Empirical Software Engineering _####################_
https://doi.org/10.1007/s10664-026-10819-1

How challenging it is to identify real code authors: an empirical study

Siyi Gong[1] · **Hao Zhong**[1]

Received: 6 November 2023 / Accepted: 30 January 2026

Abstract
Identifying code authors is an important research topic in software forensics and plagiarism detection. Although researchers reported promising results on their datasets, the true effectiveness of real source files is still unknown. However, the single prior large-scale study selected Google Code Jam programs as their subjects, but such programs are quite different from the source files that programmers write in daily development. To understand their effectiveness and challenges, we replicate their study with source files that are retrieved from real projects. Our study produces many different results. For example, we find that in 85.48% of pairs of training and testing sets, the accuracy of a trained model is less effective when the temporal effect is considered, and in total, the average accuracy decreases by 42.98%. Our study confirms that instead of a solved problem, identifying code authors for real source files is rather challenging, and many questions are still open.

Keywords Code authorship attribution · Coding style evolution · Empirical study

1 Introduction

When implementing malware, programmers often hide their true identities. Even if code authors do not hidden their identities on purpose, 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

Communicated by: Rafael Prikladnicki.

✉ Hao Zhong
zhonghao@sjtu.edu.cn

Siyi Gong
gongsiyi@sjtu.edu.cn

[1] Shanghai Key Laboratory of Trusted Data Circulation and Governance, and Web3, Department of Computer Science and Engineering, Shanghai Jiao Tong University, Shanghai, China

recorded. Due to the above considerations, researchers (Alsulami et al. 2017; Caliskan-Islam et al. 2015; Yang et al. 2017) have proposed various approaches to identify code authors. Given a code snippet, the task of identifying its author is known as code authorship attribution. This task is important in various other research topics (e.g., bug report assignments Anvik et al. 2006, software forensics Spafford and Weeber 1993, and plagiarism detection Parker and Hamblen 1989). For example, if the author of a piece of buggy code is identified, assigning the corresponding bug report to this code author is straightforward. As another example, for software forensics, the identification of code authors is useful to trace malware samples.

Although identifying code authors is critical in many scenarios, it has many open challenges. For example, in recent studies, researchers report that the code styles of code authors can change over time (Abuhamad et al. 2018; Bogomolov et al. 2021; Kalgutkar et al. 2019; Petrik and Chuda 2021), and the distribution of files can affect their effectiveness (Abuhamad et al. 2018). In the above studies, researchers evaluate approaches on the Google Code Jam dataset. To build this dataset, researchers collect programs from a programming competition hosted by Google, since their labels are recorded. Although these programs are useful to build datasets, these programs are quite different from those in real development. For instance, Gong and Zhong (2021) report that most source files in real development have more than one code author, but in the Google Code Jam dataset, each source file has only a code author. As a result, although the above studies are large scale, their settings are quite different from the wild, and their findings can be different in real development. As a result, although these studies are interesting, it needs more studies to ensure the reliability of their answers.

To meet the timely needs, in our earlier work (Gong and Zhong 2022), we replicate the prior study (Abuhamad et al. 2018) with two major extensions. First, we select an extended version of the approach that is used in the prior study (Abuhamad et al. 2018). This approach uses a two-layer bi-LSTM to identify code authors. Second, our setting is closer to the real development (see Section 3 for details). For example, the prior study (Abuhamad et al. 2018) mainly uses Google Code Jam programs that are submitted in two years, but we collect 14,657 commits from real projects over twenty years. For example, the prior study (Abuhamad et al. 2018) mainly uses Google Code Jam programs that are submitted in two years, but we collect 14,657 commits from real projects over twenty years. Although flagship projects can have even more commits, we have collected much more complicated data than the above studies. As another example, we design more pairs of training and testing sets to analyze the impact of the time. In our position paper (Gong and Zhong 2022), we pursued the following research question:

– **RQ1.** *How does time affect identifying code authors from real development?*
Motivation. After a model is trained, it inevitably becomes obsolete, since the styles of code authors can evolve over time. The prior study (Abuhamad et al. 2018) shows that the impacts are minor. However, their analyzed source files are not from real development. Meanwhile, as most lines of code are written by several authors (Gong and Zhong 2021), some authors have insufficient files for mining. To explore the impacts, the prior study (Abuhamad et al. 2018) reduces the files per author from nine to five. In their setting, authors have equal files, but when source files are retrieved from a real project, a few authors write many more files than others (Gong and Zhong 2021).

Protocol. To answer this question, we use source files from real development and carefully design the aging setting to explore the impact of the time. Besides the overall results, for each training-testing pair, we calculate the accuracy for each long-term contributor.

Answer. In 85.48% of cases, the accuracy of a trained model is less effective when it becomes obsolete (Finding 2). In total, the average median accuracy decreases by 42.98% (Finding 1). In addition, identifying individual long-term contributors also becomes less effective, when the trained model is obsolete (Finding 3).

In this extended study, we analyze more research questions that are untouched by our early study (Gong and Zhong 2022). Besides the impacts of time and the distribution of files per author, there are other factors with practical values. For example, an author can write code for multiple projects. When an author joins a new project, this project may not contain sufficient history for mining. Although it is feasible to use models trained from other projects, the team of a project can define quite different naming conventions and code styles from other teams (Krsul and Spafford 1997). As a result, a trained model may not work well on other projects. It is interesting to explore to what degree a trained model becomes less effective in this situation. Although the coding style of an author inherently evolves over time and across projects, there can be some features that are more stable than others. This study explores the following research questions:

- **RQ2.** *How effective is a model, if it is trained on a project but used to predict code authors of other projects?*
  Motivation. A programmer might participate in multiple projects for collecting more experience in programming. When a code author can be identified by the model trained from one project, is it feasible to use this trained model to predict this code author of other projects?
  Protocol. To answer this question, we use real code from real development and carefully design the cross-project setting to explore the project's impact on predicting code authors.
  Answer. Finding 5 shows that in 94.46% of cases, the accuracy decreases when a trained model is used on other projects, and Finding 4 shows that the average median accuracy decreases by 61.89%.
- **RQ3.** *How reliable are code metrics over time and across projects?*
  Motivation. As we discussed in our prior study (Gong and Zhong 2021), most existing code authorship identification techniques (Caliskan et al. 2018; Ding and Samadzadeh 2004; Lange and Mancoridis 2007; Shevertalov et al. 2009; Yang et al. 2017) extract a set of code metrics (e.g., the ratio of blank lines) of a given source code that could be used as inputs to identify the author of the given code. These metrics can be seen as a fingerprint of the code author (Ding and Samadzadeh 2004). In this RQ, we locate some software metrics that mitigated the evolution of coding style and cross-project learning when attributing authorship.
  Protocol. To answer this question, we use statistical hypothesis testing to find features that are reliable over time and across projects.
  Answer. Finding 6 shows that only when a project has fewer than three code authors, the stability values over time are more than or equal to 0.5, except for `configuration`.

Finding 7 shows that except for *the ratio of variable naming starting with lowercase letters*, the stability values across projects are zero.

– **RQ4.** *How effective are advanced machine learning techniques?*
Motivation. Transfer learning (Liu et al. 2020; Nguyen et al. 2019, 2020) and incremental learning (Chen et al. 2023; Narayanan et al. 2016; Qiang et al. 2022) have been successfully applied to resolve the challenges of temporal evolution and project shifts, e.g., malware detection. Inspired by their successful story, Li et al. (2024) and we Gong and Zhong (2025) use transfer learning approach and incremental learning to mitigate the two challenges, respectively. Our evaluation results (Gong and Zhong 2025) demonstrate that both approaches effectively reduce temporal effect. However, these conclusions are drawn from the Google Code Jam dataset, and their effectiveness in real development is still unclear. In this research question, we evaluate the two approaches in a more real setting.
Protocol. To address this question, we evaluate (Li et al. 2024) and our approach (Gong and Zhong 2025) on the benchmark in Section 3.1. Different from the Google Code Jam dataset, our benchmark includes data from Github projects.
Answer. Finding 8 shows that the implementation of incremental and transfer learning frameworks effectively mitigates the impact of temporal shifts, resulting in average accuracy improvements of 29.33% and 10.32%, respectively. Finding 9 shows that the implementation of incremental and transfer learning frameworks effectively mitigates the impact of project shifts, resulting in average accuracy improvements of 18.44% and 6.00%, respectively.

2 Preliminary

Researchers already noticed that 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. For example, Burrows et al. (2009) 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. (2014) examine the practical feasibility of using limited and recent writing samples from students for authorship attribution. Caliskan-Islam et al. (2015) 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 (2021) 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. (2021) also demonstrate that the evolution of programming style affects the accuracy of their authorship attribution model. 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. However, there is only a

large study (Abuhamad et al. 2018) conclude that both the time and the files per author have only minor impacts. We next introduce its dataset, settings, and metrics.

As shown in Row "**Dataset**" of Table 1, Abuhamad et al. used Google Code Jam programs (Google Code Jam 2019). Google Code Jam is a code competition. Although Google Code Jam programs are real code, they are quite different from the real source files from the wild. First, most coding problems do not involve complicated dependencies, but real tasks often involve them. Second, the programs from Code Jam programs resolve much simpler tasks than source files from a real project. Third, programs from Code Jam programs are written by individual authors, but source files from a real project are typically written by multiple code authors.

As shown in Row "**Author selection**", Abuhamad et al. require that authors must appear in both the training set and the testing set. In the wild, when the authors are unidentified, it is infeasible to determine whether they appear in the training set. As a result, the requirement is unrealistic. As shown in Row "**Time strategy**", Abuhamad et al. use the programs from 2014 to 2015 as the training set and the programs from 2015 to 2016 as the testing set. The time intervals between their training set and testing set are zero, and the impact of the time is not fully tested. Here, Table 3 of their paper (Abuhamad et al. 2018) lists the programs of only two years, and Table 9 of their paper (Abuhamad et al. 2018) lists only an accuracy value for each technique. They must present more values if they tried more combinations. As shown in Row "**File strategy**", the dataset of Abuhamad et al. is still balanced and different from the real scenarios.

Abuhamad et al. use accuracy as the metric of their study. Indeed, all the other approaches (Abuhamad et al. 2018, 2020; Alsulami et al. 2017; Bogomolov et al. 2021; Caliskan et al. 2018; Caliskan-Islam et al. 2015; Li et al. 2022; Petrik and Chuda 2021; Yang et al. 2017) use this metric to evaluate their effectiveness. As a result, in this study, we select accuracy as the metric. Besides accuracy, we select F1-score (Yao and Shepperd 2020) as our measure.

3 Methodology

This section introduces the methodology of our study.

