files. (C) We provide the data pre-processing steps employed to clean the dataset. (D) We discuss how we extracted deep embedding vectors of the source code files. (E) We provide an analysis to check the goodness of the deep embedding vectors. (F) We discuss how we performed authorship attribution using deep embedding vectors. (G) We present the results of our proposed approach under different settings. (H) Finally, we discuss an initial experiment to address the open-world aspect of the authorship attribution problem.

A. DATA COLLECTION

For our study, we chose to work with the Google Code Jam (GCJ)[4] dataset. GCJ is an international programming competition organized annually by Google. There are multiple rounds, and each round requires writing a program to solve several problems within the allocated time. The source codes' public availability makes the GCJ a valuable resource to address the authorship attribution problem. The three most popular programming languages in GCJ are C++, Java, and Python. We gathered GCJ data of these programming languages from the year 2008 to 2019 to conduct our experiments.

We split the dataset into training, validation, and test sets across different years. The source codes written in the year 2018 are used to train the CNN. Source code files written between 2009-2017 and 2019 are used as test sets, while source codes written in the year 2008 are used as a validation set. Figure 3 shows the number of source code files of different languages across the train, test, and validation datasets.

B. SOURCE CODE OBFUSCATION

We also obfuscate the source code files belonging to each of the programming languages. Numerous obfuscation tools are available for each programming language. However, we decided to use PyObfx,[5] Stunnix,[6] and JavaSourceCodeObfuscator[7] to obfuscate Python, C++ and Java source codes, respectively. We used the default settings for all the

[2]https://www.tensorflow.org/

[3]https://scikit-learn.org/

[4]https://codingcompetitions.withgoogle.com/codejam

[5]https://github.com/PyObfx/PyObfx

[6]http://stunnix.com/prod/cxxo

[7]https://github.com/veylence/JavaSourceCodeObfuscator

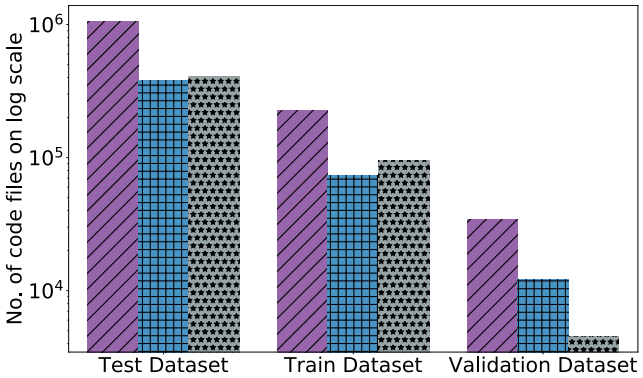


FIGURE 3. Number of source code files associated with different languages across different datasets. Here, the y-axis shows the number of source code files on the log scale (base 10) while the x-axis shows different datasets.

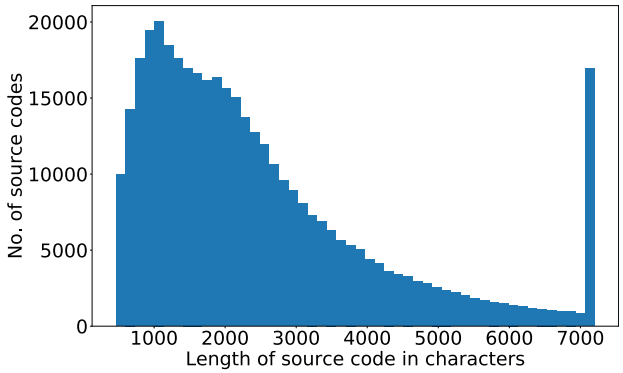


FIGURE 4. Distribution of source code files length. Here the y-axis shows the number of source code files while the x-axis shows the length of source code files in number of characters.

obfuscation tools. Also, these tools obfuscate the source code such that the code's functionality remains the same.

The PyObfx tool is a popular python source code obfuscator that obfuscates the source code by assigning a cryptic look to the strings, integers, floats, and booleans. Moreover, it also changes the variable names and imported libraries' names to random non-recognizable strings. Stunnix is a sophisticated C/C++ obfuscator which replaces the symbol name, numeric constant, characters into non-recognizable strings. It also removes the spaces, tabs, and comments from the source code. The JavaSourceCodeObfuscator tool is an off-the-shelf Java source code obfuscator that renames the classes, interfaces, methods, parameters, fields, and variables to random alphabetic non-recognizable strings. All the aforementioned obfuscation tools are randomized such that re-executing the obfuscator on the same source code yields different results.

C. DATA PRE-PROCESSING

The following are the pre-processing steps employed before conducting our experiments:

1) We only keep source code for the three most popular languages used in GCJ, C++, Java, and Python. We identified the language of the source code files using their file extension.

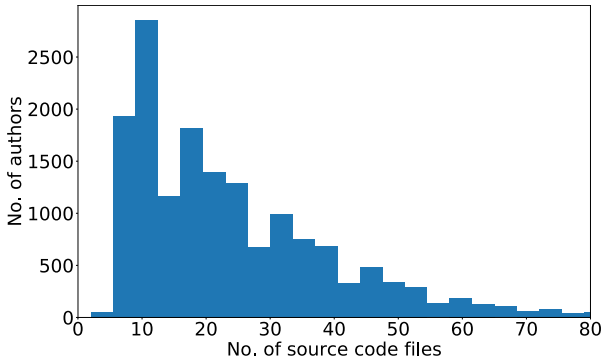


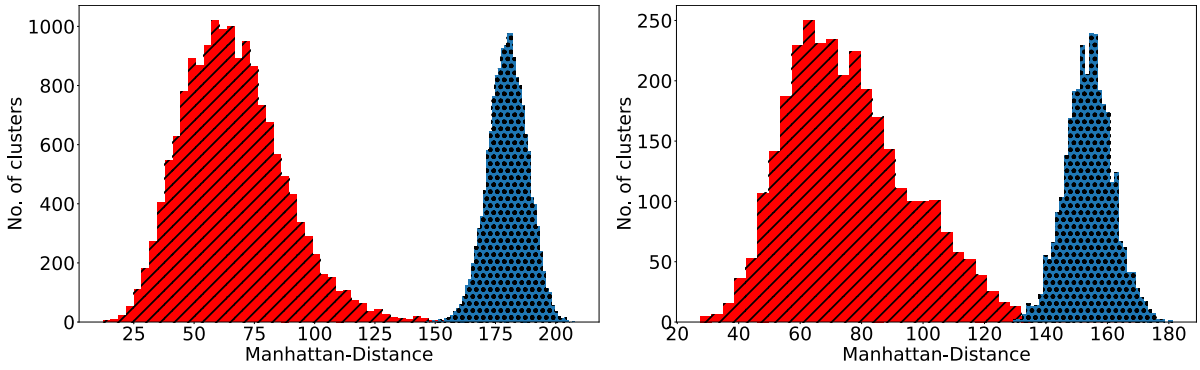
FIGURE 5. Distribution of number of source code files per author. Here, the y-axis shows the number of authors while the x-axis shows the number of source code files.

2) We remove duplicate source code files from the dataset.
3) We only keep the source codes that were processable by the obfuscators. We also remove the source codes whose obfuscated version is the same as the original version. This can arise due to a multitude of reasons, such as error while parsing and syntactically incorrect source codes.
4) We remove all the users with less than two source code files, which can be either obfuscated or original.
5) We remove the source code files with less than 465 (0.01 percentile of source code length distribution) characters. We used this lower limit to ensure that our model receives only source codes that contain some distinctive features of the author.
6) We only consider the first $7,200$ characters of the source files based on the 95th percentile of the distribution of source code length.

