ON THE APPLICABILITY OF DEEP METRIC LEARNING TO ADDRESS SOURCE CODE AUTHORSHIP ATTRIBUTION PROBLEM UNDER SIMULATED REAL-WORLD CONSTRAINTS.

A Thesis
Submitted to the Graduate Faculty
of the
North Dakota State University
of Agriculture and Applied Science

By

Sarim Zafar

In Partial Fulfillment of the Requirements
for the Degree of
MASTER OF SCIENCE

Major Department:
Computer Science

October 2020

Fargo, North Dakota

NORTH DAKOTA STATE UNIVERSITY

Graduate School

Title

ON THE APPLICABILITY OF DEEP METRIC LEARNING TO ADDRESS SOURCE CODE AUTHORSHIP ATTRIBUTION PROBLEM UNDER SIMULATED REAL-WORLD CONSTRAINTS.

By

Sarim Zafar

The supervisory committee certifies that this thesis complies with North Dakota State University's regulations and meets the accepted standards for the degree of

MASTER OF SCIENCE

SUPERVISORY COMMITTEE:

Dr. Muhammad Zubair Malik
Chair

Dr. Saeed Salem

Dr. María de los Ángeles Alfonseca-Cubero

Approved:

16 November 2020
Date

Prof. Simone Ludwig
Department Chair

ABSTRACT

Source code authorship attribution is a widely studied research topic in the information security domain. In this dissertation, we develop and evaluate models that enable us to solve source code authorship attribution using deep metric learning. In particular, first, we simulate a real-world setting. Second, we use a number of loss functions from the deep metric learning domain to train neural network models. Thirdly, we evaluate these different models' performance on a benchmark and determine whether there is a quantifiable performance difference between these deep metric loss functions. Lastly, we demonstrate how we can extend our proposed methodology address the open world scenario. We argue that these models, and the techniques they take advantage of, are a stepping stone towards achieving real-world source code authorship attribution that can work across multiple programming languages and even under large scale obfuscated settings.

ACKNOWLEDGEMENTS

There are many people I must thank for contributing to the two beautiful years of my experience as a Masters's student at NDSU.

First and foremost, I would like to thank Dr. Muhammad Zubair Malik wholeheartedly. He has supported me through thick and thin. His sage-like wisdom kept me on track and focused. I genuinely believe that he has made a better researcher. He helped me on all sorts of fronts, from improving my writing to enhancing my understanding of the different nuances of conducting experiments.

I have also been very fortunate to meet and learn from extremely talented and driven people here at NDSU. They have always been very patient with me and never flinched from taking out time to help me. Dr. Saeed Salem has been one of those people. Even though I was his student, he never made me feel that way and always treated me as a peer. Dr. Gursimran Singh Walia has also been very kind and considerate to me over the past two years. Our "quick" hallway conversations have always concluded with a powerful lesson. Your constant uplifting statements got me through the gloomiest of days. I want to especially acknowledge Dr. Brian Slator for the fun environment he provided in his class. He made our lives richer with his vigor and batman apparel; I feel poorer after his untimely passing.

I want to thank Muhammad Usman Sarwar for his constant support through the highs and the lows of my time here. I will always look up to you for your uncanny ability to play around with words like a magician and the stable positive aura you always emanate. Thank you for going over my drafts at a moment's notice and providing useful and tangible feedback. Apart from the people at the university, I would also like to extend my warmest thanks to those whom I've interacted with on a day to day basis. From the wonderful neighbors to the heartwarming interactions at the local grocery store around the block; You all made me feel welcomed.

Lastly, I would like to thank in no particular order Aizaz Qaisarani, Ali Dehalvi, Usman Zafar, Taimur Hassan, Abdullah Moazzam, and Tazeen Shaukat who all kept me sane during the early Covid-19 days and patiently listened to all my ramblings and gave me carefully curated advice.

DEDICATION

This thesis is dedicated to my father(بابا). With this, I am now one step closer to your dream for me. I want to dedicate the following few verses, written in Urdu, taken from a poem by Shahzad Ahmad to him.

رخصت ہوا تو آنکھ ملا کر نہیں گیا
وہ کیوں گیا ہے یہ بھی بتا کر نہیں گیا

یوں لگ رہا ہے جیسے ابھی لوٹ آئے گا
جاتے ہوئے چراغ بجھا کر نہیں گیا

گھر میں ہے آج تک وہی خوشبو بسی ہوئی
لگتا ہے یوں کہ جیسے وہ آکر نہیں گیا

شہزادؔ یہ گلہ ہی رہا اس کی ذات سے
جاتے ہوئے وہ کوئی گلہ کر نہیں گیا

TABLE OF CONTENTS

ABSTRACT iii

ACKNOWLEDGEMENTS iv

DEDICATION v

LIST OF TABLES ix

LIST OF FIGURES x

LIST OF APPENDIX TABLES xi

1. INTRODUCTION 1

1.1. Overview 1

1.2. Contributions and outline 3

1.2.1. Large scale authorship attribution 3

1.2.2. Multi-lingual scenario 3

1.2.3. Obfuscated constraint 4

1.2.4. Simulated real-world constraint 4

1.2.5. Close world assumption and open world assumption 4

2. THEORETICAL BACKGROUND FOR DEEP METRIC LEARNING 6

2.1. Deep metric learning 6

2.2. Contrastive loss 7

2.3. Triplet loss 7

2.3.1. Miners for finding interesting triplets 8

2.3.2. Semi-hard triplet loss 9

2.3.3. Soft margin variant of triplet loss 9

2.4. Lifted structure loss 9

3. SOURCE CODE AUTHORSHIP ATTRIBUTION [1] 11

3.1. Related work 11

3.2. Authorship attribution system 15

3.3. Source code representation . . . 15
3.4. Dataset collection . . . 16
3.5. Dataset preparation . . . 16
3.6. Obfuscation tools . . . 17
3.7. Dataset preprocessing . . . 18
3.8. Dataset analysis . . . 18
3.9. Problem framing . . . 19
3.10. Loss functions . . . 22
3.11. Train, validation, and test split for NNs . . . 22
3.12. Model architecture . . . 22
3.13. Optimizer . . . 23
3.14. Batching methodology . . . 24
3.15. Training parameters . . . 24
3.16. Classification using embeddings . . . 25
3.17. Attribution evaluation . . . 25
4. EXPERIMENT RESULTS . . . 26
4.1. Embedding evaluation . . . 26
4.2. Closed world source code authorship attribution . . . 31
4.2.1. Performance on original source code only dataset . . . 31
4.2.2. Performance on obfuscated source code dataset . . . 40
4.2.3. Performance on simulated real-world dataset . . . 53
4.2.4. Discussion . . . 64
4.3. Open world setting . . . 65
4.4. Are the different models language oblivious? . . . 68
5. FUTURE WORK . . . 70
6. CONCLUSION . . . 71

REFERENCES 72

APPENDIX. SUPPLEMENTARY TABLES 78

LIST OF TABLES

Table Page

3.1. Results of past source coode authorship attribution systems. 15

4.1. LS: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages using the original GCJ dataset. . . . 39

4.2. TSH: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages using the original GCJ dataset. . . . 41

4.3. TH: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages using the original GCJ dataset. . . . 41

4.4. TH-soft: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages using the original GCJ dataset. 42

4.5. LS: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages under obfuscated setting. 53

4.6. TSH: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages under obfuscated setting. 55

4.7. TH: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages under obfuscated setting. 55

4.8. TH-soft: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages under obfuscated setting. 56

4.9. LS: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages under simulated real-world constraint. 63

4.10. TSH: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages under simulated real-world constraint. 65

4.11. TH: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages under simulated real-world constraint. 65

4.12. TH-soft: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages under simulated real-world constraint. 66

4.13. Accuracy of authorship attribution across different number of source code files per author, different unseen programming languages mined from the original GCJ dataset. . . 68

LIST OF FIGURES

Figure Page

3.1. 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. 19

3.2. 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. 20

3.3. 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. 21

3.4. CNN model architecture 23

4.1. Lifted struct: Distribution of inter and intra-cluster distances in training (2018), validation set (2008), and one of the test years (2020). Here, the red distribution shows the inter cluster distances while dotted blue distribution shows the intra-cluster distances. . 26

4.2. Triplet semi-hard: Distribution of inter and intra-cluster distances in training (2018), validation set (2008), and one of the test years (2020). Here, the red distribution shows the inter cluster distances while dotted blue distribution shows the intra-cluster distances. 27

4.3. Triplet hard: Distribution of inter and intra-cluster distances in training (2018), validation set (2008), and one of the test years (2020). Here, the red distribution shows the inter cluster distances while dotted blue distribution shows the intra-cluster distances. . 29

4.4. Triplet hard soft: Distribution of inter and intra-cluster distances in training (2018), validation set (2008), and one of the test years (2020). Here, the red distribution shows the inter cluster distances while dotted blue distribution shows the intra-cluster distances. 30

4.5. Loss function performance comparison using the original GCJ dataset 40

4.6. Distributions of accuracy loss when moving from original to obfuscated setting. Here the y-axis shows the density while the x-axis shows the accuracy difference. The box plot on top of the distributions shows an alternative visualization of the same distribution and shares the x-axis with the same distribution. 52

4.7. Loss function performance comparison on obfuscated source code 54

4.8. Loss function performance comparison on simulated real world dataset. 64

4.9. Distributions of accuracy difference across all settings between loss functions. Here the y-axis shows the density while the x-axis shows the accuracy difference. 66

4.10. Open world setting 67

LIST OF APPENDIX TABLES

Table Page

A.1. LS: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages using the original GCJ dataset. . . . 78

A.2. TSH: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages evaluated on obfuscated dataset. . . 79

A.3. TH: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages using the original GCJ dataset. . . . 80

A.4. TH-Soft: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages under non-obfuscated settings. . 81

A.5. LS: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages using obfuscated dataset. 82

A.6. TSH: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages under obfuscated setting. 83

A.7. TH: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages under obfuscated setting. 84

A.8. TH-soft: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages under obfuscated setting. 85

A.9. LS: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages under simulated real-world constraint. 86

A.10. TSH: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages under simulated real-world constraint. 87

A.11. TH: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages under simulated real-world constraint. 88

A.12. TH-soft: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages under simulated real-world constraint. 89

1. INTRODUCTION

1.1. Overview

Source code often contains distinctive patterns that represent a programmer's 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].

In the last decade, artificial intelligence has made commendable strides primarily due to access to large volumes of data and astronomical increase in the processing power of computers. The AI agents have started making a noticeable impact on our everyday lives ranging from artificial assistants such as Apple's Siri to Tesla's self-driving cars. But the Achilles heel of many of these agents is the number of training samples required to perform well in a real-world scenario. For example, Tesla cars have driven around 3 billion miles across the world, but their driving agent has still not achieved completely autonomous self-driving capability. We want our AI agents to learn from a small number of training samples just like an average human would recognize a dog or a cat without seeing tens of thousands of examples of each. The few-shot learning domain came into existence to deal with learning new concepts from a limited training dataset. The progress in this domain has been extremely promising in the last few years. For example, they can now perform person/vehicle re-identification using only a few examples for each instance with a very high success rate [4, 5, 6].

We propose that we can similarly address source code authorship attribution as we address the vehicle re-identification problem. Instead of images of cars from different viewpoints under different lighting conditions, we observe source codes authored by programmers to solve various tasks written across multiple programming languages. Source code authorship attribution is usually solved by capturing distinctive patterns of style in source code and matching them with a programmer's programming style. For a computer, characterizing such patterns of style to capture an author's style accurately is challenging. You have to remember that the source code for a computer is an array of numbers. It must understand what a particular combination of numbers means and the relationship between them to perform any meaningful analysis.

Source code authorship attribution becomes especially hard to address when:

- The number of known source code files associated with each author is limited.
- The attribution has to be done among thousands of authors.
- The source codes are authored in multiple programming languages.
- The source codes are obfuscated.

In this work, we are specifically interested in solving this problem while addressing the aforementioned points. We do so by taking advantage of the work done in the deep metric learning domain, which is sometimes used to solve the few-shot learning problems similar to what we are trying to address. Deep metric learning algorithms enable a neural network to learn a mapping such that similar inputs are mapped closer to each other in the embedding space, and dissimilar samples are easily differentiated by being mapped farther apart in the embedding space. We aim to evaluate such algorithms' applicability and determine how well they can perform in solving the source code authorship attribution problem under various constraints. To do so, we choose a variety of deep metric learning loss functions and train neural network models using those. We then compare each technique's performance on a large scale diverse dataset consisting of source codes written in different programming languages. To make the problem even harder to solve, we further obfuscate the source codes using off the shelf obfuscators. We want to demonstrate that the techniques can map source code files originating from the same author closer to each other in the embedding space even when they are obfuscated.

Despite the difficulty of the task, commendable progress has been made. Researchers in the past have tried to solve this problem using both handcrafted features [7] and by using term-frequency based features [3]. But the experiments have either been limited to a single programming language [8, 9] or a small number of authors [10]. The past papers' proposed techniques are rarely evaluated on obfuscated source code [11], especially on the same scale as the regular source code.

1.2. Contributions and outline

This dissertation is primarily based on and is an extension of our work published in IEEE Access in Oct. 2020 [12]. This dissertation shows that we can solve the source code authorship attribution problem by treating it as an information retrieval task and by leveraging deep metric learning based loss functions. Our models ingest a source code and output a vector that can then be used to look up other similar vectors in a priors database to identify the author of the source code. The latter part enables us to bring down the system's overall complexity by a large margin compared to the existing state of the art. Our goal is to determine whether we can learn such a transformation function to map diverse source code files and use these embeddings to perform authorship attribution in a real-world like setting. We further want to determine whether there is a performance difference between different loss functions if trained using a similar configuration.

1.2.1. Large scale authorship attribution

Typically authorship attribution problem is evaluated on the scale of a few thousand authors at max. In this work, we consider a much larger pool of authors and demonstrate that our proposed methodologies can work well beyond the traditional upper limit and can even operate when the number of authors is tens of thousands. We do this by removing the static classifier component and replacing it with a lazy learner. We use the Nearest Neighbor Classifier (NNC) to store source code vectors and their respective authors. When our system is queried to determine the author of a source code file, we map it to a vector using our deep learning model and then ask the NNC to decide who could be the potential author. The NNC part is critical. It allows us to add more authors without re-training any classifier from scratch. Because it is an information retrieval problem, the complexity is drastically lower than traditional classifiers.

1.2.2. Multi-lingual scenario

The source code authorship attribution problem is traditionally evaluated, assuming that all the source code files are written using a single programming language. However, recent literature

has looked into assessing the problem under a multi-lingual scenario but is currently limited to only two languages together. We take this a step further and consider three programming languages. For example, under this scenario, it could be that given two source code files that are written in **C++** and **Python** authored by the same person, We want to determine whether a third source code file, written in **Java**, is also authored by that same person or not.

1.2.3. Obfuscated constraint

One of the critical experiments missing in the existing literature is evaluating the approach using obfuscated source code files, especially on the same scale as the regular source code files. In the case of Abuhamad et al. [3], they only evaluated on C++ obfuscated source codes with only up to 120 authors.

In this work, we not only evaluate our approach on obfuscated source codes for each of the primary languages separately, but we also assess it in a multi-lingual scenario. For example, under this specific constraint and scenario, given two obfuscated source code files that are written in **C++** and **Python** authored by the same person, We want to determine whether a third obfuscated source code file, written in **Java**, is also written by that same person or not.

1.2.4. Simulated real-world constraint

We make the evaluation more rigorous by further increasing the difficulty of the problem by trying to solve the problem in a multi-lingual setting where both Obfuscated & the original version of the source codes are included. To the best of our knowledge, this is the first study that evaluates a proposed approach for solving the source code authorship attribution problem under such a rigorous constraint. This enables us to claim more confidently that our work can be used to solve this problem in the real world where the source codes associated with an author could be written in multiple programming languages, and some of them could be obfuscated as well.

1.2.5. Close world assumption and open world assumption

The source code authorship attribution problem could either be solved with a closed-world assumption or an open-world one. Under a closed-world assumption, given a source code, you would need to identify the most likely author from a set of authors. In an open world scenario, the system should also be able to say whether the given source code was authored by someone who is not present in the set of authors. In this work, we primarily work under the closed-world assumption, but we also demonstrate an extension that can work in the open-world setting as

well by utilizing distance-based thresholding. This is possible primarily because we calculate the distance between different source codes to determine their authors, and we can say that if a source code vector is farther than a specific distance from all the prior source code vectors, then the source code is probably authored by someone that our system does not know about.

In **Chapter 2**, we briefly discuss the relevant mathematical background for deep metric learning and related concepts. We primarily discuss three different deep metric learning loss functions and a few different mining methods.

In **Chapter 3**, we formally define the source code authorship attribution problem. Furthermore, source code collection and pre-processing steps are described. We briefly discuss the obfuscation tools used to obfuscate the source codes. We argue the need for simulated real-world constraint and discuss how we simulated one. We design our deep learning models for learning meaningful embeddings. We discuss the training strategies across different loss functions.

In **Chapter 4**, we evaluate our models under a number of different scenarios described in the earlier chapter. We also evaluate and discuss the goodness of our embeddings, both qualitatively and quantitatively. We also compare our performance and results with existing literature.

Finally, in **Chapter 5**, we identify the remaining challenges and discuss the path moving forward.

2. THEORETICAL BACKGROUND FOR DEEP METRIC LEARNING

This chapter provides the necessary technical background on deep metric learning. For a more thorough and slower-paced introduction to underlying concepts such as Neural Networks and how they work, we recommend the Deep Learning book from Goodfellow et al. [13]. For those who prefer video content, the Deep Learning course taught by Andrew Ng at Stanford is a good point to get started as well.

2.1. Deep metric learning

We, as humans, are good at determining the similarity between different objects. Most of the data that we work with, such as images, are usually represented by numbers, so one can argue that we can use some distance metric to determine the similarity between two images. However, there are infinite numbers of distance metrics that one can use. At its core, the idea of deep metric learning is to learn an embedding function such that the samples that we consider to be similar should be closer to each other while the samples that we consider to be dissimilar should be further away according to a selected distance metric. This approach has a number of advantages over traditional classification based approaches. In traditional approaches, you would get a probability distribution over a set of known classes. In contrast, a deep metric learning based approach allows us to reduce the problem to nearest neighbor lookup, which further allows us to solve open set classification problems. There are essentially three components of deep metric learning, which are informative input samples, the encoder or the deep learning model, and the loss function. We will talk about all three in this work wherever it is required.

Assume there's a dataset $\mathcal{X} = \{x_1, \ldots, x_N\}$ with corresponding labels $\mathcal{C} = \{c_1, \ldots, c_N\}$. The idea is to minimize the distance $D_{a,b}$, where $D_{a,b} = D(f(x_a), f(x_b))$, such that $c_a = c_b$ and maximize the distance $D_{a,d}$ such that $c_a \neq c_d$. Where $f(.)$ is the deep neural network's embedding output. In a deep learning based setting in order to achieve this we try to minimize the following loss function:

$$\min \ell(d, ((x_1, c_1), \ldots, (x_m, c_m))). \tag{2.1}$$

, where d is a distance metric and m, is the batch size.

In this work, the loss functions that we will be discussing use the $Squared - L2$ distance to determine the distance between the embeddings. There are other recent works in the deep metric learning domain that use cosine similarity as well [14, 15, 16].

2.2. Contrastive loss

Contrastive loss [17] is computed using three inputs $\mathbf{x}_i, \mathbf{x}_j, y_{ij}$. The idea is to minimize the distance between a pair of examples with the same class label. We penalize a pair of examples with different class labels if their distance is smaller than that of the margin parameter α. Following is the formulation of the contrastive loss function:

$$J = \frac{1}{m} \sum_{(i,j)}^{m/2} y_{i,j} D_{i,j}^2 + (1 - y_{i,j}) \left[\alpha - D_{i,j}\right]_+^2 \tag{2.2}$$

where m is the number of sample in a batch, $D_{i,j} = ||f(\mathbf{x}_i) - f(\mathbf{x}_j)||_2$, and the label $y_{i,j} \in \{0, 1\}$ indicates whether $\mathbf{x}_i, \mathbf{x}_j$ is from the same class or not. The α is the margin parameter. The $[\cdot]_+$ operation indicates the hinge function $\max(0, \cdot)$.

2.3. Triplet loss

Triplet loss [18] computed using three values $\mathbf{x}_a^{(i)}, \mathbf{x}_p^{(i)}, \mathbf{x}_n^{(i)}$ where $\mathbf{x}_a^{(i)}, \mathbf{x}_p^{(i)}$ have the same class labels and $\mathbf{x}_a^{(i)}, \mathbf{x}_n^{(i)}$ have different class labels. The $\mathbf{x}_a^{(i)}$ term is referred to as an *Anchor* of a triplet. The $\mathbf{x}_p^{(i)}$ term is referred to as an *Positive* of a triplet and $\mathbf{x}_n^{(i)}$ term is referred to as an *Negative* of a triplet. The idea is to help the network learn such a mapping from input to output domain such that the distance between $\mathbf{x}_a^{(i)}$ and $\mathbf{x}_n^{(i)}$ is larger than the distance between $\mathbf{x}_a^{(i)}$ and $\mathbf{x}_p^{(i)}$ plus the margin parameter α. The cost function is defined as,

$$J = \frac{3}{2m} \sum_{i}^{m/3} \left[D_{ia,ip}^2 - D_{ia,in}^2 + \alpha\right]_+ \tag{2.3}$$

where $D_{ia,ip} = ||f(\mathbf{x}_i^a) - f(\mathbf{x}_i^p)||$ and $D_{ia,in} = ||f(\mathbf{x}_i^a) - f(\mathbf{x}_i^n)||$.

2.3.1. Miners for finding interesting triplets

The idea of miners is motivated by the importance of good informative samples. Up until now, we focused on the loss function and did not really discuss how we should sample the triplets from the dataset. As one can imagine, random sampling might give you non-optimal samples to work with. As the potential number of triplets scales cubically in training set size, you will encounter more and more non-optimal triplets if the sampling strategy is naive. During the training process, more and more of those non-optimal triplets are correctly ordered and effectively provide no training signal [18], thus impairing the remaining training process. The idea is to get informative triplets from the dataset so that we can learn 'meaningful' representation and allow the network to learn faster. There are two ways to do this, either we mine for samples in an 'offline' strategy, or we mine for them in an 'online' strategy. In an offline mining strategy, you would identify meaningful samples before each epoch starts. So in case of triplet loss, you would mine for 'meaningful' triplets by going through the training dataset and feed in the curated triplets to train the network and repeat the process after a certain time because the embeddings have been updated. Overall this way of mining is considered to be not very efficient since we need to do a full pass on the training set to generate the set of 'meaningful' triplets. In an online mining strategy, you would form 'meaningful' triplets using samples from within the batch. This is done in two parts; first, the samples in a batch are not fed as triplets, but independent samples, just like for a normal classification problem, and secondly, by modifying the vanilla loss function and including the mining process before the actual loss is computed. As you can imagine, both approaches have their pros and cons. Researchers often resort to online mining because of its cheap compute cost and decent results on a variety of datasets.

In literature, there are primarily two kinds of 'interesting' triplets, Hard triplets and Semi-Hard triplets.

2.3.1.1. Hard triplet loss

Hard triplet [19] loss is a variant of triplet loss that dictates that when forming a triplet it should hold the following property:

$$D_{ia,ip} > D_{ia,in} \tag{2.4}$$

The motivation behind this triplet formation strategy is that in vanilla triplet loss, most of the trivial examples are learned to map easily by the mapping function quickly hence making them uninformative for the rest of the training process. Thus mining hard triplets become a crucial step for learning throughout the training process. Intuitively, reading very similar source code originating from different authors (hard negatives) or source codes that are wildly different but are authored by the same author (hard positives) dramatically helps the network to solve the problem.

2.3.2. Semi-hard triplet loss

Semi-Hard triplet [18] loss is a variant of triplet loss that dictates that when forming a triplet it should hold the following property:

$$D_{ia,ip} + \alpha > D_{ia,in} > D_{ia,ip} \tag{2.5}$$

The motivation behind this triplet formation strategy is that hard triplet loss would select outliers in the data disproportionately and will not allow the neural network to learn 'normal' associations. This is why we choose to form triplets with this new constraint.

2.3.3. Soft margin variant of triplet loss

The role of the hinge function $[\cdot]_+$ in triplet loss is to avoid correcting 'already correct' triplets. But for some tasks, it can be beneficial to pull together samples from the same class as much as possible [20, 21]. For this purpose, it is possible to replace the hinge function with a smooth approximation using the softplus function. The softplus function has similar behavior to the hinge, but it decays exponentially instead of having a hard cut-off.

2.4. Lifted structure loss

Lifted Structure loss was presented by Song et al. [22]. The idea is to extract as much information from each batch as you can. This is done by considering all possible triplets one can

form given an anchor and a positive sample in the batch. So the loss function will force to push away all the possible negatives while bringing the anchor and positive closer together.

$$\begin{aligned}\tilde{J}_{i,j} =& \log\left(\sum_{(i,k)\in\mathcal{N}} \exp\{\alpha - D_{i,k}\} + \sum_{(j,l)\in\mathcal{N}} \exp\{\alpha - D_{j,l}\}\right) + D_{i,j} \\ \tilde{J} =& \frac{1}{2|\mathcal{P}|} \sum_{(i,j)\in\mathcal{P}} \max\left(0,\ \tilde{J}_{i,j}\right)^2 \end{aligned} \tag{2.6}$$

Here, $\mathcal{P}$ denotes the set of positive pairs in the batch, $\mathcal{N}$ denotes the set of negative pairs in the batch, and $D_{i,j}$ is the distance between sample i and sample j. So the source code vectors belonging to one author are pushed together in the distance space, and the source code vectors belonging to different authors are pushed further apart from each other.

Usually, all of these loss functions are used to fine-tune existing pre-trained vision models for a specific use case, such as person re-identification.

3. SOURCE CODE AUTHORSHIP ATTRIBUTION [1]

This chapter discusses the related work and configuration of different experiments conducted to address the source code authorship attribution problem under various constraints and scenarios.

3.1. Related work

Numerous studies conducted in the past to explore source code authorship attribution problem. These studies use machine learning based approaches on different handcrafted features, such as lexical features, syntactical features, semantic features, textual features, and graph-based features. Table 3.1 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. They represent the source codes using TF-IDF and word embedding based vectors. These code representations are further fed into CNN to learn 'deep representations'. They use these deep representations to train a random forest classifier to extract source code authorship. They used source codes from Google Code Jam (GCJ) and Github to evaluate their approach. They reported accuracy of 99.4% for 150 programmers and 96.2% for 1600 programmers. They claim that their approach can scale to several hundred programmers and across various programming languages.

Abuhamad et al. [11] 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 RNN to generate the deep feature vectors. Further, the deep representations are fed into a random forest classifier to classify the source code authors. They used source codes from Google Code Jam (GCJ) and Github to evaluate their approach. 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. They claim that their approach is resilient to obfuscation and is scalable to multiple languages. However, they created their large-scale dataset by assuming that if the same username occurs across different years of the code jam dataset, they

[1] Some of the material in this chapter was taken from a journal paper that was co-authored by Sarim Zafar, Usman Sarwar, Saeed Salem, and M. Zubair Malik. Sarim Zafar had primary responsibility for devising all the experiments and conducting them. Sarim Zafar was the primary developer of the conclusions that are advanced here. Sarim Zafar also drafted and revised all versions of this chapter. M. Zubair Malik served as proofreader and checked the validity of the content and experiments conducted by Sarim Zafar.

must be the same author. They acknowledge that they induce both false positive and false negative errors to the experiment by making this assumption.

Ullah et al. [23] 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. [7] utilized machine learning models to de-anonymize source code authors. The extensive feature extraction process for programmer code stylometry involves code parsing. They derive the stylometry feature set of the coding style 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. [9] analyzed the source code files extracted from the open-source version control systems. They provided an extension of Caliskan-Islam et al. [7] work, which performs stylistic authorship identification on the source code sample with high accuracy. Following are some of the highlights of their work:

- They ensemble the output probabilities of sample source code files of the same author for the same classifier, which results in improved classification.
- They were able to link several samples to the same programmer.
- They used calibration-curves to prove the quality of authorship attribution for a given source code sample.

Yang et al. [24] proposed a novel approach to determine the authors of Java source code files. They used Neural Network (NN) based particle swarm optimization (PSO) and lexical, layout, structural, and syntactic features to perform the source code attribution. The source code is fed into the NN where the PSO backpropagation algorithm learns the network's weights. They used

3,022 java files authored by 40 authors, extracted from GitHub, To evaluate their approach. They were able to achieve 91.06% accuracy to identify 40 Java programmers.

Burrow et al. [1] 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, as reported in [25]. Results were reported on the tokenized representations of C language source code files for 100 authors.

Frantzeskou et al. [2] presented an approach called Source Code Author Profiles (SCAP). It utilizes the byte level n-grams and 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. [26], 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. [27] 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 works, different kinds of limitations persist throughout. First, most of the earlier studies' features limit the applicability to usually one language because of the feature engineering step's handcrafted nature. Features extracted for one particular language cannot be used to identify authors in other languages. However, the work by Abuhamad et al. [3, 11] addresses 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 [17], triplet loss [18], Quadruplet loss [28], and lifted structured loss [22]. These loss functions have been shown to perform well to solve few-shot problems. 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.

Many approaches proposed for solving few-shot problems, aimed at explicitly solving computer vision based problem of vehicle/person re-identification/verification task [18]. In a verification task, given two images, we want to determine whether they are the same person/vehicle. Naturally, re-identification is an extension of this task. Our work takes inspiration from the approaches proposed to solve these problems and argues that our task is analogous to the re-identification task. Rather than looking at head-shots, we look at source codes and want to determine whether the same person authored them. So, consequently, we decided to evaluate the applicability of deep metric learning based loss functions.

Our neural networks models take the source code as input and output a dense vector. We also applied specific constraints on the embedding vector, such as L2-normalization, as required by some specific loss functions. Some of the previous works [3, 11] obtain these embeddings from a NN's penultimate layer trained using the categorical cross-entropy loss function. We specifically use Convolutional Neural Network (CNN), based models. Researchers use these type of neural networks often to solve vision-based tasks, but they have also been shown to work well on NLP related tasks as well[29]. We use models based on this due to its ability to train faster as compared to LSTM and GRU, especially when you are considering very long sequences. This is due to the inherent time/space complexity advantage of CNNs over RNN based models.

Table 3.1. Results of past source coode authorship attribution systems.

Reference	No. of Authors	Languages	Accuracy (%)	Approach
Pellin [30]	2	Java	88.47%	SVM with tree kernel
MacDonell et al. [31]	7	C++	81.10%	Machine Learning (FNN). Statistical analysis (MDA)
MacDonell et al. [31]	2	C++	88.00%	Machine learning (case-based reasoning)
Frantzeskou et al. [2]	8	C++	100.00%	Rank similarity using KNN
Burrows et al. [32]	10	C	76.78%	Information retrieval using mean reciprocal ranking
Elenbogen & Seliya [33]	12	C++	74.70%	Statistical analysis using decision tree
Lange & Mancoridis [34]	20	Java	55.00%	Rank similarity measurements (nearest neighbor)
Krsul & Spafford [35]	29	C	73.00%	Statistical analysis (discriminant analysis)
Frantzeskou et al. [2]	30	C++	96.90%	Rank similarity measurements
Yang et al. [24]	40	Java	91.10 %	NN with particle swarm optimization
Ding & Samadzadeh [8]	46	Java	62.70%	Statistical analysis using canonical discriminant
Burrows et al. [25]	100	C, C++	79.90%	Support Vector Machines
Burrows et al. [25]	100	C, C++	80.37%	Random Forest
Caliskan-Islam et al. [7]	229	Python	53.91%	Random Forest
Meng et al. [27]	284	C, C++ Binaries	65.00%	Combined model with Conditional Random Field (CRF)
Farhan Ullah et al. [23]	1000	C#	98%	PDG with Neural Network
Farhan Ullah et al. [23]	1000	C++	100%	PDG with Neural Network
Farhan Ullah et al. [23]	1000	Java	98%	PDG with Neural Network
Caliskan-Islam et al. [7]	1,600	C++	92.83%	Random Forest
Abuhamad et al. [11]	1000	Java	95.8%	Stacked Convolutional Neural Network
Abuhamad et al. [11]	1500	Python	94.6%	Stacked Convolutional Neural Network
Abuhamad et al. [11]	1600	C++	96.2%	Stacked Convolutional Neural Network
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

3.2. Authorship attribution system

The authorship attribution system consists of two primary components. One is the deep neural network that encodes the source code files in fixed dimensional vectors, and second, a lookup system takes a set of source code vectors of 'known' authors and takes some query vectors of whom we want to know the authors. The lookup system then assigns each vector an author based on the priors. For the lookup system, we choose to use a simple Nearest Neighbor Classifier(NNC). Usage of NNC is an intuitive choice here as the same author's vectors are supposed to be close to each other in the embedding space as compared to vectors of codes authored by others. To determine the distance between two vectors, we choose to work with L1 distance. It has been shown to work very well in high dimensional space by Aggarwal et al. [36]. Our experiments suggest a modest improvement when we use L1 distance instead of L2.

3.3. Source code representation

The source codes cannot be fed into the neural network as is because it is essentially a string. There are many ways to compute a vector representation of a given text that can then be fed to a neural network. It can be represented as a fixed size term frequency vector. It can also

be represented as a sequence of vector tokens where each token represents a character or a word. In this work, we chose to represent the source code as a sequence of characters. Character-level representation allows us to maximize the number of styles our model can capture from the source codes. For example, in a term-based representation, if the tokenizer has not seen a particular term in the camel case but only in the upper case, it would be treated as a UNK (unknown) token. Whereas in a character-level encoding, we can still represent this new term as a sequence of previously known upper and lower case characters. We first convert each character to an integer, and then we map these series of integers to a sequence of vectors. This mapping can be done either by one-hot encoding the integers or by learning an embedding matrix that can map the said integers to a vector. This representation almost always will result in variable-sized sequences, and Tensorflow [37] requires the sequence sizes to remain constant within a batch. To address this, we can choose a cut-off point and pad each sequence of integers with zeros. We mask these zeros during the training and testing process so that the network does not use this information. Even though we chose to work with character level tokens in this work, there is a drawback associated with it. You can end up with really long sequences, and that can increase computation requirements.

3.4. Dataset collection

For this dissertation, we chose to work with the Google Code Jam (GCJ) [2] dataset. GCJ is a global coding competition that is held annually online. All of the past and present submissions are publicly available. The GCJ dataset compiled out of the past submissions to the competition over the years. In particular, we downloaded this dataset from this link[3].

3.5. Dataset preparation

In the GCJ dataset for each year, we can see the source code submission of candidates, the source code file name, and their username. From here on, we can consider many approaches to prepare the dataset.

- We can assume that the same username across all the years represents the same author. This way, we can combine the data across all the years resulting in a massive dataset.
- We can treat each year as a separate dataset altogether. This will help us to avoid inducing false positive and false negative errors in the dataset preparation phase.

[2] `https://codingcompetitions.withgoogle.com/codejam`
[3] `https://github.com/Jur1cek/gcj-dataset`

- We can assume that an author never changes the username in the competition and remove any occurrence from the later years and only keep the code files from the first year the author participated.

- We can assume that an author never re-appears in the competition again and rename all the usernames and assign them a unique id and then combine data across the years.

In this work, we choose to treat each year as a separate dataset. We strongly believe that making assumptions that can induce false positive or false negative errors to increase authors' scale can lead to misleading results. Abuhamad et al. chose to work with the first assumption, where they combine authors with the same username across the years to demonstrate the performance of their proposed approach [3]. However, they noted that their dataset preparation approach could induce false-positive errors. We choose to work primarily with the three most popular languages in the competition CPP, Python, and Java. We assess the popularity of the programming languages by looking at the number of source code files associated with each language. To determine a source code's programming language, we look at a source code file's extension.

