Source Code Stylometry and Authorship Attribution for Open Source

by

Daniel Watson

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2019

Author's Declaration

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

Public software repositories such as GitHub make transparent the development history of an open source software system. Source code commits, discussions about new features and bugs, and code reviews are stored and carefully attributed to the appropriate developers. However, sometimes governments may seek to analyze these repositories, to identify citizens who contribute to projects they disapprove of, such as those involving cryptography or social media. While developers who seek anonymity may contribute under assumed identities, their body of public work may be characteristic enough to betray who they really are. The ability to contribute anonymously to public bodies of knowledge is extremely important to the future of technological and intellectual freedoms. Just as in security hacking, the only way to protect vulnerable individuals is by demonstrating the means and strength of available attacks so that those concerned may know of the need and develop the means to protect themselves.

In this work, we present a method to de-anonymize source code contributors based on the authors' intrinsic programming style. First, we present a partial replication study wherein we attempt to de-anonymize a large number of entries into the Google Code Jam competition. We base our approach on Caliskan-Islam et al. 2015[8], but with modifications to the feature set and modelling strategy for scalability and feature-selection robustness. We did not achieve 0.98 F1 achieved in this prior work, but managed a still reasonable 0.71 F1 under identical experimental conditions, and a 0.88 F1 given more data from the same set.

Second, we present an exploratory study focused on de-anonymizing programmers who have contributed to a repository, using other commits from the same repository as training data. We train random-forest classifiers using programmer data collected from 37 medium to large open-source repositories. Given a choice between active developers in a project, we were able to correctly determine authorship of a given function about 75% of the time, without the use of identifying meta-data or comments. We were also able to correctly validate a contributor as the author of a questioned function with 80% recall and 65% precision. This exploratory study provides empirical support for our approach.

Finally, we present the results of a similar, but more difficult study wherein we attempt de-anonymize a repository in the same manner, but without using the target repository as training data. To do this, we gather as much training data as possible from the repository's contributors through the Github API. We evaluate our technique over 3 repositories: Bitcoin, Ethereum (crypto-currencies) and TrinityCore (a game engine). Our results in this experiment starkly contrast our results in the intra-repository study showing accuracies of 35% for Bitcoin, 22% for Ethereum, and 21% for TrinityCore which had candidate set

sizes of 6, 5, and 7 respectively. Our results indicate that we can do somewhat better than random guessing, even under difficult experimental conditions, but they also indicate some fundamental issues with the state of the art of Code Stylometry. In this work we present our methodology, results, and some comments on past empirical studies, the difficulties we faced, and likely hurdles for future work in the area.

Acknowledgements

I would like to thank Michael W. Godfrey for being my advisor and mentor during this process, as well as Patrick Juola for giving me my first exposure to Stylometry and encouraging me to pursue my graduate studies. I must also acknowledge Azoacha Ntonghanwah Forcheh and Jason P'ng for their diligent work developing CoderID. This work would not have been possible without their contributions.

Dedication

This work is dedicated to those who have lost their lives or freedoms for speaking truth or for working selflessly to make the truth available to others.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Anonymity and privacy are prized resources in the modern world, and are increasingly hard to achieve in endeavors involving public contributions of any sort. As information technology grows in influence, governments throughout the world will often do what they can to retain some control over the information their citizens consume and produce. Dedicated groups of volunteer engineers work to develop open source tools like Tor [11] and Shadowsocks [4] to enable users to browse the web anonymously and without restriction. However, developing and distributing such tools can be a dangerous enterprise in nations with strict laws against such circumvention.

In 2014, Xu Dong, a Chinese internet rights activist and software engineer was arrested in Beijing on charges of "creating a disturbance"[3]. He, among others, had successfully developed software that allowed users to bypass the so-called "Great Firewall of China". On October 19th, 2014 Dong tweeted about the project through an anonymous Twitter account, attaching a guide to allow users to install the software and gain access to the internet unfiltered, much to the dismay of authorities. Days later, on November 4th, he was arrested and the authorities seized his computer equipment. According to his friends[3], those involved were pressured by the authorities to not publicise his arrest, so little information beyond this is available.

Given the repressive media climate in many parts of the world, one may wonder how many more such cases have likely occurred, but have not been reported in the media. More importantly, we might ask, what does this mean for the future of open-source software when writing code can be a crime? In the near future, more contributors may seek to keep their identities anonymous, so as not to be tied to a potentially "criminal" piece of software down the line. An important question remains open: How anonymous can one be when

working a large, complex system? Do programmers leave detectable "fingerprints" on their work like a painter leaves distinctive brush strokes on their canvas?

There are several example systems that are partly comprised of anonymously written source code; of these, Bitcoin is perhaps the best known. Bitcoin was released as free and open-source software in 2009 following the release of a white paper [22] that laid out the fundamental concept of block-chain based proof-of-work as electronic currency. The creator, who used the pseudonym "Satoshi Nakamoto", has remained incognito ever since and their Bitcoin wallet became inactive in 2010 after mining approximately one million Bitcoins, worth about 3.7 billion USD today. Many individuals have claimed to be Nakamoto, and others have been suggested to be the enigmatic engineer, but all such claims have failed to be substantiated and their true identity remains unknown, amid much speculation. One such speculative claim is that Nakamoto is not one individual, but a small group of talented academics who devised the idea, implemented the system, and wrote the white paper.

Technologies capable of analyzing and modelling programming style are an emerging threat to those who wish to remain anonymous. Just as in security hacking, the only way to protect those affected is, counter-intuitively to publicize these techniques and make them available to all. Government labs around the world are already working on such tools and their operators would rather see them kept secret. In presenting these methodologies to the world, further tools may be developed and evaluated which reduce the authorial fingerprints present in source code and thereby allow anonymous authors make their work more secure from such attacks.