After the pre-processing step, we are left with 2,680,120 code files in total. These include obfuscated, and the original source code files across 11 years. We note that we end up with slightly more obfuscated files than original source-code files due to the minimum length cut-off. Figure 4 shows the distribution of the length of the source code files after pre-processing. According to this distribution, 50% of the source code files have a length of less than 2056 characters. Moreover, the distribution of the number of source code files against the number of authors can be seen in figure 5. We limit the visualization to 80 source code files in figure 5, as the distribution is right-tailed. According to this distribution, 50% of the authors have less than 20 source code files.

D. DEEP EMBEDDING EXTRACTION

We transformed the source code files, both original and obfuscated, to character level embedding vectors, as discussed in Section V-A. In our training data, there are 16,019 unique authors and $1,525$ unique characters. The *UNK* token represents any new character that was was not part of the training set. Source code samples can be of different lengths, so we padded them with 0 to make them of $7,200$ character length. We choose a $7,200$ character length because it is the 95th percentile of the source code character length in



(a) Inter and Intra-cluster distances of training set.

(b) Inter and Intra-cluster distances of validation set.

FIGURE 6. Distribution of inter and intra-cluster distances in training (2018) and validation set (2008). Here, the "dotted" distribution plot shows the inter-cluster distances while the "striped" distribution shows the intra-cluster distances.

our training dataset. Further, these character-level vectors are fed to CNN to learn deep embedding vectors.

CNN Training Assumption: In a real-world setting, a significant chunk of the source code files could be obfuscated. If trained with only the original source code files from GCJ, our network could not maintain a good accuracy under the simulated real-world setting. To identify source code authors under such a setting, the underlying network needs to be trained using both obfuscated and the original source code files. In this way, the model can extract features that can work with both kinds of source codes. Hence, to evaluate our approach using different experiments, we trained our network on a dataset that contains both obfuscated and the original source code files.

E. ANALYSIS OF EMBEDDING VECTORS

We calculated the inter and intra-cluster distances between the embedding vectors under the simulated real-world setting to ensure the deep embedding vectors' quality for the classification task. Here, a cluster represents a unique author. For each author in the training set, we calculated the mean of the Manhattan-distances between the embedding vectors of all the source code files to get the mean intra-cluster distance. Further, we also calculated the mean Manhattan-distances between the mean embedding vectors of each cluster (author) to get the mean inter-cluster distance between clusters. We assessed the significance of the difference between inter and intra-cluster distances with the null hypothesis that the two groups have the same mean and the alternate hypothesis that they are not. The p-value for the t-test that the inter and intra-cluster distances are drawn the same distribution and have equal means is less than 0.001, which indicates that the inter and intra-cluster distances are significantly different.

Figure 6 shows the inter and intra-cluster distance distributions to illustrate the quality of our deep representations on both training and validation sets. For the year 2018, The average distance between embeddings within each cluster is 66.42, with a standard deviation of 20.95, while the average distance between the means of different clusters is 180.01, with a standard deviation of 8.17. The minimum distance between the cluster means for 2018 was 146.35. Only 34, which is 0.21% of the total number of clusters, were found to have an average embedding distance greater than that value. For the year 2008, The average distance between embeddings within each cluster is 76.01, with a standard deviation of 20.06. The average distance between the means of different clusters is 153.93, with a standard deviation of 7.61. The minimum distance between the cluster means for 2008 was 129.55. Only 19, which is 0.6% of the total number of clusters, were found to have an average embedding distance greater than that value. We observe that the intra-cluster distances are significantly lower than the inter-cluster distances. This means that the distance in the embedding space has good discriminative power and can discriminate between different authors' source code files. We can also see that the distributions across training and validation sets are quite similar, reflecting that our model did not overfit and generalize well.

For qualitative evaluation, we sampled random authors from the validation dataset and visualized their respective source code embedding vectors using the Incremental Principal Component Analysis (IPCA) method [42]. IPCA is helpful when data cannot fit in memory due to memory constraints. It allows us to input the data in batches. Using IPCA, we reduce the dimensionality of the data to two dimensions (principal components). The first two principal components capture 10.20% and 8.50% of the variance of data, which illustrates that most of the dimensions learned by the model are meaningful. As we can see in figure 7, even with such low variance coverage of the first two components, we can get well-formed source code clusters.

F. CLASSIFICATION

We report the results by randomly sampling n source code files belonging to a particular author, i.e., five files per author, seven files per author, or nine files per author. Further, we feed $n - 1$ source code files to initialize the KNN and the remaining 1 source code file to evaluate the KNN. We use the Manhattan-distance metric to find the nearest code sample. It has been shown in the literature that Manhattan-distance has exceptional performance when

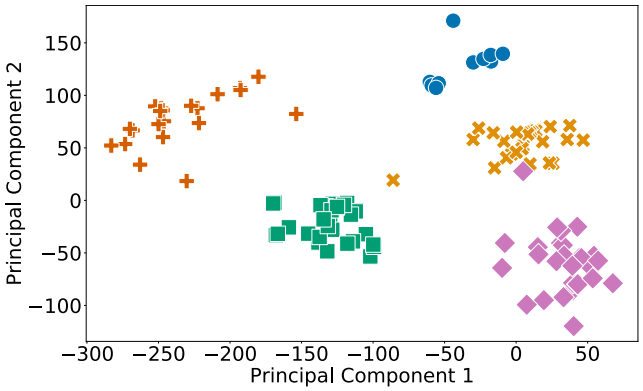


FIGURE 7. Embedding visualization after using Incremental Principle Component Analysis. The visualization shows source code sample representations for five randomly sampled authors. Here, source code files for the same author are depicted by the same shape and figure.

compared with many other distance metrics in high dimensional space [40]. To report the results, we repeat all our experiments ten times and calculate the accuracy scores' mean and standard deviation. Here accuracy metric is appropriate because our test dataset is not unbalanced, as every author has precisely one source code file in the testing set.

G. EXPERIMENT RESULTS

In this section, we discuss the results of our experiments conducted across different scenarios. We conducted our experiments for three programming languages separately, and we also consider a mixed scenario in which source codes authored in all three programming languages are considered. The motivation for including a mixed scenario is to address the authorship attribution problem under the assumption that an author can also write code in multiple programming languages. Here, there might be a case for a particular author in a mixed scenario such that files in the KNN training dataset are of one language, and files in the KNN test dataset might be of another language. For instance, the KNN training set contains Python and C++ code samples, and the test set contains Java code samples. It is important to note here that not every author's source code needs to be written in multiple languages. The primary difference here is that before all the prior source code files for all the authors were written in one programming language, our priors consist of all three languages. We report Top-1 and Top-3 accuracy of the results. Here, Top-N accuracy means that the correct author prediction is in the top N predictions by the KNN classifier. We discuss the results in the subsequent sections: (1) We discuss the results of our proposed approach over the original GCJ source code files. (2) We discuss the performance of our approach over obfuscated source code files. (3) We discuss our simulated real-world setting results, created by combining both original and obfuscated source code files.