3.1 Dataset

To build a more realistic usage scenario, we use real projects to build our dataset. As shown in Table 2, we select 5 projects from the Apache Foundation (2018). Column "**Name**" lists project names. We choose these projects since they have long maintenance histories. Col-

Table 1 The comparison of settings

Setting	Abuhamad et al. (2018)	Our study
Dataset	Google Code Jam programs from 2014 to 2016	Real projects from 2001 to 2021
Author selection	Authors must appear in both the training set and the testing set	No requirement
Time strategy	A training set (2014 to 2015) and a testing set (2015 and 2016)	Much more combinations as shown in Fig. 3
File strategy	Five vs seven vs nine files per author	The real distribution

Table 2 Overall dataset

Name	Time	Commit	Author	LOC
Collections	2001-2021	3,559	89	301,881
dbcp	2004-2021	2,027	62	94,764
Configuration	2003-2021	3,415	38	269,997
Compress	2010-2021	2,493	65	87,177
io	2002-2021	2,973	94	118,634
Total	20 years	14,467	221	872,453

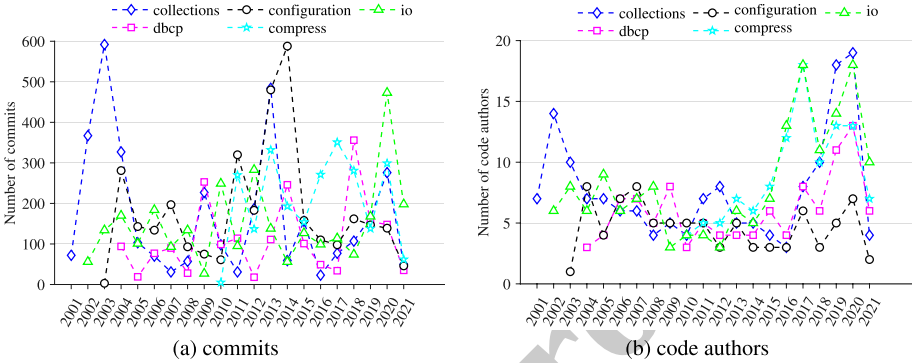

Fig. 1 The distributions of commits and code authors

umn "**Time**" lists the maintained time of the project. In total, these projects have 14,467 commits that were submitted from 2001 to 2021. Column "**Commit**" lists the number of commits. Figure 1 shows the distribution of commits and code authors over time for each project. We inspect the first commits of all the projects. The first commit of collections added 5 files; the first commit of dbcp added 28 files; the first commit of configuration added 49 files; the first commit of compress added 2 files; and the first commit of io added 28 files. Only two commits modify much more files than others. Still, their impacts are minor. As shown in Fig. 3, the first commits are not used anymore when we move our sliding windows. As a result, in most training and testing sets, the first commits are not used.

As the prior studies (Abuhamad et al. 2020; Gong and Zhong 2021) did, from each commit, we extract its added lines as code snippets and assign the author of the commit as the label of these code snippets. In particular, we build the links between the code snippets and their authors with the following steps:

Step 1. Extracting patches of each commit. From the Github code repository of each project, we extract all its commits. From each commit, we store its original and modified source files in a local directory. By comparing the original and modified version of each file, we build a patch. In addition, for each commit, we record its commit time and its author name with an email.

Step 2. Extracting code snippets, authors, and submission time. From each patch, we extract its added code lines to construct a code snippet. The author of the commit is considered as the author of the added lines, e.g., the label of the code snippet, and the submission time is considered as the time of the code snippet.

Step 3. Extracting lexical metrics, code-style metrics and structure and syntax metrics. The author of the commit is considered as the label of the item of metrics, and the submission time is considered as the time of the item of metrics.

Column "**Author**" lists the number of authors. Column "**LOC**" lists the lines of extracted code snippets. Although we use the code authors recorded by GitHub as our gold standard, code attribution approaches are still useful. First, some in-house projects are not hosted on Git, and their authors are not recorded. Second, even if a project is hosted on GitHub, their owners can deliberately delete revision histories, if they intend to steal source files from other projects.

3.2 Subject

We selected the state-of-the-art approach (Abuhamad et al. 2020) as the subject approach since it uses deep learning techniques (BiLSTM Schuster and Paliwal 1997) and is effective on their datasets (an accuracy of 79.62%). Abuhamad et al. (2020) divide source files into small segments. The segments are then encoded as a sequence of n-dimensional data with word2vec. The sequences and their label are used to train the identification model. As Abuhamad et al. (2020) did not release their tool, we implemented the tool upon Keras (Géron 2019), according to their paper (Abuhamad et al. 2020). Their approach is evaluated on balanced inputs, but the distribution in real development is not. To build a similar setting, we select the top ten programmers from Table 2 and construct a balanced dataset. We select the identical parameters as Abuhamad et al. (2020) did. On this dataset and with identical parameters, our implementation achieved an accuracy of 86.38%, and the result is close to what is reported in their paper.

As a deep learning approach, Abuhamad et al. (2020) do not explicitly list its features. To understand which metrics are stable, we select (Yang et al. 2017), since they explicitly list all their metrics to determine code authors.

3.3 Analysis Overview

Figure 2 shows our analysis overview. To explore how the state-of-the-art approaches identify real code authors, we conducted an empirical study on the data set listed in Section 3.1. Our study has three research angles.

RQ1. Exploring the impact of time. To explore the impact of time, we split the commits of each project by time intervals, and build the pairs of training sets and testing sets. We then use our built pairs to train and test the approach (Abuhamad et al. 2020). To explore the impact on individual authors, we calculate the testing accuracy of each long-term contributor with the trained models in RQ1.

RQ2. Exploring the impact of projects. To explore the impact of projects, we split the commits in the same time intervals by project, and we build the pairs of training sets and testing sets. We then use our built pairs to train and test the approach (Abuhamad et al. 2020).

In RQ1 and RQ2, we find that the time and the project both have significant impacts on the state-of-the-art deep learning-based approach (Abuhamad et al. 2020). The results lead to our RQ3, which explores the reliable features over time and across projects. We do not analyze the approach (Abuhamad et al. 2020) in RQ1 and RQ2, since it does not explic-

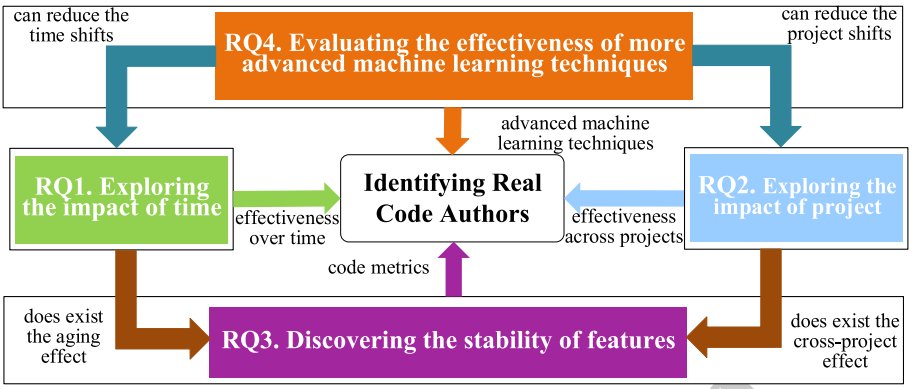

Fig. 2 Analysis overview

itly define code metrics. Instead, we analyze (Yang et al. 2017) in RQ3, since it explicitly defines all its used metrics.

RQ3. Discovering the reliability of features. In RQ3, we use ANOVA (St et al. 1989) and Kruskal-Wallis test to check whether the code metrics defined by Yang et al. (2017) can distinguish code authors over time and across projects.

Our findings in RQ3 indicate that no fully reliable code metric exists to consistently distinguish code authors over time and across projects. A promising alternative is to adopt incremental learning, enabling models to adapt continuously amid rapidly evolving software projects. This insight motivates RQ4, in which we evaluate the effectiveness of incremental learning in mitigating the impact of both temporal changes and project shifts on code author identification.

RQ4. Evaluating the effectiveness of more advanced machine learning techniques. In RQ4, we evaluate two recent approaches, CIcaL (Gong and Zhong 2025) and TimeDA (Li et al. 2024). The two approaches use incremental learning and transfer learning, respectively.

4 Result

This section presents our detailed protocols and results. More details are listed on our project website: DOI: https://doi.org/10.5281/zenodo.8058290

4.1 RQ1. The Impact of the Time

4.1.1 Protocol

We select the stratified 10-fold cross-validation (Parsons 2014) as the compared setting, since it is popular (Piryonesi and El-Diraby 2020; Willis and Riley 2017). Compared with the classical cross-validation (Browne 2000), the stratified cross-validation uses stratified sampling (Parsons 2014) to ensure that the training set and the testing set have the same proportion of labeled data as the original dataset has. In each iteration, we split the submitted

code snippets and the items of metrics of a whole year into ten groups. In each fold, we use 9 groups as the training set and the remaining 1 group as the validation set. From each training set, we train 10 models, and we collect their validation accuracy values as the baseline, as the prior approaches (Abuhamad et al. 2018, 2020) did.

Figure 3 shows our settings to explore the impact of the time. To train each model, we use the commits of a whole year. For example, in Fig. 3, the first and last commit of the training set are submitted on Jan. 16, 2002, and Dec. 31, 2002, respectively. We use the same parameters as the baseline to train the model. A limitation of the prior study (Abuhamad et al. 2018) is that the time interval from their training set and their testing sets are zero and one year, and their study does not fully explore the impacts when the interval increases. In our study, as shown in Fig. 3, we increase the time interval from 3 months to 48 months, with three months as a gap. To obtain more pairs of training sets and testing sets, after each iteration, we apply a time-sliding window. In particular, we move the whole training set to that of six months later. After all the models are trained and tested, we move the first commit of the training set to 6 months later, as shown in the sliding window of Fig. 3. For collections, we move the training set 18 times, and we get 19 pairs of training and testing sets; For dbcp, we move the training set 14 times, and we get 15 pairs of training and testing sets; For configuration, we move the training set 22 times, and we get 23 pairs of training and testing sets; For compress, we move the training set 9 times, and we get 10 pairs of training and testing sets; For io, we move the training set 25 times, and we get 26 pairs of training and testing sets. Here, different from Abuhamad et al. (2018), we do not remove unseen authors from the testing set.

To simulate the situation where each author has different files, the prior study (Abuhamad et al. 2018) reduces the files per author from nine to five, but in real development, a few core programmers write most code lines (Gong and Zhong 2021). In our study, we use the real distribution to train and test the models. After that, we record the accuracy for individual long-term contributors. In prior studies, researchers identify core members by their commit contribution (Canedo et al. 2020), privileged issue events (Bock et al. 2023), and the truck factor (Ferreira et al. 2017). In our study, we need to identify authors who have long-term contributions to a project. As Bao et al. (2019) did, we select authors whose first and last commit has a gap of more than one year. We call such authors long-term contributors. We select long-term contributors to analyze the temporal effect. If programmers contribute only within a short period, we cannot split their data by time internals.

