Proceedings on Privacy Enhancing Technologies ; 2021 (4):464–479

Edwin Dauber*, Robert Erbacher, Gregory Shearer, Michael Weisman, Frederica Nelson, and Rachel Greenstadt

Supervised Authorship Segmentation of Open Source Code Projects

Abstract: Source code authorship attribution can be used for many types of intelligence on binaries and executables, including forensics, but introduces a threat to the privacy of anonymous programmers. Previous work has shown how to attribute individually authored code files and code segments. In this work, we examine authorship segmentation, in which we determine authorship of arbitrary parts of a program. While previous work has performed segmentation at the textual level, we attempt to attribute subtrees of the abstract syntax tree (AST). We focus on two primary problems: identifying the primary author of an arbitrary AST subtree and identifying on which edges of the AST primary authorship changes. We demonstrate that the former is a difficult problem but the later is much easier. We also demonstrate methods by which we can leverage the easier problem to improve accuracy for the harder problem. We show that while identifying the author of subtrees is difficult overall, this is primarily due to the abundance of small subtrees: in the validation set we can attribute subtrees of at least 25 nodes with accuracy over 80% and at least 33 nodes with accuracy over 90%, while in the test set we can attribute subtrees of at least 33 nodes with accuracy of 70%. While our baseline accuracy for single AST nodes is 20.21% for the validation set and 35.66% for the test set, we present techniques by which we can increase this accuracy to 42.01% and 49.21% respectively. We further present observations about collaborative code found on GitHub that may drive further research.

Keywords: Stylometry, code authorship attribution, segmentation

DOI 10.2478/popets-2021-0080
Received 2021-02-28; revised 2021-06-15; accepted 2021-06-16.

***Corresponding Author: Edwin Dauber:** Drexel University, Email: egd34@drexel.edu
Robert Erbacher: United States Army Research Laboratory, Email: robert.f.erbacher.civ@mail.mil
Gregory Shearer: ICF International, Email: gregory.g.shearer.ctr@mail.mil
Michael Weisman: United States Army Research Laboratory, Email: michael.j.weisman2.civ@mail.mil

1 Introduction

The attribution of source code has numerous potential security and forensic applications, but also poses serious risks to privacy.While there are clear benefits to being able to attribute malware, cyber-attacks, and deliberate software vulnerabilities, the same forensics techniques used for these purposes may also be used to violate the privacy of other programmers. Activists developing censorship circumvention tools or programmers contributing to open source projects when discouraged by their employers are likely to suffer significant consequences if their anonymous or pseudonymous code is attributed.

While the problem of attributing source code samples known to be written by a single author has been examined in some depth, attribution of collaborative code files has been the subject of less research. In this work, we perform supervised authorship segmentation of open source code projects. We note that many modern programs, both proprietary and open source, are the result of collaboration. We also note that while many open source contributors are known by name, many others are only known by pseudonyms. While identifying the contributors to a code project is a privacy risk by itself, fine-grained attribution of portions of code may pose even greater threats. Additionally, such techniques may prove useful for reinforcing membership in the set of collaborators.

Unlike natural language, which can only be segmented at the textual level, source code can be segmented at either the textual level or the underlying level - the abstract syntax tree, or AST. The AST is a tree representation of source code with nodes representing syntactic constructs in the code [3]. While segmenting by lines of code offers readability, segmenting the abstract syntax tree may allow greater granularity and segmentation by functional components. This may al-

Frederica Nelson: United States Army Research Laboratory, Email: frederica.f.nelson.civ@mail.mil
Rachel Greenstadt: New York University, Email: greenstadt@nyu.edu

low for identification of authors of small but significant changes to a codebase and be extensible to binaries.

1.1 Contributions

Our exhaustive literature review demonstrates this to be the first work which performs authorship attribution of collaborative source code while guaranteeing that training data and the code being attributed come from different code projects. We searched on Google Scholar for combinations and variations of "stylometry," "stylometric," "segmentation," "authorship," "programmer," "attribution," "collaborative," "open source," "code," and "deanonymization." While we found many papers in our search, most were either about single programmer source code or about collaborative natural language documents. The only two papers on attribution of collaborative code found at the time of this writing are cited in the next section, and while both guarantee not training and testing on code from the same code file, neither makes any such guarantee at the level of projects.

While this restricts our ability to capture authors in our suspect set, it provides a more realistic scenario for many use cases. This is also the first work to apply source code authorship segmentation under open world conditions, in which we attempt to identify when code originates outside our suspect set. We show that it is possible to detect changes in authorship based on the abstract syntax tree. We present techniques for attributing subtrees of the abstract syntax tree and demonstrate their effectiveness under varying conditions. We demonstrate that we can attribute arbitrary abstract syntax tree subtrees of suitable size with relatively high accuracy; we present techniques which can alleviate some of the difficulty of attributing small subtrees.

2 Related Work

2.1 Source Code Authorship Attribution

Our work builds primarily on two previous pieces of work. First is the work of Caliskan-Islam et al. using random forests to attribute Google Code Jam submissions, using features extracted from the abstract syntax tree (AST) [9]. They achieved over 90% accuracy with suspect sets of 1600 authors on samples averaging 70 lines of code. This method was extended by Dauber et al. to work for code segments belonging to accounts on version control systems such as GitHub [10]. Our work similarly extracts AST based features, but instead of using these features to attribute files or code segments, we use them to attribute segments of the AST. As a result, we also use a reduced feature set, leaving out layout and lexical features.

Abuhamed et al. used deep learning to perform highly accurate code segmentation using a system they called Multi-χ [2]. Their technique uses a sliding-window approach over lines of code to perform attribution. We note four main differences between their work and ours, one technical and three in evaluative methodology. First, their technique relies on *word2vec*, a method of representing text for deep learning, for features, as opposed to our method using AST nodes [19]. While this works well for source code, we expect it would adapt poorly to compiled binaries. Second, they make no strong guarantee that they do not train and test on files from the same project, instead randomly selecting files for testing and training. Ours is the first work to add this guarantee to the collaborative code problem on GitHub. We note that allowing training and testing on files from the same project is necessary for building a large training set with good coverage of the set of true authors, but it admits code into the training set which may be inherently related to the code in the testing set. Thus, not enforcing this guarantee both potentially inflates the accuracy and coverage of the method compared to many interesting use cases. Third, they select a set of nine major collaborative projects. We collected a larger set of projects, including a mix of project sizes. Finally, they specifically exclude test segments from programmers outside of their training set, considering the open world problem out of scope. Conversely, we consider addressing the open world problem an essential part of our methodology, as it allows us to segment files with only a single author with available training samples.

Abuhamad et al. proposed a system called DL-CAIS which effectively attributes code at large scale and cross-language [1]. While they do test on code from GitHub, their work does not address the problem of collaborative code, nor the problem of small code samples.

There are many other proposed methods and feature sets for source code authorship attribution but compared to the previously discussed methods these had worse accuracy, smaller suspect sets, or were highly specialized. Frantzeskou et al. used byte level n-grams to achieve high accuracy with small suspect sets [13–15]. Ding and Samadzadeh studied a set of 46 programmers and Java using statistical methods [11]. MacDonnel et

al. analyzed C++ code from a set of 7 professional programmers using neural networks, multiple discriminant analysis, and case-based reasoning [18]. Burrows et al. proposed techniques that achieved high accuracy for small suspect sets but had poor scalability [6–8].

2.2 Text Authorship Attribution

Fifield et al. used a sliding window approach for segmentation [12]. In a series of papers, Koppel et al. and later Akiva and Koppel, performed segmentation using sentence level classification using a technique involving clustering to identify representative sentences to use to classify the remaining sentences with a gap filling technique to deal with difficult to attribute sentences [4, 5, 16]. Both of these perform unsupervised segmentation of large documents. The work of Abuhamed et al. on Mulit-χ is similar to the work of Fifield et al., while our work is closer to the work by Akiva and Koppel.

