Efficient Source Code Authorship Attribution using Code Stylometry Embeddings

David Álvarez-Fidalgo[1][a], Francisco Ortin[1,2][b]
[1]*Computer Science Department, University of Oviedo, c/Calvo Sotelo 18, Oviedo, Spain*
[2] *Computer Science Department, Munster Technological University, Rossa Avenue, Bishopstown, Cork, Ireland*
{uo270571, ortin}@uniovi.es, francisco.ortin@mtu.ie

Keywords: Source Code Authorship Attribution, Code Stylometry Embeddings, CLAVE, Machine Learning.

Abstract: Source code authorship attribution or identification is used in the fields of cybersecurity, forensic investigations, and intellectual property protection. Code stylometry reveals differences in programming styles, such as variable naming conventions, comments, and control structures. Authorship verification, which differs from attribution, determines whether two code samples were written by the same author, often using code stylometry to distinguish between programmers. In this paper, we explore the benefits of using CLAVE, a contrastive learning-based authorship verification model, for Python authorship attribution with minimal training data. We develop an attribution system utilizing CLAVE stylometry embeddings and train an SVM classifier with just six Python source files per programmer, achieving 0.923 accuracy for 85 programmers, outperforming state-of-the-art deep learning models for Python authorship attribution. Our approach enhances CLAVE's performance for authorship attribution by reducing the classification error by 45.4%. Additionally, the proposed method requires significantly lower CPU and memory resources than deep learning classifiers, making it suitable for resource-constrained environments and enabling rapid retraining when new programmers or code samples are introduced. These findings show that CLAVE stylometric representations provide an efficient, scalable, and high-performance solution for Python source code authorship attribution.

1 INTRODUCTION

Authorship attribution or identification is the task of identifying the author of an anonymous document based on a set of known authors, using linguistic, stylistic, and structural features as distinguishing markers, comprising what is also known as stylometry (He et al., 2024). This task has been widely studied in the context of both literary and forensic applications, where determining the origin of a text can provide valuable insights into authorship disputes, criminal investigations, and intelligence analysis.

When applied to source code, authorship attribution is the task of identifying the programmer who wrote of a given piece of source code (Kalgutkar et al., 2019). Source code authorship attribution commonly draws upon concepts from natural language processing (NLP), machine learning, and software forensics to uncover coding characteristics that differentiate one programmer from another. This task has applications in cybersecurity, intellectual property protection, and forensic investigations (Ou et al., 2023). In cybersecurity, authorship attribution is used for tracking malware creators, identifying the source of security vulnerabilities, and detecting unauthorized code reuse. In legal and intellectual property contexts, it aids in verifying ownership claims and detecting plagiarism in software development. In forensic investigations, code authorship attribution can assist law enforcement in identifying cybercriminals by analyzing malicious scripts, ransomware, or illicit software.

Figure 1 illustrates how the same program in a given language can be written in different styles, called code stylometry. The programmer who wrote the Python code in the upper part of Figure 1 includes type annotations for function parameters and return values (Ortin et al., 2022), whereas the one in the lower part does not. Additionally, the former provides

[a] https://orcid.org/0009-0006-9121-1349
[b] https://orcid.org/0000-0003-1199-8649

```python
def calculate_factorial(number: int) -> None:
    """Calculate the factorial of a given number iteratively."""
    if number == 0 or number == 1:
        return 1 # Base case: factorial of 0 or 1 is 1
    result = 1
    for i in range(2, number + 1):
        # Start from 2 since 1 is redundant
        result *= i
    return result
```

```python
def fact(n): return 1 if n in (0, 1) else n * fact(n - 1)
```

Figure 1: Variations in code stylometry for a Python function computing the factorial.

explicit documentation through comments and selects longer, more descriptive identifier names, while the latter opts for brevity. Distinct programming styles are also evident in the choice of control structures: one implementation follows an imperative approach using an `if-else` statement, whereas the other adopts a more functional style, utilizing a ternary conditional expression. These stylistic choices not only impact readability and maintainability but also result in differences in code length and structure. Furthermore, the first function implementation computes the factorial using iteration, while the second one relies on recursion, highlighting variations in algorithmic preference.

Code authorship verification aims to determine whether two code excerpts were written by the same programmer, without having seen the author during training (Stamatatos et al., 2023). The key difference between authorship verification and authorship attribution is that attribution seeks to identify the specific author of a code sample from a predefined set of candidates, whereas verification focuses on determining whether two code samples originate from the same author without prior knowledge of potential candidates. In this sense, attribution is a multi-class classification problem in which the authors are known during training, while verification is a binary classification task that assesses whether two given code samples share the same authorship, even when the author was not part of the training data.

In previous work, we developed CLAVE, a deep learning (DL) model for source code authorship verification that employs contrastive learning and transformer encoders (Álvarez-Fidalgo and Ortin, 2025). CLAVE is pre-trained on 270,602 Python source code files and fine-tuned using code from 61,956 distinct programmers for Python authorship verification. The model achieves an AUC of 0.9782, outperforming state-of-the-art source code authorship verification systems. Given CLAVE's strong performance in authorship verification, we investigate its potential effectiveness for authorship attribution when only a few source code files written by a predefined set of possible programmers are available. To explore this possibility, we develop several code authorship attribution systems based on CLAVE, other authorship verification systems, and Large Language Models (LLMs) encoders pre-trained on source code. These systems are then compared with current DL approaches to assess their relative performance.