Chapter 2

Motivation

This work explores an approach that could be used to identify the authors of anonymously-written source code; the existence of such techniques threatens the anonymity of developers who work on projects that some governments may consider sensitive or even illegal. We build upon prior work in program stylometry towards the goal of demonstrating that an effective and practical approach to identifying and verifying source code authorship is well within the reach of researchers and governments alike. We note that previous stylometric studies have focused mainly on small and/or uniquely contrived datasets. Our work is novel in that we seek to evaluate the effectiveness of applying robust ensemble learning to real world data pulled from public GitHub repositories. This data source provides a rich real-world space in which to explore the potential of these techniques to be applied more broadly, in more situations, and to more developers than has previously been demonstrated.

The problem of authorship attribution in general can be broken down into 3 sub-problems: attribution, authentication, and attribution-authentication. Attribution and authentication ask fundamentally different questions and require different approaches. As the name implies, attribution-authentication problems involve elements of both problems, and are more complex than either.

In the **attribution** problem, we are given code that is unattributed, either anonymously written or improperly signed. Then, from a list of candidate authors for whom we have some sample code, we must determine the most likely author. In principle, the candidate set may comprise only two authors or it may include hundreds or even thousands of candidates. However, a typical assumption is that the true author of the work does indeed appear in our candidate set. In modelling terms, we may consider this a k-class classification problem, and is therefore amenable to the application of well-studied classification strategies, with

special consideration given to ensuring the modelling strategy fits the task.

In the **authentication** problem, we have a code sample and a single individual for whom we have an independent sample. We must determine whether the single candidate is indeed the author of the questioned work. In some ways, this is a trickier problem than the attribution problem, as we have true training data only for the "positive" examples. Certainly, we have an enormous amount of "negative" data in the form of everything in the domain not written by the questioned author, but this has caveats. Choosing a relevant set of negative examples is often non-trivial and many classic machine-learning models struggle to learn from highly imbalanced training sets.

Then of course, there is the worst case scenario, which happens to be the most common scenario encountered in forensic practice, the **attribution-authentication** problem. This is the case where we have many authors in our candidate set, but we have no guarantee that the true author of the questioned work exists in the candidate set at all. This task is generally twofold. First, we must identify the most likely candidate, then we must attempt to validate the candidate's authorship. This is clearly the most difficult of the three principle tasks, as it combines the difficulties of the two aforementioned tasks. If we can carry out this task with reasonable accuracy, we will have demonstrated that existing techniques generalize well to large datasets, posing a substantial threat to anonymity in open-source contributions.

As Caliskan Islam et al.[8] have shown, these tasks can be carried out quite effectively using standard natural language modelling techniques on fairly large sets of authors. This pioneering work demonstrates our capability to model programming style with relatively simple feature sets and model designs. However, previous efforts have not attempted perform this task at on a set real, collaborative software projects, under constraints comparable to those faced by practitioners in forensic settings. Simply put, our study attempts to bring this technology closer to practise by evaluating its capabilities on real life data under such constraints.

Virtually all modern software development uses a version control system such as git to manage the evolution of the source code; however, the use of such a system can complicate the authorship attribution problem as, unlike the Google Code Jam data set or natural language sources like books and articles, we cannot consider the final work to be the sum total of code written for the project. Code all too often is committed, obsolesces, and is then removed from the project. Further, code that does remain is often edited by multiple individuals throughout the evolution of the project. This requires us to employ repository mining techniques to retrieve a longitudinal view of the project's code history to build our models with as much data as possible.

Another problem created by mining information from Git repositories is that commits come in many shapes and sizes, not all of which provide useful data for attribution purposes. A commit that fixes a bug may only include a small change of a couple of lines, containing little stylistic information. Worse yet, refactoring commits may involve large sections of "added" code that are actually merely relocated from other areas of the project, often written by other people. Although relatively simple methods [16] can be used to filter such commits for some repositories, our early experiments indicated that such techniques did not generalize well to the wide variety of project cultures on Github. This makes an effective, general, commit classification approach highly desirable to improve the quality of authorship oriented data sources.

Analysis of programming style, such as that necessary for attribution/authentication problems, has potential to become an interesting source of insight in other software research problems. One such area of potential interest is the effect that stylistic diversity has on the generation of bugs, the effectiveness of code review, and other software engineering team dynamics questions. Liang et al. [20] have commented on the impact of diversity and cohesion in engineering settings. They found that knowledge diversity was positively correlated with team performance. We believe that stylometric techniques could have applications in characterising programmer style, tendencies, behavior, and skills, which could answer interesting questions in the more social topics in Software Engineering research.

One research topic where stylometric analyses not unlike ours have already found application is in identity merging. Kouters et al. [18] apply Latent Syntactic analysis of source code to merge identities in software projects. Many Software Research projects are hindered by data quality related problems, such as when project contributors make commits under multiple aliases. Indeed, in our own experiments, we observed such cases on Github, where an author would have some commits not attributed to them for one reason or another. This could be because the project was moved to Github after a long period of development, so older commits were not linked with the Github profiles of committers. In every case of this that we observed, the authors only have one alias under which they work consistently and others they use only once or sporadically. This does not pose much of an obstacle to us, but to other research endeavors, identity linking can be very important. Goeminne et al.[13] present a comparison of practical identity linking approaches. They do not include stylometric analysis in their comparison, but this work underscores the desire for good identity merging and there is clear potential for stylometric data to inform identity merging algorithms.