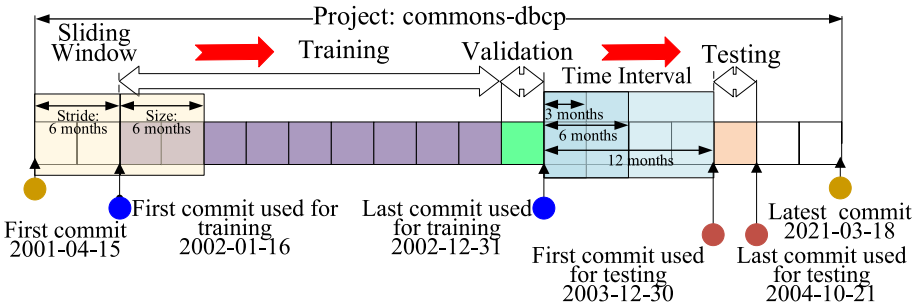

Fig. 3 The protocol of RQ1

4.1.2 Result

Figure 4 shows the box plots of the two settings. The horizontal axes list when the first commits of training sets are submitted. In each group, the cross-validation setting shows the accuracy values of the ten models that are trained from a training set, and the aging setting shows the accuracy values when we increase the time interval. Except for the four exceptions in `configuration` and the one exception in `collections`, in 94.62% (88 out of

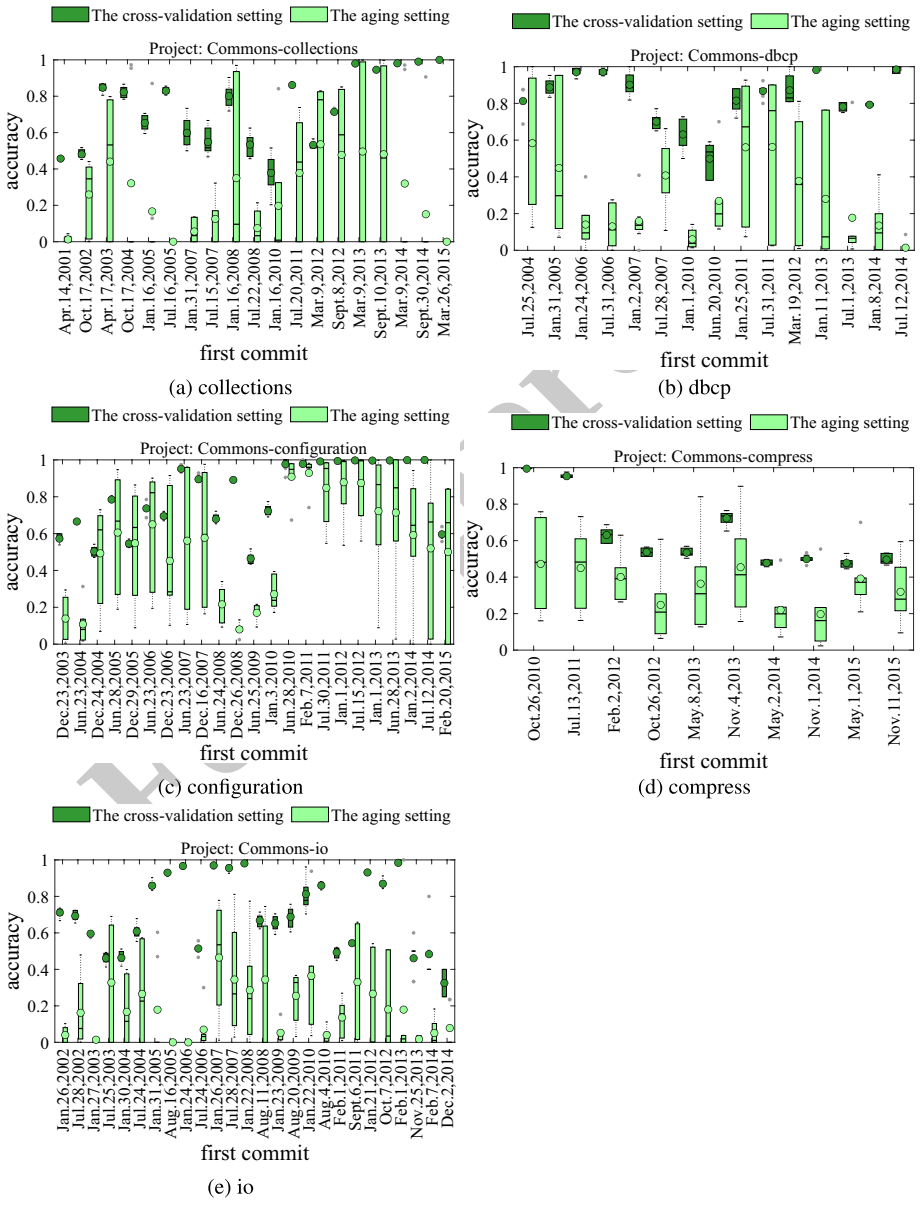

Fig. 4 The overall impact of the time (accuracy)

93) cases, the medians of the cross-validation setting are higher than those of the aging setting. We further calculate the average median reductions from the cross-validation setting to the aging setting. The reductions are 51.49% (`collections`), 56.10% (`dbcp`), 22.43% (`configuration`), 29.89% (`compress`), and 54.99% (`io`). For all the projects, the average reduction is 42.98%. The above observations lead to our first finding:

Finding 1. When the time issue is considered, the average median reduction of accuracy values is 42.98%.

As shown in Fig. 4, in several cases, the aging setting produces even better accuracy values than the cross-validation setting. For example, on Jun. 23, 2006, of `configuration`, the median of the cross-validation setting is even lower than that of the aging setting, the difference is 8.79%. We calculate the cases where the accuracy of the aging setting is more than the median of the cross-validation setting. The results are 15.79% (`collections`), 13.33% (`dbcp`), 25.36% (`configuration`), 11.67% (`compress`), and 5.77% (`io`). In total, such cases account for 14.52% of all cases. After inspecting those cases, we find that the distributions of authors lead to those extreme results. For example, for the data points on Jan. 31st, 2005 of `dbcp`, the training set mainly contains the code snippets of Dir* (58 lines of code) and Phi* (950 lines of code). After three months, the testing set mainly contains the code snippets of Dai* (2 lines of code) and Phi* (40 lines of code). The trained model predicts most authors as Phi*, but still produces high accuracy since the data are imbalanced (see below for more discussions). The observations lead to another finding:

Finding 2. After the time issue is considered, in more than 85.48% of training and testing pairs, the accuracy values of the aging setting become lower than the cross-validation setting.

Figure 5 shows the F1-score results of the two settings. In each group, the cross-validation setting shows the F1-score values of the ten models that are trained from a training set, and the aging setting shows the F1-score values when we increase the time interval. Except for the seven exceptions in `configuration`, the one exception in `collections`, the one exception in `dbcp`, in 90.32% (84 out of 93) cases, the medians of the cross-validation setting are higher than those of the aging setting. We further calculate the average median reductions from the cross-validation setting to the aging setting. The reductions are 50.99% (`collections`), 46.98% (`dbcp`), 12.41% (`configuration`), 22.20% (`compress`), and 54.62% (`io`). For all the projects, the average reduction is 37.41%. As the F1-score and accuracy scores are consistent. Due save space, we present only the accuracy scores in the following research questions.

Figure 6 shows the results of the long-term contributors. Except for `compress`, the trained models become less effective when the time interval is larger. For example, in `dbcp`, when the time interval is 3-months, the trained model predicts the code of Phi* and Mar* accurately, but when it is 6-months, the trained model predicts accurately only the code of Phi*. The observations lead to the following finding:

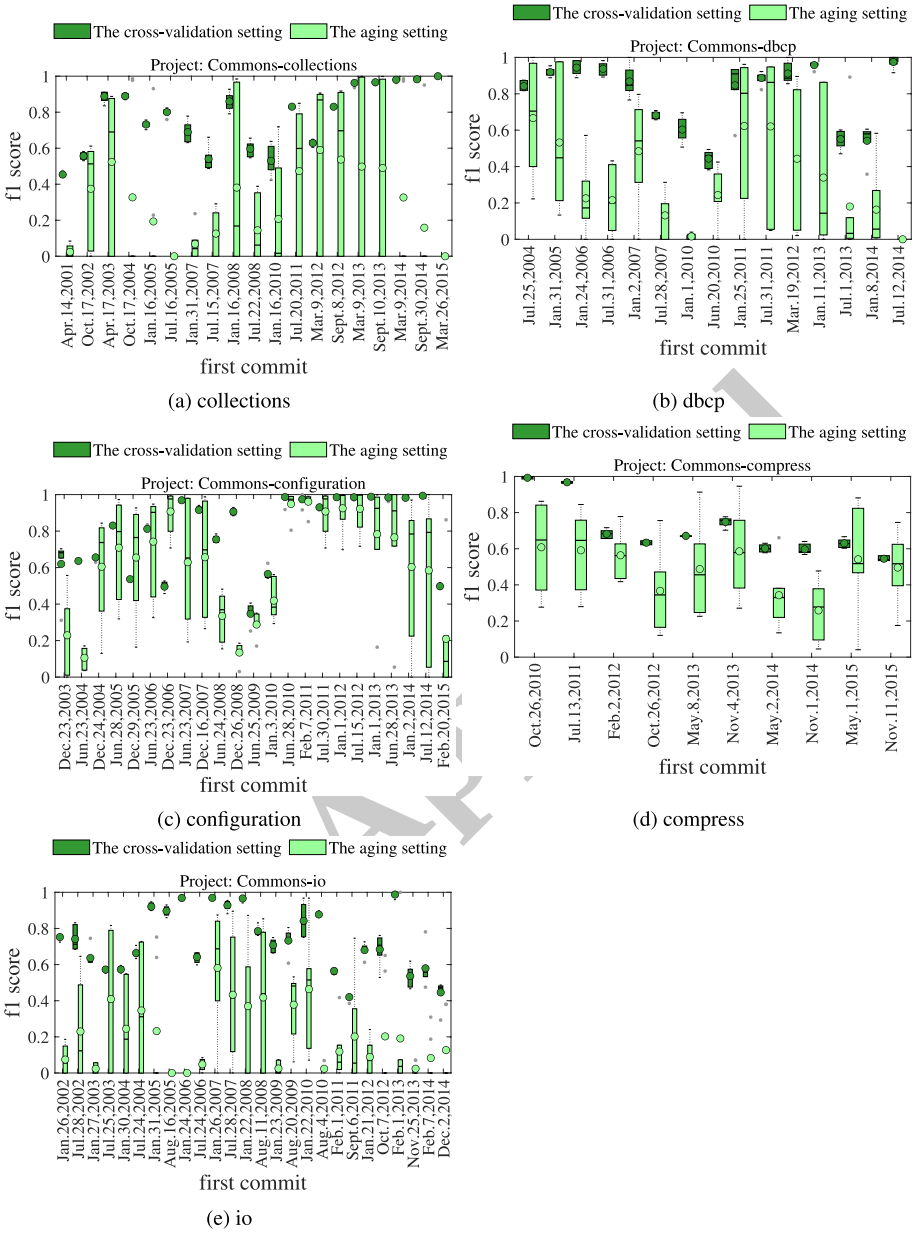

Fig. 5 The overall impact of the time (f1 score)