Therefore, the main contribution of this paper is the development of a source code authorship attribution system for the Python programming language that, using CLAVE, achieves classification performance similar to existing DL approaches with just six source files per programmer. Moreover, it consumes significantly fewer CPU and memory resources than similar DL approaches, enabling rapid retraining when a new programmer needs to be classified, or additional source code samples are included in the dataset. The accuracy of the authorship attribution system significantly improves CLAVE's capability to function as a zero-shot classifier.

Derived from the previous contribution, these are research questions addressed in this paper:

- RQ1. Is CLAVE a suitable system for building a source code authorship attribution model?
- RQ2. Is there any performance benefit on training a classifier from CLAVE embeddings to develop an authorship attribution system with just a few samples of code for each programmer (compared to using the authorship verification system)?
- RQ3. Is it possible to build an authorship attribution system with CLAVE with just six samples of code per programmer with a classification performance similar to the existing deep learning authorship attribution systems?

- RQ4. Is it possible to (re)train an authorship attribution system with CLAVE with significantly lower CPU and memory resources than deep-learning approaches?

The rest of this paper is structured as follows. Section 2 reviews related work, while Section 3 provides a summary of CLAVE. Section 4 describes the methodology, and Section 5 presents the results. Section 6 offers a discussion, and Section 7 concludes the paper.

2 RELATED WORK

Alsulami et al. (2017) utilize Abstract Syntax Tree (AST) information extracted from source code to develop a DL model for source code authorship attribution. They implement both a Long Short-Term Memory (LSTM) and a Bidirectional Long Short-Term Memory (BiLSTM) model to automatically extract relevant features from the AST representations of programmers' source code. Their dataset consists of 700 Python files from 70 programmers across 10 problems, sourced from the Google Code Jam annual coding competition. Their best model, BiLSTM, achieves an accuracy of 0.96 when classifying 25 Python programmers and 0.88 when the number of programmers increases to 70.

Kurtukova et al. (2020) employ a hybrid neural network (HNN) that combines Convolutional Neural Networks (CNN) with Bidirectional Gated Recurrent Units (BiGRU) to build a code authorship attribution system for Python. They conducted different experiments with 5, 10, and 20 different programmers (10 to 30 files per programmer), achieving a 0.85 accuracy for 20 programmers. Their approach leverages CNNs to capture local structural patterns in the code, while BiGRU layers help model long-range dependencies.

DL-CAIS is a Deep Learning-based Code Authorship Identification System designed for code authorship attribution (Abuhamad et al., 2018). It employs a DL architecture that integrates a TF-IDF-based deep representation with multiple Recurrent Neural Network (RNN) layers and fully connected layers. This deep representation is then fed into a random forest classifier, enhancing scalability for de-anonymizing authors. The system is trained on the entire Google Code Jam dataset from 2008 to 2016, as well as real-world code samples from 1,987 public repositories on GitHub. DL-CAIS achieves an accuracy of 94.38% in identifying 745 C programmers.

Li et al. (2022) create the RoPGen DL model for authorship attribution. The key idea is to combine data augmentation and gradient augmentation during adversarial training to make RoPGen robust against adversarial attacks. This approach enhances the diversity of training examples, generates meaningful perturbations to neural network gradients, and helps learn varied representations of coding styles. They used four datasets of Java, C, and C++ programs to train their model.

White and Sprague (2021) also present a neural network architecture based on a bidirectional LSTM that learns stylometric representations of source code. The authors train this network using the NT-Xent loss function that is part of the SimCLR framework. The trained network is evaluated for authorship verification by comparing the cosine similarity of the representations. It is also used for authorship attribution with a support vector machine classifier. The network achieves a verification accuracy of 92.94% on a dataset consisting of C/C++ programs from the years 2008 to 2017 of Google Code Jam.

Hozhabrierdi et al. (2020) propose FDR (Feedforward Duplicated Resolver), a method for authorship verification that employs a neural network originally trained for authorship attribution. The network is fed novel features called Variable-Independent Nested Bigrams that are extracted from the abstract syntax tree of each program. The network is initially trained to classify source code files into a set of known authors. Then, a portion of the trained network is repurposed to generate representations from source code, which are subsequently compared for authorship verification. They achieve an AUC of 0.96 for the task of predicting whether a pair of samples from 43 unknown authors have been written by the same person.

Ou et al. (2023) propose a neural network architecture for source code authorship verification that combines a bidirectional Long Short-Term Memory (LSTM) with a Generative Adversarial Network (GAN). This architecture is trained as a Siamese network to learn stylometric representations of source code, with the GAN ensuring that the derived representations do not contain information about the functionality of the original source code files. These representations are then compared using cosine similarity to determine whether a pair of source code files were written by the same author.

Wang et al. (2018) introduce a neural network that accepts a set of manually engineered features extracted from pairs of programs and outputs the probability that the pair was written by the same author. These features include both static (extracted