3 Methodology

3.1 Problem Statement

In this paper, we take on the role of an analyst attempting to perform authorship segmentation on a collaborative code project. We assume that we have accurate information about the set of authors involved in the project but that this information may not be complete. In other words, we assume that while we may not have identified the complete set of contributors, any programmer we believe to be a contributor is one. Our goal is to segment over the known programmers, identifying when code segments may be by an unidentified programmer.

More formally, given a code project P and a set of n programmers, $S = \{A_1 \dots A_n\}$, known to be contributors to P, we attempt to attribute every possible AST subtree in P to one of $n+1$ classes: one for each of the n programmers and one for everyone outside of S. We note that while there are alternate ways to address the open world problem, we believe that using a class to represent out of world authors is essential to enhance the range of projects for which our techniques can be applied. Our dataset includes many projects for which we would otherwise only have a single programmer in our training set, which is not well suited for classification and would require a separate method to handle.

3.2 Problem Model

We assume an analyst has access to a corpus of source code repositories containing collaborative code projects of known authorship history, including contributions from known contributors of the target project. While it may be possible to supplement training data with individually written code, we do not assume access to such. We assume programmers have contributed to collaborative code files and have not concealed their coding style more than necessary for collaboration.

3.3 Data

We collected a set of 155 collaborative C++ projects from GitHub . These projects were collected by starting with a collection of seed projects, some of which were large with many programmers and some of which were small with few programmers. For each programmer, we collected other projects that programmer contributed to, and repeated the process of identifying programmers and collection projects until we terminated for computational reasons. Some projects are related, and some are forks. Some projects lacked even a single programmer for whom we were able to build a training set. Ultimately, we built a validation set of 53 projects and a test set of 29 projects. In building our validation and test sets, our first step was to remove all projects which either had parsing errors or which only had unique programmers. We further extended this criterion to remove projects which only had programmers if we were to admit identical code samples or related projects. Additionally, we discarded projects for which we were unable to programmatically link AST nodes back to lines of code. We then grouped the remaining projects into groups based on the number of classes, and for each group randomly selected members for the validation and test sets, ensuring that each group was represented in both sets proportionally to the overall size of the group. Our dataset was additionally constrained by language and by practical considerations including computational resources and time.

As part of our data processing, we use the fuzzy parser joern to extract ASTs for each file in our dataset [20]. We also use git blame to assign ground truth to every line of code in the file and programmatically link each AST node back to the line or lines of code which created it to propagate ground truth to every AST node.

Our validation set spanned 43 classes while our test set spanned 39 classes. We further identified a subset of the validation set of 35 projects spanning 14 classes for

which we were able to build training sets of at least 100 samples per class to use for additional experimentation. We note that our data is inherently unbalanced, with projects being of different sizes and authors contributing different amounts.

3.4 Features

In this work, we use a feature set derived from the work of Caliskan-Islam et al., focusing primarily on AST features [9]. In line with our objective to make our system independent of the text of the code, the only non-AST based features we use are any keywords which are preserved in the parsing process. The full list of keywords can be found in Table 1. The other features include the number of AST nodes, the number of functions, the maximum and average depth of the AST, the maximum and average breadth of the AST, counts and average depths of types of AST nodes, and the counts of AST bigrams. Additionally, we count sibling bigrams and trigrams of the same node type. We define a sibling ngram as n AST nodes which share a parent and are adjacent in left-to-right read order. In order to address the size difference in AST subtree sizes, we normalize our feature space by the number of AST nodes. For each experiment, we discard features which do not vary at all or which only apply to a single programmer. Our final feature vectors are sparse, but smaller than those observed in the work of Dauber et al. [10]. In the validation set we have an average of 11.03 non-zero features and 210.64 zero-valued features per subtree. In the test set we have an average of 11.38 non-zero features and 210.47 zero-valued features per subtree. We note that this reduced feature set will necessarily result in a weaker classifier than the ones produced in these prior works.

Table 1. C++ Keywords

alignas	alignof	and_eq	and
asm	auto	bitand	bitor
bool	break	case	catch
char	char16_t	char32_t	class
compl	const_cast	const	constexpr
continue	decltype	default	delete
do	double	dynamic_cast	else
enum	explicit	export	extern
FALSE	final	float	for
friend	goto	if	inline
int	long	mutable	namespace
new	noexcept	not_eq	not
nullptr	operator	or_eq	or
override	private	protected	public
register	reinterpret_cast	return	short
signed	sizeof	static_assert	static_cast
static	struct	switch	template
this	thread_local	throw	TRUE
try	typedef	typeid	typename
union	unsigned	using	virtual
void	volatile	wchar_t	while
xor_eq	xor		

3.5 Learning Methodology

We use a random forest classifier with 100 trees and a max depth of 50. These parameters are based on the prior works. While a full parameter search may result in improved accuracy, this is beyond the scope of this work. We examine the classification results for four different attribution techniques. First, we directly attribute each subtree to the class predicted by the classifier, defining our baseline.

Second, we perform a technique we call *adjustment* in which we adjust the output confidence distribution based on the confidence distributions of the parent and child as well as predictions of the likelihood of authorship change. Specifically, for class i, original classifier confidence $sd[i]$, parent contribution $pc[i]$, children contributions $cc[j][i]$ for each child j from 1 to n, we calculate the new confidence $nd[i]$ using Equation 1.

$$nd[i] = sd[i] + pc[i] + \frac{\sum_{j=0}^{n-1} cc[j][i]}{n} \tag{1}$$

We calculate the individual parent and child contributions $c[i]$ based on weight parameter w, authorship change factor cf, original parent or child confidence $d[i]$, and parent or child prediction $dmax$ using Equation 2.

$$\begin{cases} i \neq dmax & c[i] = w * cf * d[i] \\ i = dmax & c[i] = w * cf * d[i] * -1 \\ \text{Parent/Child does not exist} & c[i] = 0 \end{cases} \tag{2}$$

We compute the change factor cf as a function of a threshold parameter t and classifier confidence of authorship change da using Equation 3, noting that when $t = 0$, $cf = da$. Based on our parameterization experiments, we selected a threshold of 0 and a weight 0.02.

$$\begin{cases} da = t & cf = 0.0 \\ da > t & cf = \frac{da-t}{1.0-t} \\ da < t & cf = \frac{da-t}{t} \end{cases} \tag{3}$$

Third, we perform a technique we call *blocking* in which we set a threshold based on which we group connecting subtrees which are predicted to belong to the same author and average the classification results to attribute the entire block at once. Optionally, we can also apply a penalty between adjacent blocks to decrease the likelihood that adjacent blocks get attributed to the same programmer. For blocking, we note that a threshold of 0 will result in the entire project being treated as single-authored while a threshold of 1 will require a classifier confidence of 1 to link two subtrees. Based on our parameterization experiments, we selected a threshold of 0.85 and no penalty. This technique is based on the work of Dauber et al. on account code segments, but instead of having a known account we must first predict whether the samples belong to the same individual [10].

Fourth, we perform a technique we call *blocking-adjustment*. This technique starts out by blocking and then performing adjustment on the new distributions using the original distributions, with one modification: we add a weighted contribution from the original distribution of the target subtree. The result of this is that we gain the benefit of a larger number of linked samples, but then can offset in the event that our linking of samples was incorrect. Based on our parameterization experiments, we set our blocking threshold and penalty to 0, our adjustment threshold to 0, and our adjustment weight to 0.02.

Finally, we examine a combination technique where we can vary technique and parameters based on the size of the subtree we are examining. Based on our parameterization experiments, we identified best results with blocking-adjustment with blocking threshold of 0.05 and penalty of 0.8 and adjustment weight of 0.01 and threshold of 0 for subtrees up to 9 nodes. From 10 to 35 nodes, we continue with blocking-adjustment but change the adjustment weight to 0.03. For all larger subtrees, we just use adjustment, with the same parameters as identified previously.