3.6. Obfuscation tools

We obfuscate the source code files belonging to each programming language to teach our models how to perform authorship attribution even when the source codes are obfuscated. Numerous obfuscation tools are available for the programming languages under consideration. However, we decide to use PyObfx [4], Stunnix [5], and JavaSourceCodeObfuscator[6] to obfuscate Python, C++ and Java source codes respectively. We use the default settings for all the obfuscation tools. It is also important to note that 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 giving a cryptic look to the strings, integers, floats, and booleans. Moreover, it also changes the variables name and imported libraries' names to random non-recognizable strings. Stunnix is a sophisticated C/C++ obfuscator that replaces the symbol name, numeric constants with non-recognizable strings. It also removes the spaces, tabs, and comments from the source

[4] `https://github.com/PyObfx/PyObfx`
[5] `http://stunnix.com/prod/cxxo`
[6] `https://github.com/veylence/JavaSourceCodeObfuscator`

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 above-mentioned obfuscation tools are randomized such that re-executing the obfuscator on the same source code yields different results.

3.7. Dataset preprocessing

We performed the following pre-processing steps:

- We first remove any duplicate source codes.
- We only keep source code files with the following extension: cpp, Java, py, python3, python, cc, cxx, and c++
- We run obfuscation tools on the remaining source code files.
- We remove the obfuscated version of the source code files that were empty. It could be due to many reasons. Either because of the timeout or some error in the obfuscation process.
- We remove any file that has 400 characters or less content. The reason for doing so is to remove tiny source code files. We arrived at the 400 number by calculating the 5th percentile of all the source code file lengths.
- We remove any author that has only one source code file
- We also cut-off excessively long source code files. We do this by only keeping the first 7200 characters of each source code file. We arrived at the 7200 number by calculating the 95th percentile of all the source code file lengths.

3.8. Dataset analysis

In this section we briefly discuss our findings about the prepared dataset. After applying all the pre-processing steps we are left with 3,487,865 source code files. These include obfuscated, and original source code files across 13 years. We note that we end up with slightly more obfuscated files as compared to original source-code files due the minimum length cut-off. In the figure 3.1 you can see the porgramming language distribution across train,validation and test sets.

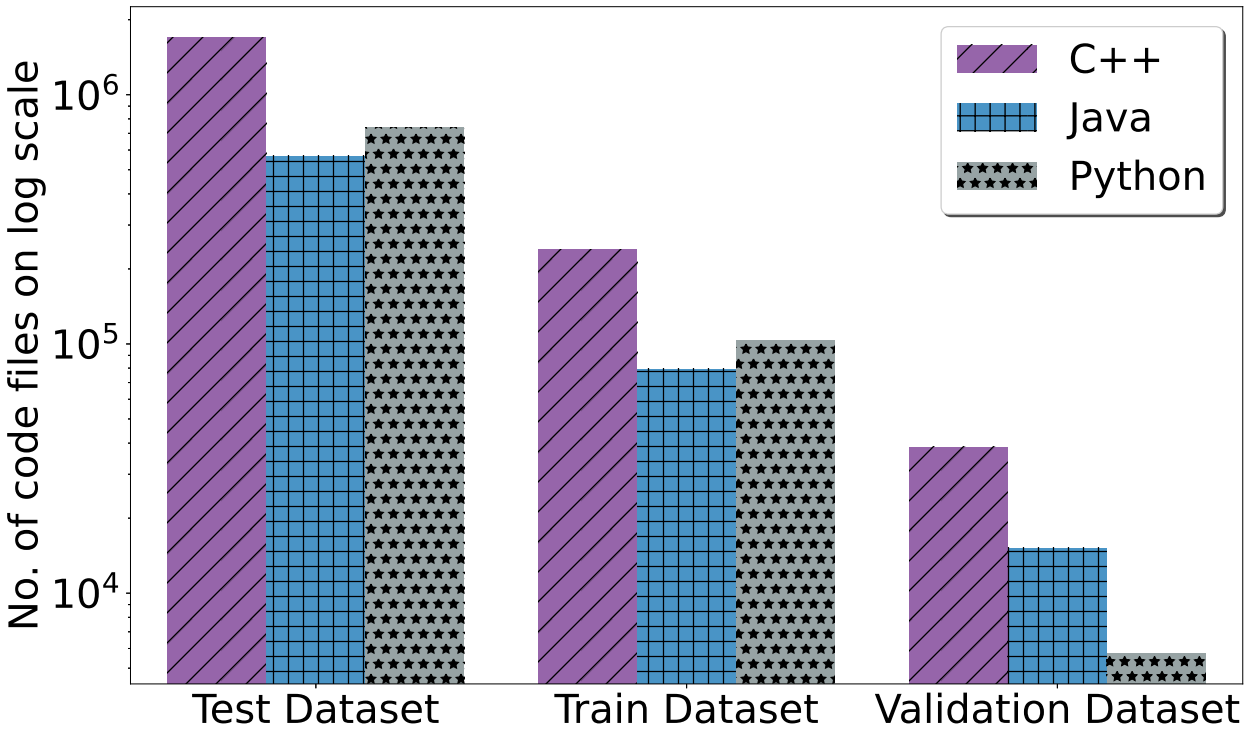


Figure 3.1. Number of source code files associated with different languages across different datasets. Here, the y-axis shows the number of source code files on the log scale (base 10) while the x-axis shows different datasets.

Figure 3.2 shows the distribution of the length of the source code files after pre-processing for the training dataset. According to this distribution, 50% of the source code files have a length of fewer than 2026 characters.

Moreover, the distribution of the number of source code files against the number of authors can be seen in figure 3.3 for the training dataset. We limit the visualization to 80 source code files in figure 3.3, as the distribution is right-tailed. According to this distribution, 50% of the authors have less than 12 source code files. While the average number of files associated with an author within 2018 was 18.65. We discovered that 21076 authors wrote in one programming language, while 1535 wrote in two programming languages, and only 73 authors wrote in all three programming languages in the year 2018.

3.9. Problem framing

As briefly discussed in the introduction, We evaluate our proposed approach to solving source code authorship attribution under three constraints. It is important to note that we assume

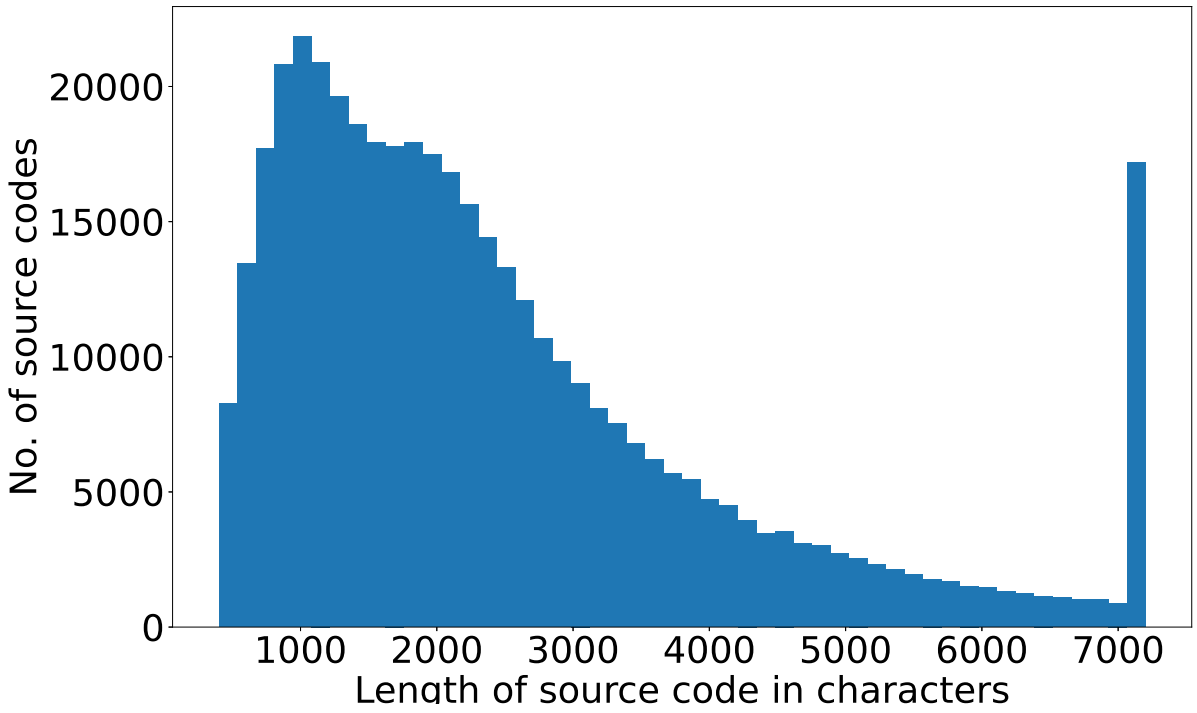


Figure 3.2. Distribution of source code files length. Here the y-axis shows the number of source code files while the x-axis shows the length of source code files in number of characters.

that we are primarily addressing the problem in the close world domain. Given n number of authors and 4, 6, or 8 source code files as priors for each author, we are asked to perform attribution on n source code files where each of those files belongs to precisely one author.

First, we apply the constraint that we will only use the original source codes from the competition to evaluate the models.

Secondly, we apply the constraint that we will evaluate the different models using only the obfuscated version of the source codes mined from the GCJ competition.

We argue that the first constraint is too relaxed as in the real world, all of the files would not be what the author originally wrote. We also believe that the second constraint is too unrealistic. In the real world, All of the files would not be obfuscated. It is safe to assume that we would have a mix of obfuscated source codes and original/non-obfuscated source codes to work in the real world. We also argue that the source codes would not be limited to one specific programming language in the real world because many programmers can write code in multiple programming languages. Hence, we combine the first two constraints to simulate a real-world constraint. We apply the

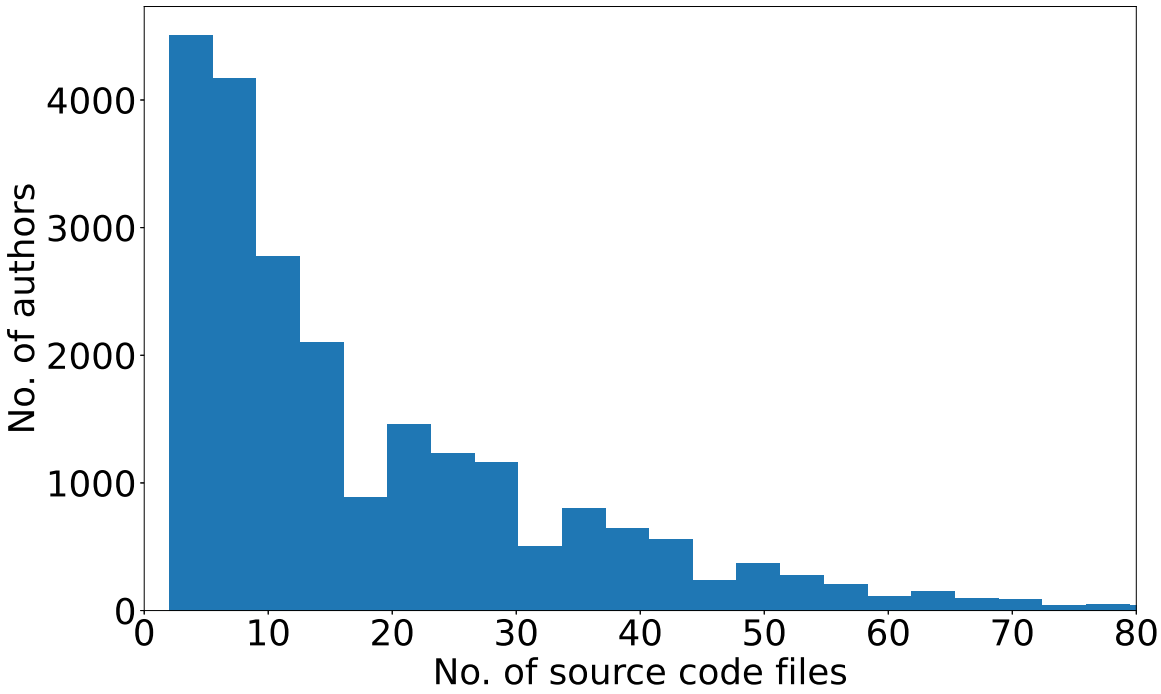


Figure 3.3. Distribution of number of source code files per author. Here, the y-axis shows the number of authors while the x-axis shows the number of source code files.

constraint that a source code being considered can be the original source code or an obfuscated version.

For each constraint discussed above, we evaluate different approaches in two scenarios. In the first scenario, all the training and test source codes belong to only one programming language. In the second scenario, we sample the training and the testing files from all the source code files associated with an author regardless of their programming language. As we are dealing with three programming languages, we will report each programming language's results separately, and we will also report results for the 'Mixed' scenario.

Moreover, we hypothesize that the character-level representation, obfuscation, and the multi-lingual constraint, should force our models to extract meaningful obfuscation-oblivious stylistic features that can even work on programming languages the model has not seen before. To test this hypothesis, we create a separate dataset. We extract source code from the GCJ competition of the year 2020 written in C, C#, Javascript, Ruby, PHP, and Go and evaluate our models' performance on these.

Lastly, we go beyond the closed world approach and address the problem using data from the year 2008 with an open world assumption under the simulated real-world constraint.

3.10. Loss functions

We evaluated the following loss functions in this work:

- Triplet semi-hard loss (TSH)
- Triplet hard loss (TH)
- Soft margin variant of triplet hard loss (TH-Soft)
- Lifted structured loss (LS)

3.11. Train, validation, and test split for NNs

We use all the source codes in the year 2018 to train the NNs with different loss functions. This means that we train the NN on the simulated real-world constraint across all three programming languages. We used the year 2008 as validation to fine-tune the hyper-parameters. Lastly, we use the rest of the individual years as test datasets.

3.12. Model architecture

We use a CNN based architecture inspired by Ruder et al. [29]. Our network takes a character level vector as an input. Here, each character is represented by an integer, as discussed before. 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 [38] with a 25% dropout. This helps the neural network to avoid overfitting. We then use four 1-D convolution layers consisting of 512 filters with kernel sizes of 2, 3, 5, and 7, respectively. The convolution filter weights were initialized using he-normal initialization [39]. Further, to add non-linearity we applied ReLU activation function [40]. To condense the feature maps and retain the most important features, we use global max-pooling operation over time [41], 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's activations. We feed this output to a 256-dimensional dense layer. After this, depending upon the choice of the loss function, we modify the architecture. For lifted structure loss, we apply the ReLU activation function on the output of the dense layer. The weights of the dense layer were initialized using

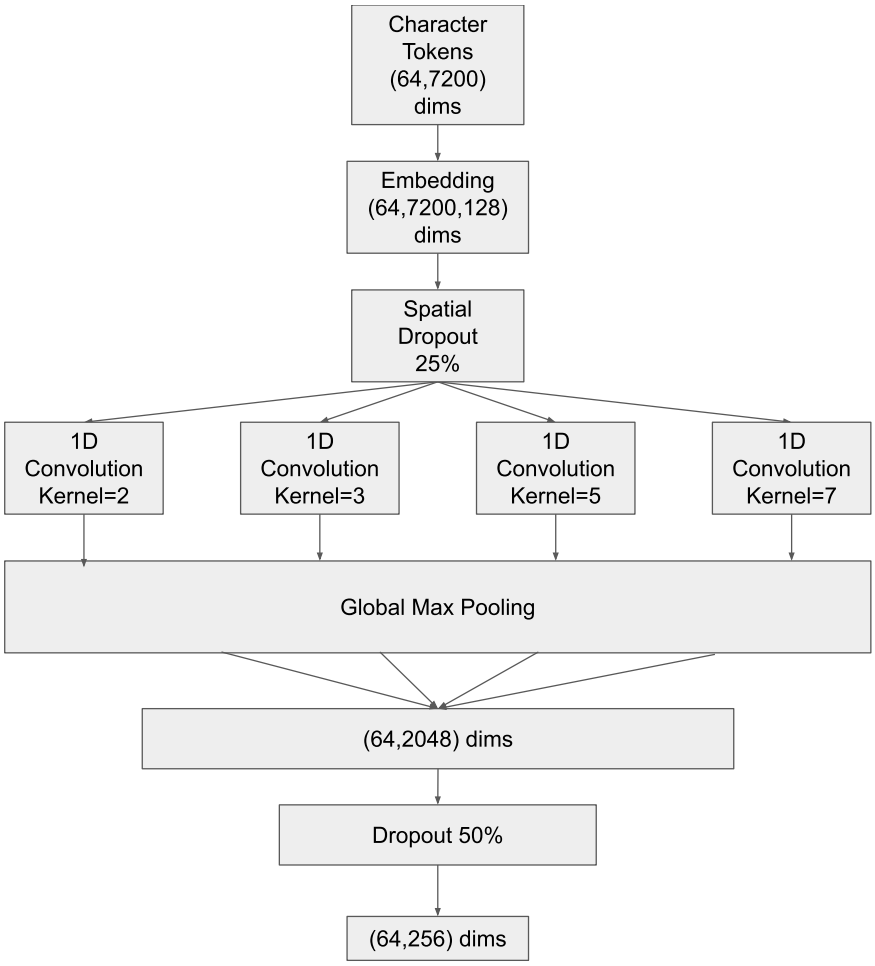


Figure 3.4. CNN model architecture

he-normal initialization [39] in this case. We applied the Tanh activation function on the dense layer output for the triplet loss functions. Every dimension has an upper and lower bound because of this activation. We follow this by an l2 normalization on the resultant vector. The weights of the dense layer were initialized using glorot-uniform initialization [42]. The hyper-parameters of our CNN architecture were chosen after various iterations. Figure 3.4 shows the high-level illustration of our CNN architecture.

3.13. Optimizer

We used rectified adam presented by Liu et al. [43] in conjunction with the Lookahead technique proposed by Zhang et al. [44] to train our deep learning models. Liu et al. claim that adaptive learning rate based optimizers struggle to generalize during the first few batch updates and have very high variance during training. They address these problems as mentioned earlier

by utilizing warm up with a low initial earning rate and not using momentum for the first few training iterations. There is another reason for using learning rate warm-up as it has been argued that by taking big steps at the start of the training process, we can get stuck in a local minima. As for Lookahead, the algorithm chooses a search direction by looking ahead at the 'fast weights' sequence generated by another optimizer. We would encourage the readers to read the respective papers to gain more intuition.

3.14. Batching methodology

Analyzing the dataset reveals that source code files are not evenly spread across authors, so randomly selecting authors and sampling files for training might induce a bias as authors with more files might not get proper representation. To address this concern, we sample an author based on the prior distribution of source code files such that the author with more source code files gets sampled more often in each iteration, but we only select two source code files from that author and put them in a 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.

3.15. Training parameters

Usually, these loss functions are used to fine-tune pre-trained NNs designed to solve image classification problems. Instead of first pre-training our model and then fine-tuning it similarly, we show that we can train our NN from scratch as well. We train all of the neural networks for 100,000 train steps. We utilize the learning rate warm-up, as it has been shown to work well to stabilize the training process. 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 $1e-5$ over the rest of the training process. The intuition behind reducing the learning rate in our context is straightforward. We don't want an outlier batch to drastically modify our neural networks' weights, especially in the later parts of the training process. We noted that this technique allows the network to converge more stably than a fixed learning rate based training regime. We also kept the highest learning rate to be $1-e3$ because anything higher than that would cause the network to start diverging.

Before we proceed, the reader must note that we are not claiming that one loss function is objectively better than the other. We are just reporting the results for our particular bench-

mark with the configuration, as mentioned earlier. It is completely possible that a change in the configuration can change the standing of different loss functions.

Both hard triplet and its soft margin variant collapsed after training for a couple of epochs. Collapsing here means that the network would learn to map everything to a single point. We tried to inspect if the optimizers' specific parameters were causing it, but modifying them did not work. We also tried changing the learning rate and applied gradient clipping, but that didn't help either. We switched to the Stochastic Gradient Descent Optimizer to see if we could train the network, but embedding yielded were stuck in a local minima. We did, however, find an alternative methodology to train the networks with these loss functions. We first train the neural network with the TSH loss function for 35% of the training steps. We then switch the loss function to either TH or its soft margin variant. This is possible because they all share the same architecture. We hypothesize that a NN trained using the TSH loss function allows it not to be stuck in the local minima as before, and neither does the network's embedding's collapse to a single point.

3.16. Classification using embeddings

Intuitively, We use the Nearest Neighbor classifier(NNC) to perform authorship attribution using the learned embeddings. Previously more complex classifiers have been used to perform the attribution step but using the deep metric learning based loss function allows us to the NNC, which is much more scalable than other typically used classifiers.

3.17. Attribution evaluation

We use the accuracy metric to determine our proposed approach's performance in line with previous works that try to address this problem. We also calculate top-3 accuracy to show that even if the nearest source code did not belong to the correct author, we could drastically increase our approach's performance if we look at the top 3 candidates instead. Researchers usually employ a k-fold split in the existing literature to report results over a fixed set of sampled files. We diverge from this because the associated number of files with each author is different; we could induce a sampling bias because of this. To address this concern, We repeat every experiment ten times and report the scores' average and standard deviation. This is done to show the stability of the method and demonstrate that there is no selection bias in our results.

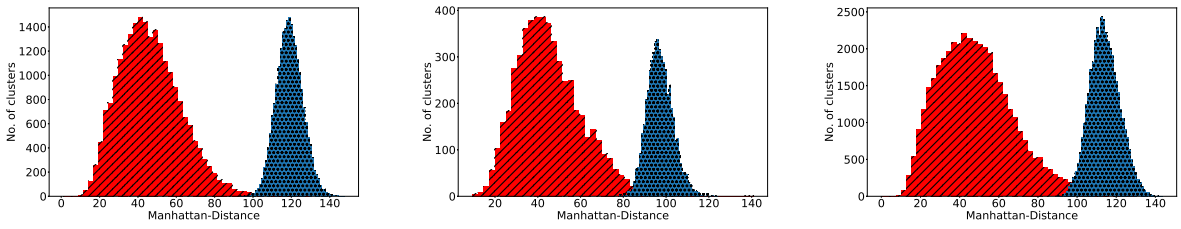
4. EXPERIMENT RESULTS

In this chapter, we discuss the results of the different experiments proposed in the last chapter. We do not discuss the results for all of the years for the sake of brevity. All the results for all of the years have been added in the appendix located at the end of the thesis.

4.1. Embedding evaluation

We visualize the intra-cluster and inter-cluster distance distributions in the year 2018, 2008, and 2020 which are our training, validation, and one of the test years, respectively. Here, a cluster represents a unique author, and we consider all of the files regardless of the programming language and whether it was obfuscated or not. 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 of each cluster's embedding vectors to get the mean inter-cluster distance between the clusters. This kind of analysis aims to get an idea of how well behaved the learned embedding function is without running further authorship attribution experiments. If the distribution structures are visually similar across train, validation, and test and similar overlap between the two distributions, we can get a good idea of how well it would perform in the tests moving forward. This was one of the ways we were able to re-iterate quickly over our proposed NN architectures.

Figure 4.1 shows the inter and intra-cluster distance distributions for Lifted Struct Loss based embeddings. For the year 2018, The average intra-cluster is 46.47±16.21 while the average

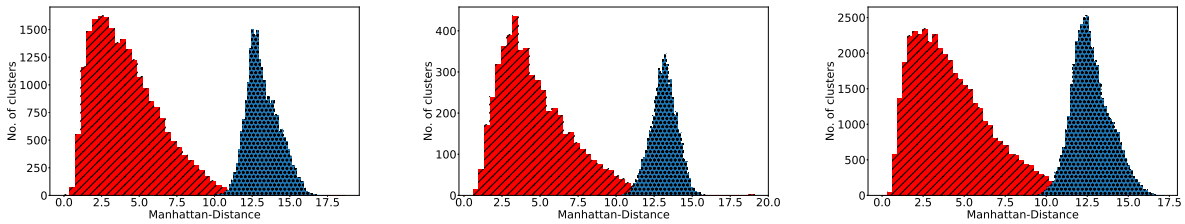


(a) Inter and Intra cluster distances of the year 2018.

(b) Inter and Intra cluster distances of the year 2008.

(c) Inter and Intra cluster distances of the year 2020.

Figure 4.1. Lifted struct: Distribution of inter and intra-cluster distances in training (2018), validation set (2008), and one of the test years (2020). Here, the red distribution shows the inter cluster distances while dotted blue distribution shows the intra-cluster distances.



(a) Inter and Intra cluster distances of the year 2018.

(b) Inter and Intra cluster distances of the year 2008.

(c) Inter and Intra cluster distances of the year 2020.

Figure 4.2. Triplet semi-hard: Distribution of inter and intra-cluster distances in training (2018), validation set (2008), and one of the test years (2020). Here, the red distribution shows the inter cluster distances while dotted blue distribution shows the intra-cluster distances.

inter-cluster distance is 119.03±6.80. We note that the minimum inter-cluster for 2018 is 93.52, while the maximum intra-cluster distance is 128.69. We also noted that only 158, which is 0.69% of the total number of clusters, have an intra-cluster distance greater than the minimum of inter-cluster distances.

For the year 2008, The average intra-cluster is 45.38±14.99. In comparison, the average inter-cluster distance is 97.20±6.07. The minimum inter-cluster for the year 2008 is 78.20, while the maximum intra-cluster distance is 140.88. We also note that only 133, which is 2.53% of the total number of clusters, have an intra-cluster distance greater than the minimum of inter-cluster distances.

For the year 2020, The average intra-cluster is 48.79±19.61. At the same time, the average inter-cluster distance is 114.13±7.65. The minimum inter-cluster for the year 2020 is 89.08, while the maximum intra-cluster distance is 128.69. We also note that only 1600, which is 3.89% of the total number of clusters, have an intra-cluster distance greater than the minimum of inter-cluster distances.

We observe that the intra-cluster distances are significantly lower than the inter-cluster distances across different years. This means that the embedding space's distance can discriminate between different authors' source code files. We can also see that the distributions across training, validation, and test sets are quite similar, reflecting that our model did not overfit and generalize well.

Figure 4.2 shows the inter and intra-cluster distance distributions for Triplet SemiHard Loss based embeddings. It is important to note that the distance ranges between triplet based

loss functions and Lifted Struct loss are because we employed normalization in one and not the other. For 2018, The average intra-cluster is 4.21±2.27, while the average inter-cluster distance is 13.21±1.06. The minimum inter-cluster for the year 2018 is 9.79, while the maximum intra-cluster distance is 18.67. We also note that only 431, which is 1.90% of the total number of clusters, have an intra-cluster distance greater than the minimum of inter-cluster distances.

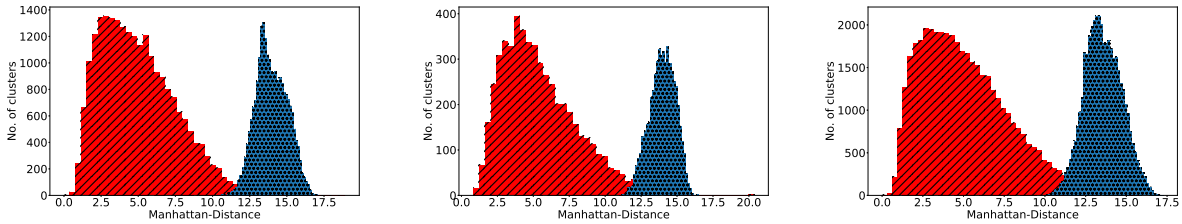
For the year 2008, The average intra-cluster is 4.71±2.32. In comparison, the average inter-cluster distance is 13.14±0.83. The minimum inter-cluster for the year 2008 is 10.41, while the maximum intra-cluster distance is 19.07. We also note that only 100, which is 1.90% of the total number of clusters, have an intra-cluster distance greater than the minimum of inter-cluster distances.

For the year 2020, The average intra-cluster is 4.31±2.43. In contrast, the average inter-cluster distance is 12.75±1.15. The minimum inter-cluster for the year 2020 is 9.25, while the maximum intra-cluster distance is 15.34. We also note that only 1841, which is 4.48% of the total number of clusters, have an intra-cluster distance greater than the minimum of inter-cluster distances.

We observe that the intra-cluster distances are again significantly lower than the inter-cluster distances across different years. This means that the embedding space's distance can discriminate between different authors' source code files reliably. We can also see that the distributions across training, validation, and test sets are quite similar, reflecting that our model did not overfit and generalize well. Although it is hard to compare these numbers with that of the Lifted Struct Loss based embeddings, we can observe a slight degradation when we look at the distribution overlap for the year 2020.

Figure 4.3 shows the inter and intra-cluster distance distributions for Triplet Hard Loss based embeddings. For the year 2018, The average intra-cluster is 4.97±2.49, while the average inter-cluster distance is 13.96±1.05. The minimum inter-cluster for the year 2018 is 10.18, while the maximum intra-cluster distance is 18.92. We also note that only 716, which is 3.15% of the total number of clusters, have an intra-cluster distance greater than the minimum of inter-cluster distances.

For the year 2008, The average intra-cluster is 5.39±2.51. In comparison, the average inter-cluster distance is 13.97±0.89. The minimum inter-cluster for the year 2008 is 11.10, while



(a) Inter and Intra cluster distances of the year 2018.

(b) Inter and Intra cluster distances of the year 2008.

(c) Inter and Intra cluster distances of the year 2020.

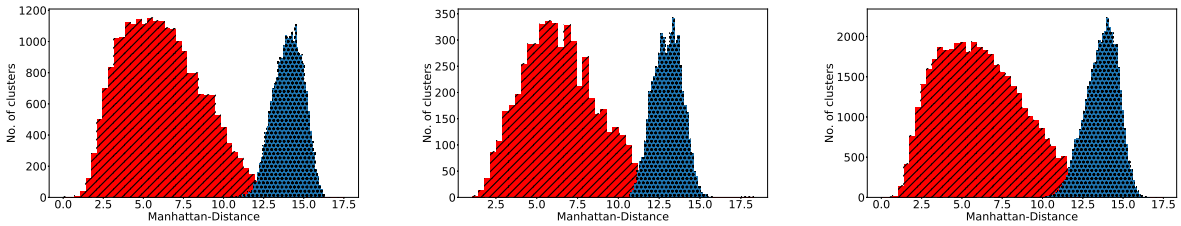
Figure 4.3. Triplet hard: Distribution of inter and intra-cluster distances in training (2018), validation set (2008), and one of the test years (2020). Here, the red distribution shows the inter cluster distances while dotted blue distribution shows the intra-cluster distances.

the maximum intra-cluster distance is 20.37. We also note that only 132, which is 2.51% of the total number of clusters, have an intra-cluster distance greater than the minimum of inter-cluster distances.

For the year 2020, The average intra-cluster is 5.08±2.66. In contrast, the average inter-cluster distance is 13.57±1.13. The minimum inter-cluster for the year 2020 is 9.85, while the maximum intra-cluster distance is 15.78. We also note that only 2381, which is 5.80% of the total number of clusters, have an intra-cluster distance to be greater than the minimum of inter-cluster distances.

We observe that the intra-cluster distances are again significantly lower than the inter-cluster distances across different years. This means that the embedding space's distance can discriminate well between different authors' source code files. We can also see that the distributions across training, validation, and test sets are quite similar, reflecting that our model did not overfit and generalize well. However, we notice a slight degradation in performance when comparing the distance numbers with Triplet SemiHard based embedding.

Figure 4.4 shows the inter and intra-cluster distance distributions for Triplet Hard Soft Margin Variant Loss based embeddings. For the year 2018, The average intra-cluster is 6.18±2.50, while the average inter-cluster distance is 14.04±0.91. The minimum inter-cluster for the year 2018 is 10.47, while the maximum intra-cluster distance is 17.54. We also note that only 1267, which is 5.58% of the total number of clusters, have an intra-cluster distance greater than the minimum of inter-cluster distances.



(a) Inter and Intra cluster distances of the year 2018.

(b) Inter and Intra cluster distances of the year 2008.

(c) Inter and Intra cluster distances of the year 2020.

Figure 4.4. Triplet hard soft: Distribution of inter and intra-cluster distances in training (2018), validation set (2008), and one of the test years (2020). Here, the red distribution shows the inter cluster distances while dotted blue distribution shows the intra-cluster distances.

For the year 2008, The average intra-cluster is 6.51±2.34. At the same time, the average inter-cluster distance is 13.00±0.85. The minimum inter-cluster for the year 2008 is 10.70, while the maximum intra-cluster distance is 18.27. We also note that only 263, which is 5.01% of the total number of clusters, have an intra-cluster distance greater than the minimum of inter-cluster distances.

For the year 2020, The average intra-cluster is 6.23±2.72. In contrast, the average inter-cluster distance is 13.66±0.99. The minimum inter-cluster for the year 2020 is 10.02, while the maximum intra-cluster distance is 17.45. We also noted that only 4133, which is 10.07% of the total number of clusters, were found to have an intra-cluster distance greater than the minimum of inter-cluster distances.

We observe that the intra-cluster distances are again significantly lower than the inter-cluster distances across different years. This means that the embedding space's distance can discriminate between the source code files written by different authors. We can also see that the distributions across training, validation, and test sets are quite similar, reflecting that our model did not overfit and generalize well. We, however, again notice a degradation in performance when comparing the distance numbers with Triplet SemiHard and Triplet Hard based embedding.

According to this preliminary analysis, we thought that LS based embeddings would yield the best performance while TSH based embedding would be in second place. Third and fourth place would be secured by TH and TH-Soft, respectively. However, further analysis upsets our early guesses.

4.2. Closed world source code authorship attribution

In this section, we will discuss results from all the different constraints across all the loss functions.

4.2.1. Performance on original source code only dataset

This subsection discusses and compares the results across different loss functions using only the original source code mined for the GCJ dataset.

4.2.1.1. Five files per author

LS based embeddings achieve an accuracy of 84.79±0.4% in the mixed language scenario for the year 2018. We see an average -1.63±3.44% difference in the accuracy of all years from the year 2018 in this particular scenario. For C++, these embeddings achieve an 88.18±0.2% accuracy for 2018 with an average difference of -2.95±2.98% compared to the rest of the years. While for Python, it achieves an accuracy of 82.26±0.5% in the year 2018 with an average difference of 4.32±6.0% compared to the rest of the years. Lastly, for Java, these embeddings achieve an accuracy of 84.79±0.4% in the year 2018 with an average difference of -1.0±3.51% compared to the rest of the years.

These embeddings achieve a top-3 accuracy of 89.14±0.3% in the mixed language scenario for the year 2018, resulting in a difference of 4.35 compared to simple accuracy for the year 2018. We note an average 5.3±1.04% difference in the top-3 accuracy and simple accuracy across all years. For C++, these embeddings achieve a top-3 accuracy of 92.12±0.3% for the year 2018, resulting in a difference of 3.94 compared to simple accuracy for the year 2018. We note an average 4.98±0.95% difference in the top-3 accuracy and simple accuracy across all years. While for Python, these embeddings achieve a top-3 accuracy of 87.75±0.3% for the year 2018, resulting in a difference of 5.49 compared to simple accuracy for the year 2018. We note an average 4.99±1.33% difference in the top-3 accuracy and simple accuracy across all years. Lastly, for Java, these embeddings achieve a top-3 accuracy of 89.42±0.4% for the year 2018, resulting in a difference of 4.63 compared to simple accuracy for the year 2018. We note an average 5.5±1.05% difference in the top-3 accuracy and simple accuracy across all years.

TSH based embeddings achieve an accuracy of 75.87±0.2% in the mixed language scenario for the year 2018. We see an average 2.03±3.64% difference in the accuracy of all years from the year

2018 in this particular scenario. For C++, these embeddings achieve an accuracy of 80.37±0.2% in the year 2018 with an average difference of -1.4±3.31% compared to the rest of the years. While for Python, they achieve an accuracy of 71.4±1.2% in the year 2018 with an average difference of 11.88±7.73% compared to the rest of the years. Lastly, for Java, these embeddings achieve an accuracy of 75.77±0.5% in the year 2018 with an average difference of 4.75±4.36% compared to the rest of the years.