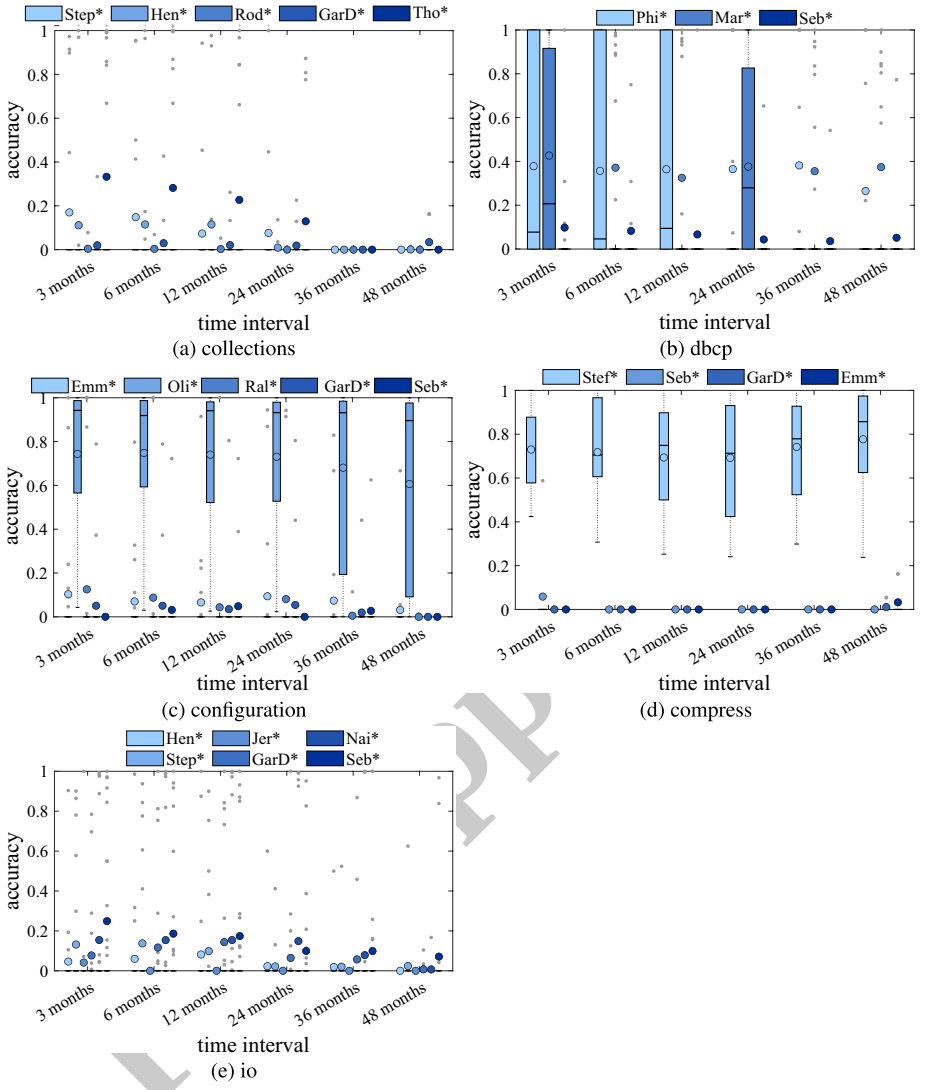

Fig. 6 The impact of time on individual authors

Finding 3. A model becomes less effective to identify individual long-term contributors when it is obsolete.

4.1.3 Answer to RQ

Our result shows that the median accuracy decreases significantly when temporal factors are taken into account. The results are consistent with those from other research topics. For instance, Turhan et al. (2009) report that temporal factors also significantly reduces the effectiveness of software defection prediction.

4.2 RQ2. The Impact of Project

4.2.1 Protocol

In this RQ, we use the stratified 10-fold cross-validation (Parsons 2014) as the baseline. Figure 7 shows our settings to explore the impact of projects. To train each model, we use the commits of a whole year from one project and the same parameters as the baseline. For example, in Fig. 7, the first commit and the last commit of the training set and testing set are submitted on Jan. 20, 2002, and Dec. 15, 2002, respectively. We test each trained model with the source files from another project as shown in Table 3. After all the models are trained and tested, we move the first commit of the training set and the first commit of the testing set to 12 months later and redo the training and testing. We move the first commit from 2002 to 2021 and redo the training and testing process 19 times. As shown in Table 3, some authors contribute to multiple projects. For instance, 30 code authors contribute to both `collections` and `dbcp`, 37 code authors contribute to both `collections` and `io`. If a pair of a training set and a testing set has no common code authors, we ignore this

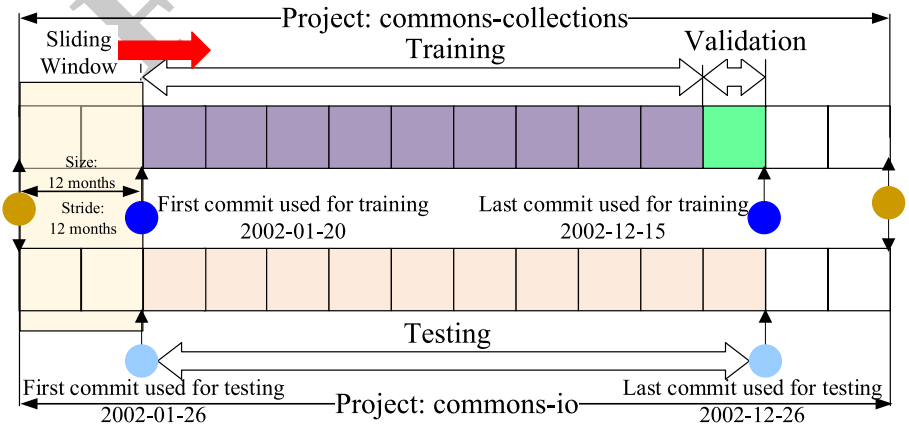

Fig. 7 The protocol of RQ2

Table 3 The common authors across projects

Common author / Project	collections	dbcp	configuration	compress	io
collections	89				
dbcp	30	62			
configuration	23	24	38		
compress	16	16	11	65	
io	37	30	24	23	94

pair. After that, in total, we obtain 16 valid pairs for `collections`, 13 pairs for `dbcp`, 8 pairs for `configuration`, 10 pairs for `compress`, and 18 pairs for `io`.

When a user needs to predict the authors of a source file, the user is unlikely to know whether this source file is written by known authors or not in advance. To analyze the true effectiveness, we do not remove unseen authors from the testing set. As the existing approaches cannot predict unseen authors, their effectiveness can be significantly reduced in the wild.

4.2.2 Result

Figure 8 shows the box plots of the two settings. The horizontal axes list the first-commit years of the training sets and the testing sets. In each pair, the cross-validation setting shows the accuracy values whose training and testing sets are from the same projects, and the cross-project setting shows the accuracy values whose training and testing sets are from different projects. In 95.38% (62 out of 65) pairs, the accuracy medians of the cross-validation setting are higher than those of the cross-project setting. The average reductions are 64.80% (`collections`), 59.56% (`dbcp`), 67.63% (`configuration`), 52.90% (`compress`), and 64.57% (`io`). For all the projects, the average reduction is 61.89%. The above observations lead to the following finding:

> **Finding 4.** When the training set and the testing set come from different projects, the accuracy median is reduced by 61.89%.

As shown in Fig. 8, only in several cases, the cross-project setting produces better accuracy values. For example, in 2010 of `dbcp`, the accuracy median increased by 3.05%. In total, such cases account for only 5.53% of all cases. After inspecting such cases, we find that the abnormal results are caused by the distributions of authors. For example, for the data points on 2021 of `compress`, when the model was trained from the `compress` project, the source files in the training set are written by Gar* (130 lines of code), Pet* (3 lines of code) and Stef* (70 lines of code). The source files in the testing set are written by Bor* (51 lines of code), Flo* (2 lines of code), Art* (319 lines of code), and Gar* (1776 lines of code). The trained model predicts the authors of most source files as Gar* and obtains high accuracy values. The above observations lead to another finding:

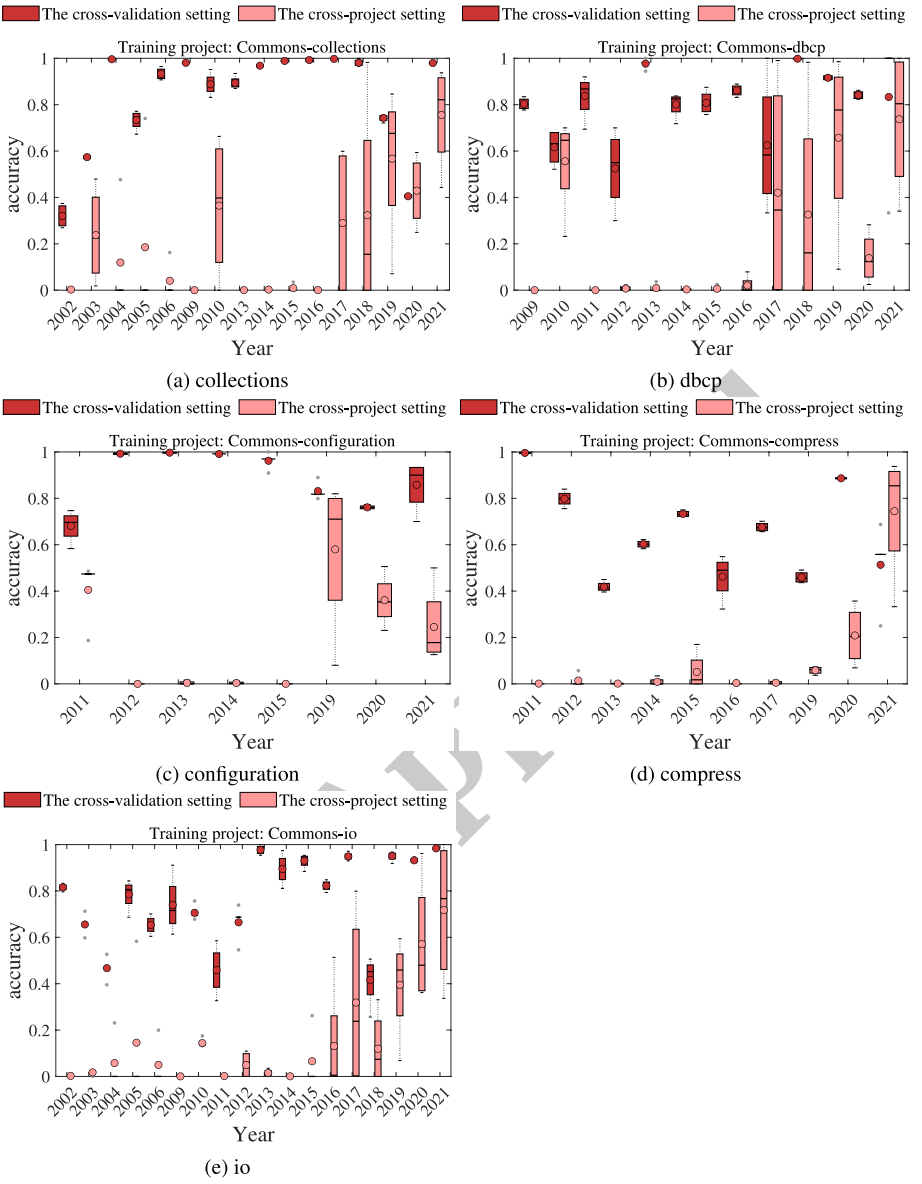

Fig. 8 The overall impact of projects

Finding 5. When training and testing sets come from different projects, in more than 94.47% of training and testing pairs, accuracy values become lower.

4.2.3 Answer to RQ

Our results show that the median accuracy values decrease significantly when the model is trained and tested on different projects.

