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

Richard Čerňanský
Department Name
Organization
City, Country
email@example.com

Ing. Juraj Petrík
Department Name
Organization
City, Country
email@example.com

***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 written communication AA, and there are new models to be tested on different sized datasets. 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 a dataset of 3,004 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.**

I. Introduction

Authorship attribution in source code analysis is a crucial task with applications in plagiarism detection, forensic investigations, and intellectual property protection, as well as software maintenance and quality analysis when author information 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.

Despite the success of deep learning models in natural language processing (NLP), their application to source code authorship attribution across 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—from direct tokenized representations of raw source code to abstract syntax tree (AST) transformations that capture program logic—but it remains unclear which representation is most effective for distinguishing authorship across various levels of dataset complexity.

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 3,004 C language functions from Google Code Jam (GCJ) spanning 2009 to 2018 [**?**]. 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. Methods

A. Source Code Representation

In [**?**], the authors present a division of features based on the extraction method:

- **Stylometric features:** Capture structural characteristics that represent an author's preference for various statements, keywords, and nesting sizes.
- **N-gram features:** Extract contiguous sequences of bytes, characters, or words using a fixed-length sliding window.

Identify applicable funding agency here. If none, delete this.

Listing 1: Simple C function for addition

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

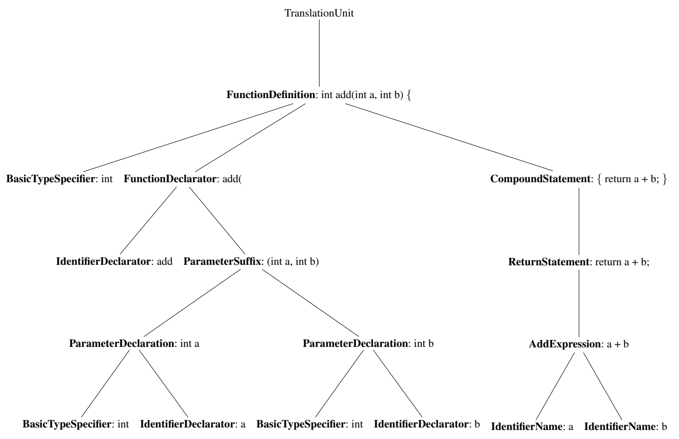

TranslationUnit
FunctionDefinition: int add(int a, int b) {
BasicTypeSpecifier: int
FunctionDeclarator: add(
CompoundStatement: { return a + b; }
IdentifierDeclarator: add
ParameterSuffix: (int a, int b)
ReturnStatement: return a + b;
ParameterDeclaration: int a
ParameterDeclaration: int b
AddExpression: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

Fig. 1: (a) C function code listing and (b) corresponding AST diagram.

- **Graph-based features:** Capture underlying traits in the code such as deep nesting, control flow, and data flow. This includes representations such as the AST, Program Dependency Graph (PDG), Control Flow Graph (CFG), and RFG.
- **Behavioral features:** Focus on the program's runtime behavior (e.g., CPU and memory usage) and instruction-level features from binaries.
- **Neural Network Generated Representations:** Recently, neural networks have been used to generate embeddings of source code. This approach is applied in this study by using AST analysis to provide input for the neural network.

B. Abstract Syntax Tree (AST)

Based on the definition in [**?**], an Abstract Syntax Tree (AST) is a tree-based representation of source code where nodes represent elements of the code and edges represent the relationships between these elements. The AST is an essential component in the compilation process, serving as an intermediate representation used for optimization and machine code generation. Nodes in the AST can be either internal (non-leaf), which define the program's constructs, or leaf nodes, which store actual textual values.

Below is an example of a simple C function and its corresponding AST:

1) AST — Types of Features: Features extracted from the AST fall into four main categories:

- **Structural:** Capture the AST's complexity (e.g., depth, number of nodes, average branching factor).
- **Semantic:** Reflect semantic information such as the distribution of node types or distinct root-to-leaf paths.

