Incremental Learning of Code Authors Over Time

Siyi Gong and Hao Zhong

[a]*Shanghai Jiao Tong University, China*

Abstract

Identifying code authors is important in many research topics, and various approaches have been proposed. Recent studies show that the temporal effect can significantly affect existing approaches: their trained models rapidly become outdated and ineffective due to the evolution of code styles over time. To the best of our knowledge, only a recent approach tries to alleviate the temporal effect. This approach treats the temporal effect problem as a cross-domain problem and uses transfer learning to reduce the temporal effect. Here, a domain refers to the time when the programmers are writing the source files. Although this approach achieves promising results on their datasets, the evaluation of this approach shows that the effectiveness of transferred models decreases with the increasing intervals between the source and the target domains.

In this paper, we propose a novel approach to combat the temporal effect. In real development, source files are accumulated over time and trained models should be continuously updated with continuous data. Based on this insight, we use chunk-based incremental learning: we treat the accumulated source files as data streams, and each incoming batch of data is treated as a chunk. With this concept, we utilize an ensemble framework to maintain an ensemble of base classifiers that are incrementally trained (with no access to previous data) on incoming chunks of data. The base classifiers are dynamically weighted according to their effectiveness on the current data chunk, and these classifiers are combined with a dynamically weighted voting. We evaluate the effectiveness of our approach using two datasets of programs written in C++ and Java. Our evaluation results show that our incremental learning-based approach leads to significant improvements, compared to the previously published transfer learning-based approach. Our approach improves the average accuracy from 0.7343 to 0.9017 on the Java dataset and from 0.8016 to 0.9022 on the C++ dataset. Indeed, in the testing of all seven

years, our approach consistently outperforms the prior approaches.

Keywords: code authorship attribution, coding style evolution, incremental learning

---

1. Introduction

Given a source file or piece of code and a set of authors, the task of code authorship attribution is to attribute the author of this source file. Identifying the authors of source files is critical in many scenarios. For example, code repositories can record wrong code authors, since true code authors may not have the right to submit their changes. Due to some restricts of licences, it needs a complicated procedure to accept changes from outside programmers. True code authors can bypass the procedure, and their true identities are not recorded. Due to the above considerations, researchers [1, 2, 3, 4, 5, 6, 7, 8, 9] have proposed various approaches to identify code authors. Although researchers report highly positive results, they do not consider many factors in real development. For example, in most papers, researchers assume that code styles will not change over time, but this assumption does not hold in real development. With the evolution of software, the code style of an author can change over time. Traditionally, models are trained once forever, but such models may not work well for authors whose styles are changed. As a result, Gong and Zhong [10] report that the effectiveness of existing approaches is significantly reduced when the impact of time is considered.

The evolving nature of open-source development calls for adaptive and efficient approaches. However, to the best of our knowledge, only a recent approach [11] considers the temporal effect. In particular, Li *et al.* [11] introduce transfer learning [12] to update decayed models. Their approach tunes a model trained from the data of a year according to the data of another year. Although source files of code authors are accumulated each year, a source domain can be quite different from a target domain. The effectiveness of transfer learning will be significantly reduced if source and target domains have many differences. As a result, although Li *et al.* [11] achieve promising results, their evaluations show that the effectiveness of transferred models decreases if the year gaps between the source and the target domains become larger.

To improve the state of the art, we propose a novel approach called CIcaL (Chunk-based Incremental Code Authors Learning). To resolve the

limitation of Li *et al.* [11], we introduce incremental learning since it can incrementally update a trained model.

This paper makes the following contributions:

- **The first approach called CIcaL uses incremental learning to identify code authors.** CICAL treats the accumulated source files as data streams, and each incoming batch of source files is treated as a data chunk. CICAL utilizes an ensemble framework to maintain an ensemble of base classifiers that are incrementally trained (with no access to previous data) on incoming chunks of data, and the base classifiers are dynamically weighted according to their performance on the current data chunk, then these classifiers are combined with a dynamically weighted voting. CICAL incrementally tunes a trained model year by year. In this way, CICAL accumulates new knowledge and continuously improves the prediction ability by dynamically updating the trained model.
- **Promising evaluation results on benchmarks.** We evaluate the effectiveness of CICAL using a Java dataset and a C++ dataset. Compared with the transfer learning-based approach [11], CICAL improves the average accuracy from 0.7343 to 0.9017 on the Java dataset, and improves the average accuracy from 0.8016 to 0.9022 on the C++ dataset. In addition, it works better on imbalanced data. Compared with the transfer learning-based approach [11], CICAL improves the average accuracy from 0.5880 to 0.7772 on the imbalanced Java dataset, and improves the average accuracy from 0.6266 to 0.7691 on the imbalanced C++ dataset.

2. Preliminary

The problem. A code author's programming style evolves over time, which results in the earliest code samples becoming the least reliable indicators of the current programming style [13]. Burrows *et al.* [13] use a collection of six programming assignments with guaranteed relative timestamps from 272 students to examine the evolution of coding style, they conclude that coding style does change over time and it takes at least three programming tasks for coding style to settle. Considering the coding style of students can evolve during their studies, Hansen *et al.* [14] examine the practical feasibility of using limited and recent writing samples from students for authorship

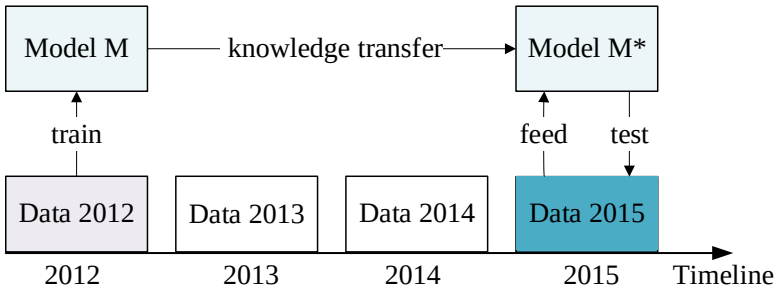


Figure 1: Transfer learning

attribution. Caliskan *et al.* [2] propose a random forest and abstract syntax tree-based approach based on the data set extracted from Google Code Jam, they find that skilled programmers (who can complete the more difficult tasks) are easier to attribute than less skilled programmers. To analyze the influence of time on source code authorship attribution, Petrik and Chuda [8] use the Google Code Jam dataset to test if there are any significant changes in the author's style over time. Their results reveal significant changes in the code style features in one year difference, which enlarges as the difference of time increases. Bogomolov *et al.* [7] also demonstrate that the evolution of programming style affects the accuracy of authorship attribution. The above papers suggest that the temporal effect is a challenge for code authorship identification, since the programming style of programmers evolves rapidly with time due to their education and experience. Besides the temporal effect, the class imbalance problem also limits the effectiveness of learned models. To evaluate the proposed code authorship attribution models, some prior studies [5, 6] use custom-built source code collections that include balanced training sets. Note that an author can have many more or fewer files in real scenarios [15]. Thus, long samples (multiple lines of code) may be available for some authors and short samples for other authors. This can also be viewed as a case of class imbalance. Most prior approaches ignore the above problems. Only a recent approach starts to resolve it with transfer learning. We next introduce its basic idea.

Transfer learning. In the scenarios of transfer learning, a model learned from a task is reused as the starting point for learning the model for another task [12]. When training the new model, transfer learning changes the weights and adds new data points to the old model. Transfer learning has been widely used in natural language processing [16] and networking [17].

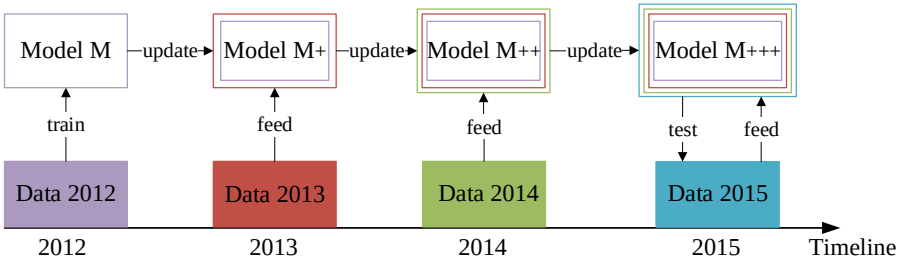


Figure 2: Incremental learning

```
1  public static void main(String[] args) {
2          try {
3                  in = new BufferedReader(new InputStreamReader(System.in));
4                  out = new PrintWriter(new OutputStreamWriter(System.out));
5                  int tests = nextInt();
6                  for (test = 1; test <= tests; test++) {
7                          solve();
8                  }
9                  in.close();
10                 out.close();
11         } catch (Throwable e) {
12                 e.printStackTrace();
13                 exit(1);...
```

Figure 3: A source file written by eatmore* in 2015

Recently, Li *et al.* [11] used transfer learning to identify code authors. Li *et al.* [11] demonstrated that if a model is trained according to the data of 2012, it can become obsolete in 2015. To update the model, as shown in Figure 1, they introduce transfer learning to refine the model with the data of 2015. In this way, Model M is updated to Model $M*$. Transfer learning has two inherent limitations. First, if a model was trained years ago, the source and the target domains can have huge differences. The evaluation of Li *et al.* shows that the effectiveness of transferred models decreases with the increasing of intervals between the source and the target domains. Second, transfer learning ignores the data between the source and target domains. In this example, the data of 2013 and 2014 are ignored, although they can be useful to refine models. As a result, although their proposed approach can reduce the impact of time, the evaluation of Li *et al.* shows that the effectiveness of transferred models decreases with the increasing of intervals between the source and the target domains.