These embeddings achieve a top-3 accuracy of 81.05±0.3% in the mixed language scenario for the year 2018, resulting in a difference of 5.18 compared to simple accuracy for the year 2018. We note an average 5.92±1.06% difference in the top-3 accuracy and simple accuracy across all years. For C++, these embeddings achieve a top-3 accuracy of 84.81±0.5% for the year 2018, resulting in a difference of 4.44 compared to simple accuracy for the year 2018. We note an average 5.76±1.21% difference in the top-3 accuracy and simple accuracy across all years. While for Python, these embeddings achieve a top-3 accuracy of 77.0±0.3% for the year 2018, resulting in a difference of 5.6 compared to simple accuracy for the year 2018. We note an average 5.44±1.26% difference in the top-3 accuracy and simple accuracy across all years. Lastly, for Java, these embeddings achieve a top-3 accuracy of 80.55±0.4% for the year 2018, resulting in a difference of 4.78 compared to simple accuracy for the year 2018. We note an average 5.37±0.93% difference in the top-3 accuracy and simple accuracy across all years.

TH based embeddings achieve an accuracy of 74.4±0.3% in the mixed language scenario for the year 2018. We see an average 2.08±3.63% difference in the accuracy of all years from the year 2018 in this particular scenario. For C++, these embeddings achieve an accuracy of 78.6±0.3% in the year 2018 with an average difference of -1.0±3.07% compared to the rest of the years. While for Python, they achieve an accuracy of 70.22±0.9% in the year 2018 with an average difference of 11.64±7.58% compared to the rest of the years. Lastly, for Java, these embeddings achieve an accuracy of 74.79±0.8% in the year 2018 with an average difference of 3.93±4.55% compared to the rest of the years.

These embeddings achieve a top-3 accuracy of 79.49±0.2% in the mixed language scenario for the year 2018, resulting in a difference of 5.09 compared to simple accuracy for the year 2018. We note an average 6.11±1.04% difference in the top-3 accuracy and simple accuracy across all years. For C++, these embeddings achieve a top-3 accuracy of 83.31±0.4% for the year 2018, resulting in

a difference of 4.71 compared to simple accuracy for the year 2018. We note an average 6.1±1.2% difference in the top-3 accuracy and simple accuracy across all years. While for Python, these embeddings achieve a top-3 accuracy of 76.35±0.7% for the year 2018, which results in a difference of 6.13 compared to simple accuracy for the year 2018. We note an average 6.03±1.36% difference in the top-3 accuracy and simple accuracy across all years. Lastly, for Java, these embeddings achieve a top-3 accuracy of 79.37±1.0% for the year 2018, resulting in a difference of 4.58 compared to simple accuracy for the year 2018. We note an average 5.62±1.15% difference in the top-3 accuracy and simple accuracy across all years.

TH-soft based embeddings achieve an accuracy of 79.57±0.2% in the mixed language scenario for the year 2018. We see an average 0.45±4.07% difference in the accuracy of all years from the year 2018 in this particular scenario. For C++, these embeddings achieve an accuracy of 82.82±0.3% in the year 2018 with an average difference of -1.86±3.72% compared to the rest of the years. While for Python, they achieve an accuracy of 76.14±0.6% in the year 2018 with an average difference of 9.36±7.62% compared to the rest of the years. Lastly, for Java, these embeddings achieve an accuracy of 81.06±0.7% in the year 2018 with an average difference of 1.56±4.37% compared to the rest of the years.

These embeddings achieve a top-3 accuracy of 84.37±0.2% in the mixed language scenario for the year 2018, resulting in a difference of 4.8 compared to simple accuracy for the year 2018. We note an average 5.48±0.97% difference in the top-3 accuracy and simple accuracy across all years. For C++, these embeddings achieve a top-3 accuracy of 87.1±0.2% for the year 2018, resulting in a difference of 4.28 compared to simple accuracy for the year 2018. We note an average 5.42±1.01% difference in the top-3 accuracy and simple accuracy across all years. While for Python, these embeddings achieve a top-3 accuracy of 82.13±0.5% for the year 2018, which results in a difference of 5.99 compared to simple accuracy for the year 2018. We note an average 5.05±1.19% difference in the top-3 accuracy and simple accuracy across all years. Lastly, for Java, these embeddings achieve a top-3 accuracy of 85.81±0.5% for the year 2018, resulting in a difference of 4.75 compared to simple accuracy for the year 2018. We note an average 5.18±1.33% difference in the top-3 accuracy and simple accuracy across all years.

4.2.1.2. Seven files per author

LS based embeddings achieve an accuracy of 88.99±0.3% in the mixed language scenario for the year 2018. We see an average -0.33±2.99% difference in the accuracy of all years from the year 2018 in this particular scenario. For C++, these embeddings achieve an accuracy of 91.86±0.2% in the year 2018 with an average difference of -1.61±2.43% compared to the rest of the years. While for Python, they achieve an accuracy of 86.49±0.5% in the year 2018 with an average difference of 5.59±5.0% compared to the rest of the years. Lastly, for Java, these embeddings achieve an accuracy of 89.2±0.5% in the year 2018 with an average difference of 0.44±2.89% compared to the rest of the years.

These embeddings achieve a top-3 accuracy of 92.91±0.3% in the mixed language scenario for the year 2018, resulting in a difference of 3.92 compared to simple accuracy for the year 2018. We note an average 4.15±0.92% difference in the top-3 accuracy and simple accuracy across all years. For C++, these embeddings achieve a top-3 accuracy of 94.83±0.2% for the year 2018, resulting in a difference of 2.97 compared to simple accuracy for the year 2018. We note an average 3.94±0.98% difference in the top-3 accuracy and simple accuracy across all years. While for Python, these embeddings achieve a top-3 accuracy of 91.66±0.5% for the year 2018, resulting in a difference of 5.17 compared to simple accuracy for the year 2018. We note an average 3.46±1.09% difference in the top-3 accuracy and simple accuracy across all years. Lastly, for Java, these embeddings achieve a top-3 accuracy of 93.07±0.6% for the year 2018, resulting in a difference of 3.87 compared to simple accuracy for the year 2018. We note an average 4.13±0.89% difference in the top-3 accuracy and simple accuracy across all years.

TSH based embeddings achieve an accuracy of 81.57±0.3% in the mixed language scenario for the year 2018. We see an average of 2.84±3.3% difference in the accuracy of all years from 2018 in this particular scenario. For C++, these embeddings achieve an accuracy of 85.41±0.5% in the year 2018 with an average difference of -0.06±2.89% compared to the rest of the years. While for Python, they achieve an accuracy of 77.44±0.7% in the year 2018 with an average difference of 12.07±7.09% compared to the rest of the years. Lastly, for Java, these embeddings achieve an accuracy of 81.98±0.6% in the year 2018 with an average difference of 5.16±3.84% compared to the rest of the years.

These embeddings achieve a top-3 accuracy of 86.43±0.3% in the mixed language scenario for the year 2018, resulting in a difference of 4.86 compared to simple accuracy for the year 2018. We note an average of 4.97±0.81% difference in the top-3 accuracy and simple accuracy across all years. For C++, these embeddings achieve a top-3 accuracy of 89.49±0.4% for the year 2018, resulting in a difference of 4.08 compared to simple accuracy for the year 2018. We note an average 5.06±1.11% difference in the top-3 accuracy and simple accuracy across all years. While for Python, these embeddings achieve a top-3 accuracy of 83.13±0.4% for the year 2018, resulting in a difference of 5.69 compared to simple accuracy for the year 2018. We note an average 4.06±1.34% difference in the top-3 accuracy and simple accuracy across all years. Lastly, for Java, these embeddings achieve a top-3 accuracy of 85.78±0.7% for the year 2018, resulting in a difference of 3.8 compared to simple accuracy for the year 2018. We note an average 4.31±0.85% difference in the top-3 accuracy and simple accuracy across all years.

TH based embeddings achieve an accuracy of 80.23±0.2% in the mixed language scenario for the year 2018. We see an average of 2.95±3.35% difference in the accuracy of all years from 2018 in this particular scenario. For C++, these embeddings achieve an accuracy of 83.92±0.5% in the year 2018 with an average difference of 0.28±2.84% compared to the rest of the years. While for Python, they achieve an accuracy of 76.03±0.6% in the year 2018 with an average difference of 12.77±6.92% compared to the rest of the years. Lastly, for Java, these embeddings achieve an accuracy of 80.49±0.9% in the year 2018 with an average difference of 5.22±3.74% compared to the rest of the years.

These embeddings achieve a top-3 accuracy of 85.01±0.3% in the mixed language scenario for the year 2018, resulting in a difference of 4.78 compared to simple accuracy for the year 2018. We note an average 5.34±0.94% difference in the top-3 accuracy and simple accuracy across all years. For C++, these embeddings achieve a top-3 accuracy of 87.95±0.3% for the year 2018, resulting in a difference of 4.03 compared to simple accuracy for the year 2018. We note an average 5.39±1.18% difference in the top-3 accuracy and simple accuracy across all years. While for Python, these embeddings achieve a top-3 accuracy of 82.06±0.6% for the year 2018, resulting in a difference of 6.03 compared to simple accuracy for the year 2018. We note an average 4.09±1.32% difference in the top-3 accuracy and simple accuracy across all years. Lastly, for Java, these embeddings achieve a top-3 accuracy of 84.94±0.6% for the year 2018, resulting in a difference of 4.45 compared to

simple accuracy for the year 2018. We note an average 4.89±0.77% difference in the top-3 accuracy and simple accuracy across all years.

TH-soft based embeddings achieve an accuracy of 84.49±0.3% in the mixed language scenario for the year 2018. We see an average 1.58±3.59% difference in the accuracy of all years from the year 2018 in this particular scenario. For C++, these embeddings achieve an accuracy of 87.6±0.5% in the year 2018 with an average difference of -0.61±3.14% compared to the rest of the years. While for Python, they achieve an accuracy of 81.14±0.8% in the year 2018 with an average difference of 9.87±6.73% compared to the rest of the years. Lastly, for Java, these embeddings achieve an accuracy of 86.37±0.4% in the year 2018 with an average difference of 2.19±3.63% compared to the rest of the years.

These embeddings achieve a top-3 accuracy of 89.04±0.2% in the mixed language scenario for the year 2018, resulting in a difference of 4.55 compared to simple accuracy for the year 2018. We note an average 4.55±0.75% difference in the top-3 accuracy and simple accuracy across all years. For C++, these embeddings achieve a top-3 accuracy of 91.47±0.3% for the year 2018, resulting in a difference of 3.87 compared to simple accuracy for the year 2018. We note an average 4.56±0.99% difference in the top-3 accuracy and simple accuracy across all years. While for Python, these embeddings achieve a top-3 accuracy of 86.64±0.5% for the year 2018, resulting in a difference of 5.5 compared to simple accuracy for the year 2018. We note an average 3.59±1.26% difference in the top-3 accuracy and simple accuracy across all years. Lastly, for Java, these embeddings achieve a top-3 accuracy of 90.39±0.6% for the year 2018, resulting in a difference of 4.02 compared to simple accuracy for the year 2018. We note an average 4.15±1.06% difference in the top-3 accuracy and simple accuracy across all years.

We note a noticeable increase in performance with just two more files as priors, but one could also argue that this increase in performance is due to decreased author count.

4.2.1.3. Nine files per author

LS based embeddings achieve an accuracy of 91.59±0.2% in the mixed language scenario for the year 2018. We see an average 0.65±2.47% difference in the accuracy of all years from the year 2018 in this particular scenario. For C++, these embeddings achieve an accuracy of 93.89±0.3% in the year 2018 with an average difference of -0.47±1.91% compared to the rest of the years. While for Python, they achieve an accuracy of 89.85±0.5% in the year 2018 with an average difference

of 5.2±4.56% compared to the rest of the years. Lastly, for Java, these embeddings achieve an accuracy of 91.92±0.6% in the year 2018 with an average difference of 1.28±2.34% compared to the rest of the years.

These embeddings achieve a top-3 accuracy of 94.63±0.2% in the mixed language scenario for the year 2018, resulting in a difference of 3.04 compared to simple accuracy for the year 2018. We note an average 2.87±0.73% difference in the top-3 accuracy and simple accuracy across all years. For C++, these embeddings achieve a top-3 accuracy of 96.34±0.2% for the year 2018, resulting in a difference of 2.45 compared to simple accuracy for the year 2018. We note an average 2.77±0.85% difference in the top-3 accuracy and simple accuracy across all years. While for Python, these embeddings achieve a top-3 accuracy of 93.59±0.5% for the year 2018, resulting in a difference of 3.74 compared to simple accuracy for the year 2018. We note an average 1.99±1.29% difference in the top-3 accuracy and simple accuracy across all years. Lastly, for Java, these embeddings achieve a top-3 accuracy of 95.04±0.5% for the year 2018, resulting in a difference of 3.12 compared to simple accuracy for the year 2018. We note an average 2.86±0.83% difference in the top-3 accuracy and simple accuracy across all years.

TSH based embeddings achieve an accuracy of 85.25±0.3% in the mixed language scenario for the year 2018. We see an average 3.49±3.19% difference in the accuracy of all years from the year 2018 in this particular scenario. For C++, these embeddings achieve an accuracy of 88.45±0.5% in the year 2018, with an average difference of 1.32±2.48% compared to the rest of the years. While for Python, they achieve an accuracy of 81.72±0.9% in the year 2018 with an average difference of 11.37±6.39% compared to the rest of the years. Lastly, for Java, these embeddings achieve an accuracy of 84.94±0.7% in the year 2018 with an average difference of 6.41±3.46% compared to the rest of the years.

These embeddings achieve a top-3 accuracy of 89.19±0.4% in the mixed language scenario for the year 2018, resulting in a difference of 3.94 compared to simple accuracy for the year 2018. We note an average 3.91±0.7% difference in the top-3 accuracy and simple accuracy across all years. For C++, these embeddings achieve a top-3 accuracy of 92.16±0.4% for the year 2018, resulting in a difference of 3.71 compared to simple accuracy for the year 2018. We note an average 3.95±0.91% difference in the top-3 accuracy and simple accuracy across all years. While for Python, these embeddings achieve a top-3 accuracy of 86.1±0.6% for the year 2018, resulting in a difference of

4.38 compared to simple accuracy for the year 2018. We note an average 2.65±1.02% difference in the top-3 accuracy and simple accuracy across all years. Lastly, for Java, these embeddings achieve a top-3 accuracy of 88.92±0.5% for the year 2018, resulting in a difference of 3.98 compared to simple accuracy for the year 2018. We note an average 3.14±0.79% difference in the top-3 accuracy and simple accuracy across all years.

TH based embeddings achieve an accuracy of 83.88±0.3% in the mixed language scenario for the year 2018. We see an average of 3.96±3.16% difference in the accuracy of all years from 2018 in this particular scenario. For C++, these embeddings achieve an accuracy of 86.84±0.3% in the year 2018 with an average difference of 1.97±2.46% compared to the rest of the years. While for Python, they achieve an accuracy of 80.24±0.6% in the year 2018 with an average difference of 12.31±6.41% compared to the rest of the years. Lastly, for Java, these embeddings achieve an accuracy of 83.76±0.6% in the year 2018 with an average difference of 6.4±3.46% compared to the rest of the years.

These embeddings achieve a top-3 accuracy of 87.89±0.2% in the mixed language scenario for the year 2018, resulting in a difference of 4.01 compared to simple accuracy for the year 2018. We note an average 4.14±0.69% difference in the top-3 accuracy and simple accuracy across all years. For C++, these embeddings achieve a top-3 accuracy of 90.63±0.4% for the year 2018, resulting in a difference of 3.79 compared to simple accuracy for the year 2018. We note an average 4.33±1.0% difference in the top-3 accuracy and simple accuracy across all years. While for Python, these embeddings achieve a top-3 accuracy of 85.63±0.8% for the year 2018, resulting in a difference of 5.39 compared to simple accuracy for the year 2018. We note an average 2.81±1.14% difference in the top-3 accuracy and simple accuracy across all years. Lastly, for Java, these embeddings achieve a top-3 accuracy of 87.84±0.5% for the year 2018, resulting in a difference of 4.08 compared to simple accuracy for the year 2018. We note an average 3.75±0.71% difference in the top-3 accuracy and simple accuracy across all years.

TH-soft based embeddings achieve an accuracy of 87.89±0.2% in the mixed language scenario for the year 2018. We see an average 2.17±3.13% difference in the accuracy of all years from the year 2018 in this particular scenario. For C++, these embeddings achieve an accuracy of 90.44±0.4% in the year 2018 with an average difference of 0.44±2.64% compared to the rest of the years. While for Python, they achieve an accuracy of 84.92±0.7% in the year 2018 with an average

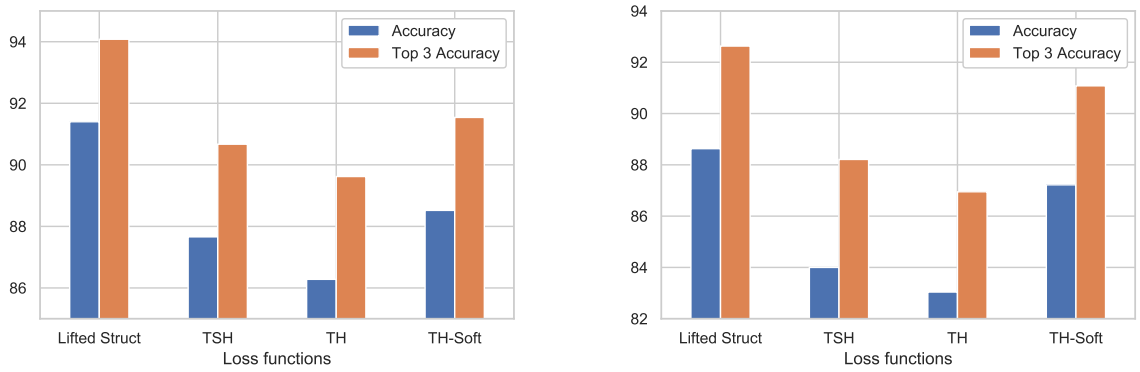
difference of 9.0±5.86% compared to the rest of the years. Lastly, for Java, these embeddings achieve an accuracy of 89.38±0.7% in the year 2018 with an average difference of 3.07±2.81% compared to the rest of the years.

These embeddings achieve a top-3 accuracy of 91.46±0.3% in the mixed language scenario for the year 2018, resulting in a difference of 3.57 compared to simple accuracy for the year 2018. We note an average 3.42±0.7% difference in the top-3 accuracy and simple accuracy across all years. For C++, these embeddings achieve a top-3 accuracy of 93.48±0.2% for the year 2018, resulting in a difference of 3.04 compared to simple accuracy for the year 2018. We note an average 3.55±1.01% difference in the top-3 accuracy and simple accuracy across all years. While for Python, these embeddings achieve a top-3 accuracy of 89.25±0.6% for the year 2018, resulting in a difference of 4.33 compared to simple accuracy for the year 2018. We note an average 2.49±1.13% difference in the top-3 accuracy and simple accuracy across all years. Lastly, for Java, these embeddings achieve a top-3 accuracy of 92.36±0.6% for the year 2018, resulting in a difference of 2.98 compared to simple accuracy for the year 2018. We note an average 2.85±0.92% difference in the top-3 accuracy and simple accuracy across all years.

We note a noticeable increase in performance yet again with the increase in the number of files as priors. However, again, one could argue that this increase in performance is due to the decrease in author count and not due to the increase in priors. In Figure 4.5 we compare the performance of the different loss functions in different scenarios under this particular scenario for the year 2020. As we can LS based embeddings always outperforms the TSH, TH, and its Soft margin variant based embeddings.

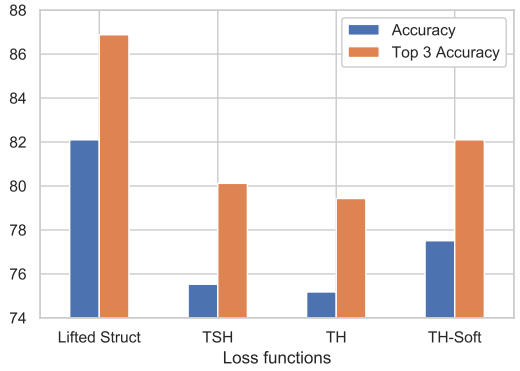
Table 4.1. LS: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages using the original GCJ dataset.

		C++			Python			Java			All Languages		
Year	**Files**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3 Accuracy**
2018	5	7830	88.18±0.2	92.12±0.3	3525	82.26±0.5	87.75±0.3	2726	84.79±0.4	89.42±0.4	13983	84.79±0.4	89.14±0.3
	7	6376	91.86±0.2	94.83±0.2	2620	86.49±0.5	91.66±0.5	2083	89.2±0.5	93.07±0.6	11152	88.99±0.3	92.91±0.3
	9	5277	93.89±0.3	96.34±0.2	2054	89.85±0.5	93.59±0.5	1608	91.92±0.6	95.04±0.5	9053	91.59±0.2	94.63±0.2
2019	5	10122	86.69±0.2	90.09±0.2	6281	81.85±0.5	85.38±0.4	3921	82.25±0.5	86.35±0.6	19672	81.86±0.2	85.37±0.2
	7	8087	90.8±0.5	93.57±0.2	4840	86.78±0.4	89.86±0.4	3069	87.25±0.5	90.54±0.5	15973	86.65±0.2	89.91±0.2
	9	6468	93.19±0.3	95.27±0.2	3759	89.6±0.4	92.05±0.3	2373	90.1±0.7	92.95±0.3	12952	89.65±0.2	92.25±0.1
2020	5	16872	83.59±0.4	87.62±0.2	9124	70.65±0.3	76.16±0.5	3982	79.65±0.7	84.96±0.4	29838	77.15±0.3	81.83±0.2
	7	13694	88.41±0.3	91.86±0.2	7130	77.9±0.3	83.04±0.3	3125	85.25±0.6	89.71±0.4	24281	83.11±0.1	87.25±0.1
	9	11143	91.4±0.3	94.08±0.2	5513	82.1±0.4	86.88±0.4	2384	88.63±0.6	92.63±0.4	19529	86.95±0.2	90.3±0.2

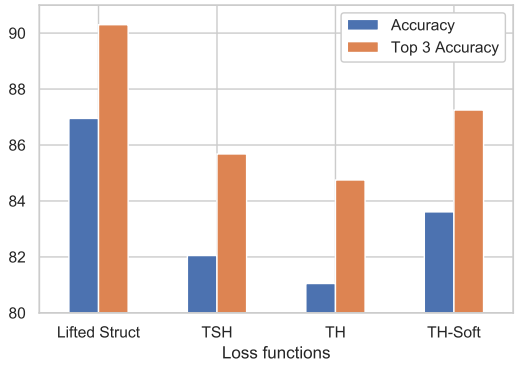


(a) CPP

(b) Java



(c) Python



(d) Mixed

Figure 4.5. Loss function performance comparison using the original GCJ dataset

4.2.2. Performance on obfuscated source code dataset

In this subsection, we compare the results across different loss functions on the obfuscated only constraint. In addition to the similar analysis that we did for the original source code only constraint, we can also compare the two constraints' performance difference. We can do so because almost every original source code file has an obfuscated version, which leads to an almost equal number of authors in each year in each scenario.

4.2.2.1. Five files per author

LS based embeddings achieve an accuracy of 74.48±0.3% in the mixed language scenario for the year 2018. We see an average -2.82±3.41% difference in the accuracy of all years from the year 2018 in this particular scenario. For C++, these embeddings achieve an accuracy of 80.56±0.4% in the year 2018 with an average difference of -3.26±3.43% compared to the rest of the years. While for Python, they achieve an accuracy of 68.35±0.8% in the year 2018 with an average difference of

Table 4.2. TSH: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages using the original GCJ dataset.

		C++			Python			Java			All Languages		
Year	**Files**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3 Accuracy**
2018	5	7830	80.37±0.2	84.81±0.5	3525	71.4±1.2	77.0±0.3	2726	75.77±0.5	80.55±0.4	13983	75.87±0.2	81.05±0.3
	7	6376	85.41±0.5	89.49±0.4	2620	77.44±0.7	83.13±0.4	2083	81.98±0.6	85.78±0.7	11152	81.57±0.3	86.43±0.3
	9	5277	88.45±0.5	92.16±0.4	2054	81.72±0.9	86.1±0.6	1608	84.94±0.7	88.92±0.5	9053	85.25±0.3	89.19±0.4
2019	5	10122	81.54±0.2	85.5±0.3	6281	74.77±0.7	78.69±0.4	3921	75.08±0.7	79.37±0.4	19672	75.55±0.3	79.64±0.3
	7	8087	86.78±0.2	89.87±0.2	4840	80.34±0.6	84.11±0.4	3069	81.24±0.6	84.92±0.5	15973	81.51±0.3	84.98±0.2
	9	6468	89.65±0.3	92.51±0.2	3759	83.87±0.3	86.98±0.4	2373	85.2±0.5	87.96±0.6	12952	85.03±0.3	88.28±0.2
2020	5	16872	78.03±0.3	82.36±0.2	9124	62.82±0.2	68.23±0.4	3982	73.05±0.7	78.25±0.6	29838	70.89±0.2	75.72±0.2
	7	13694	83.73±0.3	87.59±0.3	7130	70.43±0.8	75.77±0.6	3125	79.53±0.7	84.17±0.8	24281	77.43±0.2	81.92±0.3
	9	11143	87.66±0.2	90.67±0.2	5513	75.53±0.4	80.12±0.7	2384	84.0±0.3	88.21±0.7	19529	82.05±0.2	85.68±0.2

Table 4.3. TH: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages using the original GCJ dataset.

		C++			Python			Java			All Languages		
Year	**Files**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3 Accuracy**
2018	5	7830	78.6±0.3	83.31±0.4	3525	70.22±0.9	76.35±0.7	2726	74.79±0.8	79.37±1.0	13983	74.4±0.3	79.49±0.2
	7	6376	83.92±0.5	87.95±0.3	2620	76.03±0.6	82.06±0.6	2083	80.49±0.9	84.94±0.6	11152	80.23±0.2	85.01±0.3
	9	5277	86.84±0.3	90.63±0.4	2054	80.24±0.6	85.63±0.8	1608	83.76±0.6	87.84±0.5	9053	83.88±0.3	87.89±0.2
2019	5	10122	80.25±0.3	84.04±0.3	6281	74.34±0.4	78.11±0.6	3921	74.11±0.6	77.87±0.5	19672	74.32±0.2	78.34±0.2
	7	8087	85.47±0.3	88.89±0.2	4840	80.19±0.4	83.44±0.5	3069	80.26±0.6	83.78±0.7	15973	80.44±0.4	84.26±0.2
	9	6468	88.78±0.2	91.54±0.1	3759	83.25±0.4	86.76±0.5	2373	84.13±0.8	87.05±0.7	12952	84.1±0.4	87.34±0.2
2020	5	16872	76.54±0.3	80.88±0.4	9124	61.84±0.3	67.29±0.2	3982	71.69±0.7	76.73±0.3	29838	69.52±0.2	74.39±0.1
	7	13694	82.4±0.3	86.45±0.3	7130	69.85±0.4	75.14±0.7	3125	78.4±0.7	83.03±0.4	24281	76.36±0.2	80.71±0.2
	9	11143	86.28±0.3	89.62±0.3	5513	75.18±0.5	79.43±0.5	2384	83.04±0.5	86.95±0.8	19529	81.05±0.2	84.75±0.2

3.6±7.9% compared to the rest of the years. Lastly, for Java, these embeddings achieve an accuracy of 71.33±0.7% in the year 2018 with an average difference of -4.76±3.41% compared to the rest of the years.

These embeddings achieve a top-3 accuracy of 80.82±0.3% in the mixed language scenario for the year 2018, which results in a difference of 6.34 compared to simple accuracy for the year 2018. We note an average 7.89±1.28% difference in the top-3 accuracy and simple accuracy across all years. For C++, these embeddings achieve a top-3 accuracy of 86.04±0.3% for the year 2018, resulting in a difference of 5.48 compared to simple accuracy for the year 2018. We note an average 7.02±1.28% difference in the top-3 accuracy and simple accuracy across all years. While for Python, these embeddings achieve a top-3 accuracy of 76.19±0.6% for the year 2018, resulting in a difference of 7.84 compared to simple accuracy for the year 2018. We note an average 8.67±1.45% difference in the top-3 accuracy and simple accuracy across all years. Lastly, for Java, these embeddings achieve a top-3 accuracy of 77.64±0.7% for the year 2018, resulting in a difference of 6.31 compared to

Table 4.4. TH-soft: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages using the original GCJ dataset.

		C++			Python			Java			All Languages		
Year	**Files**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3 Accuracy**
2018	5	7830	82.82±0.3	87.1±0.2	3525	76.14±0.6	82.13±0.5	2726	81.06±0.7	85.81±0.5	13983	79.57±0.2	84.37±0.2
	7	6376	87.6±0.5	91.47±0.3	2620	81.14±0.8	86.64±0.5	2083	86.37±0.4	90.39±0.6	11152	84.49±0.3	89.04±0.2
	9	5277	90.44±0.4	93.48±0.2	2054	84.92±0.7	89.25±0.6	1608	89.38±0.7	92.36±0.6	9053	87.89±0.2	91.46±0.3
2019	5	10122	83.07±0.2	86.64±0.3	6281	77.22±0.7	81.41±0.5	3921	79.55±0.3	83.18±0.5	19672	77.93±0.3	81.7±0.3
	7	8087	87.95±0.4	90.98±0.4	4840	82.77±0.3	86.64±0.6	3069	84.65±0.5	88.03±0.4	15973	83.37±0.2	86.96±0.2
	9	6468	90.73±0.3	93.19±0.3	3759	85.74±0.4	89.05±0.5	2373	88.08±0.6	90.8±0.6	12952	86.72±0.3	89.82±0.2
2020	5	16872	79.32±0.3	83.66±0.2	9124	64.93±0.4	70.67±0.3	3982	77.05±0.6	82.26±0.3	29838	72.75±0.2	77.52±0.2
	7	13694	84.9±0.3	88.57±0.2	7130	72.24±0.5	77.86±0.6	3125	82.96±0.2	87.55±0.4	24281	79.2±0.3	83.61±0.2
	9	11143	88.52±0.2	91.54±0.3	5513	77.51±0.4	82.1±0.3	2384	87.22±0.5	91.08±0.5	19529	83.61±0.2	87.25±0.2

simple accuracy for the year 2018. We note an average 8.98±1.94% difference in the top-3 accuracy and simple accuracy across all years.

For the mixed-language scenario, the accuracy difference with the original setting for the year 2018 is -10.31, while the average difference is -11.51±0.89%. For C++, We note that the difference for 2018 is -7.62, while the average difference between the original and obfuscated setting is -7.93±1.07%. Lastly, For Python, the difference for the year 2018 is -13.91, while the average difference with the obfuscated setting is -14.62±3.03%. For Java, the difference for the year 2018 is -13.46, while the average difference with the obfuscated setting is -17.22±2.03%

TSH based embeddings achieve an accuracy of 66.01±0.3% in the mixed language scenario for the year 2018. We see an average 2.13±4.63% difference in the accuracy of all years from the year 2018 in this particular scenario. For C++, these embeddings achieve an accuracy of 74.46±0.4% in the year 2018 with an average difference of -1.49±2.96% compared to the rest of the years. While for Python, they achieve an accuracy of 57.53±0.5% in the year 2018 with an average difference of 11.64±11.27% compared to the rest of the years. Lastly, for Java, these embeddings achieve an accuracy of 58.9±1.1% in the year 2018 with an average difference of 5.53±6.01% compared to the rest of the years.

These embeddings achieve a top-3 accuracy of 72.39±0.3% in the mixed language scenario for the year 2018, which results in a difference of 6.38 compared to simple accuracy for the year 2018. We note an average 7.38±0.73% difference in the top-3 accuracy and simple accuracy across all years. For C++, these embeddings achieve a top-3 accuracy of 79.8±0.3% for the year 2018, resulting in a difference of 5.34 compared to simple accuracy for the year 2018. We note an average 6.64±0.95% difference in the top-3 accuracy and simple accuracy across all years. While for Python,

these embeddings achieve a top-3 accuracy of 66.49±0.3% for the year 2018, resulting in a difference of 8.96 compared to simple accuracy for the year 2018. We note an average 9.1±1.17% difference in the top-3 accuracy and simple accuracy across all years. Lastly, for Java, these embeddings achieve a top-3 accuracy of 66.29±0.8% for the year 2018, resulting in a difference of 7.39 compared to simple accuracy for the year 2018. We note an average 8.16±0.95% difference in the top-3 accuracy and simple accuracy across all years.

For the mixed-language scenario, the accuracy difference with the original setting for the year 2018 is -9.86, while the average difference is -9.76±2.13%. For C++, We note that the difference for 2018 is -5.91, while the average difference between the original and obfuscated setting is -6.0±0.83%. Lastly, For Python, the difference for the year 2018 is -13.87, while the average difference with the obfuscated setting is -14.11±4.05%. For Java, the difference for the year 2018 is -16.87, while the average difference with the obfuscated setting is -16.09±2.34%.

TH based embeddings achieve an accuracy of 63.7±0.3% in the mixed language scenario for the year 2018. We see an average 2.51±4.77% difference in the accuracy of all years from the year 2018 in this particular scenario. For C++, these embeddings achieve an accuracy of 72.77±0.3% in the year 2018 with an average difference of -1.14±2.95% compared to the rest of the years. While for Python, they achieve an accuracy of 54.77±0.6% in the year 2018 with an average difference of 10.71±11.35% compared to the rest of the years. Lastly, for Java, these embeddings achieve an accuracy of 55.56±1.0% in the year 2018 with an average difference of 6.67±6.29% compared to the rest of the years.

These embeddings achieve a top-3 accuracy of 70.71±0.3% in the mixed language scenario for the year 2018, resulting in a difference of 7.01 compared to simple accuracy for the year 2018. We note an average 7.72±0.69% difference in the top-3 accuracy and simple accuracy across all years. For C++, these embeddings achieve a top-3 accuracy of 78.25±0.5% for the year 2018, resulting in a difference of 5.48 compared to simple accuracy for the year 2018. We note an average 6.74±0.98% difference in the top-3 accuracy and simple accuracy across all years. While for Python, these embeddings achieve a top-3 accuracy of 64.02±1.0% for the year 2018, resulting in a difference of 9.25 compared to simple accuracy for the year 2018. We note an average 10.08±1.08% difference in the top-3 accuracy and simple accuracy across all years. Lastly, for Java, these embeddings achieve a top-3 accuracy of 64.09±0.8% for the year 2018, resulting in a difference of 8.53 compared to

simple accuracy for the year 2018. We note an average 8.47±0.83% difference in the top-3 accuracy and simple accuracy across all years.

For the mixed-language scenario, the accuracy difference with the original setting for the year 2018 is -10.7, while the average difference is -10.28±2.34%. For C++, We note that the difference for 2018 is -5.83, while the average difference between the original and obfuscated setting is -5.97±0.74%. Lastly, For Python, the difference for the year 2018 is -15.45, while the average difference with the obfuscated setting is -16.38±4.43%. For Java, the difference for the year 2018 is -19.23, while the average difference with the obfuscated setting is -16.49±3.02%.

TH-soft based embeddings achieve an accuracy of 69.51±0.4% in the mixed language scenario for the year 2018. We see an average 0.76±4.26% difference in the accuracy of all years from the year 2018 in this particular scenario. For C++, these embeddings achieve an accuracy of 75.98±0.4% in the year 2018 with an average difference of -0.79±3.66% compared to the rest of the years. While for Python, they achieve an accuracy of 63.89±0.7% in the year 2018 with an average difference of 8.76±9.46% compared to the rest of the years. Lastly, for Java, these embeddings achieve an accuracy of 65.15±0.8% in the year 2018 with an average difference of -0.29±4.95% compared to the rest of the years.

