Comparing Machine Learning and Deep Learning Models with Attention Mechanisms for Multiple Source Code Authorship Attribution

1st Bc. Richard Čerňanský
Faculty of Informatics and Information Technologies
Slovak University of Technology
Bratislava, Slovakia
richard.cernansky3@gmail.com

2nd Ing. Juraj Petrík
Faculty of Informatics and Information Technologies
Slovak University of Technology
Bratislava, Slovakia
juraj.petrik@stuba.sk

Abstract*—Authorship attribution (AA) in source code is crucial for plagiarism detection, forensic analysis, and software maintenance, especially when author information is missing or ambiguous. This field is not as well explored as the written communication AA and there are new models to be tested on different sized dataset. This study compares traditional machine learning models, such as Random Forests trained on TF-IDF tokenized representations, with deep learning models, including BERT and an Attention-based Neural Network (AttentionNN), for the task of multiple authorship attribution. We evaluate different representations of source code, including raw tokenized text and Abstract Syntax Tree (AST) structures, to determine their effectiveness in distinguishing authorship. The models are tested on various sized but relatively small datasets compared to the existing experiments, the largest consisting of 3,756 C language functions from Google Code Jam spanning 2009–2018, assessing their accuracy, precision, recall, and training efficiency across varying author set sizes. Our findings highlight the strengths and limitations of each approach, showing that while deep learning models, particularly BERT trained on raw source code, achieve the highest accuracy, Random Forests offer competitive results with significantly lower computational cost. The study also explores the impact of function complexity on model performance, revealing that AttentionNN benefits from structural complexity while struggling with short functions. The results emphasize the trade-offs between accuracy, computational efficiency, and data representation, providing insights into the optimal model selection for source code authorship attribution.

***Index Terms*—Authorship attribution, machine learning, deep learning, BERT, attention mechanisms, source code analysis**

I. Introduction

Authorship attribution in source code analysis is a crucial task with applications in plagiarism detection, forensic investigations, and intellectual property protection, but also software maintenance and software quality analysis [?], where the information about an author is missing, inaccurate or ambiguous. Given a function written in a programming language, determining its author requires capturing distinct stylistic patterns. Traditional machine learning (ML) methods, such as Random Forests trained on TF-IDF tokenized representations, have shown promise in text-based authorship identification. However, recent advancements in deep learning (DL) have introduced transformer-based models, such as BERT, which leverage contextual embeddings and self-attention mechanisms to extract more meaningful patterns from textual data.

The work reported here was supported by the Cultural and Educational Grant Agency of Slovak Republic (KEGA) under grant No. KG 014STU-4/2024.

Despite the success of deep learning models in natural language processing (NLP), their application to source code authorship attribution on different dataset sizes remains relatively unexplored. Unlike natural language, source code exhibits a more rigid syntax and structure, making its representation a key challenge. Several approaches have been proposed, ranging from direct tokenized representations of the raw source code to abstract syntax tree (AST) transformations that capture program logic. However, it remains unclear which type of representation is most effective for distinguishing authorship across different levels of dataset complexity.

In the first part we present important related work results in the field of source code authorship attribution. Then we continue with explaining methods and techniques that we used for developing and evaluating comparative framework of different models. Section 3 describes the models in more detail. The main part of this study systematically compares machine learning and deep learning models, including Random Forests, BERT trained on tokenized source code, BERT trained on AST representations, and an Attention-based Neural Network (AttentionNN) utilizing AST-derived node-to-node paths. The evaluation is conducted on a dataset of 3,756 C language functions from Google Code Jam (GCJ) spanning 2009 to 2018 petrik. Most of the works in the field predict the authors based on files consisting of multiple language constructs, our research however focuses on prediction of author given single function. The models are assessed based on their accuracy, precision, recall, and training efficiency across different author set sizes. Through this comparison, we aim to determine which model best balances accuracy and computational efficiency, as well as the effectiveness of different function representations for authorship attribution.

II. Related Work

A number of studies have explored source code authorship attribution using varying data representations and model architectures. Early work focused on using statistical models trained on token frequency vectors or stylometric features. More recent works, displayed on the table **??** show that traditional Machine Learning (ML) models like Random Forest (RF), Support Vector Machine (SVM) or Neural Network (NN) can provide reliable results on different sized datasets. Specifically, Caliskan et al. caliskan used TF-IDF with RF on Java code and demonstrated strong attribution capabilities using features extracted from abstract syntax trees and code formatting. However, there is no common agreement among researchers for selecting the best machine learning classifier and data features for source code AA.

In recent years, the development of transformer-based models like Bidirectional Encoder Transformer (BERT) has significantly impacted Natural Language Processing (NLP), particularly in areas such as Natural language understanding and generation (NLG). However, their potential in the field of authorship attribution, especially for source code, remains relatively underexplored. In some works, they were compared to the state-of-the-art models, showing promising results fi. We are particularly interested in pretrained models such as CodeBERT and GraphCodeBERT codebert, to source code. These models were initially developed for general code understanding tasks (e.g., function naming, defect detection) but have shown promising results in Authorship Attribution task when fine-tuned on labeled datasets. Choi et. al. choi shows that Large Language Models (LLM) can be successfully utilized in code authorship attribution tasks. However, their ultimate utility in real-world deployment will struggle with costs, sophisticated prompt engineering, model selection, hyperparameter choices, and an understanding of the models' sensitivity to context.