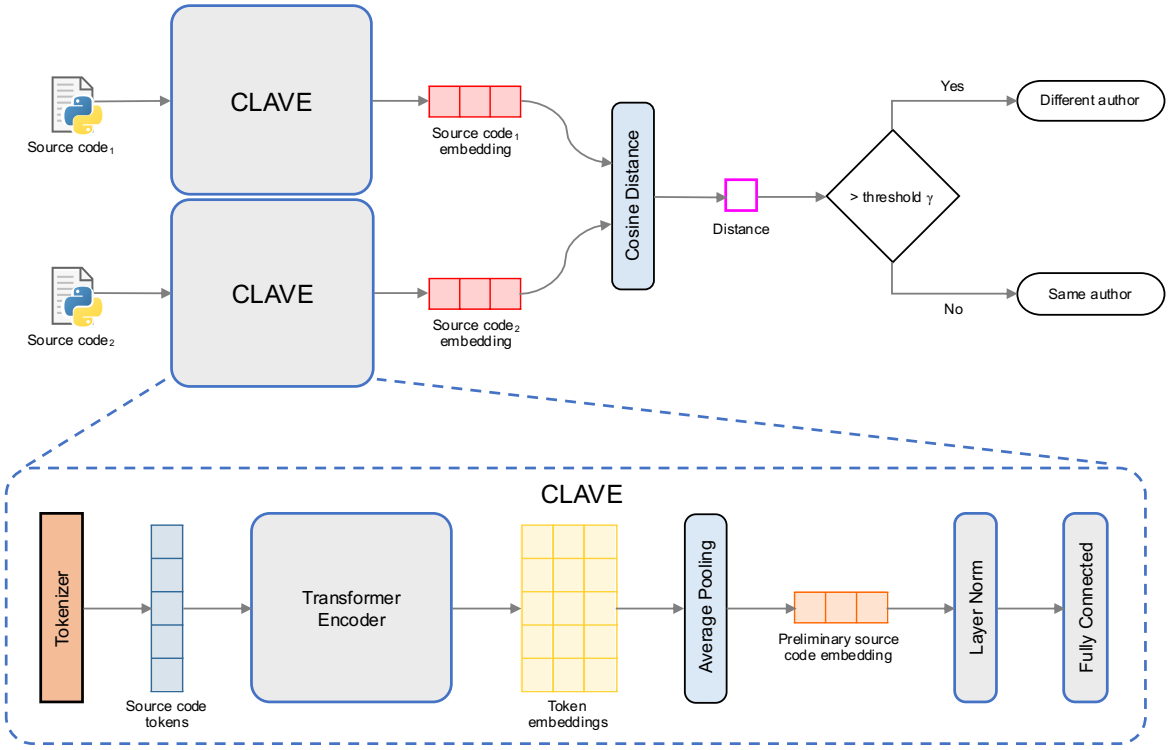

Figure 2: Architecture of the CLAVE source code authorship verification system.

without running the program) and dynamic (extracted after running the program) information, such as run time or memory consumption.

3 CLAVE

As previously mentioned, the code attribution system presented in this article builds upon our prior work, CLAVE: Contrastive Learning for Authorship Verification with Encoder Representations (Álvarez-Fidalgo and Ortin 2025). Figure 2 illustrates CLAVE's architecture, which processes source code as input and produces a single vector (embedding) that encapsulates the stylometric representation of the code. Specifically, embeddings of code written by the same author are positioned closely in the vector space, whereas those from different authors are farther apart.

Since CLAVE generates a single embedding that captures the stylometry of the input code, it is well-suited for code authorship verification. To this end, two source code excerpts are passed to CLAVE, producing two stylometric embeddings (upper section of Figure 2). If the cosine distance (1 − cosine similarity) between these embeddings exceeds a threshold γ, the excerpts are predicted to be written by different authors. Otherwise, they are considered to be from the same author.

The first component of CLAVE (lower section of Figure 2) is a SentencePiece tokenizer for Python, which converts input source code into a vector representation. This vector is then fed into a 6-layer Transformer Encoder that outputs a matrix containing an embedding for each input token. The Transformer Encoder follows the architecture proposed by Vaswani et al. (2017) and, similar to BERT (Devlin et al., 2019), employs learned positional embeddings instead of sinusoidal encoding. Additionally, it uses the GELU activation function instead of ReLU, as GELU has demonstrated superior performance in various language processing tasks (Hendrycks and Gimpel, 2023). The encoder is configured with a model dimension d_{model} = 512, feedforward network dimension d_{ff} = 2048, and h = 8 attention heads.

To obtain a single embedding representing the entire source code input, the token embedding matrix generated by the encoder undergoes pooling, producing a fixed-length vector regardless of input size. This embedding is then normalized to mitigate internal covariate shift, enhancing network stability and performance during training. Finally, a fully connected layer with a ReLU activation function refines the representation, leveraging the information encoded in the pooled embedding. The resulting

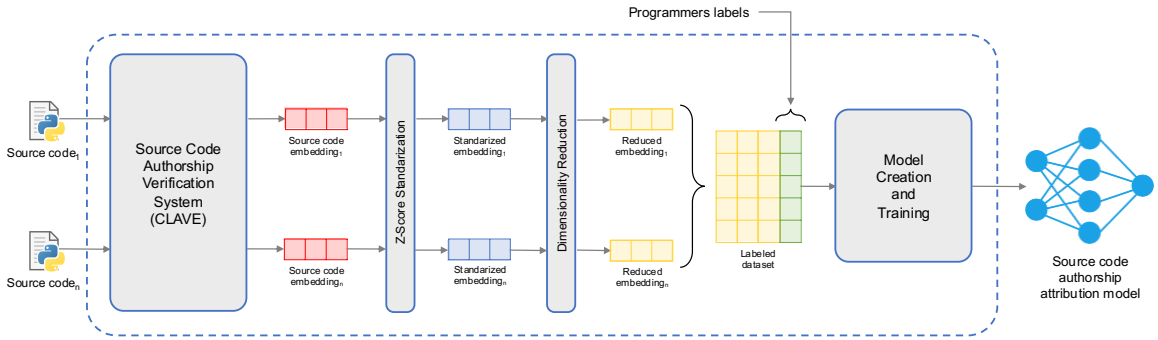

Figure 3: Architecture of the source code authorship attribution system.

vector serves as the model's output, encapsulating the stylometric characteristics of the source code.