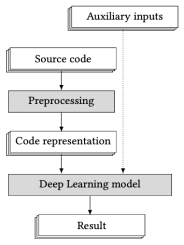


Fig. 2: [**?**] High-level overview of the function classification prediction task.

- **Syntactic:** Describe the programming paradigm and logical complexity (e.g., number of functions, control flow units).
- **Combined:** Use-case specific features that combine the aforementioned strategies for tasks like authorship attribution or function name classification.

2) Halstead Complexity Measures: To evaluate function complexity, we used Halstead complexity measures (Difficulty, Volume, Effort) as defined in [**?**]. These measures were aggregated into a single value using a weighted sum with ratios 4:3:3. (Halstead length was not incorporated, as it was analyzed separately.)

C. Our Data Representation

For training the models, a prediction task data pipeline was constructed as follows:

Various representations of each function's raw source code were utilized:

- Word-tokenized TF-IDF vectors for Random Forests.
- Byte Pair Encoding (BPE) of raw source code for BERT.
- Linearized AST pre-order traversal for BERT.
- A set of 600 randomly selected node-to-node paths for AttentionNN.

The PsycheC project's C language frontend parser [**?**] was used to convert code into ASTs. Models were trained in the TensorFlow.Keras framework using different author set sizes (110, 27, 11, and 3) with minimum function counts per author of 25, 40, 50, and 58 respectively.

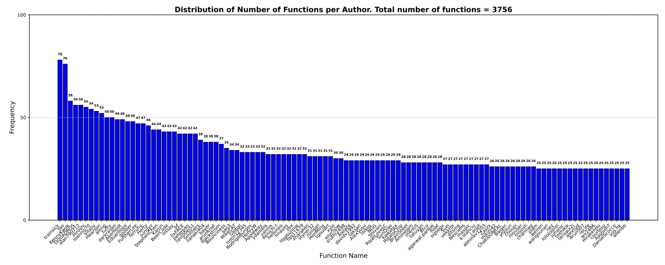
D. Data Preprocessing & Exploratory Data Analysis

The dataset consisted of thousands of solutions from hundreds of authors. From each solution, the function and its author were extracted. The largest dataset comprised 3 004 C language functions from GCJ solutions (2009–2018). Preprocessing involved duplicate removal, elimination of authors with disproportionately many samples, and exclusion of functions that could not be parsed into an AST. Frequency distributions of authors and function lengths were analyzed for further insights.

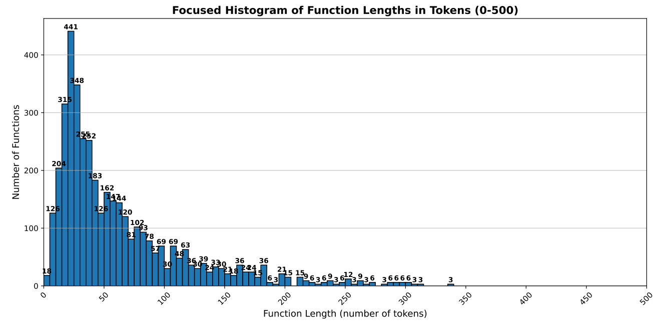
Figure 3a shows that most function lengths are between 10 and 50 tokens, making single-function authorship attribution challenging when only short code snippets are available.

TABLE I: 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



(a) Frequency distribution of author names



(b) Frequency distribution of function lengths

Fig. 3: Important frequency distributions of our dataset.

III. Models Setup & Training

Random Forests and AttentionNN models were trained using a stratified cross-validation strategy (5 folds: 3 for training, 1 for validation, and 1 for testing). Because BERT training requires considerably more time, it was repeated three times, with evaluation metric variance below 1%.

A. Random Forests

Random Forests were chosen as a baseline for text-based authorship attribution. They were trained on word-level tokens (using the regex `r"\b\w+\b"`). A grid search was performed to optimize 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}

The best hyperparameters were then used to train the final model.