Alon et al. introduced code2vec code2vec, an attention-based neural network that learns embeddings from paths within Abstract Syntax Tree (AST). This architecture enables learning from structural representations of code while remaining lightweight compared to transformer models. Attention Neural Network (ANN), a similar concept, leverages node-to-node ANN paths to predict function authorship. Bogomolov et. al bogomolov also examined path-based models on JAVA dataset consisting of 200 000 methods and reached accuracies around 98% with Path-based Neural Network and Path-based Random Forests.

However, much of the existing work focuses on large datasets or file-level attribution. For example, studies using GitHub datasets typically assume access to full repositories or complete files. In contrast, authorship attribution at the function level, especially on small datasets such as our collection of GCJ solutions, remains unexplored. Moreover, another issue lies in the absence of standardized benchmark datasets for the task, as the current research relies on datasets with different properties in terms of size, author set size, source code background and other.

Moreover, little attention has been paid to the impact of function complexity on model performance. Most prior evaluations rely on global accuracy metrics without analyzing how model effectiveness varies across functions of differing structural or semantic richness.

III. Methods

A. Source Code Representation

In the paper information, authors present division of features based on the method they use for extraction:

- **Stylometric features:** They express the structural characters that represent the author's preference for different statements, keywords, nesting sizes, etc.
- **N-gram features:** N-gram is a contiguous sequence of an n byte, character, or words extracted from the source. A sliding that has a fixed length generates the sequence.
- **Graph based features:** These capture underlying traits in the code such as deeply nesting code, control flow and data flow. It includes : AST, Program Dependency Graph (PDG), Control Flow Graph (CFG), RFG.
- **Behavioral features:** This type completely different from the previous as it is not looking at the source code itself but rather on the way that the program of the source code communicates with the computer. In this category there are dynamic features which collect runtime measurements as CPU and memory usage. Instruction features that are retrieved from binaries and reflect code properties like registers or immediate operands.
- **Neural Networks Generated Representations of Source Code or Features:** Lastly, in recent years, neural networks have been increasingly utilized to generate representations (such as embeddings) of source code. This approach is also applied in this thesis, using AST analysis to provide input for the neural network.

B. Abstract Syntax Tree (AST)

Based on the full definition sun$_a stAbstractsyntaxtree(AST)isatree - basedrepresentationofsourcecode, wherenodesrepresentvariouselem$ $leaforleafnodes.Internalnodesdefinetheprogram'sconstructsorope$

In the figure **??** is an example of a simple C function and its corresponding Abstract Syntax Tree (AST):

1) AST - types of features: Features divide into 4 main kinds according to the nature of the information extracted from the Abstract Syntax Tree (AST):

- **Structural** Structural features capture the structural complexity of AST. These are often some numerical quantitative values like depth of the graph, number of nodes, or average branching factor of internal nodes.
- **Semantic**
  Semantic features reflect the semantic information encoded in AST. It could be as simple as the distribution of node kinds (that are defined by the compiler that creates the AST) or the distribution of distinct root-to-leaf paths.

Caliskan et al. [35]	2015	250	9	2250	68-83	70	120000	lay,lex,synt	C++	GCJ	RF	98.04%
Caliskan et al. [19]	2015	1600	9	14400	68-83	70	-	lay,lex,synt	C++	GCJ	RF	92.83%
Wisse et al. [36]	2015	34	-	-	-	-	1689	lay,lex,synt	JS	GitHub	SVM	85%
Yang et al. [37]	2017	40	11-712	3022	16-11418	98.63	19	lay,lex,synt	Java	GitHub	PSOBP	91.1%
Alsulami et al. [20]	2017	10	20	200	-	-	53	synt	C++	GitHub	NN	85%
Alsulami et al. [20]	2017	70	10	700	-	-	130	synt	Python	GCJ	NN	88.86%
Gull et al. [22]	2017	9	-	153	100-12000	-	24	lay, lex	Java	PlanetSourceCode	NaiveBayes	75%
Zhang et al. [23]	2017	53	-	502	-	-	6043	lay,lex	Java	PlanetSourceCode	SVM	83.47%
Dauber et al. [27]	2017	106	150	15900	1-554	4.9	451368	lay,lex, synt	C++	GitHub	RF	70%
McKnight et al. [26]	2018	525	2-11	1261	27-3791*	336*	265	lay,lex, synt	C++	GitHub	RF	66.76%
Simko et al. [21]	2018	50	7-61*	805	10-297*	74*	-	lay,lex, synt	C	GCJ	RF	84.5%*
Abuhamad et al. [38]	2018	8903	7	62321	-	71.53	-	lex	LO	GCJ	RNN, RF	92.30%
Ullah et al. [39]	2019	1000	-	-	-	-	-	lex, CFG	LO	GCJ	NN	99%
Zafar et al.[40]	2020	20,458	5-7-9	-	-	-	-	lex	C++,Java Python	GCJ	CNN	84.94%

-No data.
**Data obtained from personal communication.*
^*LO: language-oblivious approach.*

TABLE I: Table summarizing state-of-the-art works in Source code Authorship Attribution (AA) with machine learning approach datasets.

