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

Richard Čerňanský, Ing. Juraj Petrík

March 9, 2025

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 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.

1 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.

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.

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 [7]. 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.

2 Methods

2.1 Source code representation

In the paper [5], 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.

2.2 Abstract Syntax Tree

Based on the full definition [8] Abstract syntax tree (AST) is a tree-based representation of source code, where nodes represent various elements of the source code and paths represent relationships of the structure. AST is an inseparable part of compilation process for many reasons. It is used as an intermediate representation of program utilized for optimization and generation of machine code. Nodes of the tree can be either internal (non-leaf) or leafs. Internal nodes define the program's constructs or operations for their children. Leaf nodes store the actual textual value.

Below is an example of a simple C function and its corresponding Abstract Syntax Tree (AST):

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

The corresponding AST is structured as follows:

```
TranslationUnit
|--FunctionDefinition ‘int add(int a, int b) {‘
   |--BasicTypeSpecifier ‘int‘
   |--FunctionDeclarator ‘add(‘
   |  |--IdentifierDeclarator ‘add‘
   |  |--ParameterSuffix ‘(int a, int b)‘
   |     |--ParameterDeclaration ‘int a‘
   |     |  |--BasicTypeSpecifier ‘int‘
   |     |  |--IdentifierDeclarator ‘a‘
   |     |--ParameterDeclaration ‘int b‘
   |        |--BasicTypeSpecifier ‘int‘
```

```
|        |--IdentifierDeclarator ‘b‘
|--CompoundStatement ‘{ return a + b; }‘
   |--ReturnStatement ‘return a + b;‘
      |--AddExpression ‘a + b‘
         |--IdentifierName ‘a‘
         |--IdentifierName ‘b‘
```

2.2.1 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.
- **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.

2.2.2 Halstead complexity measures

For evaluation of complexity of examined functions we have used Halstead complexity measures Difficulty, Volume, Effort [4]. To aggregate them into single value, we used weighted sum of the three metrics weighting them in ratios 4:3:3 respectively. We did not incorporate Halstead length because we did separate analysis on length.

2.3 Our Data Representation

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

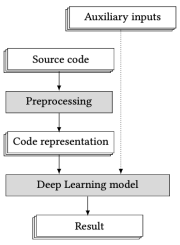


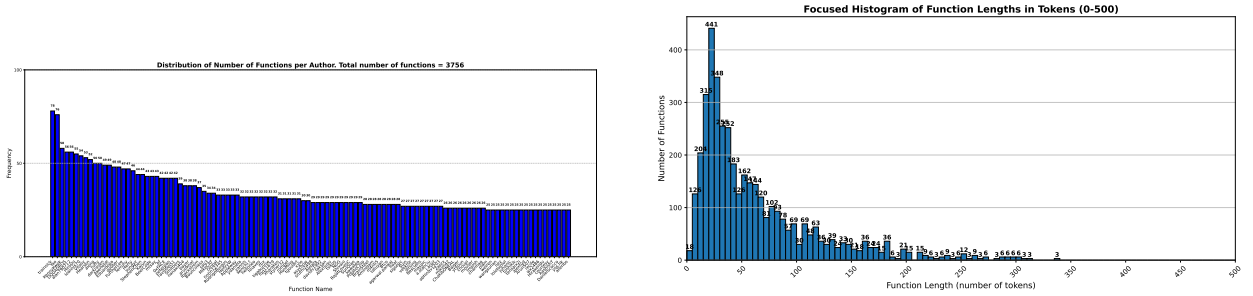
Figure 1: [2] High level overview of the function classification preditction task.

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 [6] was used for parsing the code into ASTs. Each model was trained in Tensorflow.Keras framework and with different author set sizes to evaluate scalabillity. 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.

2.4 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 3 004 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.



(a) Frequency distribution of author names

(b) Frequency distribution of function lengths

Figure 2: Important frequency distributions of our dataset

From the figure 7a 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.

3 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 traing 3 times and observed variance in evaluation metrics lower than 1%.

3.1 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. Firt 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.

3.2 BERT

For both BERT training setups we used Microsoft's pretrained graphcodebert-base [3] 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.

3.3 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 [1] 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 reveil the author the most.

3.4 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,

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

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.

4 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. The following table 1 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 4 visualizes the achieved performances on Accuracy, Precision and Recall. From table 1 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. 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.