2.1 Related Work

Some of the earliest publications in the area of Code Stylometry were produced by Oman and Cook in the late 1980s; they extend the earlier work of Dakin et al. who proposed a code stylometry as a practice and suggest a formatting-based approach, but did not validate it against real-world data. With their 1988 work[25], Oman and Cook began to solidify these ideas. This study focuses on applications of Code Stylometry to program comprehension and hence, productivity. They had not yet ventured into Stylometry based authorship attribution. Oman and Cook first propose the use of software complexity metrics to group a collection of programs by author. Despite the intuitive nature of such metrics, they found in early experiments that typographic information was far more effective than complexity-based metrics. They extract said typographic information by simply checking for the presence or absence of a typographic sequence in each program. They carry out cluster analysis on the resulting bit vectors and found that despite simplistic metrics that generate simple feature spaces, they were able to cluster their samples surprisingly well. This work, and Oman and Cook's work[25][24] more generally is considered seminal in the field of code stylometry.

The most relevant recent work in this area was done by Aylin Caliskan-Islam et al. in 2015 and Dauber et al. in 2018, applying the same approach. Caliskan-Islam et al. present a technique for source code attribution that is based on a hybrid lexical-syntactic analysis of C++ programs [8]; they tested their approach against the Google Code Jam (GCJ) dataset, a large collection of short programs by hundreds of authors. The GCJ dataset represents an idealized case for automated stylometry: multiple authors writing the "same program", with each author contributing several such small programs. The authors achieve a 94% attribution rate out of 1600 authors.[1] However, the GCJ dataset is also unrealistic for real-world author attribution tasks, where training data is less comprehensive and not systematically organized. Furthermore, the feature set employed by the authors is non-trivial to obfuscate, a critical concern for adversarial stylometry.

For their syntactic features, the authors use a fuzzy parser, Joern [2], to generate abstract syntax trees (ASTs) for incomplete code snippets, such as are found in version control system commits. They employ a "shotgun" approach of diligently assembling a long list of features that have been proposed in prior work; they then pare down this list through a feature selection protocol. They employ a relatively standard entropy-based feature selection procedure and then test the stability of the feature set, essentially by

[1] Comparatively, it would be practically inconceivable to achieve a similar author attribution success rate using NLP stylometry over a large body of English language prose works.

bagging authors and ensuring the resulting feature set is relatively stable across groups. They employed a Random-Forest classifier, as this technique has been demonstrated to be effective in multi-class problems. The authors also examined the code of a number of GCJ participants who submitted programs over several years to see how consistent their style is over time; they found that author style does not change over time sufficiently to confuse their classifier.

A related group of researchers has made inroads into practical de-anonymization of Git repositories. In 2018 Dauber et al.[10] demonstrate an approach for attribution of Git repositories. They collect contiguous program fragments using Git Blame and analyze them with a similar analytical approach they employed in their 2015 work. In this work, they demonstrate a method of extracting features from source code fragments obtained from git commit logs. They obtained their dataset by mining 1,649 repositories and dropping authors with insufficient data to arrive at a sample of 104 programmers collected from some unknown subset of the repositories. They evaluate their models through 10-fold cross validation as in Caliskan-Islam 2015. They found that the Calisk-Islam et al. approach is much less effective on this data than on Google Code Jam data, reporting an attribution rate of around 50% from a 104 programmer dataset with at least 150 samples of at least 1 line in length. They found that they were able to improve classification rates from around 50% to to about 90% by grouping samples in batches of 30 or so, making each sample richer, but also requiring more code for any given number of samples.

Other recent work has focused on measuring and improving the scalability of authorship attribution techniques, more-so in natural language than in source-code. Narayanan et al. found their approach was able to correctly identify the author of 20% of samples from 100,000 unique authors gathered from blogs [23]. They employ a fairly standard combination of lexical and syntactic features. They also evaluate simple and complex classification algorithms including nearest-neighbor, regularized least squares classification, and Support-Vector Machines on their data. They found that normalizing the feature matrix before training can improve classification rates substantially. This work illustrates that relatively simple models can be surprisingly strong in attribution tasks, and that automated stylometry in general can be very effective at large tasks when applied with care. They found that regularizing their feature matrix by enforcing row-norm=1, feature mean=0, feature variance=1, and feature-mean-nonzero constraints on their feature matrices made a significant difference to their classification scores. This study illustrates that relatively mature classification methods can do surprisingly well on large attribution problems. However, as in previous studies the authors made no attempt to prevent the type of cross-sample contamination mentioned previously, how effectively this process will generalize to attributing content of unseen software projects remains to be seen.

These works demonstrate that code stylometry can be applied to Git samples with some efficacy; however, none of these projects have tested their approaches under realistic conditions. They gather data from one or more repositories for each author in their evaluation set, then cross-validate with bootstrapped samples. Although this approach follows standard machine learning practices, it fails to account for important constraints that would typically be faced by practitioners. For example, when we are faced with attributing a body of work, such as the commit history of a an author to a repository, we will most likely not have any history in the questioned repository from any of the authors in the candidate set. Why would an author make anonymous contributions to a project where they have also contributed under their name? When it comes to evaluating a model to demonstrate efficacy, we must do what we can to avoid *contamination*. Contamination is usually defined as somehow leaking information about the test set into the training set. This can happen in subtle ways and there is no definitive test for it. In these cases, the authors are splitting up their candidate set's samples randomly for cross validation, meaning that samples from the same author and same repository occur in both the training and test sets. Clearly, samples from the same author and project will share a great deal of similarities, especially when tokens are used a features. To their credit, Dauber et al. mention this limitation of their work in the paper: "[A] limitation (of our work) is that many authors only contributed to a single collaborative project. As a result, code samples are not from as different environments as we may expect in real application. A smaller scale study which enforces a project split for training and testing would help better understand the effects of this issue." One of the contributions of this work is to carry out just such a study.

