future internet

Article

Complex Cases of Source Code Authorship Identification Using a Hybrid Deep Neural Network

Anna Kurtukova *, Aleksandr Romanov , Alexander Shelupanov and Anastasia Fedotova

Department of Security, Tomsk State University of Control Systems and Radioelectronics, 634050 Tomsk, Russia
* Correspondence: av.kurtukova@gmail.com

Abstract: This paper is a continuation of our previous work on solving source code authorship identification problems. The analysis of heterogeneous source code is a relevant issue for copyright protection in commercial software development. This is related to the specificity of development processes and the usage of collaborative development tools (version control systems). As a result, there are source codes written according to different programming standards by a team of programmers with different skill levels. Another application field is information security—in particular, identifying the author of computer viruses. We apply our technique based on a hybrid of Inception-v1 and Bidirectional Gated Recurrent Units architectures on heterogeneous source codes and consider the most common commercial development complex cases that negatively affect the authorship identification process. The paper is devoted to the possibilities and limitations of the author's technique in various complex cases. For situations where a programmer was proficient in two programming languages, the average accuracy was 87%; for proficiency in three or more—76%. For the artificially generated source code case, the average accuracy was 81.5%. Finally, the average accuracy for source codes generated from commits was 84%. The comparison with state-of-the-art approaches showed that the proposed method has no full-functionality analogs covering actual practical cases.

Keywords: authorship; source code; commits; generation; neural network; deep neural network

check for updates

Citation: Kurtukova, A.; Romanov, A.; Shelupanov, A.; Fedotova, A. Complex Cases of Source Code Authorship Identification Using a Hybrid Deep Neural Network. *Future Internet* **2022**, *14*, 287. https://doi.org/10.3390/fi14100287

Academic Editor: Filipe Portela

Received: 16 September 2022
Accepted: 28 September 2022
Published: 30 September 2022

1. Introduction

The problem of identifying the author of a source code in case of considered code as artificial language text is relevant. Solutions to this problem are based on defining an individual author's code style (specific development methods). Such solutions can be especially useful in information security, educational and copyrighting fields. Researchers are interested in improving existing approaches to source code authorship identification as well as developing new ones based on them. Any of these approaches are suitable and highly accurate only for simple authorship attribution cases. Simple case refers to pure source codes without any manual or automatic transformations of the integrated development environment (linters, code formatting based on programming language paradigms) or external tools (obfuscators), as well as cases where data are exactly authentic and homogeneous. The solution to the applied problems of identifying the source code author involves the analysis of program texts, which are complicated by certain factors. These factors could be divided into two groups:

1. Factors that arise during or as a result of writing source code. This group includes various transformations of source codes. For example, an earlier mentioned case: obfuscation of the source code—modification to an unclear and misunderstood form that makes it difficult to analyze but retains functionality; writing source code following coding standards—and the development of the source code, taking into account the conventions and general rules adopted by a group of programmers.
2. Factors that result from the specificity of the development process. This group includes other cases that complicate the process of determining the author of the source code:

for example, identification of the author whose code samples are written in different programming languages (on the basis of mixed data), as well as finding a distinction of the source code authorship between a human and generative model. Another complex case is the determination of authorship based on source code samples written as part of group development.

The scientific novelty of the work lies in the technique proposed by the authors, which for the first time takes into account both simple and all complex cases of identifying the source code author. Difficult cases include identifying the author:

1. Source code formed on separate code fragments (commits);
2. The artificially generated source code of the program;
3. Source code, the author of which is writing in two or more programming languages;
4. Obfuscated source codes;
5. The source code is written according to coding standards.

The rest of the paper is divided into seven sections. Section 2 is devoted to our previous works aimed at solving the problem of source code authorship attribution, as well as the achieved results. Section 3 contains the analysis of modern related studies in the subject area and describes the methods proposed by other researchers, and the limitations and drawbacks of the existing techniques. A formal statement is presented in Section 4. The author's technique for source code authorship identification is described in Section 5. Section 6 includes the description of the first investigations of the complex cases of source code attribution, as well as the information about the obtained datasets regarding these cases. Section 7 describes the experiments, results, and their comparison with analogs. Section 8 contains a summary of the results and a discussion of the author's technique limitations and future possibilities.

2. Our Earlier Research

Simple cases of source code author attribution were considered in previous work [1]. Approaches based on a deep neural network (NN) and support vector machine (SVM) combined with a fast correlation filter were considered in the mentioned study. Both of them demonstrated high classification accuracy—97% on average depending on the programming language for NN, and SVM was only 1% less efficient at 96%, respectively. An important point is the fact that the NN result was obtained from the analysis of the raw data, and informative features for deciding the authorship were identified by NN independently on the deep layers. SVM, in turn, was expertly trained on the manually performed feature set and filtered by a correlation algorithm. Thus, this research hypothesized the ability of deep NN to identify new, implicit patterns in the original data, which may also be uncontrolled by programmers at the conscious level. The classical SVM method did not demonstrate such an ability as NN, which makes it unstable to deliberate source code transformations and confuse authorship identification. The conducted experiments led us to conclusions about the independence of both methods of identifying the program source code author from the programming language in which the development of the analyzed software is carried out. The qualification of programmers also did not negatively impact the method's effectiveness when solving real-life tasks.

Further work [2] includes the first group of factors that complicated source code author identification. The authors decided to use a technique based on a deep NN as an author's hybrid NN (HNN) based on previous experience. The result of this decision can approve or decline hypotheses about NN's ability to select informative implicit features. First, the experiments considered source codes in five programming languages transformed by different kinds of obfuscators. The obtained results demonstrate the resistance of the chosen approach to lexical obfuscation—the model's accuracy loss did not exceed 10% on average. Second, some experiments were conducted to evaluate the accuracy of authorship determination under conditions when programmers follow a unified coding standard. Linux kernel source codes written following a set of rules presented to contributing users were used as training data. In contrast to obfuscation, coding standards harmed accuracy.

However, in the series of additional experiments, it was revealed that an increase in the number of training data enables a 40% increase in accuracy, and the effectiveness of the model suffices for solving real-life tasks. Thus, the hypothesis proposed in the first study was proven—the developed methodology based on HNN allowed classification in accordance with the authorship both obfuscated and written, according to the coding standards source codes of programs. The second group of factors requires the same careful research and consideration

3. Related Works

Simple statistical methods are not sufficient to achieve the required effectiveness of solving authorship in complex cases. Thus, models that can extract new patterns and dependencies in the data, which are implicit to the researcher, are necessary. These will have been written at the unconscious level and contain intellectual content and features of program code implementation. Such models include deep NN architectures—in particular, their modifications developed in the last five years.

Deep Learning-based Code Authorship Identification System (DL-CAIS) was first presented in [3]. The system, which is based on a random forest (RF) classifier, is appropriate for four different programming languages—C++, Java, Python, and pure C, with the average classification accuracy of 95%. This result is achieved by scaling the classifier through deep vector representations. The representations themselves are formed by computing word frequencies and inverse document frequencies (TF-IDF) and multilevel recurrent NN (RNN).

The authors of [4] propose their deep learning approach named Robust coding style Patterns Generation (RoPGen). This approach uses templates of a unique author's code style. Such types of templates are complicated to imitate by attackers. The main point is to simultaneously augment data and the gradient. It leads to an increase in the diversity of training samples, creates meaningful perturbations of deep NN's gradients, and learns a variety of code style representations. The effectiveness of the proposed method is evaluated on four datasets of source codes in C, C++, and Java. Experimental results show that RoPGen can significantly improve the reliability of deep learning-based code authorship attribution, reducing the success rate of targeted and untargeted attacks to 22.8% and 41%.

Graph NNs are used for determining the source code author in [5]. The proposed approach appears because of the limitations of the dependency expression and semantic relations in source codes in convolutional NNs (CNN). In the author's approach, the program presents in the form of graphs demonstrating complex relations of the author's features. An evaluation of the method was provided on the Google Code Jam (GCJ) [6] dataset. The obtained accuracy reaches 60%.

The authors of [7,8] submit a solution to the problem of the impossibility of adding new authors with no retraining and no interpretability. The decision includes explainable artificial intelligence (XAI) methods. The results of experiments for different programming languages confirm that the model is able to extract distinctive features of the source code authors. The average accuracy of the proposed approaches is 75%.

The work [9] is devoted to a programming language-independent approach to a source code authorship. The authors pay attention to the limitations of existing synthetic attribution datasets and propose a new data collection technique that better reflects aspects that are important for potential commercial use. The authors also argue that the accuracy of modern decision models for author identification drops sharply when their effectiveness is evaluated on more realistic data. They propose two models: path-based NN (PbNN) and modified RF (PbRF). Both models are language-independent and operate on path-based representations of code that can be built for any syntactically correct code fragment. The model accuracy for Java programming language is 97.9% for PbNN and 98.5% for PbRF, respectively.