These embeddings achieve a top-3 accuracy of 75.53±0.5% in the mixed language scenario for the year 2018, resulting in a difference of 6.02 compared to simple accuracy for the year 2018. We note an average 7.15±0.97% difference in the top-3 accuracy and simple accuracy across all years. For C++, these embeddings achieve a top-3 accuracy of 81.14±0.4% for the year 2018, resulting in a difference of 5.16 compared to simple accuracy for the year 2018. We note an average 6.38±1.01% difference in the top-3 accuracy and simple accuracy across all years. While for Python, these embeddings achieve a top-3 accuracy of 71.88±0.3% for the year 2018, resulting in a difference of 7.99 compared to simple accuracy for the year 2018. We note an average 8.39±1.35% difference in the top-3 accuracy and simple accuracy across all years. Lastly, for Java, these embeddings achieve a top-3 accuracy of 71.54±0.6% for the year 2018, which results in a difference of 6.39 compared to simple accuracy for the year 2018. We note an average 8.01±1.13% difference in the top-3 accuracy and simple accuracy across all years.

For the mixed-language scenario, the accuracy difference with the original setting for the year 2018 is -10.06, while the average difference is -9.75±1.25%. For C++, We note that the

difference for 2018 is -6.84, while the average difference between the original and obfuscated setting is -5.77±0.53%. Lastly, For Python, the difference for the year 2018 is -12.25, while the average difference with the obfuscated setting is -12.85±2.86%. For Java, the difference for the year 2018 is -15.91, while the average difference with the obfuscated setting is -17.76±2.32%.

4.2.2.2. Seven files per author

LS based embeddings achieve an accuracy of 80.04±0.5% in the mixed language scenario for the year 2018. We see an average -0.93±3.21% difference in the accuracy of all years from the year 2018 in this particular scenario. For C++, these embeddings achieve an accuracy of 85.49±0.6% in the year 2018 with an average difference of -1.79±2.94% compared to the rest of the years. While for Python, they achieve an accuracy of 74.63±0.6% in the year 2018 with an average difference of 5.57±7.61% compared to the rest of the years. Lastly, for Java, these embeddings achieve an accuracy of 76.37±0.8% in the year 2018 with an average difference of -1.5±2.9% compared to the rest of the years.

These embeddings achieve a top-3 accuracy of 85.78±0.1% in the mixed language scenario for the year 2018, which results in a difference of 5.74 compared to simple accuracy for the year 2018. We note an average 6.92±1.05% difference in the top-3 accuracy and simple accuracy across all years. For C++, these embeddings achieve a top-3 accuracy of 90.45±0.4% for the year 2018, resulting in a difference of 4.96 compared to simple accuracy for the year 2018. We note an average 6.17±1.26% difference in the top-3 accuracy and simple accuracy across all years. While for Python, these embeddings achieve a top-3 accuracy of 82.17±0.8% for the year 2018, resulting in a difference of 7.54 compared to simple accuracy for the year 2018. We note an average 7.52±1.14% difference in the top-3 accuracy and simple accuracy across all years. Lastly, for Java, these embeddings achieve a top-3 accuracy of 82.62±0.7% for the year 2018, which results in a difference of 6.25 compared to simple accuracy for the year 2018. We note an average 8.52±1.57% difference in the top-3 accuracy and simple accuracy across all years.

For the mixed-language scenario, the accuracy difference with the original setting for the year 2018 is -8.95, while the average difference is -9.55±0.88%. For C++, We note that the difference for 2018 is -6.37, while the average difference between the original and obfuscated setting is -6.55±0.92%. Lastly, For Python, the difference for the year 2018 is -11.86, while the average

difference with the obfuscated setting is -11.88±3.37%. For Java, the difference for the year 2018 is -12.83, while the average difference with the obfuscated setting is -14.78±1.75%.

TSH based embeddings achieve an accuracy of 71.99±0.4% in the mixed language scenario for the year 2018. We see an average of 3.65±4.69% difference in the accuracy of all years from the year 2018 in this particular scenario. For C++, these embeddings achieve an accuracy of 80.2±0.3% in the year 2018 with an average difference of -0.25±2.71% compared to the rest of the years. While for Python, they achieve an accuracy of 63.45±0.5% in the year 2018 with an average difference of 13.8±11.46% compared to the rest of the years. Lastly, for Java, these embeddings achieve an accuracy of 65.22±0.8% in the year 2018 with an average difference of 7.83±5.66% compared to the rest of the years.

These embeddings achieve a top-3 accuracy of 78.72±0.3% in the mixed language scenario for the year 2018, resulting in a difference of 6.73 compared to simple accuracy for the year 2018. We note an average 7.0±0.61% difference in the top-3 accuracy and simple accuracy across all years. For C++, these embeddings achieve a top-3 accuracy of 85.38±0.2% for the year 2018, resulting in a difference of 5.18 compared to simple accuracy for the year 2018. We note an average 6.16±0.91% difference in the top-3 accuracy and simple accuracy across all years. While for Python, these embeddings achieve a top-3 accuracy of 73.27±0.5% for the year 2018, resulting in a difference of 9.82 compared to simple accuracy for the year 2018. We note an average 7.88±1.33% difference in the top-3 accuracy and simple accuracy across all years. Lastly, for Java, these embeddings achieve a top-3 accuracy of 73.38±0.9% for the year 2018, resulting in a difference of 8.16 compared to simple accuracy for the year 2018. We note an average 8.01±0.75% difference in the top-3 accuracy and simple accuracy across all years.

For the mixed-language scenario, the accuracy difference with the original setting for the year 2018 is -9.58, while the average difference is -8.77±2.14%. For C++, We note that the difference for 2018 is -5.21, while the average difference between the original and obfuscated setting is -5.39±0.85%. Lastly, For Python, the difference for the year 2018 is -13.99, while the average difference with the obfuscated setting is -12.25±4.76%. For Java, the difference for the year 2018 is -16.76, while the average difference with the obfuscated setting is -14.09±2.57%.

TH based embeddings achieve an accuracy of 70.13±0.4% in the mixed language scenario for the year 2018. We see an average of 3.74±4.86% difference in the accuracy of all years from the year

2018 in this particular scenario. For C++, these embeddings achieve an accuracy of 78.81±0.4% in the year 2018 with an average difference of -0.14±2.71% compared to the rest of the years. While for Python, they achieve an accuracy of 60.56±0.7% in the year 2018 with an average difference of 13.55±11.92% compared to the rest of the years. Lastly, for Java, these embeddings achieve an accuracy of 61.87±0.6% in the year 2018 with an average difference of 8.9±6.11% compared to the rest of the years.

These embeddings achieve a top-3 accuracy of 76.86±0.1% in the mixed language scenario for the year 2018, which results in a difference of 6.73 compared to simple accuracy for the year 2018. We note an average 7.32±0.65% difference in the top-3 accuracy and simple accuracy across all years. For C++, these embeddings achieve a top-3 accuracy of 83.91±0.4% for the year 2018, resulting in a difference of 5.1 compared to simple accuracy for the year 2018. We note an average 6.38±1.06% difference in the top-3 accuracy and simple accuracy across all years. While for Python, these embeddings achieve a top-3 accuracy of 70.33±0.7% for the year 2018, resulting in a difference of 9.77 compared to simple accuracy for the year 2018. We note an average 9.05±1.21% difference in the top-3 accuracy and simple accuracy across all years. Lastly, for Java, these embeddings achieve a top-3 accuracy of 70.78±1.0% for the year 2018, resulting in a difference of 8.91 compared to simple accuracy for the year 2018. We note an average 8.36±0.98% difference in the top-3 accuracy and simple accuracy across all years.

For the mixed-language scenario, the accuracy difference with the original setting for 2018 is -10.1, while the average difference is -9.31±2.48%. For C++, We note that the difference for 2018 is -5.11, while the average difference between the original and obfuscated setting is -5.53±0.68%. Lastly, For Python, the difference for the year 2018 is -15.47, while the average difference with the obfuscated setting is -14.69±5.32%. For Java, the difference for the year 2018 is -18.62, while the average difference with the obfuscated setting is -14.95±2.99%.

TH-soft based embeddings achieve an accuracy of 75.95±0.2% in the mixed language scenario for the year 2018. We see an average 1.74±4.09% difference in the accuracy of all years from the year 2018 in this particular scenario. For C++, these embeddings achieve an accuracy of 81.55±0.4% in the year 2018, with an average difference of 0.39±3.35% compared to the rest of the years. While for Python, they achieve an accuracy of 69.52±0.8% in the year 2018 with an average difference of 11.0±9.01% compared to the rest of the years. Lastly, for Java, these embeddings

achieve an accuracy of 71.74±0.8% in the year 2018 with an average difference of 1.48±4.47% compared to the rest of the years.

These embeddings achieve a top-3 accuracy of 81.42±0.3% in the mixed language scenario for the year 2018, resulting in a difference of 5.47 compared to simple accuracy for the year 2018. We note an average 6.64±0.96% difference in the top-3 accuracy and simple accuracy across all years. For C++, these embeddings achieve a top-3 accuracy of 86.6±0.6% for the year 2018, resulting in a difference of 5.05 compared to simple accuracy for the year 2018. We note an average 5.89±0.98% difference in the top-3 accuracy and simple accuracy across all years. While for Python, these embeddings achieve a top-3 accuracy of 77.61±0.7% for the year 2018, resulting in a difference of 8.09 compared to simple accuracy for the year 2018. We note an average 7.15±0.99% difference in the top-3 accuracy and simple accuracy across all years. Lastly, for Java, these embeddings achieve a top-3 accuracy of 77.96±0.6% for the year 2018, resulting in a difference of 6.22 compared to simple accuracy for the year 2018. We note an average 8.15±1.22% difference in the top-3 accuracy and simple accuracy across all years.

For the mixed-language scenario, the accuracy difference with the original setting for 2018 is -8.54, while the average difference is -8.38±1.25%. For C++, We note that the difference for 2018 is -6.05, while the average difference between the original and obfuscated setting is -5.06±0.78%. Lastly, For Python, the difference for the year 2018 is -11.62, while the average difference with the obfuscated setting is -10.49±3.13%. For Java, the difference for the year 2018 is -14.63, while the average difference with the obfuscated setting is -15.34±2.37%.

We note a noticeable increase in performance with just two more files as priors, but one could also argue that this increase in performance is due to decreased author count.

4.2.2.3. Nine files per author

LS based embeddings achieve an accuracy of 83.54±0.4% in the mixed language scenario for the year 2018. We see an average 0.98±3.16% difference in the accuracy of all years from the year 2018 in this particular scenario. For C++, these embeddings achieve an accuracy of 88.32±0.3% in the year 2018 with an average difference of -0.05±2.43% compared to the rest of the years. While for Python, they achieve an accuracy of 78.51±0.6% in the year 2018 with an average difference of 6.98±7.36% compared to the rest of the years. Lastly, for Java, these embeddings achieve an

accuracy of 79.88±1.1% in the year 2018 with an average difference of 1.64±3.15% compared to the rest of the years.

These embeddings achieve a top-3 accuracy of 88.7±0.4% in the mixed language scenario for the year 2018, resulting in a difference of 5.16 compared to simple accuracy for the year 2018. We note an average 5.64±0.82% difference in the top-3 accuracy and simple accuracy across all years. For C++, these embeddings achieve a top-3 accuracy of 92.58±0.3% for the year 2018, resulting in a difference of 4.26 compared to simple accuracy for the year 2018. We note an average 4.82±1.14% difference in the top-3 accuracy and simple accuracy across all years. While for Python, these embeddings achieve a top-3 accuracy of 85.24±0.6% for the year 2018, resulting in a difference of 6.73 compared to simple accuracy for the year 2018. We note an average 5.81±1.49% difference in the top-3 accuracy and simple accuracy across all years. Lastly, for Java, these embeddings achieve a top-3 accuracy of 85.96±0.7% for the year 2018, resulting in a difference of 6.08 compared to simple accuracy for the year 2018. We note an average 7.56±1.9% difference in the top-3 accuracy and simple accuracy across all years.

For the mixed-language scenario, the accuracy difference with the original setting for the year 2018 is -8.05, while the average difference is -7.72±1.0%. For C++, We note that the difference for 2018 is -5.57, while the average difference between the original and obfuscated setting is -5.15±0.87%. Lastly, For Python, the difference for the year 2018 is -11.34, while the average difference with the obfuscated setting is -9.56±3.34%. For Java, the difference for the year 2018 is -12.04, while the average difference with the obfuscated setting is -11.68±1.76%.

TSH based embeddings achieve an accuracy of 76.48±0.4% in the mixed language scenario for the year 2018. We see an average of 4.77±4.78% difference in the accuracy of all years from the year 2018 in this particular scenario. For C++, these embeddings achieve an accuracy of 83.49±0.6% in the year 2018, with an average difference of 1.66±2.43% compared to the rest of the years. While for Python, they achieve an accuracy of 67.94±0.9% in the year 2018 with an average difference of 14.44±11.21% compared to the rest of the years. Lastly, for Java, these embeddings achieve an accuracy of 70.26±1.0% in the year 2018 with an average difference of 9.43±6.03% compared to the rest of the years.

These embeddings achieve a top-3 accuracy of 82.46±0.4% in the mixed language scenario for the year 2018, resulting in a difference of 5.98 compared to simple accuracy for the year 2018.

We note an average 6.0±0.62% difference in the top-3 accuracy and simple accuracy across all years. For C++, these embeddings achieve a top-3 accuracy of 88.07±0.3% for the year 2018, resulting in a difference of 4.58 compared to simple accuracy for the year 2018. We note an average 5.09±0.71% difference in the top-3 accuracy and simple accuracy across all years. While for Python, these embeddings achieve a top-3 accuracy of 77.24±0.7% for the year 2018, resulting in a difference of 9.3 compared to simple accuracy for the year 2018. We note an average 6.71±1.79% difference in the top-3 accuracy and simple accuracy across all years. Lastly, for Java, these embeddings achieve a top-3 accuracy of 77.27±1.0% for the year 2018, resulting in a difference of 7.01 compared to simple accuracy for the year 2018. We note an average 6.97±1.03% difference in the top-3 accuracy and simple accuracy across all years.

For the mixed-language scenario, the accuracy difference with the original setting for the year 2018 is -8.77, while the average difference is -7.49±2.21%. For C++, We note that the difference for 2018 is -4.96, while the average difference between the original and obfuscated setting is -4.62±0.61%. Lastly, For Python, the difference for the year 2018 is -13.78, while the average difference with the obfuscated setting is -10.7±5.19%. For Java, the difference for the year 2018 is -14.68, while the average difference with the obfuscated setting is -11.66±2.91%.

TH based embeddings achieve an accuracy of 74.65±0.5% in the mixed language scenario for the year 2018. We see an average 5.12±5.05% difference in the accuracy of all years from 2018 in this particular scenario. For C++, these embeddings achieve an accuracy of 82.17±0.4% in the year 2018, with an average difference of 1.84±2.42% compared to the rest of the years. While for Python, they achieve an accuracy of 65.94±0.9% in the year 2018 with an average difference of 13.97±11.98% compared to the rest of the years. Lastly, for Java, these embeddings achieve an accuracy of 67.4±1.0% in the year 2018 with an average difference of 10.65±6.7% compared to the rest of the years.

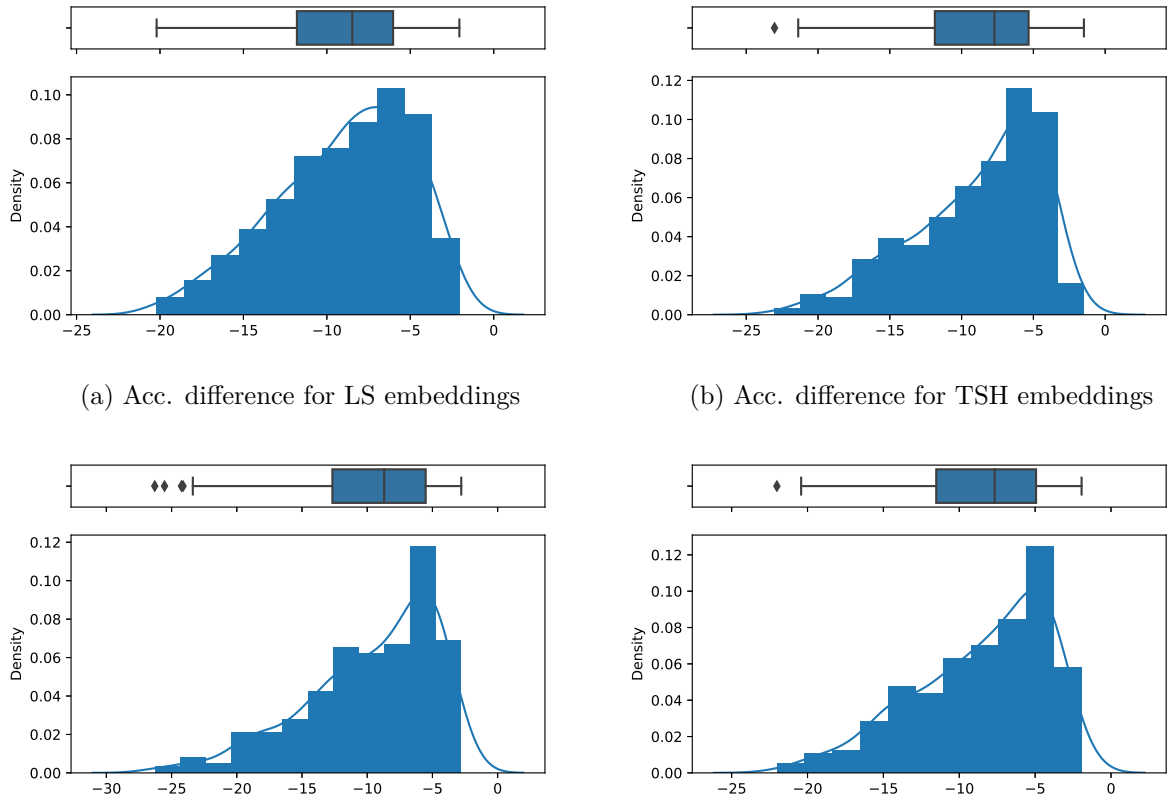
These embeddings achieve a top-3 accuracy of 80.91±0.4% in the mixed language scenario for the year 2018, which results in a difference of 6.26 compared to simple accuracy for the year 2018. We note an average 6.36±0.71% difference in the top-3 accuracy and simple accuracy across all years. For C++, these embeddings achieve a top-3 accuracy of 86.88±0.5% for the year 2018, resulting in a difference of 4.71 compared to simple accuracy for the year 2018. We note an average 5.53±1.05% difference in the top-3 accuracy and simple accuracy across all years. While for Python,

these embeddings achieve a top-3 accuracy of 74.54±0.7% for the year 2018, resulting in a difference of 8.6 compared to simple accuracy for the year 2018. We note an average 7.45±1.38% difference in the top-3 accuracy and simple accuracy across all years. Lastly, for Java, these embeddings achieve a top-3 accuracy of 74.7±1.0% for the year 2018, resulting in a difference of 7.3 compared to simple accuracy for the year 2018. We note an average 7.26±0.84% difference in the top-3 accuracy and simple accuracy across all years.

For the mixed-language scenario, the accuracy difference with the original setting for the year 2018 is -9.23, while the average difference is -8.07±2.51%. For C++, We note that the difference for 2018 is -4.67, while the average difference between the original and obfuscated setting is -4.8±0.62%. Lastly, For Python, the difference for the year 2018 is -14.3, while the average difference with the obfuscated setting is -12.63±5.81%. For Java, the difference for the year 2018 is -16.36, while the average difference with the obfuscated setting is -12.11±3.74%.

TH-soft based embeddings achieve an accuracy of 79.36±0.3% in the mixed language scenario for the year 2018. We see an average of 3.98±4.01% difference in the accuracy of all years from the year 2018 in this particular scenario. For C++, these embeddings achieve an accuracy of 84.96±0.4% in the year 2018, with an average difference of 1.92±2.7% compared to the rest of the years. While for Python, they achieve an accuracy of 74.33±0.7% in the year 2018 with an average difference of 11.27±8.73% compared to the rest of the years. Lastly, for Java, these embeddings achieve an accuracy of 76.0±0.8% in the year 2018 with an average difference of 4.14±4.61% compared to the rest of the years.

These embeddings achieve a top-3 accuracy of 84.78±0.3% in the mixed language scenario for the year 2018, resulting in a difference of 5.42 compared to simple accuracy for the year 2018. We note an average 5.4±0.65% difference in the top-3 accuracy and simple accuracy across all years. For C++, these embeddings achieve a top-3 accuracy of 89.2±0.4% for the year 2018, resulting in a difference of 4.24 compared to simple accuracy for the year 2018. We note an average 4.65±0.79% difference in the top-3 accuracy and simple accuracy across all years. While for Python, these embeddings achieve a top-3 accuracy of 81.41±0.9% for the year 2018, resulting in a difference of 7.08 compared to simple accuracy for the year 2018. We note an average 5.26±1.62% difference in the top-3 accuracy and simple accuracy across all years. Lastly, for Java, these embeddings achieve a top-3 accuracy of 81.02±0.7% for the year 2018, resulting in a difference of 5.02 compared to



(a) Acc. difference for LS embeddings

(b) Acc. difference for TSH embeddings

(c) Acc. difference for TH embeddings

(d) Acc. difference for TH-soft embeddings

Figure 4.6. Distributions of accuracy loss when moving from original to obfuscated setting. Here the y-axis shows the density while the x-axis shows the accuracy difference. The box plot on top of the distributions shows an alternative visualization of the same distribution and shares the x-axis with the same distribution.

simple accuracy for the year 2018. We note an average 6.68±1.19% difference in the top-3 accuracy and simple accuracy across all years.

For the mixed-language scenario, the accuracy difference with the original setting for the year 2018 is -8.53, while the average difference is -6.72±1.51%. For C++, We note that the difference for 2018 is -5.48, while the average difference between the original and obfuscated setting is -4.0±0.54%. Lastly, For Python, the difference for the year 2018 is -10.59, while the average difference with the obfuscated setting is -8.31±3.49%. For Java, the difference for the year 2018 is -13.38, while the average difference with the obfuscated setting is -12.3±2.63%.

We note a noticeable increase in performance yet again with the increase in the number of files as priors. However, again, one could argue that this increase in performance is due to

the decrease in author count and not due to the increase in priors. In figure 4.7 we compare the performance of the different loss functions in different scenarios under this particular scenario for the year 2020. As we can LS based embeddings always outperforms the TSH, TH, and its Soft margin variant based embeddings. In figure 4.6, we show different distributions of the differences between the accuracy of the original and the obfuscated setting across different loss functions.

According to the LS distribution, The mean difference between the two settings is -9.03±4.03%, with the median value being -8.47. According to the TSH distribution, The mean difference between the two settings is -8.89±4.44%, with the median value being -7.71. According to the TH distribution, The mean difference between the two settings is -9.68±5.00%, with the median value being -8.70. According to the TH-Soft distribution, The mean difference between the two settings is -8.55±4.36%, with the median value being -7.67. According to this analysis, TH-Soft yields the least difference on average and median, While LS-based embeddings have a lower STD score, which points to better stability, as can be seen by the box plots.

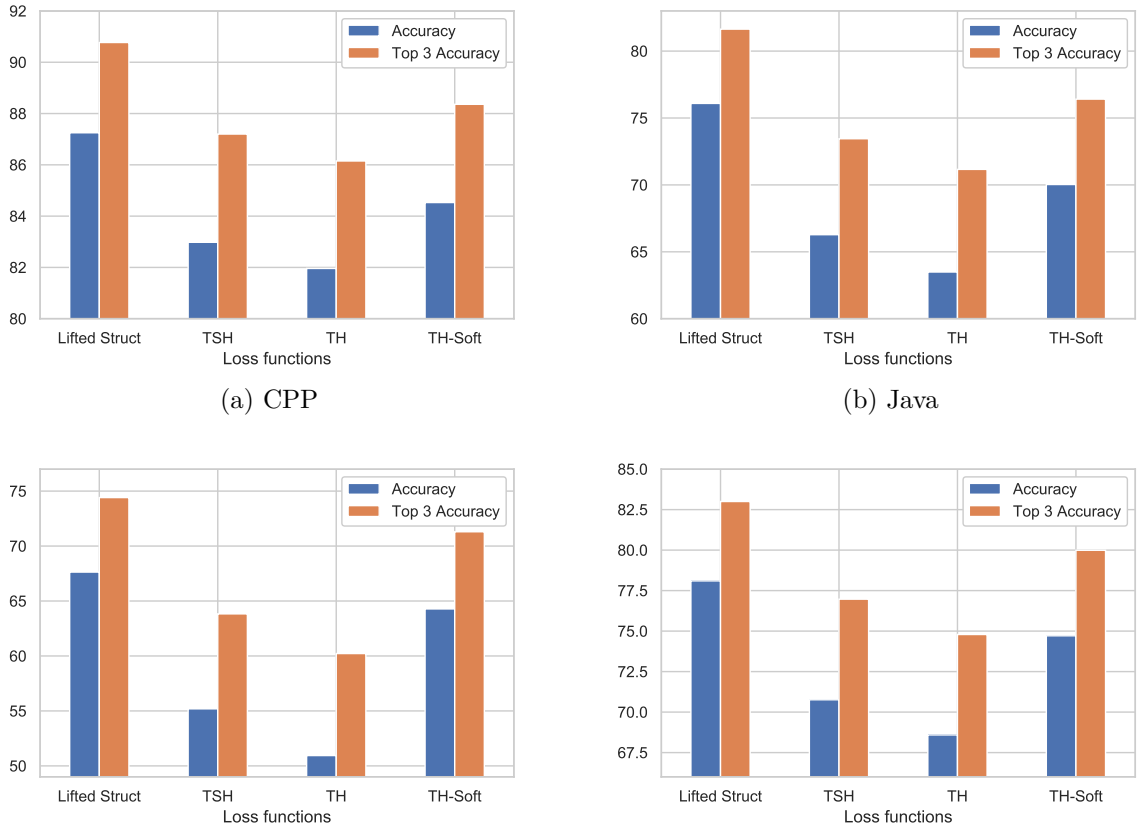
If we combine these distributions to form a new distribution, we can perform some further analysis. The mean difference between the two settings, regardless of the loss function, is -9.04±4.49%, with the median value being -8.07 according to the new distribution. This shows us how much degradation we can expect if the approaches are evaluated solely using obfuscated source code files.

Table 4.5. LS: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages under obfuscated setting.

		C++			Python			Java			All Languages		
Year	**Files**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3 Accuracy**
2018	5	7857	80.56±0.4	86.04±0.3	3641	68.35±0.8	76.19±0.6	2720	71.33±0.7	77.64±0.7	14093	74.48±0.3	80.82±0.3
	7	6399	85.49±0.6	90.45±0.4	2737	74.63±0.6	82.17±0.8	2080	76.37±0.8	82.62±0.7	11278	80.04±0.5	85.78±0.1
	9	5293	88.32±0.3	92.58±0.3	2155	78.51±0.6	85.24±0.6	1602	79.88±1.1	85.96±0.7	9164	83.54±0.4	88.7±0.4
2019	5	10581	80.34±0.4	84.97±0.3	7771	63.77±0.6	70.11±0.3	3912	69.3±0.8	74.44±0.7	21554	69.48±0.2	74.88±0.3
	7	8464	85.47±0.4	89.48±0.2	5996	70.52±0.5	76.92±0.4	3059	75.75±0.7	80.81±0.4	17490	76.19±0.3	81.29±0.2
	9	6783	88.99±0.4	91.99±0.1	4658	75.27±0.5	80.8±0.3	2372	80.44±0.7	84.0±0.6	14168	80.54±0.4	84.87±0.3
2020	5	16925	77.48±0.3	82.65±0.2	9471	53.87±0.3	61.6±0.4	3974	63.63±0.4	69.77±0.7	30153	66.58±0.2	72.33±0.3
	7	13743	83.39±0.1	87.8±0.2	7457	61.94±0.5	69.1±0.6	3112	70.81±0.6	77.07±0.8	24644	73.41±0.2	78.94±0.2
	9	11220	87.25±0.3	90.77±0.2	5827	67.63±0.7	74.41±0.4	2377	76.09±0.7	81.64±0.5	19905	78.1±0.3	83.0±0.3

4.2.3. Performance on simulated real-world dataset

In this subsection, we compare the results across different loss functions under the simulated real-world constraint.



(a) CPP

(b) Java

(c) Python

(d) Mixed

Figure 4.7. Loss function performance comparison on obfuscated source code

4.2.3.1. Five files per author

LS based embeddings achieve an accuracy of 76.78±0.4% in the mixed language scenario for the year 2018. We see an average -2.29±3.65% difference in the accuracy of all years from the year 2018 in this particular scenario. For C++, these embeddings achieve an accuracy of 82.75±0.4% in the year 2018 with an average difference of -2.42±2.48% compared to the rest of the years. While for Python, they achieve an accuracy of 74.76±0.5% in the year 2018 with an average difference of 2.17±7.99% compared to the rest of the years. Lastly, for Java, these embeddings achieve an accuracy of 68.29±0.6% in the year 2018 with an average difference of -2.96±3.58% compared to the rest of the years.

These embeddings achieve a top-3 accuracy of 84.06±0.3% in the mixed language scenario for the year 2018, resulting in a difference of 7.28 compared to simple accuracy for the year 2018. We note an average 8.24±0.76% difference in the top-3 accuracy and simple accuracy across all

Table 4.6. TSH: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages under obfuscated setting.

		C++			Python			Java			All Languages		
Year	**Files**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3 Accuracy**
2018	5	7857	74.46±0.4	79.8±0.3	3641	57.53±0.5	66.49±0.3	2720	58.9±1.1	66.29±0.8	14093	66.01±0.3	72.39±0.3
	7	6399	80.2±0.3	85.38±0.2	2737	63.45±0.5	73.27±0.5	2080	65.22±0.8	73.38±0.9	11278	71.99±0.4	78.72±0.3
	9	5293	83.49±0.6	88.07±0.3	2155	67.94±0.9	77.24±0.7	1602	70.26±1.0	77.27±1.0	9164	76.48±0.4	82.46±0.4
2019	5	10581	74.9±0.4	79.72±0.3	7771	51.74±0.5	60.16±0.4	3912	57.63±0.6	64.21±0.5	21554	60.87±0.3	67.17±0.3
	7	8464	81.06±0.4	85.37±0.4	5996	58.96±0.6	67.33±0.4	3059	65.2±0.8	71.64±0.7	17490	67.89±0.3	74.08±0.2
	9	6783	85.06±0.2	88.83±0.2	4658	63.94±0.7	71.51±0.5	2372	69.97±0.5	75.4±0.7	14168	72.85±0.2	78.16±0.2
2020	5	16925	72.59±0.2	77.71±0.3	9471	42.84±0.4	51.4±0.5	3974	52.44±0.8	59.96±0.4	30153	58.95±0.2	65.2±0.3
	7	13743	78.63±0.2	83.57±0.3	7457	49.51±0.4	58.8±0.5	3112	60.21±1.2	67.75±0.5	24644	65.57±0.3	72.24±0.3
	9	11220	82.98±0.3	87.2±0.3	5827	55.19±0.5	63.83±0.6	2377	66.28±0.8	73.45±0.6	19905	70.76±0.2	76.97±0.2

Table 4.7. TH: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages under obfuscated setting.

		C++			Python			Java			All Languages		
Year	**Files**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3 Accuracy**
2018	5	7857	72.77±0.3	78.25±0.5	3641	54.77±0.6	64.02±1.0	2720	55.56±1.0	64.09±0.8	14093	63.7±0.3	70.71±0.3
	7	6399	78.81±0.4	83.91±0.4	2737	60.56±0.7	70.33±0.7	2080	61.87±0.6	70.78±1.0	11278	70.13±0.4	76.86±0.1
	9	5293	82.17±0.4	86.88±0.5	2155	65.94±0.9	74.54±0.7	1602	67.4±1.0	74.7±1.0	9164	74.65±0.5	80.91±0.4
2019	5	10581	73.6±0.4	78.7±0.3	7771	48.05±0.4	56.37±0.6	3912	54.78±0.6	62.21±0.6	21554	58.54±0.3	65.07±0.3
	7	8464	79.81±0.3	84.47±0.2	5996	54.65±0.4	63.92±0.4	3059	61.51±0.6	69.27±0.5	17490	65.76±0.3	72.08±0.3
	9	6783	84.14±0.6	87.99±0.4	4658	59.88±0.7	68.59±0.6	2372	66.45±1.0	73.39±0.8	14168	70.5±0.4	76.26±0.3
2020	5	16925	71.05±0.3	76.19±0.3	9471	39.12±0.4	47.79±0.6	3974	49.26±0.6	57.32±0.9	30153	56.66±0.2	63.05±0.2
	7	13743	77.64±0.4	82.44±0.2	7457	45.73±0.5	55.11±0.5	3112	57.33±0.5	65.58±1.0	24644	63.34±0.2	70.07±0.2
	9	11220	81.96±0.2	86.15±0.3	5827	50.94±0.5	60.22±0.5	2377	63.49±0.9	71.16±0.9	19905	68.59±0.4	74.79±0.2

years. For C++, these embeddings achieve a top-3 accuracy of 89.03±0.3% for the year 2018, which results in a difference of 6.28 compared to simple accuracy for the year 2018. We note an average 7.17±0.78% difference in the top-3 accuracy and simple accuracy across all years. While for Python, these embeddings achieve a top-3 accuracy of 82.94±0.4% for the year 2018, resulting in a difference of 8.18 compared to simple accuracy for the year 2018. We note an average 8.26±1.04% difference in the top-3 accuracy and simple accuracy across all years. Lastly, for Java, these embeddings achieve a top-3 accuracy of 77.17±0.6% for the year 2018, resulting in a difference of 8.88 compared to simple accuracy for the year 2018. We note an average 10.54±1.61% difference in the top-3 accuracy and simple accuracy across all years.

TSH based embeddings achieve an accuracy of 64.49±0.3% in the mixed language scenario for the year 2018. We see an average 2.69±4.61% difference in the accuracy of all years from the year 2018 in this particular scenario. For C++, these embeddings achieve an accuracy of 72.09±0.3% in the year 2018, with an average difference of 0.01±2.49% compared to the rest of the years. While for Python, they achieve an accuracy of 61.08±0.5% in the year 2018 with an average difference

Table 4.8. TH-soft: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages under obfuscated setting.

		C++			Python			Java			All Languages		
Year	**Files**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3 Accuracy**
2018	5	7857	75.98±0.4	81.14±0.4	3641	63.89±0.7	71.88±0.3	2720	65.15±0.8	71.54±0.6	14093	69.51±0.4	75.53±0.5
	7	6399	81.55±0.4	86.6±0.6	2737	69.52±0.8	77.61±0.7	2080	71.74±0.8	77.96±0.6	11278	75.95±0.2	81.42±0.3
	9	5293	84.96±0.4	89.2±0.4	2155	74.33±0.7	81.41±0.9	1602	76.0±0.8	81.02±0.7	9164	79.36±0.3	84.78±0.3
2019	5	10581	76.68±0.3	81.29±0.3	7771	60.4±0.3	67.0±0.4	3912	63.49±0.6	69.46±0.5	21554	65.89±0.4	71.07±0.3
	7	8464	82.45±0.3	86.71±0.3	5996	67.58±0.6	73.86±0.5	3059	70.55±0.7	76.21±0.5	17490	72.69±0.3	77.93±0.3
	9	6783	86.53±0.3	89.57±0.3	4658	72.07±0.7	77.71±0.5	2372	75.47±0.6	79.89±1.1	14168	77.33±0.4	81.81±0.3
2020	5	16925	74.08±0.4	78.9±0.3	9471	50.95±0.4	58.69±0.4	3974	56.63±0.6	63.25±0.5	30153	62.88±0.2	68.83±0.1
	7	13743	80.25±0.3	85.08±0.2	7457	59.04±0.5	66.2±0.4	3112	64.11±0.7	71.53±0.5	24644	69.87±0.2	75.57±0.2
	9	11220	84.53±0.3	88.36±0.3	5827	64.28±0.5	71.3±0.5	2377	70.03±0.6	76.41±0.6	19905	74.71±0.2	79.99±0.3

of 11.94±11.19% compared to the rest of the years. Lastly, for Java, these embeddings achieve an accuracy of 53.71±0.5% in the year 2018 with an average difference of 4.07±5.42% compared to the rest of the years.