Models' Author Attribution Accuracy for Different Author Set Sizes				
Model	**Author Set Size**			
	110	27	11	3
Random Forests (TF–IDFs w/o comments)	66.24% | 3.63 min	70.60% | 0.23 min	80.56% | 0.08 min	85.84% | 0.05 min
BERT (source code w/o comments)	72.47% | 30.33 min	82.72% | 9.49 min	89.84% | 5.3 min	**90.70% | 1.82 min**
BERT (AST pre_order traversal)	24.07% | 33.53 min	35.66% | 10.3 min	40.62% | 6.2 min	83.72% | 1.22 min
AttentionNN (AST NtN paths)	36.93% | 3.53 min	53.42% | 1.19 min	60.82% | 0.54 min	69.3% | 0.19 min

Table 1: Comparison of models across author attribution accuracy and training time for different author set sizes.

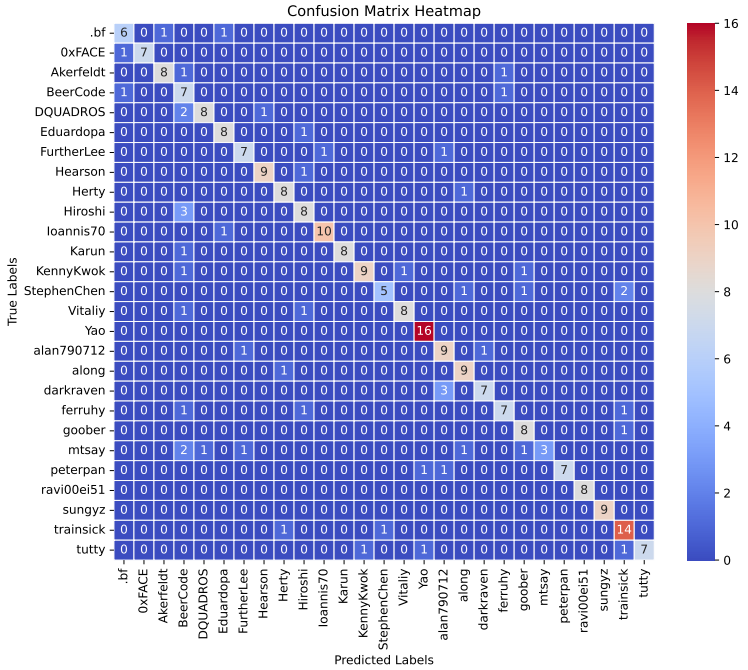


Figure 3: Confusion matrix for best performative model BERT trained on author set of size 27.

The confusion matrix shows, that the model performed best on 2 most represented authors (see 7a). Although if we take 8 most represented authors, only 5 of them are also the most accurately classified.

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.