4 METHODOLOGY

The proposed system for source code attribution is illustrated in Figure 3. The system receives *n* source code files written by *p* different programmers (Section 4.1). Each input file is processed by a code authorship verification system—CLAVE and other systems are evaluated, as detailed in Section 4.2— to generate an embedding that captures the stylometric features of the code. The embedding is then standardized using a z-score transformation to ensure that all features have a mean of zero and a standard deviation of one, enhancing the stability and performance of subsequent processing steps. Since the dimensionality of these embeddings ranges from 512 to 1024, and our system is designed to operate with a limited number of samples, we explore various dimensionality reduction techniques (Section 4.4). The reduced embeddings are then compiled into a dataset, with each instance labeled according to the programmer who authored the code.

Next, we train multiple classifiers (Section 4.3) using different numbers of programmers to build source code authorship attribution models. The trained models are then evaluated and compared against existing work, while we also analyze CPU and memory usage during training (Section 4.5).

During inference, the classification model operates similarly. The source code from an unknown programmer is processed by the code authorship verification system, generating an embedding. That embedding is subsequently standardized and reduced using the optimal dimensionality reduction technique (if any) identified during hyperparameter tuning (Section 4.4). The resulting vector is then passed to the classifier, which predicts the most likely programmer label.

4.1 Dataset

To train the classifiers, we collected Python source code files from different programmers. Since our goal is for the classifiers to learn the stylometric features of programmers rather than the functionality of the code, we selected source code written to solve the same problems by different individuals. To this end, we used data from the Google Kick Start programming competitions held between 2019 and 2022.

In these competitions, participants solve seven distinct programming problems in their preferred language. They can submit multiple versions of their solutions, but only the final submission is evaluated. We considered only Python code from programmers who submitted a non-empty solution for all seven tasks, using their final submission in each case. Additionally, to ensure no overlap with CLAVE's training data, we excluded files and programmers that were used during CLAVE's training phase.

The final dataset consists of 85 Python programmers, each contributing seven source files, resulting in a total of 595 programs. Since all programs address the same set of problems, classification cannot be based on functionality. The dataset is intentionally small to assess whether the classifiers can be trained efficiently in terms of time and memory while still achieving high classification performance with limited samples.

Figure 4 presents the distribution of non-empty lines of code (LoC) across the dataset. As shown, the Python programs are relatively short: 80% of the files contain 36 LoC or fewer, with a median of 21 and a mean of 29.13 LoC. This short length of code per file presents an additional challenge for the classifier.

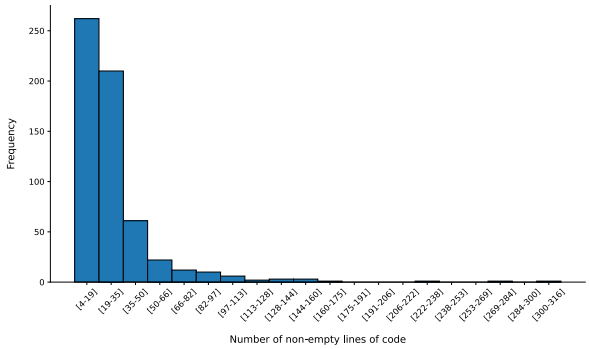

Figure 4: Distribution of Python non-empty lines of code in the dataset.

4.2 Selected Code Authorship Verification Systems

The first component of the code authorship attribution system in Figure 3 is the authorship verification system. We assess the state-of-the-art systems discussed in Section 2, including CLAVE (Álvarez-Fidalgo and Ortin, 2025), FDR (Hozhabrierdi et al., 2020), SCS-GAN (Ou et al, 2023), and the model proposed by White and Sprague (2021). All models were trained using the same fine-tuning dataset, as detailed by Álvarez-Fildago and Ortin (2025).

Additionally, we leveraged cutting-edge Large Language Models (LLMs) specifically trained for source code. Based on the recent survey of LLMs for code by Zhang et al. (2024), we selected the specialized encoder models pre-trained on source code: CodeBERT, GraphCodeBERT, SynCoBERT, Code-MVP, SCodeR, and CodeSage, excluding models unavailable for download (SynCoBERT, Code-MVP, and SCodeR). We also included StarEncoder, a recent high-performing encoder-based LLM for source code that is not covered in Zhang et al.'s survey.

Since these LLMs generate a matrix of token embeddings as output, we applied average pooling to obtain a single embedding representation of the source code, following the same approach we used for CLAVE.

4.3 Classification Models

Another key component of the code authorship attribution system in Figure 3 is the classification model. We have selected the following classifiers:

- k-Nearest Neighbors (k-NN), a non-parametric, instance-based learning algorithm that classifies a new data point based on the most common label among its k nearest neighbors in the training set. This approach is suitable for our problem, as the authorship verification system generates embeddings that place files written by the same programmer in close proximity.
- Support Vector Machines (SVM), which identify a hyperplane that best separates classes in a high-dimensional space, using maximum margin and kernel functions to handle non-linearity. SVM is well suited for high-dimensional embeddings, ensuring good generalization, and is robust even with small datasets, as in our case.
- Multiclass Logistic Regression (LR), a linear classifier that models the probability of each class using the softmax function. It serves as a simple yet effective baseline model for classification.
- Random Forest (RF), an ensemble of decision trees where each tree is trained on a random subset of features and data. By aggregating predictions across multiple trees, Random Forest reduces variance and improves generalization. It is capable of capturing non-linear patterns in the data.
- Multi-Layer Perceptron (MLP), a fully connected feedforward neural network that learns non-linear mappings between inputs and outputs through multiple layers of neurons. It can capture complex stylometric patterns in embeddings through non-linear transformations and is particularly effective when programmers' styles exhibit hierarchical relationships in the embedding space. MLP requires tuning multiple hyperparameters (Section 4.4).
- Extreme Gradient Boosting (XGBoost), an ensemble learning method that builds decision trees sequentially, where each new tree corrects errors from the previous ones to improve accuracy. XGBoost is well suited for high-dimensional embeddings, performs effectively on structured data such as stylometric embeddings, and includes built-in regularization and pruning mechanisms to mitigate overfitting.