Incremental learning. In the scenarios of incremental learning, models are incrementally updated with continuous data. It represents a dynamic technique that is designed for the scenarios when training data are accumulated over time [19]. In real development, source files are accumulated, and

Table 1: Metrics of eatmore* defined by [18, 3]

Metrics	sample	2012	2013	2014	2015
`while`	0.6667	0.2101	0.3152	0.2252	0.2634
`for`	0.3333	0.7807	0.6848	0.7710	0.7366
`do`	0	0.0092	0	0.0038	0
cyclic	1	2	1.6667	1.9	1.9286
`if`	0.125	1	1	1	08255
`switch`	0.875	0	0	0	0.1744
`if-else`	1	0.8771	0.7966	0.8090	0.9131
`static`	0.1809	0.1395	0.1637	0.1489	0.1652
`class`	0.0106	0.0104	0.0117	0.0099	0.0102
`import`	0.1170	0.0966	0.1072	0.0996	0.1042
`new`	0.0532	0.0778	0.0711	0.0736	0.073
`public`	0.0213	0.0206	0.0196	0.0184	0.0193
`this`	0	0.0065	0.0074	0	0.0015
`try`	0.0106	0.0077	0.0087	0.0082	0.0089
`throw`	0	0.0038	0	0.0017	0.0022
`catch`	0.0106	0.0077	0.0087	0.0082	0.0089
`final`	0	0.0082	0.0057	0.0032	0.0039
`private`	0	0.0009	0	0	0
`instanceof`	0	0.0004	0.0008	0	0
`implements`	0	0.0012	0.0006	0.0004	0.0005
`super`	0	0	0	0.0003	0

this scenario well fits the target of incremental learning. Under this circumstance, we decided to build an incremental learning-based model. As shown in Figure 2, to better identify the authors of 2015, we use incremental learning to accumulate new knowledge from 2013 and 2014 and continuously improve the ability of the model. In this way, Model M is updated to Model $M++$. Compared to transfer learning, incremental learning has the ability to sustain classification accuracy. Besides, incremental learning can incrementally learn the evolving coding style of code authors year by year.

In summary, transfer learning aims to use the knowledge gained from an existing task to solve a new task that is related to the existing one [20]. Incremental learning aims to incrementally update the model with new data while retaining the knowledge gained from previous data [21]. Different from transfer learning, incremental learning aims to gradually accumulate new knowledge and continuously improve the ability of a model. Incremental learning is useful in situations where you need to continuously increase the data set and update the model, such as when you want to continuously train the model to adapt to data changes over time.

3. Motivating Example

In this section, we introduce the benefits of incremental learning. The code styles of a code author can change over time. For example, Figure 3 shows a source file written by eatmore* in 2015. To protect the privacy of programmers, we use asterisks to hide their real names. To define the code styles of code authors, the prior approaches [18, 3] extract a set of code metrics, and these approaches rely on the code metrics to identify code authors. To illustrate how the code styles evolve over time, we calculate the code metrics of source files written by eatmore* from 2012 to 2015. Table 1 shows the result. Column "**Metrics**" lists the code metrics. From each source file, we build an abstract syntax and analyze the tree to collect the metrics. The top three rows list the ratio of `while`, `for`, and `do` statements in all loop statements, respectively. The fourth row lists the preference for cyclic statements. According to the most frequent cyclic statement, we set the values to 1, 2, and 3, respectively. The fifth and sixth rows list the ratio of `if` and `switch` statements in all conditional statements, respectively. The seventh row lists the percentage of `if` in all `if` and `else`. The remaining rows list the ratio of corresponding keywords to lines of non-comment code lines. Column "**sample**" lists the values of a program. Columns "**2012**", "**2013**", "**2014**", and "**2015**" list the averages of the code metrics of source files written by eatmore* in the corresponding year.

Although the differences of a code author look minor, Li *et al.* [11] report that the prediction accuracy of the prior approaches decreases significantly since the differences accumulate over time. For instance, in 2012, eatmore* preferred to write `for` loops, but in 2015, this code author changed to write `while` loops. If an approach trains the models from the data of past years, it is unlikely to identify the correct code authors. As a result, when it is trained on the data of 2012, Multi-χ wrongly identifies the author of Figure 3 as goalboy1015*.

Transfer learning learns a source-domain model from the data of 2012, and tunes a target-domain model with the data of 2015. Combining the data of the two years can lead to better models. For example, the ratio of `private` keywords to lines of non-comment code in the source file of 2015 shown in Figure 3 is 0, but this value in 2012 is more than zero. Meanwhile, as Table 1 shows that this value in 2015 is 0, transfer learning can tune the model that is trained from the data of 2012 accordingly. However, the sample file in Figure 3 contains values that are different from the data of 2012 and 2015.

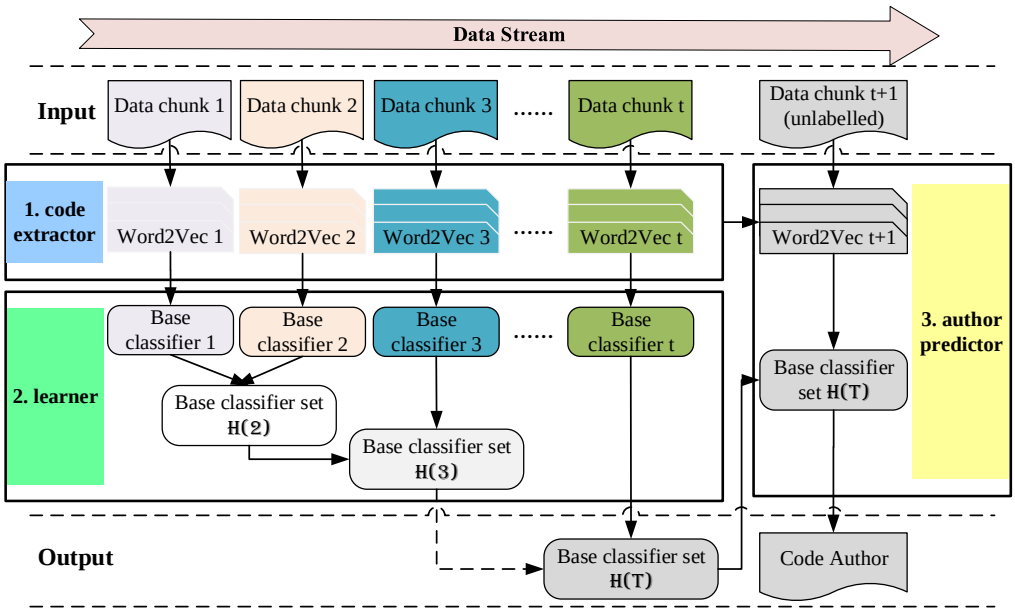


Figure 4: Approach overview

For example, in this sample file, eatmore* writes no `throw` statements, but in the source files of 2012 and 2015 both, this author wrote `throw` statements. Partially due to this difference, the tuned model wrongly identifies the author of Figure 3 as it3*, when TimeDA trains a model from the data of 2012 and tunes this model with the data of 2015.

With incremental learning, CICAL can learn code styles from the data of all years. For example, Table 1 shows that eatmore* started to develop a preference for writing `while` loops in 2013. Meanwhile, some styles can be inferred from the years between 2012 and 2015. For example, the ratio of `this` keywords to lines of non-comment code in the source file of 2015 is 0, but this value in 2012 is more than zero, while Table 1 shows that this value in 2014 is 0, indicating that when an approach trains the models from the data of 2012 and 2014, it is likely to learn this code metrics. As a result, CICAL correctly identifies the author of Figure 3.

We next introduce how CICAL builds such models.

4. Approach

Figure 4 shows the overview of our approach CICAL. It consists of a code extractor (Section 4.1), a learner (Section 4.2), and an author predictor (Section 4.3). The input of CICAL is a series of data chunks (source files

written within a period of time), and the output of CICAL is an updated model. Given a source file whose author is unknown, the updated model can predict its code author.

4.1. Code Extractor

As the first step, CICAL transforms source code to a numerical representation. Based on their representations, the prior approaches can be roughly divided into two categories. The first category of approaches [2, 3] extract code meatrics. For example, Yang *et al.* [3] extract 19 code metrics. Table 1 provides some examples of their code metrics. As it is challenging to identify general code metrics [7], most other approaches use the off-the-shelf techniques such as TF-IDF [5] and word2vec [22]. Most prior approaches [23, 24, 25] use word2vec [22] to encode source files, and some approaches [6] use both representations. We use word2vec to encode Java and C++ source files, and we compare the results of encoding with word2vec and TF-IDF in our evaluations. Figure 4 shows that CICAL is built upon a set of base classifiers. The prior approaches typically reduce the problem of identifying code authors to a classification problem. If we select the same representation, it is easier to integrate CICAL with the prior approaches.