B. BERT

For both BERT setups, Microsoft's pretrained `graphcodebert-base` [**?**] was used. The graphcodebert tokenizer uses Byte Pair Encoding, which is suitable for programming languages with rigid syntax. Comments were removed to ensure fairness when comparing models that utilize ASTs. BERT was trained on two distinct representations: tokenized raw source code and tokenized AST pre-order traversal.

C. Attention Neural Network

The AttentionNN model leverages AST-derived node-to-node paths (path-contexts) as proposed in [**?**]. Each path is represented as a triplet (start node, path of node kinds, terminal node) and encoded into vector space. The model then learns to weigh these contexts via an attention mechanism to distinguish authors.

D. Evaluation Methods

Evaluation metrics included validation accuracy, precision, recall, training time, and Halstead complexity measures. Deep learning models were trained with the categorical cross-entropy loss function:

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

where:

- $\mathcal{L}_i$ is the loss for sample i.
- C is the number of classes.
- $y_{i,j}$ is the one-hot encoded ground truth for class j.
- $\hat{y}_{i,j}$ is the predicted probability for class j (obtained from a softmax function).

Learning curves were analyzed to assess overfitting. The overfit ratio was calculated as:

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

IV. Results

The models were evaluated for scalability as the number of functions per author increased while the distinct author set size decreased. Table I summarizes model performance (accuracy and training time) for various author set sizes.

Figure 7 shows the confusion matrix for the best performing model (BERT on raw source code) for an author set size of 27.

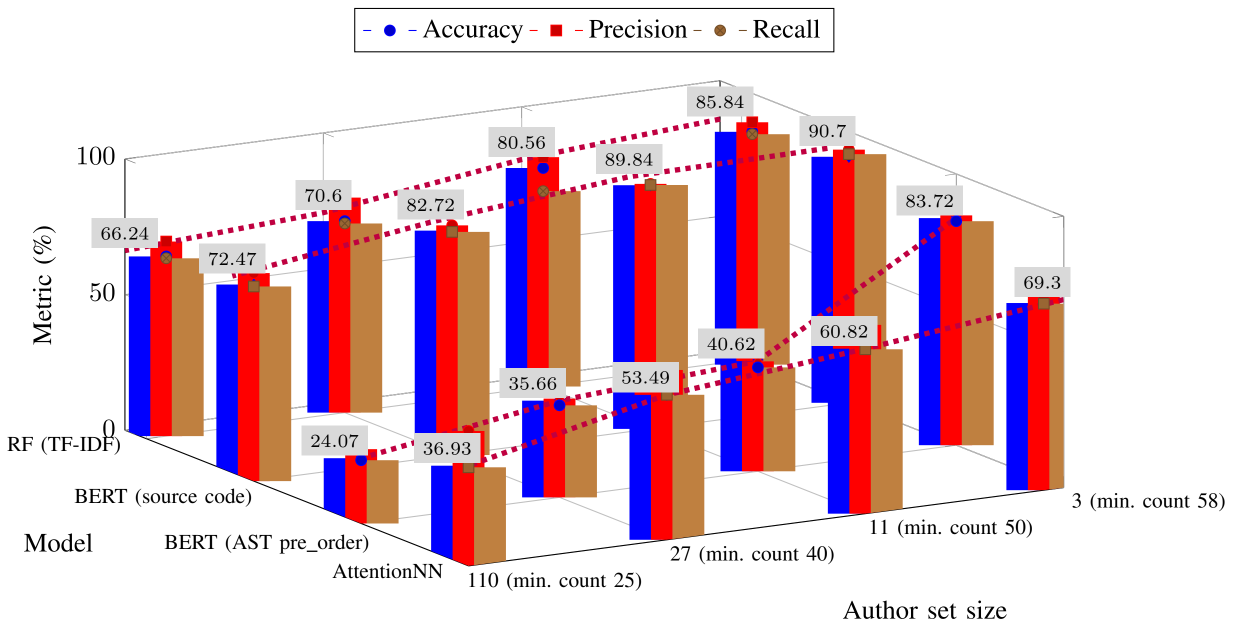


Fig. 4: 3D bar chart comparing Accuracy, Precision, and Recall across models and author set sizes.