4.4 Hyperparameter Search and Model Training

We selected six files per programmer for training (510 files) and one file per programmer for testing (85 files). To identify the best hyperparameters for each configuration of the architecture in Figure 3, we

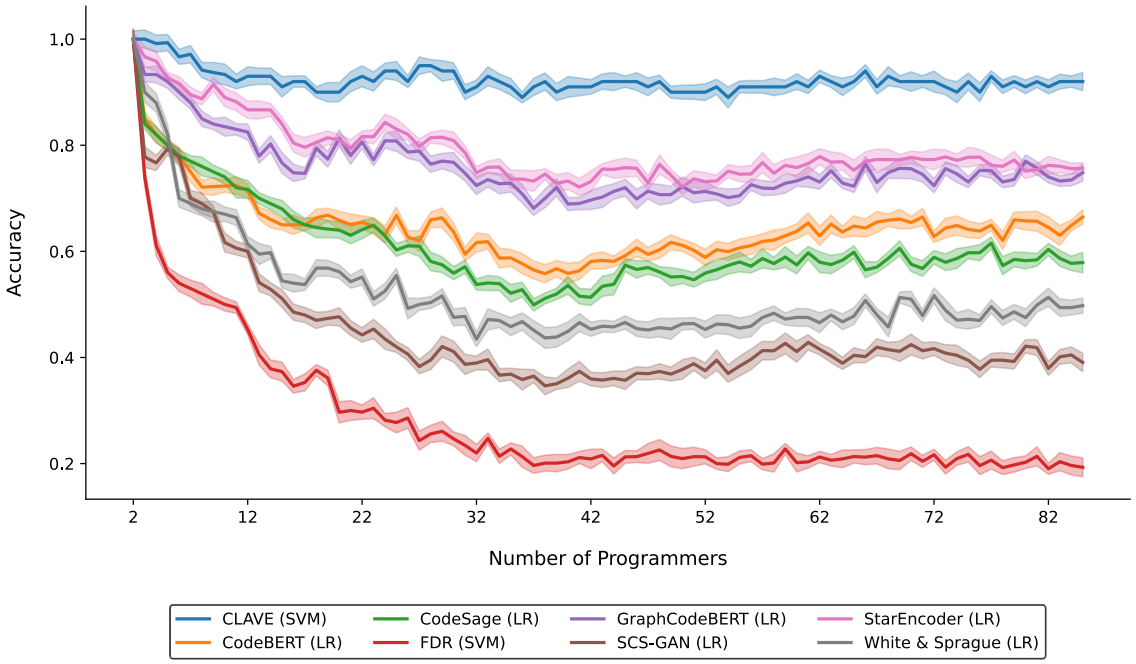

Figure 5. Accuracy of the source code authorship attribution systems as the number of programmers increases, using the best classifier for each verification system and LLM. Shaded areas represent 95% confidence intervals. SVM denotes Support Vector Machine, and LR stands for Logistic Regression.

conducted stratified cross-validation with six folds. At each iteration, five programs from each programmer were used for training, while the remaining program was reserved for validation. This ensures that the classifier is evaluated on unseen data while maintaining a balanced representation of each programmer across the folds.

The architecture outlined in Figure 3 involves several hyperparameters that significantly influence the models' learning and generalization capabilities. To identify the best combinations of hyperparameters, we performed an exhaustive grid search. In each iteration, the model was trained using a specific set of hyperparameters, and its performance was evaluated based on the accuracy on the sample used for validation, as the dataset is perfectly balanced. Accuracy is defined as the ratio of correctly classified instances to the total number of instances in the dataset.

An important architectural hyperparameter is the dimensionality reduction component shown in Figure 3, which we treat as an additional hyperparameter. We explored the following alternatives. We used Principal Component Analysis (PCA) to reduce the embedding dimensions while retaining 99% and 95% of the explained variance. PCA is an effective and computationally efficient algorithm when the stylometric features of the code embeddings exhibit linear dependencies. Since the dataset is labeled, we also experimented with Linear Discriminant Analysis (LDA) for projecting the data onto a space where the classes are more distinct, thus improving the classifier's discriminative power, reducing to *programmers-1* dimensions. Additionally, we applied Uniform Manifold Approximation and Projection (UMAP) for non-linear dimensionality reduction, testing reductions to 10, 30, and 50 dimensions. The last option in the hyperparameter search is no dimensionality reduction algorithm.

We also searched for different hyperparameters for each of the six classification models discussed in Section 4.3. The list of parameters searched and the best-performing combinations can be found in Ortin and Álvarez-Fidalgo (2025).