Another related issue not addressed by Dauber et al. 2018 is that of subject-matter driven variance of features. An analogously flawed task in natural language stylometry would be to "determine if a given chapter was written by Agatha Christie or H.G. Wells." Clearly, one *could* provision an interesting model to solve the problem using invariant features like stop word or punctuation frequency, but an important question is whether one would *need* to. Clearly, a more trivial model based on simple term frequency would do the job as sci-fi books and mystery novels tend to have very different subject matter. Such is the case with source code. It is easy to distinguish between the work of two programmers when one works solely on game engines and another works solely on crypto-currencies. In Dauber et al., the dataset consists of 106 programmers with undisclosed niches, but it is likely that most do not have very similar repertoires, as open source projects are diverse and plentiful. Further, in an envisionable practical case where we would like to de-anonymize some portion of repository's history, our candidate set is likely to contain only authors whose work histories are composed mostly of work in a similar domain. If

the repository in question is an encryption tool, any candidate with no demonstrated experience in encryption can be easily ruled out by simple reasoning on the experimenter's part.

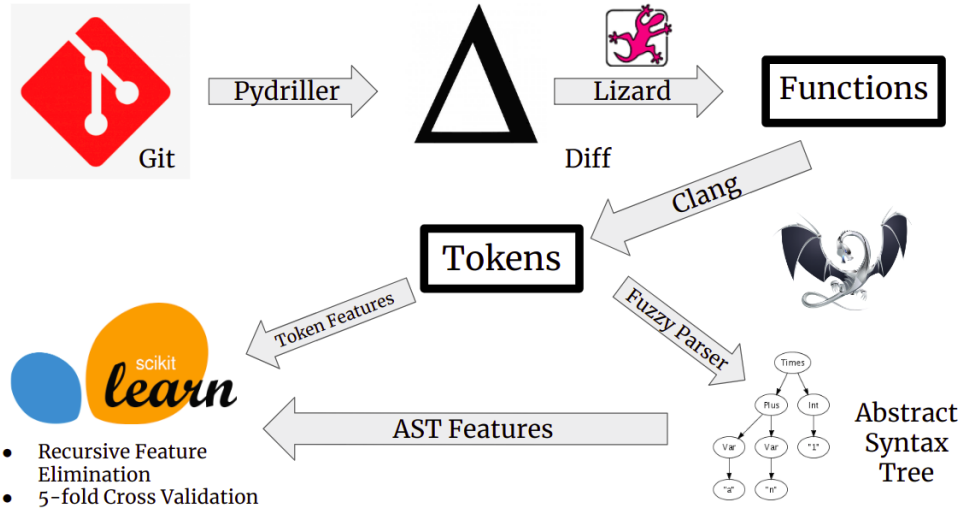
This is not to say that previous work in Code Stylometry presents invalid conclusions per se, but there is work to be done before this practice can by considered empirically validated for use in practical forensic applications. We must first prove the efficacy of stylometry-based authorship attribution in cases where our candidate set consists of authors with identical subject-matter repertoires, then we must prove that we can de-anonymize the work of authors contributing to projects outside of our training domain. Only then can we make a sound case that Code Stylometry is ready to be a tool in the arsenal of forensic practitioners.

Chapter 3

Methodology

Broadly speaking, we approach the problems of authorship attribution as classic machine learning problems with large, complex, and noisy sources of data. This data source poses unique challenges of data collection and modelling. A broad overview of our process structure can be found in Figure 3.1.

Figure 3.1: Our Process



3.1 Problem Formulation

As a starting point, we formulate two problems in a small, somewhat restricted context to act as a proof of concept for other applications. Given a repository with many active authors, we would like to build a strong model of authorship. For this first problem, we attempt to predict the author of an unseen code sample written by one of the repository's contributors. This formulation of the attribution problem is a closed-world, multi-class classification problem. That is, given an unseen code sample and a specific candidate author, we will attempt to accept or reject this new code being written by the candidate. This formulation of the authentication problem is a closed world binary classification problem. We first build and evaluate our models on this sort of problem to hone our techniques and prove out our data collection and training strategy. This problem is somewhat easier to solve than the problems that a forensic practitioner would encounter in the course of an investigation, but the closed world nature of this task makes it a good proving ground where we can generate meaningful and replicable results, before we move on to more difficult problems.

We will also propose and evaluate strategies for approaching more realistic attribution tasks. Using the knowledge gained from our simple intra-repository de-anonymization efforts, we will attempt to devise strong de-anonymization techniques for inter-repository attribution/validation. In the inter-repository attribution problem, we enforce a repository split between our training data and our target repository. That is, given a target repository and a candidate set, we may train on whatever we want except for content from the target repository. This more closely parallels a real scenario where we have a piece of anonymously written software completely unattributed and some candidate set and we would like to know who from the candidate set likely contributed to the project. This is a difficult problem to explore in its entirety, so we focus on a restricted problem that adheres to the repository split criterion. We evaluate our models over a target repository and use for a training set as many program samples as we can from the known contributors to the target repository. Even this approach leaves some aspects to be desired for a true test of Code Stylometry's efficacy in Authorship Attribution. For one, our candidate set consists of authors known to have contributed to the target repository. In practice, this is almost never the case; we would typically be interested in answering whether any of the candidates contributed at all, with outright negatives available as an output. Also, the target repository was written with no intention of hiding the identities of the authors, giving us no read on how difficult it is to detect an author who is trying to conceal their programming style. Both of these issues however, would be difficult to address at this stage in any reasonable empirical experiment, so we limit our problem to one that lends itself to an experimental setup. The

next stage in Code Stylometry research will be to begin addressing these issues.

3.2 Research Questions

The overall goal of this work is to adapt and improve methods demonstrated in prior work bring the state of the art of code stylometry closer to a reliable, practical tool capable of giving good answers to real world problems.