We consider segments of source code as sequences of terms and expressions for training a word2vec model, which in turn is used to generate representations of code terms and expressions. The word2vec models are constructed using the training dataset only. When applying the representation scheme, the out-of-vocabulary (OOV) problem may occur during the validation and testing part of the experiment. There are several approaches for handling the OOV problem [26]. In this study, unseen terms are represented with zero-vectors when using word2vec.

4.2. Learner

At timestamp $t-1$, CICAL maintains m classifiers in the set $\mathcal{H}(t-1) = H_1^{(t-1)}, \cdots, H_m^{(t-1)}$ trained on the data chunks from timestamp 1 to $t-1$. When CICAL receives a new data chunk $\mathcal{D}(t)$ at timestamp t, it learns a new classifier H on the current data chunk and merges H with $\mathcal{H}(t-1)$ to form $\mathcal{H}(t)$. The classifiers trained on each chunk are associated with the vector of weights, denoted as $w^{(t)} = [w_1^{(t)}, \cdots, w_m^{(t)}]^T$, which measures the importance of the classifiers in the ensemble. When the new classifier H is created, its initial weight is set at 1 and m is increased by 1. To adapt the previously learned classifiers to new concepts, the weight $w_j^{(t)}$ for classifier

Algorithm 1: Train CIcaL

Input: Data chunk at timestamp t: $\mathcal{D}(t) = x_i \in X, y_i \in Y, i = 1, \cdots, N$, the threshold for deleting base classifiers θ, the base classifier set $\mathcal{H}(t-1) = H_1^{(t-1)}, \cdots, H_m^{(t-1)}$, the weights of base classifiers, $w^{(t-1)}$, the number of base classifiers m, and the ensemble size T.

for $i \leftarrow 1$ **to** N **do**

Predict x_i by the ensemble classifier:

$\bar{y} = sign(\sum_{j=1}^{m} w_j^{(t-1)} H_j^{(t-1)}(x_i))$;

end for

for $j \leftarrow 1$ **to** m **do**

Calculate the error $\epsilon_j^{(t)}$ for classifier $H_j^{(t-1)}$ on $\mathcal{D}(t)$;

Update weight of base classifiers:

$w_j^{(t)} = (1 - \epsilon_j^{(t)}) w_j^{(t-1)}$;

end for

Remove classifiers with weights less than θ:

$\mathcal{H}(t) \leftarrow \mathcal{H}(t-1) \backslash \left\{ H_j^{(t-1)} | w_j^{(t)} < \theta \right\}$;

$m \leftarrow |\mathcal{H}(t)|$;

Create new base classifier and initialize its weight:

$m \leftarrow m - 1$;

$H \leftarrow UnderBagging(\mathcal{D}(t), T)$;

$\mathcal{H}(t) \leftarrow \mathcal{H}(t) \cup H$);

$w_m^{(t)} \leftarrow 1$;

Output: Base classifier set $\mathcal{H}(t)$, weight of base classifiers $w^{(t)}$, number of base classifiers m, prediction $\bar{y}$.

$H_j^{(t)}$ is reduced on each timestamp after it is created: $w_j^{(t)} = (1 - \epsilon_j^{(t)}) w_j^{(t-1)}$, where $j = 1, \cdots, m-1$, and $\epsilon_j^{(t)}$ is the testing error of $H_j^{(t)}$ on the current data chunk $\mathcal{D}(t)$. The error $\epsilon_j^{(t)}$ can be calculated by an error function like F1 or G-mean, since F1 and G-mean can be used to evaluate the model performance under unbalanced data. Thus, the weights of the classifiers trained on the past chunks are reduced based on their effectiveness on the current data chunk. As this weight reduction is accumulated over time, the weight $w_j^{(t)}$ is equal to:

$$w_j^{(t)} = \prod_{(\tau=l+1)}^{t} (1 - \epsilon_j^{(\tau)}) \tag{1}$$

where l is the timestamp when $H_j^{(t)}$ is created. As $(1 - \epsilon_j^{(\tau)}) \leq 1$, the classifier weight is getting smaller over time according to its error on each chunk after it is created. Then, the classifiers with weight less than the threshold θ are removed and the counter m is also reduced according to the number of classifiers left. If a classifier is going to be removed, there are two factors that make its weight lower than θ. One is that the classifier is trained on a

Algorithm 2: UnderBagging

Input: Data $\mathcal{D}(t) = x_i \in X, y_i \in Y, i == 1, \cdots, N$, the number of positive samples N_p, the number of negative samples N_n, ensemble size T.

for $t \leftarrow 1$ **to** T **do**
  if $N_p < N_n$ **then**
    $N_s \leftarrow N_p$;
  else
    $N_s \leftarrow N_n$;
  end if
  $\mathcal{D}(p) \leftarrow$ Bootstrap N_s positive samples;
  $\mathcal{D}(n) \leftarrow$ Bootstrap N_s negative samples;
  $h_t \leftarrow BaseLearner(\mathcal{D}(p), \mathcal{D}(n))$;
end for

Output: Base classifier$H(x) = sign(\sum_{t=1}^{T} h_t(x))$.

quite early timestamp that makes the production in Equation 1 small. The other is that the concept changes in recent chunks and the testing error of the classifier is large on those chunks. Thus, this kind of classifier is less likely to provide positive effects to the prediction on the current and following chunks. Finally, the model predicts the incoming data x in $\mathcal{D}(t+1)$ by the ensemble of $\mathcal{H}(t)$ associated with $w^{(t)}$:

$$sign(\sum_{j=1}^{m} w_j^{(t)} H_j^{(t)}(x)) \tag{2}$$

Algorithm 1 shows the training process of CICAL. In real development, a few core programmers write most code lines, and in some extreme cases, some programmers only contribute several code lines [15], which means that researchers encounter extreme data imbalance problems when identifying real authors. In CICAL, we use UnderBagging [27] as the base learner to handle imbalanced data. In each bagging iteration, we carry on undersampling on the majority class to make the training data balanced.

4.3. Author Predictor

As shown in Figure 4, for a new unlabelled data chunk $t+1$ (a piece of new source code or a new source file), we first feed it to our code extractor. After it obtains the code embeddings for the data chunk $t+1$, CICAL uses the ensemble of base classifiers (base classifier set $H(t)$ trained by the learner component) to predict the code embeddings of data chunk $t+1$. This new unlabelled data chunk $t+1$ is also being used to train a new base classifier and being merged into the ensemble base classifier set. Specifically, for the

learner component, the number of base classifiers in the ensemble T is set at 5 to prevent draw result since we use dynamically weighted voting to combine these base classifiers. The threshold to remove the dated classifier θ is set at 0.001. As pointed out by Kolter *et al.* [28], the value of θ has nearly no influence on the accuracy, and it only affects the number of stored classifiers. Geometric mean (G-mean) error $\epsilon_{gm} = 1 - \sqrt{TPR \cdot TNR}$ is chosen as the error function used in the learner component, where TPR is the true positive rate and TNR is the true negative rate. We select the G-mean error as the error function. As recurrent neural networks, LSTMs [28] are widely used as sequence classifiers in various research topics. We select LSTM [28] as the base classifier of CIcaL.

5. Evaluation

Our study aims to answer the following research questions:

(RQ1) How does CIcaL improve the state of the art?

(RQ2) How effective is CIcaL on the imbalanced data?

(RQ3) How effective are the code extractor techniques?

More details of the evaluations are listed on our website:
`https://github.com/gongsiyi/authorship/`

5.1. Setting

5.1.1. Benchmark

In our evaluation, we use the same datasets of Li *et al.* [11]. Li *et al.* [11] collected C++ and Java source files between 2012 and 2018 from Google Code Jam (GCJ) entries to build the datasets. In particular, they remove authors who write less than 7 program files or write source files only in a year. As GCJ records the authors of source files, its recorded authors are considered as the labels of source files. Table 2 shows the two datasets (GCJ_Java and GCJ_C++). Column "**Year**" lists the time of source files. Column "**File**" lists the number of source files. Column "**LOC**" lists the number of code lines. Column "**Author**" lists the number of authors. In total, the two datasets contain more than 3,000 source files that are written by 48 authors.

Table 2: Our datasets

Year	Java			C++		
	File	LOC	Author	File	LOC	Author
2012	195	25,055	20	324	31,069	28
2013	189	28,942	20	322	30,462	28
2014	251	38,472	20	357	37,350	28
2015	239	38,164	20	328	33,906	28
2016	248	40,033	20	333	32,673	28
2017	229	42,281	20	304	31,648	28
2018	243	30,673	20	363	30,645	28
total	1,594	243,620	20	2,331	227,753	28

5.1.2. Measures

Like all the prior approaches [11, 1, 2, 3, 4, 5, 6, 7, 8, 9], we select accuracy to measure the results. Accuracy is defined as follows:

$$Accuracy = \frac{f_c}{f_{total}} \tag{3}$$

where f_c denotes the number of correctly attributed source files, and f_{total} denotes the total number of source files.