- **Syntactic**
  Syntactic features describe the paradigm in which the source code is written and the logical complexity of the program. These might include the number of functions or functor structures, or the number of control flow units.
- **Combined**
  Combined features are use-case specific, and it is up to the data analyst to design the best fit solution for information extraction using the combinations and alternations of the aforementioned strategies. Combined features are also the ones that are usually used when tackling real-world problems like source code authorship attribution or function name classification.

Listing 1: Simple C function for addition

```
        int add(int a, int b) {
            return a + b;
        }
```

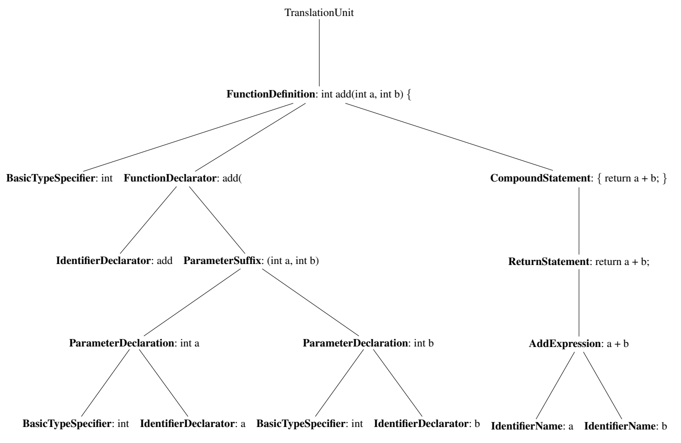

TranslationUnit
FunctionDefinition: int add(int a, int b) {
BasicTypeSpecifier: int
FunctionDeclarator: add(
CompoundStatement: { return a + b; }
IdentifierDeclarator: add
ParameterSuffix: (int a, int b)
ReturnStatement: return a + b;
ParameterDeclaration: int a
ParameterDeclaration: int b
AddExpression: a + b
BasicTypeSpecifier: int
IdentifierDeclarator: a
BasicTypeSpecifier: int
IdentifierDeclarator: b
IdentifierName: a
IdentifierName: b

Fig. 1: (a) C function for addition; (b) its Abstract Syntax Tree.

Left panel: numbered C code listing "int add(int a, int b) return a + b; ". Right panel: AST tree with nodes for TranslationUnit, FunctionDefinition, BasicTypeSpecifier "int", FunctionDeclarator "add(. . .)", CompoundStatement, ReturnStatement, AddExpression, and IdentifierName leaves "a" and "b".

2) Halstead complexity measures: From Figure **??**, we observe that functions vary both in their length and the same applies also to the structural complexity of their ASTs. Therefore, we decided to evaluate the models' performance in terms of function complexity. There are various measures of complexity and for our purposes, we selected two: the simple length in tokens and a measure called Halstead complexity, which quantifies Halstead Difficulty, Volume, and Effort halstead. To aggregate them into single value, we used weighted sum of the three metrics weighting them in ratios 4:3:3 respectively. We chose these ratios because it reflects both the theoretical importance and the empirical scale of the three metrics. Halstead Difficulty is most directly tied to cognitive complexity, so we give it a slightly larger share (40 %), while Volume (30 %) and Effort (30 %) each make substantial but secondary contributions. Before applying these weights we normalize each metric to the same range, so that the weight split yields a balanced composite score—capturing "how hard" (Difficulty), "how big" (Volume), and "how much work" (Effort) in one single complexity value. We did not incorporate Halstead length because we did separate analysis on length.

C. Our Data Representation

For training the models, we constructed our prediction task data pipeline like this:

We utilized various types of representations of single function's raw source code to provide the models with data. These include word tokenized TF-IDF vectors for Random Forests, BPE of raw source code for BERT, linearized AST pre-order traversal for BERT and representation set of randomly selected 600 node-to-node paths for AttentionNN. PsycheC project's C language frontend parser psychec was used for parsing the code into ASTs. Each model was trained in Tensorflow.Keras framework and with different author set sizes to evaluate scalability. The different set sizes were 110, 27, 11, 3 with each having increased demands on minimal function count than the one before. The minimal function counts per author for the sizes were 25, 40, 50, 58 respectively. Table **??** summarizes the basic properties of our datasets.

D. Data preprocessing & Exploratory data analysis

The dataset consisted of thousands of solutions from hundreds of authors. From each solution we extracted the function and its author. The largest of the different sized datasets we used consisted of 3 756 C language functions' source code gathered from Google Code Jam (GCJ) solutions from years 2009 to 2018. The data was preprocessed exclude duplicates, remove outlying authors that have multiple times more samples than is the average number of samples per author to reduce the models' bias towards these names. Functions that could not be parsed into an AST were also eliminated to ensure consistency in the datasets. This allows for a fair assessment of each model, as they utilize different representations of function data. For further analysis, we needed to see the frequency distributions of authors.