To assess the statistical significance of the results, we employed bootstrapping with 10,000 repetitions to calculate 95% confidence intervals for each metric (Noma et al., 2021). This process involved repeatedly sampling with replacement from the test set to generate multiple resampled datasets. The confidence intervals for the average of each metric were then computed across all resampled datasets, allowing us to evaluate whether there were statistically significant

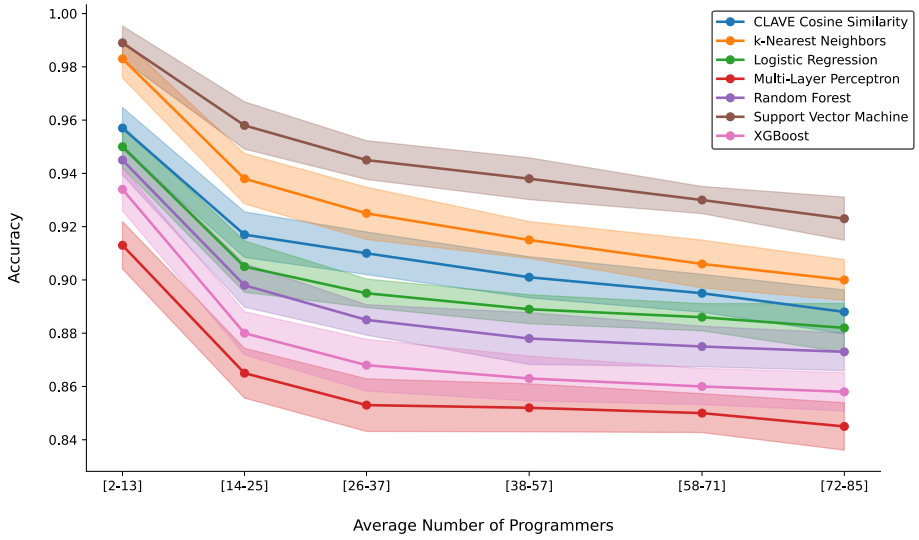

Figure 6. Accuracy of the classifiers trained with the CLAVE source code authorship verification systems as the number of programmers increases. Shaded areas represent 95% confidence intervals. The CLAVE cosine similarity classifier assigns authorship based on the closest embedding to the sample to be predicted, using cosine similarity distance.

differences between the compared systems (Georges et al., 2007).

4.5 Execution Time and Memory Consumption

Besides its classification performance, one of the strengths of the proposed system is its low CPU and memory requirements for training a code authorship attribution model with data from multiple programmers. As discussed in Section 2, the systems with the highest performance are typically based on DL models, which require considerable time to train (Rodriguez-Prieto et al., 2025).

We measure the execution time of training each model with the full training dataset, using the best hyperparameters identified. Training is performed 30 times to compute the average execution time and its 95% confidence interval, following the methodology suggested by Georges et al. (2007).

For memory consumption, we utilized the `psutil` tool to monitor the memory usage during the training process (Rodola, 2025). We measure the maximum resident set size throghout the training process execution, which represents the non-swapped physical memory a process has used (Ortin et al., 2023). This allows us to quantify the RAM consumption and evaluate the efficiency of the system in terms of memory resources.

Hyperparameter search and model training were conducted on an Intel Core i9-10900X CPU (3.7GHz), with 48 GB of RAM, without the use of a GPU, and running an updated 64-bit version of Windows 11. We used Python 3.13.2 for model training and hyperparameter search, and scikit-learn 1.6.1 and XGBoost 2.1.4 for model creation.

5 RESULTS

Figure 5 presents the accuracies of the source code authorship attribution systems trained using the method described in Section 4. For each system, the classifier with the best performance is shown—all the values can be consulted in Ortin and Álvarez-Fidalgo (2025). As the number of programmers increases, the accuracy of all models tends to decline. The SVM classifier with CLAVE embeddings achieves the highest accuracy, outperforming all other systems significantly—for 85 distinct programmers, its accuracy is 22% higher than that of the second-best system. The best classifiers for the different systems were SVM and LR.

Figure 6 presents the accuracies of classifiers trained with CLAVE embeddings, averaged over groups of 12 programmers to smooth fluctuations and better visualize trends. SVM achieves the highest accuracy, with statistically significant differences from the next-best classifier (k-NN)—95% confidence intervals do not overlap (Georges et al. 2007). The third-best system, CLAVE Cosine Similarity, assigns authorship based on the closest

Table 1: Execution times (in seconds) for training each classifier with 85 programmers and 6 program files per programmer (510 files in total), without hyperparameter search. 95% confidence intervals were below 3.5%.

	CLAVE	CodeBERT	CodeSage	FDR	Graph CodeBERT	SCS-GAN	Star Encoder	White & Sprague
k-NN	0.136	0.119	0.074	0.062	0.158	0.021	0.123	0.043
Logistic Regression	3.765	5.559	4.863	4.707	5.193	5.134	4.487	4.819
MLP	0.550	1.002	1.226	18.160	4.965	13.076	6.652	18.813
Random Forest	1.160	0.632	0.963	0.459	0.870	0.807	1.301	0.495
SVM	0.156	0.185	0.240	0.133	0.192	0.123	0.200	0.119
XGBoost	37.402	25.827	75.195	11.384	56.108	9.946	54.237	8.512

Table 2: Memory consumption (in MBs) for training each classifier with 85 programmers and 6 program files per programmer (510 files in total), without hyperparameter search. 95% confidence intervals were below 1%.

	CLAVE	CodeBERT	CodeSage	FDR	Graph CodeBERT	SCS-GAN	Star Encoder	White & Sprague
k-NN	155.20	155.31	159.47	154.65	155.71	155.07	155.66	157.60
Logistic Regression	158.31	160.74	163.12	155.36	160.98	155.49	160.92	156.32
MLP	162.01	162.18	167.71	158.07	163.07	157.03	162.86	160.21
Random Forest	187.10	167.57	167.68	161.38	167.52	178.56	212.83	173.37
SVM	157.58	163.56	160.89	156.41	159.38	155.74	159.26	156.79
XGBoost	343.79	212.25	232.49	179.46	294.43	212.15	216.77	234.80
Verification	154.01	154.66	155.24	153.86	155.33	153.99	154.59	154.09

embedding to the prediction sample using cosine similarity distance. All classifiers exhibit a similar trend, with accuracy decreasing as the number of programmers increases.