These embeddings achieve a top-3 accuracy of 72.56±0.2% in the mixed language scenario for the year 2018, resulting in a difference of 8.07 compared to simple accuracy for the year 2018. We note an average 8.68±0.8% difference in the top-3 accuracy and simple accuracy across all years. For C++, these embeddings achieve a top-3 accuracy of 79.35±0.4% for the year 2018, resulting in a difference of 7.26 compared to simple accuracy for the year 2018. We note an average 8.2±0.99% difference in the top-3 accuracy and simple accuracy across all years. While for Python, these embeddings achieve a top-3 accuracy of 71.03±0.4% for the year 2018, resulting in a difference of 9.95 compared to simple accuracy for the year 2018. We note an average 8.86±1.15% difference in the top-3 accuracy and simple accuracy across all years. Lastly, for Java, these embeddings achieve a top-3 accuracy of 61.23±0.7% for the year 2018, resulting in a difference of 7.52 compared to simple accuracy for the year 2018. We note an average 9.56±1.48% difference in the top-3 accuracy and simple accuracy across all years.

TH based embeddings achieve an accuracy of 62.52±0.2% in the mixed language scenario for the year 2018. We see an average 2.7±4.72% difference in the accuracy of all years from the year 2018 in this particular scenario. For C++, these embeddings achieve an accuracy of 70.77±0.4% in the year 2018, with an average difference of 0.22±2.46% compared to the rest of the years. While for Python, they achieve an accuracy of 58.67±0.4% in the year 2018 with an average difference of 11.18±11.46% compared to the rest of the years. Lastly, for Java, these embeddings achieve an

accuracy of 50.84±0.6% in the year 2018 with an average difference of 4.28±5.36% compared to the rest of the years.

These embeddings achieve a top-3 accuracy of 70.52±0.3% in the mixed language scenario for the year 2018, resulting in a difference of 8.0 compared to simple accuracy for the year 2018. We note an average 8.95±0.93% difference in the top-3 accuracy and simple accuracy across all years. For C++, these embeddings achieve a top-3 accuracy of 77.66±0.4% for the year 2018, resulting in a difference of 6.89 compared to simple accuracy for the year 2018. We note an average 8.3±0.97% difference in the top-3 accuracy and simple accuracy across all years. While for Python, these embeddings achieve a top-3 accuracy of 68.42±0.5% for the year 2018, resulting in a difference of 9.75 compared to simple accuracy for the year 2018. We note an average 9.46±1.18% difference in the top-3 accuracy and simple accuracy across all years. Lastly, for Java, these embeddings achieve a top-3 accuracy of 59.3±0.6% for the year 2018, resulting in a difference of 8.46 compared to simple accuracy for the year 2018. We note an average 9.76±1.42% difference in the top-3 accuracy and simple accuracy across all years.

TH-soft based embeddings achieve an accuracy of 70.14±0.2% in the mixed language scenario for the year 2018. We see an average 1.21±4.23% difference in the accuracy of all years from 2018 in this particular scenario. For C++, these embeddings achieve an accuracy of 75.66±0.3% in the year 2018 with an average difference of 0.09±2.85% compared to the rest of the years. While for Python, they achieve an accuracy of 70.12±0.3% in the year 2018 with an average difference of 8.04±9.33% compared to the rest of the years. Lastly, for Java, these embeddings achieve an accuracy of 61.36±0.8% in the year 2018 with an average difference of 1.12±4.75% when compared to the rest of the years.

These embeddings achieve a top-3 accuracy of 77.7±0.2% in the mixed language scenario for the year 2018, resulting in a difference of 7.56 compared to simple accuracy for the year 2018. We note an average 8.3±0.78% difference in the top-3 accuracy and simple accuracy across all years. For C++, these embeddings achieve a top-3 accuracy of 82.43±0.4% for the year 2018, which results in a difference of 6.77 when compared to simple accuracy for the year 2018. We note an average 7.74±0.82% difference in the top-3 accuracy and simple accuracy across all years. While for Python, these embeddings achieve a top-3 accuracy of 78.4±0.4% for the year 2018, resulting in a difference of 8.28 compared to simple accuracy for the year 2018. We note an average

7.42±1.09% difference in the top-3 accuracy and simple accuracy across all years. Lastly, for Java, these embeddings achieve a top-3 accuracy of 69.71±0.7% for the year 2018, resulting in a difference of 8.35 compared to simple accuracy for the year 2018. We note an average 9.94±1.34% difference in the top-3 accuracy and simple accuracy across all years.

4.2.3.2. Seven files per author

LS based embeddings achieve an accuracy of 82.24±0.3% in the mixed language scenario for the year 2018. We see an average -1.33±3.28% difference in the accuracy of all years from the year 2018 in this particular scenario. For C++, these embeddings achieve an accuracy of 87.5±0.4% in the year 2018 with an average difference of -1.79±1.93% compared to the rest of the years. While for Python, they achieve an accuracy of 80.54±0.5% in the year 2018 with an average difference of 2.56±7.34% compared to the rest of the years. Lastly, for Java, these embeddings achieve an accuracy of 75.49±0.8% in the year 2018 with an average difference of -1.9±3.39% compared to the rest of the years.

These embeddings achieve a top-3 accuracy of 88.49±0.2% in the mixed language scenario for the year 2018, resulting in a difference of 6.25 compared to simple accuracy for the year 2018. We note an average 7.19±0.6% difference in the top-3 accuracy and simple accuracy across all years. For C++, these embeddings achieve a top-3 accuracy of 92.52±0.2% for the year 2018, resulting in a difference of 5.02 compared to simple accuracy for the year 2018. We note an average 6.11±0.63% difference in the top-3 accuracy and simple accuracy across all years. While for Python, these embeddings achieve a top-3 accuracy of 87.35±0.4% for the year 2018, resulting in a difference of 6.81 compared to simple accuracy for the year 2018. We note an average 6.74±1.24% difference in the top-3 accuracy and simple accuracy across all years. Lastly, for Java, these embeddings achieve a top-3 accuracy of 83.44±0.7% for the year 2018, resulting in a difference of 7.95 compared to simple accuracy for the year 2018. We note an average 9.46±1.38% difference in the top-3 accuracy and simple accuracy across all years.

TSH based embeddings achieve an accuracy of 71.06±0.2% in the mixed language scenario for the year 2018. We see an average 3.17±4.45% difference in the accuracy of all years from the year 2018 in this particular scenario. For C++, these embeddings achieve an accuracy of 78.34±0.3% in the year 2018 with an average difference of -0.08±2.14% compared to the rest of the years. While for Python, they achieve an accuracy of 67.74±0.5% in the year 2018 with an average difference

of 11.36±10.56% compared to the rest of the years. Lastly, for Java, these embeddings achieve an accuracy of 61.45±0.6% in the year 2018 with an average difference of 5.6±5.49% compared to the rest of the years.

These embeddings achieve a top-3 accuracy of 78.61±0.2% in the mixed language scenario for the year 2018, resulting in a difference of 7.55 compared to simple accuracy for the year 2018. We note an average 8.04±0.58% difference in the top-3 accuracy and simple accuracy across all years. For C++, these embeddings achieve a top-3 accuracy of 84.46±0.3% for the year 2018, resulting in a difference of 6.12 compared to simple accuracy for the year 2018. We note an average 7.48±0.88% difference in the top-3 accuracy and simple accuracy across all years. While for Python, these embeddings achieve a top-3 accuracy of 76.93±0.3% for the year 2018, resulting in a difference of 9.19 compared to simple accuracy for the year 2018. We note an average 7.68±1.37% difference in the top-3 accuracy and simple accuracy across all years. Lastly, for Java, these embeddings achieve a top-3 accuracy of 69.49±0.9% for the year 2018, resulting in a difference of 8.04 compared to simple accuracy for the year 2018. We note an average 9.19±1.39% difference in the top-3 accuracy and simple accuracy across all years.

TH based embeddings achieve an accuracy of 69.06±0.3% in the mixed language scenario for the year 2018. We see an average 3.38±4.65% difference in the accuracy of all years from the year 2018 in this particular scenario. For C++, these embeddings achieve an accuracy of 76.77±0.4% in the year 2018, with an average difference of 0.48±2.09% compared to the rest of the years. While for Python, they achieve an accuracy of 65.0±0.6% in the year 2018 with an average difference of 11.09±10.92% compared to the rest of the years. Lastly, for Java, these embeddings achieve an accuracy of 59.51±0.6% in the year 2018 with an average difference of 4.96±5.43% compared to the rest of the years.

These embeddings achieve a top-3 accuracy of 76.92±0.3% in the mixed language scenario for the year 2018, resulting in a difference of 7.86 compared to simple accuracy for the year 2018. We note an average 8.33±0.67% difference in the top-3 accuracy and simple accuracy across all years. For C++, these embeddings achieve a top-3 accuracy of 83.31±0.4% for the year 2018, resulting in a difference of 6.54 compared to simple accuracy for the year 2018. We note an average 7.7±0.91% difference in the top-3 accuracy and simple accuracy across all years. While for Python, these embeddings achieve a top-3 accuracy of 75.03±0.5% for the year 2018, resulting in a difference of

10.03 compared to simple accuracy for the year 2018. We note an average 8.52±1.2% difference in the top-3 accuracy and simple accuracy across all years. Lastly, for Java, these embeddings achieve a top-3 accuracy of 67.13±0.7% for the year 2018, resulting in a difference of 7.62 compared to simple accuracy for the year 2018. We note an average 9.39±1.4% difference in the top-3 accuracy and simple accuracy across all years.

TH-soft based embeddings achieve an accuracy of 76.45±0.2% in the mixed language scenario for the year 2018. We see an average 1.58±3.88% difference in the accuracy of all years from the year 2018 in this particular scenario. For C++, these embeddings achieve an accuracy of 81.33±0.4% in the year 2018, with an average difference of 0.35±2.43% compared to the rest of the years. While for Python, they achieve an accuracy of 75.96±0.5% in the year 2018 with an average difference of 7.62±8.31% compared to the rest of the years. Lastly, for Java, these embeddings achieve an accuracy of 69.11±0.7% in the year 2018 with an average difference of 1.77±4.49% compared to the rest of the years.

These embeddings achieve a top-3 accuracy of 83.28±0.3% in the mixed language scenario for the year 2018, resulting in a difference of 6.83 compared to simple accuracy for the year 2018. We note an average 7.5±0.56% difference in the top-3 accuracy and simple accuracy across all years. For C++, these embeddings achieve a top-3 accuracy of 87.37±0.3% for the year 2018, resulting in a difference of 6.04 compared to simple accuracy for the year 2018. We note an average 6.93±0.76% difference in the top-3 accuracy and simple accuracy across all years. While for Python, these embeddings achieve a top-3 accuracy of 83.25±0.5% for the year 2018, resulting in a difference of 7.29 compared to simple accuracy for the year 2018. We note an average 6.3±1.13% difference in the top-3 accuracy and simple accuracy across all years. Lastly, for Java, these embeddings achieve a top-3 accuracy of 76.53±0.6% for the year 2018, resulting in a difference of 7.42 compared to simple accuracy for the year 2018. We note an average 9.24±1.17% difference in the top-3 accuracy and simple accuracy across all years.

4.2.3.3. Nine files per author

LS based embeddings achieve an accuracy of 85.56±0.3% in the mixed language scenario for the year 2018. We see an average -0.56±3.07% difference in the accuracy of all years from the year 2018 in this particular scenario. For C++, these embeddings achieve an accuracy of 89.9±0.2% in the year 2018 with an average difference of -0.87±1.65% compared to the rest of the years. While

for Python, they achieve an accuracy of 83.41±0.3% in the year 2018 with an average difference of 3.14±6.45% compared to the rest of the years. Lastly, for Java, these embeddings achieve an accuracy of 79.56±0.6% in the year 2018 with an average difference of -0.7±3.05% compared to the rest of the years.

These embeddings achieve a top-3 accuracy of 90.64±0.2% in the mixed language scenario for the year 2018, resulting in a difference of 5.08 compared to simple accuracy for the year 2018. We note an average 5.7±0.45% difference in the top-3 accuracy and simple accuracy across all years. For C++, these embeddings achieve a top-3 accuracy of 94.06±0.2% for the year 2018, resulting in a difference of 4.16 compared to simple accuracy for the year 2018. We note an average 4.74±0.53% difference in the top-3 accuracy and simple accuracy across all years. While for Python, these embeddings achieve a top-3 accuracy of 89.68±0.4% for the year 2018, which results in a difference of 6.27 compared to simple accuracy for the year 2018. We note an average 5.45±1.2% difference in the top-3 accuracy and simple accuracy across all years. Lastly, for Java, these embeddings achieve a top-3 accuracy of 86.39±0.5% for the year 2018, resulting in a difference of 6.83 compared to simple accuracy for the year 2018. We note an average 8.0±1.19% difference in the top-3 accuracy and simple accuracy across all years.

TSH based embeddings achieve an accuracy of 75.34±0.3% in the mixed language scenario for the year 2018. We see an average 3.44±4.41% difference in the accuracy of all years from the year 2018 in this particular scenario. For C++, these embeddings achieve an accuracy of 81.77±0.4% in the year 2018, with an average difference of 0.54±1.9% compared to the rest of the years. While for Python, they achieve an accuracy of 71.65±0.7% in the year 2018 with an average difference of 11.1±9.85% compared to the rest of the years. Lastly, for Java, these embeddings achieve an accuracy of 66.85±0.7% in the year 2018 with an average difference of 6.42±5.33% compared to the rest of the years.

These embeddings achieve a top-3 accuracy of 81.82±0.4% in the mixed language scenario for the year 2018, resulting in a difference of 6.48 compared to simple accuracy for the year 2018. We note an average 6.75±0.48% difference in the top-3 accuracy and simple accuracy across all years. For C++, these embeddings achieve a top-3 accuracy of 87.26±0.4% for the year 2018, resulting in a difference of 5.49 compared to simple accuracy for the year 2018. We note an average 6.23±0.65% difference in the top-3 accuracy and simple accuracy across all years. While for Python,

these embeddings achieve a top-3 accuracy of 79.6±0.5% for the year 2018, resulting in a difference of 7.95 compared to simple accuracy for the year 2018. We note an average 6.3±1.23% difference in the top-3 accuracy and simple accuracy across all years. Lastly, for Java, these embeddings achieve a top-3 accuracy of 73.27±0.6% for the year 2018, resulting in a difference of 6.42 compared to simple accuracy for the year 2018. We note an average 7.63±1.02% difference in the top-3 accuracy and simple accuracy across all years.

TH based embeddings achieve an accuracy of 73.39±0.4% in the mixed language scenario for the year 2018. We see an average of 3.67±4.55% difference in the accuracy of all years from 2018 in this particular scenario. For C++, these embeddings achieve an accuracy of 80.64±0.5% in the year 2018, with an average difference of 0.73±1.85% compared to the rest of the years. While for Python, they achieve an accuracy of 68.99±0.7% in the year 2018 with an average difference of 11.03±10.29% compared to the rest of the years. Lastly, for Java, these embeddings achieve an accuracy of 64.33±0.7% in the year 2018 with an average difference of 6.52±5.44% compared to the rest of the years.

These embeddings achieve a top-3 accuracy of 80.17±0.1% in the mixed language scenario for the year 2018, which results in a difference of 6.78 when compared to simple accuracy for the year 2018. We note an average 7.2±0.6% difference in the top-3 accuracy and simple accuracy across all years. For C++, these embeddings achieve a top-3 accuracy of 85.75±0.3% for the year 2018, resulting in a difference of 5.11 compared to simple accuracy for the year 2018. We note an average 6.4±0.74% difference in the top-3 accuracy and simple accuracy across all years. While for Python, these embeddings achieve a top-3 accuracy of 77.49±0.6% for the year 2018, resulting in a difference of 8.5 compared to simple accuracy for the year 2018. We note an average 7.44±1.19% difference in the top-3 accuracy and simple accuracy across all years. Lastly, for Java, these embeddings achieve a top-3 accuracy of 71.5±0.7% for the year 2018, which results in a difference of 7.17 compared to simple accuracy for the year 2018. We note an average 8.07±1.21% difference in the top-3 accuracy and simple accuracy across all years.

TH-soft based embeddings achieve an accuracy of 80.15±0.3% in the mixed language scenario for the year 2018. We see an average 2.17±3.66% difference in the accuracy of all years from the year 2018 in this particular scenario. For C++, these embeddings achieve an accuracy of 84.54±0.4% in the year 2018, with an average difference of 0.96±2.2% compared to the rest of the

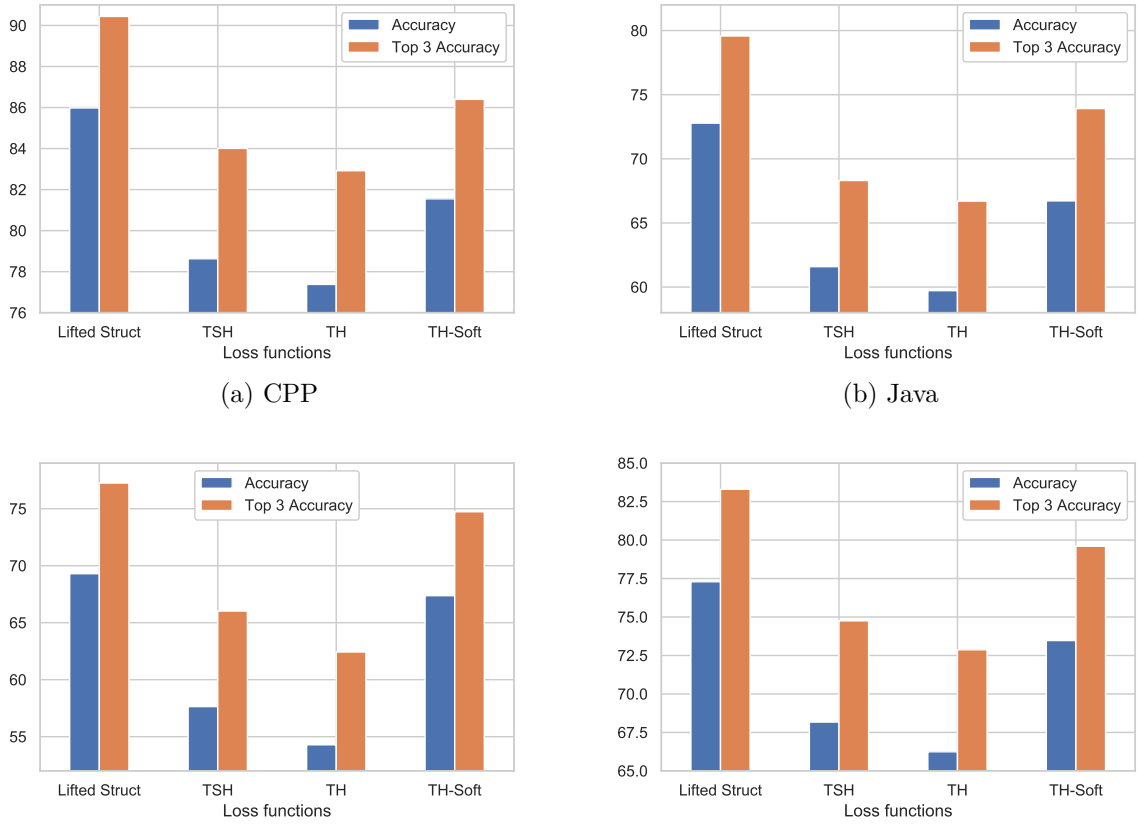
years. While for Python, they achieve an accuracy of 79.2±0.6% in the year 2018 with an average difference of 7.53±7.46% compared to the rest of the years. Lastly, for Java, these embeddings achieve an accuracy of 73.89±1.0% in the year 2018 with an average difference of 2.78±4.36% compared to the rest of the years.

These embeddings achieve a top-3 accuracy of 85.76±0.2% in the mixed language scenario for the year 2018, resulting in a difference of 5.61 compared to simple accuracy for the year 2018. We note an average 6.04±0.43% difference in the top-3 accuracy and simple accuracy across all years. For C++, these embeddings achieve a top-3 accuracy of 89.63±0.3% for the year 2018, resulting in a difference of 5.09 compared to simple accuracy for the year 2018. We note an average 5.52±0.59% difference in the top-3 accuracy and simple accuracy across all years. While for Python, these embeddings achieve a top-3 accuracy of 86.13±0.5% for the year 2018, which results in a difference of 6.93 compared to simple accuracy for the year 2018. We note an average 5.16±1.28% difference in the top-3 accuracy and simple accuracy across all years. Lastly, for Java, these embeddings achieve a top-3 accuracy of 80.12±0.7% for the year 2018, which results in a difference of 6.23 compared to simple accuracy for the year 2018. We note an average 7.57±1.01% difference in the top-3 accuracy and simple accuracy across all years.

We note a noticeable increase in performance yet again with the increase in the number of files as priors. However, again, one could argue that this increase in performance is due to the decrease in author count and not due to the increase in priors. In Figure 4.8 we compare the performance of the different loss functions in different scenarios under this particular scenario for the year 2020. As we can LS based embeddings always outperforms the TSH, TH, and its Soft margin variant based embeddings.

Table 4.9. LS: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages under simulated real-world constraint.

		C++			Python			Java			All Languages		
Year	**Files**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3 Accuracy**
2018	5	10090	82.75±0.4	89.03±0.3	4885	74.76±0.5	82.94±0.4	3734	68.29±0.6	77.17±0.6	18231	76.78±0.4	84.06±0.3
	7	8852	87.5±0.4	92.52±0.2	4169	80.54±0.5	87.35±0.4	3152	75.49±0.8	83.44±0.7	15922	82.24±0.3	88.49±0.2
	9	7850	89.9±0.2	94.06±0.2	3611	83.41±0.3	89.68±0.4	2726	79.56±0.6	86.39±0.5	14070	85.56±0.3	90.64±0.2
2019	5	13523	80.15±0.3	85.91±0.3	9597	68.22±0.4	75.63±0.4	5152	64.0±0.5	71.3±0.6	26224	70.36±0.2	77.05±0.2
	7	11832	85.28±0.2	90.41±0.3	8236	74.92±0.4	81.49±0.4	4475	71.52±0.6	78.14±0.6	23321	76.8±0.2	83.01±0.2
	9	10432	88.62±0.3	92.29±0.3	7123	79.21±0.3	84.37±0.4	3925	76.54±0.6	82.0±0.2	20833	81.13±0.3	85.83±0.2
2020	5	20677	76.54±0.3	83.14±0.2	11800	56.43±0.5	65.73±0.2	5032	59.26±0.5	67.34±0.4	36157	66.37±0.2	73.95±0.1
	7	18818	82.2±0.1	87.98±0.2	10499	63.95±0.4	72.82±0.3	4510	66.84±0.3	74.96±0.6	33174	73.02±0.2	80.05±0.1
	9	16916	85.98±0.3	90.44±0.2	9373	69.3±0.4	77.25±0.3	3985	72.78±0.5	79.58±0.7	30084	77.29±0.2	83.3±0.1



(a) CPP

(b) Java

(c) Python

(d) Mixed

Figure 4.8. Loss function performance comparison on simulated real world dataset.

4.2.4. Discussion

In this subsection, we discuss some of the further analysis that we conducted across all settings.

Compared with the other loss functions, LS achieves, on average 4.18±3.38 difference in accuracy across all settings and scenarios. Here a higher average score represents better performance. When compared with the other loss functions, TSH achieves an average of -1.77±3.34 difference in accuracy across all settings and scenarios. For TH, a similar comparison yields an average difference of -3.87±3.12. Lastly, When TH-Soft is compared with others, the average difference is 1.46±3.32. We can see the distributions in figure 4.9 from which these numbers are derived.

With this, we can confidently claim that LS loss function is superior to others by a significant margin while TH-Soft loss function comes in second, followed by TSH and TH, respectively.

Table 4.10. TSH: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages under simulated real-world constraint.

		C++			Python			Java			All Languages		
Year	**Files**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3 Accuracy**
2018	5	10090	72.09±0.3	79.35±0.4	4885	61.08±0.5	71.03±0.4	3734	53.71±0.5	61.23±0.7	18231	64.49±0.3	72.56±0.2
	7	8852	78.34±0.3	84.46±0.3	4169	67.74±0.5	76.93±0.3	3152	61.45±0.6	69.49±0.9	15922	71.06±0.2	78.61±0.2
	9	7850	81.77±0.4	87.26±0.4	3611	71.65±0.7	79.6±0.5	2726	66.85±0.7	73.27±0.6	14070	75.34±0.3	81.82±0.4
2019	5	13523	71.32±0.2	77.42±0.3	9597	56.21±0.5	64.46±0.4	5152	52.01±0.7	58.7±0.7	26224	59.82±0.2	66.7±0.2
	7	11832	77.52±0.3	83.31±0.2	8236	63.06±0.5	70.9±0.5	4475	60.05±0.5	66.91±0.8	23321	66.95±0.2	73.73±0.2
	9	10432	81.58±0.4	86.23±0.3	7123	67.83±0.4	74.35±0.4	3925	65.6±0.8	71.56±0.5	20833	71.65±0.3	77.41±0.2
2020	5	20677	67.97±0.3	74.76±0.2	11800	45.79±0.3	54.41±0.5	5032	47.34±0.6	54.98±0.5	36157	56.87±0.3	64.43±0.2
	7	18818	74.25±0.2	80.72±0.2	10499	52.9±0.3	61.86±0.3	4510	55.6±0.4	63.03±0.7	33174	63.63±0.1	70.94±0.1
	9	16916	78.63±0.2	84.01±0.2	9373	57.64±0.3	66.02±0.4	3985	61.59±0.4	68.31±0.5	30084	68.17±0.2	74.75±0.1

Table 4.11. TH: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages under simulated real-world constraint.

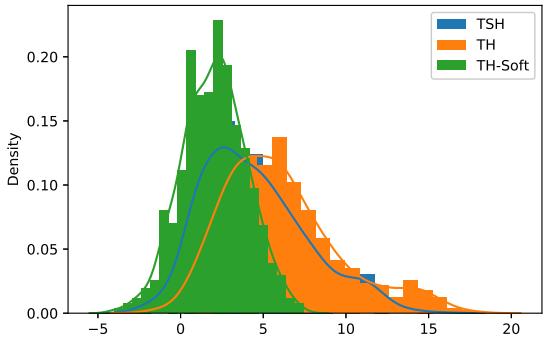
		C++			Python			Java			All Languages		
Year	**Files**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3 Accuracy**
2018	5	10090	70.77±0.4	77.66±0.4	4885	58.67±0.4	68.42±0.5	3734	50.84±0.6	59.3±0.6	18231	62.52±0.2	70.52±0.3
	7	8852	76.77±0.4	83.31±0.4	4169	65.0±0.6	75.03±0.5	3152	59.51±0.6	67.13±0.7	15922	69.06±0.3	76.92±0.3
	9	7850	80.64±0.5	85.75±0.3	3611	68.99±0.7	77.49±0.6	2726	64.33±0.7	71.5±0.7	14070	73.39±0.4	80.17±0.1
2019	5	13523	70.0±0.2	76.39±0.3	9597	52.68±0.4	61.1±0.4	5152	49.88±0.8	56.55±0.6	26224	57.66±0.3	64.72±0.2
	7	11832	76.38±0.3	82.23±0.3	8236	59.67±0.5	68.01±0.4	4475	57.79±0.6	64.14±1.0	23321	64.85±0.2	71.79±0.2
	9	10432	80.56±0.2	85.54±0.2	7123	64.43±0.5	72.02±0.4	3925	63.43±0.4	69.44±0.6	20833	69.62±0.4	75.74±0.3
2020	5	20677	66.54±0.2	73.54±0.2	11800	42.35±0.3	50.79±0.3	5032	45.32±0.9	53.04±0.5	36157	54.84±0.2	62.3±0.2
	7	18818	72.99±0.3	79.38±0.3	10499	49.2±0.3	57.96±0.3	4510	53.0±0.8	60.86±0.4	33174	61.46±0.2	68.88±0.2
	9	16916	77.38±0.1	82.92±0.2	9373	54.29±0.4	62.43±0.5	3985	59.72±0.5	66.7±0.8	30084	66.24±0.2	72.87±0.2

4.3. Open world setting

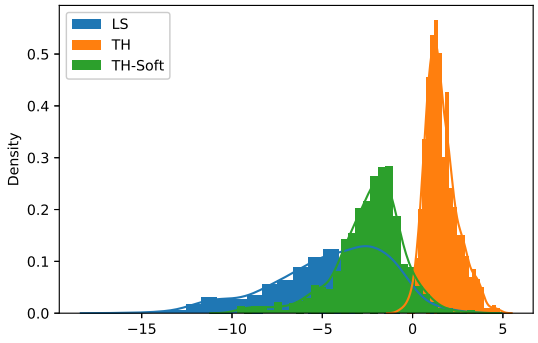
This section will discuss different loss functions' performance if we try to treat the problem in an open-world setting. For this setting, we only evaluated the approaches to the source code files from the year 2008. We use distance-based thresholding to determine whether the source code belongs to a known author or if it originated from 'outside the world". We essentially modify the nearest neighbor classifier to classify a particular input sample from outside if its distance from that known source code file is larger than a specific threshold. However, we need some priors to do such thresholding, and for that, we choose our training dataset, the year 2018. We calculate a distribution of mean distances within each cluster. We then take the mean of this distribution and calculate the standard deviation. We then select different distance cut-offs within the two standard deviations of the mean. To form the training and testing data for this setting, we sample 8 source codes from each author, and those who do not have that many source codes are put in the test dataset and labeled as outside. We evaluate our approach ten times for each cut-off point and get an average precision and recall score.

Table 4.12. TH-soft: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages under simulated real-world constraint.

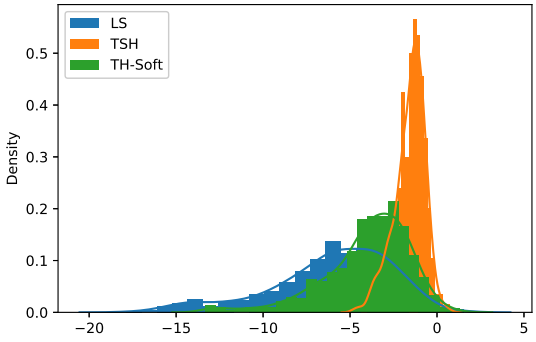
		C++			Python			Java			All Languages		
Year	**Files**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3 Accuracy**
2018	5	10090	75.66±0.3	82.43±0.4	4885	70.12±0.3	78.4±0.4	3734	61.36±0.8	69.71±0.7	18231	70.14±0.2	77.7±0.2
	7	8852	81.33±0.4	87.37±0.3	4169	75.96±0.5	83.25±0.5	3152	69.11±0.7	76.53±0.6	15922	76.45±0.2	83.28±0.3
	9	7850	84.54±0.4	89.63±0.3	3611	79.2±0.6	86.13±0.5	2726	73.89±1.0	80.12±0.7	14070	80.15±0.3	85.76±0.2
2019	5	13523	74.66±0.5	80.75±0.2	9597	65.93±0.6	72.9±0.3	5152	58.68±0.4	66.05±0.3	26224	65.84±0.2	72.61±0.2
	7	11832	80.63±0.3	86.01±0.3	8236	72.53±0.5	78.88±0.3	4475	66.52±0.7	73.2±0.5	23321	72.73±0.2	78.98±0.3
	9	10432	84.41±0.3	88.74±0.2	7123	76.69±0.6	82.15±0.4	3925	71.64±0.4	77.29±0.7	20833	77.15±0.3	82.37±0.2
2020	5	20677	71.3±0.3	78.02±0.1	11800	54.75±0.4	63.62±0.4	5032	53.24±0.5	61.3±0.7	36157	62.23±0.1	69.63±0.3
	7	18818	77.22±0.2	83.57±0.2	10499	62.23±0.4	70.63±0.4	4510	60.69±0.7	68.93±0.4	33174	68.74±0.2	75.98±0.2
	9	16916	81.55±0.3	86.4±0.3	9373	67.37±0.4	74.73±0.4	3985	66.72±0.7	73.92±0.7	30084	73.47±0.2	79.6±0.1



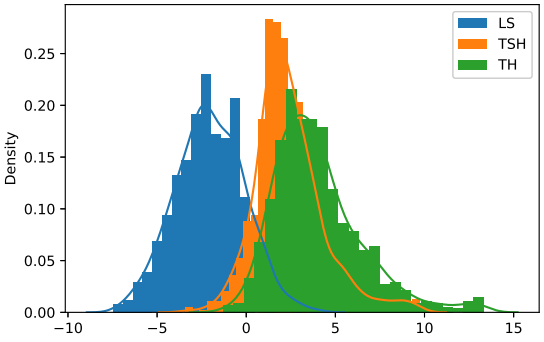
(a) Acc. difference for LS embeddings



(b) Acc. difference for TSH embeddings



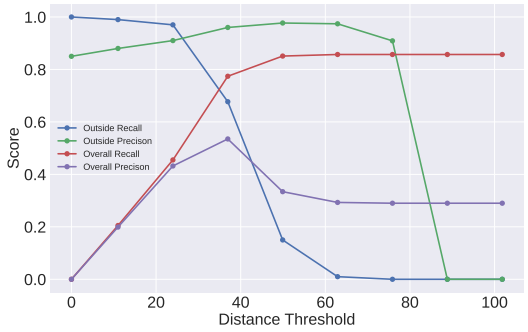
(c) Acc. difference for TH embeddings



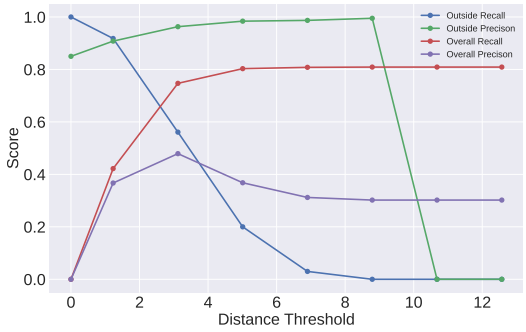
(d) Acc. difference for TH-soft embeddings

Figure 4.9. Distributions of accuracy difference across all settings between loss functions. Here the y-axis shows the density while the x-axis shows the accuracy difference.

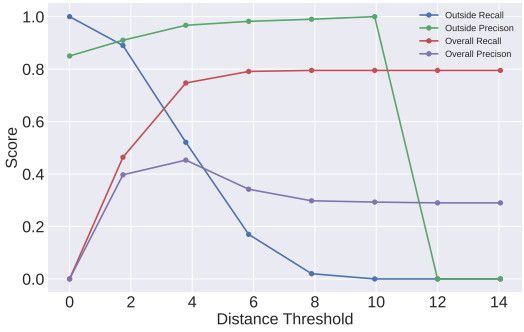
We want the reader to note that we are not claiming that a specific threshold will work all the time; instead, we are just reporting the results on our specific dataset. We have plotted out the performance of the models at different thresholds in figure 4.10.



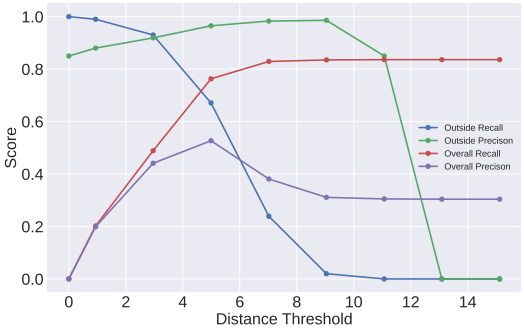
(a) Lifted structured loss



(b) Triplet semi-hard loss



(c) Triplet hard loss



(d) Triplet hard soft margin

Figure 4.10. Open world setting

For LS embeddings, we find that 36.95 was the most optimal distance threshold. We achieved an overall mean precision and recall score of 53.5% and 77.4%, respectively. For the outside class on the same threshold, we achieve an average recall of 67.7% and an average precision score of 96.0%.