Besides accuracy, we select matthew correlation coefficient (MCC) [29] and F1 score [30] as our measures. MCC is a reliable measure when the data are imbalanced.

5.1.3. Statistical hypothesis testing

To ensure the reliability of our results, we use statistical hypothesis testing. In this approach, a null hypothesis, which is represented by H_0, and its opposite, alternative hypothesis which is represented by H_a, are defined first. Then, acceptance of the null hypothesis (and rejection of the alternative) or vice versa would be evaluated by considering a particular confidence level of the corresponding statistical test. If the confidence level of rejecting a null hypothesis exceeds a specified threshold, the alternative hypothesis will be accepted (and the null hypothesis will be rejected). Otherwise, the alternative one would be rejected (and the null hypothesis would be accepted). We will accept an alternative hypothesis in this paper if the confidence level of accepting it is greater than 95%, *i.e.*, the p-value is less than 0.05.

5.2. RQ1. Comparison with baselines

5.2.1. Baseline

In our evaluation, we select two baselines. From classical approaches, we select Multi-χ [6], since it is a recent approach. Multi-χ divides source files into segments, and encodes each segment as a sequence of n-dimensional data with TF-IDF [5]. Taking the sequences and their label as its input, Multi-χ trains a model with BiLSTM [31]. As Abuhamad *et al.* [6] did not release their tool, we implemented the tool upon Keras [32], according to their paper [6]. On the C++ dataset of 2015, our implementation on Multi-χ achieved an accuracy of 80.18%, and the result is close to what is reported in their paper (an accuracy of 79.62%).

We select Li *et al.* [11], since it is the first approach that considers the decay of models. Their approach is called TimeDA. It uses domain adaptation [11] to tune decayed models. As Li *et al.* [11] released their tool and dataset on Github, we use their released tool. TimeDA uses DL-CAIS [5] and PbNN [7]. In particular, DL-CAIS uses recurrent neural networks and fully connected layers for learning, and uses random forests for authorship attribution. PbNN uses a fully connected layer with softmax activation. The released version of TimeDA uses only PbNN, and we select this version.

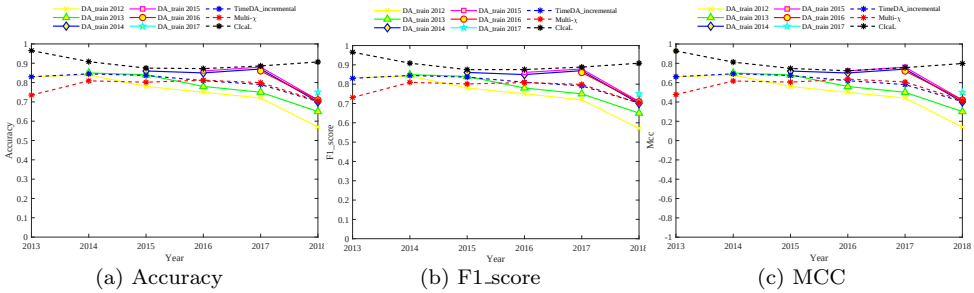
5.2.2. Training and testing process

Li *et al.* [11] provide a Java dataset and a C++ dataset. We divide each dataset into 7 chunks by year.

For Multi-χ, we train a model from the data of a year, and test the trained model on the data of the next year. In total, for 7 chunks of each dataset, we obtain 6 trained models and 6 tested results.

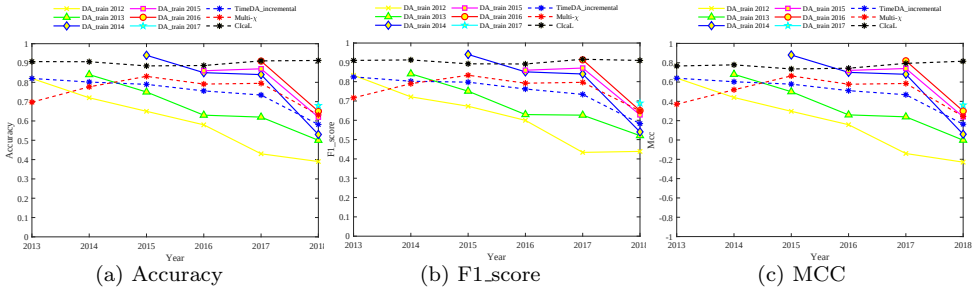
For TimeDA, we follow the setting of Li *et al.* [11]. In particular, we train a model from the data of a year. After that, we transfer this model to subsequent years and test transferred models on corresponding years. For instance, we train a model from the data of 2012. After that, we transfer and test this model on the data from 2013 to 2018 separately. In total, we train 6 models and obtain 21 tested results.

For CIcaL, we train a model from the data of a year. After that, we update this model with the data of the follow-up years subsequently and test the updated model on the data of the follow-up years. For instance, the model trained from 2012 is updated with the data from 2013 to 2018, and each updated model is tested on the data of the corresponding year. In total, we obtain 1 model and 6 evaluation results.



(a) Accuracy (b) F1_score (c) MCC

Figure 5: The improvement on GCJ_C++ dataset



(a) Accuracy (b) F1_score (c) MCC

Figure 6: The improvement on GCJ_Java dataset

In a previous submission, a reviewer asks us to test TimeDA in an envisaged setting. In this setting, we incrementally update the models of TimeDA. In particular, we train a model of TimeDA from the data of a year. After that, we incrementally transfer this model to subsequent years and test the transferred model on the data of corresponding years. For instance, the model trained from 2012 is incrementally transferred to 2013 to 2018, and each transferred model is tested on the data of the corresponding year. Eventually, we obtain 1 model and 6 evaluation results from each dataset. We call this setting incremental TimeDA.

We calculate the average accuracy, f1 score, and MCC to compare the effectiveness.

5.2.3. Result

Figures 5 and 6 show the results. The horizontal axes list the years of the testing sets, and the vertical axes list the corresponding measures. The red and the black dashed lines depict the results of Multi-χ and CIcaL, respectively. Compared with Multi-χ, CIcaL achieves better results in all measures on both benchmarks. The least improvements occur in the data of 2016 in the C++ dataset and the data of 2015 in the Java dataset. Even in

the two cases, CICAL improves the accuracy, f1 score, and MCC of Multi-χ on the 2016 C++ dataset by 0.0598, 0.0672, and 0.0862, respectively. CICAL improves the accuracy, f1 score, and MCC of Multi-χ on the 2015 Java dataset by 0.0940, 0.1004, and 0.1581, respectively. We use the McNemar's test [33] to check whether the difference between Multi-χ and CICAL is significant. For both datasets, the p-value results are below 0.05. The results show that the difference between the results of Multi-χ and CICAL is significant.

The solid lines depict the results when TimeDA [11] trains a model from the data of a year and transfers the model to the corresponding year. CICAL achieves better results than TimeDA except in 2015 of the C++ dataset. When the year gap between the training and testing set is fewer than three, TimeDA achieves better results than Multi-χ, although it is poorer than CICAL. When the year gap is larger than three, TimeDA achieves even poorer results than Multi-χ. We check whether the difference between TimeDA and CICAL is significant. The p-value results are below 0.05. The above observations lead to a finding:

Result 1: When the temporal effect is considered, CICAL significantly outperforms Multi-χ and TimeDA.

When the year gap between the training and testing set is 1, the values of TimeDA are close to ours. As shown in the blue lines of Figures 6a, 6b, and 6c, the accuracy, f1 score, and MCC of TimeDA are more than CICAL by 0.0550, 0.0484, and 0.1445 when TimeDA trains in 2014 of the Java dataset and tests the model in 2015 of the Java dataset. Still, when the gap between the training and the testing years becomes larger, all values drop significantly. For instance, as shown in the blue lines of Figures 6a, 6b, and 6c, although the initial models of TimeDA are better than ours, they become poorer than ours when TimeDA transfers the model to follow-up years. This finding is consistent with the evaluation results of Li *et al.* [11]. In Figures 5 and 6, the results of TimeDA and Multi-χ significantly drop in the data point of 2018. The results are consistent with the evaluation results of Li *et al.* [11]. Their results also significantly drop in the data of 2018. The data for this year can have more differences. Still, CICAL is more effective in handling the time issue than TimeDA since its lines are smoother. The observations lead to a finding:

Result 2: CICAL is more effective in handling temporal changes than TimeDA.

The blue dashed lines depict the results when TimeDA [11] trains a model from the data of 2012 and transfers this model incrementally over time. Compared with red dashed lines, TimeDA produces mixed results under the settings of Li *et al.* [11] and our new setting. In both datasets, the blue and red dashed lines twist together. Still, CICAL always produces better results when TimeDA is used in a way that is similar to incremental learning. Under this setting, CICAL improves the average accuracy, f1 score, and MCC by 0.1546, 0.1549, and 0.2780 on the Java dataset, respectively. On the C++ dataset, the improvements are 0.1000, 0.1010, and 0.1890, respectively. We also check whether the difference between incremental TimeDA and CICAL is significant. As the p-value results are below 0.05, the difference is significant. The above observations lead to a finding:

Result 3: Even if TimeDA is used incrementally (the incremental TimeDA setting), it is not as effective as CICAL.