1) RESULTS ON ORIGINAL GCJ DATASET

This subsection explores the authorship attribution problem on the original source-codes mined from the GCJ competition. Table 2 shows the results achieved by our proposed framework under this particular setting. As we can observe, across all the programming languages and even under the mixed language scenario, we achieve similar results for 5, 7, and 9 files. We also see that the test years' results are similar to the results obtained in the training year, which shows that the model is not overfitting.

a: FIVE FILES PER AUTHOR

For the mixed-language scenario, we see an average −1.8% difference in all years' accuracy from 2018. For C++, we observe an average difference of −3.6%, while for Python and Java, we note an average difference of 4.75% and −0.60%, respectively. Here, the "negative" difference means that our model performs worse on test data than the training set.

b: SEVEN FILES PER AUTHOR

For C++, we observe an average difference of −1.80%, while for Python, we note an average difference of 5.20%. For Java, we observe an average difference of 0.61%. Finally, for the mixed-language scenario, the average difference of the accuracy is only −0.525%.

c: NINE FILES PER AUTHOR

For C++, we observe an average difference of −0.59%, while for Python, we observe an average difference of 4.23%. For Java, we note an average difference of 0.86%, while for the mixed-language scenario, we report an average difference of 0.16%.

2) RESULTS ON OBFUSCATED GCJ DATASET

Source code is often obfuscated in an attempt to hide the authorship. In this experiment, we examine how obfuscation affects our approach. For this, we first obfuscate our source code using different source code obfuscation tools, as described in section VI-B. We only sample obfuscated source code vectors from the set of source codes. We computed the average difference of accuracy between obfuscated and the original source-code setting to observe our model's performance trend in the obfuscated setting. Table 3 shows the results achieved by our proposed framework under this particular setting.

a: FIVE FILES PER AUTHOR

We observe an average difference of −10.6% for mixed-language scenario between obfuscated and original source-code setting. For C++, we see an average difference of −8.12%. While for Python and Java, we note an average accuracy difference of −13.2% and −14.1%, respectively.

b: SEVEN FILES PER AUTHOR

For mixed-language scenario, we observe an average difference of −8.71% between obfuscated and the original source code setting. For C++, we see an average difference of

TABLE 2. Top-1 and Top-3 accuracy of authorship attribution across different number of source code files per author, different years and programming languages under original source-code setting.

		C++			Python			Java			Mix-Language		
Year	Files	Authors	Top-1 Acc	Top-3 Acc	Authors	Top-1 Acc	Top-3 Acc	Authors	Top-1 Acc	Top-3 Acc	Authors	Top-1 Acc	Top-3 Acc
2008	5	1536	83.92±0.5	89.62±0.6	234	92.86±1.1	96.41±1.2	608	85.03±1.0	89.7±0.7	2407	84.45±0.5	89.96±0.6
	7	1059	87.89±0.6	93.75±0.5	101	96.63±1.9	98.51±1.2	306	89.18±1.5	94.38±0.9	1489	88.32±0.9	93.45±0.4
	9	801	91.79±0.9	95.53±0.6	36	98.61±1.4	99.17±1.8	173	93.29±1.0	96.99±0.8	1030	91.58±0.9	95.33±0.5
2009	5	2400	79.66±0.9	86.47±0.4	370	87.68±0.9	94.73±1.3	790	81.89±1.2	89.05±1.1	3606	80.05±0.6	86.92±0.4
	7	1726	86.47±0.7	92.48±0.5	208	95.53±1.2	98.22±1.2	495	88.79±1.1	94.02±1.2	2499	86.49±0.5	92.33±0.5
	9	1104	91.69±0.9	95.59±0.6	94	97.55±1.3	98.62±1.0	280	93.36±1.2	96.39±0.9	1529	91.64±0.9	95.58±0.5
2010	5	2379	87.85±0.7	92.53±0.4	418	95.17±1.1	97.3±0.5	855	91.82±0.7	95.06±0.8	3799	86.3±0.5	90.61±0.6
	7	1597	92.69±0.5	95.56±0.4	229	97.95±0.8	98.73±0.7	561	95.47±0.8	97.75±0.5	2556	90.7±0.4	93.65±0.3
	9	958	95.45±0.5	97.39±0.5	112	99.29±0.9	99.64±0.4	333	97.0±0.7	98.65±0.5	1545	93.53±0.4	95.32±0.5
2011	5	3746	88.87±0.3	93.28±0.5	746	94.36±0.8	96.96±0.5	1372	89.14±0.6	93.48±0.4	5925	88.93±0.2	92.94±0.3
	7	2852	92.8±0.4	96.12±0.3	444	96.98±0.8	98.38±0.5	941	92.54±0.8	95.56±0.4	4315	92.57±0.4	95.86±0.2
	9	1951	95.3±0.4	97.63±0.3	233	98.33±0.6	99.44±0.6	608	95.39±0.7	97.43±0.5	2872	95.22±0.4	97.17±0.3
2012	5	3319	85.88±0.8	90.95±0.5	713	92.09±0.9	94.77±0.7	1248	87.72±0.6	91.8±0.5	5402	86.07±0.4	90.78±0.4
	7	2100	91.86±0.4	94.9±0.6	336	95.09±1.1	97.8±0.9	617	92.38±1.4	95.41±0.8	3154	91.24±0.3	94.77±0.4
	9	1237	94.41±0.7	96.65±0.5	151	97.42±1.2	98.48±0.7	339	95.4±1.0	97.73±0.8	1796	94.29±0.4	96.34±0.7
2013	5	4565	86.75±0.3	91.97±0.3	1221	92.38±0.7	95.39±0.5	1953	87.1±0.8	91.97±0.5	7963	85.68±0.3	90.84±0.3
	7	2911	92.22±0.4	95.76±0.3	617	96.03±0.9	98.02±0.4	1072	92.47±0.6	95.43±0.5	4839	91.11±0.1	94.53±0.3
	9	1795	95.07±0.4	97.45±0.3	301	97.71±0.8	99.14±0.3	596	95.25±0.7	97.28±0.7	2889	94.01±0.2	96.18±0.3
2014	5	5091	88.18±0.3	92.62±0.3	1360	93.51±0.7	96.51±0.4	1936	88.03±0.7	92.54±0.4	8481	88.07±0.3	92.29±0.2
	7	3212	92.84±0.5	96.04±0.2	663	96.79±0.2	98.48±0.3	1035	93.39±0.6	96.46±0.7	5039	92.76±0.3	95.83±0.2
	9	2114	95.35±0.4	97.71±0.3	305	98.49±0.8	99.15±0.4	595	95.61±0.8	97.26±0.6	3127	95.09±0.3	97.13±0.2
2015	5	4516	89.98±0.3	94.06±0.3	943	94.92±0.6	97.25±0.4	1340	91.32±0.6	95.04±0.4	6884	90.08±0.3	93.87±0.2
	7	3199	93.73±0.2	96.48±0.3	555	97.66±0.6	98.25±0.5	827	94.73±0.5	97.29±0.5	4687	93.44±0.2	96.19±0.2
	9	2158	95.9±0.4	97.88±0.2	293	98.74±0.6	99.28±0.6	500	96.68±0.7	98.78±0.6	3046	95.63±0.4	97.62±0.2
2016	5	6394	87.39±0.3	91.99±0.3	2364	87.65±0.5	92.35±0.5	2602	84.76±0.7	90.2±0.4	11626	85.13±0.3	90.01±0.1
	7	4531	92.08±0.3	95.6±0.2	1430	93.83±0.6	96.55±0.5	1680	90.83±0.9	94.98±0.3	7924	90.66±0.3	94.42±0.1
	9	3091	95.06±0.4	97.11±0.2	801	96.24±0.6	98.09±0.3	1058	94.02±0.8	96.86±0.5	5185	93.99±0.3	96.4±0.3
2017	5	5859	91.09±0.4	95.1±0.3	1858	92.08±0.4	95.71±0.3	1699	88.59±0.6	93.36±0.3	9538	89.88±0.2	94.02±0.1
	7	4032	95.02±0.3	97.58±0.1	1050	96.29±0.3	98.36±0.5	977	94.23±0.7	96.86±0.3	6204	94.43±0.3	97.04±0.2
	9	2702	97.03±0.4	98.64±0.2	586	98.0±0.5	99.13±0.4	548	96.33±0.7	98.08±0.4	3972	96.66±0.2	98.28±0.2
2018	5	7793	90.76±0.3	93.99±0.2	3414	86.96±0.4	91.32±0.3	2724	87.92±0.6	91.88±0.3	13847	88.1±0.3	91.77±0.2
	7	6338	93.63±0.3	96.25±0.2	2534	90.5±0.5	94.4±0.3	2078	91.48±0.5	94.9±0.3	11035	91.52±0.3	94.71±0.2
	9	5247	95.29±0.3	97.31±0.2	1976	93.27±0.6	96.02±0.4	1606	94.04±0.6	96.56±0.3	8947	93.77±0.2	96.12±0.2
2019	5	9746	89.3±0.2	92.22±0.2	5959	86.18±0.3	89.13±0.3	3893	85.08±0.5	88.43±0.5	18963	84.85±0.1	88.1±0.2
	7	7805	92.57±0.3	94.97±0.3	4593	89.88±0.3	92.92±0.4	3050	89.04±0.4	92.13±0.4	15439	89.22±0.2	91.92±0.2
	9	6238	94.63±0.1	96.41±0.2	3567	92.18±0.3	94.42±0.4	2355	91.66±0.3	94.42±0.3	12498	91.67±0.2	93.73±0.1