For TSH embeddings, we determined that 3.12 was the most optimal distance threshold. We achieved an overall mean precision and recall score of 47.9% and 74.7%, respectively. For the outside class on the same threshold, we achieve an average recall of 56.1% and an average precision score of 96.3%.

For TH embeddings, we find that 3.78 was the most optimal distance threshold. We achieved an overall mean precision and recall score of 45.3% and 74.7%, respectively. For the outside class on the same threshold, we achieve an average recall of 52.1% and an average precision score of 96.7%.

For TH-soft embeddings, we determined that 4.99 was the most optimal distance threshold. We achieved an overall mean precision and recall score of 52.7% and 76.3%, respectively. For the outside class on the same threshold, we achieve an average recall of 67.1% and an average precision score of 96.5%.

As we can see, LS embeddings again outperform the other three embeddings. TH-Soft based embeddings perform almost as well with only minor degradation in overall precision and recall noted to be -0.79% and -1.1% respectively when compared with LS based embeddings.

4.4. Are the different models language oblivious?

This section will discuss the performance of different loss functions on the authorship attribution task but using source codes written in previously unseen programming languages. For this setting, we only evaluated the approaches using the source code files from the year 2020. In table 4.13, we show the performance of different loss functions on a number of previously unseen programming languages as we can observe that all of the models can extract language oblivious stylistic features.

Table 4.13. Accuracy of authorship attribution across different number of source code files per author, different unseen programming languages mined from the original GCJ dataset.

			LS		TSH		TH		TH-Soft	
Prog. Lang.	**Files**	**Authors**	**Acc.**	**Top 3 Acc.**	**Acc.**	**Top 3 Acc.**	**Acc.**	**Top 3 Acc.**	**Acc.**	**Top 3 Acc.**
C	5	579	89.4±1.2	93.07±1.1	82.45±1.4	86.8±1.5	82.11±1.4	85.99±1.4	81.38±1.3	86.3±1.1
	7	409	93.2±1.0	95.43±0.9	87.29±1.6	90.05±1.2	87.04±1.9	90.2±1.4	87.14±1.3	90.32±1.5
	9	262	95.04±0.7	96.34±0.8	90.23±1.1	92.75±1.6	90.42±1.5	91.07±0.8	90.57±1.8	91.49±1.9
Go	5	163	88.53±2.4	93.93±1.7	85.71±2.4	92.39±2.5	84.79±2.5	91.41±1.8	84.17±1.8	91.04±1.7
	7	136	93.16±1.4	96.54±1.4	90.0±1.9	94.04±1.6	90.15±2.9	91.99±1.3	89.12±2.4	93.6±1.8
	9	109	94.04±1.6	97.52±1.2	92.66±1.2	95.5±1.9	92.29±2.5	94.13±1.2	90.09±3.5	93.85±1.5
Ruby	5	95	84.21±3.5	91.16±3.2	83.58±3.4	88.63±2.0	78.84±2.3	83.79±2.6	81.05±3.8	85.68±3.1
	7	80	91.75±1.8	95.38±2.6	89.12±2.1	93.5±1.9	86.88±4.0	91.62±1.5	85.75±3.3	91.0±3.4
	9	60	92.83±2.9	95.83±2.6	91.0±1.7	95.5±2.7	91.17±4.1	93.83±2.1	88.5±3.8	94.0±1.7
PHP	5	51	97.65±2.9	99.61±0.8	96.08±2.3	98.24±1.1	96.67±1.3	99.22±1.3	96.47±3.0	99.41±0.9
	7	37	99.19±1.2	100.0±0.0	98.11±2.1	99.46±1.1	97.57±2.2	98.92±1.3	97.03±3.3	99.19±1.2
	9	32	99.06±1.4	99.69±0.9	98.75±2.1	99.06±1.4	97.81±2.4	99.06±1.4	98.75±1.5	99.06±1.4
C#	5	476	81.13±0.4	86.7±1.4	79.81±1.9	84.89±1.4	78.3±1.1	82.44±1.4	77.94±2.5	82.56±1.2
	7	393	86.26±0.9	90.08±1.5	85.04±2.0	89.67±1.5	84.43±1.1	88.42±1.8	84.86±1.1	86.9±1.3
	9	309	88.51±1.6	92.82±1.4	88.54±1.7	92.59±1.1	87.73±1.0	91.17±1.6	87.31±1.9	91.04±1.3
JS	5	259	90.46±1.3	94.17±1.1	88.19±1.3	92.36±1.6	83.78±1.9	90.08±2.0	83.13±1.6	88.42±1.5
	7	205	92.78±1.4	96.98±0.8	91.76±1.9	93.9±1.9	88.78±2.4	92.98±2.4	87.71±2.1	92.24±2.0
	9	164	93.9±2.0	97.68±0.9	93.6±1.5	94.88±1.0	92.13±2.0	95.12±1.1	91.34±1.5	94.21±0.8
Mixed	5	1623	86.64±0.7	90.96±0.6	82.8±0.8	87.78±0.7	80.97±0.6	86.31±0.6	81.2±0.9	85.78±0.5
	7	1262	91.07±0.6	94.02±0.5	87.62±0.6	91.48±0.6	87.1±0.6	90.43±0.5	86.02±0.7	89.55±0.5
	9	936	93.32±0.7	95.96±0.5	90.93±0.8	93.8±0.5	90.04±0.9	92.56±0.7	89.78±0.9	92.49±0.9

LS embeddings have an average accuracy difference of 3.18±1.93, while the median value is 3.12 compared to the rest of the loss functions. Here a higher positive value represents better performance. While for TSH, we observe an average difference of 0.15±2.23 and a median value of 0.47. We calculate the average difference for TH and TH-Soft to be -1.47±2.11 and -1.86±2.02. The median value for TH and TH-Soft was -0.96 and -1.32.

As we observed before, LS embeddings dominated the other three loss functions by a significant margin. Though this is not a definitive answer, the number of authors associated with each of these languages is pretty low compared to the three most popular languages in the competition, but it still shows promise.

5. FUTURE WORK

The following are some of the avenues we can consider to extend this work. In the data selection phase, we can evaluate our approach on other source code datasets, especially those sampled from the wild. In the neural network training phase, we can consider pre-training a neural network and fine-tuning that using such loss functions may lead to faster convergence. During this same phase, we can work on the source code files sampling so that there is an equal representation of all the programming languages and one does not overpower the others. For instance, in our experiments, the C++ language always outperforms the other programming languages due to a larger number of source files. Moreover, we can explore other loss functions that use cosine similarity as the "distance" metric [45] and determine their performance. We can also experiment with a Hybrid neural network that consists of both convolutions and recurrent layers. Also, we can consider encoding schemes other than character level representation, such as byte-level encoding. We believe that some of these approaches might lead to significantly smaller sequence sizes and make the evaluation of purely recurrent and even Transformers, proposed by Vaswani et al. [46], feasible. Furthermore, we can consider non-gradient based optimization techniques such as the Particle Swarm Optimization technique. We can use Neighborhood Component Analysis to transform the embeddings further, as our initial experiment shows around a 3% improvement over the vanilla NN embeddings. We can also consider applying regularization on the embedding layer, such as the one proposed by Zhang et al. [47], which enforces the embeddings to be spread out over the hyper-sphere by using an orthogonality based constraint. Also, we can use more complex obfuscators to determine the effect state of the art obfuscation techniques on the performance of our proposed system. Finally, our current work is based on the assumption that source code can be attributed to a single author. However, in real-world scenarios, multiple authors usually author source codes collectively, mainly when they belong to a single project. Identifying multiple programmers in a source code could be an exciting avenue to explore using our proposed approach.

6. CONCLUSION

Identifying the author of source code is crucial for multiple applications such as plagiarism detection, software forensic, and copyright 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 convolutional neural networks using various deep metric learning based loss functions 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 primarily on the three most popular programming languages within GCJ: Python, C++, Java. Results show that our models can achieve decent results under various constraints and scenarios. We evaluated our approach under obfuscated source-code only, original source-code only, and simulated real-world constraint. We also conducted two initial experiments. In the first initial experiment, we try to address the problem's open-world aspect, where the author could be someone outside the training set. While in the second experiment, we try to show that our approach can perform the attribution task using source codes written in previously unseen programming languages.

REFERENCES

[1] Steven Burrows and Seyed MM Tahaghoghi. Source code authorship attribution using n-grams. In *Proceedings of the Twelth Australasian Document Computing Symposium, Melbourne, Australia, RMIT University*, pages 32–39. Citeseer, 2007.

[2] Georgia Frantzeskou, Efstathios Stamatatos, Stefanos Gritzalis, and Sokratis Katsikas. Effective identification of source code authors using byte-level information. In *Proceedings of the 28th international conference on Software engineering*, pages 893–896, 2006.

[3] Mohammed Abuhamad, Tamer AbuHmed, Aziz Mohaisen, and DaeHun Nyang. Large-scale and language-oblivious code authorship identification. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 101–114, 2018.

[4] Yutian Lin, Liang Zheng, Zhedong Zheng, Yu Wu, Zhilan Hu, Chenggang Yan, and Yi Yang. Improving person re-identification by attribute and identity learning. *Pattern Recognition*, 95:151–161, 2019.

[5] Liang Zheng, Yi Yang, and Alexander G Hauptmann. Person re-identification: Past, present and future. *arXiv preprint arXiv:1610.02984*, 2016.

[6] Yantao Shen, Tong Xiao, Hongsheng Li, Shuai Yi, and Xiaogang Wang. Learning deep neural networks for vehicle re-id with visual-spatio-temporal path proposals. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1900–1909, 2017.

[7] Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt. De-anonymizing programmers via code stylometry. In *24th {USENIX} Security Symposium ({USENIX} Security 15)*, pages 255–270, 2015.

[8] Haibiao Ding and Mansur H Samadzadeh. Extraction of java program fingerprints for software authorship identification. *Journal of Systems and Software*, 72(1):49–57, 2004.

[9] Edwin Dauber, Aylin Caliskan, Richard Harang, Gregory Shearer, Michael Weisman, Frederica Nelson, and Rachel Greenstadt. Git blame who?: Stylistic authorship attribution of

small, incomplete source code fragments. *Proceedings on Privacy Enhancing Technologies*, 2019(3):389–408, 2019.

[10] Bander Alsulami, Edwin Dauber, Richard Harang, Spiros Mancoridis, and Rachel Greenstadt. Source code authorship attribution using long short-term memory based networks. In *European Symposium on Research in Computer Security*, pages 65–82. Springer, 2017.

[11] Mohammed Abuhamad, Ji-su Rhim, Tamer AbuHmed, Sana Ullah, Sanggil Kang, and DaeHun Nyang. Code authorship identification using convolutional neural networks. *Future Generation Computer Systems*, 95:104–115, 2019.

[12] S. Zafar, M. U. Sarwar, S. Salem, and M. Z. Malik. Language and obfuscation oblivious source code authorship attribution. *IEEE Access*, pages 1–1, 2020.

[13] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[14] Jian Wang, Feng Zhou, Shilei Wen, Xiao Liu, and Yuanqing Lin. Deep metric learning with angular loss. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 2593–2601, 2017.

[15] Xun Wang, Xintong Han, Weilin Huang, Dengke Dong, and Matthew R Scott. Multi-similarity loss with general pair weighting for deep metric learning. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 5022–5030, 2019.

[16] Kihyuk Sohn. Improved deep metric learning with multi-class n-pair loss objective. In *Advances in neural information processing systems*, pages 1857–1865, 2016.

[17] Raia Hadsell, Sumit Chopra, and Yann LeCun. Dimensionality reduction by learning an invariant mapping. In *2006 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'06)*, volume 2, pages 1735–1742. IEEE, 2006.

[18] Florian Schroff, Dmitry Kalenichenko, and James Philbin. Facenet: A unified embedding for face recognition and clustering. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 815–823, 2015.

[19] Alexander Hermans, Lucas Beyer, and Bastian Leibe. In defense of the triplet loss for person re-identification. *arXiv preprint arXiv:1703.07737*, 2017.

[20] Li Zhang, Tao Xiang, and Shaogang Gong. Learning a discriminative null space for person re-identification. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 1239–1248, 2016.

[21] De Cheng, Yihong Gong, Sanping Zhou, Jinjun Wang, and Nanning Zheng. Person re-identification by multi-channel parts-based cnn with improved triplet loss function. In *Proceedings of the iEEE conference on computer vision and pattern recognition*, pages 1335–1344, 2016.

[22] Hyun Oh Song, Yu Xiang, Stefanie Jegelka, and Silvio Savarese. Deep metric learning via lifted structured feature embedding. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 4004–4012, 2016.

[23] Farhan Ullah, Junfeng Wang, Sohail Jabbar, Fadi Al-Turjman, and Mamoun Alazab. Source code authorship attribution using hybrid approach of program dependence graph and deep learning model. *IEEE Access*, 7:141987–141999, 2019.

[24] Xinyu Yang, Guoai Xu, Qi Li, Yanhui Guo, and Miao Zhang. Authorship attribution of source code by using back propagation neural network based on particle swarm optimization. *PloS one*, 12(11), 2017.

[25] Steven Burrows, Alexandra L Uitdenbogerd, and Andrew Turpin. Comparing techniques for authorship attribution of source code. *Software: Practice and Experience*, 44(1):1–32, 2014.

[26] Aylin Caliskan, Fabian Yamaguchi, Edwin Dauber, Richard Harang, Konrad Rieck, Rachel Greenstadt, and Arvind Narayanan. When coding style survives compilation: De-anonymizing programmers from executable binaries. *arXiv preprint arXiv:1512.08546*, 2015.

[27] Xiaozhu Meng, Barton P Miller, and Kwang-Sung Jun. Identifying multiple authors in a binary program. In *European Symposium on Research in Computer Security*, pages 286–304. Springer, 2017.

[28] Weihua Chen, Xiaotang Chen, Jianguo Zhang, and Kaiqi Huang. Beyond triplet loss: a deep quadruplet network for person re-identification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 403–412, 2017.

[29] Sebastian Ruder, Parsa Ghaffari, and John G Breslin. Character-level and multi-channel convolutional neural networks for large-scale authorship attribution. *arXiv preprint arXiv:1609.06686*, 2016.

[30] Brian N. Pellin. Using classification techniques to determine source code authorship. 2006.

[31] Stephen G MacDonell, Andrew R Gray, Grant MacLennan, and Philip J Sallis. Software forensics for discriminating between program authors using case-based reasoning, feedforward neural networks and multiple discriminant analysis. In *ICONIP'99. ANZIIS'99 & ANNES'99 & ACNN'99. 6th International Conference on Neural Information Processing. Proceedings (Cat. No. 99EX378)*, volume 1, pages 66–71. IEEE, 1999.

[32] Steven Burrows, Alexandra L Uitdenbogerd, and Andrew Turpin. Application of information retrieval techniques for source code authorship attribution. In *International Conference on Database Systems for Advanced Applications*, pages 699–713. Springer, 2009.

[33] Bruce S Elenbogen and Naeem Seliya. Detecting outsourced student programming assignments. *Journal of Computing Sciences in Colleges*, 23(3):50–57, 2008.

[34] Robert Charles Lange and Spiros 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*, pages 2082–2089, 2007.

[35] Ivan Krsul and Eugene H Spafford. Authorship analysis: Identifying the author of a program. *Computers & Security*, 16(3):233–257, 1997.

[36] Charu C Aggarwal, Alexander Hinneburg, and Daniel A Keim. On the surprising behavior of distance metrics in high dimensional space. In *International conference on database theory*, pages 420–434. Springer, 2001.

[37] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[38] Jonathan Tompson, Ross Goroshin, Arjun Jain, Yann LeCun, and Christoph Bregler. Efficient object localization using convolutional networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 648–656, 2015.

[39] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Delving deep into rectifiers: Surpassing human-level performance on imagenet classification. In *Proceedings of the IEEE international conference on computer vision*, pages 1026–1034, 2015.

[40] Vinod Nair and Geoffrey E Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[41] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of machine learning research*, 12(Aug):2493–2537, 2011.

[42] Xavier Glorot and Yoshua Bengio. Understanding the difficulty of training deep feedforward neural networks. In *Proceedings of the thirteenth international conference on artificial intelligence and statistics*, pages 249–256, 2010.

[43] Liyuan Liu, Haoming Jiang, Pengcheng He, Weizhu Chen, Xiaodong Liu, Jianfeng Gao, and Jiawei Han. On the variance of the adaptive learning rate and beyond. *arXiv preprint arXiv:1908.03265*, 2019.

[44] Michael Zhang, James Lucas, Jimmy Ba, and Geoffrey E Hinton. Lookahead optimizer: k steps forward, 1 step back. In *Advances in Neural Information Processing Systems*, pages 9597–9608, 2019.

[45] Kevin Musgrave, Serge Belongie, and Ser-Nam Lim. A metric learning reality check. *arXiv preprint arXiv:2003.08505*, 2020.

[46] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008, 2017.

[47] Xu Zhang, Felix X Yu, Sanjiv Kumar, and Shih-Fu Chang. Learning spread-out local feature descriptors. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 4595–4603, 2017.

APPENDIX. SUPPLEMENTARY TABLES

Table A.1. LS: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages using the original GCJ dataset.

		C++			Python			Java			All Languages		
Year	**Files**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**
2008	5	1547	82.34±0.7	88.17±0.6	244	89.26±1.7	93.81±1.1	609	81.84±1.2	87.47±0.5	2427	81.44±0.8	88.22±0.5
	7	1063	86.52±1.2	92.14±1.0	108	95.74±1.8	97.87±1.1	308	87.56±1.4	92.31±1.3	1503	86.35±0.7	92.0±0.5
	9	807	89.76±0.6	94.55±0.8	37	98.38±1.3	98.11±1.2	176	90.91±1.9	95.34±1.9	1041	89.6±0.7	94.29±1.0
2009	5	2402	77.6±0.9	84.81±0.5	386	82.67±1.5	91.37±1.1	791	76.99±1.0	85.26±1.3	3625	76.93±0.5	84.6±0.5
	7	1729	84.88±0.6	91.19±0.4	216	91.94±2.1	96.94±0.4	495	85.19±1.2	91.43±0.7	2513	84.06±0.7	90.58±0.5
	9	1107	90.18±1.1	94.69±0.5	97	95.36±1.4	97.84±1.6	281	91.67±1.3	95.2±0.8	1539	90.51±0.7	94.16±0.5
2010	5	2463	86.09±0.3	91.38±0.5	445	91.15±1.2	95.35±1.2	861	89.16±0.5	94.17±0.8	3911	83.95±0.5	89.15±0.3
	7	1673	90.68±0.3	94.87±0.5	247	94.25±1.2	97.61±1.2	568	94.47±0.6	97.2±0.6	2667	88.92±0.4	92.8±0.6
	9	1006	94.28±0.7	96.85±0.4	126	97.78±1.1	98.89±0.8	335	96.33±0.8	98.18±0.9	1623	92.8±0.7	94.79±0.6
2011	5	3772	87.08±0.5	91.99±0.3	759	91.95±0.6	95.72±0.5	1373	86.74±0.8	91.51±0.6	5966	86.62±0.3	91.95±0.2
	7	2880	91.84±0.6	95.73±0.2	460	95.09±0.9	97.72±0.6	947	90.83±0.8	94.75±0.6	4367	91.38±0.2	95.21±0.3
	9	1989	94.72±0.5	97.17±0.4	242	97.48±0.7	98.72±0.7	612	94.72±0.7	96.85±0.6	2926	94.59±0.2	96.99±0.3
2012	5	3340	84.43±0.6	90.01±0.4	749	89.19±0.8	93.68±0.6	1252	83.35±0.7	89.34±0.8	5467	83.67±0.5	89.3±0.3
	7	2127	90.57±0.6	94.35±0.5	343	94.2±1.6	96.91±0.6	621	90.05±0.8	94.09±0.5	3193	89.85±0.5	93.52±0.2
	9	1255	93.28±0.6	96.12±0.6	155	97.29±0.8	99.16±0.6	340	94.65±0.6	96.71±0.8	1823	92.95±0.6	95.99±0.3
2013	5	4605	85.29±0.4	90.88±0.3	1263	89.26±0.3	94.05±0.5	1957	84.19±0.6	89.82±0.6	8046	83.72±0.3	89.08±0.3
	7	2941	90.82±0.3	95.08±0.3	648	94.44±0.6	97.22±0.5	1077	90.55±0.5	94.26±0.5	4916	89.3±0.4	93.53±0.2
	9	1820	94.25±0.7	97.02±0.3	314	96.91±0.9	98.09±0.6	597	93.69±0.6	96.3±0.6	2945	93.0±0.4	95.9±0.3
2014	5	5185	86.4±0.4	91.4±0.2	1422	90.44±0.7	94.9±0.5	1939	85.53±0.5	90.75±0.4	8648	85.35±0.3	90.77±0.3
	7	3296	91.37±0.3	95.23±0.3	725	94.86±0.6	97.45±0.4	1039	91.57±0.5	95.38±0.8	5186	91.21±0.3	94.64±0.3
	9	2166	94.52±0.4	96.95±0.4	336	97.17±1.1	98.78±0.2	599	94.11±1.0	97.11±0.6	3222	93.82±0.4	96.56±0.2
2015	5	4615	88.32±0.3	92.84±0.4	998	92.21±0.7	96.06±0.5	1344	89.01±0.7	93.43±0.4	7038	87.98±0.3	92.25±0.3
	7	3293	92.35±0.3	95.77±0.3	593	96.07±0.6	98.41±0.8	830	93.29±0.7	96.86±0.5	4823	91.79±0.4	95.41±0.4
	9	2239	95.07±0.4	97.1±0.2	329	98.02±0.5	99.0±0.4	500	95.58±0.7	98.02±0.6	3166	94.88±0.3	96.92±0.2
2016	5	6511	85.31±0.2	90.46±0.2	2532	82.02±0.5	88.53±0.4	2605	81.19±0.5	87.99±0.6	11896	81.77±0.3	87.72±0.1
	7	4624	90.55±0.5	94.29±0.4	1567	89.59±0.8	94.36±0.4	1684	87.81±0.5	93.33±0.5	8181	88.19±0.3	92.77±0.2
	9	3217	94.08±0.3	96.62±0.3	915	93.56±0.8	96.82±0.4	1060	92.34±0.6	95.94±0.6	5435	92.41±0.3	95.41±0.3
2017	5	5911	89.62±0.3	93.97±0.2	1960	88.29±0.4	93.25±0.2	1703	85.58±0.5	91.3±0.6	9698	87.54±0.3	92.32±0.2
	7	4070	94.16±0.3	97.16±0.1	1105	94.05±0.5	97.34±0.3	979	91.88±0.7	95.63±0.5	6305	93.13±0.4	96.37±0.1
	9	2752	96.32±0.2	98.18±0.2	621	96.96±0.7	98.42±0.5	550	95.67±0.9	97.27±0.8	4072	95.71±0.3	97.64±0.2
2018	5	7830	88.18±0.2	92.12±0.3	3525	82.26±0.5	87.75±0.3	2726	84.79±0.4	89.42±0.4	13983	84.79±0.4	89.14±0.3
	7	6376	91.86±0.2	94.83±0.2	2620	86.49±0.5	91.66±0.5	2083	89.2±0.5	93.07±0.6	11152	88.99±0.3	92.91±0.3
	9	5277	93.89±0.3	96.34±0.2	2054	89.85±0.5	93.59±0.5	1608	91.92±0.6	95.04±0.5	9053	91.59±0.2	94.63±0.2
2019	5	10122	86.69±0.2	90.09±0.2	6281	81.85±0.5	85.38±0.4	3921	82.25±0.5	86.35±0.6	19672	81.86±0.2	85.37±0.2
	7	8087	90.8±0.5	93.57±0.2	4840	86.78±0.4	89.86±0.4	3069	87.25±0.5	90.54±0.5	15973	86.65±0.2	89.91±0.2
	9	6468	93.19±0.3	95.27±0.2	3759	89.6±0.4	92.05±0.3	2373	90.1±0.7	92.95±0.3	12952	89.65±0.2	92.25±0.1
2020	5	16872	83.59±0.4	87.62±0.2	9124	70.65±0.3	76.16±0.5	3982	79.65±0.7	84.96±0.4	29838	77.15±0.3	81.83±0.2
	7	13694	88.41±0.3	91.86±0.2	7130	77.9±0.3	83.04±0.3	3125	85.25±0.6	89.71±0.4	24281	83.11±0.1	87.25±0.1
	9	11143	91.4±0.3	94.08±0.2	5513	82.1±0.4	86.88±0.4	2384	88.63±0.6	92.63±0.4	19529	86.95±0.2	90.3±0.2

Table A.2. TSH: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages evaluated on obfuscated dataset.

		C++			Python			Java			All Languages		
Year	**Files**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**
2008	5	1547	74.82±1.1	82.29±0.7	244	89.1±2.2	94.06±1.5	609	80.51±0.6	86.06±1.2	2427	76.54±0.4	83.84±0.6
	7	1063	80.7±0.7	87.42±0.6	108	95.74±1.2	98.24±1.2	308	86.01±1.4	91.14±1.5	1503	82.17±0.6	88.53±0.6
	9	807	84.3±0.7	90.27±1.0	37	97.57±1.9	98.92±1.8	176	89.77±2.5	93.12±2.0	1041	85.08±0.7	90.69±1.0
2009	5	2402	70.71±0.7	79.25±0.7	386	81.76±0.9	90.41±1.6	791	75.09±1.2	82.72±0.8	3625	71.9±0.7	80.36±0.4
	7	1729	78.81±0.8	86.09±0.8	216	91.2±1.4	96.81±0.9	495	84.34±1.5	90.59±1.4	2513	80.29±0.8	86.77±0.5
	9	1107	86.03±0.7	91.7±0.3	97	96.6±1.5	97.94±1.4	281	90.85±1.0	94.48±1.0	1539	86.47±0.8	91.55±0.6
2010	5	2463	79.85±0.5	85.61±0.5	445	90.0±0.9	94.83±0.8	861	87.42±1.1	91.51±0.6	3911	79.08±0.4	84.87±0.5
	7	1673	86.46±0.8	91.32±0.7	247	95.43±1.1	97.09±0.8	568	92.99±1.2	96.36±0.7	2667	85.85±0.5	89.94±0.4
	9	1006	91.36±1.2	95.21±0.7	126	96.67±1.0	98.65±1.1	335	95.73±0.9	97.13±1.1	1623	89.84±0.4	93.42±0.6
2011	5	3772	79.44±0.6	85.89±0.3	759	89.18±0.5	93.89±0.6	1373	84.21±1.2	88.87±0.6	5966	80.79±0.3	86.78±0.3
	7	2880	85.57±0.4	91.28±0.4	460	93.11±0.4	96.59±0.6	947	89.52±0.3	93.03±0.8	4367	86.66±0.4	91.65±0.2
	9	1989	90.26±0.8	94.24±0.4	242	95.7±1.3	98.43±1.0	612	93.63±0.7	95.95±0.8	2926	90.73±0.4	94.61±0.4
2012	5	3340	78.41±0.5	84.41±0.6	749	86.41±1.1	91.67±0.7	1252	81.83±0.9	87.05±0.8	5467	78.85±0.6	84.9±0.4
	7	2127	85.33±0.4	91.21±0.5	343	92.33±1.0	95.54±1.0	621	88.2±1.2	92.45±0.9	3193	85.66±0.7	90.82±0.5
	9	1255	90.06±0.7	94.42±0.3	155	95.74±1.4	98.0±1.1	340	93.21±0.9	96.35±0.9	1823	90.28±0.5	94.32±0.6
2013	5	4605	79.38±0.3	85.37±0.4	1263	87.09±0.8	91.88±0.5	1957	80.85±0.4	86.43±0.6	8046	78.89±0.2	84.73±0.4
	7	2941	86.63±0.5	91.62±0.6	648	91.99±1.3	96.22±0.7	1077	87.83±0.9	91.68±0.9	4916	85.32±0.4	90.53±0.2
	9	1820	91.1±0.3	94.9±0.5	314	96.02±0.4	98.12±0.6	597	91.54±1.1	94.87±0.5	2945	90.15±0.6	93.88±0.4
2014	5	5185	79.86±0.5	85.7±0.4	1422	87.29±1.0	92.18±0.5	1939	82.76±0.9	88.08±0.7	8648	80.37±0.3	86.4±0.2
	7	3296	86.6±0.5	91.61±0.4	725	92.59±0.9	96.33±0.9	1039	89.41±0.8	93.51±0.5	5186	86.88±0.4	91.75±0.2
	9	2166	90.67±0.4	94.73±0.3	336	96.22±1.1	98.57±0.4	599	92.62±0.7	96.24±0.6	3222	91.06±0.5	94.79±0.3
2015	5	4615	82.66±0.5	87.88±0.3	998	89.74±0.8	93.73±0.6	1344	86.27±1.0	91.11±0.5	7038	83.2±0.4	88.49±0.3
	7	3293	87.64±0.5	92.48±0.3	593	94.22±0.6	96.9±0.7	830	91.4±0.8	95.4±0.7	4823	88.08±0.3	92.6±0.2
	9	2239	91.67±0.4	95.37±0.4	329	96.81±0.6	98.42±0.7	500	94.2±0.8	97.46±0.9	3166	92.04±0.6	95.11±0.5
2016	5	6511	79.2±0.3	85.06±0.3	2532	77.94±0.7	84.95±0.5	2605	77.26±0.6	83.98±0.6	11896	76.49±0.3	83.11±0.2
	7	4624	86.0±0.3	91.24±0.3	1567	85.4±0.7	91.74±0.4	1684	85.33±0.6	91.05±0.7	8181	84.01±0.4	89.63±0.2
	9	3217	90.71±0.3	94.16±0.4	915	91.57±0.7	95.1±0.5	1060	90.74±0.6	94.57±0.8	5435	89.05±0.3	93.26±0.2
2017	5	5911	83.7±0.4	88.77±0.3	1960	83.27±0.8	89.94±0.5	1703	81.89±0.5	87.82±0.5	9698	82.22±0.3	87.76±0.2
	7	4070	89.94±0.4	94.15±0.3	1105	91.29±0.6	95.78±0.4	979	89.86±1.0	93.62±0.6	6305	89.11±0.2	93.54±0.2
	9	2752	93.73±0.2	96.66±0.2	621	94.75±0.9	97.92±0.6	550	94.69±0.6	96.69±0.5	4072	93.08±0.3	96.21±0.3
2018	5	7830	80.37±0.2	84.81±0.5	3525	71.4±1.2	77.0±0.3	2726	75.77±0.5	80.55±0.4	13983	75.87±0.2	81.05±0.3
	7	6376	85.41±0.5	89.49±0.4	2620	77.44±0.7	83.13±0.4	2083	81.98±0.6	85.78±0.7	11152	81.57±0.3	86.43±0.3
	9	5277	88.45±0.5	92.16±0.4	2054	81.72±0.9	86.1±0.6	1608	84.94±0.7	88.92±0.5	9053	85.25±0.3	89.19±0.4
2019	5	10122	81.54±0.2	85.5±0.3	6281	74.77±0.7	78.69±0.4	3921	75.08±0.7	79.37±0.4	19672	75.55±0.3	79.64±0.3
	7	8087	86.78±0.2	89.87±0.2	4840	80.34±0.6	84.11±0.4	3069	81.24±0.6	84.92±0.5	15973	81.51±0.3	84.98±0.2
	9	6468	89.65±0.3	92.51±0.2	3759	83.87±0.3	86.98±0.4	2373	85.2±0.5	87.96±0.6	12952	85.03±0.3	88.28±0.2
2020	5	16872	78.03±0.3	82.36±0.2	9124	62.82±0.2	68.23±0.4	3982	73.05±0.7	78.25±0.6	29838	70.89±0.2	75.72±0.2
	7	13694	83.73±0.3	87.59±0.3	7130	70.43±0.8	75.77±0.6	3125	79.53±0.7	84.17±0.8	24281	77.43±0.2	81.92±0.3
	9	11143	87.66±0.2	90.67±0.2	5513	75.53±0.4	80.12±0.7	2384	84.0±0.3	88.21±0.7	19529	82.05±0.2	85.68±0.2

Table A.3. TH: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages using the original GCJ dataset.