We also analyzed the effect of function complexity. Figure 5 visualizes the frequency distribution of misclassified function lengths (in tokens) for Random Forests, BERT, and AttentionNN after scaling each model's misclassification count by an index:

$$\text{index}_i = \frac{\text{Number of misclassifications of the worst performing model}}{\text{Number of misclassifications of model } i} \quad (3)$$

Figure 6 shows the distribution of misclassified function Halstead complexity for Random Forests and BERT, revealing that while Random Forests struggle with higher complexity, BERT exhibits more misclassifications when complexity is low.

Furthermore, we assessed the AttentionNN model's performance based on function complexity by evaluating accuracy across bins of AST node count and AST depth. Figure 3 presents subfigures for these two metrics, indicating that the model learns better from structurally complex functions.

A 3D bar chart (Figure 4) further visualizes Accuracy, Precision, and Recall across models and author set sizes. (The code for this chart uses TikZ and pgfplots.)

V. Discussion

Our findings indicate that deep learning models—especially BERT trained on raw source code—consistently outperform traditional methods like Random Forests in terms of accuracy, albeit at the expense of longer training times. Random Forests,

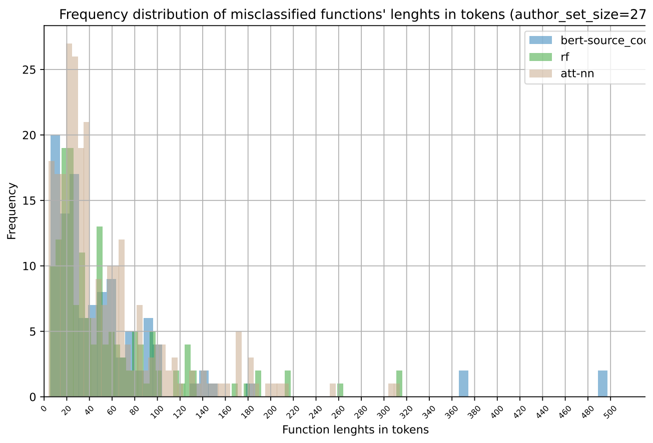


Fig. 5: Frequency distribution of misclassified function lengths (in tokens) for Random Forests, BERT, and AttentionNN.

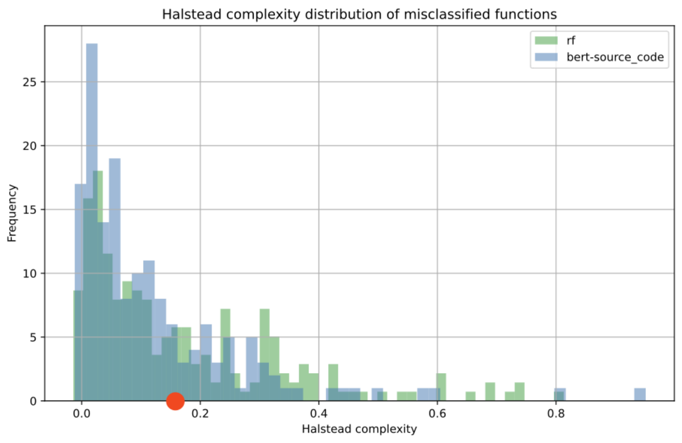


Fig. 6: Frequency distribution of misclassified function Halstead complexity for Random Forests and BERT.

despite their simplicity and efficiency, lag behind in accuracy when compared to BERT. Meanwhile, AttentionNN benefits from capturing structural complexity but tends to overfit with limited data. These results highlight the trade-offs between model complexity, training time, and overall performance. Future work might explore hybrid approaches (e.g., combining BERT embeddings with Random Forest classifiers) and improved regularization to balance accuracy and computational efficiency.