From the figure **??** we can see that the most of our dataset's functions lengths lie between 10 and 50 tokens which makes our goal of authorship attribution of single function harder than if given a whole file consisting of multiple functions or code blocks in sense that for the prediction most of the time we only have from 10 to 50 tokens available to find the correct author. After having all of the functions' source codes ready in the suitable form for the model to train we performed the training and evaluation on 4 different model setups that are presented in next section. The high-level overview of the pipeline is illustrated in the figure **??**.

IV. Models setup & training

In this section we will discuss the training strategies and setup for the examined models. Random Forests and AttentionNN models were trained using Stratified Cross validation strategy that ensures, that each author is evenly distributed across the folds. Number of folds was set to 5 and 3 folds were used for training, 1 for validation and 1 for testing. As training of BERT takes a lot more time than the other 2, we just repeated the training 3 times and observed variance in evaluation metrics lower than 1%.

TABLE II: Basic properties of our various datasets.

Author set size	No. of functions	Min. function count / author
110	3756	25
27	1357	40
11	638	50
3	212	58

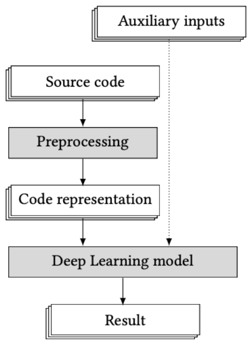
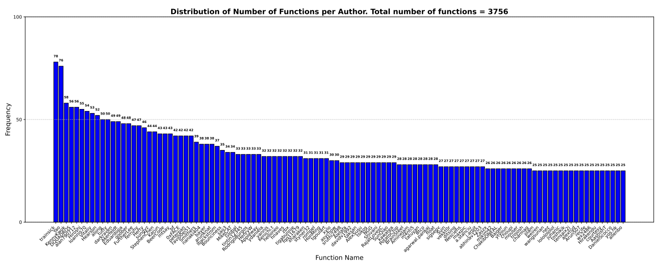
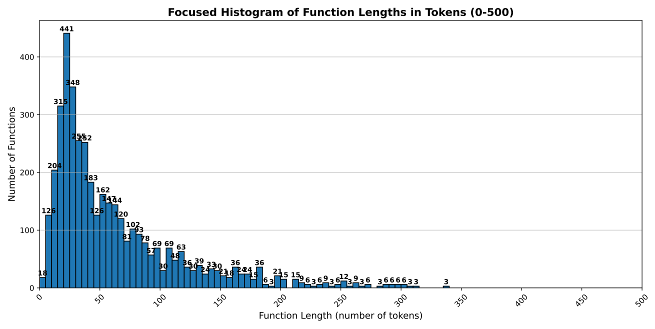


Fig. 2: compiler$_{b}asedPipelineoverviewforsource-codeauthorshipclassification.$

Flowchart showing raw functions parsed into tokens and ASTs, features extracted (TF-IDF, AST paths, BERT embeddings), and three classifiers (Random Forest, BERT, AttentionNN) producing author predictions.



(a) Functions per author.



(b) Function length in tokens.

Fig. 3: Frequency distributions of functions per author and of function lengths.

Two histograms: left shows the long-tail distribution of functions across authors; right shows most functions fall between 10 and 50 tokens.

A. Random Forests

For comparison purposes, we have chosen Random Forests as standard approach to text based authorship attribution. They were trained on word-level tokens using the regex pattern `r"\b\w+\b"`, which matches words separated by word boundary characters. First we used Grid search to optimize these hyperparameters:

- **n_estimators**: {50, 100, 200, 500}
- **max_depth**: {10, 20, 30, None}
- **min_samples_split**: {2, 5, 10}
- **min_samples_leaf**: {1, 2, 4}
- **max_features**: {"sqrt", "log2", None}
- **bootstrap**: {True, False}

and then we trained the model using the best hyperparameters found. Since each dataset was tuned separately via grid search, the optimal hyperparameters vary and are not listed here.

B. Bidirectional Encoder Representations from Transformers - BERT

Transformer-based models such as BERT leverage the self-attention mechanism to compute contextual embeddings for each code token by weighting interactions across the entire sequence, thus capturing both local syntactic patterns and long-range semantic dependencies. Pre-trained on massive corpora via masked language modeling and next-sentence prediction, BERT encodes code snippets into rich representations summarized by the special `[CLS]` token. By fine-tuning this encoder with a lightweight classification head on labeled code–author pairs, the model learns author-specific stylistic cues like naming conventions, formatting habits, and preferred idioms.

For both BERT training setups we used Microsoft's pre-trained graphcodebert-base codebert that already understands general programming concepts like control structures, loops or functions and is able to capture differences between authors faster. We also used the graphcodebert tokenizer which uses Byte pair encoding that merges frequently occurring character sequences into tokens, which is appropriate for data like source code of programming language that has very rigid syntax. The comments in all code samples were filtered out so the attribution task is 'fair' across the models that utilize ASTs.