Table 1 presents the execution times for training the models with 85 programmers and 6 files per programmer, without hyperparameter search, on the computer described in Section 4.5. k-NN and SVM require the least CPU resources, while XGBoost has the highest computational cost. No single authorship verification system consistently achieves the shortest training times: CLAVE and White & Sprague yield the lowest times for two classifiers each, while FDR and SCS-GAN do so for one classifier each.

The memory consumption during model training is presented in Table 2. The last row displays the memory usage of the base code authorship verification model (or base LLM) without a classification model, which is a value between 153.86 and 155.33 MBs. The differences in memory consumption across verification systems are less than 1%, and all the classifiers increase the baseline memory requirement. On average, k-NN, SVM, and LR require, respectively, 1%, 2.7%, and 2.9% more memory than the base verification model. RF increases memory usage by 14.6%, while XGBoost leads to a 55.9% growth.

6 DISCUSSION

The SVM classification algorithm trained with CLAVE embeddings has been evaluated as the code authorship attribution system with the highest performance, outperforming the second-best classifier not using CLAVE embeddings by 22% (Figure 5). This demonstrates that the stylometry vectors generated by CLAVE are well-suited for source code attribution tasks (**RQ1**).

Figure 5 illustrates the power of LLM encoders when trained with large datasets of source code. With just six Python files per programmer, LLMs achieve higher accuracy for authorship attribution than the three state-of-the-art code authorship verification systems (Section 2). CLAVE is the only exception, outperforming all four LLM models for source code evaluated.

The classification error (*1-accuracy*) when using SVM, compared to using only CLAVE authorship verification (selecting the closest embedding to the prediction sample using cosine similarity), is reduced by 45.4% (Figure 6). This result suggests that training a classifier with CLAVE embeddings for authorship attribution is preferable to simply using the authorship verification system, even when only six

Python files per programmer are available for training (**RQ2**). However, it is worth noting that this improvement is observed with SVM and k-NN algorithms for such a small number of samples; the other algorithms show similar or worse performance than using the CLAVE authorship verification system.

We compare our SVM CLAVE system with state-of-the-art DL models for Python authorship attribution, as discussed in Section 2. Alsulami et al. (2017) train a BiLSTM model on 700 Python files from 70 different programmers. With more samples and fewer programmers than our study, their classification error (0.12) is 55.8% higher than ours (0.077). Kurtukova et al. (2020) propose another authorship attribution model for Python, training a hybrid CNN and BiGRU neural network that achieves 0.85 accuracy for 20 programmers. The rest of the systems discussed in Section 2 use C, C++, or Java code. These results show that the SVM CLAVE authorship attribution system for Python provides competitive performance when compared to existing DL models (**RQ3**).

The design of our model facilitates better generalization than DL authorship attribution models. By utilizing source code embeddings derived from a one-shot verification system, our approach captures deeper stylistic and structural patterns, avoiding overfitting to dataset-specific artifacts. Unlike end-to-end DL classifiers, which require large labeled datasets for each author and often struggle with unseen authors (He et al., 2024), our model projects code into a structured embedding space where similar authorial styles naturally cluster. This design enables effective classification without the need for extensive retraining, enhancing adaptability to new authors and improving robustness against adversarial modifications (Abuhamad et al., 2023).

The last research question concerns the CPU and memory resources required to train the CLAVE authorship attribution system. Training the model with the entire dataset took 156 milliseconds and consumed 157.6 MB on the computer described in Section 4.5, with no GPU. These resource requirements are significantly lower than those of deep learning-based authorship attribution methods, which often demand several gigabytes of memory and much longer training times (**RQ4**). This efficiency highlights CLAVE's suitability for resource-constrained environments, enabling rapid model updates and deployment on standard computing hardware.

7 CONCLUSIONS

We show how a contrastive learning-based authorship verification model (CLAVE) can be leveraged for source code authorship attribution, achieving high classification performance with minimal training data, while requiring significantly fewer computational resources than DL-based approaches. The SVM model, trained with just six source files per programmer, achieves an accuracy of 0.923 for 85 programmers, outperforming state-of-the-art systems evaluated using the same methodology. Additionally, the SVM-based authorship attribution system offers a substantial advantage for classification over CLAVE's verification system, reducing classification error by 45.4%. Our system obtains higher performance with fewer training examples than the existing DL-based approaches for Python authorship attribution.

A key strength of our approach is its computational efficiency, which is significantly higher than that of deep learning-based methods. With a training time of only 156 milliseconds and memory consumption of 157.6 MB, it enables rapid retraining and deployment on standard computing hardware. These findings suggest that CLAVE's stylometric representations provide a scalable and efficient solution for source code authorship attribution, even in scenarios where only a few code samples are available.

Future work will focus on extending CLAVE-based authorship attribution to other programming languages and evaluating its performance on more diverse datasets to assess its generalizability across different coding styles. Additionally, we plan to analyze the model's robustness against adversarial attacks, such as code obfuscation or style imitation, and explore strategies to enhance its reliability for security and forensic applications.