To understand the impact of programming languages, we use the McNemar's test [33] to check the result difference between the Java and C++ datasets. The p-value results are 0.4915 (accuracy), 0.4896 (f1 score), and 0.5150 (MCC). The results show that the difference is not significant, and the impact of programming languages is minor.

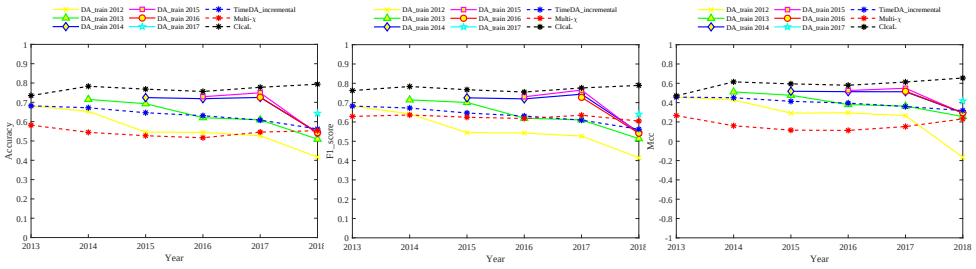
Result 4: CICAL performs similarly on both Java and C++ datasets.

In summary, CICAL significantly improves Multi-χ and TimeDA. Although TimeDA considers the time issue, CICAL is more effective in handling the changes over time than TimeDA, even if TimeDA incrementally transfers its model.

5.3. RQ2. The Impact of Imbalanced Data

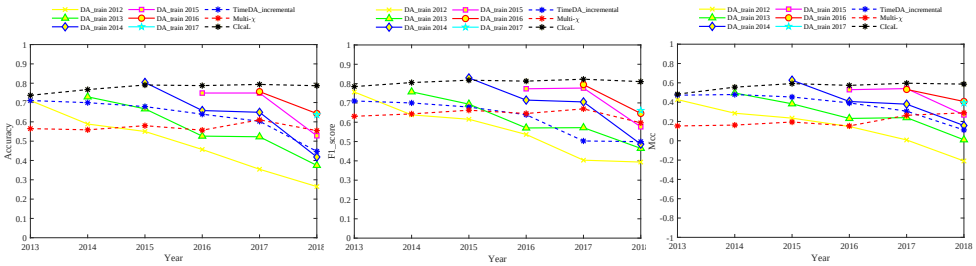
5.3.1. Setup

In real development, a few core programmers write most code lines [15], but the dataset of Li *et al.* [11] is relatively balanced since they remove authors whose files are less than 7. In this RQ, we build an imbalanced Java dataset and an imbalanced C++ dataset. In each imbalanced dataset, the



(a) Accuracy (b) F1_score (c) MCC

Figure 7: The impact of imbalanced data (GCJ_C++ dataset)



(a) Accuracy (b) F1_score (c) MCC

Figure 8: The impact of imbalanced data (GCJ_Java dataset)

percentage of the minority in each year is fixed at 1% of the data size. For each dataset, we choose five minority classes. Both imbalanced datasets were divided into 7 chunks by year. To adjust for imbalanced data, as introduced in Section 4.2, in each bagging iteration, we carry on undersampling on the majority class to make the training data balanced. The training and testing process is the same as the process in RQ1 (see Section 5.2.2 for details).

5.3.2. Result

Figures 7 and 8 show the results. Their horizontal axes, vertical axes, and lines have the same definitions as those of Figures 5 and 6.

Comparing the original setting (Figures 5 and 6) with the imbalanced setting (Figures 7 and 8), we find that the imbalanced data affect all the approaches. For Multi-χ, the average accuracy, the f1 score, and the MCC values decrease by 0.1828, 0.1215, and 0.2904 on the Java dataset, respectively. On the C++ dataset, the values decrease by 0.2323, 0.1509, and 0.3870, respectively. For TimeDA, the average accuracy, the f1 score, and the MCC values decrease by 0.1115, 0.0707, and 0.0884 on the Java dataset, respectively. On the C++ dataset, the values decrease by 0.1548, 0.1534, and 0.1824, respectively. For incremental TimeDA, the average accuracy,

the f1 score, and the MCC values by 0.1174, 0.1289, and 0.1250 on the Java dataset, respectively. On the C++ dataset, the values decrease by 0.1687, 0.1689, and 0.2060, respectively. For CIcaL, the average accuracy, the f1 score, and the MCC values decrease by 0.1245, 0.0966, and 0.2077 on the Java dataset, respectively. On the C++ dataset, the values decrease by 0.1331, 0.1316, and 0.2059, respectively.

The differences in both imbalanced datasets are significant since the p-value results of the McNemar's test [33] are below 0.05. In particular, MCC values of each approach decrease more than accuracy and f1 score values. This observation is consistent with the findings of Chicco and Jurman [29]. The above observations lead to a finding:

Result 5: Imbalanced data significantly affects all the approaches.

Although the imbalanced setting affects all the approaches, CIcaL achieves the best results on both datasets.

Comparing the results between Multi-χ and CIcaL, we find that the differences become larger. In RQ1, the least improvements occur in 2016 in the C++ dataset and 2015 in the Java datasets. In the two cases, CIcaL improves the accuracy, f1 score, and MCC of Multi-χ on the 2016 C++ dataset by 0.0598, 0.0672, and 0.0862, respectively. CIcaL improves the accuracy, f1 score, and MCC of Multi-χ on the 2015 Java dataset by 0.0940, 0.1004, and 0.1581, respectively. When data are imbalanced, the least improvements occur in 2013 of the imbalanced C++ dataset and the imbalanced Java dataset. In the two cases, CIcaL improves the accuracy, f1 score, and MCC of Multi-χ on the imbalanced C++ dataset by 0.1532, 0.1338, and 0.2049, respectively. CIcaL improves the accuracy, f1 score, and MCC of Multi-χ on the imbalanced Java dataset by 0.1724, 0.1539, and 0.3294, respectively. The gaps of the accuracy, f1 score, and MCC between the two settings on the C++ dataset increase 0.0934, 0.0666, and 0.1186, respectively. The gaps of the accuracy, f1 score, and MCC between the two settings on the Java dataset increase 0.0784, 0.0532, and 0.1713, respectively.

Comparing the results between TimeDA and CIcaL, we notice that the initial models of TimeDA become poorer. For instance, in RQ1, when the model is trained from the 2014 data of the Java dataset, the MCC value of TimeDA is more than CIcaL by 0.1445. When the data are imbalanced, the MCC value of TimeDA is more than CIcaL by only 0.0354. When the year gap becomes larger, the effectiveness of TimeDA also decreases significantly

Table 3: The impact of representation (GCJ_C++ dataset)

Year	Improvement_w			Improvement_t			Delta		
	Acc	F1	Mcc	Acc	F1	Mcc	Acc	F1	Mcc
2013	0.23	0.23	0.45	0.20	0.20	0.39	0.03	0.03	0.06
2014	0.10	0.10	0.20	0.07	0.07	0.16	0.03	0.03	0.03
2015	0.07	0.07	0.14	0.04	0.04	0.09	0.03	0.04	0.05
2016	0.06	0.07	0.09	0.02	0.02	0.03	0.04	0.04	0.06
2017	0.09	0.09	0.15	0.05	0.05	0.09	0.04	0.04	0.06
2018	0.20	0.21	0.38	0.17	0.17	0.34	0.03	0.03	0.04
average	0.12	0.13	0.23	0.09	0.09	0.18	0.03	0.04	0.05

on both datasets. As a result, its results are poorer than ours.

Comparing the results between incremental TimeDA and CIcaL, we find that the differences between initial models become smaller. For instance, in RQ1, when the model is tested from the 2013 data of the C++ dataset, the MCC value of incremental TimeDA is less than CIcaL by 0.2665. When the data are imbalanced, the MCC value of incremental TimeDA is less than CIcaL by only 0.0136. Still, CIcaL always produces better results when TimeDA is used in a way that is similar to incremental learning. Under this setting, CIcaL improves the average accuracy, f1 score, and MCC by 0.1475, 0.1872, and 0.1953 on the Java dataset, respectively. On the C++ dataset, the improvements are 0.1356, 0.1383, and 0.1891, respectively. The above observations lead to a finding:

Result 6: CIcaL is the less affected approach by the imbalanced dataset.

In summary, although the imbalanced data problem significantly affects all the approaches, CIcaL is the least affected.

5.4. RQ3. The Impact of Representation

5.4.1. Setup

To encode source code, Multi-χ uses TF-IDF [5]; CIcaL uses word2vec [22]; and the version of TimeDA in our study uses code2vec [7]. Although we achieved better results in RQs1 and 2, a reviewer from our past submission criticizes that the improvements in RQs 1 and 2 may not be caused by incremental learning but by the different encoding techniques.