As mentioned, we have trained the BERT model on two distinct data representations. Tokenized RAW source code and tokenized AST pre-order traversal to see how the model reacts to these representations and how useful it may be to learn from the AST structural patterns in comparison to AttentionNN that utilizes node-to-node paths.

C. Attention Neural Network

To capture all three of the structural, semantic and syntactic features in a number vector that could be somehow fed into a neural network classifier. Code2vec code2vec proposes solution design that extracts so called 'path-contexts' which are encoded into vector space using the embedding values of its components. Path-contexts capture the node-to-node path in concise manner as it consist of triplets of (start node, the path of nodes kinds, terminal node). Each distinct node is encoded with the data value it carries and each distinct path is encoded as a string of nodes' kinds in between the start and terminal node. The model then learns to distinguish the authors utilizing the attention mechanism, giving larger weights to contexts that help to reveal the author the most.

D. Evaluation methods

For evaluation purposes metrics such as validation Accuracy, Precision, Recall, Time to train model and Halstead complexity measures were captured to explain the models behavior with different kinds of data. The deep-learning models were trained with the Categorical Cross-Entropy loss function guiding the optimization process,

$$\mathcal{L}_i = -\sum_{j=1}^{C} y_{i,j} \log(\hat{y}_{i,j}) \quad (1)$$

where:

- $\mathcal{L}_i$ is the loss for the i-th sample.
- C is the number of possible classes.
- $y_{i,j}$ represents the ground truth label for class j. It is a one-hot encoded vector, meaning:
  - $y_{i,j} = 1$ if the sample belongs to class j.
  - $y_{i,j} = 0$ otherwise.
- $\hat{y}_{i,j}$ is the predicted probability for class j, obtained from a softmax function.
- $\log(\hat{y}_{i,j})$ takes the logarithm of the predicted probability.

We also looked at the learning curves of the deep leaning models (BERT, AttentionNN) to assess the overfit of each one and compared the models in terms of accuracy for functions of different complexity obtaining interesting results revealing their strengths and weaknesses.

V. Results

As we have mentioned, the models were mainly tested for their scalability with having more functions per author while decreasing the distinct author set size. Table **??** summarizes how each model performed in terms of accuracy and the time required for training (until the validation loss stopped decreasing) across diverse datasets and figure **??** visualizes the achieved performances on Accuracy, Precision and Recall. From table **??** we can say that BERT trained on tokens extracted from the raw source code performed the best at each of the author set sizes, especially with accuracy over 90% on the smallest set, however with the longest required time to train the model, more than 9.5 times longer than it took to train AttentionNN to its full potential and 36.4 times longer than the time required for Random Forests which scored just 4.8 percent less. Our RF experiment on dataset with 110 reached results similar to this study rf1, where they had 525 authors, but with 2-11 files per author and also included information from comments which we filtered out. In rf2, authors reached accuracy of 84.5% on 50 authors with and average of 42.8 files each similar to our setup for 27 authors. They scored 14% more than we did, using the the features presented in caliskan.

What also interested us is that from `set_size=11` to `set_size=3`, BERT improved only by 0.86% which is the smallest improvement across all the experiments and indicates that increasing the minimum function count per author from 50 to 58 had little impact on the model's ability to distinguish authors. On the other hand, from being the worst, BERT fed with AST pre-order traversal scored a 40% higher under these conditions. Unfortunately, we were unable to find any relevant sources against which to compare our BERT results. Nevertheless the following experiments conducted on other models, helped us understand advantages and disadvantages of BERT model for authorship attribution.

The confusion matrix shows **??**, that the model performed best on 2 most represented authors (see **??**). Although if we take 8 most represented authors, only 5 of them are also the most accurately classified. Additionally, BERT and other models were not biased towards the most represented authors when tested on the less represented authors. The only type of bias we have seen was on author that was not in the most

TABLE III: Comparison of models across author attribution accuracy and training time for different author set sizes.

Model Performance on Author Set Sizes				
Model	**110**	**27**	**11**	**3**
Random Forests (TF-IDF)	66.24% / 3.63 min	70.60% / 0.23 min	80.56% / 0.08 min	85.84% / 0.05 min
BERT (source code)	72.47% / 30.33 min	82.72% / 9.49 min	89.84% / 5.30 min	**90.70% / 1.82 min**
BERT (AST traversal)	24.07% / 33.53 min	35.66% / 10.30 min	40.62% / 6.20 min	83.72% / 1.22 min
AttentionNN (AST paths)	36.93% / 3.53 min	53.42% / 1.19 min	60.82% / 0.54 min	69.30% / 0.19 min