The authors of [10] proposed the Program Dependence Graph with Deep Learning (PDGDL) methodology, which aimed to identify the authors of source codes written in

C++, Java, and C#. The dataset includes codes of 1000 programmers' from GCJ. The Dependence Graph module is needed to extract features that are later converted into small dimension vectors. Each feature was passed into the Term Frequency Inverse Document Frequency (TF-IDF) method to evaluate its importance and select valuable features. The Deep Learning module is the final step of the methodology. The author's model is RNN with five hidden layers and 25,949 parameters. The average obtained accuracy is 99% for three classes, but the authors did not declare all the results for each of the considered programming languages.

The paper [11] focuses on identifying the unique programmer's writing style to classify a program's source codes. In this way, code features are combined with programmers' demographic—in particular, gender and region. Code features are *n*-grams and statistical values (code length, lines, and function count). All data for the experiments include only C++ source codes. The maximal classification accuracy obtained with RF is 75% in the case of 20 programmers.

In [12], source codes are considered as texts in natural language. Several semantic features such as variable names, indentation style, and length of code are used for training the LSTM-based end-to-end model. The reason to extract these features lies in the fact that such parameters present the coding style of a programmer. Only C++ source codes were extracted from two open-source datasets, GCJ and Codeforces [13]. In total, the data include 2,200 authors and 29,600 code samples. The best accuracies for 25 authors are 75% and 71% for the GCJ and Codeforces datasets, respectively.

The method employed by R. Mateless et al. [14] is appropriate for real-world and team development cases. The main idea is the author's approach called Pkg2Vec, which is based on a hierarchical deep NN. The authors mention several obfuscation methods (e.g., code-naming, encryption, and dead-code-insertion) and claim that their method is robust, since Pkg2Vec is based on the code structure and uses resilient features. The data consist of Android application packages and contains 3297 APKs by 43 teams. The resulting accuracy is 79.8% for a set of 20 programmers.

A recent study by Gorshkov et al. [15] demonstrated their system called StyleIndex, which is based on tokenization, semantic, and unique tokens in code. Only three programming languages—C++, Java, and JavaScript are used as data collected from GitHub, and each language includes 40 authors with 1-300 code files per author. The number of repositories per author for experiments is 1. All noise data that could not be created by the author are removed. Based on these, data are used in the ideal for learning, but not in real-life development. In this form, data have several disadvantages, e.g., authors do not take into account different projects and programming languages per author, their commits, and possible changes in the programmer's writing style in time. The authors use non-standard dataset splitting—60% of data for training and 40% for tests. The most identified programmers write in Java (95.18% for 40 authors), then C++ (94.2% for 40 authors), and finally, Java with 94.16% for 40 programmers.

The following study [16] is conducted on the balanced AI-SOCO dataset for the FIRE-2020 competition aimed at identifying a source code's authorship. AI-SOCO provides 100,000 C++ source codes per 1000 programmers and is divided into training, test, and validation sets for 50,000, 25,000, and 25,000 source codes, respectively. As a model, the stacking of three different classifiers (extra tree classifier, RF, and XG-Boost) is used. Additional research focuses on feature selection and separated experiments provided on different feature sets, such as word and character *n*-grams and splitting code into tokens. The best result of 82.95% was achieved using word bigrams.

García-Díaz J. A. and Valencia-García R. [17] also took part in the FIRE-2020 competition. Their method is based on statistical features (character *n*-grams, letter case, using keywords), TF-IDF, and classical machine learning models (RF, Multi-Layer Perceptron, k-neighbors, SVM, and Naive Bayes). The authors present each source code in two versions: with comments and pure codes. The best accuracy of 91.16% was achieved with the RF classifier on code with comments.

Despite high results, only a small section of the studies considered programs as a complex text structure with integral elements such as comments, the proportions of each programmer's contribution, and, in particular, the heterogeneity of the data. In addition, the mentioned open datasets (GCJ and AI-SOCO '2020) have critical drawbacks and are not suitable for complex experiments due to a lack of programming languages and data for experiments in complex cases. A comparison of the obtained results that are possible for the commercial software development methodology is incorrect due to the following critical factors that still need to be taken into account:

1. Creative element. Each programmer has their own preference for using different patterns and structures. It is impossible to declare all rules, so code writing has a creative part.
2. Number of languages. In most cases, well-known datasets include no more than three languages and do not provide data to evaluate the same programmer's codes written in two or more programming languages, but in practice, it is quite normal to use two and more programming languages in solving everyday routine tasks. However, the author's habits and favorite practices can flexibly move from one language to another, and an optimal approach should take it into account.
3. Experience. With professional growth, a person improves their skills and step-by-step changes to write better code. This fact is important, and training data should include several samples from different time intervals for the same programmer.
4. Team development. Code review and discussion are generally used for a project's practices and act as strong recommendations. Both procedures also have an impact on human factors and are based on the personal experience of the reviewer, so some code specifics change from one team to another, but features that are implicit and uncontrolled by the programmer do not. These features could be helpful in the identification of the source code author.
5. Advanced cases. At present, commits, mixed data, and generated source codes are inseparable from development and methods and should be resistant to complicated tasks. Careful preprocessing and removing noise elements transforms data into the perfect condition but does not provide a realistic view. A novel and accurate approach should keep up to date with modern development techniques and tools.

4. Formal Task Statement

Due to the complicating factors, the previously proposed formal task statement [1] summarized the following: identification of the source code author based on heterogeneous data consists in finding the objective function $y* : S \to P$, where S is the set of source codes, and P is the set of authors–programmers. Models that are obfuscated or aggregated from many fragments of codes can act as $s \in S$. Generative language models can act as $p \in P$. True authors of source codes are known only on a finite training set $S^m = \{(s_1, p_1), \ldots, (s_m, p_m)\}$, where m is the number of source codes with a known author. In this case, the task is to build a classifier $\alpha : S \to P$ capable of determining whether an arbitrary source code $s \in S$ belongs to the true author $p \in P$.

To find the objective function y*, the classifier α must be trained on the feature space. Each individual feature of this space can be described as $f : S \to D_f$, where f is the set of allowed values, and D is the set of text features. Then, the set of features of an arbitrary source code $s \in S$ is the vector $s = \{(f_1, s_1), \ldots, (f_n, s_n)\}$, where n is the dimension of the feature space.

5. Technique for Determining the Author of a Source Code

The great experience of previous scientific works [1,2] confirms the usefulness of the author's HNN-based technique (Figure 1) in identifying the author of a heterogeneous source code in complex cases. The author's HNN demonstrates high efficiency in simple and complex cases determining the authorship of a program and resistance to complicating factors: obfuscation and coding standards. The presented model is a fully independent tool.

It does not require preprocessing and extracting informative features from source codes but works with samples at the symbol level and identifies informative features independently.

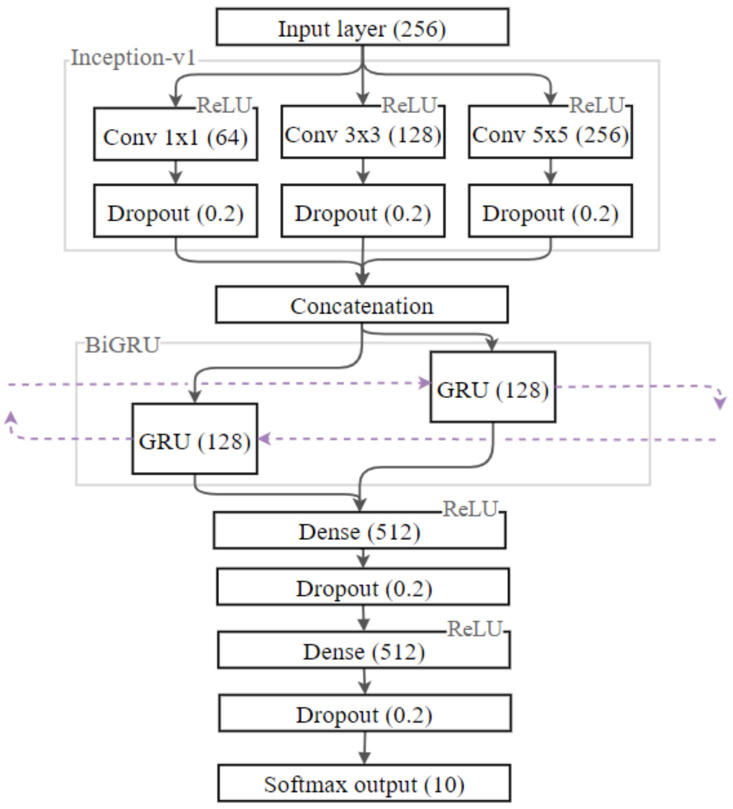

Figure 1. HNN architecture.