−6.86%. While for Python and Java, we observe an accuracy difference of −10.0% and −10.9%, respectively.

c: NINE FILES PER AUTHOR

We observe an average difference of −7.04% for mixed-language scenario between obfuscated and original source code setting. For C++, we see an average difference of −5.75%. For Python and Java, we note an average accuracy difference of −7.85% and −8.32%, respectively.

Results show that our approach scales for a large number of authors under obfuscated settings and achieves high accuracy. However, this experiment results show some degradation in accuracy, compared to results achieved in the original source code setting, which is expected as the sole purpose of obfuscation is to hide the traits that can be used to identify the author. We also note that with the increase in the number of files, the accuracy difference drops by almost 3 points, making the gap even smaller.

TABLE 3. Top-1 and Top-3 accuracy of authorship attribution across different number of source code files per author, different years and programming languages under obfuscated setting.

		C++			Python			Java			Mix-Language		
Year	Files	Authors	Top-1 ACC.	Top-3 Acc	Authors	Top-1 Acc	Top-3 Acc	Authors	Top-1 Acc	Top-3 Acc	Authors	Top-1 Acc	Top-3 Acc
2008	5	1548	75.6±0.9	83.03±0.8	258	81.55±1.8	87.91±1.7	606	73.22±0.9	83.05±1.0	2440	74.62±1.0	82.92±0.9
	7	1064	80.2±0.6	87.5±0.8	113	90.0±3.0	95.13±1.5	303	81.65±2.1	89.54±1.4	1505	79.82±0.8	87.36±0.6
	9	807	83.27±1.4	90.48±1.2	39	93.59±3.5	96.92±1.9	173	86.53±2.2	93.01±1.9	1040	84.12±0.9	90.51±1.0
2009	5	2407	69.99±0.6	78.87±0.7	399	72.56±1.3	82.26±1.3	789	68.87±1.9	79.84±1.5	3643	69.23±0.5	78.3±0.4
	7	1734	77.87±0.8	85.59±0.5	225	85.16±1.3	91.42±1.2	495	80.1±2.0	87.98±1.1	2525	78.11±0.8	85.85±0.5
	9	1111	85.1±0.9	90.32±0.6	99	91.21±2.3	93.94±2.1	281	87.37±1.3	92.85±1.3	1547	84.58±0.6	90.58±0.6
2010	5	2578	78.2±0.6	84.71±0.5	519	84.35±1.4	90.56±1.9	856	78.97±1.1	87.04±0.9	4088	75.53±0.6	82.87±0.4
	7	1799	84.34±0.7	90.14±0.8	308	89.25±1.9	93.99±1.4	567	86.21±1.4	93.02±0.9	2850	81.81±0.5	88.03±0.6
	9	1094	89.0±0.6	93.33±0.7	150	92.27±2.7	94.6±1.4	335	92.21±0.5	96.3±0.4	1768	86.82±0.9	91.35±0.6
2011	5	3793	80.98±0.6	87.9±0.4	788	83.64±1.3	89.82±0.9	1372	74.67±1.1	82.97±0.8	6013	79.02±0.4	86.29±0.3
	7	2902	86.03±0.4	92.07±0.5	490	88.94±0.9	93.84±0.7	943	82.13±0.6	88.96±0.7	4420	84.6±0.3	90.97±0.3
	9	2018	90.02±0.4	94.51±0.4	271	92.77±1.4	96.42±0.9	608	87.81±1.0	93.65±0.5	2985	89.4±0.2	93.98±0.3
2012	5	3359	78.35±0.6	85.45±0.4	783	82.89±1.6	88.97±0.9	1252	74.26±0.8	82.0±0.8	5525	76.58±0.5	83.68±0.4
	7	2136	85.25±0.5	90.51±0.6	360	89.42±1.6	93.53±0.9	618	80.58±0.9	87.91±1.6	3233	83.35±0.5	89.55±0.6
	9	1258	88.86±0.8	93.35±0.5	167	92.4±1.7	96.11±1.1	338	87.4±1.7	93.64±0.9	1842	88.14±0.8	92.48±0.5
2013	5	4641	78.65±0.3	85.68±0.4	1316	80.03±1.0	87.99±0.9	1954	73.08±0.5	82.21±0.5	8136	75.39±0.3	82.71±0.3
	7	2965	85.85±0.4	91.76±0.4	694	88.3±1.3	92.78±0.7	1072	81.87±0.9	88.96±1.1	5000	82.75±0.4	89.15±0.2
	9	1831	89.76±0.5	94.19±0.4	332	90.96±1.1	94.73±1.1	595	86.79±0.8	92.27±0.7	2993	87.7±0.4	92.54±0.3
2014	5	5241	78.95±0.4	85.96±0.5	1531	77.99±1.1	85.56±0.9	1936	71.61±0.6	81.01±0.8	8807	76.27±0.6	83.54±0.3
	7	3344	85.6±0.4	91.21±0.4	781	85.17±0.9	90.68±0.7	1035	80.77±1.2	87.61±0.6	5295	83.37±0.7	89.47±0.3
	9	2195	89.7±0.3	93.81±0.6	384	89.48±1.6	93.46±1.0	595	85.45±1.2	91.98±0.7	3301	87.99±0.4	92.44±0.5
2015	5	4690	82.79±0.4	88.75±0.2	1102	83.24±0.6	89.73±0.7	1343	76.25±1.3	84.3±1.0	7206	80.62±0.3	87.1±0.2
	7	3373	87.5±0.5	92.51±0.3	663	88.52±0.7	93.41±0.9	829	82.86±0.9	89.45±0.8	4993	85.63±0.5	91.3±0.2
	9	2307	90.82±0.4	94.92±0.4	382	92.77±1.1	95.63±0.9	499	87.21±1.0	93.25±1.0	3307	89.38±0.5	93.69±0.3
2016	5	6621	79.42±0.4	85.74±0.5	2902	69.02±0.5	78.33±0.6	2604	67.0±0.4	76.44±0.5	12337	72.41±0.2	80.35±0.2
	7	4730	84.8±0.6	90.36±0.3	1838	77.54±0.8	85.27±0.7	1687	75.39±0.7	84.16±0.8	8589	79.32±0.4	86.15±0.3
	9	3314	89.42±0.4	93.23±0.4	1116	84.03±0.9	89.76±0.9	1055	82.85±0.8	89.77±0.7	5785	85.05±0.4	90.44±0.3
2017	5	5929	83.56±0.3	89.23±0.4	2090	75.84±0.6	84.4±0.5	1698	73.3±0.9	82.54±0.5	9850	79.33±0.4	85.78±0.3
	7	4087	88.91±0.5	93.26±0.3	1189	83.96±0.9	90.02±0.5	976	82.87±0.5	89.74±0.7	6409	85.77±0.4	91.0±0.2
	9	2760	92.11±0.4	94.99±0.4	670	87.57±0.7	91.78±0.8	548	88.5±1.3	93.03±0.8	4141	89.44±0.4	93.56±0.4
2018	5	7857	83.86±0.3	88.75±0.4	3638	75.09±0.4	81.83±0.4	2719	75.52±0.5	81.08±0.7	14088	78.63±0.2	84.24±0.2
	7	6397	88.31±0.3	92.38±0.3	2734	79.66±0.4	86.51±0.3	2079	80.87±0.8	86.21±0.5	11274	83.61±0.2	88.58±0.2
	9	5293	90.37±0.3	94.04±0.2	2154	83.37±0.6	88.91±0.4	1601	84.18±0.7	89.1±0.5	9163	86.59±0.3	91.08±0.3
2019	5	10575	81.75±0.3	86.17±0.4	7647	70.74±0.4	76.49±0.4	3913	72.03±0.4	77.4±0.5	21437	73.34±0.3	78.38±0.2
	7	8458	86.71±0.4	90.49±0.3	5896	77.19±0.5	82.35±0.6	3057	78.03±0.4	83.21±0.5	17381	79.75±0.2	83.91±0.2
	9	6782	89.5±0.3	92.73±0.2	4596	81.27±0.4	85.61±0.4	2370	81.91±0.9	85.97±0.8	14100	83.41±0.2	87.13±0.2