		C++			Python			Java			All Languages		
Year	**Files**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**
2008	5	1547	74.4±1.0	81.84±0.9	244	87.34±2.0	93.2±1.4	609	78.44±1.5	84.27±0.9	2427	75.94±0.4	82.66±0.6
	7	1063	79.7±0.5	87.2±0.5	108	94.35±1.6	96.94±1.3	308	85.19±2.4	91.36±0.9	1503	80.57±0.9	87.64±0.7
	9	807	83.85±0.9	90.5±1.0	37	97.84±1.6	99.46±1.1	176	89.03±1.3	93.86±1.6	1041	84.64±0.7	90.49±0.8
2009	5	2402	69.8±0.6	78.06±1.0	386	79.79±1.9	88.89±1.0	791	72.38±1.2	80.29±1.3	3625	70.1±0.5	78.55±0.4
	7	1729	77.99±1.2	85.44±0.5	216	90.88±1.4	95.65±0.7	495	82.57±1.8	88.63±0.9	2513	78.85±0.8	86.12±0.5
	9	1107	84.99±0.6	90.55±1.2	97	94.02±1.6	96.08±1.6	281	88.43±1.7	93.77±0.9	1539	85.61±0.4	90.62±0.8
2010	5	2463	78.52±0.8	84.99±0.7	445	88.74±0.9	94.31±1.2	861	86.19±1.1	90.77±0.9	3911	78.08±0.5	83.87±0.4
	7	1673	85.97±0.7	90.99±0.3	247	93.85±1.4	95.99±1.0	568	91.51±0.9	95.7±0.9	2667	84.57±0.7	89.62±0.4
	9	1006	90.7±0.8	94.9±0.6	126	96.9±1.6	98.65±0.7	335	94.81±0.9	97.55±0.7	1623	89.29±0.5	92.99±0.3
2011	5	3772	77.92±0.6	84.94±0.4	759	87.8±1.3	92.75±0.5	1373	82.75±0.8	87.83±0.9	5966	79.54±0.4	85.92±0.3
	7	2880	84.38±0.5	90.33±0.6	460	92.37±0.6	96.13±0.9	947	87.51±0.8	92.71±0.5	4367	85.62±0.4	90.82±0.4
	9	1989	89.38±0.6	94.05±0.4	242	96.32±1.0	97.77±1.0	612	91.94±1.0	95.54±0.7	2926	90.3±0.5	94.17±0.6
2012	5	3340	76.6±0.6	83.05±0.4	749	85.35±0.9	90.92±1.0	1252	80.38±1.2	86.57±0.6	5467	77.09±0.5	83.74±0.4
	7	2127	84.46±0.9	90.05±0.4	343	92.13±0.9	95.19±0.8	621	87.54±0.8	91.66±0.8	3193	84.27±0.5	89.91±0.4
	9	1255	89.03±1.0	93.9±0.6	155	95.81±1.3	97.68±0.7	340	92.0±1.1	95.56±0.9	1823	89.25±0.5	93.59±0.4
2013	5	4605	77.92±0.3	84.59±0.3	1263	84.47±0.9	91.08±0.9	1957	78.56±0.7	84.57±0.5	8046	77.27±0.4	83.41±0.3
	7	2941	85.18±0.8	90.86±0.6	648	92.15±0.6	95.23±0.9	1077	86.21±0.7	90.58±0.7	4916	84.18±0.4	89.55±0.2
	9	1820	89.6±0.4	94.63±0.4	314	94.65±1.5	97.58±0.6	597	90.6±0.6	94.09±1.2	2945	89.12±0.7	93.01±0.5
2014	5	5185	78.54±0.4	84.82±0.5	1422	86.07±1.0	91.53±0.5	1939	80.87±0.6	86.9±0.7	8648	78.88±0.3	85.24±0.4
	7	3296	85.24±0.5	90.71±0.3	725	91.41±0.9	96.17±0.5	1039	87.8±0.5	92.83±0.8	5186	85.81±0.5	90.95±0.3
	9	2166	89.84±0.3	94.04±0.5	336	95.68±1.1	98.01±0.8	599	91.92±1.1	95.28±0.5	3222	89.77±0.3	93.97±0.4
2015	5	4615	81.21±0.5	86.77±0.4	998	88.54±0.7	93.3±0.7	1344	84.87±0.7	89.26±0.7	7038	81.8±0.4	87.57±0.3
	7	3293	86.3±0.3	91.72±0.3	593	93.52±0.7	96.63±0.9	830	89.88±0.9	94.93±0.6	4823	86.92±0.2	91.74±0.2
	9	2239	90.79±0.6	94.55±0.4	329	95.99±0.9	98.27±0.3	500	93.34±0.8	97.44±0.6	3166	90.68±0.5	94.81±0.3
2016	5	6511	77.65±0.4	84.1±0.4	2532	76.1±0.7	84.13±0.6	2605	74.43±0.7	81.88±0.3	11896	74.95±0.2	81.85±0.2
	7	4624	84.31±0.3	90.31±0.2	1567	84.88±0.7	91.41±0.6	1684	83.41±0.6	89.32±0.5	8181	82.47±0.2	88.5±0.4
	9	3217	89.52±0.9	93.72±0.4	915	90.93±0.8	94.82±0.6	1060	89.37±1.0	93.26±0.4	5435	87.84±0.4	92.52±0.3
2017	5	5911	81.85±0.3	87.65±0.3	1960	81.95±0.8	89.05±0.7	1703	79.97±0.8	86.12±0.7	9698	80.31±0.3	86.66±0.2
	7	4070	89.06±0.3	93.55±0.3	1105	89.98±0.6	94.83±0.6	979	88.29±1.0	93.22±0.6	6305	88.05±0.3	92.9±0.3
	9	2752	92.91±0.4	96.23±0.4	621	93.98±0.8	97.17±0.6	550	93.36±0.6	96.31±0.7	4072	92.4±0.5	95.65±0.3
2018	5	7830	78.6±0.3	83.31±0.4	3525	70.22±0.9	76.35±0.7	2726	74.79±0.8	79.37±1.0	13983	74.4±0.3	79.49±0.2
	7	6376	83.92±0.5	87.95±0.3	2620	76.03±0.6	82.06±0.6	2083	80.49±0.9	84.94±0.6	11152	80.23±0.2	85.01±0.3
	9	5277	86.84±0.3	90.63±0.4	2054	80.24±0.6	85.63±0.8	1608	83.76±0.6	87.84±0.5	9053	83.88±0.3	87.89±0.2
2019	5	10122	80.25±0.3	84.04±0.3	6281	74.34±0.4	78.11±0.6	3921	74.11±0.6	77.87±0.5	19672	74.32±0.2	78.34±0.2
	7	8087	85.47±0.3	88.89±0.2	4840	80.19±0.4	83.44±0.5	3069	80.26±0.6	83.78±0.7	15973	80.44±0.4	84.26±0.2
	9	6468	88.78±0.2	91.54±0.1	3759	83.25±0.4	86.76±0.5	2373	84.13±0.8	87.05±0.7	12952	84.1±0.4	87.34±0.2
2020	5	16872	76.54±0.3	80.88±0.4	9124	61.84±0.3	67.29±0.2	3982	71.69±0.7	76.73±0.3	29838	69.52±0.2	74.39±0.1
	7	13694	82.4±0.3	86.45±0.3	7130	69.85±0.4	75.14±0.7	3125	78.4±0.7	83.03±0.4	24281	76.36±0.2	80.71±0.2
	9	11143	86.28±0.3	89.62±0.3	5513	75.18±0.5	79.43±0.5	2384	83.04±0.5	86.95±0.8	19529	81.05±0.2	84.75±0.2

Table A.4. TH-Soft: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages under non-obfuscated settings.

		C++			Python			Java			All Languages		
Year	**Files**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**
2008	5	1547	76.39±0.9	83.53±0.5	244	89.88±1.6	95.0±1.4	609	81.84±1.1	86.98±1.2	2427	78.63±0.8	84.71±0.4
	7	1063	81.62±0.8	88.46±1.0	108	95.74±1.8	98.24±1.2	308	87.27±1.1	91.66±1.1	1503	83.39±1.2	89.17±0.5
	9	807	85.29±1.5	91.62±1.2	37	97.03±3.7	99.73±0.8	176	90.91±1.3	94.03±1.2	1041	86.53±0.4	91.76±0.7
2009	5	2402	71.71±0.9	78.94±0.8	386	85.6±0.9	92.67±1.1	791	74.45±1.1	83.25±0.9	3625	72.29±0.2	80.32±0.8
	7	1729	80.28±0.8	86.08±0.8	216	92.55±1.5	97.45±0.9	495	82.89±1.6	89.88±1.2	2513	80.43±0.5	86.72±0.4
	9	1107	86.56±0.4	91.42±0.5	97	95.57±1.9	98.56±0.7	281	90.46±1.8	94.7±0.5	1539	87.23±0.6	91.59±0.9
2010	5	2463	81.88±0.3	87.69±0.6	445	92.54±0.7	96.07±1.1	861	89.62±0.7	92.86±0.6	3911	81.52±0.5	86.66±0.6
	7	1673	88.42±0.5	92.82±0.3	247	96.23±1.0	97.69±1.0	568	93.87±0.9	96.53±0.9	2667	87.26±0.6	91.28±0.5
	9	1006	92.37±1.0	95.53±0.4	126	97.86±1.0	98.81±1.0	335	96.36±0.6	97.76±0.4	1623	90.99±0.5	93.91±0.6
2011	5	3772	82.84±0.3	88.06±0.4	759	91.36±1.1	95.19±0.4	1373	86.28±0.8	91.13±0.7	5966	83.76±0.4	89.24±0.3
	7	2880	88.24±0.4	92.78±0.2	460	94.5±1.0	97.74±0.8	947	90.7±1.1	94.63±0.6	4367	88.91±0.4	93.28±0.4
	9	1989	91.84±0.6	95.85±0.3	242	96.94±1.2	98.76±0.8	612	94.66±0.6	96.52±0.6	2926	92.35±0.4	95.72±0.2
2012	5	3340	79.49±0.4	85.64±0.3	749	88.5±1.0	92.96±0.8	1252	82.86±0.7	87.92±1.0	5467	80.38±0.4	86.24±0.4
	7	2127	86.53±0.7	92.24±0.4	343	93.76±1.1	96.88±0.9	621	88.45±1.3	92.98±0.9	3193	87.02±0.6	91.82±0.5
	9	1255	90.85±0.7	94.79±0.3	155	96.77±1.0	98.06±1.3	340	92.97±1.0	96.21±0.8	1823	91.05±0.3	94.63±0.5
2013	5	4605	82.39±0.6	87.81±0.3	1263	88.75±0.7	93.17±0.5	1957	82.6±0.8	87.74±0.6	8046	81.33±0.3	86.9±0.3
	7	2941	88.55±0.6	93.12±0.4	648	94.01±0.9	96.62±0.8	1077	88.53±1.1	92.88±0.7	4916	87.53±0.4	91.79±0.3
	9	1820	92.71±0.4	95.6±0.6	314	96.31±1.0	98.28±0.9	597	91.69±0.9	95.93±0.7	2945	91.43±0.5	94.59±0.3
2014	5	5185	82.4±0.4	88.02±0.3	1422	89.73±0.8	94.25±0.5	1939	85.3±0.6	90.37±0.4	8648	83.18±0.4	88.56±0.2
	7	3296	88.71±0.5	92.96±0.4	725	94.43±0.5	97.03±0.4	1039	91.92±0.7	95.37±0.5	5186	88.91±0.4	93.24±0.2
	9	2166	92.46±0.5	95.44±0.3	336	97.23±0.9	98.27±0.6	599	94.22±0.8	97.15±0.4	3222	92.83±0.2	95.41±0.2
2015	5	4615	85.26±0.5	90.28±0.3	998	91.77±0.7	95.28±0.5	1344	88.47±0.6	92.92±0.7	7038	85.62±0.4	90.3±0.3
	7	3293	89.69±0.3	94.06±0.5	593	95.41±0.9	97.93±0.5	830	93.34±0.7	96.35±0.4	4823	90.37±0.3	94.01±0.3
	9	2239	92.75±0.6	96.16±0.3	329	97.36±0.6	99.15±0.4	500	95.02±0.8	97.7±0.6	3166	93.54±0.4	96.05±0.2
2016	5	6511	81.03±0.3	86.59±0.2	2532	80.41±0.6	87.27±0.6	2605	78.87±0.7	85.77±0.5	11896	78.55±0.3	84.78±0.2
	7	4624	87.32±0.4	91.95±0.3	1567	88.02±1.0	93.04±0.4	1684	86.85±0.5	92.0±0.4	8181	85.72±0.2	90.77±0.2
	9	3217	91.77±0.3	95.11±0.2	915	93.23±0.9	96.49±0.6	1060	92.02±0.5	94.63±0.6	5435	90.51±0.3	94.01±0.3
2017	5	5911	85.79±0.3	90.85±0.3	1960	85.3±0.4	91.71±0.5	1703	84.58±0.8	89.72±0.5	9698	84.27±0.4	89.73±0.3
	7	4070	91.72±0.3	95.29±0.2	1105	92.44±0.7	96.12±0.5	979	91.29±0.5	94.84±0.5	6305	90.75±0.3	94.77±0.3
	9	2752	94.72±0.3	97.42±0.1	621	95.44±0.7	97.83±0.3	550	95.75±0.9	96.96±0.8	4072	93.97±0.3	96.93±0.3
2018	5	7830	82.82±0.3	87.1±0.2	3525	76.14±0.6	82.13±0.5	2726	81.06±0.7	85.81±0.5	13983	79.57±0.2	84.37±0.2
	7	6376	87.6±0.5	91.47±0.3	2620	81.14±0.8	86.64±0.5	2083	86.37±0.4	90.39±0.6	11152	84.49±0.3	89.04±0.2
	9	5277	90.44±0.4	93.48±0.2	2054	84.92±0.7	89.25±0.6	1608	89.38±0.7	92.36±0.6	9053	87.89±0.2	91.46±0.3
2019	5	10122	83.07±0.2	86.64±0.3	6281	77.22±0.7	81.41±0.5	3921	79.55±0.3	83.18±0.5	19672	77.93±0.3	81.7±0.3
	7	8087	87.95±0.4	90.98±0.4	4840	82.77±0.3	86.64±0.6	3069	84.65±0.5	88.03±0.4	15973	83.37±0.2	86.96±0.2
	9	6468	90.73±0.3	93.19±0.3	3759	85.74±0.4	89.05±0.5	2373	88.08±0.6	90.8±0.6	12952	86.72±0.3	89.82±0.2
2020	5	16872	79.32±0.3	83.66±0.2	9124	64.93±0.4	70.67±0.3	3982	77.05±0.6	82.26±0.3	29838	72.75±0.2	77.52±0.2
	7	13694	84.9±0.3	88.57±0.2	7130	72.24±0.5	77.86±0.6	3125	82.96±0.2	87.55±0.4	24281	79.2±0.3	83.61±0.2
	9	11143	88.52±0.2	91.54±0.3	5513	77.51±0.4	82.1±0.3	2384	87.22±0.5	91.08±0.5	19529	83.61±0.2	87.25±0.2

Table A.5. LS: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages using obfuscated dataset.

		C++			Python			Java			All Languages		
Year	**Files**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3 Accuracy**
2008	5	1548	73.69±0.8	82.14±0.8	258	76.86±2.1	84.3±2.1	608	67.6±1.7	77.25±0.8	2442	70.94±1.0	80.77±0.6
	7	1064	79.65±0.8	87.3±1.1	113	87.43±2.5	93.01±3.3	306	75.72±1.8	85.46±1.1	1508	77.67±0.7	85.97±0.6
	9	807	83.48±0.7	90.26±0.7	39	93.08±3.8	95.13±2.4	176	80.17±2.8	90.97±1.7	1043	81.7±1.0	89.44±0.4
2009	5	2407	68.89±0.5	77.96±0.6	399	70.53±1.2	81.8±1.2	790	60.35±1.2	72.53±1.4	3644	66.23±0.8	75.68±0.7
	7	1735	76.73±1.0	85.34±0.5	225	83.29±2.5	91.73±1.3	495	71.13±1.8	81.39±1.6	2526	75.32±0.7	84.21±0.4
	9	1111	83.64±0.7	90.95±0.7	100	88.7±3.3	94.4±2.7	282	79.15±2.0	88.87±1.6	1548	83.6±0.5	89.7±0.6
2010	5	2578	76.01±0.5	84.18±0.7	523	79.58±1.3	87.13±1.5	856	72.43±1.3	82.97±0.9	4094	72.21±0.6	80.65±0.5
	7	1799	82.7±0.8	90.01±0.6	310	85.35±1.7	92.68±0.6	567	80.32±0.9	89.74±1.2	2852	78.98±0.7	86.51±0.7
	9	1094	87.8±0.7	93.24±0.8	150	90.6±1.8	96.33±1.7	336	87.86±1.8	94.88±0.9	1771	85.09±0.8	90.53±0.5
2011	5	3794	79.25±0.4	86.67±0.6	789	78.97±1.1	88.14±1.0	1375	67.11±1.1	77.61±0.8	6018	75.35±0.3	83.88±0.3
	7	2902	85.11±0.6	91.67±0.6	491	86.54±1.5	93.91±1.1	946	74.88±1.0	84.21±0.8	4423	81.99±0.4	89.49±0.2
	9	2018	89.69±0.3	94.49±0.4	271	91.44±1.7	96.09±0.9	609	81.95±1.4	90.62±0.9	2986	87.81±0.7	93.23±0.4
2012	5	3360	76.33±0.5	84.12±0.6	786	79.39±1.2	87.42±0.7	1253	66.34±0.9	74.81±0.9	5530	72.74±0.5	80.87±0.6
	7	2136	83.6±0.7	90.31±0.5	361	85.57±1.7	92.58±1.0	618	73.92±0.7	83.69±1.1	3234	80.68±0.7	87.65±0.3
	9	1258	88.65±0.5	93.68±0.5	168	89.64±3.2	94.88±2.1	338	82.28±1.3	90.27±1.3	1844	86.17±0.6	92.18±0.4
2013	5	4641	77.33±0.5	84.85±0.5	1316	75.68±1.2	84.03±1.1	1956	66.53±0.7	75.3±1.0	8137	71.91±0.4	80.12±0.2
	7	2965	84.43±0.6	90.77±0.4	694	83.92±1.0	91.21±0.7	1076	75.32±0.9	84.15±1.0	5004	80.05±0.4	86.9±0.4
	9	1831	89.43±0.6	94.22±0.4	334	89.37±1.8	94.52±0.7	595	82.47±1.4	89.31±1.2	2997	85.37±0.7	91.35±0.6
2014	5	5242	77.57±0.3	85.1±0.5	1531	73.66±0.9	82.95±0.7	1937	66.49±0.8	75.58±0.7	8809	72.93±0.4	81.47±0.3
	7	3344	84.93±0.5	91.5±0.4	782	81.68±0.7	89.3±0.7	1037	75.01±1.4	83.59±1.1	5299	80.96±0.4	88.21±0.4
	9	2195	89.74±0.4	94.36±0.4	386	86.45±1.7	93.78±1.3	596	81.14±1.6	89.26±0.9	3304	86.52±0.6	91.97±0.2
2015	5	4690	80.93±0.2	87.53±0.4	1103	78.67±1.0	86.77±0.6	1344	70.83±1.2	80.23±1.0	7208	77.27±0.3	84.76±0.3
	7	3373	85.99±0.6	91.57±0.4	665	84.66±0.7	91.55±1.1	829	77.49±1.0	85.77±0.7	4995	83.31±0.4	89.79±0.5
	9	2307	89.69±0.4	94.37±0.4	382	89.58±1.6	95.18±1.0	500	82.26±1.6	91.32±1.3	3309	87.73±0.5	93.1±0.2
2016	5	6623	77.1±0.4	84.38±0.3	2914	61.82±0.6	72.95±0.6	2607	61.02±0.5	71.17±0.7	12353	68.23±0.3	76.82±0.4
	7	4730	83.66±0.2	89.99±0.4	1847	71.52±1.2	82.02±0.7	1689	70.17±0.5	79.89±0.6	8599	76.51±0.4	84.06±0.4
	9	3314	88.48±0.6	93.21±0.4	1125	78.65±0.7	87.31±1.3	1058	77.88±1.4	86.83±0.6	5801	82.51±0.5	88.97±0.4
2017	5	5929	82.66±0.2	88.84±0.5	2095	70.65±0.7	81.13±0.8	1701	67.23±1.0	77.57±0.8	9858	76.04±0.2	83.88±0.3
	7	4088	88.72±0.4	93.86±0.3	1190	79.97±0.8	88.54±0.5	976	77.88±0.9	87.16±1.1	6411	84.29±0.3	90.51±0.3
	9	2761	92.39±0.4	96.12±0.3	672	85.51±1.2	91.88±0.7	548	86.55±0.9	92.5±0.9	4141	89.07±0.3	93.99±0.2
2018	5	7857	80.56±0.4	86.04±0.3	3641	68.35±0.8	76.19±0.6	2720	71.33±0.7	77.64±0.7	14093	74.48±0.3	80.82±0.3
	7	6399	85.49±0.6	90.45±0.4	2737	74.63±0.6	82.17±0.8	2080	76.37±0.8	82.62±0.7	11278	80.04±0.5	85.78±0.1
	9	5293	88.32±0.3	92.58±0.3	2155	78.51±0.6	85.24±0.6	1602	79.88±1.1	85.96±0.7	9164	83.54±0.4	88.7±0.4
2019	5	10581	80.34±0.4	84.97±0.3	7771	63.77±0.6	70.11±0.3	3912	69.3±0.8	74.44±0.7	21554	69.48±0.2	74.88±0.3
	7	8464	85.47±0.4	89.48±0.2	5996	70.52±0.5	76.92±0.4	3059	75.75±0.7	80.81±0.4	17490	76.19±0.3	81.29±0.2
	9	6783	88.99±0.4	91.99±0.1	4658	75.27±0.5	80.8±0.3	2372	80.44±0.7	84.0±0.6	14168	80.54±0.4	84.87±0.3
2020	5	16925	77.48±0.3	82.65±0.2	9471	53.87±0.3	61.6±0.4	3974	63.63±0.4	69.77±0.7	30153	66.58±0.2	72.33±0.3
	7	13743	83.39±0.1	87.8±0.2	7457	61.94±0.5	69.1±0.6	3112	70.81±0.6	77.07±0.8	24644	73.41±0.2	78.94±0.2
	9	11220	87.25±0.3	90.77±0.2	5827	67.63±0.7	74.41±0.4	2377	76.09±0.7	81.64±0.5	19905	78.1±0.3	83.0±0.3

Table A.6. TSH: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages under obfuscated setting.

		C++			Python			Java			All Languages		
Year	**Files**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3 Accuracy**
2008	5	1548	70.51±0.4	78.27±0.7	258	78.41±1.8	86.36±2.0	608	67.06±1.9	74.67±1.7	2442	70.15±0.8	77.24±0.7
	7	1064	77.03±0.9	83.85±0.8	113	86.64±3.3	92.83±1.7	306	74.71±1.8	83.53±1.9	1508	76.17±1.1	83.37±0.8
	9	807	80.62±1.2	87.19±0.8	39	94.1±2.0	97.44±2.8	176	79.83±2.4	88.98±2.6	1043	79.6±1.6	86.62±1.0
2009	5	2407	65.39±0.9	73.5±0.4	399	69.72±1.3	81.6±1.3	790	60.82±1.0	71.56±1.7	3644	63.95±0.6	72.77±0.7
	7	1735	73.23±0.8	81.22±0.3	225	83.78±1.1	90.89±1.2	495	73.6±1.1	82.87±1.2	2526	72.57±0.5	81.17±0.4
	9	1111	80.72±1.0	86.93±0.7	100	86.3±1.9	93.9±3.6	282	81.35±2.6	88.55±1.5	1548	80.28±0.7	87.57±1.0
2010	5	2578	72.11±0.6	79.46±0.8	523	79.73±0.9	87.93±0.9	856	75.13±1.1	83.71±1.0	4094	70.11±0.4	77.7±0.6
	7	1799	79.13±0.7	86.01±0.5	310	85.71±1.3	92.06±1.3	567	82.15±1.1	89.21±1.1	2852	77.58±0.7	84.2±0.5
	9	1094	85.62±0.6	90.67±0.6	150	89.33±1.8	94.87±2.3	336	89.2±1.6	95.0±0.9	1771	82.59±0.6	88.75±0.5
2011	5	3794	73.86±0.2	80.92±0.6	789	78.1±1.0	87.9±0.8	1375	68.35±0.8	76.72±1.1	6018	72.26±0.4	80.03±0.4
	7	2902	80.63±0.6	87.23±0.3	491	85.44±1.1	92.2±0.9	946	75.36±1.3	83.41±0.9	4423	79.14±0.5	86.24±0.3
	9	2018	86.16±0.5	91.27±0.6	271	89.34±1.9	95.57±1.3	609	83.68±1.5	89.16±0.9	2986	85.5±0.5	90.95±0.5
2012	5	3360	72.48±0.9	79.51±0.7	786	77.06±0.9	84.95±0.8	1253	67.12±1.1	75.11±0.9	5530	70.65±0.6	77.9±0.5
	7	2136	80.81±0.3	86.98±0.6	361	84.32±1.7	91.25±1.5	618	75.11±1.1	83.37±1.4	3234	78.39±0.4	85.19±0.3
	9	1258	86.14±0.8	91.16±0.9	168	88.99±1.7	94.4±1.9	338	81.54±1.7	89.38±1.2	1844	84.61±0.4	90.09±0.5
2013	5	4641	73.66±0.5	80.51±0.5	1316	74.67±0.8	83.09±0.7	1956	66.27±0.7	73.83±0.8	8137	69.62±0.4	76.95±0.4
	7	2965	81.42±0.6	87.71±0.5	694	82.55±0.8	89.77±1.0	1076	75.42±1.2	82.98±0.8	5004	77.96±0.3	84.73±0.3
	9	1831	86.88±0.8	92.04±0.5	334	88.77±1.6	93.23±0.6	595	81.14±0.9	88.64±1.1	2997	84.07±0.4	89.58±0.4
2014	5	5242	73.29±0.3	80.04±0.4	1531	72.93±1.0	82.31±0.5	1937	65.56±0.8	74.02±1.0	8809	70.3±0.4	78.05±0.4
	7	3344	80.99±0.2	87.38±0.4	782	80.06±1.1	87.92±0.9	1037	73.98±1.1	82.71±1.2	5299	78.39±0.4	85.3±0.3
	9	2195	86.36±0.6	91.5±0.6	386	85.93±1.6	92.46±1.7	596	80.81±0.8	88.17±1.1	3304	84.09±0.5	89.85±0.6
2015	5	4690	76.87±0.3	83.5±0.5	1103	77.09±1.2	85.11±1.1	1344	70.58±0.8	78.81±0.7	7208	74.36±0.3	81.82±0.2
	7	3373	82.15±0.5	88.58±0.6	665	84.05±0.9	91.07±0.7	829	77.47±0.5	85.89±1.0	4995	80.42±0.5	87.28±0.3
	9	2307	86.57±0.6	91.86±0.5	382	88.61±1.3	94.11±1.1	500	83.38±1.2	90.1±1.3	3309	85.25±0.6	90.87±0.3
2016	5	6623	72.54±0.4	79.55±0.4	2914	60.45±0.5	70.79±0.8	2607	57.82±0.6	66.7±0.4	12353	64.62±0.3	72.96±0.2
	7	4730	79.78±0.5	86.16±0.5	1847	68.79±0.7	79.23±0.6	1689	67.67±1.4	76.02±0.9	8599	72.9±0.4	80.86±0.3
	9	3314	85.37±0.6	90.79±0.3	1125	75.88±1.1	85.31±1.2	1058	75.54±0.7	83.44±1.1	5801	79.37±0.3	86.11±0.5
2017	5	5929	77.44±0.3	83.87±0.3	2095	67.26±0.9	77.79±0.8	1701	64.4±0.8	72.56±0.7	9858	71.81±0.3	79.37±0.2
	7	4088	84.59±0.2	90.24±0.3	1190	77.22±0.9	86.32±1.1	976	75.74±0.9	83.15±1.0	6411	80.75±0.4	87.34±0.2
	9	2761	89.33±0.6	93.9±0.4	672	82.23±0.8	89.91±1.2	548	83.54±1.4	89.54±1.3	4141	86.01±0.4	91.54±0.4
2018	5	7857	74.46±0.4	79.8±0.3	3641	57.53±0.5	66.49±0.3	2720	58.9±1.1	66.29±0.8	14093	66.01±0.3	72.39±0.3
	7	6399	80.2±0.3	85.38±0.2	2737	63.45±0.5	73.27±0.5	2080	65.22±0.8	73.38±0.9	11278	71.99±0.4	78.72±0.3
	9	5293	83.49±0.6	88.07±0.3	2155	67.94±0.9	77.24±0.7	1602	70.26±1.0	77.27±1.0	9164	76.48±0.4	82.46±0.4
2019	5	10581	74.9±0.4	79.72±0.3	7771	51.74±0.5	60.16±0.4	3912	57.63±0.6	64.21±0.5	21554	60.87±0.3	67.17±0.3
	7	8464	81.06±0.4	85.37±0.4	5996	58.96±0.6	67.33±0.4	3059	65.2±0.8	71.64±0.7	17490	67.89±0.3	74.08±0.2
	9	6783	85.06±0.2	88.83±0.2	4658	63.94±0.7	71.51±0.5	2372	69.97±0.5	75.4±0.7	14168	72.85±0.2	78.16±0.2
2020	5	16925	72.59±0.2	77.71±0.3	9471	42.84±0.4	51.4±0.5	3974	52.44±0.8	59.96±0.4	30153	58.95±0.2	65.2±0.3
	7	13743	78.63±0.2	83.57±0.3	7457	49.51±0.4	58.8±0.5	3112	60.21±1.2	67.75±0.5	24644	65.57±0.3	72.24±0.3
	9	11220	82.98±0.3	87.2±0.3	5827	55.19±0.5	63.83±0.6	2377	66.28±0.8	73.45±0.6	19905	70.76±0.2	76.97±0.2

Table A.7. TH: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages under obfuscated setting.

		C++			Python			Java			All Languages		
Year	**Files**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3 Accuracy**
2008	5	1548	69.3±0.6	76.82±0.6	258	73.88±1.7	84.53±2.0	608	64.61±2.0	74.62±1.7	2442	68.2±0.6	75.97±0.7
	7	1064	74.59±1.0	82.99±0.7	113	83.45±2.1	91.59±1.9	306	71.54±1.2	82.12±2.0	1508	74.68±0.8	82.63±1.1
	9	807	79.53±1.2	86.8±0.6	39	91.28±3.7	96.67±2.8	176	79.03±2.2	88.01±1.9	1043	78.24±1.1	86.37±1.1
2009	5	2407	63.96±0.4	72.66±0.7	399	66.54±1.6	78.3±1.3	790	60.25±1.5	69.16±1.2	3644	62.45±0.5	71.03±0.4
	7	1735	72.48±0.5	80.32±1.1	225	79.02±2.3	89.33±2.2	495	71.8±1.2	80.46±1.6	2526	71.32±0.9	79.73±0.6
	9	1111	79.54±0.9	87.03±0.7	100	85.1±3.3	91.8±2.1	282	80.6±2.1	87.94±1.8	1548	79.92±0.8	86.61±0.7
2010	5	2578	70.8±0.8	78.09±0.4	523	75.81±1.7	85.32±1.7	856	73.87±0.8	81.83±1.2	4094	68.74±0.6	76.67±0.5
	7	1799	78.48±0.9	85.28±0.5	310	83.13±1.5	90.45±1.5	567	80.21±1.5	88.24±1.4	2852	75.55±0.7	82.89±0.5
	9	1094	85.16±0.8	90.06±0.5	150	87.33±3.1	94.2±1.5	336	87.98±1.2	93.93±1.5	1771	82.15±0.6	87.37±0.8
2011	5	3794	72.74±0.5	79.98±0.6	789	75.27±1.9	84.64±1.0	1375	64.87±0.9	74.8±0.6	6018	70.27±0.4	78.62±0.5
	7	2902	79.3±0.6	86.36±0.6	491	82.48±1.0	90.77±1.4	946	73.11±1.1	81.97±0.9	4423	77.55±0.4	85.33±0.6
	9	2018	84.62±0.4	91.03±0.6	271	87.31±1.9	93.8±1.2	609	81.86±0.7	89.08±0.6	2986	83.59±0.4	90.42±0.5
2012	5	3360	71.36±0.9	78.32±0.5	786	74.03±1.3	83.75±1.1	1253	65.02±0.6	73.27±1.3	5530	69.26±0.4	76.72±0.3
	7	2136	79.41±0.5	85.81±0.8	361	82.71±1.4	90.17±1.9	618	73.25±0.9	80.71±1.6	3234	77.31±0.4	84.16±0.3
	9	1258	85.06±1.1	90.5±0.9	168	88.39±2.0	94.4±1.7	338	81.51±2.2	88.67±1.0	1844	83.34±0.5	89.06±0.7
2013	5	4641	72.16±0.6	79.14±0.4	1316	70.15±0.4	80.38±1.2	1956	65.19±1.0	72.46±0.9	8137	67.92±0.4	75.59±0.4
	7	2965	79.9±0.5	86.89±0.3	694	80.48±1.4	88.54±1.5	1076	73.98±0.5	80.73±0.7	5004	76.25±0.7	83.18±0.5
	9	1831	85.7±0.8	91.43±0.5	334	85.27±1.4	91.56±1.3	595	80.35±1.0	86.54±0.9	2997	82.53±0.4	88.64±0.5
2014	5	5242	71.94±0.4	79.15±0.4	1531	68.47±0.7	79.83±0.5	1937	62.9±0.7	72.13±0.6	8809	68.07±0.4	76.46±0.4
	7	3344	79.72±0.6	86.2±0.4	782	77.53±1.4	86.8±1.0	1037	72.47±1.2	80.98±1.2	5299	76.47±0.3	84.06±0.2
	9	2195	85.24±0.7	90.62±0.4	386	83.78±1.6	91.17±1.4	596	79.35±1.6	86.74±1.3	3304	82.54±0.6	88.77±0.5
2015	5	4690	75.69±0.6	82.01±0.3	1103	73.99±1.3	83.54±1.0	1344	68.35±1.1	76.47±0.7	7208	72.47±0.5	80.15±0.2
	7	3373	81.1±0.4	87.21±0.4	665	80.99±1.2	89.29±1.0	829	75.04±1.1	83.99±0.7	4995	78.58±0.5	85.83±0.3
	9	2307	85.61±0.6	91.31±0.5	382	86.6±1.4	93.43±0.9	500	82.02±1.0	88.56±0.8	3309	83.95±0.5	89.97±0.3
2016	5	6623	71.18±0.5	78.1±0.6	2914	55.64±0.5	67.31±0.9	2607	55.54±1.1	64.09±0.6	12353	62.32±0.3	70.84±0.4
	7	4730	78.32±0.4	84.81±0.5	1847	64.87±0.6	76.48±0.8	1689	64.77±0.6	73.78±0.6	8599	70.61±0.4	79.08±0.3
	9	3314	83.52±0.5	89.57±0.4	1125	72.55±0.8	82.89±1.2	1058	72.8±1.3	81.57±1.0	5801	77.46±0.4	84.67±0.3
2017	5	5929	75.74±0.5	82.47±0.4	2095	64.81±0.8	75.85±0.7	1701	62.1±0.8	69.9±0.6	9858	69.6±0.3	77.62±0.2
	7	4088	83.3±0.5	89.12±0.5	1190	74.24±0.7	84.65±0.8	976	74.17±0.9	81.07±0.9	6411	79.02±0.4	85.89±0.3
	9	2761	88.01±0.4	92.77±0.5	672	80.54±1.3	88.44±1.2	548	81.17±1.2	88.05±1.2	4141	84.41±0.4	90.73±0.3
2018	5	7857	72.77±0.3	78.25±0.5	3641	54.77±0.6	64.02±1.0	2720	55.56±1.0	64.09±0.8	14093	63.7±0.3	70.71±0.3
	7	6399	78.81±0.4	83.91±0.4	2737	60.56±0.7	70.33±0.7	2080	61.87±0.6	70.78±1.0	11278	70.13±0.4	76.86±0.1
	9	5293	82.17±0.4	86.88±0.5	2155	65.94±0.9	74.54±0.7	1602	67.4±1.0	74.7±1.0	9164	74.65±0.5	80.91±0.4
2019	5	10581	73.6±0.4	78.7±0.3	7771	48.05±0.4	56.37±0.6	3912	54.78±0.6	62.21±0.6	21554	58.54±0.3	65.07±0.3
	7	8464	79.81±0.3	84.47±0.2	5996	54.65±0.4	63.92±0.4	3059	61.51±0.6	69.27±0.5	17490	65.76±0.3	72.08±0.3
	9	6783	84.14±0.6	87.99±0.4	4658	59.88±0.7	68.59±0.6	2372	66.45±1.0	73.39±0.8	14168	70.5±0.4	76.26±0.3
2020	5	16925	71.05±0.3	76.19±0.3	9471	39.12±0.4	47.79±0.6	3974	49.26±0.6	57.32±0.9	30153	56.66±0.2	63.05±0.2
	7	13743	77.64±0.4	82.44±0.2	7457	45.73±0.5	55.11±0.5	3112	57.33±0.5	65.58±1.0	24644	63.34±0.2	70.07±0.2
	9	11220	81.96±0.2	86.15±0.3	5827	50.94±0.5	60.22±0.5	2377	63.49±0.9	71.16±0.9	19905	68.59±0.4	74.79±0.2

Table A.8. TH-soft: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages under obfuscated setting.