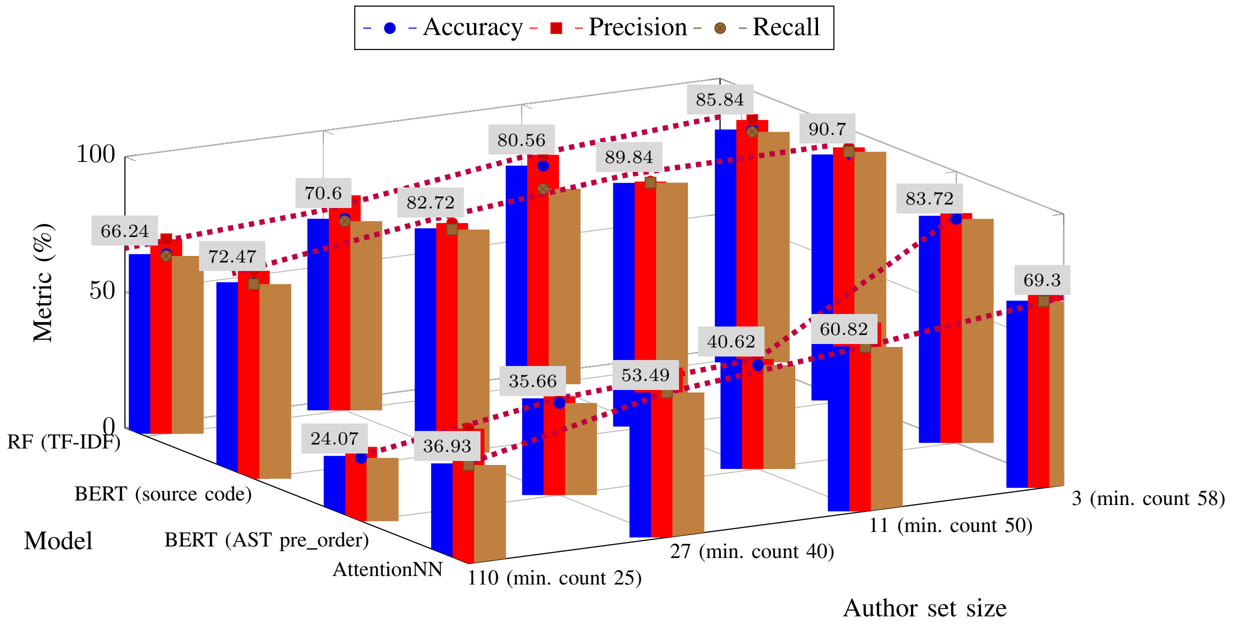


Fig. 4: Accuracy, precision, and recall across models and author set sizes.
3D bar plot where the x-axis lists models (RF, BERT raw, BERT AST, AttentionNN), the y-axis shows author set sizes (110, 27, 11, 3), and the z-axis gives metric values; bars are blue for accuracy, red for precision, and brown for recall.

represented group. Therefore, we conclude that the bias arises not from class imbalance but from the characteristics of each author's functions (e.g., their purpose and functionality).

In terms of overfit, BERT with tokenized source codes was able to generalize the datasets in effective manner reaching 1.41, 1.24, 1.05, 1.12 for overfit ratios calculated like this

$$\text{Overfit Ratio} = \frac{\text{Train Accuracy}}{\text{Test Accuracy}} \tag{2}$$

and only the latter two are under the limit of 1.2 - being optimally fitted.

Random forests reached 1.51, 1.3, 1.24, 1.15 overfit ratios which still show a considerable amount of generalization. However AttentionNN was not that successful with generalizing and with the dataset consisting of 3 distinct authors it got overfit of 1.4 and only increasing with each larger dataset, meaning insufficient data is being provided for the network so it starts memorizing particular patterns instead of actually generalizing above authors, which does not allow the model to perform better on unseen data affecting its validation accuracy.

A. Comparing models' performance depending on the complexity of function

In our task of identifying an author given a single function, a natural question arises: how did the models perform on functions of different complexities? The basic indicator of function complexity is its length. So we came up with strategy, how to visually compare models, even when the number of each model's misclassifications is different. We scaled the models histogram bin values by an index that is obtained like this:

$$\text{index}_i = \frac{N_\text{w}}{N_i} \tag{3}$$

where:

- N_w is the number of misclassifications of the worst-performing model.
- N_i is the number of misclassifications made by the model i.
- index_i is the scaling factor for the i-th model.

This way we obtain visual representation that objectively accounts for the fact that the number of errors is different. Figure **??** shows visualized frequencies of misclassified function

lengths in tokens for Random Forests, BERT and Attention]
scaling each model with its index.

From the figure **??** the only statement we can say is t
the model AttentionNN particularly struggled with functi
of length lower than 40 tokens but did quite well on the lon
ones. This will be discussed in the following subsection m
deeply **??**. Other models performance is not really distingui
able from this visualization, so we performed the same in
strategy with the Halstead complexity metrics in figure
Measuring each misclassified function's Halstead comple
and plotting it for Random Forests and BERT trained
tokenized source code, we obtained following figure.

This error distribution reveals that Random Forests struggled more with Halstead complexity increasing from the marked red point compared to the BERT Specifically there are 1.59 times more misclassifications made by the Random Forest classifier than there are by BERT measured from the red point. However, closer we look to 0 on x axis, the more errors we see made by BERT. This would be the main difference between the deep learning and machine learning method. The deep learning method utilizes self-attention that enables it to capture more complex relationships but struggles to categorize when limited or not complex enough data is provided, not being able to find the relationships that characterize the author of the function. On the other hand, the machine learning TF-IDF method struggles with the more complex data because the representation lacks context understanding and solely relies on the number of occurrences in the sample.