RQ1: *Can the methods presented Caliskan-Islam[8] be modified to to be applicable to large Git repositories?* In the 2015 study, the authors train their models on submissions to the Google Code Jam competition. While this is an excellent test-bed dataset, the format of the data present is quite different from that which we would find in Git repositories. Therefore, we must devise a mining approach that produces data suitable for such tasks, as well as a feature selection and training scheme that accounts for the variety of contexts (projects) in which we find source code on Github. Also, very large repositories pose a serious threat to experimental feasibility as they can take a very long time to mine and would produce absolutely enormous feature matrices were we to directly utilize Caliskan-Islam's feature sets, so we must also engineer our approach to be more scalable.

RQ2: *How well would our modified approach work on the Google Code Jam dataset?* Since we are proposing only a new mining strategy and more light-weight feature extraction, our new approach should work nearly as well on the original dataset, if not equivalently. This will be good verification that our more light-weight analysis does not cost us too much in classifier performance.

RQ3: *Can we improve multiclass classification rates by training in a 1-vs-all manner?* Other authors [17] have demonstrated that an m-class classification problem can be approached by building m binary logistic regression models and considering the class with the maximum predicted likelihood as the output class. Can we use the same 1-vs-all modeling approach as in RQ3 to improve our attribution rates? In this way, we can select a unique feature set for each author that best differentiates them from others, potentially improving scores in the attribution and validation tasks.

RQ4: *Can our procedure successfully identify the author of a sample that comes from a body of work that was not included in our training data?* This follows from the inter-repository problem proposed above. We would like to see how well our model generalizes when the sample to classify comes from a project outside of those we include in our training set.

3.3 Data Collection

When collecting code, we consider complete functions to be considered a sample. In any Natural Language Stylometry task, the granularity with which one conducts their analysis critically impacts the significance and generalizability of the results. If we had chosen to perform our analysis at a line level, this would have given us many samples to work with; however, such small chunks of code offer little in the way of interesting features and in real-world scenarios, a single line of code is unlikely to be of interest. We also could have divided our data at the commit level, but we found commits to be too unstructured for this type of problem. Some advanced code metrics may work well on large commits to one file, but generalizing this to commits containing many changes to different files would have been challenging to do in a consistent way. Doing our analysis at a function level offers many benefits:

- Functions share a common syntax that is defined by the programming language.
- Almost all "interesting" code in a software project falls within a function.
- Functions are often long enough to exhibit many stylistic features.
- Functions can be fuzzy-parsed to generate a partial AST, which we can also analyze.

For these reasons, we chose to carry out our analysis at the function level. For this exploratory study, we chose to gather our data from Github as it is readily accessible and the largest store of open-source software projects at this time. To choose our dataset, we needed repositories with multiple authors who have made substantial contributions. Many of the projects we examined did not satisfy this criterion; many moderately-sized repositories consisted of a large volume of work from only one or two individuals, along with much smaller contributions from many others. For our intra-repository study, we wanted our dataset to be large and diverse, so we started with the 200 most starred repositories on Github that were primarily (>50%) C++. We then mined these repositories for commit data. After extracting all the commit data and parsing out the complete functions from commits, we chose to further limit our dataset to repositories with three or more authors with at least 50 complete functions throughout the commit history to ensure we were in all cases dealing with a multi-class problem without overly-sparse classes. We also limit our training and test sets to only these authors. This left us with 37 repositories with between 3 and 37 authors each in our intra-repository study.

3.3.1 Mining Commit Information

To model an individual's programming style and test our ability to distinguish their work from that of others in a repository, we must first acquire a dataset of code created by an author in question, as well as all the code ever committed to a repository. As discussed above, we chose to model programmer contributions at the function level, which we felt gave a good trade off of "interestingness" to the model versus number of units to train on. As well, we found that it is common for multiple authors to collaborate on a single class which could confound the modelling problem, but it is less likely — but not impossible — for multiple authors to collaborate so on a function.

3.3.2 Feature Extraction

The collected features can be broken down into three types of features by the method of extraction. These types are character-level, token-level, and AST-level features.

character-level features are all extracted directly from the string representation of the function without headers. We gather these by iterating through the function string itself, line-by-line. Most of the features created at this level are normalized by the number of non-whitespace characters in the function string. These features are generated from the lexical properties of the function, or from its style properties. Table 1 provides an overview of these features. These largely include whitespace and other formatting characteristics.

Token-level features are created by tokenizing the function using the `clang` compiler, LLVM's[19]native compiler, and then iterating through these tokens, considering each token a term. There can be several thousand unique token level features generated for any given repository as variable names, class names, and keywords are extracted literally. Table 2 provides an overview of these features.

We consider each token a term and compute its TF-IDF (Term Frequency-Inverse Document Frequency) with each function considered to be a document. For training our models, we build a unique feature set for each classification task. A token is as defined by C++ and the text representation of the token is used. Multi-character operators such as "<<" and "++" are distinguished from one another, but syntactic distinctions like the multiple uses of "∗" are not obeyed. String literals are not taken literally but are instead replaced with a stand-in feature, "`stringLiteral`".

The last category of features that we collect are at the AST-level, i.e. the features that are created from the structure and properties of the Abstract Syntax Tree of the function.Descriptions of these features are provided in Table 3. Note that the feature

Feature	Definition
comments_per_char	Log of the number of comments divided by the number of non-whitespace characters.
ternary_ops_per_char	Log of the number of ternary operators divided by the number of non-whitespace characters.
macros_per_char	Log of the number of preprocessor directives divided by the number of non-whitespace characters.
num_tabs	Log of the number of tab characters divided by the number of non-whitespace characters in the function.
num_spaces	Log of the number of space characters divided by the number of characters in the function.
num_empty_lines	Log of the number of empty lines divided by the number of non-whitespace characters.
avg_line_len	The average/mean length of the lines in the function, excluding white-space character.
std_line_len	The standard deviation of the length of the lines, excluding white-space characters.
whitespace_ratio	The number of white-space characters divided by the number of non-whitespace characters.
new_line_open_brace	Boolean flag representing whether over half of the code-block braces are preceded by a new-line character.
tab_lead_lines	Boolean flag representing whether over half of the indented lines begin with a tab instead of spaces.