3.6 Experimental Design

We attribute each project individually and train on the remaining projects. We perform experiments on the full validation and test sets and also control for the number of classes and training examples. To control for number of classes, we exclude projects with different numbers of classes, rather than removing classes or adding distractors. While this results in fewer projects in the datasets, it reduces other potential confounding factors. To control for the number of training examples, we randomly remove excess examples. We note that training examples are AST subtrees, not files or projects.

We evaluate our methods primarily using accuracy. This accuracy represents the percentage of total AST subtrees in the dataset correctly attributed. This metric gives a high-level measure of success. However, our dataset is unbalanced. We have projects of varying scales and programmers with varying amounts of contributions to the dataset. As a result, we also consider project average accuracy and balanced accuracy. Project average accuracy, shortened to average accuracy, is computed by computing the accuracy for each individual project and taking the average. Balanced accuracy is computed by computing the accuracy for each programmer and taking the average. These metrics allow us to account for the fact that the dominating projects and classes may not be representative of the entire set.

To further contextualize our results, we break down accuracy based on sample size. In this case, our reported accuracy represents the percentage of samples of that size which are correctly attributed. In the case of larger samples which may have multiple contributors, the correct author is considered to be the author with the most nodes contributed to that AST subtree.

We note that while the projects we are examining are typically larger than the samples studied by Caliskan et al., the individual samples are much smaller [9]. Additionally, we use a smaller feature set due to omitting textual features. However, our suspect sets are smaller, with $n+1$ classes for each project where n is the number of project contributors for whom we were able to prepare training data. The result is that while there is less information to be used for the learning task, the baseline random-chance probability of success is higher.

4 Results

For brevity, we have omitted the details of our parameterization experiments. We only summarize the relevant results here. Interested readers can find more details in Appendix A.

4.1 Change in Authorship Detection

Our techniques depend on our ability to identify changes in authorship. In these experiments, we use the entire validation set. We performed two-class classification to

determine if each node had the same author as the parent node, excluding project and file root nodes. Figure 1 shows our accuracy for identifying authorship changes above the threshold.

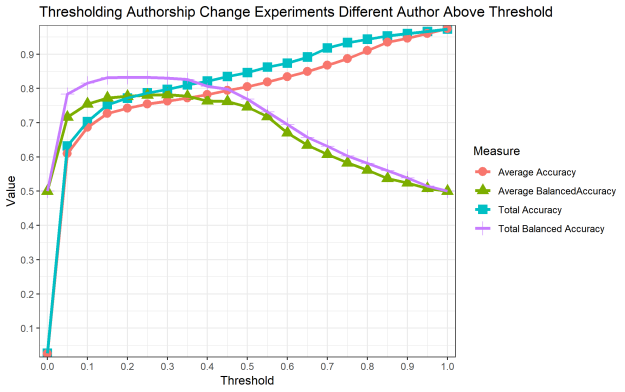

Fig. 1. This graph shows the results for identifying a change in authorship as percent accuracy vs. threshold. Average refers to computing accuracy or balanced accuracy for each project and taking the average, while total refers to computing accuracy or balanced accuracy for the entire set of nodes in the dataset.

For identifying authorship change above the threshold, we observe that thresholds between .15 and .30 maximize balanced accuracy, while overall accuracy is maximized at a threshold of 1.00 because of the imbalanced data. For identifying that authorship remains the same above the threshold, we observe a near mirror image with the critical thresholds occurring between .70 and .80. Because the results are a near mirror image, we have omitted a figure for these results. We note that authorship remains the same for most nodes. We also note that *adjustment* relies on identifying changes above a threshold while *blocking* relies on identifying consistency above a threshold.

4.2 Validation Set Experiments

We performed our parameterization experiments on our validation set. While full details are left for the appendix, we summarize the results for the selected parameterizations in Table 2. We note that our parameters were chosen based on improvement for all three observed metrics over the baseline. Because we noted a tension between improving all accuracy metrics and optimizing a single metric, and because overall accuracy is the easiest metric to compute and interpret, for the combination approach we chose to optimize overall accuracy, and did not compute average and balanced accuracy. The overall accuracy for this approach was 46.63%. Not only is this greater than our accuracy for our other approaches when selecting parameters based on all three metrics, but it also exceeds any other overall accuracy for any set of parameters evaluated for any individual technique.

Table 2. Summary of Validation Experiments

Analysis Method	Total	Average	Balanced
Baseline	**24.97%**	**42.98%**	**63.42%**
Adjustment	**25.15%**	**43.33%**	**64.56%**
Blocking	**25.67%**	**46.60%**	**66.36%**
Blocking-Adjustment	**28.30%**	**45.92%**	**64.84%**

Dauber et al. proposed associating samples with their classification confidence and using this to determine whether or not to trust the attribution [10]. We attempted a similar analysis on our results. Figure 2 shows these results. We note that blocking-adjustment shows no notable change in accuracy as confidence changes. Each of the other techniques shows minimal change until reaching high confidence, at which point accuracy decreases. As a result, for these circumstances we cannot rely on classification confidence for enhanced attribution. We note that due to poor performance as a predictor of success on the individual methods, we did not perform this analysis for our combined technique. Considering that the abundance of small samples might be a factor in these results, we examined confidence for our baseline for selected subtree sizes. Figure 3 shows these results. We note that for medium sized subtrees there is some improvement for high confidence attributions, but the range of subtree sizes for which classification confidence may be useful is limited.

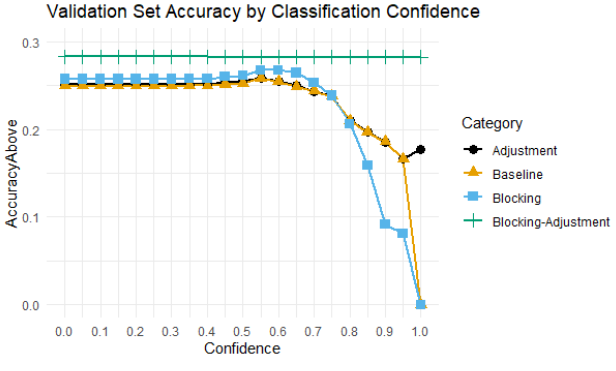

Fig. 2. This graph shows the accuracy based on classification confidence for our validation set.

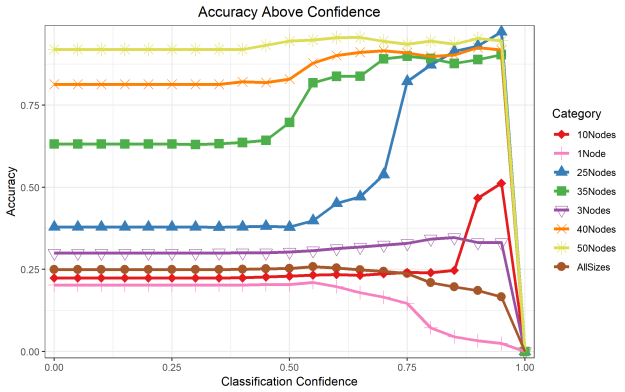

Fig. 3. This graph shows how accuracy changes as we discard attributions with low classification confidence for various sized subtrees in the base experiments.

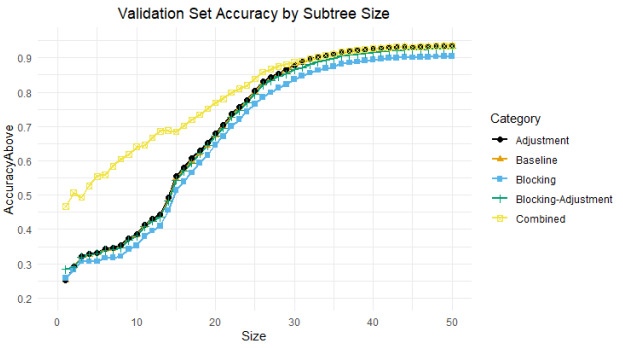

Fig. 4. This graph shows the accuracy based on subtree size for our validation set.