4.3 RQ3. The Stability of Features

4.3.1 Protocol

To identify code authors, in total, Yang et al. (2017) defined 19 metrics. Table 4 shows the metrics. *PRO* denotes lexical metrics; *STY* denotes code-style metrics; and *PSM* denotes metrics from code structures and syntaxes. From each code snippet, we build an abstract syntax and analyze the tree to collect the metrics. All the metrics are straightforward except *PSM3* and *PSM11*. *PSM3* denotes the preference for cyclic statements. We check three types of cyclic statements such as `while`, `for`, and `do` loops. According to the most frequent cyclic statement, we set the value of *PSM3* to 1, 2, and 3, respectively. To extract *PSM11*, we count the depth as the longest distance from all nodes to the root.

In this RQ, we use statistical hypothesis testing inspired by the approach proposed by Hayes and Offutt (2010). 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.

Table 4 The code metrics in Yang et al. (2017)

Metrics	Description
PRO1	Ratio of blank lines to code lines
PRO2	Ratio of comment lines to code lines
PRO3	Ratio of block comments to all comment lines
PRO4	Ratio of open braces ({) alone in a line
PRO5	Ratio of close braces (}) alone in a line
STY1	Ratio of variables without uppercase letters
STY2	Ratio of variables starting with lowercase letters
STY3	Average variable name length
PSM1	Ratio of macro variables
PSM2	Ratio of `for` statements to all loop statements
PSM3	Preference for cyclic statements
PSM4	Ratio of `if` statements to all conditional statements
PSM5	Ratio of branch statements
PSM6	Average number of methods per class
PSM7	Ratio of `try` structure
PSM8	Ratio of `catch` statements
PSM9	Average number of interfaces per class
PSM10	Average character number per Java file
PSM11	Maximum depth of an AST

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.

When analyzing the stability of code metrics over time, for each project, we divide the source files by year, since the prior study (Abuhamad et al. 2018) and our previous RQs all use this criterion to divide source files. Here, we ignore years in which only a programmer wrote source files. With only one programmer, we cannot test whether their differences are significant. For each code metric m_i, we group the code snippets of each year by programmer, and construct a dataset d_i. For each year, we build different groups of datasets (one group for per programmer). After building the datasets, we use Shapiro-Wilk test (Mudholkar et al. 1995) to check the normality of all groups, and conduct Levene's test (Schultz 1985) to check the homogeneity across groups. When the normality and homoscedasticity assumptions both hold, we apply ANOVA (St et al. 1989) test to determine whether code metrics are useful for identifying code authors. If the assumptions do not hold, we use Kruskal-Wallis test (Vargha and Delaney 1998) to determine whether code metrics are useful. For example, for a code metric m_i, if the datasets of every two code authors a_i and a_j are D_i and D_j, the null hypothesis is defined as follows:

$$H_0 : \mu(D_i) = \mu(D_j) \tag{1}$$

The null hypothesis H_0 denotes that the differences between the two datasets are insignificant. It is rejected if the p-value is 0.05 or below 0.05. That is to say, if the null hypothesis for a code metric m_i is rejected, this code metric m_i is useful to distinguish two code authors. For each year, we apply the ANOVA test or Kruskal-Wallis test on all the combinations of programmer pairs, and in total there are C_n^2 pairs (n is the number of programmers). Thus, for each code metric, if all the C_n^2 p-values are equal to or less than 0.05, we consider that this code metric is useful to identify all code authors in that year. For each code metric (m_i) and the datasets of a project (D), we apply an ANOVA test or a Kruskal-Wallis test for each year:

$$ratio_{m_i} = \frac{\sum |\mu(D_i) \neq \mu(D_j)|}{C_n^2} \tag{2}$$

In the above equation, $\sum |\mu(D_i) \neq \mu(D_j)|$ denotes the total cases when $\mu(D_i)$ is significantly different from $\mu(D_j)$, and C_n^2 denotes the total number of programmer pairs. By the above equation, a code metric is able to distinguish all code authors only when the above ratio is one.

We count the stability value over time as follows:

$$stability_t = \frac{\sum |ratio_{m_i} = 1|}{all} \tag{3}$$

In the above equation, m_i is a metric, and $\sum |ratio_{m_i} = 1|$ denotes the number of years when the ratio is one, and *all* denotes the total of years. The total years are 16 (`collections`), 11 (`compress`), 12 (`configuration`), 17 (`dbcp`), and 18 (`io`).

When analyzing the stability of code metrics across projects, we select only programmers who write code on all five projects, i.e., Emm*, GarD*, Gar*, and Seb*. If a programmer does not write source files for a project, we cannot conclude whether the code metrics of this programmer are stable on this project. For a code metric m_i, we extract the code snippets of each programmer from a project, and the code snippets construct a dataset D_i. For each project, we build four datasets (one group for each programmer), and use (2) to calculate the ratio. As there are 4 programmers across projects, the total number of programmer pairs is 6.

For each metric (m_i), we count the stability value across projects as follows:

$$stability_p = \frac{\sum |ratio_{m_i} = 1|}{5} \tag{4}$$

In the above equation, $\sum |ratio_{m_i} = 1|$ denotes the number of projects when the ratio is one, and 5 denotes the total number of projects. When performing ANOVA or Kruskal-Wallis test, the sample size must not be too small to ensure the reliability of results (Kastenbaum et al. 1970). As a result, we ignore a year if this year has fewer than 10 commits.

4.3.2 Result

Figure 9 shows the stability values over time. The horizontal axis lists the projects. The vertical axis lists the 19 code metrics. A darker color denotes a lower stability value. We find that only in several cases, the stability values are greater than or equal to 0.5 (e.g., *PRO1* and *PSM10* of `collections`). The stability values of `io` are much lower than those of `configuration`. We find that the differences are caused by the number of code authors. In most years, `io` has four or more code authors, but `configuration` has only two code authors. As a result, `io` has more combinations of programmer pairs, and is thus more challenging for code metrics to identify all code authors. Only in 2013 of `collections`, there are four code authors such as Emm*, GarD*, Seb*, and Tho*. For *PSM10*, the mean values of the four code authors are 146.7, 721.7, 267.8, and 2747.2, respectively. As a result, *PSM10* is able to distinguish them. Here, PRO1 fails to identify the four code authors in 2013, but it identifies code authors in earlier years when `collections` has only two code authors.

As `configuration` has only two code authors in most years, the two code authors can have a clear division of responsibility. As a result, only for this project, the stability values of several code metrics are greater than or equal to 0.5. The observations lead to our finding:

> **Finding 6.** Except for `configuration`, only when a project has fewer than three code authors, the stability values over time are more than or equal to 0.5 (e.g., *the ratio of blank lines to code lines*).

Figure 10 shows the ratios across projects. The horizontal axis lists the projects. The vertical axis lists the code metrics. A darker color denotes a lower ratio. We find that no code metric can identify all code authors for all the projects. Based on Fig. 10, we use (4)

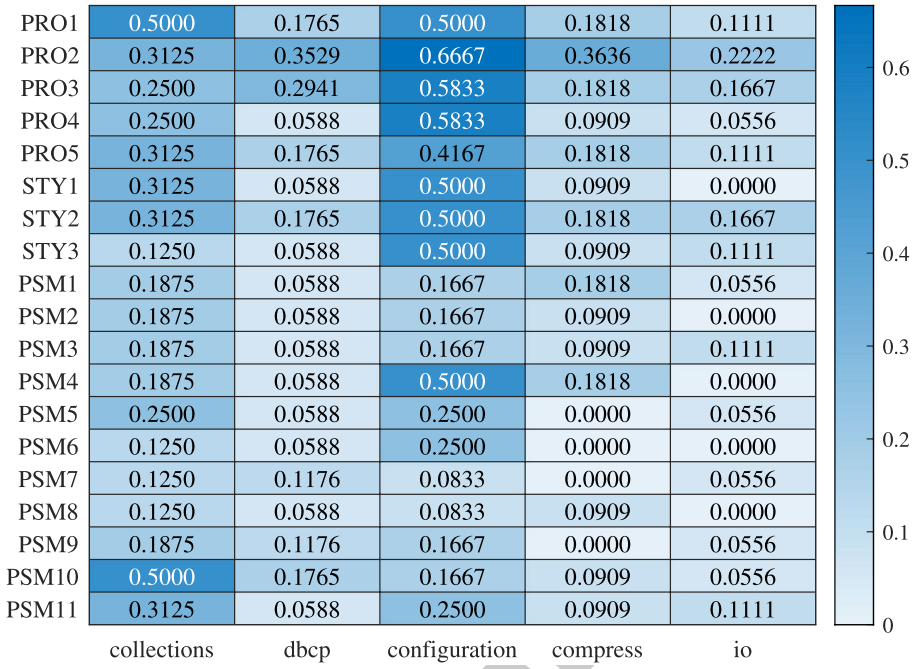

Fig. 9 The stability values over time

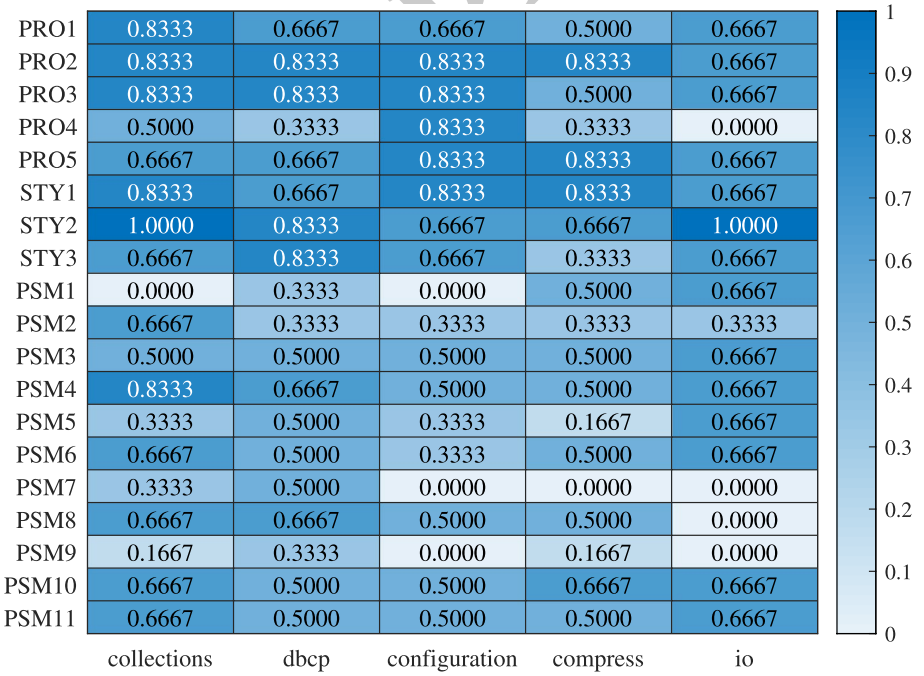

Fig. 10 The ratios across projects

to calculate the stability values of all metrics, but only the value of *STY2* is more than zero. When building Fig. 10, we consider the four code authors that appear in all the projects, but the code authors of `io` change rapidly with time. In total, 14 code authors contribute to `io`. As a result, Figs. 9 and 10 show different results. The observations lead to a finding:

> **Finding 7.** Except for a metric (*the ratio of variable naming starting with lowercase letters*), the stability values of other metrics across projects are 0.