As word2vec is more advanced than TF-IDF, the criticism can hold when we compare Multi-χ with CIcaL. To resolve this concern, we replace the

Table 4: The impact of representation (GCJ_Java dataset)

Year	Improvement_w			Improvement_t			Delta		
	Acc	F1	Mcc	Acc	F1	Mcc	Acc	F1	Mcc
2013	0.21	0.19	0.40	0.15	0.13	0.32	0.06	0.07	0.08
2014	0.13	0.12	0.26	0.10	0.09	0.23	0.03	0.03	0.03
2015	0.05	0.06	0.07	0.00	0.00	0.00	0.05	0.06	0.07
2016	0.10	0.10	0.17	0.05	0.06	0.12	0.04	0.04	0.05
2017	0.12	0.12	0.121	0.09	0.10	0.20	0.02	0.02	0.01
2018	0.28	0.26	0.57	0.25	0.23	0.52	0.03	0.03	0.05
average	0.15	0.14	0.28	0.11	0.10	0.23	0.04	0.04	0.05

word2vec encoder of CICAL with TF-IDF, and compare the results after the replacement. Meanwhile, word2vec is trained for encoding natural language texts, but code2vec is trained for encoding source code. If we update the encoder of CICAL from word2vec to code2vec, the results of CICAL should be even better. As the improvements are clear, we do not replace our encoder with code2vec in this RQ. The two encoders are as follows:

In TF-IDF, a term t in file d of a corpus D is assigned a weight using $TF - IDF(t, d, D) = TF(t, d) \times IDF(t, D)$, where $TF(t, d)$ is the term frequency (TF) of t in d and $IDF(t, D) = \log\left(|D| / DF(t, D)\right) + 1$, where $|D|$ is the number of documents in D and $DF(t, D)$ is the number of documents containing the term t. We represent code segments with the top 3,000 TF-IDF features based on the order of term frequencies across all code segments.

In word2vec, source code is encoded to vectors. In particular, segments with n lines are represented as a sequence of size $n \times Linecommon \times d$, where d is the dimension of terms. For example, a segment with one line is represented as a tensor of size $1 \times 100 \times 128$, since $Linecommon = 100$ and the dimension of word2vec representations is 128.

5.4.2. Result

Tables 3 and 4 show the results. Columns "**Improvement_w**" list the improvement of CICAL compared to Multi-χ. Columns "**Improvement_t**" list the improvement of CICAL using TF-IDF representation compared to Multi-χ. Columns "**Delta**" list the delta between Columns "**Improvement_w**" and Columns "**Improvement_t**". After the encoders are aligned, CICAL improves the average accuracy, f1 score, and MCC by 0.11, 0.10, and 0.23 on the Java dataset, respectively. On the C++ dataset, the improvements are 0.09, 0.09, and 0.18, respectively. Compared with Multi-χ, CICAL us-

ing Word2vec representation improves the average accuracy, f1 score, and MCC by 0.15, 0.14, and 0.28 on the Java dataset, respectively. On the C++ dataset, the improvements are 0.12, 0.13, and 0.23, respectively. The results show that CIcaL using TF-IDF representation still improves Multi-χ compared to CIcaL using Word2vec representation. The above observations lead to a finding:

> **Result 7:** CIcaL achieves better results than Multi-χ even if its encoder is aligned with Multi-χ.

As the encoder has only minor contributions, the findings in RQs 1 and 2 are reliable and our improvements are mainly caused by incremental learning.

6. Limitations and Future Work

In this section, we analyze our limitations and how to improve our approach in future work.

Limitation 1: No existing approach can identify unseen code authors to the best of our knowledge. According to Results 5 and 6, CIcaL significantly improves the prior approach [11] when handling imbalanced data. Although it is difficult to identify unseen code authors, it should be feasible to identify code authors with few observations. Still, we do not evaluate our approach in such extreme cases. To learn from a few observations, an approach should keep learning and recognizing new classes with few labeled instances. In image [34] and language domains [35], some approaches can resolve the few-shot learning problem, and some other few-shot techniques [36, 37] can also be helpful. In future work, we plan to borrow their ideas and identify rarely-seen code authors.

Limitation 2: In our benchmark, each source file contains modifications from only a code author. This setting is used in the evaluations of all prior approaches [6, 5, 11]. Theoretically, it is feasible to identify multiple code authors for a given source file if we divide this source file into code segments. Gong and Zhong [10] build such a benchmark, in which each source file can have multiple code authors. However, Li *et al.* [11] report that this benchmark is too challenging for even advanced techniques. To make a meaningful comparison, we reuse the benchmark built by Li *et al.* [11]. As this benchmark is less challenging, the effectiveness of all approaches can be reduced. We plan to try more challenging benchmarks in future work.

Limitation 3: We use naive techniques to encode source code. When encoding source code, researchers [2, 3] use explicitly designed language-specific features. For example, Yang *et al.* [3] define 19 software metrics of Java programs to identify code authors. Caliskan *et al.* [2] define 30 software metrics of C and C++ programs. To work with code in various programming languages in a uniform way, Abuhamad *et al.* [6, 5] represent code using word2vec and TF-IDF, respectively. Bogomolov *et al.* [7] represent code using path-based representation [38]. As a learning method for generating distributed representations of tokens, word2vec has shown remarkable success in a wide range of source code embedding applications (*e.g.*, [23, 24, 25, 39]). Some more advanced techniques like CodeBERT [40] and CodeT5 [41] are more suitable to encode source code. However, we use word2vec to encode code since we must make a fair comparison with prior approaches. Result 7 shows that the difference between word2vec and TF-IDF is minor. Even if we use word2vec and TimeDA uses a better model like code2vec, our result is already better, which highlights the significance of our incremental learning framework. In addition, other code metrics like identifier lengths and word compound separations can also be useful and worthy of further exploration.

Limitation 4: We did not evaluate the impact of the chunk size. To align our setting with that of TimeDA [11], we set the chunk size as one year. This setting is not optimized, and other chunk sizes can further improve our results. Even if this parameter is not optimized, we have achieved better results than TimeDA. In future work, we will explore its impact.

7. Related Work

Our work is related to the following research topics:

Source code authorship attribution. Considering the coding style of programmers may evolve during their studies, Caliskan *et al.* [2] extracted a set of 25 programmers from 2012 who are also contestants in 2014's competition. They trained a random forest classifier on some files from these 25 programmers' submissions in 2012 and tested the trained model on the instances from 2014. They concluded that coding style is preserved up to some degree throughout the years. Abuhamad *et al.* [5] used the programs from 2014 to 2015 as the training set and the programs from 2015 to 2016 as the testing set to train and test their method. According to their results, they concluded that the temporal effect on their accuracy is minor. However, Bogomolov *et al.* [7] conducted an empirical study to explore the influence

of time on their proposed approach, they found that as programmers' coding practices evolve over time, learning on older contributions to attribute authorship of the new code leads to a lower accuracy of attribution on a span of several months to years. Only a recent approach [11] offered a solution to reduce the impact of time on code authorship attribution. This approach [11] proposed to use a transfer learning-based approach to reduce the impact of time. In this paper, we propose to use an incremental learning-based approach to reduce the impact of time on code authorship attribution.

Code provenance analysis. In art and antiques, the term "provenance" is used to denote a set of evidence as to the origin and history of an artifact. Increasingly, the term provenance is being used within the context of software development. Developers, managers, QA team members, and other stakeholders often wish to understand how and why a feature, component, chunk of code, test suite, or other development artifact came to be where it is [42]. Davies *et al.* [43] introduce the general concept of software Bertillonage, a method to reduce the search space when trying to locate a software entity's origin within a corpus of possibilities. Rousseau *et al.* [44] conduct an empirical study to explore the possibilities to track provenance of software source code artifacts within the largest publicly accessible corpus of publicly available source code. As introduced by Li *et al.* [45], code provenance techniques including clone detection [46, 47, 48] and authorship attribution [9, 49, 50, 5].

Code ownership analysis. As defined by Hattori *et al.* [51], the ownership of a programmer on a file quantifies the amount of knowledge the programmer has on this file. Greiler *et al.* [52] mine the relation between code ownership and software quality. Diaz *et al.* [53] use code ownership to assist the recovery of links between source files and high-level designs (*e.g.*, use cases). Other researchers [54, 55] analyze the relationship between code ownership and software quality. Penta and German [56] reveal that explicit contributors and copyright owners are not necessarily the most frequent committers. The concepts of code owners and code authors are related. We notice that some approaches use similar techniques to determine the owner of a file. For example, Corley *et al.* [57] extract all the commits on a file and add all their authors to the owners of the file. Their strategy is identical to Meng *et al.* [58]. In our study, we propose CIcaL to identify authors of source files, whose results can be also useful to identify code owners.