Thus far we attempt to attribute all subtrees of the AST, regardless of size. But we know that low information samples are innately harder to classify than higher information samples. In order to examine the effect of sample size on attribution accuracy, we separated our samples by subtree size. Figure 4 shows these results. We note that we have larger samples than those shown here, but accuracy does not continue to substantively change beyond this point. This graph confirms that our low overall accuracy is due to the abundance of small samples in our dataset. Further examination determines that approximately half of our dataset consists of subtrees consisting of only a single leaf node. Once subtrees become large enough, they become easy to attribute. We can use this graph to set a limit on the granularity reported by our method based on acceptable error rate. We note that the large improvement in our combined approach suggests that not only should we change analysis techniques for different subtree sizes, but also parameterizations. We also note that each line of code typically corresponds to many AST nodes, with Leßenich et al. reporting an average of 7 AST nodes per line of code [17]. Thus, a sample with 35 AST nodes, for which we obtain approximately 90% accuracy, would average to 5 lines of code. We note that this is similar in size to the samples on which Multi-χ perform well [2].

4.3 Test Set Experiments

Because our techniques are highly parameterized, we withheld a set of projects to use as a test set. Using the parameters selected on our validation set, we repeated our experiments on the test set. Because we observed no notable positive change in accuracy relating to classification confidence, we omitted that analysis. Table 3 shows the results from our experiments. For the combined technique, our accuracy was 42.43%.

Table 3. Summary of Results on Test Set

Experiment	Total	Average	Balanced
Baseline	**29.53%**	**44.11%**	**63.11%**
Adjustment	**29.42%**	**44.38%**	**65.41%**
Blocking	**29.79%**	**46.82%**	**66.97%**
Blocking-Adjustment	**29.43%**	**45.62%**	**64.74%**

Due to the importance of subtree size in attribution accuracy, we repeated our size analysis for the test set. Figure 5 shows these results. We note again that we have larger samples than shown here. Blocking accuracy continues to slightly increase, while the accuracy for the remaining techniques slightly decreases. We note a few important differences between our test results from our validation results. First, we note that small subtrees start at a higher accuracy in the test set, but large subtrees reach higher accuracy in the validation set. While it would take additional analysis beyond the scope of this paper to explain, our test set leaf nodes are easier to attribute than in the validation set, but the larger subtrees are harder to attribute than in the validation set. However, we can still set a size threshold based on acceptable attribution error rate.

5 Discussion and Analysis

5.1 Discussion

Dauber et al. discussed the importance of uncorrelated prediction errors between samples in their aggregation

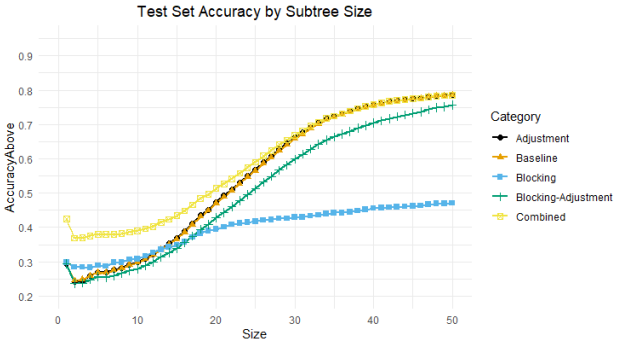

Fig. 5. This graph shows the accuracy based on subtree size for the test set.

approach [10]. The small number of classes and the relationship between samples makes this condition impossible to guarantee in our work, but we still see benefit from similar approaches. However, they also found that higher classification confidence corresponded to increased accuracy. This trend did not hold for our data, and in fact very high confidence samples yielded lower accuracy. While it would require further analysis to fully explain this, the abundance of low information samples provides one likely explanation. When samples have such little information, there is little possibility for alternate decisions by decision trees. Since smaller samples are harder to attribute and more abundant than larger samples, this would explain the low accuracy corresponding to high confidence.

Our overall accuracy is less than 50%. However, once we reach samples over 35 nodes, which correspond to an average of 5 lines of code, accuracy for the validation set exceeds 90% while accuracy for the test set is approximately 75%. Table 4 summarizes the accuracy for key node sizes in both the validation and test accuracy, along with an estimate of number of lines of code based on the average of 7 nodes per line. While these techniques have the potential for greater granularity offered than lines of code, lines of code makes for an easily understandable metric for discussing the privacy threat of these techniques. Contributors of patches even 3 lines of code long can be identified in our validation set with nearly 80% accuracy and over 50% accuracy in our test set.

A determined analyst will use multiple complimentary techniques to attribute a code segment. Even if the target attribution is wrong, if attribution of nearby code segments, including ancestors and descendants of the target segment, are correct, those can be used to inform other investigative methods. While relaxed attribution is beyond the scope of this work, it is known that

Table 4. Key Size Threshold Summary

Nodes	Est. LOC	Validation Acc.	Test Acc.
1	< 1	46.63%	42.43%
2	< 1	50.62%	36.84%
7	1	58.43%	37.93%
14	2	68.82%	42.39%
21	3	78.06%	52.74%
28	4	87.48%	64.04%
35	5	91.11%	72.42%
42	6	93.01%	76.76%
49	7	93.43%	78.40%

accuracy for relaxed attribution can be no worse than the accuracy we report. With all of these factors, even attributions with medium accuracy can pose a greater privacy threat. With these techniques, even minor contributors to a project risk attribution.

5.2 Observations

This subsection presents a qualitative analysis of the state of authorship on GitHub, and is limited to the subset of projects encountered in our data collection process. We acknowledge that these observations are anecdotal, but they may provide insight when considering future research directions.

We found many GitHub projects have only a single author. When collaboration does occur, it tends to be primarily in the form of projects consisting of multiple nearly singly-authored files. When collaborative files exist, they typically have a single dominant author with other authors providing small changes. Many of the authors we came across in our study either only participated in a single collaborative project or used a variety of programming languages.

5.3 Baseline Control Experiments

Our dataset contains projects with a variety of number of classes for attribution. Additionally, the classes have varying numbers of training samples. While this represents real-world use cases well, it leads to additional possible factors that can affect accuracy results.

In these experiments we use subsets of the validation set to control for the number of classes and training samples. Figure 6 shows the overall accuracy results for fixed numbers of classes. We note that average accuracy has a similar graph, with slower drops in accuracy, corresponding to greater resilience when projects are

dropped. The results for balanced accuracy begin similarly, but after approximately 300 training samples balanced accuracy no longer changes. We omit those figures for simplicity. Figure 7 shows the number of projects in each of these experiments.

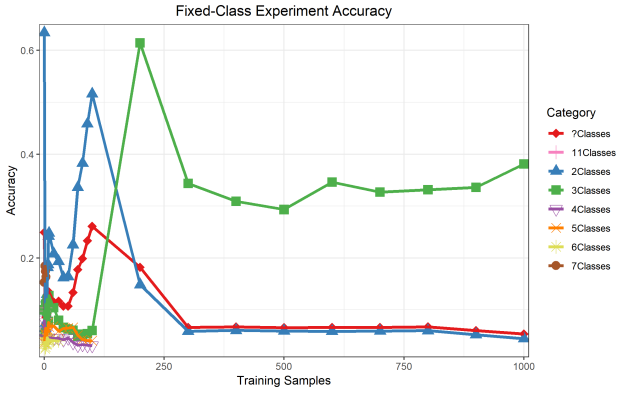

Fig. 6. This graph shows overall accuracy for fixed numbers of classes, represented as percent accuracy vs. number of training samples per class. ?Classes refers to using all available projects. 0 training samples refers to using all available training samples.