HNN performs analysis on vectorized texts. This conversion is performed using one-hot encoding. The method was chosen since the categorical features of the source codes are nominal and require transformation into a binary form. During vectorization, each character of the text sequence is converted into a vector. The vectorized source codes are passed to the input of HNN. The architecture of the proposed HNN is presented in Figure 1:

1. An input layer with dimension corresponds to the vector length. In this case, the length is 256, which corresponds to a vector of 255 zeros and 1 one at the position equal to the character code, according to the ASCII encoding.
2. Inception-v1 layer group. This group includes convolutional layers with kernel dimensions of 1, 3, and 5. Convolutions form a filter that passes only informative features. Convolutions work in parallel. In order to avoid overfitting after each convolution, a Dropout layer with a rate of 0.2 is added, which resets 20% of incoming neurons. The results of the convolutions concatenate into a single vector.
3. Bidirectional Gated Recurrent Units (GRU)—a layer consisting of two independent GRUs. The result of Inception-v1 layers is fed in a direct order to the input of the first network and in reverse order to the input of the second. The outputs of both networks are combined into one vector.
4. Layers with a feedforward connection. The result of the Bidirectional GRU is transmitted to the input of two sequential layers with a feedforward connection. Both layers have dimensions of 512 neurons in order to map outputs to a higher dimensional space that make it easier to classify. Similar to Inception-v1, Dropout layers are applied to feedforward layers.
5. Output layer. Softmax is used as the output layer. This layer can obtain the probability distribution about the belonging of the input sample to each of the classes. The

dimension depends on the number of prediction classes for a particular case. In the figure, the dimension of the layer is 10, which corresponds to 10 potential authors.

The Rectified Linear Unit (ReLU) function is used as an activation function for individual Convolutional and Dense layers. Since the author's architecture includes a large number of neurons, the activation of all of them can lead to high costs of RAM. The ReLU function is non-linear, and thus, provides the possibility of activating only a part of the neurons (sparse activation). This allows a higher learning rate and better convergence to be achieved.

To optimize the HNN, the adaptive learning step method (Adadelta) was chosen. The choice was due to the optimizer's ability to automatically select the learning rate, as well as a stable weights update. The Softmax activation function is a generalized logistic function to define output values as probabilities of belonging to target classes. This function was chosen because of its resistance to the vanishing gradient problem. The loss function was categorical cross-entropy, which is the default for Softmax.

6. Experimental Data

The process of source code author identification should be based on a representative and voluminous dataset. To obtain such data, it is necessary to choose a free access platform. Searching for such a platform was provided among the most popular hosting IT project sites. Given the relevance of languages and source codes written in them, as well as the importance of the possibility of using the application programming interface (API) when collecting data, it was decided to consider the hosts GitHub [18] and GitLab [19]. These are the most popular platforms with regular updates even for the rarest or latest programming languages. Since GitLab is more common among commercial software developers, it is dominated by closed repositories (virtual project repositories), while the collection requires a large number of open ones, which is more typical for the GitHub platform. Thus, GitHub hosting was chosen as a source for collecting and forming a dataset.

Then, a set of programming languages was defined. The main criteria were the popularity of the language and a sufficient number of open repositories on GitHub. According to the rating based on the opinion of commercial software developers, JavaScript, Java, C#, Python, PHP, Kotlin, Swift, C++, Go, Ruby, C, Groovy, and Perl languages were selected because of a sufficient number of repositories on GitHub hosting. Information about the received set of source codes is presented in Table 1.

Table 1. Information about the source code dataset.

Dataset Characteristic	Value of the Characteristic
Number of code lines	20,967,040
Number of projects	569
Number of projects with one author	71
Average commit length in lines	13
Maximum commit length in lines	119,892
Average number of source codes in project	231
Average source code length in lines	169
General number of symbols in code	212,336,637
General number of codes	102,908
General number of authors in all projects	383

6.1. Mixed Data

An important problem of modern studies aimed at source code authorship attribution is the lack of experiments aimed at evaluating developed approaches on mixed datasets.

That there is a need to obtain such an assessment is due to the specificity of modern software development. In most cases, the product is implemented in several programming languages. The set of programming languages used in the development of a particular product can vary and is called its stack. Moreover, each developer has their own stack consisting of languages they use in teamwork.

In most cases, the main part of a program is developed in one language, and the need for another appears only for creating additional modules and tools. However, there are situations when development is carried out equally in two or even three programming languages. Thus, the amount of data collected for training classification models aimed at the source code author attribution may be insufficient for the developed tool to obtain a reliable result. Finally, there is a need to assess the ability of the proposed methodology to demonstrate a comparable result to homogeneous data concerning heterogeneous data—in this case, the source codes of programs implemented in different programming languages.

In order to obtain mixed datasets, it was decided to search for authors–programmers who are contributors to projects in different programming languages. Obtained source codes written by a specific author and located in the repository corresponding to the language were extracted. When forming the dataset, samples of source code were not separated.

6.2. Artificially Generated Source Codes

One of the promising trends in deep learning is text generation in different languages. Modern models designed to generate texts from scratch or based on context show high efficiency [4,20–26]. Machine-written text is often close to human text. This is due to the ability of language models to preserve the author's characteristics—the distinctive features of the author's writing style.

Natural language text generation has proven itself in various areas of human activity and suggested a new direction—the generation of artificial language texts, including source codes. Models allowing the creation of source code by themselves or the completion of the program text for a developer reduce a programmer's time spent writing routine code which does not require any research or creative activity. Using generative models to implement source code for commercial software development is very promising and useful. Nevertheless, the appearance of the described tools only increases the importance of identifying the source code author issue.

The technique for identifying the author of an artificially generated source code should not only effectively identify the informative features of the author's writing style, but also be able to, at minimum, distinguish authorship between different generative models and hence, identify distinctive features of creating code artificially.

The developers of solutions that are designed to generate source codes also consider the positive experience of GPT models. The most famous tools (Open AI Codex, GitHub Copilot, AlphaCode, JARVIS Sber AI, and PolyCoder [27–31]) appeared based on them. All mentioned tools are retrained on the voluminous source code datasets. Samples that were freely available on GitHub IT hosting or on major competitive platforms were used for training some tools; for others, their source code base. An excess of benchmark data suitable for training generative models leads to the achievement of sufficient accuracy for them to create a compiled and workable code.

Although most of the existing source code generation solutions show impressive results, their use in this study is impossible since they are close to the development community. AlphaCode, the only open-source solution, is unsuitable since it is designed only for developing a competition code, and, therefore, the use of ready-made solutions is not possible. According to the analysis, all considered tools are based on models of the GPT family. Thus, we decided to use our own GPT-3 model, which was retrained on the code base collected from GitHub hosting.

6.3. Source Code Commits

The most common among the complex cases of heterogeneous data analysis is the identification of the source code author using commits (the last changes of the source code in the repository). Since the tools of program authorship identification are considered the most demanded in the commercial environment, the specifics of work in this field should be taken into account. Modern IT companies use different technologies for the flexible collaborative development of software products and use different version control systems in their daily activities.

According to the principles of teamwork development, most programs and software packages are created by several developers together. Development is carried out by the whole team, not a lone programmer. Contributions of team members can have different weights—for example, one developer wrote 90% of all code, while the other only fixed the bugs found during testing and wrote a couple of lines in total. To identify the source code author, it is important to be able to correctly extract the programmer's code samples and distinguish commits according to the names of the users who push this code.

By generating mixed datasets, GitHub API was used to extract information about individual developers' commits in the repository. In this case, the obtained information was used to generate a training dataset. Lines of source code uploaded by a particular programmer are written to a separate file, thus, creating a program text that may not always compile or work, but which still stores the author's style.

7. Experiment Setup and Results

A generalized experimental methodology is demonstrated as a framework diagram in Figure 2. A detailed description of each phase can be found below in this section.

A dataset that was appropriate to each complex case was generated to train and evaluate the author's HNN model. The code samples meet the following criteria:

1. The length of the source code must be at least 30 symbols and no more than 3000 symbols.
2. No more than 30 samples of source code per author are selected.
3. Sample selection is random.

According to dataset criteria, the minimum number of code lines for training per author is 90, and the maximum is 90,000. These limits were chosen based on the statistics of the collected dataset. The number of epochs for training HNN is five iterations of 10 epochs, for a total of 50 epochs. A callback procedure named Early Stopping, which automatically interrupts the training, was applied in this study. Practice shows that 50 epochs are sufficient to obtain a minimum error rate and high accuracy. A larger number of epochs leads to an increase in the value loss and overfitting. A smaller number of epochs does not enable high accuracy to be achieved.