All the source code used in this article, the dataset employed to train the models, the CLAVE source code and binary models, the performance results obtained during model evaluation, the best hyperparameters found for each model, and the CPU and memory measurement data are available for download at https://www.reflection.uniovi.es/bigcode/download/2025/icsoft2025.

ACKNOWLEDGEMENTS

This work has been funded by the Government of the Principality of Asturias, with support from the European Regional Development Fund (ERDF) under project IDE/2024/000751 (GRU-GIC-24-070). Additional funding was provided by the University of Oviedo through its support for official research groups (PAPI-24-GR-REFLECTION).

REFERENCES

Abuhamad, M., AbuHmed, T., Mohaisen, A., Nyang, D. (2018). Large-scale and language-oblivious code authorship identification. In Proceedings of the ACM SIGSAC Conference on Computer and Communications Security (CCS '18) (pp. 101–114).

Abuhamad, M., Jung, C., Mohaisen, D., Nyang, D. (2023). SHIELD: Thwarting code authorship attribution. *ArXiv*: 2304.13255

Alsulami, B., Dauber, E., Harang, R., Mancoridis, S., Greenstadt, R. (2017). Source code authorship attribution using long short-term memory-based networks. In *Computer Security* – ESORICS 2017 (Vol. 10492).

Álvarez-Fidalgo, D., Ortin, F. (2025). CLAVE: A deep learning model for source code authorship verification with contrastive learning and transformer encoders. *Information Processing & Management*, 62(3), 104005-104020.

Devlin, J., Chang, M.-W., Lee, K., Toutanova, K. (2019). BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies* (pp. 4171–4186). Minneapolis, MN.

Georges, A., Buytaert, D., Eeckhout, L. (2007). Statistically rigorous Java performance evaluation. *ACM SIGPLAN Notices*, 42(10), 57–76.

He, X., Lashkari, A. H., Vombatkere, N., Sharma, D. P. (2024). Authorship attribution methods, challenges, and future research directions: A comprehensive survey. *Information*, 15(3), 131.

Hendrycks, D., Gimpel, K. (2023). Gaussian error linear units (GELUs). *ArXiv*: 1606.08415v5.

Hozhabrierdi, P., Hitos, D. F., Mohan, C. K. (2020). Zero-shot source code author identification: A lexicon- and layout-independent approach. In *2020 International Joint Conference on Neural Networks* (IJCNN) (pp. 1–8).

Kurtukova, A., Romanov, A., Shelupanov, A. (2020). Source code authorship identification using deep neural networks. *Symmetry*, 12(12), 2044.

Li, Z., Chen, G. Q., Chen, C., Zou, Y., Xu, S. (2022). RoPGen: Towards robust code authorship attribution via automatic coding style transformation. In *Proceedings of the 44th International Conference on Software Engineering* (ICSE '22) (pp. 1906–1918).

Noma, H., Matsushima, Y., Ishii, R. (2021). Confidence interval for the AUC of SROC curve and some related methods using bootstrap for meta-analysis of diagnostic accuracy studies. *Communications in Statistics: Case Studies, Data Analysis and Applications*, 7, 344–358.

Ortin, F., Garcia, M. Perez-Schofield, B. G., Quiroga, J. (2022). The *StaDyn* programming language. *SoftwareX*, 20, 101211–101222.

Ortin, F., Facundo, G., Garcia, M. (2023). Analyzing syntactic constructs of Java programs with machine learning. *Expert Systems with Applications*, 215, 119398–119414.

Ortin, F., Álvarez-Fidalgo, D. (2025). Support webpage of the article Efficient source code authorship attribution using code stylometry embeddings. Retrieved from https://www.reflection.uniovi.es/bigcode/download/2025/icsoft2025.

Ou, W., Ding, S. H. H., Tian, Y., Song, L. (2023). SCS-GAN: Learning functionality-agnostic stylometric representations for source code authorship verification. *IEEE Transactions on Software Engineering*, 49, 1426–1442.

Rodola, G. (2025). Psutil: A cross-platform library for process and system monitoring in Python. Retrieved from https://psutil.readthedocs.io

Rodriguez-Prieto, O., Pato, A., Ortin, F. (2023) PLangRec: Deep-learning model to predict the programming language from a single line of code. *Future Generation Computer Systems*, 166, 107640–107655.

Stamatatos, E., Kredens, K., Pezik, P., Heini, A., Bevendorff, J., Stein, B., Potthast, M. (2023). Overview of the authorship verification task at PAN 2023. In *Working Notes of the Conference and Labs of the Evaluation Forum* (CLEF 2023) (pp. 2476–2491). Thessaloniki, Greece.

Kalgutkar, V., Kaur, R., Gonzalez, H., Stakhanova, N., Matyukhina, A. (2019). Code authorship attribution: Methods and challenges. *ACM Computing Surveys*, 52(1).

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, L. U., Polosukhin, I. (2017). Attention is all you need. In *Advances in Neural Information Processing Systems* (Vol. 30). Curran Associates, Inc.

Wang, N., Ji, S., Wang, T. (2018). Integration of static and dynamic code stylometry analysis for programmer de-anonymization. *In Proceedings of the 11th ACM Workshop on Artificial Intelligence and Security* (AISec '18) (pp. 74–84).

White, R., Sprague, N. (2021). Deep metric learning for code authorship attribution and verification. In *20th IEEE International Conference on Machine Learning and Applications* (ICMLA) (pp. 1089–1093).

Zhang, Z., Chen, C., Liu, B., Liao, C., Gong, Z., Yu, H., Li, J., Wang, R. (2024). Unifying the perspectives of NLP and software engineering: A survey on language models for code. *Transactions on Machine Learning Research* 9/2024.