3) SIMULATED REAL-WORLD SETTING

In this experiment, we try to simulate a real-world setting where we would have both obfuscated and the original source code files written in multiple programming languages. Here, source code vectors are sampled from the set of both obfuscated and original source codes. Table 4 shows the results for each programming language and mixed programming language scenario under the simulated real-world settings. To make our model truly obfuscation oblivious, it must work well in such a real-world setting. Given the random nature of the sampling technique, Our model may have only seen the original source codes written by a particular author. If an obfuscated source code of that author comes up, the model is expected to make a correct prediction. To determine our model's performance trend, we computed the average accuracy difference of the simulated real-world setting with obfuscated and original source-code settings. As expected, our results fall within the lower bound determined by the obfuscated setting and the upper bound determined by the original source code setting.

TABLE 4. Top-1 and Top-3 accuracy of authorship attribution across different number of source code files per author, different years and programming languages under simulated real world setting.

		C++			Python			Java			Mix-Language		
Year	Files	Authors	Top-1 Acc	Top-3 Acc	Authors	Top-1 Acc	Top-3 Acc	Authors	Top-1 Acc	Top-3 Acc	Authors	Top-1 Acc	Top-3 Acc
2008	5	1891	79.55±0.8	87.03±0.6	389	86.22±1.1	91.85±0.9	869	72.57±1.7	83.97±0.8	3121	77.42±0.5	85.93±0.4
	7	1878	85.8±0.9	92.0±0.7	364	91.37±2.3	95.58±0.9	836	80.16±1.1	90.08±0.8	3094	84.17±0.5	90.99±0.4
	9	1547	87.88±0.7	93.45±0.5	255	93.69±1.1	96.04±1.5	608	84.44±1.6	92.6±0.7	2439	87.5±0.5	92.79±0.5
2009	5	2794	74.85±0.5	83.08±0.6	523	81.2±1.4	88.59±1.2	1015	67.75±0.9	80.03±1.5	4290	73.24±0.6	82.24±0.6
	7	2765	82.1±0.6	89.28±0.3	502	87.39±1.5	92.31±1.1	985	75.82±1.3	86.16±1.1	4268	79.97±0.4	88.36±0.3
	9	2407	86.38±0.8	91.69±0.7	394	89.67±1.1	94.44±0.8	790	81.09±1.1	90.16±0.9	3638	84.88±0.3	90.92±0.4
2010	5	3129	80.91±0.5	87.77±0.5	687	88.75±0.8	93.92±1.1	1088	76.53±1.0	87.14±0.6	4825	78.07±0.6	85.66±0.6
	7	2994	86.52±0.2	91.99±0.3	621	93.01±1.0	95.93±0.7	1040	83.47±0.8	91.82±0.7	4733	84.24±0.4	90.37±0.4
	9	2530	89.8±0.7	93.98±0.3	483	94.58±0.6	97.27±0.5	858	87.09±0.6	94.5±0.7	4007	87.38±0.5	92.53±0.5
2011	5	4335	83.13±0.3	90.12±0.3	1028	87.44±0.8	93.67±0.9	1706	73.48±0.7	83.97±0.8	6983	81.06±0.4	88.44±0.3
	7	4293	88.27±0.4	93.89±0.4	979	92.21±0.7	95.72±0.4	1672	80.39±0.8	89.23±0.7	6943	86.36±0.2	92.57±0.3
	9	3786	91.05±0.4	95.36±0.4	774	94.15±1.0	96.98±0.9	1373	84.41±0.9	91.4±0.6	5995	89.48±0.3	94.48±0.2
2012	5	4193	82.03±0.6	88.88±0.6	1257	87.06±0.9	92.26±0.6	1813	73.65±0.7	83.44±0.4	7173	79.48±0.4	87.05±0.2
	7	4125	86.79±0.2	92.84±0.4	1152	91.39±0.6	95.32±0.7	1768	80.18±0.7	88.8±0.5	7133	85.24±0.3	91.59±0.3
	9	3353	90.05±0.3	94.58±0.4	765	93.56±0.8	96.55±0.5	1253	85.43±0.6	92.17±0.7	5499	88.92±0.3	93.62±0.2
2013	5	5911	81.37±0.6	88.03±0.4	1926	84.28±0.8	90.29±0.7	2672	70.98±0.8	80.78±0.7	10321	77.29±0.4	85.09±0.3
	7	5771	86.84±0.4	92.55±0.3	1790	89.78±0.5	94.15±0.3	2574	78.16±0.8	87.03±0.4	10256	83.55±0.5	90.2±0.2
	9	4636	90.23±0.4	94.75±0.4	1299	92.66±0.9	95.54±0.5	1956	83.33±1.0	90.78±0.5	8122	87.74±0.3	92.76±0.2
2014	5	6661	81.04±0.3	88.46±0.4	2162	83.91±0.7	90.01±0.4	2656	71.42±0.9	81.46±0.7	11337	78.62±0.3	86.14±0.3
	7	6527	86.52±0.3	92.58±0.3	2056	88.45±0.5	93.37±0.7	2616	77.44±0.8	86.87±0.5	11252	84.18±0.3	90.77±0.2
	9	5218	90.39±0.4	94.71±0.3	1478	91.41±0.7	94.65±0.7	1937	83.43±0.5	90.39±0.5	8741	88.13±0.4	93.37±0.3
2015	5	5848	85.3±0.4	91.3±0.2	1569	87.44±0.4	92.77±0.6	1872	76.13±1.0	85.22±0.4	9176	82.86±0.4	89.46±0.2
	7	5747	89.28±0.2	94.37±0.3	1469	91.28±0.6	95.11±0.7	1829	81.84±0.7	89.57±0.8	9058	87.56±0.2	92.87±0.2
	9	4643	91.58±0.4	95.53±0.4	1046	93.21±0.3	95.96±0.5	1343	86.37±0.9	92.3±0.8	7111	90.16±0.4	94.39±0.3
2016	5	8248	83.13±0.4	88.93±0.2	3749	76.07±0.4	84.23±0.5	3435	70.15±0.9	79.94±0.4	15165	76.97±0.2	84.31±0.3
	7	8039	87.93±0.3	92.89±0.2	3465	83.19±0.6	89.67±0.5	3361	77.52±0.4	85.75±0.2	14933	83.21±0.3	89.56±0.2
	9	6575	90.6±0.4	94.45±0.2	2718	86.39±0.6	91.78±0.6	2605	81.75±0.6	89.31±0.6	12130	86.65±0.1	91.76±0.3
2017	5	7422	86.17±0.2	91.77±0.4	2878	82.25±0.5	89.3±0.4	2414	74.2±0.5	83.88±0.7	12572	82.55±0.4	89.01±0.2
	7	7307	90.62±0.2	95.06±0.2	2715	87.63±0.6	92.94±0.5	2366	80.81±0.5	89.12±0.5	12484	87.24±0.2	92.71±0.2
	9	5922	92.83±0.2	96.19±0.2	2017	90.03±0.7	94.35±0.6	1699	85.5±0.8	92.25±0.4	9777	90.42±0.3	94.6±0.3
2018	5	8994	86.44±0.2	91.4±0.2	4311	79.62±0.6	86.78±0.4	3230	73.06±0.7	81.53±0.5	15984	81.0±0.2	87.05±0.3
	7	8840	90.63±0.2	94.54±0.2	4130	85.08±0.5	91.23±0.2	3151	80.18±0.5	86.91±0.6	15880	86.15±0.2	91.25±0.2
	9	7839	92.54±0.3	95.6±0.3	3578	88.06±0.6	92.63±0.5	2724	83.3±0.9	89.32±0.8	14031	88.97±0.2	92.87±0.2
2019	5	12251	83.71±0.3	88.42±0.3	9048	75.55±0.4	81.24±0.3	4657	68.32±0.4	75.43±0.6	23850	75.07±0.1	80.88±0.3
	7	11655	88.42±0.3	92.18±0.2	7994	81.35±0.3	86.26±0.3	4461	75.34±0.6	81.37±0.4	22929	81.29±0.3	86.26±0.3
	9	10280	90.98±0.3	93.76±0.2	6910	84.79±0.5	88.65±0.4	3911	79.36±0.6	84.81±0.3	20458	84.94±0.2	88.8±0.2