A 10-block cross-validation method was used to evaluate the model's accuracy. The principle of this algorithm comprises 10 iterations, each of which includes sequential training of the model on nine blocks of data and evaluating it on the one remaining block.

In this study, the following experimental cases were considered:

- Mixed data, including two programming languages (language pairs);
- Mixed data, including three or more languages;
- Distinction of authorship between man and generative models;
- Determining the unity of the generative model for pairs of samples;
- Distinction of authorship between generative models;
- Data obtained from contributors' commits.

According to the results, the average classification accuracy for authors who use two programming languages is 87% (Figure 3). At the same time, it should be noted that the accuracy of identifying the author based on language pairs with similarities in syntax exceeds the accuracy of language pairs with no similarities. This finding is especially correct for JavaScript–C++, Swift–Java, and JavaScript–Python language pairs.

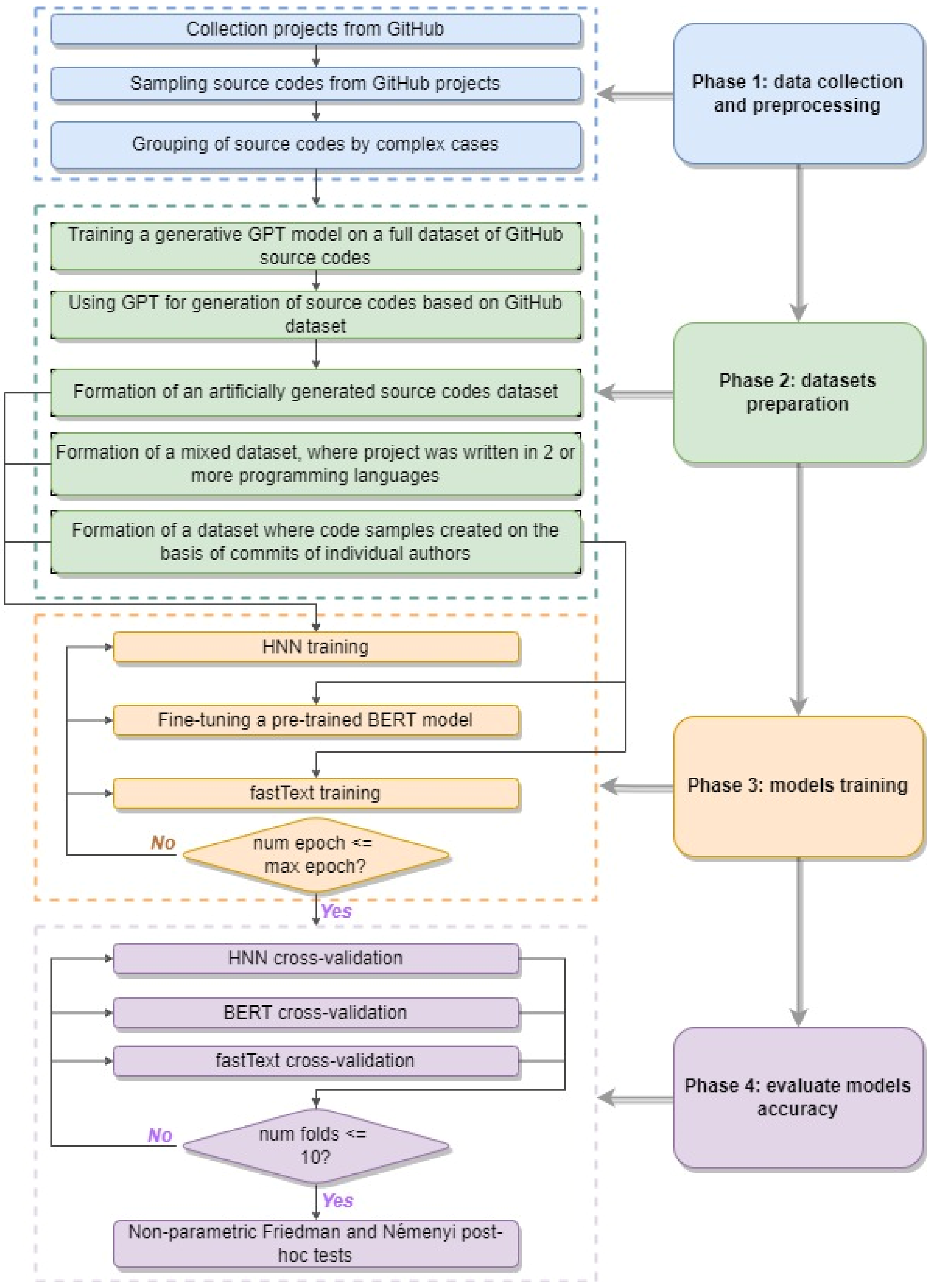
Collection projects from GitHub
Sampling source codes from GitHub projects
Grouping of source codes by complex cases
Phase 1: data collection and preprocessing
Training a generative GPT model on a full dataset of GitHub source codes
Using GPT for generation of source codes based on GitHub dataset
Formation of an artificially generated source codes dataset
Formation of a mixed dataset, where project was written in 2 or more programming languages
Formation of a dataset where code samples created on the basis of commits of individual authors
Phase 2: datasets preparation
HNN training
Fine-tuning a pre-trained BERT model
fastText training
No
num epoch <= max epoch?
Yes
Phase 3: models training
HNN cross-validation
BERT cross-validation
fastText cross-validation
No
num folds <= 10?
Yes
Non-parametric Friedman and Némenyi post-hoc tests
Phase 4: evaluate models accuracy

Figure 2. A generalized experimental methodology.

The average accuracy for authors developing in three or more programming languages is 76% (Figure 4). Since datasets are totally mixed and can include more than ten programming languages per author, it is impossible to evaluate the importance level of each language. It should be noted that this case is particularly of research interest since it is extremely rare in practice.

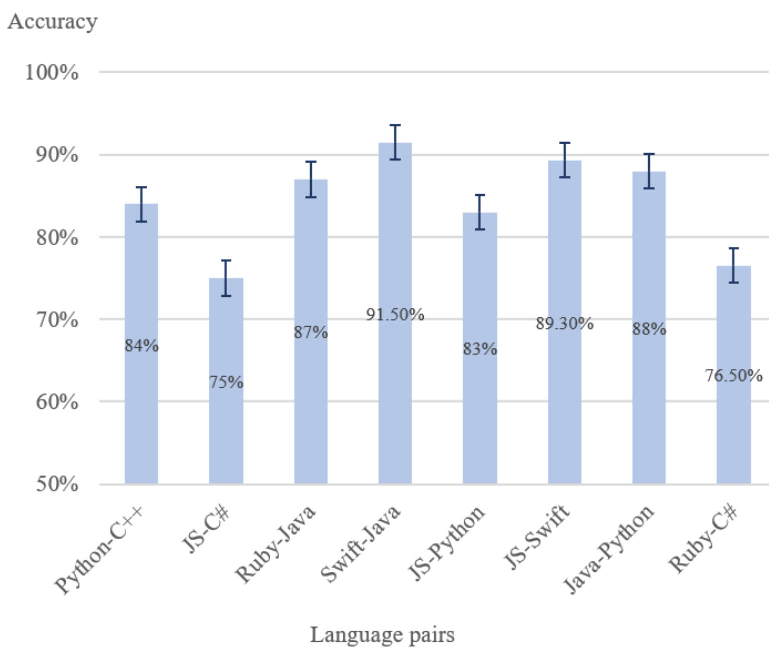

Figure 3. Classification accuracy for language pairs.

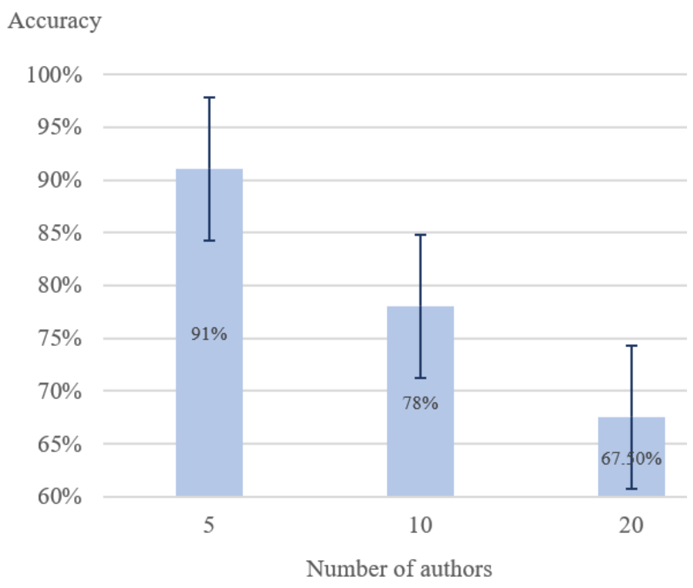

Figure 4. Classification accuracy for different language combinations.