4.3.3 Answer to RQ

Except `STY2`, the stability values of code metrics are below 0.5. These results highlight the challenges of identifying code authors in real development.

4.4 RQ4. The Effectiveness of Advanced Learning

4.4.1 Protocol

In this research question, we select two recent approaches, TimeDA (Li et al. 2024) and CICaL (Gong and Zhong 2025), since they use more advanced techniques than prior approaches. In particular, TimeDA (Li et al. 2024) employs transfer learning (Bozinovski and Fulgosi 1976) to refresh outdated models, and CICaL (Gong and Zhong 2025) incorporates incremental learning. CICaL treats the accumulated code snippets as data streams, and each incoming batch of code snippets is treated as a data chunk. CICaL utilizes an ensemble framework and incrementally trains its model with incoming chunks of data.

To evaluate the time impact on TimeDA (Li et al. 2024), we use the commits of a whole year to train a classifier. For instance, in Fig. 3, the first and last commit of the training set are submitted on Jan. 16, 2002, and Dec. 31, 2002, respectively. The parameters of this classifier are identical to those in RQ1. As we did in RQ1, each testing set comprises commits from an entire year. The time intervals between the training and testing sets range from 3 to 48 months, with a step of 3 months. As depicted in Fig. 3, to generate a sufficient number of training-testing pairs, we employ a sliding time window. Specifically, after each iteration, the training window is shifted forward by six months. To evaluate the impact of cross-project authors, we use the commits of a whole year from one project to train a classifier. For instance, in Fig. 7, the first commit and the last commit of the training set and testing set are submitted on Jan. 20, 2002, and Dec. 15, 2002, respectively. As we did in RQ2, each testing set consists of one year's worth of commits from a different project. As depicted in Fig. 7, after each iteration, both the training and testing windows are shifted forward by 12 months.

To evaluate the impact of time on CICaL, we use the commits of a whole year to train each base classifier. For instance, in Fig. 3, the first and last commit of the training set are submitted on Jan. 16, 2002, and Dec. 31, 2002, respectively. The parameters of each base classifier are identical to those in RQ1. As we did in RQ1, each testing set comprises commits from an entire year. The time intervals between the training and testing sets range from 3 to 48 months, with a step of 3 months. As depicted in Fig. 3, to generate a suffi-

cient number of training-testing pairs, we employ a sliding time window. Specifically, after each iteration, the entire training window is shifted forward by six months. To evaluate the impact of cross-project authors, we use the commits of a whole year from one project to train each base classifier. For instance, in Fig. 7, the first commit and the last commit of the training set and testing set are submitted on Jan. 20, 2002, and Dec. 15, 2002, respectively. As we did in RQ2, each testing set consists of one year's commits from a different project. As depicted in Fig. 7, after each iteration, both the training and testing windows are shifted forward by 12 months.

We compare the results of RQ1 to demonstrate the impact.

4.4.2 Result

Figure 11 shows the impact of time. The horizontal axes list when the first commits of training sets are submitted. TimeDA improves the average accuracy of Multi-χ by 5.58% (`collections`), 14.37% (`dbcp`), 7.92% (`configuration`), 12.98% (`compress`), and 10.76% (`io`). For all the projects, the average improvement is 10.32%. In 89 out of 93 cases, the medians of TimeDA are higher than those of Multi-χ. As shown in Fig. 11, in 4 cases, Multi-χ produces the same accuracy values as TimeDA. In these 4 cases, all the testing accuracy of Multi-χ and TimeDA is 0. After inspecting these cases, we find that the unseen authors lead to these extreme results. For example, for the data points on Aug. 16, 2005 of `io`, the training set contains the authors Jam*, Hen* and Step*. The testing sets contain the authors GarD*, Nai*, Joc*, Juk*, Seb*, and Mat*. The authors in the testing sets are unseen in the training set. In all cases, the medians of CICAL are higher than those of Multi-χ. As a comparison, CICAL improves the average accuracy of Multi-χ by 31.39% (`collections`), 31.22% (`dbcp`), 24.10% (`configuration`), 37.00% (`compress`), and 22.93% (`io`). For all the projects, the average improvement is 29.33%.

On the Google Code Jam dataset, our previous evaluation (Gong and Zhong 2025) shows that CICAL and TimeDA also achieves better accuracy scores than Multi-χ, and the scores are much higher. CICAL achieved an average accuracy of 90.17% on the Google Code Jam dataset, but only 61.50% in this study. The average accuracy of TimeDA also drops from 73.43% to 42.50%. The above observations lead to our finding:

> **Finding 8.** When the impact of time is considered, CICAL and TimeDA improve the average accuracy of Multi-χ by 29.33% and 10.32%, respectively.

Figure 12 shows the impact of cross-project authors. The horizontal axes list when the first commits of training sets are submitted. In 89 out of 93 cases, the medians of TimeDA are higher than those of Multi-χ. TimeDA improves the average accuracy of Multi-χ by 5.02% (`collections`), 4.86% (`dbcp`), 8.12% (`configuration`), 6.17% (`compress`), and 5.82% (`io`), with an overall average improvement of 6.00%. As a comparison, CICAL improves the average accuracy of Multi-χ by 21.92% (`collections`), 13.74% (`dbcp`), 22.26% (`configuration`), 19.93% (`compress`), and 15.37% (`io`), with an overall average improvement of 18.44%. The observation leads to a finding:

> **Finding 9.** Under the cross-project setting, CICAL and TimeDA improve the average accuracy of Multi-χ by 18.44% and 6.00%, respectively.

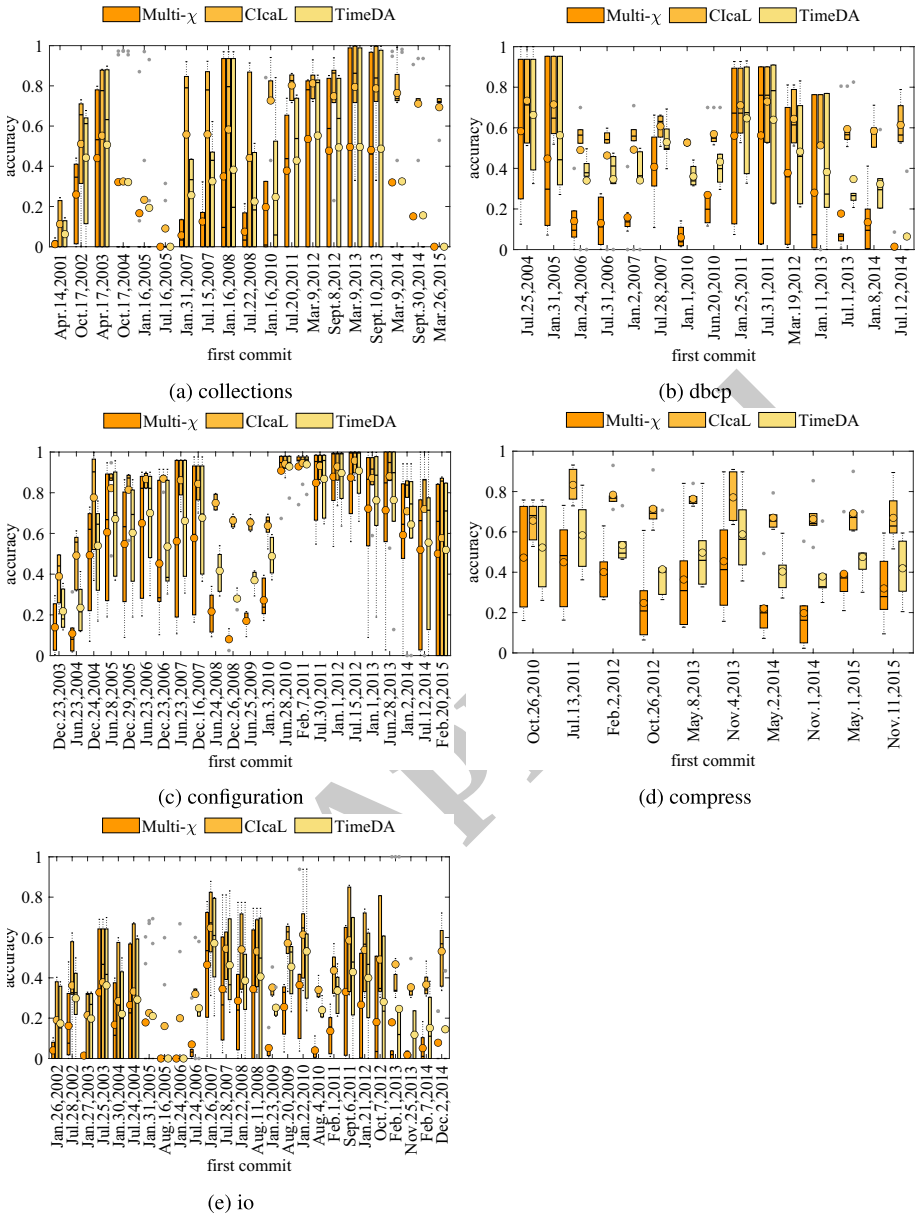

Fig. 11 The impact of time on techniques

4.4.3 Answer to RQ

Our results and our previous evaluation (Gong and Zhong 2025) both show that advanced machine learning techniques, e.g., transfer learning and incremental learning, alleviate the