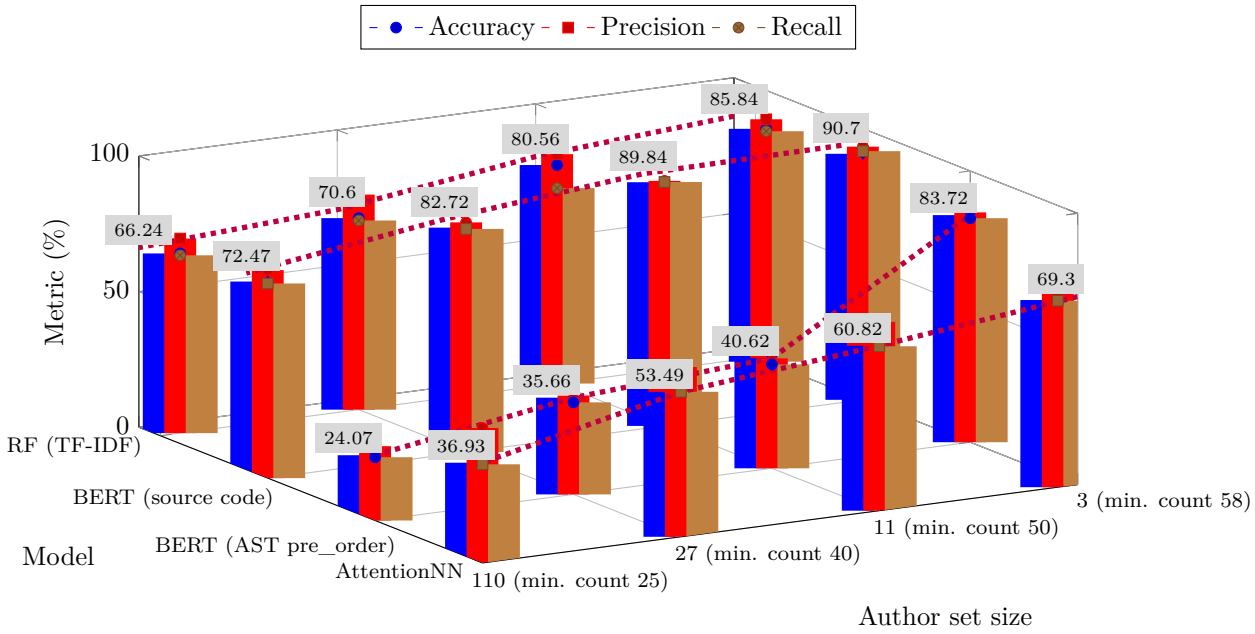


Figure 4: Bar chart comparing Accuracy (with accuracy values labeled), Precision, and Recall across models and author set sizes with minimum function count per author.

4.1 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:

$$\text{index}_i = \frac{\text{Number of misclassifications of the worst performative model}}{\text{Number of misclassifications of model}_i} \tag{3}$$

This way we obtain visual representation that objectively accounts for the fact that the number of errors is different. In the following figure we visualized the frequencies of misclassified function lengths in tokens for Random Forests, BERT and AttentionNN scaling each model with its index.

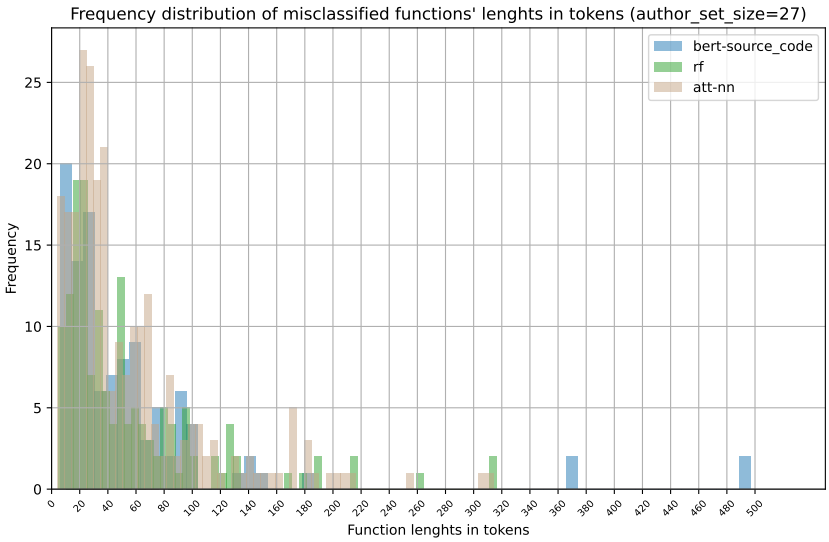


Figure 5: The frequency distribution of misclassified function lengths in tokens for Random Forests, BERT and AttentionNN

From the figure 5 the only statement we can say is that the model AttentionNN particularly struggled with functions of length lower than 40 tokens but did quite well on the longer ones. This will be discussed in the following subsection more deeply 4.2. Other models performance is not really distinguishable from this visualization, so we performed the same index strategy with the Halstead complexity metrics. Measuring each misclassified function's Halstead complexity and plotting it for Random Forests and BERT trained on tokenized source code, we obtained following figure.