The results achieved on the mixed datasets prove the high efficiency of the author's technique. The obtained accuracy of 76% for datasets containing three or more programming languages, and 87% for datasets consisting of language pairs, exceed the accuracy of some technique analogs [32,33] that were obtained for simple identification cases.

Three complex cases were addressed using artificially generated source code. The first and the simplest case was finding a difference between human and generative models (Figure 5). In this case, the evaluation was performed as follows: for each experiment (5, 10, and 20 authors), one of the authors was a generative model. One part of the experimental samples was written by humans, and another was generated by models of the GPT family: GPT-2, GPT-3, RuGPT-3 (from Sber AI). All samples were written in the most popular programming language—Java. The average accuracy for GPT-2 was 97%, and 94% for GPT-3 and RuGPT-3. The difference in accuracy can be explained by the fact that the third generation models show better generative ability than the second generation. Consequently, the separability of the author's model also decreases.

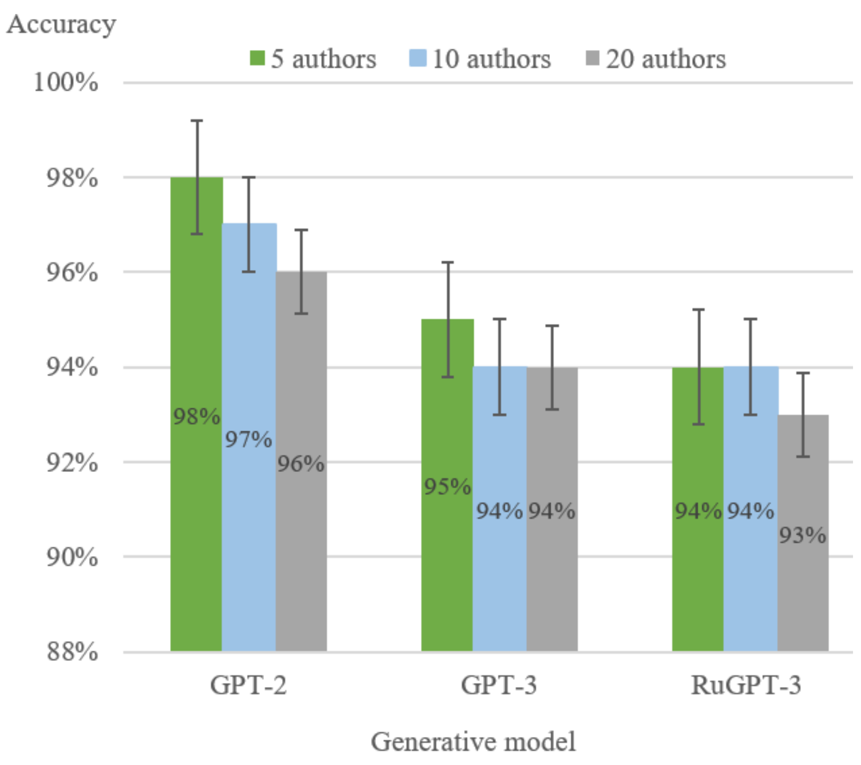

Figure 5. Difference between human and generative models.

The second experiment was devoted to assessing the accuracy of determining the unity of the generative model using two samples of source code (Figure 6). Two source code samples generated by the same language model were used in this experiment. The study involved 50 Java source code samples split into pairs for each generative model, respectively. These pairs were alternately fed into the input of the model trained for 5, 10, and 20 authors, one of which was the target generative model. The average accuracy was 94% for GPT-2, 92% for GPT-3, and 91% for RuGPT-3.

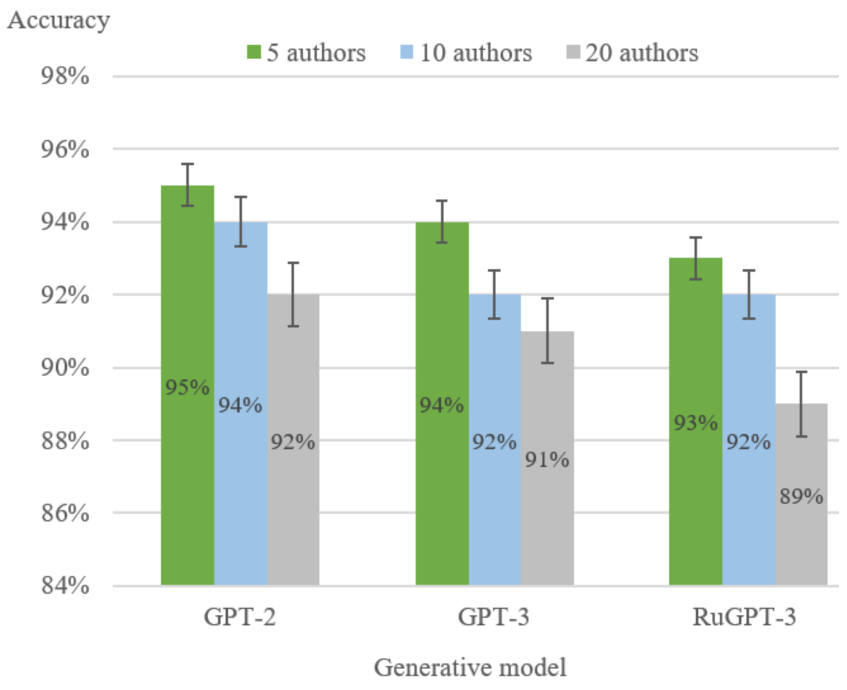

Figure 6. The unity of the generative model for pairs of source code samples.

The third and final experiment on artificially generated source codes was aimed at evaluating model accuracy in distinguishing authorship between three different generative models (Figure 7). In this case, the experiment was set up on three generative models that participated in the training: GPT-2, GPT-3, and RuGPT-3. The purpose of HNN was to determine whether the anonymous source code, which was artificially generated, belonged to one of the language models. The experiment was conducted in five programming languages: Java, C++, Python, JavaScript, and Go. The average accuracy of determining the authorship of a source code generated by a language model was 94%.

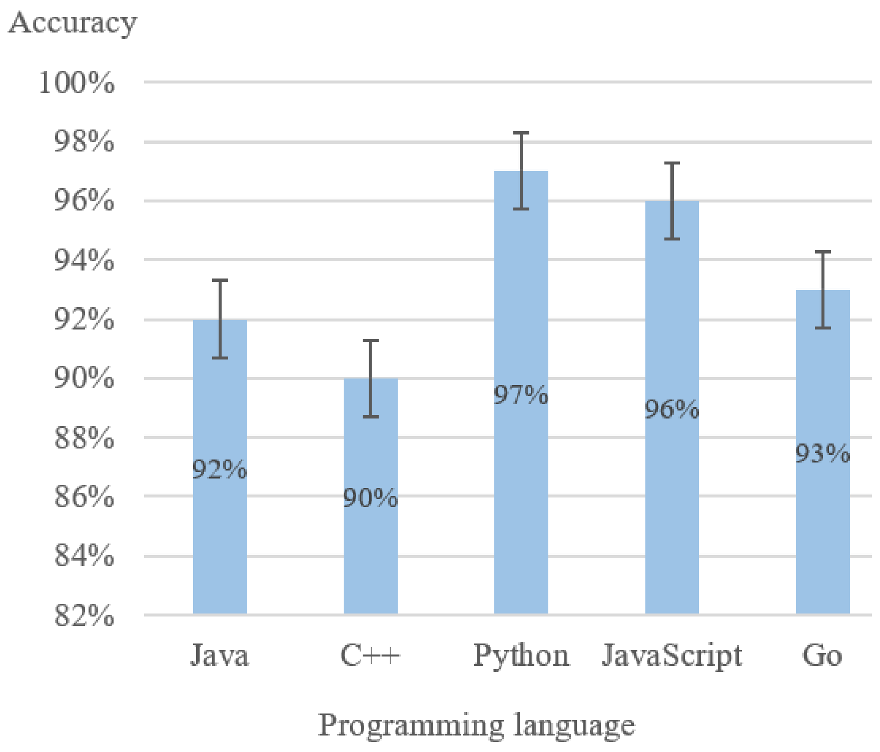

Figure 7. Difference between generative models.

The last experiment focused on repository contributors' commits (Figure 8). The dataset was generated from the commit samples (fragments modified in a common source code file by a particular user) of individual users. To identify methodological recommendations for identifying the authors, which develop commercial software and use version control systems, we decided to experiment with a different number of files—5, 10, and 20, respectively. According to the results, at least 10 source code files per programmer should be used to obtain a highly accurate tool for making decisions about authorship. If this condition is not met, the accuracy may be insufficient for efficient problem solving.