a: FIVE FILES PER AUTHOR

For C++, We note that the average difference between simulated real-world setting and obfuscated setting is −2.96%, and it drops to 5.16% when compared with the original source-code setting. For Python, the average difference with the obfuscated setting is −5.24%, and it drops to 8.00% compared with the original source-code setting. For Java, the average difference with the obfuscated and original source-code setting is 0.87% and 15.0%, respectively. Lastly, for the mixed-language scenario, the average difference with the obfuscated setting is −2.72%, and it drops to 7.82% when compared with the original source-code setting.

b: SEVEN FILES PER AUTHOR

We observe that the average difference between the simulated real-world setting and obfuscated setting is −2.36%. The average difference between the simulated real-world and the original source-code setting is 4.50% for C++. For Python, the average difference with the obfuscated setting is −3.25%, and it drops to 6.75% when the average difference is

computed with the original source-code setting. For Java, the average difference with the obfuscated setting is 1.83%, and this difference drops to 0.33% when the average difference is computed with the original source-code setting. Finally, for the mixed-language scenario, the average difference is −1.05% compared with the obfuscated setting, and it drops to 3.93% when computed with the original source-code setting.

c: NINE FILES PER AUTHOR

We see the average difference between simulated real-world setting and the obfuscated setting is −1.36%, and it drops to 4.38% when the average difference is computed between simulated real-world setting and original source-code settings for C++. For Python, the average difference with the obfuscated setting is −1.71%, and it drops to 6.13 when the average difference is computed with original source-code settings. For Java, the average difference is 2.73% when computed with obfuscated settings, and it drops to 11.04% when calculated with original source-code settings. Lastly, the average difference with obfuscated settings is −2.72% compared to the average difference with original source-code settings for the mixed-language scenario.

Our study presents the largest and the most diverse code authorship identification work by extracting the authorship of 20,458 programmers who write source code in multiple programming languages while maintaining high Top-1 accuracy, i.e., 84.94%.

H. OPEN WORLD

One of the advantages of utilizing this deep metric-learning-based approach is that we can use the similarity measure to address our problem's open-world aspect. We perform the authorship attribution by using a distance threshold to differentiate between the known and unknown authors (out of the world). We conducted an initial experiment to demonstrate our approach's applicability to identify whether an unknown author wrote a given source code. We restricted our experiment to the source code files written in the year 2008 of the GCJ competition under the simulated real-world setting with nine source code files. Here, selecting the right threshold to differentiate between known and unknown authors is critical. The threshold is generally dependent on the application because the precision and recall measures' importance may vary [4]. Furthermore, it is also reliant on the approach used. For instance, we used the distance threshold to differentiate between known and unknown authors. We are using KNN as a classification method, which classifies based on the closeness between instances.

To determine the threshold, we calculated the mean (68.92) and standard deviation (35.06) of the intra-cluster distances, as discussed in section VI-E over the year 2018, which was also used to train the neural network. We experimented with different distance thresholds within the range of two standard deviations of the mean intra-cluster distances. We then evaluated the thresholds in terms of precision and recall both

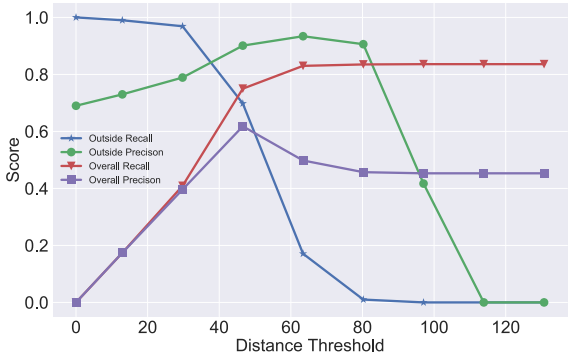


FIGURE 8. Precision and recall trend for our open world experiment against different distance thresholds. Here x-axis shows the distance threshold to differentiate between inside and outside of world authors while y-axis shows the precision/recall of our open world experiment. Here, we report the results for both outside (unknown) authors and overall (unknown and known) authors.

inside and outside the world authors. Figure 8 shows the results of our open world experiment. We note that as we increase the threshold, the recall for outside the world attribution decreases as expected. However, precision for outside world samples increases up to a point and then decreases sharply. This analysis can help evaluate the acceptable trade-off, which can result in choosing the right threshold. For instance, in figure 8, we can see that the optimal distance threshold is 51.41, where we were able to identify unknown authors with 84.0% recall and 88.5% precision. However, for the overall (unknown and known) authors, we were able to achieve 65.3% precision and 72.2% recall at a distance threshold of 51.41.