VI. Conclusion

This study compared traditional machine learning models and deep learning approaches for authorship attribution of individual functions. While BERT-based models achieved state-of-the-art accuracy, they required significantly longer training times and computational resources compared to Random Forests. The choice of input representation (raw source code tokens versus AST-based traversal) greatly influenced performance. In resource-constrained environments, simpler models like Random Forests remain a viable option, whereas applications demanding higher accuracy may favor deep learning models despite their overhead. Future research could investigate hybrid models and advanced regularization techniques to further improve generalization.

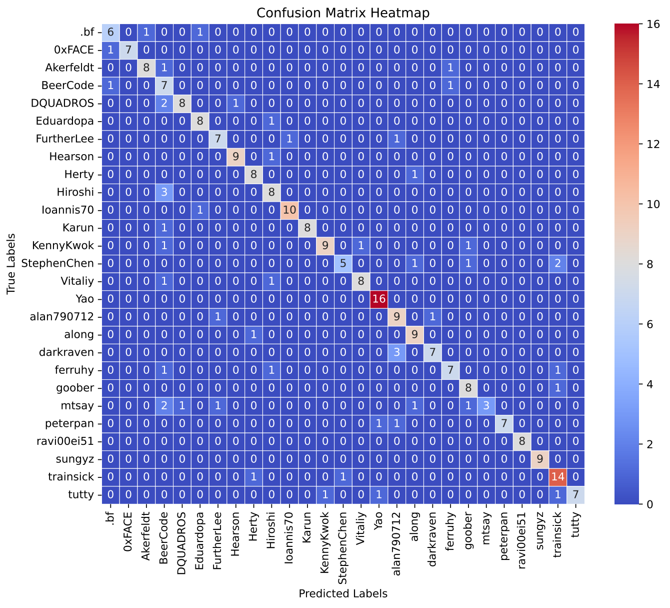
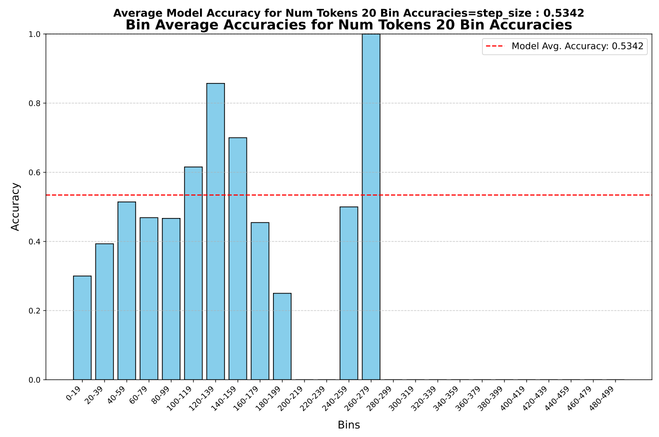
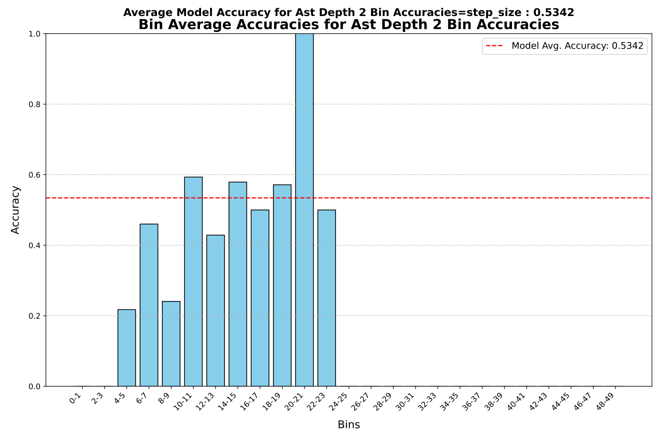


Fig. 7: Confusion matrix for BERT trained on raw source code (author set size 27).



(a) Accuracy vs. number of AST nodes



(b) Accuracy vs. AST depth

Fig. 8: AttentionNN accuracy for functions with different structural complexities.