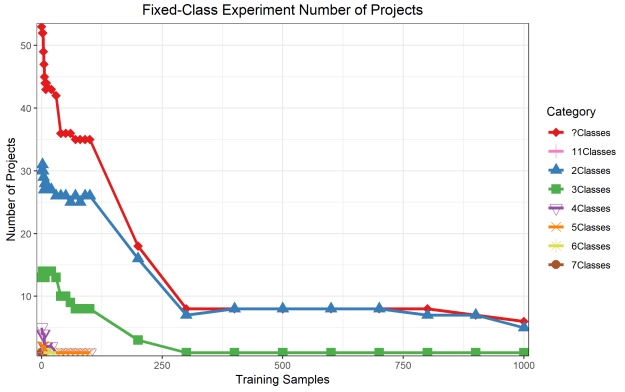

Fig. 7. This graph shows the number of validation projects for fixed numbers of classes. The ?Classes line and x-axis origin are as defined as in Figure 6.

These results show that when the included projects remain fixed, accuracy tends to increase or stay constant as the number of training examples increases. However, large changes in accuracy in either direction tend to correspond to changes in the number of projects, suggesting that the projects themselves are the dominating factors. It is worth noting that the base result for our large experiment is a balanced accuracy of 63.42%, average per project accuracy of 42.98%, and overall accuracy of 24.97%.

Figure 8 shows the overall accuracy for baseline results with a fixed number of training examples per class. Figure 9 shows the balanced accuracy results for the same experiments. The results for average accuracy start higher, for small numbers of classes, but otherwise the graphs look the same, so those results are omitted. Figure 10 shows the number of projects in each experiment. Together, these sets of experiments suggest a relationship between accuracy and the number of classes. It will require additional data and experiments to determine if this relationship is coincidence or causation.

Fig. 8. This graph shows overall accuracy for fixed numbers of training examples per class, represented as percent accuracy vs. number of classes. ?TrainingSamples refers to using all available training samples. 0 classes refers to using all available projects.

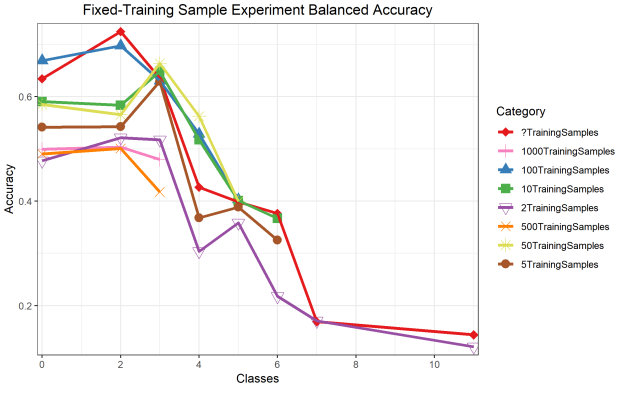

Fig. 9. This graph shows the balanced accuracy results for fixed numbers of training examples per class, represented as percent accuracy vs. number of classes. The ?TrainingSamples line and 0 classes are defined as in Figure 8.

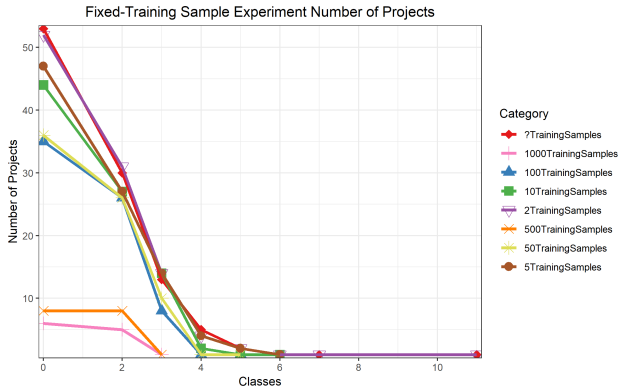

Fig. 10. This graph shows the number of validation projects for the experiments with a variable number of classes and fixed number of training examples per class. The ?TrainingSamples line and 0 classes are defined as in Figure 8.

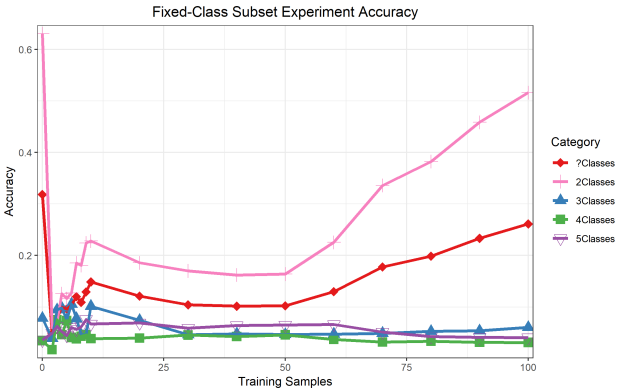

Fig. 11. This graph shows the overall accuracy results for fixed numbers of classes on a subset of our validation set, represented as percent accuracy vs. number of training samples per class. 0 training samples and the ?Classes line are defined as in Figure 6.

Based on the relationships observed in the previous experiments, we selected a subset of projects for which each class has at least 100 training examples, and varied the number of training examples per class from 2 to 100. Table 5 shows the number of projects for each number of classes. Figure 11 shows the baseline results for this subset as we vary the number of training samples for each class. The results for average accuracy are similar, except the results for projects with 3 classes are higher, so we omit a figure for these results. The balanced accuracy results for 2 classes are higher, but follow a similar pattern. For other numbers of classes, balanced accuracy is higher than overall or average accuracy, but after approximately 10 training samples the results level out, so we omit a figure for simplicity.

Table 5. Number of Projects Per Number of Classes

Number of Classes	Number of Projects
2	25
3	8
4	1
5	1
Total	35

In these experiments, changes in accuracy are more likely going to be the result of increased training data, since the set of projects will be held constant. We have not confirmed why we observe high accuracy for some experiments with very little training data, but otherwise accuracy generally increases or remains constant with increased number of training examples. Combined with our previous results, we observe that the amount of training examples is important, but less important than the specific projects and authors involved.

5.4 Accuracy Variability Between Projects

Throughout the experiments presented previously, we have noticed a high level of variability in accuracy between individual projects. In previous sections, we have discussed average accuracy. In this section, we investigate the range by examining minimum and maximum accuracy. Table 6 summarizes the accuracy and balanced accuracy minima and maxima observed in each of our experiments. Balanced accuracy is abbreviated as Ba., adjustment is abbreviated as Adj., blocking is abbreviated as Blo., average is abbreviated as avg., and values are rounded to one place after the decimal point. This table reveals that for some projects we fail to perform any meaningful attribution, while other projects can be segmented nearly perfectly without any additional steps. It will take further research to determine what makes a project easier or harder to segment. This variability can also help explain the differences in accuracy between our validation set experiments and our test set experiments.

Table 6. Summary of Minimal, Average & Maximal Accuracy

Method	Min	Avg.	Max	Min Ba.	Max Ba.
Baseline	0.6%	43.0%	99.7%	12.1%	99.9%
Adj.	1.8%	43.3%	99.7%	14.3%	99.9%
Blo.	0.0%	46.6%	100%	0.0%	100%
Blo.-Adj.	0.0%	45.9%	100%	0.0%	100%

6 Limitations

We acknowledge that our study has some limitations. First, as discussed previously, training data was difficult to obtain and even in cases where we had training data it was often insufficient. Second, while our method allows for both training and attribution at the level of AST nodes, our ground truth was at the line level. Third, there were cases in which our preprocessing techniques failed to successfully identify the ground truth for AST nodes, for which we had to create a separate class. We further acknowledge that there are combinations of techniques and parameterizations that have not been fully explored which could yield additional insights for improved attribution.

7 Future Work

We believe the most essential future work lies in the realm of data collection and processing. Difficulties with data are among the primary limiting factors for performance and application of the techniques presented in this paper. We believe it is essential to continue to build substantive collaborative data sets. It is also important to continue to refine our expectations and definitions of collaboration and authorship. This work would also be enhanced by the development of a parser capable of authorship tagging, as well as further advancements in identifying ground truth from commit histories. Given our observations about the nature of GitHub accounts belonging to prolific programmers, we believe it is also essential to develop language-agnostic techniques for source code stylometry.