1) DISCUSSION OF RESULTS

Results show that our approach can identify authors across various programming languages and can scale for a large number of programmers under diverse settings. Our approach also shows similar results in the training and testing dataset, which indicates that our model does not overfit. We also note that the number of authors and performance does not have a direct relationship. The increase in the number of authors does not necessarily result in a decrease in performance. We hypothesize that this is due to particular kinds of programming problems that force the different programmers to solve them similarly. In contrast, other programming problems might give them more freedom, making the problem different across the years and impacting overall performance.

VII. LIMITATIONS

The following are the limitations that are associated with this study.

Multiple Authors: Currently, our study is based on the assumption that source code can only be attributed to a single author. However, in reality, source codes are usually authored by multiple authors, mainly when they belong to a project. Identifying multiple programmers in source code could be an exciting avenue to explore.

Real World Dataset: Currently, our approach is evaluated on the GCJ dataset. We tried to simulate the real-world setting, but the scenario might be different in the real-world dataset, such as the one mined from Github.

VIII. FUTURE WORK

In this section, we highlight the planned future direction of our work.

Binaries: Currently, our study identifies authors of the source code written in a specific programming language. However, a study by Caliskan *et al.* [5] showed that coding style is preserved in the compilation process, and authors can be extracted from the executable binaries. A large-scale study to identify the authors of binaries generated from obfuscated and the original source code files is an appealing avenue to explore.

Model Interpretability: Analyze what the model looks for in code while extracting the author. Is it looking for spaces vs. tabs? Function declarations? Indentation of code blocks? Line of codes? No. of characters in a line? Or something more complex.

Language Agnostic: Currently, our proposed approach is trained and tested on three known languages, i.e., C++, Python, and Java. We plan to analyze our model on unseen languages.

Model Architecture and Loss Function: We plan to investigate different architectures and loss functions.

Multiple Authors: Currently, our study is based on the assumption that source code can only be written by a single author. In the future, we plan to identify the group of authors involved in writing source code.

Code Reuse: Programmer often re-use already written code. Moreover, numerous tools are available online for automatically generating code that the authors otherwise have to write themselves. In the future, we plan to explore how code reuse impacts the accuracy of our approach.

AVAILABILITY

We have released all the artifacts necessary to reproduce all the results presented in our study. Scripts can be accessed using GitHub link.[8]

IX. CONCLUSION

Identifying the author of source code is beneficial for multiple applications such as plagiarism detection, software forensic, copyright, and violation detection. Programmers often use similar patterns to write code, such as variable naming, use of *for*/*while* loop. These patterns play an important role in identifying authors from the source code. We present an efficient approach to learn novel deep representations of source code files that characterize the code in a fixed size embedding vector. Code files written by the same author are close in the embedding space compared to code files written by a different author. In this study, we provide a CNN based author attribution system. We first train our convolutional neural network using the lifted structured loss function on character-level source code representations. We then extract the deep representation vectors from the CNN and feed them to a K-nearest neighbor classifier.

We evaluate our approach using the GCJ dataset. We performed our analysis of the three most popular programming languages: Python, C++, Java. Results show that our approach is efficient and scalable while maintaining a high accuracy under several different settings and scenarios. We evaluated our approach under obfuscated source-code, original source-code, and simulated real-world settings. We also conducted an initial experiment to address the problem's open-world aspect, where the author could be someone outside the training set. With nine source code files per author, our approach can identify 20, 458 programmers who write the code in multiple languages with an accuracy of 84.94%. Moreover, our approach can identify 10, 280 C++ programmers with an accuracy of 90.98%, 6, 910 Python programmers with an accuracy of 84.79% and 3, 911 Java programmers with an accuracy of 79.36%.

[8]https://github.com/sarim-zafar/LANG_OBF_OBLIVIOUS

REFERENCES

[1] S. Burrows, S. M. M. Tahaghoghi, and J. Zobel, "Efficient plagiarism detection for large code repositories," *Software: Pract. Exper.*, vol. 37, no. 2, pp. 151–175, 2007.

[2] G. Frantzeskou, E. Stamatatos, S. Gritzalis, C. E. Chaski, and B. S. Howald, "Identifying authorship by byte-level N-Grams: The source code author profile (SCAP) method," *Int. J. Digit. Evidence*, vol. 6, no. 1, p. 1–18, 2007.

[3] M. Abuhamad, T. AbuHmed, A. Mohaisen, and D. Nyang, "Large-scale and language-oblivious code authorship identification," in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, Jan. 2018, pp. 101–114.

[4] E. Dauber, A. Caliskan, R. Harang, G. Shearer, M. Weisman, F. Nelson, and R. Greenstadt, "Git blame who?: stylistic authorship attribution of small, incomplete source code fragments," *Proc. Privacy Enhancing Technol.*, vol. 2019, no. 3, pp. 389–408, Jul. 2019.

[5] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, and R. Greenstadt, "De-anonymizing programmers via code stylometry," in *Proc. 24th USENIX Secur. Symp. (USENIX Secur.)*, 2015, pp. 255–270.

[6] F. Ullah, J. Wang, S. Jabbar, F. Al-Turjman, and M. Alazab, "Source code authorship attribution using hybrid approach of program dependence graph and deep learning model," *IEEE Access*, vol. 7, pp. 141987–141999, 2019.

[7] M. Abuhamad, J.-S. Rhim, T. AbuHmed, S. Ullah, S. Kang, and D. Nyang, "Code authorship identification using convolutional neural networks," *Future Gener. Comput. Syst.*, vol. 95, pp. 104–115, Jun. 2019.

[8] H. O. Song, Y. Xiang, S. Jegelka, and S. Savarese, "Deep metric learning via lifted structured feature embedding," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2016, pp. 4004–4012.

[9] A. Aiken. (2004). *Moss : A system for Detecting Software Plagiarism*. [Online]. Available: http://www.cs.berkeley.edu/aiken/moss.html, and [Online]. Available: https://ci.nii.ac.jp/naid/10017142577/en/

[10] X. Li and X. J. Zhong, "The source code plagiarism detection using AST," in *Proc. Int. Symp. Intell. Inf. Process. Trusted Comput.*, Oct. 2010, pp. 406–408.

[11] G. Whale, "Identification of program similarity in large populations," *Comput. J.*, vol. 33, no. 2, pp. 140–146, Feb. 1990.

[12] S. Burrows, A. L. Uitdenbogerd, and A. Turpin, "Comparing techniques for authorship attribution of source code," *Softw., Pract. Exper.*, vol. 44, no. 1, pp. 1–32, Jan. 2014.