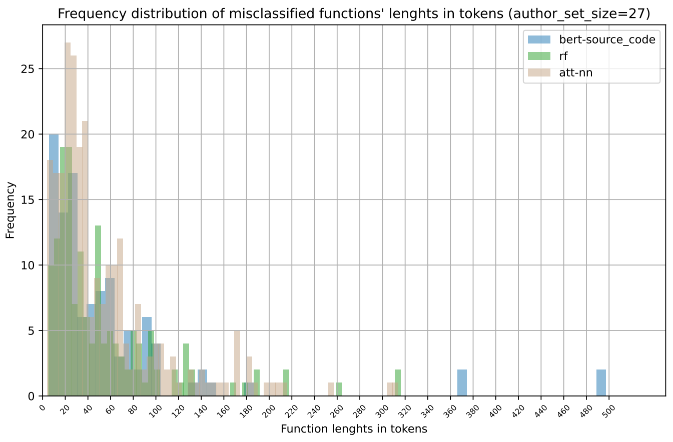


Fig. 5: Misclassification counts by function length for each model.

Overlaid line plots (blue=RF, red=BERT, brown=AttentionNN) showing number of misclassified functions binned by token length.

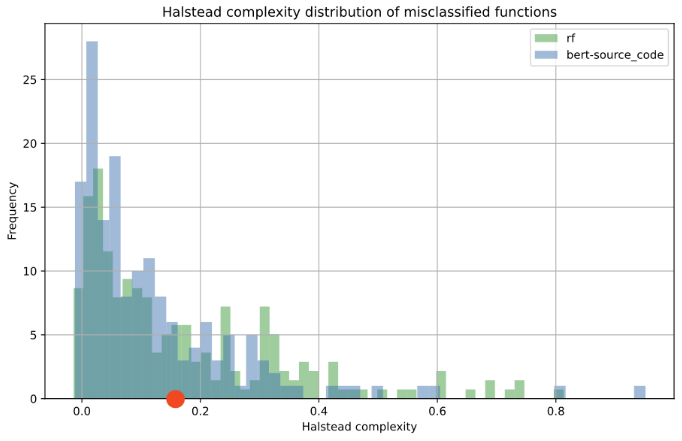


Fig. 6: Misclassification counts by Halstead complexity for RF and BERT.

Overlaid histograms (blue=RF, red=BERT) of misclassified functions versus composite Halstead complexity scores.

B. *Attention neural network - complexity over quantity*

This model showed relatively promising results, indicating increasing dataset size and providing more complex data enhances its performance. Paper pbnn presents similar model reaching 0.979% accuracy with 40 authors but 3,021 files for each in programming language Java.

Considering the observation in figure **??**, where AttentionNN performed worst on lengths between 20 and 40 tokens, we assessed the model in a deeper manner. Evaluating accuracies across functions distributed into bins according to 2 metrics. Number of nodes in the AST and depth of the AST of the function.

From the figure **??** we can deduce, there is an emerging trend in accuracy from bins of lower value to the bins of higher value. Even though the frequency distribution of both variables indicates more frequency on the lower values we observe a lower accuracy. This tells that the model is learning more about the author from structural complex functions than simple non-nesting and control-flow divergent ones.

VI. Discussion

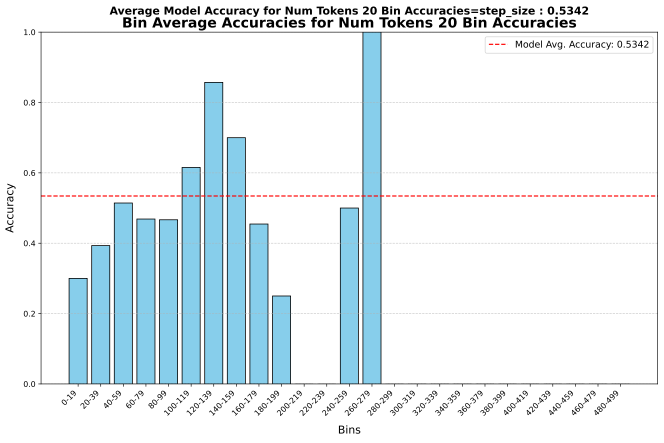
Our findings show that deep learning models, especially BERT, perform better than traditional machine learning methods for authorship attribution of single functions. But this comes at a cost—BERT needs way more computation and training time. On the other hand, Random Forest with TF-IDF tokenization held up surprisingly well, especially with larger author sets, and trained very fast.

A key takeaway is that BERT trained on raw source code tokens consistently had the best accuracy across all dataset sizes. However, its improvement from `set_size=11` to `set_size=3` was only 0.86%, meaning increasing the minimum function count per author after a certain point doesn't do much. Interestingly, BERT using AST pre-order traversal struggled with large author sets but got way better when we reduced the dataset to three authors. This shows how important it is to choose the right input representation for deep learning models.