The most important case is the commit base data, due to the prevalence of commercial software development. Based on this fact, we decided to apply for comparison two other generally well-known models—fastText [34] and BERT [35]. Our decision is driven by their high efficiency in solving problems in related fields [36,37].

Bert-base-multilingual-cased was used as a pre-trained version of BERT. Model tuning was performed according to the recommendations of the model's creators: learning rate = 3×10^{-5}, warmup proportion = 0.1, train batch size = 16 and 5 epochs. For fastText training, we used experimentally selected parameters: learning rate = 0.1, rate of updates for the learning rate = 100, size of the context window = 5, number of negative samples = 5, loss function—SoftMax, and 50 epochs.

Figure 9 shows a graph of BERT accuracies depending on the number of authors and samples for training, and a similar graph for fastText is shown in Figure 10.

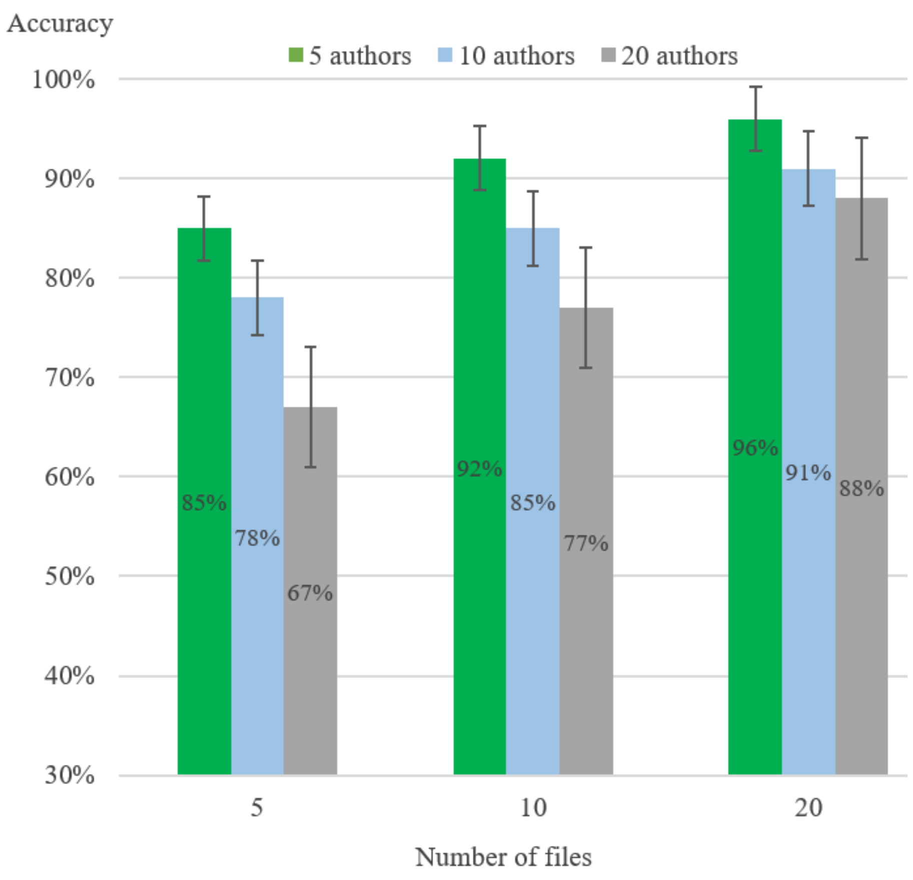

Figure 8. Authorship identification accuracy based on commits.

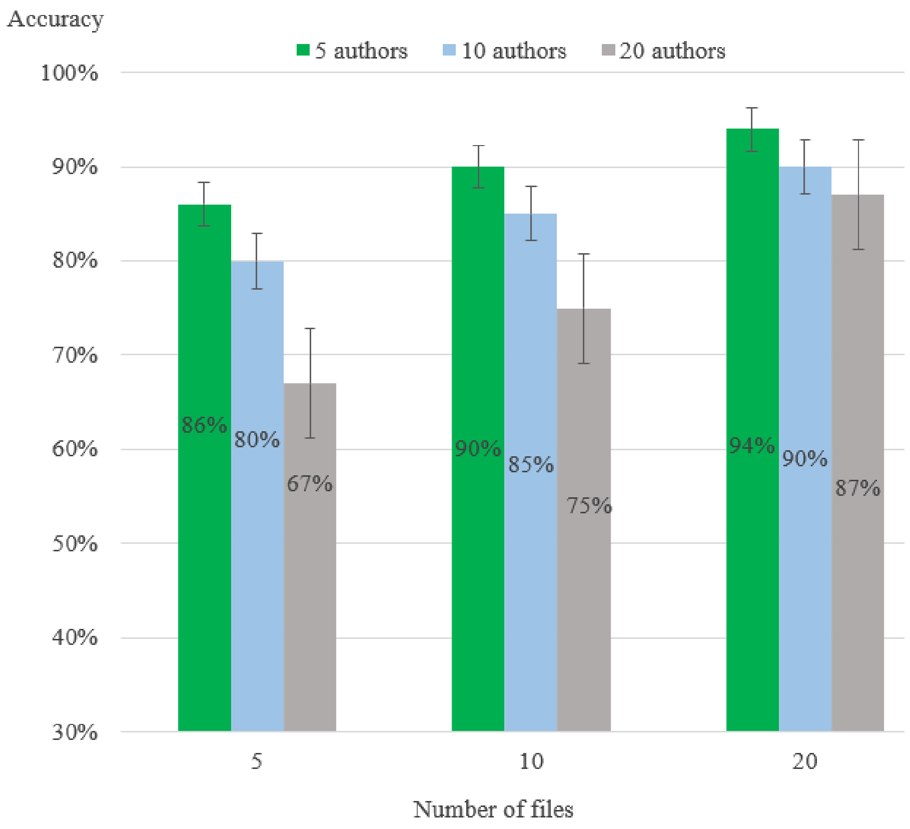

Figure 9. Authorship identification accuracy based on commits (BERT).

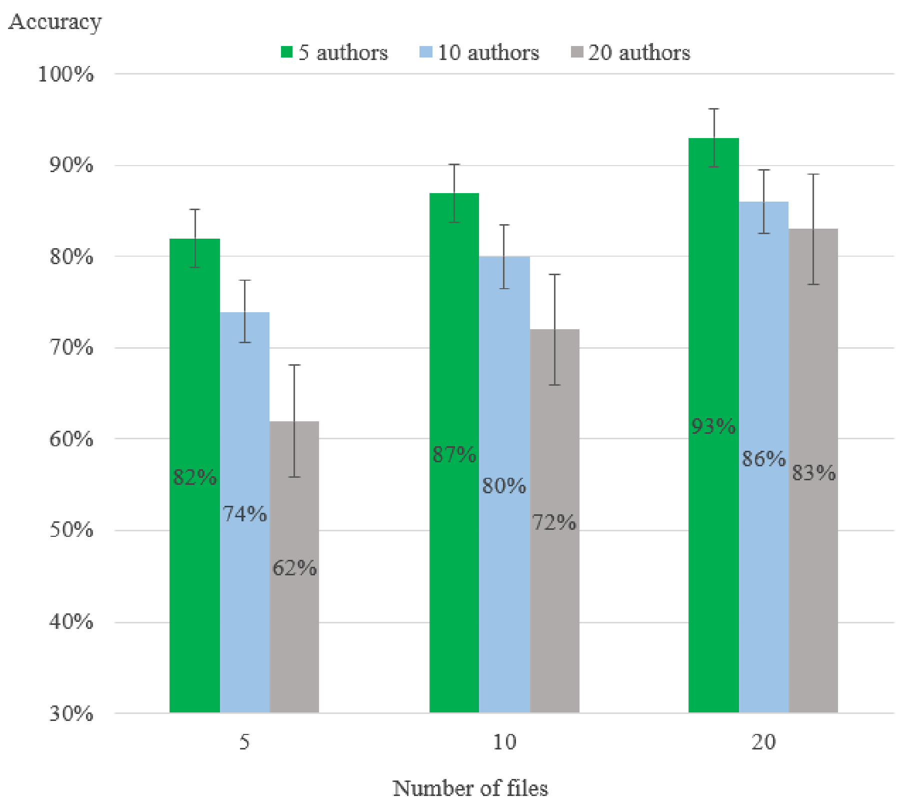

Figure 10. Authorship identification accuracy based on commits (fastText).