Transfer learning in software engineering. The goal of transfer learning is to make use of data from a source domain, which would corre-

spond to past data, in order to improve model predictions on a different, but related dataset known as the target domain, which would correspond to recent data [59]. Transfer learning can help deal with the changes in data distribution associated with concept drift in software engineering. For software vulnerability detection, traditional machine learning-based approaches suffers from the concept drift problem because the training data and test data can from different projects or they differ in the types of vulnerability. Therefore, some approaches [60, 61, 62] proposed to use domain adaptation to deal with the concept drift problem. García *et al.* [63] applied transfer learning to malware detection since the concept drift problem also exists in malware detection. Du*et al.* [64] also applied transfer learning to cross-project bug type prediction. Our positive results indicate that incremental learning can improve the above approaches.

Incremental learning in software engineering. Researchers from both cybersecurity domain and software engineering field proposed using incremental learning to address the concept drift issue in malware detection [65, 66, 67]. Chen *et al.* [66] mentioned that machine learning methods can detect malware with very high accuracy. However, these classifiers have an Achilles heel, concept drift [68, 69]: they rapidly become out of date and ineffective, due to the evolution of apps. Bhattacharya *et al.* [70] applied incremental learning to improve triaging accuracy in bug triaging. Wang *et al.* [71] also applied incremental learning to learn and process real-time software data streams in software defect prediction. Weyssow *et al.* [72] conducted an empirical study to demonstrate that incremental learning can effectively mitigate catastrophic forgetting in pre-trained language models across both API call and API usage prediction tasks. We are the first to introduce incremental learning in identifying code authors.

Identifying human-authored and ChatGPT-generated code. The ubiquitous adoption of Large Language Generation Models (LLMs) in programming has underscored the importance of differentiating between human-written code and code generated by intelligent models. Bukhari *et al.* [73] attempt to use machine learning to distinguish between 28 student-authored and 30 AI-generated solutions for a C-language programming assignment involving singly-linked lists. Their approach leverages lexical and syntactic features in conjunction with multiple machine-learning models, achieving an accuracy rate of 92%. Li *et al.* [74] propose a discriminative feature set in differentiating ChatGPT-generated code from human-authored code in binary classification tasks. Bukhari *et al.* [75] construct a classifier for de-

tecting GPT-4 generated Python code, using XGBoost and a collection of 140 code-stylometry features. Current Approaches of distinguishing human-written code and ChatGPT-generated code still follow the idea of traditional code authorship identification, which is to extract or construct features to capture the code style of human and ChatGPT, and then build machine learning-based classifiers. The findings of our empirical study can be useful to remind researchers to consider the impact of time and the class imbalance problem when developing approaches to distinguish human-written code and ChatGPT-generated code.

8. Conclusion

Authorship attribution of source code has applications in software engineering tasks related to software maintenance, software quality analysis, and plagiarism detection. While recent studies of authorship attribution report high accuracy values, they ignore the temporal effect and class imbalance problem. From the practical viewpoint, the temporal effect and class imbalance are two inevitable problems of learning from source code. The composition of these two phenomena will make learning from source code challenging. In this paper, we have proposed CIcaL to solve the problem of learning from an imbalanced data stream with the temporal effect. It creates a base classifier for each chunk and weighs them by their performance tested on the current chunk. Thus, a classifier trained recently or on a similar concept as the current chunk will receive high weight in the ensemble to help prediction. The evaluation results have shown that CIcaL performs better and is more stable compared with the prior approaches.

Acknowledgement

We appreciate reviewers for their insightful comments. Hao Zhong is the corresponding author. This work is sponsored by the National Natural Science Foundation of China Nos. 62232003 and 62272295.

References

[1] A. Caliskan, F. Yamaguchi, E. Dauber, R. Harang, K. Rieck, R. Greenstadt, A. Narayanan, When coding style survives compilation: De-anonymizing programmers from executable binaries, in: NDSS, 2018.

[2] A. Caliskan-Islam, R. Harang, A. Liu, A. Narayanan, C. Voss, F. Yamaguchi, R. Greenstadt, De-anonymizing programmers via code stylometry, in: Proc. USENIX Security, 2015, pp. 255–270.

[3] X. Yang, G. Xu, Q. Li, Y. Guo, M. Zhang, Authorship attribution of source code by using back propagation neural network based on particle swarm optimization, PloS one 12 (2017) e0187204.

[4] B. Alsulami, E. Dauber, R. Harang, S. Mancoridis, R. Greenstadt, Source code authorship attribution using long short-term memory based networks, in: Proc. ESORICS, 2017, pp. 65–82.

[5] M. Abuhamad, T. AbuHmed, A. Mohaisen, D. Nyang, Large-scale and language-oblivious code authorship identification, in: Proc. CCS, 2018, pp. 101–114.

[6] M. Abuhamad, T. Abuhmed, D. Nyang, D. Mohaisen, Multi-χ: Identifying multiple authors from source code files, in: Proc. PETS, 2020, pp. 25–41.

[7] E. Bogomolov, V. Kovalenko, Y. Rebryk, A. Bacchelli, T. Bryksin, Authorship attribution of source code: A language-agnostic approach and applicability in software engineering, in: Proc. ESEC/FSE, 2021, pp. 932–944.

[8] J. Petrik, D. Chuda, The effect of time drift in source code authorship attribution: Time drifting in source code-stylochronometry, in: Proc. CompSysTech, 2021, pp. 87–92.

[9] Z. Li, G. Chen, C. Chen, Y. Zou, S. Xu, Ropgen: Towards robust code authorship attribution via automatic coding style transformation, in: Proc. ICSE, 2022, pp. 1906–1918.

[10] S. Gong, H. Zhong, A study on identifying code author from real development, in: Proc. ESEC/FSE, 2022, pp. 1627–1631.

[11] Z. Li, S. Zhao, C. Chen, Q. Chen, Reducing the impact of time evolution on source code authorship attribution via domain adaptation, ACM Transactions on Software Engineering and Methodology (2024).

[12] S. Bozinovski, A. Fulgosi, The influence of pattern similarity and transfer learning upon training of a base perceptron b2, in: Proceedings of Symposium Informatica, volume 3, 1976, pp. 121–126.

[13] S. Burrows, A. L. Uitdenbogerd, A. Turpin, Temporally robust software features for authorship attribution, in: Proc. COMPSAC, volume 1, IEEE, 2009, pp. 599–606.

[14] N. D. Hansen, C. Lioma, B. Larsen, S. Alstrup, Temporal context for authorship attribution, in: Proc. IRFC, 2014, pp. 22–40.

[15] S. Gong, H. Zhong, Code authors hidden in file revision histories: An empirical study, in: Proc. ICPC, 2021, pp. 71–82.

[16] R. Raina, A. Y. Ng, D. Koller, Constructing informative priors using transfer learning, in: Proc. ICML, 2006, pp. 713–720.

[17] S. J. Pan, J. T. Kwok, Q. Yang, J. J. Pan, Adaptive localization in a dynamic wifi environment through multi-view learning, in: Proc. AAAI, volume 7, 2007, pp. 1108–1113.

[18] H. Ding, M. H. Samadzadeh, Extraction of java program fingerprints for software authorship identification, Journal of Systems and Software 72 (2004) 49–57.

[19] A. Gepperth, B. Hammer, Incremental learning algorithms and applications, in: European symposium on artificial neural networks (ESANN), 2016.

[20] S. J. Pan, Q. Yang, A survey on transfer learning, IEEE Transactions on knowledge and data engineering 22 (2009) 1345–1359.

[21] Z. Qiao, Q. Pham, Z. Cao, H. H. Le, P. Suganthan, X. Jiang, R. Savitha, Class-incremental learning for time series: Benchmark and evaluation, arXiv preprint arXiv:2402.12035 (2024).

[22] K. W. Church, Word2vec, Natural Language Engineering 23 (2017) 155–162.

[23] J. A. Harer, L. Y. Kim, R. L. Russell, O. Ozdemir, L. R. Kosta, A. Rangamani, L. H. Hamilton, G. I. Centeno, J. R. Key, P. M. Ellingwood, et al., Automated software vulnerability detection with machine learning, arXiv preprint arXiv:1803.04497 (2018).

[24] M. White, M. Tufano, M. Martinez, M. Monperrus, D. Poshyvanyk, Sorting and transforming program repair ingredients via deep learning code similarities, in: Proc. SANER, 2019, pp. 479–490.

[25] D. Azcona, P. Arora, I.-H. Hsiao, A. Smeaton, user2code2vec: Embeddings for profiling students based on distributional representations of source code, in: Proc. LAK, 2019, pp. 86–95.

[26] P. Bojanowski, E. Grave, A. Joulin, T. Mikolov, Enriching word vectors with subword information, Transactions of the association for computational linguistics 5 (2017) 135–146.

[27] B. S. Raghuwanshi, S. Shukla, Class imbalance learning using underbagging based kernelized extreme learning machine, Neurocomputing 329 (2019) 172–187.

[28] J. Z. Kolter, M. A. Maloof, Dynamic weighted majority: An ensemble method for drifting concepts, The Journal of Machine Learning Research 8 (2007) 2755–2790.

[29] D. Chicco, G. Jurman, The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation, BMC genomics 21 (2020) 1–13.