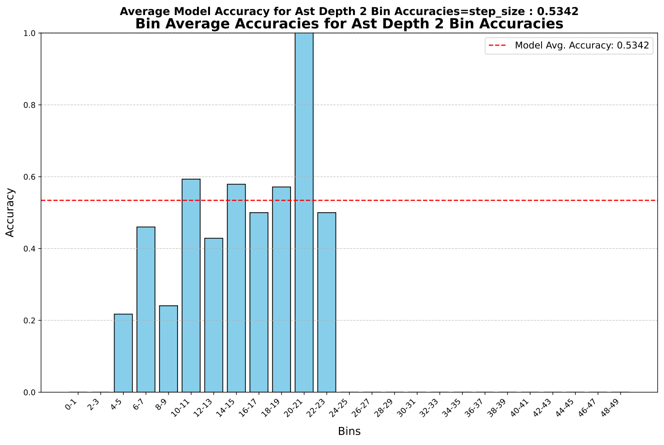
Looking at misclassifications, each model had its weak points. BERT had trouble with short functions that had low Halstead complexity— they do not provide enough meaningful context to determine the author. Random Forests, which rely more on statistical patterns than contextual understanding, started failing as function complexity increased. This makes sense since TF-IDF representations capture token frequency but do not really understand code structure.

AttentionNN, which works with AST-derived node-to-node paths, preferred structurally complex functions over simple ones. This means that networks using structural information may do better with more complex functions rather than short, flat ones. However, AttentionNN did not generalize well when data was limited, and it started overfitting when trained on small author sets. This shows we either need bigger datasets or more regularization to prevent overfitting.

The results highlight a clear trade-off between model complexity, training time, and accuracy. Deep learning models, while powerful, require serious computational resources, making them harder to scale in real-world use cases. Random Forests might not be as accurate, but they are super efficient and a solid alternative when speed is a concern.



(a) Accuracy vs. number of AST nodes



(b) Accuracy vs. AST depth

Fig. 7: AttentionNN accuracy as a function of AST complexity. Two line plots: (a) accuracy versus number of AST nodes, showing higher accuracy on larger trees; (b) accuracy versus AST depth, showing improvement on deeper structures.

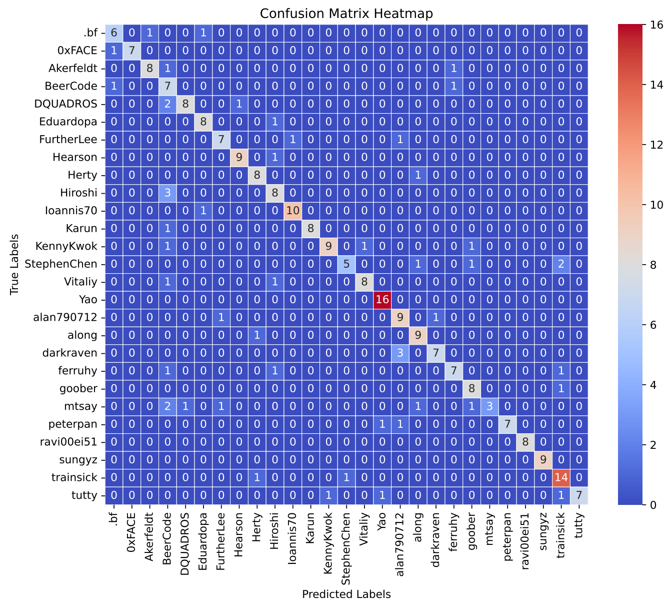


Fig. 8: Confusion matrix for BERT trained on raw source code (27 authors).
27×27 heatmap with true authors on rows and predicted on columns; dark diagonal cells show high correct predictions, especially for the two most frequent authors, lighter off-diagonals indicate errors.

Going forward, it would be interesting to combine these methods. For example a hybrid approach using BERT embeddings with Random Forest classifier could balance accuracy and efficiency. Also, using graph-based neural networks to process ASTs more effectively might push authorship attribution even further dependence$_{g}raphs.Exploringdataaugmentationtechniquesorbetterre$
$basedinputs.$

VII. Conclusion

In this study, we compared traditional machine learning models and deep learning models for authorship attribution of individual functions. Our results showed that while BERT-based models provided the highest accuracy, they required significantly longer training times and computational resources. In contrast, Random Forests trained on TF-IDF representations performed surprisingly well given their simplicity and efficiency.

The choice of model largely depends on the trade-off between accuracy and efficiency. If the goal is purely high accuracy and computational resources aren't a constraint, then BERT trained on tokenized source code is the best option. However, if speed and efficiency are more important, Random Forests provide a strong alternative with competitive accuracy.

We also found that different function representations significantly affect model performance. BERT struggled with AST-based pre-order traversal when the number of authors was high.

AttentionNN performed better on more structurally complex functions rather than shorter ones. This highlights the need for careful feature selection capturing the complex relationships when applying deep learning to authorship attribution.

Overall, this study demonstrates that while deep learning methods achieve state-of-the-art performance, traditional machine learning approaches still hold value, especially in resource-constrained environments.

The work reported here was supported by the Cultural and Educational Grant Agency of Slovak Republic (KEGA) under grant No. KG 014STU-4/2024.