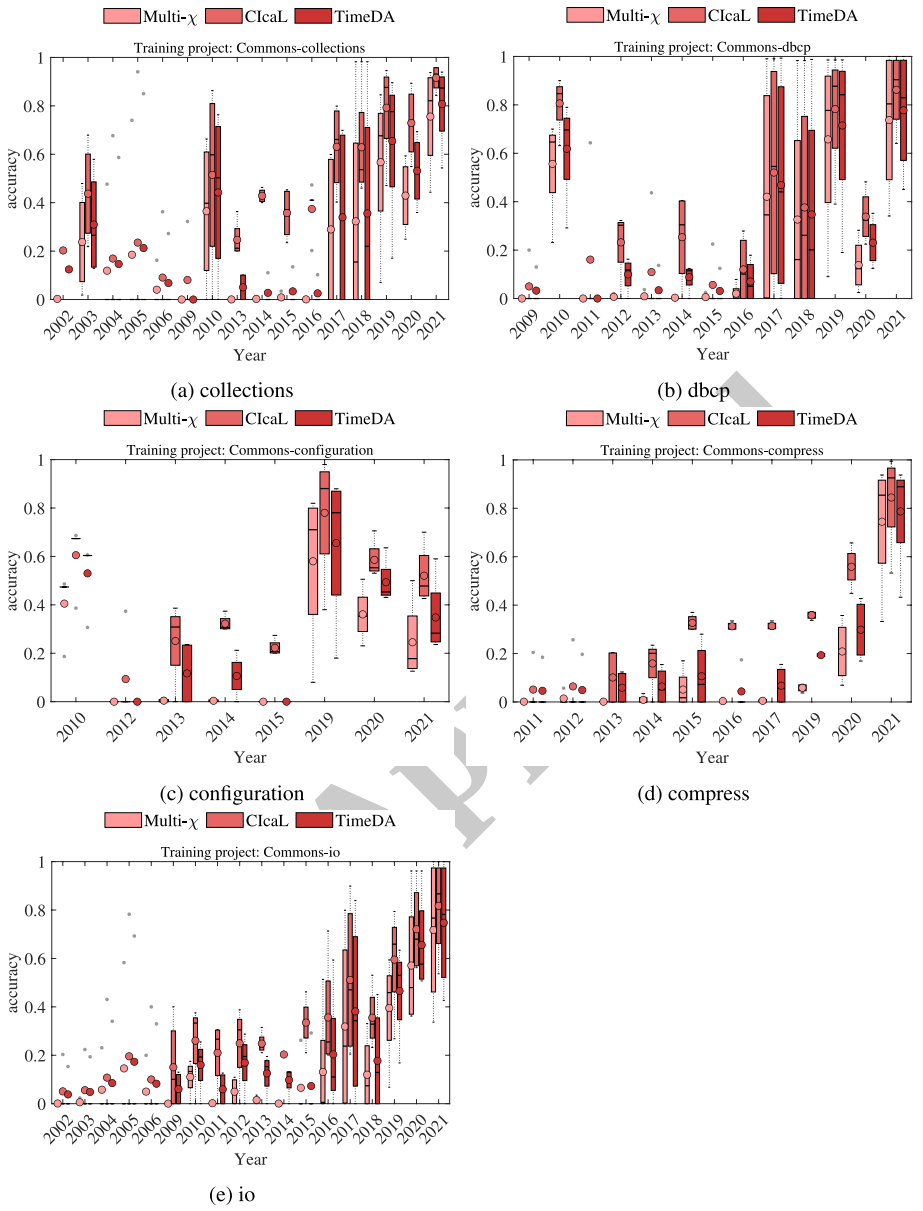

Fig. 12 The impact of cross-project authors on techniques

impact of time and cross-project authors. In particular, incremental learning improves accuracy by up to 29.33% under time shifts and 18.44% under project shifts, and transfer learning improves by 10.32% and 6.00%, respectively. On the Google Code Jam dataset, CIcaL achieves more an average accuracy of than 90%, but on our new benchmark, it drops to 61.50%. The result indicates that neither tool is perfect for practical application, but it leaves sufficient space for follow-up researchers.

4.5 Threat to Validity

The threats to internal validity include wrong labels, since commits may not be submitted by their true authors. This threat is shared by the prior studies (Abuhamad et al. 2020; Dauber et al. 2018), and its impact shall be minor since Apache projects are carefully maintained. This threat could be further reduced by data sanitization techniques (Bird et al. 2009). The threats to internal validity also include our implementation. The selected approach (Abuhamad et al. 2020) is built upon TensorFlow (Abadi et al. 2016), but we build the tool upon Keras. Despite the differences, the impact shall be minor, since our effectiveness is similar to theirs.

The threats to external validity include limited subjects and project sources. Our dataset are obtained from five Apache Software Foundation projects, and it covers a 20-year period. Still, selecting only Apache projects may not capture the diversity of coding practices, styles, and author behaviors in the broader software development communities. Although our subjects are already much more than the prior study (Abuhamad et al. 2018), our study still suffers from this threat. If follow-up researchers add subjects from more sources (Nagappan et al. 2013), our findings in RQs 1 and 2 may not change since we have compared many pairs of training and testing sets. For instance, we tried more than 90 pairs in RQ1, and more than 60 pairs in RQ2. The code standards of the Apache Foundation can affect the code lexical metrics in RQ3, e.g., the ratio of open braces.

The threats to external validity also include our considered programming language, since our subjects are all written in Java. Still, the impact shall be minor. First, Abuhamad et al. (2018) claim that their approach is language-independent, and their results from different languages are similar. Second, the code styles of authors can change over time. Finally, a few authors contribute to most code lines. The two observations are also language-independent. As a result, our findings may not change much on other languages.

In our dataset, only 4 programmers contribute to more than one project. As the observation is limited, Finding 7 suffers from a threat to the external validity. This threat could be reduced if more projects are analyzed.

5 Interpretation of Our Findings

In this section, we interpret our findings:

1. **The state-of-the-art approach is insufficient to handle real source files over time and across projects.** Findings 1, 2, and 3 show that the state-of-the-art approach is less effective when its trained model becomes obsolete. Findings 4 and 5 show that the accuracy of this approach significantly decreases when the cross-project is considered. The results are quite different from what were reported in their study (Abuhamad et al. 2018), since their subjects are from Google Code Jam. In real development, programmers can learn from existing code bases and keep their code style consistent with existing source files. In addition, some project managers even ask programmers to use linters (e.g., Checkstyle (Checkstyle 2001)), and check style inconsistencies before commits are accepted. Although our selected projects do not have such linters, these project teams suggest that code authors should follow specific coding styles like most other

projects. As a result, lexical metrics i.e., PROs listed in Table 4 can be more correlated with projects than specific programmers. Due to the above issues, it is much more challenging to identify code authors for source files from real projects than those from a code competition. Meanwhile, in many applications of information forensics and security, it needs to identify code authors for real source files, instead of toy programs from code competitions like Google Code Jam. Instead of a solved problem, it demands more research effort to correctly identify real code authors.

2. **More reliable measures are required in future evaluations.** The contributions of code authors are extremely imbalanced, and some projects have many code authors as shown in Findings 2 and 5. For large projects (e.g., Linux), a few core programmers write most source files (Gong and Zhong 2021), but these projects often have many code authors. Besides our tool, researchers can refer to alternative mature tools to build the benchmarks of large projects. For instance, the Git blame tool (Spinellis 2012) can report the code author of each code lien, and it can scale to large projects like Linux. As another example, researchers can merge identical authors by comparing their emails. To make the problem more challenging, our previous study (Gong and Zhong 2021) shows that multiple code authors often contribute to a source file. For example, as shown in Fig. 6, when we calculate the accuracy values for individual authors, for all the projects, we find that the trained models work well on only one or two authors, but their accuracy values are high. The results indicate that accuracy values are unreliable. Other measures (e.g., MCC (Chicco and Jurman 2020; Zhu 2020)) and some techniques (e.g., resampling and classifier adaptations (Tarekegn et al. 2021)) can be more reliable to evaluate the results of identifying code authors.
3. **An incremental learning approach can be useful.** No matter what techniques researchers employ, they shall extract code metrics from source files. Even if it is a learning-based approach, it shall implicitly learn them from its inputs, i.e., source files. Findings 6 and 7 show that there is no such fully reliable code metric. According to our results, a trained model rapidly becomes out of date and ineffective when projects evolve, and cross-project learning is rather challenging, even if two projects share the same code authors. A possible solution is to train models incrementally so that trained models can evolve with the rapid evolution of projects. Researchers from both cybersecurity domain and software engineering field proposed using incremental learning to address the concept drift issue in malware detection (Chen et al. 2023; Narayanan et al. 2016; Qiang et al. 2022). Findings 8 and 9 show that the incremental learning framework can be used to mitigated with source files written by more cross-project authors.

6 Related Work

Our work is related to the following research topics:

Code Authorship Identification Researchers have proposed many approaches to identify code authors. Caliskan et al. (Caliskan-Islam et al. 2015) 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. Yang et al. (2017) use the ratio of blank lines to

code lines and the ratio of comment lines to code lines to identify code authors. In natural language processing, ngram models are widely used to identify the author of a text (Clement and Sharp 2003). As Hindle et al. (2012) show that programming languages and natural languages share similarities, some approaches (e.g., Caliskan et al. 2018) use ngram models to capture the habits of programmers. For example, Caliskan et al. (2018) extract a bigram, which indicates that `func` is followed by `if`. Some approaches (e.g., Alsulami et al. 2017) bypass the extraction of metrics or ngrams, and take abstract syntax trees or control flow graphs directly as their inputs. These approaches typically rely on existing techniques to implicitly extract metrics or ngrams. For example, Alsulami et al. (2017) use a recurrent neural network (Gers et al. 2000) to analyze ASTs, and such a technique extracts code features as ngrams. Hansen et al. (2014) examine the practical feasibility of using limited and recent writing samples from students for authorship attribution. Although these selected features can be contributive to authorship identification, they ignore the time and project issues. The findings of our empirical study can be useful to extract more stable software features considering over time and across projects.

Class Imbalance Problem of Code Authorship Identification To evaluate the proposed code authorship identification models, some prior studies (Abuhamad et al. 2018, 2020) use custom-built source code collections that include balanced training sets. Note that a code author can have much more or fewer files in real scenarios (Gong and Zhong 2021). Thus, long samples (multiple lines of code) may be available for some code authors and short samples for other code authors. This can also be viewed as a case of class imbalance. Some studies (Chatzicharalampous et al. 2012; Mahbub et al. 2019; Yang et al. 2017) do not avoid the class imbalance problem and use the real distribution of data to train the code authorship identification methods. Our study takes the data imbalance problem into consideration and analyzes its effect on the performance of code authorship identification methods.

Mining Code Revision Histories From code histories, Abuhamad et al. (2020) use git-author (Meng et al. 2013) to extract code authors as their labels. Rahman and Devanbu (2011) use code histories to analyze the relationship between code ownership and developer experience. From code histories, Jiang et al. (2022) explore whether the naturalness of code is associated with the buggy or clean code. To provide a more precise analysis of code histories, Zhong and Wang (2017) propose an approach to repair the contexts of partial programs. Based on this approach, Zhong et al. (2020) infer bug signatures and use them to detect bugs. For code history mining techniques, Servant and Jones (2012) use history slices to analyze code evolution tasks. Gong and Zhong (2021) conduct an empirical study to analyze code authors in the wild. Our study is related to code authors, but it explores the limitations of existing approaches and provides insights for future improvements.

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. (2023) 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. (2023) propose a discriminative feature set in differentiating ChatGPT-generated code from human-authored code in binary classification tasks. Idialu et al. (2024) construct a classifier for detecting 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.

7 Conclusion and Future Work

Identifying code authors is important in many research topics, and various approaches have been proposed to attack this task, but this task has many challenges. For example, the code style of an author can evolve over time and across projects. Besides the impacts of time and project, the distribution of files can also affect their effectiveness. To better understand these challenges, we conduct an empirical study and explore open questions concerning the impacts of time and projects for identifying code authors. In addition, we also conduct investigations on learning code metrics that are stable over time and across projects.

In the future, we plan to explore three directions. First, we plan to explore more advanced and interpretable models since our dataset reveals the limitations of existing approaches. If we compare the different results between their original dataset and our dataset, we can find ways to improve their approaches. Second, as some programmers intensively use LLMs when they write code, researchers (Bukhari et al. 2023; Li et al. 2023) have proposed some approaches to identify human-authored and LLM-generated code. In their dataset, each source file is written by only humans or LLMs. In real development, LLM-generated programs can be modified by humans. If a dataset contains files modified by both humans and LLMs, it can be more challenging to identify their authors, which can be explored in future work.

Funding This work was supported by National Key R&D Program of China under Grant No. 2023YFB4503804 and the National Nature Science Foundation of China No. 62232003 and 62272295. Hao Zhong is the corresponding author.

Data Availability The dataset for this paper are available on Zenodo (DOI: https://doi.org/10.5281/zenodo.8058290) and Github (https://github.com/noonekowns/codeauthor).