Table 3.1: Character-level Features

Feature	**Definition**
Token Unigrams	The tf-idf representation of the tokens.
num_keywords	The number of C++ keywords in the function, as defined by the number of tokens that `clang` classifies as keywords.
num_tokens	The number of tokens generated by `clang` parsing the function.
num_literals	The number of string, character, and numeric literals in the function, as defined by the number of tokens that `clang` classifies as literals.
ctrl_nesting_depth	The maximum control structure nesting depth in the function.
bracket_nesting_depth	The maximum bracket nesting depth in the function.
num_params	The number of parameters of the function.
num_*Keyword*	The number of occurrences in the function of *Keyword*, which is one of ['do', 'if', 'else', 'switch', 'for', 'while'] .

Table 3.2: Token-level Features

names will be the set of the node types in Table 4; in the case of the bigrams, they will be combinations of 2 node types.

Feature	**Definition**
Depths by *Node Type*	The average depth of the sub-trees that have a root node of type *Node Type*.
Average Node Depth	Average depth of the sub-trees in the AST.
Maximum Node Depth	Maximum depth of the sub-trees in the AST, i.e. the depth of the AST itself.
Node Type Unigrams	Term frequencies and TF-IDF represenation of the node types as unigrams.
Node Bigrams	Term frequencies of the node bigrams, defined as when one node is an ancestor of the other.

Table 3.3: AST-level Features

As individual functions alone are not complete modules in C++, `clang` could not be used to accurately create the AST for the individual functions. Joern[2] has been applied previously by Caliskan-Islam et al.[8] as well as GreenStadt et al. [15] and has proven

Function	Declarator	Function-Body
Declaration-Specifiers	Argument-List	Argument
Decl-Specifier	Argument	If-Statement
Else-Statement	Else-If-Statement	For-Statement
While-Statement	Goto-Statement	Continue-Statement
Break-Statement	Switch-Statement	Case-Statement
Try-Statement	Catch-Statement	Return-Statement
Variable	Assignment-Statement	

Table 3.4: AST node types

powerful for this problem. Joern, however, has some drawbacks. The ASTs produced are very detailed and there is a very wide variety of node types to work with. In our early experiments, we found that this led to very large and often sparse feature vectors, leading to huge memory and time requirements for training on very large datasets. To solve this problem, we implemented a more light-weight fuzzy C++ parser to extract only high-level features of the source code and produce smaller trees. The parser uses a relaxed version of the C++ grammar rules created by Dartmouth College [21] together with knowledge of the basic C++ function structure and syntax rules to identify structural code elements. We designed the parser to be as flexible with out-of-context code samples as Joern. Unlike Joern, our parser is focused on structural elements of the input code and does not parse past the statement level. There is room for improvement of our technique in this area as syntactic features below the statement level may provide a wealth of useful information, but to keep our parser light-weight and as general as possible, such details are lost. Given this fact, we should expect to see some reduction in classifier performance in comparison to Caliskan-Islam et al.'s 2015 work, in exchange for the ability to process larger datasets with the available hardware. Due to a lack of context when parsing individual functions, there is often ambiguity when trying to accurately identify different parts of the function. In these cases, the parser defaults to the option that provides the most detailed tree, or will fail to parse if no option is valid given what is known about the surrounding context. In our experiments, less than 10% of functions extracted by lizard could not be parsed and so were excluded.

3.4 Classification

For our modelling, we chose to use a Random-Forest model. Caliskan-Islam et al.[8] and Dauber et al. [10] have had success in authorship attribution tasks with Random-Forest classifiers. The model has few hyperparameters and tends to be relatively insensitive to them in most cases. That is, a small change in a hyperparameter will not result in a substantially different model. Random-Forest models also tend to be resilient to over-fitting as each decision tree in the ensemble is built using a bootstrapped sample, preventing the model, in many practical cases, from fitting to noise. Finally, Random-Forest models offer transparency with respect to the importance of each feature in the feature set. This is an important model property for us as Git repositories are such a complex data source. We employ scikit learn[26] for our modelling as it offers many efficient implementations of data science routines provided through an accessible and well documented Python interface. There are many ways our evaluation techniques could be invalidated by the interconnectedness of software development artifacts, so being able to examine which features our model depends on allows us to validate our work and ensure we avoid pitfalls. This property also enables us to rank the importance of features, which is helpful for recursive feature elimination.

3.4.1 Model Parameter Selection

Random Forest models are, in most cases, insensitive to the changes in the available model (hyper-)parameters, but care must still be taken to ensure that reasonable parameters are used for any given problem domain. Throughout our experimentation, we evaluated our models with several hyper-parameter configurations. What follows is a description of each important hyper-paremeter and how we arrived at this particular configuration.

We employed 300 trees per forest, as we found that even on our largest data sets, using more trees did not improve model performance and increased training time. Using fewer trees reduced model performance on large datasets. In general we found that too few trees leads to a weak model, but there always exists some limit beyond which more trees has no effect on the model.

Most machine learning models work best when the training set is balanced[9]. A highly imbalanced set can lead to the "landslide effect" wherein the model can apportion excess weight to over-represented classes, leading to sub-optimal behaviors such as always predicting the same class. There has been significant empirical[27] work in addressing this issue, as data sets in real life are not usually balanced unless artificially made so. One

method for preventing landslide is letting the class weights vary inversely with the class's prevalence. By "class weights" we refer to the weight given misclassification of a given class when computing the model's error function. Simply put, while training, we penalize the model less for each misclassification of the more common classes so that the net contribution of each class to the error function is equal when the recall on each class is equal. This should, all else being equal, address the class imbalance issue, ensuring that samples representing under-represented classes contribute more strongly to the error function, and therefore, the resulting model. However, we found in our experiments that class weight balancing has a positive effect on our model's recall for authors with small samples, but authors with large training sets were still more prevalent in the models predictions, for reasons we will explore in chapter 4.2.