Non-parametric Friedman and Némenyi post hoc tests were used to check if there is a statistically significant difference between HNN, BERT, and fastText. The tests were applied to the results of the models' cross-validation for 5, 10, and 20 authors. The number of training samples per author is 20. A null hypothesis is that a difference between the results of the models is random. An alternative hypothesis is that a difference is statistically significant. According to the results of the calculations, the p-value was 0.000079 for five authors, 0.000056 for 10, and 0.000093 for 20. The null hypothesis was rejected since the p-value did not exceed 0.05 in any case. Friedman's statistical test confirmed the significance of the difference between the results. Therefore, it is considered that the efficiency of the models differs significantly if the average ranks of the models differ by a critical difference or more. The Némenyi post hoc test was used to estimate the difference after rejecting the null hypothesis. To set a graphical interpretation of the test results, a Demšar significance diagram was plotted. These diagrams show differences in accuracy between model pairs. If the difference between the average ranks of a pair of models is less than the calculated critical difference value (marked as CD in Figure 11), then the difference in their efficiency is also insignificant. This difference between the average ranks is represented by a horizontal line on the diagram.

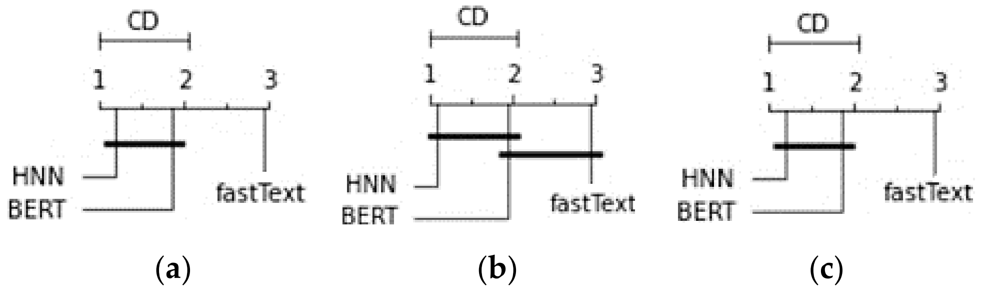

Figure 11. Demšar significance diagram: (**a**) 5 authors, (**b**) 10 authors, (**c**) 20 authors.

We make the following conclusions from the diagrams that are presented in Figure 11:

1. For the case of five authors, the difference in efficiency between HNN and BERT is insignificant. However, HNN has a higher rank, which means that it is able to achieve more accurate results in solving the problem.
2. For the case of 10 authors, HNN is the most accurate model. BERT shows a little difference in comparison with HNN and fastText. However, fastText has the lowest rank, so BERT is considered as a less efficient model.
3. For the case of 20 authors, the difference in the classification results of HNN and BERT is not significant, and fastText shows significantly lower efficiency.

Based on the analysis of the test results, we can conclude that the HNN results are statistically significant. In some cases, the difference in accuracy between HNN and BERT may not be significant. However, it is important to note that BERT is a much more resource-intensive architecture because it has numerous internal parameters. For this reason, the BERT training time exceeds HNN by a factor of 3.

Table 2 compares the results of studies on source code author identification performed over the past seven years with the results obtained in this paper.

Table 2. Comparison of source code authorship identification approaches.

Author	Method	Complex Cases	Dataset	Programming Language	Average Accuracy
Ours	HNN	Obfuscation, Encoding standards, Mixed data, Artificially generated data, Commit-based data	GitHub (ours)	C++ Java JS Python C C# Ruby PHP Swift Go Groovy Kotlin Perl	92% 97% 92% 95% 96% 96% 95% 92% 98% 93% 99% 91% 96%
			GCJ	C++ Java Python	98% 99% 98%
Abuhamad M., AbuHmed T., Mohaisen A. Nyang D [3]	DL-CAIS	Obfuscation	GCJ	C++ Java Python	97% 100% 100%
Zhen L., Chen G., Chen C., Zou Y., Xu S. [4]	RoPGen	-	GCJ	C++ Java	92% 98%
			GitHub	C Java	84% 90%
Holland C., Khoshavi N., Jaimes L.G. [5]	GNN	-	GCJ	C# C++ Java	60% (avg.)
Bogdanova A., Romanov V. [7,8]	XAI	-	GCJ	C++ Java Python	74% 77% 72%
Bogomolov E., Kovalenko V., Rebryk Y., Bacchelli A., Bryksin T. [9]	PbRF	-	GCJ	Java	98%
Caliskan-Islam A., Harang R. [38]	FuzzyAST, RF	Obfuscation	GCJ	C C++ Python	93% 98% 88%
Ullah F., Wang J., Jabbar S., Al-Turjman F. [10]	PDGDL	-	GCJ	C# C++ Java	99% (avg.)
Bayrami P., Rice J.E. [11]	RF, *n*-grams	-	GitHub	C++	75%
Caldeira R.S. [12]	LSTM	-	GCJ Codeforces	C++	75% 71%

Table 2. *Cont.*

Author	Method	Complex Cases	Dataset	Programming Language	Average Accuracy
Mateless R. et al. [14]	Pkg2Vec	-	APKs	Java	79%
Gorshkov et al. [15]	StyleIndex	-	GitHub	C++ Java JavaScript	94% 95% 94%
Suman C., Raj A., Saha S. [16]	Stacked models	-	AI-SOCO	C++	82%
García-Díaz J. A., Valencia-García R. [17]	RF	-	AI-SOCO	C++	91%

The experimental results were obtained for the dataset collected from GitHub and for the open GCJ dataset. The first dataset contains codes of developers programming in several languages. The provided samples can be noisy without the author's obvious writing style features (this is due to following coding standards) and transformed by third-party software. The source codes of the same programmer may contain solutions of completely different problems, which can also confuse the classifying models. The second dataset contains cleaned samples. The conduction of the author's features analysis is not complicated by external factors. Samples contain solutions of the same type of problems, so the model can focus only on the search for informative features and avoid noise. Therefore, the results for these datasets significantly differ. The difference can reach 10% for the same programming language.

The high accuracy obtained by [15] for the dataset from GitHub is due to careful preliminary cleaning and data preparation. Authors use one repository per author, that is, they do not take into account style changes over time. In addition, their dataset only presents projects written by the same author in the same programming language, which guarantees data homogeneity. The accuracy obtained by the authors is only valid under ideal conditions, which are impossible in real-life development processes. In the approaches of other authors where GitHub data are used without careful processing [4,10], the achieved results are inferior in accuracy to our HNN for all programming languages.

Since most modern approaches do not consider complex cases besides obfuscation, we decided to compare the three methods with the example of identifying the author of the source code that was obfuscated with the Tigress tool. The results are presented in Table 3.

The identification of an obfuscated source code of a C program was considered in [3] and [38]. The author's HNN has superior accuracy to competitors on datasets that are obfuscated with Tigress. The author's technique is effective and exceeds analogs in both simple and complex cases. Neither the complicating factors of the first group (obfuscation, coding standards) nor the second (heterogeneous data) have a significant negative impact on the results. The author's technique is highly accurate for 13 different programming languages, and, in comparison, competitors' methods are well-adapted for 2–3 of the most popular programming languages. The proposed technique takes into account changes in the programmer's style as a result of improving the skills and experience of teamwork.

The comparative analysis demonstrates that the proposed technique takes into account all possible applied problems of the source code author identification. The author's technique is effective in both simple and complex cases. Neither the complicating factors of the first group (obfuscation, coding standards) nor the second group (heterogeneous data) have a significant negative impact on the accuracy.

Table 3. Comparison of approaches for obfuscated data.

Author	Method	Programming Language	Obfuscator	Dataset	Accuracy
Ours	HNN	JS	JS Obfuscator Tool	GitHub	86%
				GCJ	91%
			JS-obfuscator	GitHub	86%
				GCJ	90%
		Python	Opy	GitHub	87%
				GCJ	91%
			Pyarmor	GitHub	70%
				GCJ	77%
		PHP	Yankpro-po	GitHub	89%
				GCJ	92%
			PHP Obfuscator	GitHub	82%
				GCJ	89%
		C++	C++ Obfuscator	GitHub	71%
				GCJ	79%
		C	Tigress	GitHub	90%
				GCJ	95%
Abuhamad M., AbuHmed T., Mohaisen A., Nyang D. [3]	DL-CAIS	C	Tigress	GCJ	93%
Caliskan-Islam A., Harang R. [38]	FuzzyAST, RF	C	Tigress	GitHub	67.2%

8. Conclusions

This paper is devoted to identifying the author of a heterogeneous source code using HNN. The author's technique has proven itself when solving the problems of authorship determination in simple and complex cases.

As part of the study, several experiments aimed at assessing the effectiveness of the author's technique for the cases of mixed datasets, artificially generated data, and heterogeneous data presented in the form of commits were conducted.

The results of the experiments confirm the high efficiency of the developed technique based on HNN in both complex and simple cases of identifying a source code author.

For authorship classification cases where an author–programmer knows two programming languages, technique accuracy is 87%; for three or more languages, this is 76%. Another case is the training of the model on commits that are aggregated from source codes. For this case, the average accuracy ranges from 87% to 96% and depends on the sufficiency of the training data.