In this work, we introduce an extension to the set of stylometry AST features. We believe that continuing to extend this feature set may further improve accuracy, both for this work and for the single programmer attribution problem.

In this work, we observe that subtree size is an important contributor to accuracy. Thus, finding ways to boost the accuracy of the more common, smaller subtrees is the best way to improve upon the techniques presented here. We also observed that varying methods and parameters based on the size of the subtree may yield much higher accuracy. While we provided an initial exploration of such combinations, further evaluation may yield better outcomes. There may also be additional ways to leverage prediction of authorship changes to further enhance attribution accuracy.

For our adjustment technique, we only look to parent and child nodes to influence our prediction. We believe that looking to other surrounding nodes, including siblings, grandparents, and grandchildren, may allow this technique to perform even better. Similarly, for our experiments to predict authorship changes we only determine if an AST subtree has a different author than its parent. We believe that a similar method could be applied to determine change in authorship between sibling subtrees (subtrees which share a parent node) but leave this to future work. If successful, adjustment and blocking mechanisms could be developed for this axis, and it may be possible to combine the two directions for even greater accuracy gains.

During our result analysis, we observed a wide range of accuracy values between projects, both before and after applying adjustment and blocking. While in-depth analysis of the reasons for this was beyond the scope of this work, it would make for a natural follow up which would have both forensic and privacy preserving implications. Similarly, questions regarding the effects of the number of collaborators and amount of training data remain and would require a larger dataset than gathered here. We also observed tension between accuracy for small subtrees and accuracy for larger subtrees which is worth further investigation. During parameterization, we also observed a tension between optimizing for a single accuracy metric and improving all metrics. As a result, further investigation of parameterization may yield more useful information.

In this work, our ground truth was obtained at the line level using git blame and we handled multi-authored subtrees by identifying the primary author of the subtree. To maximize the utility of this technique, we believe it will be important to develop tools to assign ground truth at the node level.

Finally, the ability to attribute even small segments of code means that programmer privacy is at great risk. In order to protect the privacy of programmers, it is necessary to develop highly effective and granular programmer obfuscation methods.

8 Conclusion

We have shown that it is possible to perform several segmentation operations on ASTs. We have shown that we can identify changes in authorship between parent and child nodes at over 80% balanced accuracy. In the validation set we can attribute subtrees of at least 25

nodes with accuracy over 80% and at least 33 nodes with accuracy over 90%, while in the test set we can attribute subtrees of at least 33 nodes with accuracy of 70%. Subtrees in this range correspond to segments of code of approximately 4 or 5 lines of code on average. While this level of segmentation may not be sufficient for all circumstances, it is will be for many uses.

While our baseline accuracy for single AST nodes is 20.21% for the validation set and 35.66% for the test set, we present techniques by which we can increase this accuracy to 42.01% and 49.21% respectively. To achieve this, we presented multiple techniques and examined parameterizations and combinations of these techniques. We demonstrate that even in the low information circumstances of AST subtrees we are able to perform attribution and use analysis techniques to partially counter the low accuracy inherent in such data. We also show that the low overall accuracy is due to the abundance of extremely small subtrees which are inherently difficult to attribute.

Being able to perform any level of attribution on samples this small presents a threat to privacy - even small, well distributed contributions do not guarantee protection against malicious attribution. While many open source projects already have segment authorship information available due to being developed in the public eye, there are many other code projects for which this information is not publicly available. This includes not only code written by corporations, governments, and malicious groups, but also code written by non-profit organizations and activist groups. While previous work has shown how to attribute code segments or segment based on the text of the code, here we show how it is possible to use the abstract syntax tree to segment code based on the machine interpretation of the code. While our accuracy is not exceptionally high for small segments, once we reach the equivalent of 4 or more lines of code, the privacy threat is much higher.

While some of our results may give the appearance that it is possible to anonymously contribute smaller patches to code, we would advise caution. Even the accuracy we achieve here may be sufficient to guide an active adversary to a successful attribution, and even more powerful attribution methods are likely to continue to be developed. Additionally, technologies allowing greater granularity for training labels and experimental ground truth may further extend the usefulness of this method.

9 Acknowledgement

We would like to thank DARPA Contract No. FA8750-17-C-0142 and the United States Army Research Laboratory Contract No. W911NF-15-2-0055 for funding this work. We would also like to acknowledge Dr. Richard Harang for providing the initial inspiration for this work.

References

[1] Mohammed Abuhamad, Tamer AbuHmed, Aziz Mohaisen, and DaeHun Nyang. 2018. Large-Scale and Language-Oblivious Code Authorship Identification. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*. ACM, 101–114.

[2] Mohammed Abuhamad, Tamer Abuhmed, DaeHun Nyang, and David Mohaisen. 2020. Multi-χ: Identifying Multiple Authors from Source Code Files. *Proceedings on Privacy Enhancing Technologies* 1 (2020), 17.

[3] Alfred V Aho, Ravi Sethi, and Jeffrey D Ullman. 1986. *Compilers, Principles, Techniques*. Addison wesley.

[4] Navot Akiva and Moshe Koppel. 2012. Identifying distinct components of a multi-author document. In *2012 European Intelligence and Security Informatics Conference*. IEEE, 205–209.

[5] Navot Akiva and Moshe Koppel. 2013. A generic unsupervised method for decomposing multi-author documents. *Journal of the American Society for Information Science and Technology* 64, 11 (2013), 2256–2264.

[6] Steven Burrows. 2010. *Source code authorship attribution*. Ph.D. Dissertation. RMIT University.

[7] Steven Burrows and Seyed MM Tahaghoghi. 2007. Source code authorship attribution using n-grams. In *Proceedings of the Twelth Australasian Document Computing Symposium, Melbourne, Australia, RMIT University*. Citeseer, 32–39.

[8] Steven Burrows, Alexandra L Uitdenbogerd, and Andrew Turpin. 2009. Application of information retrieval techniques for source code authorship attribution. In *Database Systems for Advanced Applications*. Springer, 699–713.

[9] Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt. 2015. De-anonymizing programmers via code stylometry. In *24th USENIX Security Symposium (USENIX Security 15)*. 255–270.

[10] Edwin Dauber, Aylin Caliskan, Richard Harang, Gregory Shearer, Michael Weisman, Frederica Nelson, and Rachel Greenstadt. 2019. Git blame who?: Stylistic authorship attribution of small, incomplete source code fragments. *Proceedings on Privacy Enhancing Technologies* 2019, 3 (2019), 389–408.

[11] Haibiao Ding and Mansur H Samadzadeh. 2004. Extraction of Java program fingerprints for software authorship identification. *Journal of Systems and Software* 72, 1 (2004), 49–57.

[12] David Fifield, Torbjørn Follan, and Emil Lunde. 2015. Unsupervised authorship attribution. *arXiv preprint arXiv:1503.07613* (2015).

[13] Georgia Frantzeskou, Stephen MacDonell, Efstathios Stamatatos, and Stefanos Gritzalis. 2008. Examining the significance of high-level programming features in source code author classification. *Journal of Systems and Software* 81, 3 (2008), 447–460.

[14] Georgia Frantzeskou, Efstathios Stamatatos, Stefanos Gritzalis, Carole E Chaski, and Blake Stephen Howald. 2007. Identifying authorship by byte-level n-grams: The source code author profile (scap) method. *International Journal of Digital Evidence* 6, 1 (2007), 1–18.

[15] Georgia Frantzeskou, Efstathios Stamatatos, Stefanos Gritzalis, and Sokratis Katsikas. 2006. Effective identification of source code authors using byte-level information. In *Proceedings of the 28th international conference on Software engineering*. ACM, 893–896.

[16] Moshe Koppel, Navot Akiva, Idan Dershowitz, and Nachum Dershowitz. 2011. Unsupervised decomposition of a document into authorial components. In *Proceedings of the 49th Annual Meeting of the Association for Computational Linguistics: Human Language Technologies-Volume 1*. Association for Computational Linguistics, 1356–1364.