3.4.2 Feature Selection

As a component of all methods presented here, we employ recursive feature elimination to reduce sparsity. After extracting the primary feature set, we perform dimensionality reduction to reduce the sparsity of our feature matrix, as Caliskan-Islam et al.[6][8] and others[14] have done. One of the advantages of the Random-Forest model is that it gives us an empirical estimate of the importance of each feature after training a classifier. Features that appear closer to the root of more trees contributes to the decision function of a greater number of samples. Therefore, the expected fraction of samples that each feature contributes can be used as an estimate of the relative importance of each feature. This allows to rank each feature by discriminatory power by cross-validating over an authentication set before training a model. Unfortunately, we can rank features only by relative importance, which does not leave us with a proper threshold for reduction purposes. In other words, although only a fraction of our features end up being useful for classifying a given dataset, this fraction, and the optimal set of features varies depending on the repository. To remedy this, we chose to perform feature selection by recursive elimination at the repository level. We set aside an authentication set, 30% of the functions from each repository, to use for this task. At each step, we train and test the model using a randomized, stratified sample from the authentication set. We then rank the features and eliminate the lowest ranked features by some constant fraction. We chose 10% as it struck a good balance between time and granularity. We then train and test the model again using the smaller feature set over a new sample, terminating when the model's out of bag classification rate stops improving. We found that in the majority of cases, our feature set retained between 60–80% of the full feature set. We believe this is the case because the Random-Forest model has a tendency to "self-select", placing minimal weight on unimpor-

tant features, rendering validation error relatively insensitive to irrelevant features in the set.

For all models, k-fold stratified cross validation with $k = 5$ was used for feature selection as well as evaluation. We chose 5-fold cross validation because after many trials, we found that our model's precision and recall were stable across folds to within .05. Additionally, Random-Forest models are computationally expensive to train and given the size of our dataset, we needed to make a trade-off to improve iteration time. Clang, the most time-consuming repository took about 7 hours to mine and about 6 hours to train and test our models. 10-fold cross validation would have taken the training step to almost 9 hours and thus with around 100 repositories to mine and test, our procedure would have taken as long as a week. Cross validation for a text classification problem such as this is especially important, as there is a high possibility of overfitting during the feature selection process. For feature selection, we evaluate the model across each fold and then take the average of the feature importances to choose the worst 10% to drop for the next iteration. This reduced the probability of selecting bad features or ignoring good ones by chance alone.

3.4.3 Intra-Repository Authentication

For the authentication task — that of determining whether a candidate author wrote a given function — our proposed method treats the problem as a binary classification problem wherein our positive samples are those written by the author, while the negative samples are those gathered from other authors in the repository. This is often called a one versus all approach [28]. Our goal is to distinguish between the work of the author in question and the work of another, so we consider the task a binary classification problem. To build a model for an individual, we let their work in the training data be a positive and all other authors be negatives. To reduce sparsity, we perform recursive feature elimination on a validation set, generating a unique feature set for each author. We do this anew for each author, as the optimal feature set for each author will vary. When the recursive feature elimination procedure terminates, the model is trained on all data in the training set to be evaluated.

3.4.4 Intra-Repository Attribution

For the attribution task, where we must attempt to select the author of a work from a pool of candidates, we propose and evaluate two similar methodologies. The first, a more traditional approach, is to treat the task as a multi-class classification problem involving a

single model. We train a multi-class Random-Forest classifier that attempts to output the correct author, with every significant author in the repository as a possible output. The difficulty of this task, intuitively, will vary between repositories depending on the number of authors present.

We also compared the strength of a multi-class model with that of a two-class model like the one we trained for the authentication task. To do this, we build a meta-model by training a binary Random-Forest classifier on each author and predicting the author via maximum likelihood estimation. A Random-Forest can be easily converted to a logistic regression model rather than a classifier, so we use it in this mode to get a predicted likelihood for each author for each test sample. We consider the author with the highest predicted likelihood to be the model's output. We hypothesized that this will give us a stronger model than a single Random-Forest model as each author will have a full ensemble of decision trees devoted to them, hopefully dealing with the problem of having a highly imbalanced training set.

3.4.5 Inter-Repository Study

After our initial, exploratory study on intra-repository classification was complete, we sought to evaluate our techniques on more difficult problems like those faced by forensic practitioners. We concluded that de-anonymizing a portion of a corpus by training on the rest of the corpus is not a feasible strategy in practice, as usually we have an entirely anonymous corpus and a set of candidates. It is more difficult, and therefore more interesting, to train our model on a set of repositories containing work from the candidate authors distinct from the repository in question, allowing us to approach tasks such as de-anonymizing whole corpora of unattributed samples, as well as identifying the author of a commit to a repository when none of the repository's commits *known* contributors to the repository in question.

So, based on much experimentation, we devised a general approach to de-anonymizing source code contributors when the candidate authors are not known contributors to the repository containing the unattributed code. The general procedure is this:

1. Collect a training corpus containing as much code from the candidate author's work history as possible.
2. Mine and run feature detection on the training corpus. This yeilds a feature set, A.
3. Mine and run feature detection on the target corpus. This yields a feature set, B.

4. Let $F = A \cap B$.

5. Perform recursive feature elimination using cross-validation samples from the training corpus, starting with feature set F, yielding an optimal feature set $F' \subseteq F$.

6. Train a model on the training corpus using F'.

7. Apply the model in logistic regression mode to the target corpus yielding the predicted probabilities of each author in the training corpus. These can be used to evaluate the predictive power of the resulting model on a per-author basis.