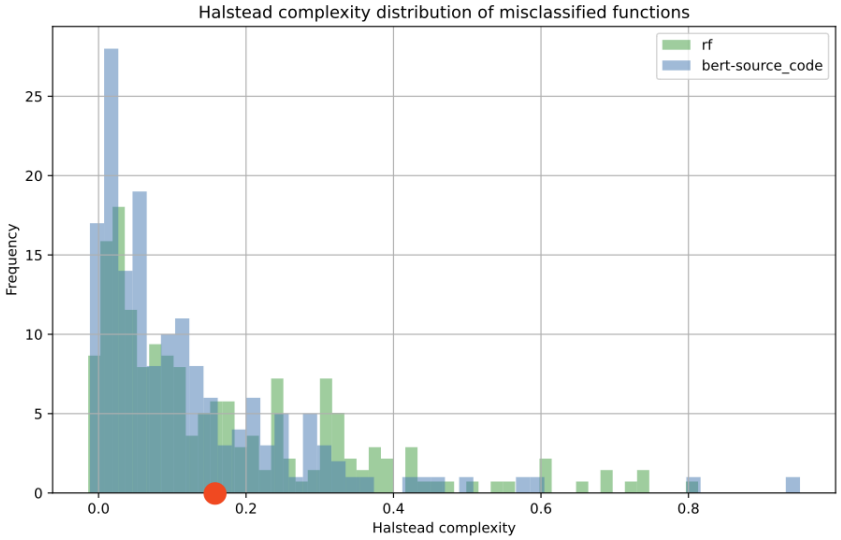


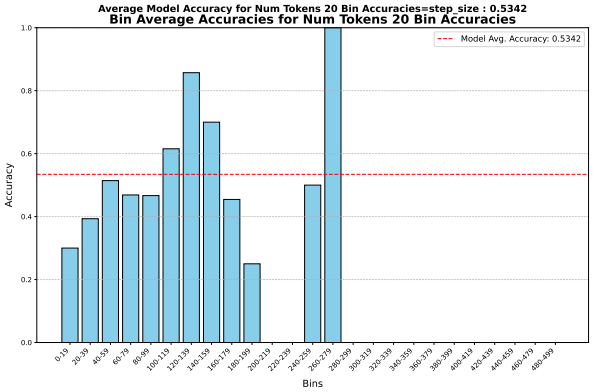
Figure 6: The frequency distribution of misclassified function Halstead complexity for Random Forests and BERT.

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

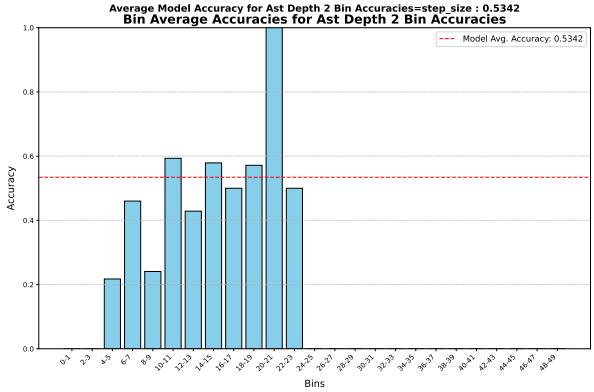
made by the Random Forest classifier than there are by BERT measured from the red point. However, closer we look to 0 on x axis, the more errors we see made by BERT. This would be the main difference between the deep learning and machine learning method. The deep learning method utilizes self-attention 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.

4.2 Attention neural network - complexity over quantity

Considering the observation in figure 5, where AttentionNN performed worst on lengths between 20 and 40 tokens, we assesed 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.



(a) Accuracy across different bins of number of nodes in function's AST



(b) Accuracy across different bins of depth in function's AST

Figure 7: AttNN Accuracy of functions with different complexities

From the figure we can deduce, there is an emerging trend in accuracy from bins of lower value to the bins of higher value. Eventhough 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 rather from more structural complex functions than simple non-nesting and control-flow divergent ones.

5 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 don't 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 don't 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 didn't 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're super efficient and a solid alternative when speed is a concern.

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. Exploring data augmentation techniques or better regularization strategies could also help deep learning models generalize better, especially with AST-based inputs.

6 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.

References

[1] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: learning distributed representations of code. *Proc. ACM Program. Lang.*, 3(POPL), January 2019.

[2] Alexander Brauckmann, Andrés Goens, Sebastian Ertel, and Jeronimo Castrillon. Compiler-based graph representations for deep learning models of code. In *Proceedings of the 29th International Conference on Compiler Construction*, CC 2020, page 201–211, New York, NY, USA, 2020. Association for Computing Machinery.

[3] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. Graphcodebert: Pre-training code representations with data flow, 2021.

[4] T Hariprasad, G Vidhyagaran, K Seenu, and Chandrasegar Thirumalai. Software complexity analysis using halstead metrics. In *2017 International Conference on Trends in Electronics and Informatics (ICEI)*, pages 1109–1113, 2017.

[5] Xie He, Arash Habibi Lashkari, Nikhill Vombatkere, and Dilli Prasad Sharma. Authorship attribution methods, challenges, and future research directions: A comprehensive survey. *Information*, 15(3), 2024.

[6] Leandro T. C. Melo, Rodrigo G. Ribeiro, Breno C. F. Guimarães, and Fernando Magno Quintão Pereira. Type inference for c: Applications to the static analysis of incomplete programs. *ACM Trans. Program. Lang. Syst.*, 42(3), November 2020.

[7] Ing. Juraj Petrik. Function name classifier. `https://github.com/Jur1cek/gcj-dataset`, 2024. Accessed: June 15, 2024.

[8] Weisong Sun, Chunrong Fang, Yun Miao, Yudu You, Mengzhe Yuan, Yuchen Chen, Quanjun Zhang, An Guo, Xiang Chen, Yang Liu, and Zhenyu Chen. Abstract syntax tree for programming language understanding and representation: How far are we?, 2023.