[17] Olaf Leßenich, Janet Siegmund, Sven Apel, Christian Kästner, and Claus Hunsen. 2018. Indicators for merge conflicts in the wild: survey and empirical study. *Automated Software Engineering* 25, 2 (2018), 279–313.

[18] Stephen G MacDonell, Andrew R Gray, Grant MacLennan, and Philip J Sallis. 1999. Software forensics for discriminating between program authors using case-based reasoning, feedforward neural networks and multiple discriminant analysis. In *Neural Information Processing, 1999. Proceedings. ICONIP'99. 6th International Conference on*, Vol. 1. IEEE, 66–71.

[19] Tomas Mikolov, Kai Chen, Greg Corrado, and Jeffrey Dean. 2013. Efficient estimation of word representations in vector space. *arXiv preprint arXiv:1301.3781* (2013).

[20] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. 2014. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *Proc. of IEEE Symposium on Security and Privacy (S&P)*.

between classes and projects. For these experiments, we test thresholds of 0, 0.15, 0.20, 0.25, 0.30, 0.95, and 1, labeled in figures as 0, 15, 20, 25, 30, 95, and 100 respectively. Figures 12, 13, and 14 show these results.

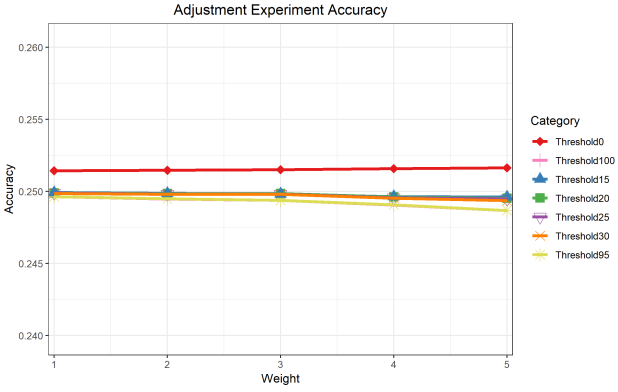

Fig. 12. This graph shows the overall accuracy results for the adjustment experiments with variable weights, represented as percent accuracy vs. adjustment weight.

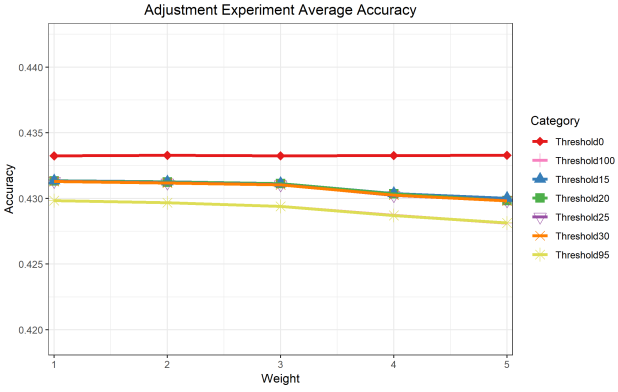

Fig. 13. This graph shows the average per project accuracy results for the adjustment experiments with variable weights, represented as percent accuracy vs. adjustment weight.

Appendix A Parameterization

In this appendix, we examine the experiments used to select our parameters. These experiments informed our selection of parameters, and may serve as guidance for future development and application.

A.1 Adjustment Experiments

In these experiments we use the entire validation set. Note that the amount of training data is widely variable

We note here that low thresholds and weights give the best accuracy and balanced accuracy. A threshold of 0 and weight of 0.02 yields overall accuracy of 25.15%, average project accuracy of 43.33%, and balanced accuracy of 64.56%. Recall that the baseline results were overall accuracy of 24.97%, average per project accuracy 42.98%, and balanced accuracy of 63.42%. We also conducted these experiments with our validation subset, and found that while a threshold of 0 was still best, the effect of weight was negligible. For the same parameters

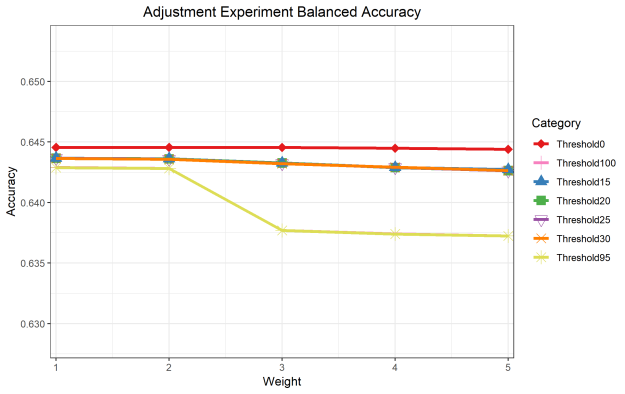

Fig. 14. This graph shows the balanced accuracy results for the adjustment experiments with variable weights, represented as percent accuracy vs. adjustment weight.

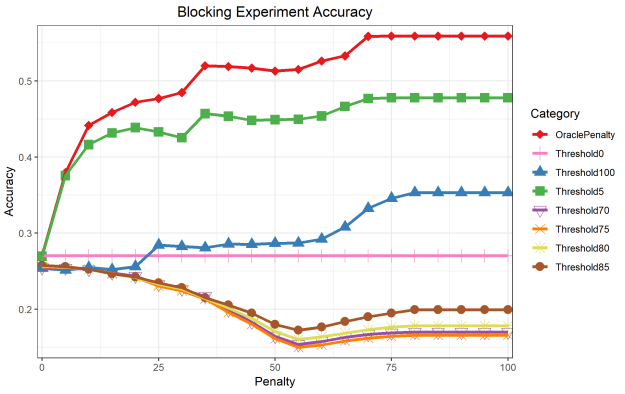

Fig. 15. This graph shows the overall accuracy results for the blocking experiments with variable penalties, represented as percent accuracy vs. blocking penalty.

above, we achieved overall accuracy of 32.08%, average accuracy of 48.98%, and balanced accuracy of 71.97%.

A.2 Blocking Experiments

In these experiments we use the entire validation set. We note in advance that the amount of training data is widely variable between classes and projects. We test thresholds of 0, 0.05, 0.70, 0.75, 0.80, 0.85, and 1, labeled in the figures as 0, 5, 70, 75, 80, 85, and 100 respectively. To further evaluate the blocking technique, we constructed a version which replaces the thresholding with an oracle that identifies change in authorship. Note that the oracle does not know the identities of the authors, only whether the subtree rooted at the child has the same author as the subtree rooted at the parent. Figures 15, 16, and 17 show the results of these experiments.

From the blocking experiments we can observe a few trends. First, we see that the oracle always leads to better accuracy and balanced accuracy than any of our thresholds for detecting changes. The effects of adding a penalty towards consecutive blocks belonging to the same author level out by a penalty of 0.80 for all of our experiments. Balanced accuracy is best with either no or low penalty and high threshold, with a threshold of 0.85 performing the best. Average accuracy is best with a threshold of 0 followed by 0.05. While accuracy with the oracle generally increases with the penalty, using a threshold we do best with no penalty. For total accuracy, we are best served with a threshold of 0.05 followed by a threshold of 1, and for these cases as well as using the oracle accuracy generally improves with increased penalty.

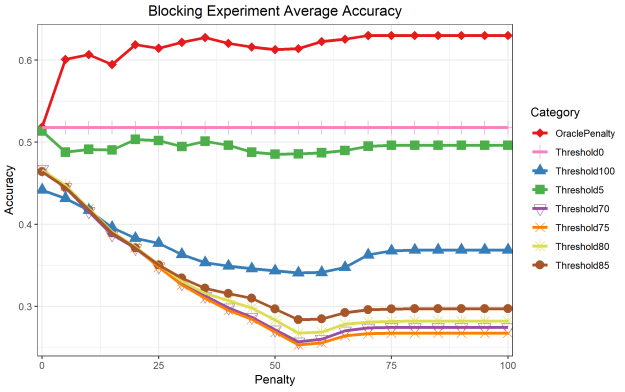

Fig. 16. This graph shows the average per project accuracy results for the blocking experiments with variable penalties, represented as percent accuracy vs. blocking penalty.