		C++			Python			Java			All Languages		
Year	**Files**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3 Accuracy**
2008	5	1548	71.25±1.0	79.04±0.9	258	80.08±2.4	88.26±1.2	608	66.74±1.5	75.38±1.6	2442	70.01±0.4	77.88±0.9
	7	1064	77.65±1.0	84.72±0.8	113	88.58±2.6	95.66±1.2	306	73.86±2.2	83.4±1.3	1508	76.37±0.6	84.64±0.7
	9	807	81.64±1.2	87.98±1.0	39	92.05±4.0	95.9±2.1	176	80.45±1.6	89.09±2.4	1043	80.98±0.9	87.73±0.9
2009	5	2407	65.77±0.6	73.84±0.7	399	72.36±2.0	82.76±1.3	790	60.06±1.2	68.13±0.9	3644	63.92±0.8	72.72±0.5
	7	1735	73.8±0.9	81.86±0.9	225	85.33±2.1	92.49±1.4	495	70.51±1.1	80.46±1.8	2526	72.91±1.1	81.5±0.8
	9	1111	82.15±0.9	87.66±0.9	100	89.0±3.0	93.7±2.4	282	80.57±2.0	87.38±1.3	1548	81.74±0.9	87.8±0.4
2010	5	2578	74.88±0.8	81.67±0.6	523	82.62±2.0	88.91±1.1	856	74.66±1.4	81.89±1.0	4094	71.98±0.6	79.56±0.4
	7	1799	81.86±0.6	87.88±0.5	310	88.03±1.0	93.74±1.3	567	80.9±1.0	89.26±1.0	2852	79.15±0.5	85.72±0.5
	9	1094	87.21±0.8	92.05±0.7	150	93.13±1.7	96.87±0.9	336	88.78±1.2	94.38±1.0	1771	84.6±0.5	89.86±0.6
2011	5	3794	77.0±0.7	84.01±0.3	789	81.03±1.2	89.58±0.6	1375	67.4±0.9	76.57±0.9	6018	74.47±0.3	82.21±0.3
	7	2902	83.17±0.7	89.59±0.5	491	87.68±1.4	93.52±0.8	946	75.5±0.8	83.69±1.7	4423	81.54±0.6	88.03±0.3
	9	2018	88.49±0.7	93.04±0.4	271	92.32±1.2	96.57±0.9	609	82.79±0.8	90.57±1.2	2986	87.05±0.5	92.12±0.3
2012	5	3360	74.51±0.5	81.43±0.7	786	80.53±1.1	87.34±1.3	1253	66.42±0.8	74.13±0.6	5530	72.48±0.5	78.99±0.5
	7	2136	81.98±0.6	88.68±0.6	361	85.68±1.5	93.16±0.7	618	74.13±1.5	81.83±0.9	3234	79.85±0.9	86.54±0.6
	9	1258	87.5±0.7	92.73±0.5	168	92.38±2.1	94.52±1.3	338	81.48±1.7	88.49±0.8	1844	85.77±0.9	91.16±0.5
2013	5	4641	76.4±0.4	83.01±0.6	1316	76.93±1.2	85.2±0.9	1956	64.89±1.1	73.02±0.6	8137	71.38±0.3	78.85±0.3
	7	2965	84.07±0.5	89.88±0.4	694	85.13±0.8	91.97±1.0	1076	74.23±1.3	82.34±1.0	5004	79.65±0.5	86.04±0.3
	9	1831	88.77±0.6	93.6±0.6	334	89.94±1.5	93.86±0.7	595	80.34±0.9	86.57±0.8	2997	85.52±0.5	90.24±0.5
2014	5	5242	76.47±0.3	82.95±0.4	1531	74.76±0.6	84.15±0.4	1937	65.14±1.1	74.18±0.7	8809	72.5±0.4	80.0±0.3
	7	3344	83.56±0.4	89.26±0.3	782	82.81±1.2	89.78±1.1	1037	74.26±1.1	82.64±1.0	5299	80.08±0.4	87.01±0.3
	9	2195	88.51±0.5	92.93±0.3	386	88.08±1.6	93.81±1.1	596	80.47±1.1	87.89±1.2	3304	85.74±0.4	91.23±0.3
2015	5	4690	79.71±0.6	85.65±0.3	1103	79.21±1.1	87.41±1.1	1344	70.16±0.9	78.36±0.8	7208	76.83±0.4	83.62±0.3
	7	3373	85.48±0.7	90.75±0.3	665	85.19±1.1	91.92±0.8	829	76.95±1.3	85.97±0.7	4995	82.57±0.4	88.8±0.3
	9	2307	89.09±0.4	93.5±0.4	382	89.48±1.2	95.08±0.9	500	81.4±1.0	89.26±1.1	3309	87.34±0.5	92.15±0.4
2016	5	6623	75.29±0.4	81.9±0.3	2914	63.12±0.9	73.7±0.7	2607	56.85±0.5	66.44±0.8	12353	66.63±0.3	74.79±0.2
	7	4730	82.0±0.3	88.16±0.2	1847	72.06±0.9	81.71±1.0	1689	66.48±0.8	76.2±1.2	8599	74.82±0.4	82.38±0.4
	9	3314	86.96±0.4	92.04±0.4	1125	79.64±0.9	87.8±0.6	1058	75.19±1.3	83.02±0.7	5801	81.22±0.5	87.71±0.5
2017	5	5929	80.26±0.2	86.33±0.3	2095	69.75±1.0	79.88±0.8	1701	65.88±1.1	75.28±0.7	9858	74.23±0.4	81.57±0.2
	7	4088	86.97±0.5	92.19±0.3	1190	79.08±1.3	87.1±0.7	976	77.15±1.2	84.85±0.8	6411	82.82±0.4	89.0±0.3
	9	2761	91.13±0.6	95.25±0.2	672	84.88±0.9	91.38±1.1	548	84.74±1.3	90.53±1.2	4141	88.07±0.3	93.01±0.4
2018	5	7857	75.98±0.4	81.14±0.4	3641	63.89±0.7	71.88±0.3	2720	65.15±0.8	71.54±0.6	14093	69.51±0.4	75.53±0.5
	7	6399	81.55±0.4	86.6±0.6	2737	69.52±0.8	77.61±0.7	2080	71.74±0.8	77.96±0.6	11278	75.95±0.2	81.42±0.3
	9	5293	84.96±0.4	89.2±0.4	2155	74.33±0.7	81.41±0.9	1602	76.0±0.8	81.02±0.7	9164	79.36±0.3	84.78±0.3
2019	5	10581	76.68±0.3	81.29±0.3	7771	60.4±0.3	67.0±0.4	3912	63.49±0.6	69.46±0.5	21554	65.89±0.4	71.07±0.3
	7	8464	82.45±0.3	86.71±0.3	5996	67.58±0.6	73.86±0.5	3059	70.55±0.7	76.21±0.5	17490	72.69±0.3	77.93±0.3
	9	6783	86.53±0.3	89.57±0.3	4658	72.07±0.7	77.71±0.5	2372	75.47±0.6	79.89±1.1	14168	77.33±0.4	81.81±0.3
2020	5	16925	74.08±0.4	78.9±0.3	9471	50.95±0.4	58.69±0.4	3974	56.63±0.6	63.25±0.5	30153	62.88±0.2	68.83±0.1
	7	13743	80.25±0.3	85.08±0.2	7457	59.04±0.5	66.2±0.4	3112	64.11±0.7	71.53±0.5	24644	69.87±0.2	75.57±0.2
	9	11220	84.53±0.3	88.36±0.3	5827	64.28±0.5	71.3±0.5	2377	70.03±0.6	76.41±0.6	19905	74.71±0.2	79.99±0.3

Table A.9. LS: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages under simulated real-world constraint.

		C++			Python			Java			All Languages		
Year	**Files**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**
2008	5	2204	79.64±0.5	87.23±0.5	473	81.69±1.9	88.73±1.1	1109	64.94±1.1	77.39±1.1	3777	75.2±0.7	83.96±0.5
	7	1878	84.68±0.8	91.41±0.7	367	89.13±1.4	94.03±1.1	836	74.84±1.5	85.25±1.3	3097	81.49±0.5	89.65±0.3
	9	1548	87.76±0.7	93.17±0.7	257	91.17±1.2	95.1±1.1	609	79.93±1.3	88.23±1.1	2442	85.98±0.6	92.04±0.6
2009	5	3161	75.51±0.8	83.77±0.6	636	77.81±1.5	86.34±1.7	1236	61.15±1.9	74.35±1.4	5016	71.18±0.4	80.77±0.4
	7	2766	82.32±0.7	89.45±0.5	503	83.76±1.4	91.29±1.1	988	69.23±1.7	81.47±1.0	4273	78.72±0.7	86.89±0.6
	9	2407	86.4±0.6	91.95±0.4	395	88.0±1.1	93.29±1.3	791	74.84±1.2	85.35±1.3	3642	83.37±0.6	89.66±0.4
2010	5	3636	80.09±0.6	87.82±0.2	826	84.96±1.0	91.62±0.8	1355	72.02±1.1	83.04±1.3	5790	76.42±0.6	84.63±0.5
	7	3018	85.67±0.4	92.23±0.3	636	89.48±0.7	94.73±0.8	1044	79.86±1.0	88.72±0.4	4772	82.04±0.8	89.31±0.3
	9	2559	88.81±0.5	93.99±0.4	500	91.26±1.2	96.02±0.9	862	84.32±1.1	91.65±0.9	4061	85.79±0.4	91.62±0.5
2011	5	4782	82.34±0.5	89.78±0.4	1218	84.15±1.1	91.61±0.5	1943	69.36±0.7	80.45±0.8	7880	78.7±0.6	86.94±0.4
	7	4301	87.11±0.4	93.59±0.3	990	89.55±0.8	95.18±0.8	1673	75.73±0.8	86.13±0.8	6960	84.28±0.4	91.28±0.2
	9	3792	90.04±0.4	95.1±0.4	780	91.71±0.8	96.22±0.7	1375	80.42±0.9	88.72±0.8	6012	87.69±0.5	93.39±0.3
2012	5	5339	80.61±0.3	88.18±0.4	1799	81.66±1.4	89.55±0.5	2678	63.43±0.4	75.2±0.8	9812	74.96±0.4	83.76±0.2
	7	4134	85.87±0.7	92.33±0.3	1166	88.05±0.9	93.77±0.7	1770	73.77±0.8	84.03±0.6	7155	82.4±0.5	89.58±0.4
	9	3359	89.18±0.5	94.3±0.3	780	91.36±0.7	95.59±0.6	1253	79.89±1.0	88.03±0.8	5518	86.58±0.3	92.25±0.4
2013	5	7139	80.74±0.6	88.33±0.3	2508	79.36±0.7	88.03±0.4	3487	64.45±0.5	74.92±0.4	13069	74.64±0.4	83.14±0.4
	7	5773	86.27±0.5	92.66±0.2	1796	86.44±0.5	92.4±0.5	2577	73.33±1.0	83.07±0.4	10269	81.52±0.3	88.86±0.3
	9	4640	89.66±0.4	94.61±0.3	1309	90.05±0.7	94.35±0.6	1959	78.9±0.8	87.87±0.6	8133	85.62±0.3	91.56±0.2
2014	5	8529	78.99±0.4	87.48±0.2	2945	79.22±0.5	88.19±0.4	3690	65.59±0.6	76.54±0.4	15105	75.17±0.3	84.24±0.2
	7	6552	85.58±0.3	92.21±0.2	2080	84.4±0.9	92.07±0.3	2619	73.14±0.5	83.72±0.6	11301	81.77±0.3	89.37±0.2
	9	5238	89.61±0.5	94.45±0.3	1511	88.16±0.8	93.98±0.6	1939	79.63±1.0	87.35±0.5	8791	86.43±0.2	92.33±0.2
2015	5	7502	83.76±0.3	90.38±0.4	2323	83.45±0.5	90.82±0.3	2894	70.55±0.8	80.22±0.7	12673	79.88±0.3	87.39±0.2
	7	5774	87.85±0.3	93.71±0.4	1493	87.32±0.5	93.7±0.3	1831	77.44±0.5	86.0±0.5	9110	84.98±0.4	91.46±0.3
	9	4674	90.59±0.3	95.22±0.3	1074	89.99±0.8	94.96±0.6	1346	81.29±0.6	89.64±0.7	7166	88.11±0.3	93.31±0.2
2016	5	9366	81.22±0.4	88.1±0.4	4340	69.53±0.5	79.74±0.6	4067	63.25±0.5	73.97±0.6	17592	73.02±0.2	81.51±0.2
	7	8077	86.5±0.4	92.39±0.3	3553	76.92±0.7	85.92±0.5	3366	71.68±0.5	81.36±0.7	15042	79.7±0.4	87.27±0.3
	9	6607	89.48±0.4	94.07±0.2	2794	81.55±0.7	89.08±0.4	2607	76.41±0.8	85.39±0.7	12239	83.7±0.3	89.96±0.2
2017	5	8923	84.42±0.3	90.87±0.3	3702	76.73±0.6	86.4±0.3	3122	65.98±0.6	77.36±0.5	15716	77.94±0.2	86.4±0.2
	7	7315	89.17±0.3	94.51±0.2	2750	83.25±0.8	90.59±0.5	2366	75.7±0.4	85.25±0.9	12529	84.21±0.3	91.36±0.2
	9	5929	92.2±0.4	96.23±0.2	2061	86.79±0.3	92.93±0.3	1703	81.34±0.8	89.69±0.3	9831	88.32±0.2	93.83±0.2
2018	5	10090	82.75±0.4	89.03±0.3	4885	74.76±0.5	82.94±0.4	3734	68.29±0.6	77.17±0.6	18231	76.78±0.4	84.06±0.3
	7	8852	87.5±0.4	92.52±0.2	4169	80.54±0.5	87.35±0.4	3152	75.49±0.8	83.44±0.7	15922	82.24±0.3	88.49±0.2
	9	7850	89.9±0.2	94.06±0.2	3611	83.41±0.3	89.68±0.4	2726	79.56±0.6	86.39±0.5	14070	85.56±0.3	90.64±0.2
2019	5	13523	80.15±0.3	85.91±0.3	9597	68.22±0.4	75.63±0.4	5152	64.0±0.5	71.3±0.6	26224	70.36±0.2	77.05±0.2
	7	11832	85.28±0.2	90.41±0.3	8236	74.92±0.4	81.49±0.4	4475	71.52±0.6	78.14±0.6	23321	76.8±0.2	83.01±0.2
	9	10432	88.62±0.3	92.29±0.3	7123	79.21±0.3	84.37±0.4	3925	76.54±0.6	82.0±0.2	20833	81.13±0.3	85.83±0.2
2020	5	20677	76.54±0.3	83.14±0.2	11800	56.43±0.5	65.73±0.2	5032	59.26±0.5	67.34±0.4	36157	66.37±0.2	73.95±0.1
	7	18818	82.2±0.1	87.98±0.2	10499	63.95±0.4	72.82±0.3	4510	66.84±0.3	74.96±0.6	33174	73.02±0.2	80.05±0.1
	9	16916	85.98±0.3	90.44±0.2	9373	69.3±0.4	77.25±0.3	3985	72.78±0.5	79.58±0.7	30084	77.29±0.2	83.3±0.1

Table A.10. TSH: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages under simulated real-world constraint.

		C++			Python			Java			All Languages		
Year	**Files**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**
2008	5	2204	73.27±0.7	82.27±0.8	473	80.15±1.2	89.34±0.9	1109	58.46±0.8	69.97±1.1	3777	69.37±0.7	78.58±0.4
	7	1878	78.46±0.7	86.99±0.4	367	88.56±1.2	93.65±1.1	836	69.23±1.1	79.58±0.9	3097	76.7±0.6	85.22±0.5
	9	1548	82.29±0.8	88.93±1.0	257	89.96±1.3	94.79±1.2	609	75.89±1.0	83.84±1.3	2442	81.27±0.9	87.8±0.5
2009	5	3161	67.61±0.7	77.25±0.6	636	76.26±1.1	86.04±1.2	1236	56.84±0.7	68.71±1.1	5016	65.45±0.5	75.55±0.6
	7	2766	74.19±0.8	83.06±0.6	503	83.32±1.5	90.48±0.6	988	64.86±1.7	77.26±1.2	4273	72.74±0.5	81.92±0.2
	9	2407	78.75±0.6	85.75±0.6	395	87.22±1.4	92.15±1.8	791	72.25±1.2	82.23±0.9	3642	77.28±0.5	84.82±0.4
2010	5	3636	72.21±0.6	81.09±0.4	826	84.07±1.1	90.77±1.0	1355	68.07±0.8	77.65±0.5	5790	70.4±0.6	79.02±0.3
	7	3018	78.74±0.6	86.75±0.5	636	88.21±0.8	93.82±0.9	1044	77.23±1.0	85.59±0.6	4772	76.64±0.5	84.58±0.3
	9	2559	82.51±0.8	89.11±0.5	500	90.7±1.4	95.78±0.7	862	81.61±0.7	89.15±1.0	4061	80.76±0.5	87.47±0.4
2011	5	4782	74.31±0.5	82.7±0.4	1218	82.95±1.1	90.59±0.9	1943	63.46±0.7	73.49±1.0	7880	71.85±0.4	80.68±0.3
	7	4301	79.52±0.4	87.25±0.4	990	87.34±0.9	93.64±0.8	1673	71.46±1.0	80.87±0.7	6960	78.14±0.5	86.14±0.3
	9	3792	82.9±0.4	89.51±0.4	780	90.09±0.9	95.22±0.5	1375	76.63±1.1	84.23±0.7	6012	81.91±0.7	88.68±0.4
2012	5	5339	72.44±0.3	81.32±0.3	1799	79.77±0.8	87.84±0.5	2678	56.28±1.3	66.65±0.9	9812	68.48±0.4	77.43±0.2
	7	4134	78.66±0.4	86.65±0.3	1166	85.1±0.6	92.68±0.6	1770	67.71±0.9	77.93±0.8	7155	76.2±0.5	84.62±0.2
	9	3359	82.63±0.5	89.57±0.4	780	89.26±1.1	94.59±0.7	1253	75.32±1.1	83.07±0.6	5518	81.42±0.4	88.12±0.4
2013	5	7139	72.39±0.5	81.11±0.2	2508	76.87±0.5	85.55±0.7	3487	56.93±0.7	66.62±0.5	13069	67.55±0.3	76.52±0.2
	7	5773	78.74±0.4	86.61±0.3	1796	83.13±0.6	91.0±0.8	2577	67.18±0.7	75.88±0.7	10269	75.26±0.3	83.32±0.4
	9	4640	83.22±0.4	89.79±0.5	1309	87.25±0.9	92.84±0.8	1959	73.86±0.7	81.31±0.8	8133	80.04±0.3	86.88±0.1
2014	5	8529	70.15±0.3	79.38±0.2	2945	77.05±0.7	86.3±0.5	3690	58.85±0.6	68.54±0.5	15105	68.04±0.3	77.32±0.2
	7	6552	77.79±0.4	85.83±0.4	2080	82.56±0.6	90.16±0.7	2619	67.94±0.6	77.79±0.6	11301	75.6±0.2	84.0±0.2
	9	5238	83.04±0.2	89.41±0.2	1511	85.59±0.7	92.38±0.4	1939	74.82±0.6	82.33±0.6	8791	80.82±0.5	87.74±0.3
2015	5	7502	76.33±0.5	83.99±0.2	2323	80.63±0.8	88.42±0.6	2894	64.84±0.6	73.84±0.5	12673	73.68±0.2	81.9±0.4
	7	5774	81.36±0.6	88.93±0.5	1493	84.55±0.5	92.06±0.6	1831	72.91±0.4	81.47±0.9	9110	79.4±0.3	87.14±0.3
	9	4674	84.85±0.5	91.11±0.4	1074	87.72±0.8	93.71±0.6	1346	78.86±1.0	86.03±0.9	7166	83.38±0.2	89.75±0.2
2016	5	9366	72.18±0.2	80.13±0.3	4340	65.5±0.7	76.42±0.5	4067	54.05±0.6	63.7±0.5	17592	65.13±0.3	74.17±0.4
	7	8077	78.55±0.3	85.8±0.3	3553	72.81±0.5	82.67±0.5	3366	63.37±0.7	72.72±0.6	15042	72.58±0.3	80.91±0.3
	9	6607	82.16±0.5	88.46±0.3	2794	77.64±0.6	85.46±0.8	2607	68.95±0.4	78.11±0.5	12239	76.71±0.5	84.49±0.2
2017	5	8923	74.99±0.4	83.03±0.2	3702	70.99±0.3	81.31±0.7	3122	56.26±0.7	67.26±0.7	15716	69.46±0.2	78.57±0.3
	7	7315	81.33±0.5	88.33±0.2	2750	77.6±0.4	86.85±0.5	2366	67.05±0.8	77.0±0.5	12529	76.87±0.2	85.14±0.2
	9	5929	85.19±0.4	91.39±0.2	2061	82.09±0.5	89.62±0.6	1703	73.81±1.4	81.8±0.6	9831	81.93±0.3	88.64±0.2
2018	5	10090	72.09±0.3	79.35±0.4	4885	61.08±0.5	71.03±0.4	3734	53.71±0.5	61.23±0.7	18231	64.49±0.3	72.56±0.2
	7	8852	78.34±0.3	84.46±0.3	4169	67.74±0.5	76.93±0.3	3152	61.45±0.6	69.49±0.9	15922	71.06±0.2	78.61±0.2
	9	7850	81.77±0.4	87.26±0.4	3611	71.65±0.7	79.6±0.5	2726	66.85±0.7	73.27±0.6	14070	75.34±0.3	81.82±0.4
2019	5	13523	71.32±0.2	77.42±0.3	9597	56.21±0.5	64.46±0.4	5152	52.01±0.7	58.7±0.7	26224	59.82±0.2	66.7±0.2
	7	11832	77.52±0.3	83.31±0.2	8236	63.06±0.5	70.9±0.5	4475	60.05±0.5	66.91±0.8	23321	66.95±0.2	73.73±0.2
	9	10432	81.58±0.4	86.23±0.3	7123	67.83±0.4	74.35±0.4	3925	65.6±0.8	71.56±0.5	20833	71.65±0.3	77.41±0.2
2020	5	20677	67.97±0.3	74.76±0.2	11800	45.79±0.3	54.41±0.5	5032	47.34±0.6	54.98±0.5	36157	56.87±0.3	64.43±0.2
	7	18818	74.25±0.2	80.72±0.2	10499	52.9±0.3	61.86±0.3	4510	55.6±0.4	63.03±0.7	33174	63.63±0.1	70.94±0.1
	9	16916	78.63±0.2	84.01±0.2	9373	57.64±0.3	66.02±0.4	3985	61.59±0.4	68.31±0.5	30084	68.17±0.2	74.75±0.1

Table A.11. TH: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages under simulated real-world constraint.

		C++			Python			Java			All Languages		
Year	**Files**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**
2008	5	2204	72.6±0.8	81.31±0.7	473	79.51±2.1	87.44±1.1	1109	55.88±1.0	67.05±1.5	3777	67.66±0.4	77.23±0.6
	7	1878	78.23±0.7	86.61±0.8	367	85.78±1.9	92.34±1.3	836	66.7±1.4	77.61±1.2	3097	75.26±0.7	83.95±0.5
	9	1548	81.43±1.1	88.54±0.8	257	89.14±1.3	94.12±1.5	609	73.66±1.8	81.9±1.8	2442	79.24±0.8	87.15±0.5
2009	5	3161	66.68±0.5	76.49±0.5	636	73.14±1.5	84.12±1.0	1236	53.41±0.8	65.76±1.3	5016	63.62±0.5	74.12±0.5
	7	2766	73.5±0.4	82.56±0.6	503	79.44±1.1	88.41±1.2	988	62.67±1.2	74.19±1.0	4273	70.86±0.6	80.53±0.4
	9	2407	78.29±0.7	85.92±0.5	395	84.41±1.6	91.97±1.1	791	68.47±1.1	79.63±1.4	3642	75.58±0.6	83.97±0.5
2010	5	3636	71.16±0.5	80.28±0.4	826	80.9±0.9	88.93±0.9	1355	65.72±1.2	76.01±1.3	5790	68.59±0.3	77.63±0.5
	7	3018	77.66±0.6	85.93±0.4	636	85.93±0.9	92.83±0.9	1044	74.03±0.7	83.44±1.2	4772	75.38±0.5	83.34±0.4
	9	2559	81.66±0.7	88.13±0.4	500	88.22±1.5	94.26±1.2	862	80.49±1.7	87.63±0.8	4061	79.41±0.5	86.48±0.4
2011	5	4782	73.03±0.6	81.68±0.6	1218	79.37±1.0	88.02±0.9	1943	60.5±0.7	70.47±1.0	7880	70.3±0.3	79.26±0.3
	7	4301	78.29±0.5	86.82±0.5	990	84.47±0.7	91.75±0.8	1673	68.68±0.7	78.41±0.7	6960	76.44±0.3	84.96±0.4
	9	3792	82.15±0.5	89.0±0.4	780	87.09±1.0	93.85±0.7	1375	73.91±1.1	82.35±0.5	6012	80.65±0.4	87.36±0.4
2012	5	5339	71.77±0.3	80.47±0.4	1799	76.66±0.9	86.01±0.5	2678	54.06±0.7	63.83±0.6	9812	66.66±0.4	76.03±0.2
	7	4134	78.02±0.4	86.04±0.5	1166	83.28±0.8	90.73±0.5	1770	65.73±0.9	75.89±0.5	7155	75.01±0.3	83.13±0.2
	9	3359	82.18±0.6	88.9±0.3	780	87.38±1.1	93.51±0.6	1253	73.34±1.2	81.47±1.1	5518	79.85±0.4	87.24±0.2
2013	5	7139	71.17±0.3	80.16±0.3	2508	73.09±0.7	82.81±0.7	3487	54.36±0.6	64.38±0.5	13069	65.44±0.3	74.71±0.2
	7	5773	77.81±0.4	85.87±0.4	1796	79.85±0.8	88.54±0.7	2577	64.4±0.9	73.39±1.0	10269	73.3±0.5	81.88±0.3
	9	4640	82.18±0.4	88.79±0.5	1309	84.2±0.8	91.15±0.8	1959	71.56±0.9	79.42±0.6	8133	78.41±0.3	85.69±0.2
2014	5	8529	69.47±0.5	78.61±0.4	2945	73.77±0.7	84.01±0.4	3690	55.84±1.0	66.25±0.5	15105	65.98±0.3	75.63±0.2
	7	6552	76.57±0.7	85.02±0.3	2080	78.7±0.8	88.16±0.7	2619	64.38±0.9	75.17±0.8	11301	73.55±0.4	82.43±0.2
	9	5238	82.12±0.5	88.74±0.5	1511	82.42±1.2	90.87±0.6	1939	71.95±1.1	80.35±1.0	8791	79.18±0.3	86.54±0.4
2015	5	7502	75.35±0.4	83.12±0.3	2323	77.68±0.7	86.39±0.5	2894	62.25±0.5	71.43±0.5	12673	71.96±0.3	80.35±0.2
	7	5774	80.27±0.5	87.8±0.3	1493	82.68±0.7	90.74±0.7	1831	71.04±0.8	79.78±1.2	9110	77.73±0.4	85.95±0.3
	9	4674	83.66±0.6	90.07±0.4	1074	85.41±0.6	92.96±0.5	1346	75.92±0.5	83.56±0.7	7166	81.81±0.4	88.7±0.3
2016	5	9366	70.97±0.5	79.39±0.3	4340	61.54±0.4	73.06±0.8	4067	50.6±0.7	60.79±0.7	17592	62.89±0.3	72.47±0.3
	7	8077	77.4±0.3	84.96±0.3	3553	69.14±0.4	79.85±0.5	3366	60.54±0.4	70.56±0.7	15042	70.53±0.4	79.34±0.2
	9	6607	80.79±0.3	87.65±0.3	2794	73.88±0.6	83.33±0.5	2607	65.84±0.7	75.24±0.4	12239	74.77±0.3	82.79±0.3
2017	5	8923	73.16±0.4	81.48±0.3	3702	67.5±0.2	78.68±0.5	3122	53.57±0.6	64.26±0.6	15716	67.06±0.2	76.58±0.2
	7	7315	79.87±0.4	87.33±0.2	2750	74.95±0.6	84.44±0.5	2366	64.64±0.6	74.64±0.6	12529	74.87±0.4	83.55±0.3
	9	5929	83.99±0.3	90.34±0.3	2061	79.33±0.7	87.94±0.5	1703	71.96±0.7	80.27±0.6	9831	80.02±0.2	87.1±0.1
2018	5	10090	70.77±0.4	77.66±0.4	4885	58.67±0.4	68.42±0.5	3734	50.84±0.6	59.3±0.6	18231	62.52±0.2	70.52±0.3
	7	8852	76.77±0.4	83.31±0.4	4169	65.0±0.6	75.03±0.5	3152	59.51±0.6	67.13±0.7	15922	69.06±0.3	76.92±0.3
	9	7850	80.64±0.5	85.75±0.3	3611	68.99±0.7	77.49±0.6	2726	64.33±0.7	71.5±0.7	14070	73.39±0.4	80.17±0.1
2019	5	13523	70.0±0.2	76.39±0.3	9597	52.68±0.4	61.1±0.4	5152	49.88±0.8	56.55±0.6	26224	57.66±0.3	64.72±0.2
	7	11832	76.38±0.3	82.23±0.3	8236	59.67±0.5	68.01±0.4	4475	57.79±0.6	64.14±1.0	23321	64.85±0.2	71.79±0.2
	9	10432	80.56±0.2	85.54±0.2	7123	64.43±0.5	72.02±0.4	3925	63.43±0.4	69.44±0.6	20833	69.62±0.4	75.74±0.3
2020	5	20677	66.54±0.2	73.54±0.2	11800	42.35±0.3	50.79±0.3	5032	45.32±0.9	53.04±0.5	36157	54.84±0.2	62.3±0.2
	7	18818	72.99±0.3	79.38±0.3	10499	49.2±0.3	57.96±0.3	4510	53.0±0.8	60.86±0.4	33174	61.46±0.2	68.88±0.2
	9	16916	77.38±0.1	82.92±0.2	9373	54.29±0.4	62.43±0.5	3985	59.72±0.5	66.7±0.8	30084	66.24±0.2	72.87±0.2

Table A.12. TH-soft: Accuracy of authorship attribution across different number of source code files per author, different years and programming languages under simulated real-world constraint.

		C++			Python			Java			All Languages		
Year	**Files**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**	**Authors**	**Accuracy**	**Top 3**
2008	5	2204	75.77±0.7	83.63±0.6	473	85.71±1.8	91.59±1.1	1109	63.91±1.2	75.59±1.1	3777	72.43±0.6	81.35±0.5
	7	1878	80.91±0.6	88.7±0.6	367	91.53±0.8	95.72±0.8	836	73.31±0.9	82.99±0.9	3097	79.67±0.7	87.57±0.6
	9	1548	84.91±0.6	90.7±0.4	257	93.11±1.2	96.07±0.9	609	78.95±1.2	87.57±0.9	2442	83.42±0.6	89.59±0.5
2009	5	3161	69.82±0.5	78.97±0.4	636	81.27±1.0	88.6±0.7	1236	61.2±1.0	72.97±0.5	5016	67.81±0.4	77.72±0.5
	7	2766	77.0±0.7	85.42±0.7	503	86.1±1.5	92.39±0.8	988	70.29±1.3	81.42±1.2	4273	75.71±0.4	84.17±0.6
	9	2407	81.19±0.4	87.95±0.5	395	89.16±1.2	94.46±0.9	791	75.74±0.6	84.88±1.0	3642	80.12±0.5	87.1±0.3
2010	5	3636	76.12±0.6	84.22±0.5	826	86.46±1.2	92.78±0.8	1355	71.51±1.1	80.6±0.9	5790	73.74±0.4	82.22±0.6
	7	3018	82.46±0.6	89.19±0.2	636	90.35±1.1	95.63±0.6	1044	78.62±0.9	87.71±1.1	4772	79.72±0.5	87.18±0.3
	9	2559	85.58±0.5	91.61±0.5	500	92.26±1.3	96.9±0.8	862	83.91±1.0	90.49±0.9	4061	83.91±0.6	89.6±0.4
2011	5	4782	78.24±0.4	86.45±0.4	1218	86.37±0.5	92.38±0.7	1943	67.31±0.8	77.81±0.8	7880	76.34±0.3	84.34±0.3
	7	4301	83.74±0.5	90.43±0.4	990	89.74±0.8	95.34±0.4	1673	74.54±0.6	84.02±0.9	6960	81.77±0.4	89.13±0.3
	9	3792	87.04±0.4	92.54±0.4	780	92.56±0.6	96.55±0.6	1375	79.47±1.0	86.4±0.9	6012	85.4±0.6	91.37±0.2
2012	5	5339	76.35±0.6	84.43±0.5	1799	82.73±0.9	90.48±0.5	2678	61.15±0.5	71.82±0.5	9812	72.46±0.4	81.08±0.4
	7	4134	82.24±0.4	89.62±0.6	1166	88.81±0.4	94.3±0.8	1770	71.3±0.9	81.19±0.5	7155	79.76±0.5	87.46±0.3
	9	3359	85.93±0.5	91.82±0.4	780	92.22±0.9	95.78±0.6	1253	78.41±0.9	85.84±0.7	5518	84.14±0.4	90.48±0.3
2013	5	7139	76.68±0.5	85.01±0.3	2508	81.24±0.4	88.4±0.4	3487	60.65±1.0	71.15±0.8	13069	71.62±0.3	80.35±0.3
	7	5773	82.6±0.3	89.88±0.5	1796	86.94±0.5	92.57±0.7	2577	69.78±0.5	79.59±0.5	10269	78.58±0.3	86.35±0.3
	9	4640	86.72±0.4	92.47±0.3	1309	90.47±0.8	94.68±0.4	1959	75.89±1.2	84.55±0.5	8133	83.38±0.4	89.59±0.3
2014	5	8529	74.94±0.3	83.45±0.4	2945	81.2±0.6	89.11±0.5	3690	63.18±0.6	73.5±0.6	15105	72.46±0.2	81.14±0.2
	7	6552	81.63±0.3	89.18±0.2	2080	85.7±0.5	92.22±0.6	2619	70.89±0.8	81.46±0.8	11301	79.3±0.3	87.13±0.3
	9	5238	86.38±0.6	92.14±0.4	1511	88.71±0.7	94.1±0.4	1939	77.7±0.8	85.16±0.6	8791	84.14±0.3	90.25±0.3
2015	5	7502	80.37±0.4	87.69±0.3	2323	85.06±0.5	91.16±0.5	2894	68.67±0.9	77.93±0.6	12673	77.81±0.3	85.43±0.2
	7	5774	85.07±0.5	91.78±0.3	1493	88.11±0.6	93.82±0.4	1831	75.48±0.8	84.59±0.8	9110	82.98±0.4	90.03±0.3
	9	4674	88.16±0.4	93.27±0.3	1074	90.05±0.9	94.65±0.6	1346	80.47±0.9	88.22±0.8	7166	86.38±0.5	92.07±0.2
2016	5	9366	75.58±0.4	83.78±0.2	4340	70.75±0.7	79.88±0.3	4067	58.27±0.8	69.11±0.6	17592	69.29±0.2	77.95±0.2
	7	8077	81.76±0.4	88.97±0.3	3553	77.82±0.3	85.93±0.3	3366	67.0±0.6	76.96±0.7	15042	76.4±0.2	84.56±0.2
	9	6607	85.45±0.2	91.19±0.2	2794	82.13±0.7	88.67±0.6	2607	72.8±0.7	81.29±0.4	12239	80.75±0.3	87.24±0.3
2017	5	8923	79.14±0.3	86.42±0.2	3702	76.5±0.5	85.19±0.5	3122	61.94±0.6	72.79±0.5	15716	74.22±0.2	82.72±0.2
	7	7315	84.86±0.3	91.43±0.3	2750	83.05±0.6	90.12±0.6	2366	72.17±0.5	81.17±0.8	12529	81.04±0.3	88.48±0.2
	9	5929	88.71±0.3	93.84±0.3	2061	85.98±0.6	92.09±0.3	1703	78.39±1.0	86.66±0.7	9831	85.61±0.3	91.46±0.3
2018	5	10090	75.66±0.3	82.43±0.4	4885	70.12±0.3	78.4±0.4	3734	61.36±0.8	69.71±0.7	18231	70.14±0.2	77.7±0.2
	7	8852	81.33±0.4	87.37±0.3	4169	75.96±0.5	83.25±0.5	3152	69.11±0.7	76.53±0.6	15922	76.45±0.2	83.28±0.3
	9	7850	84.54±0.4	89.63±0.3	3611	79.2±0.6	86.13±0.5	2726	73.89±1.0	80.12±0.7	14070	80.15±0.3	85.76±0.2
2019	5	13523	74.66±0.5	80.75±0.2	9597	65.93±0.6	72.9±0.3	5152	58.68±0.4	66.05±0.3	26224	65.84±0.2	72.61±0.2
	7	11832	80.63±0.3	86.01±0.3	8236	72.53±0.5	78.88±0.3	4475	66.52±0.7	73.2±0.5	23321	72.73±0.2	78.98±0.3
	9	10432	84.41±0.3	88.74±0.2	7123	76.69±0.6	82.15±0.4	3925	71.64±0.4	77.29±0.7	20833	77.15±0.3	82.37±0.2
2020	5	20677	71.3±0.3	78.02±0.1	11800	54.75±0.4	63.62±0.4	5032	53.24±0.5	61.3±0.7	36157	62.23±0.1	69.63±0.3
	7	18818	77.22±0.2	83.57±0.2	10499	62.23±0.4	70.63±0.4	4510	60.69±0.7	68.93±0.4	33174	68.74±0.2	75.98±0.2
	9	16916	81.55±0.3	86.4±0.3	9373	67.37±0.4	74.73±0.4	3985	66.72±0.7	73.92±0.7	30084	73.47±0.2	79.6±0.1