Declarations

Conflicts of Interest The authors declare conflict of interest with the people affiliated with Shanghai Jiao Tong University.

References

Abadi M, Barham P, Chen J, Chen Z, Davis A, Dean J, Devin M, Ghemawat S, Irving G, Isard M et al (2016) Tensorflow: a system for large-scale machine learning. In: Proc. Osdi, vol 16. Savannah, GA, USA, pp 265–283

Abuhamad M, AbuHmed T, Mohaisen A, Nyang D (2018) Large-scale and language-oblivious code authorship identification. In: Proc. CCS. pp 101–114

Abuhamad M, Abuhmed T, Nyang D, Mohaisen D (2020) Multi-χ: identifying multiple authors from source code files. In: Proc. PETS. pp 25–41

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

Anvik J, Hiew L, Murphy GC (2006) Who should fix this bug? In: Proc. ICSE. pp 361–370

Bao L, Xia X, Lo D, Murphy GC (2019) A large scale study of long-time contributor prediction for Github projects. IEEE Trans Softw Eng, 1–22

Bird C, Rigby PC, Barr ET, Hamilton DJ, German DM, Devanbu P (2009) The promises and perils of mining git. In: Proc. MSR. pp 1–10

Bock T, Alznauer N, Joblin M, Apel S (2023) Automatic core-developer identification on Github: a validation study. ACM Trans Softw Eng Methodol 32(6):1–29

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

Bozinovski S, Fulgosi A (1976) The influence of pattern similarity and transfer learning upon training of a base perceptron B2. In: Proceedings of symposium informatica, vol 3. pp 121–126

Browne MW (2000) Cross-validation methods. J Math Psychol 44(1):108–132

Bukhari S, Tan B, De Carli L (2023) Distinguishing AI-and human-generated code: a case study. In: Proc. SCORED. pp 17–25

Burrows S, Uitdenbogerd AL, Turpin A (2009) Temporally robust software features for authorship attribution. In: Proc. COMPSAC, vol 1. IEEE, pp 599–606

Caliskan A, Yamaguchi F, Dauber E, Harang R, Rieck K, Greenstadt R, Narayanan A (2018) When coding style survives compilation: de-anonymizing programmers from executable binaries. In: Proc. NDSS

Caliskan-Islam A, Harang R, Liu A, Narayanan A, Voss C, Yamaguchi F, Greenstadt R (2015) De-anonymizing programmers via code stylometry. In: Proc. USENIX security. pp 255–270

Canedo ED, Bonifácio R, Okimoto MV, Serebrenik A, Pinto G, Monteiro E (2020) Work practices and perceptions from women core developers in OSS communities. In: Proceedings of the 14th ACM/IEEE international symposium on empirical software engineering and measurement. pp 1–11

Chatzicharalampous E, Frantzeskou G, Stamatatos E (2012) Author identification in imbalanced sets of source code samples. In: Proc. ICTAI, vol 1. pp 790–797

Checkstyle (2001). https://github.com/checkstyle/checkstyle

Chen Y, Ding Z, Wagner D (2023) Continuous learning for Android malware detection. In: Proc. USENIX security. pp 1127–1144

Chicco D, Jurman G (2020) The advantages of the Matthews correlation coefficient (MCC) over F1 score and accuracy in binary classification evaluation. BMC Genom 21(1):1–13

Clement R, Sharp D (2003) Ngram and Bayesian classification of documents for topic and authorship. Lit Linguist Comput 18(4):423–447

Dauber E, Caliskan A, Harang R, Greenstadt R (2018) Git blame who? stylistic authorship attribution of small, incomplete source code fragments. In: Proc. ICSE companion. pp 356–357

Ding H, Samadzadeh MH (2004) Extraction of Java program fingerprints for software authorship identification. J Syst Softw 72(1):49–57

Ferreira M, Valente MT, Ferreira K (2017) A comparison of three algorithms for computing truck factors. In: International conference on program comprehension. pp 207–217

Gers FA, Schmidhuber J, Cummins F (2000) Learning to forget: continual prediction with LSTM. Neural Comput 12(10):2451–2471

Gong S, Zhong H (2021) Code authors hidden in file revision histories: an empirical study. In: Proc. ICPC. pp 71–82

Gong S, Zhong H (2022) A study on identifying code author from real development. In: Proc. ESEC/FSE, p. to appear

Gong S, Zhong H (2025) Incremental learning of code authors over time. J Syst Softw

Google Code Jam (2019). https://codingcompetitions.withgoogle.com/codejam

Géron A (2019) Hands-on machine learning with Scikit-Learn, Keras, and tensorflow: concepts, tools, and techniques to build intelligent systems. O'Reilly Media

Hansen ND, Lioma C, Larsen B, Alstrup S (2014) Temporal context for authorship attribution. In: Proc. IRFC. pp 22–40
Hayes JH, Offutt J (2010) Recognizing authors: an examination of the consistent programmer hypothesis. Softw Testing Verification Reliab 20(4):329–356
Hindle A, Barr ET, Su Z, Gabel M, Devanbu P (2012) On the naturalness of software. In: Proc. ICSE. pp 837–847
Idialu OJ, Mathews NS, Maipradit R, Atlee JM, Nagappan M (2024) Whodunit: classifying code as human authored or GPT-4 generated-a case study on CodeChef problems. In: Proc. MSR. pp 394–406
Jiang Y, Liu H, Zhang Y, Ji W, Zhong H, Zhang L (2022) Do bugs lead to unnaturalness of source code? In: Proc. ESEC/FSE. pp 1085–1096
Kalgutkar V, Kaur R, Gonzalez H, Stakhanova N, Matyukhina A (2019) Code authorship attribution: methods and challenges. ACM Comput Surv 52(1):1–36
Kastenbaum MA, Hoel DG, Bowman K (1970) Sample size requirements: one-way analysis of variance. Biometrika 57(2):421–430
Krsul I, Spafford EH (1997) Authorship analysis: identifying the author of a program. Comput Secur 16(3):233–257
Lange RC, Mancoridis S (2007) Using code metric histograms and genetic algorithms to perform author identification for software forensics. In: Proc. GECCO. pp 2082–2089
Li K, Hong S, Fu C, Zhang Y, Liu M (2023) Discriminating human-authored from ChatGPT-generated code via discernable feature analysis. In: Proc. ISSREW. pp 120–127
Liu S, Lin G, Qu L, Zhang J, De Vel O, Montague P, Xiang Y (2020) CD-VulD: cross-domain vulnerability discovery based on deep domain adaptation. IEEE Trans Dependable Secure Comput 19(1):438–451
Li Z, Chen G, Chen C, Zou Y, Xu S (2022) RoPGen: towards robust code authorship attribution via automatic coding style transformation. In: Proc. ICSE. pp 1906–1918
Li Z, Zhao S, Chen C, Chen Q (2024) Reducing the impact of time evolution on source code authorship attribution via domain adaptation. ACM Trans Softw Eng Methodol
Mahbub P, Oishie NZ, Haque SR (2019) Authorship identification of source code segments written by multiple authors using stacking ensemble method. In: Proc. ICCIT. pp 1–6
Meng X, Miller BP, Williams WR, Bernat AR (2013) Mining software repositories for accurate authorship. In: Proc. ICSM. pp 250–259
Mudholkar GS, Srivastava DK, Thomas Lin C (1995) Some p-variate adaptations of the Shapiro-Wilk test of normality. Commun Stat-Theory Methods 24(4):953–985
Nagappan M, Zimmermann T, Bird C (2013) Diversity in software engineering research. In: Proc. ESEC/FSE. pp 466–476
Narayanan A, Yang L, Chen L, Jinliang L (2016) Adaptive and scalable Android malware detection through online learning. In: Proc. IJCNN, pp. 2484–2491
Nguyen V, Le T, de Vel O, Montague P, Grundy J, Phung D (2020) Dual-component deep domain adaptation: a new approach for cross project software vulnerability detection. In: Proc. PAKDD. pp 699–711
Nguyen V, Le T, Le T, Nguyen K, DeVel O, Montague P, Qu L, Phung D (2019) Deep domain adaptation for vulnerable code function identification. In: Proc. IJCNN. pp 1–8
Parker A, Hamblen JO (1989) Computer algorithms for plagiarism detection. IEEE Trans Educ 32(2):94–99
Parsons VL (2014) Stratified sampling. Wiley StatsRef: Statistics Reference Online, pp 1–11
Petrik J, Chuda D (2021) The effect of time drift in source code authorship attribution: time drifting in source code-stylochronometry. In: Proc. CompSysTech. pp 87–92
Piryonesi SM, El-Diraby TE (2020) Data analytics in asset management: cost-effective prediction of the pavement condition index. J Infrastruct Syst 26(1):04019036
Qiang Q, Cheng M, Hu Y, Zhou Y, Sun J, Ding Y, Qi Z, Jiao F (2022) An incremental malware classification approach based on few-shot learning. In: Proc. ICC. pp 2682–2687
Rahman F, Devanbu P (2011) Ownership, experience and defects: a fine-grained study of authorship. In: Proc. ICSE. pp 491–500
Schultz BB (1985) Levene's test for relative variation. Syst Zool 34(4):449–456
Schuster M, Paliwal KK (1997) Bidirectional recurrent neural networks. IEEE Trans Signal Process 45(11):2673–2681
Servant F, Jones JA (2012) History slicing: assisting code-evolution tasks. In: Proc. ESEC/FSE. pp 1–11
Shevertalov M, Kothari J, Stehle E, Mancoridis S (2009) On the use of discretized source code metrics for author identification. In: Proc. SSBSE. pp 69–78
Spafford EH, Weeber SA (1993) Software forensics: can we track code to its authors? Comput Secur 12(6):585–595
Spinellis D (2012) Git. IEEE Softw 29(3):100–101
St L, Wold S et al (1989) Analysis of variance (ANOVA). Chemom Intell Lab Syst 6(4):259–272

Tarekegn AN, Giacobini M, Michalak K (2021) A review of methods for imbalanced multi-label classification. Pattern Recogn 118:107965
The Apache foundation (2018). http://www.apache.org/
Turhan B, Menzies T, Bener AB, Di Stefano J (2009) On the relative value of cross-company and within-company data for defect prediction. Empir Softw Eng 14:540–578
Vargha A, Delaney HD (1998) The Kruskal-Wallis test and stochastic homogeneity. J Educ Behav Stat 23(2):170–192
Willis BH, Riley RD (2017) Measuring the statistical validity of summary meta-analysis and meta-regression results for use in clinical practice. Stat Med 36(21):3283–3301
Yang X, Xu G, Li Q, Guo Y, Zhang M (2017) Authorship attribution of source code by using back propagation neural network based on particle swarm optimization. PLoS ONE 12(11):e0187204
Yao J, Shepperd M (2020) Assessing software defection prediction performance: why using the Matthews correlation coefficient matters. In: Proc. EASE. pp 120–129
Zhong H, Wang X (2017) Boosting complete-code tool for partial program. In: Proc. ASE. pp 671–681
Zhong H, Wang X, Mei H (2020) Inferring bug signatures to detect real bugs. IEEE Trans Software Eng 48(2):571–584
Zhu Q (2020) On the performance of Matthews correlation coefficient (MCC) for imbalanced dataset. Pattern Recogn Lett 136:71–80