While for adjustment we were able to identify superior parameters, for blocking we are forced to select different choices depending on our objective. For overall accuracy, we would select a threshold of 0.05 and a penalty of at least 0.75. For average accuracy, we would select either a threshold of 0 or of 0.05. For a threshold of 0 penalty does not matter, while for 0.05 we would select a penalty of 0. For balanced accuracy, we would select a threshold of 0.85 and no penalty. Due to the wide range of possible threshold and penalty selections, we have chosen to summarize key points from the graphs in Table 7. Recall that the results for attributing AST subtrees individually were overall accuracy of 24.97%, average project accuracy 42.98%, and balanced accuracy of 63.42%. Applying the criteria that all metrics should improve, we would decide that a threshold of .85 and no penalty is the optimal parameterization. This set of parameters yields overall accuracy of 25.67%, average accuracy of 46.60%, and balanced accuracy of 66.36%.

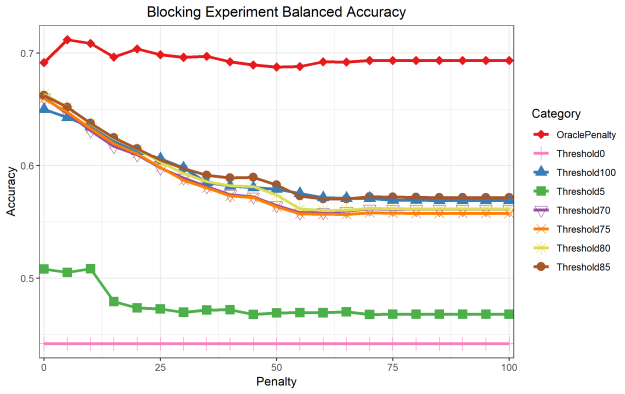

Fig. 17. This graph shows the balanced accuracy results for the blocking experiments with variable penalties, represented as percent accuracy vs. blocking penalty.

Table 7. Summary of Blocking Accuracy

Threshold	Penalty	Total	Average	Balanced
0	0	27.03%	51.79%	44.17%
0.05	0	26.95%	51.37%	50.81%
0.05	0.75	47.78%	49.62%	46.79%
0.85	0	25.67%	46.60%	66.36%
Oracle	0.05	37.96%	60.10%	71.20%
Oracle	0.80	55.89%	63.00%	69.34%

We repeated the analysis on our selected subset of data. The patterns are similar to the previous figures, with higher accuracy values. In the interest of space, we omit graphs in favor of the summary provided in Table 8. We note that the optimal parameterizations have shifted for this subset but will continue to report based on the parameterizations previously identified.

Table 8. Summary of Blocking Accuracy on Data Subset

Threshold	Penalty	Total	Average	Balanced
0	0	36.25%	60.75%	47.40%
0.05	0	36.20%	60.23%	57.06%
0.05	0.75	45.25%	48.58%	51.83%
0.85	0	33.27%	53.31%	74.27%
Oracle	0.05	41.73%	65.54%	78.51%
Oracle	0.80	59.90%	69.10%	77.78%

A.3 Blocking-Adjustment Experiments

Figures 18 and 19 show the accuracy of our experiments with the blocking-adjustment technique while Figure 20 shows the balanced accuracy. Table 9 summarizes key blocking threshold, penalty, and weight combinations. In all cases, the best adjustment threshold is 0. Recall that the results for attributing AST subtrees individually were overall accuracy of 24.97%, average project accuracy 42.98%, and balanced accuracy of 63.42%. Setting our blocking threshold to 0, the blocking penalty to 0, and the adjustment weight to .02 results in overall accuracy of 28.30%, average accuracy of 45.92%, and balanced accuracy of 64.84%.

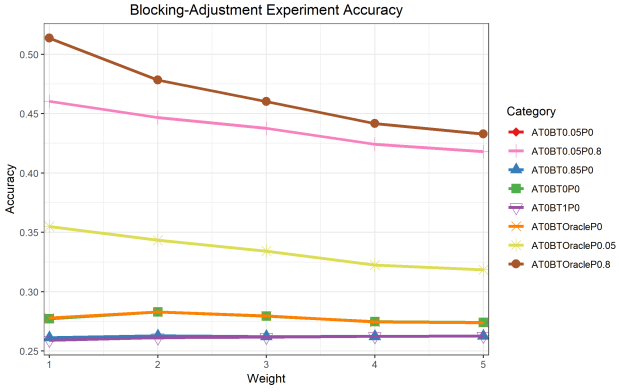

Fig. 18. This graph shows overall accuracy for the blocking-adjustment experiments, represented as percent accuracy vs. adjustment weight. AT represents adjustment threshold, BT represents blocking threshold, and P represents blocking penalty.

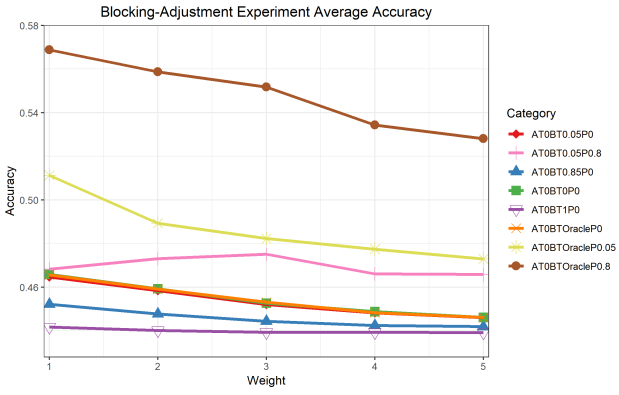

Fig. 19. This graph shows the average per project accuracy for blocking-adjustment, represented as percent accuracy vs. adjustment weight. AT, BT, and P are defined as in Figure 18.

We repeated this analysis on the validation subset. The patterns are similar to those shown previously. We summarize notable results in Table 10.

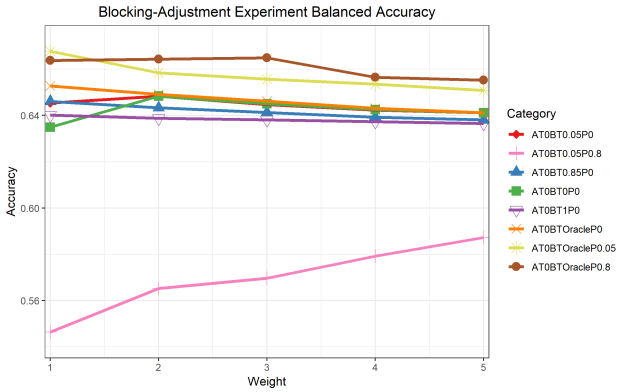

Fig. 20. This graph shows balanced accuracy for the blocking-adjustment experiments, represented as percent accuracy vs. adjustment weight. AT, BT, and P are defined as in Figure 18.

Table 9. Summary of Blocking-Adjustment Accuracy

Thresh.	Penalty	Weight	Total	Avg.	Bal.
0	0	.02	28.30%	45.92%	64.84%
0.05	0.8	.01	46.03%	46.83%	54.64%
0.05	0.8	.03	43.76%	47.51%	56.97%
Oracle	0.05	.01	35.50%	51.13%	66.77%
Oracle	0.80	.01	51.36%	56.88%	66.37%

Table 10. Summary of Blocking-Adjustment Subset Experiments

Thresh.	Penalty	Weight	Total	Avg.	Bal.
0	0	.02	36.79%	52.39%	72.80%
0.05	0.8	.01	46.81%	47.13%	59.49%
0.05	0.8	.03	49.15%	49.84%	62.76%
Oracle	0.05	.01	39.88%	54.69%	73.60%
Oracle	0.80	.01	55.84%	62.57%	74.47%