[13] A. Gray, P. Sallis, and S. MacDonell, "Software forensics: Extending authorship analysis techniques to computer programs," Univ. Otago, Dunedin, New Zealand, Inf. Sci. Discuss. Papers Series 97/14, 1997. [Online]. Available: http://hdl.handle.net/10523/872

[14] B. N. Pellin, "Using classification techniques to determine source code authorship," Dept. Comput. Sci., Univ. Wisconsin-Madison, Madison, WI, USA, White Paper, 2000.
[15] S. G. Macdonell, A. R. Gray, G. MacLennan, and P. J. Sallis, "Software forensics for discriminating between program authors using case-based reasoning, feedforward neural networks and multiple discriminant analysis," in *Proc. 6th Int. Conf. Neural Inf. Process.*, vol. 1, Nov. 1999, pp. 66–71.
[16] G. Frantzeskou, E. Stamatatos, S. Gritzalis, and S. Katsikas, "Effective identification of source code authors using byte-level information," in *Proc. 28th Int. Conf. Softw. Eng. (ICSE)*, 2006, pp. 893–896.
[17] S. Burrows, A. L. Uitdenbogerd, and A. Turpin, "Application of information retrieval techniques for source code authorship attribution," in *Proc. Int. Conf. Database Syst. Adv. Appl.* Berlin, Germany: Springer, 2009.
[18] B. S. Elenbogen and N. Seliya, "Detecting outsourced student programming assignments," *J. Comput. Sci. Colleges*, vol. 23, no. 3, pp. 50–57, 2008.
[19] R. C. Lange and S. Mancoridis, "Using code metric histograms and genetic algorithms to perform author identification for software forensics," in *Proc. 9th Annu. Conf. Genetic Evol. Comput. (GECCO)*, 2007, pp. 2082–2089.
[20] I. Krsul and E. H. Spafford, "Authorship analysis: Identifying the author of a program," *Comput. Secur.*, vol. 16, no. 3, pp. 233–257, Jan. 1997.
[21] X. Yang, G. Xu, Q. Li, Y. Guo, and M. Zhang, "Authorship attribution of source code by using back propagation neural network based on particle swarm optimization," *PLoS ONE*, vol. 12, no. 11, Nov. 2017, Art. no. e0187204.
[22] H. Ding and M. H. Samadzadeh, "Extraction of java program fingerprints for software authorship identification," *J. Syst. Softw.*, vol. 72, no. 1, pp. 49–57, Jun. 2004.
[23] X. Meng, B. Miller, and K.-S. Jun, "Identifying multiple authors in a binary program," in *Proc. Eur. Symp. Res. Comput. Secur.*, Aug. 2017, pp. 286–304, doi: 10.1007/978-3-319-66399-9_16.
[24] S. Burrows and S. M. Tahaghoghi, "Source code authorship attribution using n-grams," in *Proc. 12th Australas. Document Comput. Symp.* Melbourne, VIC, Australia: RMIT Univ., 2007, pp. 32–39.
[25] A. Caliskan, F. Yamaguchi, E. Dauber, R. Harang, K. Rieck, R. Greenstadt, and A. Narayanan, "When coding style survives compilation: De-anonymizing programmers from executable binaries," 2015, *arXiv:1512.08546*. [Online]. Available: http://arxiv.org/abs/1512.08546
[26] R. Hadsell, S. Chopra, and Y. LeCun, "Dimensionality reduction by learning an invariant mapping," in *Proc. IEEE Comput. Soc. Conf. Comput. Vis. Pattern Recognit.*, vol. 2, Jun. 2006, pp. 1735–1742.
[27] F. Schroff, D. Kalenichenko, and J. Philbin, "FaceNet: A unified embedding for face recognition and clustering," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2015, pp. 815–823.
[28] W. Chen, X. Chen, J. Zhang, and K. Huang, "Beyond triplet loss: A deep quadruplet network for person re-identification," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 403–412, doi: 10.1109/CVPR.2017.145.
[29] J. Butepage, M. J. Black, D. Kragic, and H. Kjellstrom, "Deep representation learning for human motion prediction and classification," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jul. 2017, pp. 6158–6166.
[30] H. Yao, S. Zhang, R. Hong, Y. Zhang, C. Xu, and Q. Tian, "Deep representation learning with part loss for person re-identification," *IEEE Trans. Image Process.*, vol. 28, no. 6, pp. 2860–2871, Jun. 2019.
[31] S. Ruder, P. Ghaffari, and J. G. Breslin, "Character-level and multi-channel convolutional neural networks for large-scale authorship attribution," 2016, *arXiv:1609.06686*. [Online]. Available: http://arxiv.org/abs/1609.06686
[32] N. Widiastuti, "Convolution neural network for text mining and natural language processing," in *Proc. IOP Conf., Mater. Sci. Eng.*, 2019, vol. 662, no. 5, Art. no. 052010.
[33] J. Tompson, R. Goroshin, A. Jain, Y. LeCun, and C. Bregler, "Efficient object localization using convolutional networks," in *Proc. IEEE Conf. Comput. Vis. Pattern Recognit. (CVPR)*, Jun. 2015, pp. 648–656.
[34] K. He, X. Zhang, S. Ren, and J. Sun, "Delving deep into rectifiers: Surpassing human-level performance on ImageNet classification," in *Proc. IEEE Int. Conf. Comput. Vis. (ICCV)*, Dec. 2015, pp. 1026–1034.
[35] V. Nair and G. E. Hinton, "Rectified linear units improve restricted Boltzmann machines," in *Proc. 27th Int. Conf. Mach. Learn. (ICML)*, 2010, pp. 807–814.
[36] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa, "Natural language processing (almost) from scratch," *J. Mach. Learn. Res.*, vol. 12 pp. 2493–2537, Aug. 2011.
[37] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proc. 13th Int. Conf. Artif. Intell. Statist.*, 2010, pp. 249–256.
[38] L. Liu, H. Jiang, P. He, W. Chen, X. Liu, J. Gao, and J. Han, "On the variance of the adaptive learning rate and beyond," 2019, *arXiv:1908.03265*. [Online]. Available: http://arxiv.org/abs/1908.03265
[39] M. Zhang, J. Lucas, J. Ba, and G. E. Hinton, "Lookahead optimizer: k steps forward, 1 step back," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019, pp. 9593–9604.
[40] C. C. Aggarwal, A. Hinneburg, and D. A. Keim, "On the surprising behavior of distance metrics in high dimensional space," in *Proc. Int. Conf. Database Theory*. Berlin, Germany: Springer, Jan. 2001, pp. 420–434.
[41] T. Liu, A. W. Moore, and A. Gray, "New algorithms for efficient high-dimensional nonparametric classification," *J. Mach. Learn. Res.*, vol. 7, pp. 1135–1158, Jun. 2006.
[42] D. A. Ross, J. Lim, R.-S. Lin, and M.-H. Yang, "Incremental learning for robust visual tracking," *Int. J. Comput. Vis.*, vol. 77, nos. 1–3, pp. 125–141, May 2008.

SARIM ZAFAR received the B.S. degree in computer science from Information Technology University, in 2018. He is currently a Graduate Student with North Dakota State University (NDSU), Fargo, ND, USA. He is also working as a Graduate Research Assistant (GRA), NDSU. His research interests include social media analytics and applied machine learning in software engineering.

MUHAMMAD USMAN SARWAR received the B.S. degree in computer science from Information Technology University, in 2017. He is currently a Graduate Student and a Graduate Research Assistant with North Dakota State University (NDSU), Fargo, ND, USA. His current research interests include social media analytics and applied machine learning in software engineering.

SAEED SALEM (Member, IEEE) received the Ph.D. degree in computer science from the Rensselaer Polytechnic Institute, Troy, NY, USA. He is currently an Associate Professor with North Dakota State University. His research interests include graph mining and machine learning with a focus on developing algorithms for mining frequent and significant graphs. His group developed enumeration algorithms for mining all frequent subgraphs, cross-graph dense graphs, and approximate frequent subgraphs from heterogeneous graphs.

MUHAMMAD ZUBAIR MALIK received the Ph.D. degree in software engineering from The University of Texas at Austin. He was a Postdoctoral Researcher and a Research Scientist with Carnegie Mellon University, Pennsylvania. He is currently an Assistant Professor with North Dakota State University. His research interest includes software systems. His recent research interests include automated program repair, developing language, and systems for robotics and machine learning systems.