[30] J. Yao, M. Shepperd, Assessing software defection prediction performance: Why using the matthews correlation coefficient matters, in: Proc. EASE, 2020, pp. 120–129.

[31] M. Schuster, K. K. Paliwal, Bidirectional recurrent neural networks, IEEE transactions on Signal Processing 45 (1997) 2673–2681.

[32] A. Géron, Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow: Concepts, tools, and techniques to build intelligent systems, O'Reilly Media, 2019.

[33] Q. McNemar, Note on the sampling error of the difference between correlated proportions or percentages, Psychometrika 12 (1947) 153–157.

[34] M. Masana, X. Liu, B. Twardowski, M. Menta, A. D. Bagdanov, J. Van De Weijer, Class-incremental learning: survey and performance evaluation on image classification, IEEE Transactions on Pattern Analysis and Machine Intelligence 45 (2022) 5513–5533.

[35] Z. Ke, B. Liu, Continual learning of natural language processing tasks: A survey, arXiv preprint arXiv:2211.12701 (2022).

[36] C. Xia, W. Yin, Y. Feng, P. Yu, Incremental few-shot text classification with multi-round new classes: Formulation, dataset and system, arXiv preprint arXiv:2104.11882 (2021).

[37] J.-G. Zhang, K. Hashimoto, W. Liu, C.-S. Wu, Y. Wan, P. S. Yu, R. Socher, C. Xiong, Discriminative nearest neighbor few-shot intent detection by transferring natural language inference, arXiv preprint arXiv:2010.13009 (2020).

[38] U. Alon, M. Zilberstein, O. Levy, E. Yahav, code2vec: Learning distributed representations of code, Proc. PACMPL 3 (2019) 1–29.

[39] Z. Chen, M. Monperrus, The remarkable role of similarity in redundancy-based program repair, arXiv preprint arXiv:1811.05703 (2018).

[40] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, et al., Codebert: A pre-trained model for programming and natural languages, arXiv preprint arXiv:2002.08155 (2020).

[41] Y. Wang, W. Wang, S. Joty, S. C. Hoi, Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation, arXiv preprint arXiv:2109.00859 (2021).

[42] M. W. Godfrey, Understanding software artifact provenance, Science of Computer Programming 97 (2015) 86–90.

[43] J. Davies, D. M. German, M. W. Godfrey, A. Hindle, Software bertillonage: Determining the provenance of software development artifacts, Empirical Software Engineering 18 (2013) 1195–1237.

[44] G. Rousseau, R. Di Cosmo, S. Zacchiroli, Software provenance tracking at the scale of public source code, Empirical Software Engineering 25 (2020) 2930–2959.

[45] W. Li, B. Yang, Y. Sun, S. Chen, Z. Song, L. Xiang, X. Wang, C. Zhou, Towards tracing code provenance with code watermarking, arXiv preprint arXiv:2305.12461 (2023).

[46] C. Liu, Z. Lin, J.-G. Lou, L. Wen, D. Zhang, Can neural clone detection generalize to unseen functionalitiesf, in: Proc. ASE, 2021, pp. 617–629.

[47] A. Eghbali, M. Pradel, Crystalbleu: precisely and efficiently measuring the similarity of code, in: Proc. ASE, 2022, pp. 1–12.

[48] H. Sajnani, V. Saini, J. Svajlenko, C. K. Roy, C. V. Lopes, Sourcerercc: Scaling code clone detection to big-code, in: Proc. ICSE, 2016, pp. 1157–1168.

[49] W. Ou, S. H. Ding, Y. Tian, L. Song, Scs-gan: Learning functionality-agnostic stylometric representations for source code authorship verification, IEEE Transactions on Software Engineering 49 (2022) 1426–1442.

[50] Q. Liu, S. Ji, C. Liu, C. Wu, A practical black-box attack on source code authorship identification classifiers, IEEE Transactions on Information Forensics and Security 16 (2021) 3620–3633.

[51] L. P. Hattori, M. Lanza, R. Robbes, Refining code ownership with synchronous changes, Empirical Software Engineering 17 (2012) 467–499.

[52] M. Greiler, K. Herzig, J. Czerwonka, Code ownership and software quality: a replication study, in: Proc. MSR, 2015, pp. 2–12.

[53] D. Diaz, G. Bavota, A. Marcus, R. Oliveto, S. Takahashi, A. De Lucia, Using code ownership to improve ir-based traceability link recovery, in: Proc. ICPC, 2013, pp. 123–132.

[54] P. Thongtanunam, S. McIntosh, A. E. Hassan, H. Iida, Revisiting code ownership and its relationship with software quality in the scope of modern code review, in: Proc. ICSE, 2016, pp. 1039–1050.

[55] C. Bird, N. Nagappan, B. Murphy, H. Gall, P. Devanbu, Don't touch my code! examining the effects of ownership on software quality, in: Proc. ESEC/FSE, 2011, pp. 4–14.

[56] Y. Kamei, S. Matsumoto, A. Monden, K. I. Matsumoto, B. Adams, A. E. Hassan, Revisiting common bug prediction findings using effort-aware models, in: Proc. ICSM, 2010.

[57] C. S. Corley, E. A. Kammer, N. A. Kraft, Modeling the ownership of source code topics, in: Proc. ICPC, 2012, pp. 173–182.

[58] X. Meng, B. P. Miller, K.-S. Jun, Identifying multiple authors in a binary program, in: Proc. ESORICS, 2017, pp. 286–304.

[59] S. J. Pan, Q. Yang, A survey on transfer learning, IEEE Transactions on knowledge and data engineering 22 (2010) 1345–1359.

[60] V. Nguyen, T. Le, T. Le, K. Nguyen, O. DeVel, P. Montague, L. Qu, D. Phung, Deep domain adaptation for vulnerable code function identification, in: Proc. IJCNN, 2019, pp. 1–8.

[61] V. Nguyen, T. Le, O. de Vel, P. Montague, J. Grundy, D. Phung, Dual-component deep domain adaptation: A new approach for cross project software vulnerability detection, in: Proc. PAKDD, 2020, pp. 699–711.

[62] S. Liu, G. Lin, L. Qu, J. Zhang, O. De Vel, P. Montague, Y. Xiang, Cd-vuld: Cross-domain vulnerability discovery based on deep domain adaptation, IEEE Transactions on Dependable and Secure Computing 19 (2020) 438–451.

[63] D. E. García, N. DeCastro-García, A. L. M. Castañeda, An effectiveness analysis of transfer learning for the concept drift problem in malware detection, Expert Systems with Applications 212 (2023) 118724.

[64] X. Du, Z. Zhou, B. Yin, G. Xiao, Cross-project bug type prediction based on transfer learning, Software Quality Journal 28 (2020) 39–57.

[65] Q. Qiang, M. Cheng, Y. Hu, Y. Zhou, J. Sun, Y. Ding, Z. Qi, F. Jiao, An incremental malware classification approach based on few-shot learning, in: Proc. ICC, 2022, pp. 2682–2687.

[66] Y. Chen, Z. Ding, D. Wagner, Continuous learning for android malware detection, in: Proc. USENIX Security, 2023, pp. 1127–1144.

[67] A. Narayanan, L. Yang, L. Chen, L. Jinliang, Adaptive and scalable android malware detection through online learning, in: Proc. IJCNN, 2016, pp. 2484–2491.

[68] L. Yang, W. Guo, Q. Hao, A. Ciptadi, A. Ahmadzadeh, X. Xing, G. Wang, {CADE}: Detecting and explaining concept drift samples for security applications, in: Proc. USENIX Security, 2021, pp. 2327–2344.

[69] R. Jordaney, K. Sharad, S. K. Dash, Z. Wang, D. Papini, I. Nouretdinov, L. Cavallaro, Transcend: Detecting concept drift in malware classification models, in: Proc. USENIX Security, 2017, pp. 625–642.

[70] P. Bhattacharya, I. Neamtiu, Fine-grained incremental learning and multi-feature tossing graphs to improve bug triaging, in: Proc. ICSM, 2010, pp. 1–10.

[71] S. Wang, Y. Li, W. Mi, Y. Liu, Software defect prediction incremental model using ensemble learning, International Journal of Performability Engineering 16 (2020) 1771.

[72] M. Weyssow, X. Zhou, K. Kim, D. Lo, H. Sahraoui, On the usage of continual learning for out-of-distribution generalization in pre-trained language models of code, in: Proc. ESEC/FSE, 2023, pp. 1470–1482.

[73] S. Bukhari, B. Tan, L. De Carli, Distinguishing ai-and human-generated code: a case study, in: Proc. SCORED, 2023, pp. 17–25.

[74] K. Li, S. Hong, C. Fu, Y. Zhang, M. Liu, Discriminating human-authored from chatgpt-generated code via discernable feature analysis, in: Proc. ISSREW, 2023, pp. 120–127.

[75] O. J. Idialu, N. S. Mathews, R. Maipradit, J. M. Atlee, M. Nagappan, Whodunit: Classifying code as human authored or gpt-4 generated-a case study on codechef problems, in: Proc. MSR, 2024, pp. 394–406.