Finally, the most relevant case is associated with artificially generated code due to the active development of artificial text generation technologies. The study addresses three different issues: the first is the distinction of authorship between the generative model and a human; the second is related to the question of whether two code samples belong to the same language model; the third is the definition of the language model that generated the source code. For data that are generated pre-trained on the source code from GitHub-hosting models of the GPT family, the average accuracy is 94%. The proposed technique allows not only the analysis of both human and machine-written source codes, but also separating the authorship between them, as well as identifying the distinctive features of each language model.

The proposed technique has the following advantages:

1. Identifying the author of a source code with high accuracy.
2. Independence from the programming language and author–programmer qualification.

3. The stability to deliberate source code conversions through the use of obfuscators or coding standards.
4. The ability to train the model on source codes created by the development team.
5. The ability to identify informative features indicated to source code created by the particular generative model.

This technique formed the basis of an intelligent system for identifying the author of a source code, which can be used to solve real-life problems.

Author Contributions: Supervision, A.R., A.S.; writing—original draft, A.K., A.R.; writing—review and editing, A.R., A.F.; conceptualization, A.K., A.R., A.F.; methodology, A.R., A.K.; software, A.K.; validation, A.F., A.K.; formal analysis, A.R., A.F.; resources, A.S.; data curation, A.S., A.R.; project administration, A.R.; funding acquisition, A.S. All authors have read and agreed to the published version of the manuscript.

Funding: This research was funded by the Ministry of Science and Higher Education of Russia, Government Order for 2020–2022, project no. FEWM-2020-0037 (TUSUR).

Data Availability Statement: Not applicable.

Acknowledgments: The authors express their gratitude to the editor and reviewers for their work and valuable comments on the article.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Kurtukova, A.V.; Romanov, A.S. Identification author of source code by machine learning methods. *Tr. SPIIRAN* **2019**, *18*, 741–765. [CrossRef]
2. Kurtukova, A.; Romanov, A.; Shelupanov, A. Source Code Authorship Identification Using Deep Neural Networks. *Symmetry* **2020**, *12*, 2044. [CrossRef]
3. Abuhamad, M.; AbuHmed, T.; Mohaisen, A.; Nyang, D. Large-Scale and Language-Oblivious Code Authorship Identification. In Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security, Toronto, ON, Canada, 15–19 October 2018; pp. 101–114.
4. Zhen, L.; Chen, G.; Chen, C.; Zou, Y.; Xu, S. RoPGen: Towards Robust Code Authorship Attribution via Automatic Coding Style Transformation. In Proceedings of the 2022 IEEE 44th International Conference on Software Engineering (ICSE), Pittsburgh, PA, USA, 25–27 May 2022; pp. 1906–1918.
5. Holland, C.; Khoshavi, N.; Jaimes, L.G. Code authorship identification via deep graph CNNs. In Proceedings of the 2022 ACM Southeast Conference (ACM SE '22), Virtual, 18–20 April 2022; pp. 144–150.
6. Google Code Jam. Available online: https://codingcompetitions.withgoogle.com/codejam (accessed on 18 August 2022).
7. Bogdanova, A.; Romanov, V. Explainable source code authorship attribution algorithm. *J. Phys.* **2021**, *2134*, 012011. [CrossRef]
8. Bogdanova, A. Source code authorship attribution using file embeddings. In Proceedings of the 2021 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity, Zurich, Switzerland, 17–22 October 2021; pp. 31–33.
9. Bogomolov, E.; Kovalenko, V.; Rebryk, Y.; Bacchelli, A.; Bryksin, T. Authorship attribution of source code: A language-agnostic approach and applicability in software engineering. In Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, 23–28 August 2021; pp. 932–944.
10. Ullah, F.; Wang, J.; Jabbar, S.; Al-Turjman, F.; Alazab, M. Source code authorship attribution using hybrid approach of program dependence graph and deep learning model. *IEEE Access* **2019**, *7*, 141987–141999. [CrossRef]
11. Bayrami, P.; Rice, J.E. Code authorship attribution using content-based and non-content-based features. In Proceedings of the 2021 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE), Ottawa, ON, Canada, 12–17 September 2021; pp. 1–6.
12. Caldeira, R.S. A Deep Learning Approach to Recognize Source Code Authorship. Available online: https://maups.GitHub.io/papers/tcc_008.pdf (accessed on 18 August 2022).
13. Codeforces. Available online: https://codeforces.com/ (accessed on 18 August 2022).
14. Mateless, R.; Tsur, O.; Moskovitch, R. Pkg2Vec: Hierarchical package embedding for code authorship attribution. *Future Gener. Comput. Syst.* **2021**, *116*, 49–60. [CrossRef]
15. Gorshkov, S.; Nered, M.; Ilyushin, E.; Namiot, D.; Sukhomlin, V. Source code authorship identification using tokenization and boosting algorithms. In Proceedings of the International Conference on Modern Information Technology and IT Education, Moscow, Russia, 29 November–2 December 2018; Springer: Cham, Switzerland, 2018; pp. 295–308.

16. Suman, C.; Raj, A.; Saha, S.; Bhattacharyya, P. Source Code Authorship Attribution using Stacked classifier. In Proceedings of the Forum for Information Retrieval Evaluation, FIRE (Working Notes), Hyderabad, India, 16–20 December 2020; pp. 732–737.
17. García-Díaz, J.A.; Valencia-García, R. UMUTeam at AI-SOCO '2020: Source Code Authorship Identification based on Character *N*-Grams and Author's Traits. In Proceedings of the Forum for Information Retrieval Evaluation, FIRE (Working Notes), Hyderabad, India, 16–20 December 2020; pp. 717–726.
18. GitHub. Available online: https://GitHub.com/ (accessed on 18 August 2022).
19. Gitlab. Available online: https://gitlab.com/ (accessed on 18 August 2022).
20. Rothe, S.; Narayan, S.; Severyn, A. Leveraging pre-trained checkpoints for sequence generation tasks. *Trans. Assoc. Comput. Linguist.* **2020**, *8*, 264–280. [CrossRef]
21. Du, Z. All nlp tasks are generation tasks: A general pretraining framework. *arXiv* **2021**, arXiv:2103.10360.
22. Floridi, L.; Chiriatti, M. GPT-3: Its nature, scope, limits, and consequences. *Minds Mach.* **2020**, *30*, 681–694. [CrossRef]
23. Lee, J.S.; Hsiang, J. Patent claim generation by fine-tuning OpenAI GPT-2. *World Pat. Inf.* **2020**, *62*, 101983. [CrossRef]
24. Dusheiko, A. Lead Generation of News Texts using the ruGPT-3 Neural Network. Master's Thesis, 2022.
25. Pisarevskaya, D.; Shavrina, T. WikiOmnia: Generative QA corpus on the whole Russian Wikipedia. *arXiv* **2022**, arXiv:2204.08009.
26. Cruz-Benito, J. Automated source code generation and auto-completion using deep learning: Comparing and discussing current language model-related approaches. *AI* **2021**, *2*, 1–16. [CrossRef]
27. Open AI. Available online: https://openai.com/blog/openai-codex (accessed on 18 August 2022).
28. GitHub Copilot. Available online: https://copilot.GitHub.com (accessed on 18 August 2022).
29. AlphaCode. Available online: https://deepmind.com/blog/article/Competitive-programming-with-AlphaCode (accessed on 18 August 2022).
30. Sber AI ruGPT-3. Available online: https://developers.sber.ru/portal/tools/rugpt-3 (accessed on 18 August 2022).
31. Polycoder. Available online: https://venturebeat.com/2022/03/04/researchers-open-source-code-generating-ai-they-claim-can-beat-openais-codex/ (accessed on 18 August 2022).
32. Frantzeskou, G.; Stamatatos, E.; Gritzalis, S. Identifying authorship by bytelevel *n*-grams: The source code author profile (SCAP) method. *Int. J. Digital. Evid.* **2007**, *1*, 1–18.
33. Wisse, W.; Veenman, C.J. Scripting DNA: Identifying the JavaScript Programmer. *Digit. Investig.* **2015**, *15*, 61–71. [CrossRef]
34. FastText. Available online: https://fasttext.cc/ (accessed on 18 August 2022).
35. BERT. Available online: https://huggingface.co/docs/transformers/model_doc/bert (accessed on 18 August 2022).
36. VGCN-BERT. Available online: https://arxiv.org/abs/2004.05707 (accessed on 18 August 2022).
37. Bag of Tricks for Efficient Text Classification. Available online: https://aclanthology.org/E17-2068/ (accessed on 18 August 2022).
38. Caliskan-Islam, A. Deanonymizing programmers via code stylometry. In Proceedings of the 24th USENIX Security Symposium, Washington, DC, USA, 12–14 August 2015; pp. 255–270.