8. Recommend the author with the maximum likelihood from the candidate author set.

3.4.6 Evaluation

Designing a good test for inter-repository de-anonymization is difficult. Our evaluation set consists of a "target" repository, which contains "unattributed" code. Clearly, we need ground truth to be able to evaluate our models, so we cannot use truly anonymous code. Instead, we use a few carefully selected repositories as our target repositories. We found through mining many repositories and examining the distribution of samples that many repositories are bad candidates for this. Firstly, many repos possess a sort of "founder effect" where one contributor "owns" and manages the project, meaning that all or nearly all target samples come from one individual. We instead needed to find projects with multiple principle contributors to get reasonable test sample sizes for more than one author. We also faced this problem in gathering a data-set for our intra-repository study as well, but by enforcing sample and author minimums were able to find our 37 repositories that did not suffer from this excessively. Second, we found that for many projects that had multiple principle contributors, that project (and forks of it, which were excluded) was the author's only major project, meaning we could not gather sufficient training data on those authors. This was not a problem for our intra-repository study as the authors' work outside of the target repository was irrelevant. Finally, we found that the project must have a substantial development history (>10,000 commits) or else there is unlikely to be enough test data present to get a good evaluation at all. To avoid casting too wide a net and catching repositories with these unfavorable conditions, we carried out a study on three repositories that seemed to us to be good evaluation targets:

- **bitcoin/bitcoin**: We selected this system as it is very well known and because it is a moderately sized C++ repository with multiple significant contributors who also contribute substantially to other crypto-currency projects.

- **ethereum/aleth**: This project is similarly sized and similarly diverse to **bitcoin/bitcoin** and so provides a good within-domain point of comparison.
- **TrinityCore/TrinityCore** This project is similar in size to the crypto-currency repositories, but with commits more evenly divided between principle contributors. This project is also a game engine, representing a non-cryptocurrency domain.

To maintain the closed world assumption, we take an approach in evaluating our models similar to that which we took in the intra-repository study, considering only known contributors to be candidates. Since we exclude the target repository from our training data by design, we need to gather training data from elsewhere. Since we are working with repositories on Github, the author's profiles are the obvious source for training data.

After mining the repository in question, linking as many commits to Github identities as possible, we have a set of contributors with known Github identities that becomes our candidate set. At this stage, we drop authors with fewer than 10 samples from our candidate set as we consider this the bare minimum for testing. We then gather data on our candidate set by 2 mechanisms, both using the Github API. First, for each author, we clone and mine any owned repositories where the author has committed C++ code. Then we mine the user's public event stream and mine any repos to which the author has pushed C++ code. In each case, we are mining only the commits made by the author in question to save time.

We ensure that we are not mining duplicated commit data from any of the repos we gather from the author's profile. First, we ignore any repos that are explicitly labelled as forks of the repo in question. Second, as we mine the target repository, we maintain a record of seen commits and exclude from the training set any repo that contains a commit seen in the target repository. This catches any work that is not labelled as such, but is a de facto fork, a possibility on Github.

We attempted this procedure on several medium to large repositories, but the combination of the effects mentioned above and our exclusion strategy means that in many cases, there was not enough training data to carry out a good evaluation. In the end, we chose to present results on these three repositories not for a lack of alternatives, but for the fact that these three seemed to offer the highest quality data set of the repositories we investigated.

Across all of our experiments, we employ the same mining, feature extraction and feature selection techniques. Each set of experiments only differs in how to define the problem and constrain the dataset. Performing one set of experiments under a single

set of constraints does little to show how our approach generalizes to different problems. In 4.2, we present the results from three experimental setups: *Google Code Jam*, *Intra-Repository*, and *Inter-Repository*, to compare the efficacy of our approach with different experimental constraints. In particular, the difference between our Intra-Repository and Inter-Repository results will illustrate the impact of the repository-separation criterion and the approach's probable usefulness in cases of wholly unattributed projects.

Figure 3.2: Meta-Data describing our dataset for the Intra-Repository study

id	repo	n_authors	n_fun	LOC	LOC/n_fun
1	allwpilib	7	1581	14247	9.01
2	lge_hammerheadcaf	5	2844	98430	34.61
3	moto_shamu	13	3438	124853	36.32
4	angle	11	2626	94630	36.04
5	bitcoinxt	7	1506	51035	33.89
6	caffe	6	750	14405	19.21
7	clams	6	874	40850	46.74
8	clang-logos	38	10517	137880	13.11
9	Clementine	4	543	8414	15.50
10	core	6	1723	78482	45.55
11	cpp-ethereum	11	1952	57366	29.39
12	espressopp	8	930	17855	19.20
13	fbthrift	17	2998	33871	11.30
14	gem5	3	874	21579	24.69
15	icub-main	22	5300	162158	30.60
16	IRTK	4	425	17925	42.18
17	izenelib	12	2784	145017	52.09
18	jubatus_core	8	2344	26926	11.49
19	kicad-source-mirror	18	3489	178536	51.17
20	Learn-Algorithm	3	381	7258	19.05
21	mapbox-gl-native	17	4860	83007	17.08
22	maxcso	10	1867	47410	25.39
23	micro-manager	4	1338	56622	42.32
24	openmw	11	2253	66956	29.72
25	opensim-core	7	1953	96261	49.29
26	pcsx2	9	2994	146046	48.78
27	proxygen	3	671	10828	16.14
28	psi4public	4	679	28164	41.48
29	qtwebengine	4	588	10053	17.10
30	rust-azure	4	1202	33690	28.03
31	sf1r-lite	7	999	32808	32.84
32	skia	18	4977	115630	23.23
33	SkyFireEMU_406a	23	5428	235335	43.36
34	tigervnc	3	1101	59874	54.38
35	tpjs	4	473	11866	25.09
36	upm	4	879	23338	26.55
37	vermont	5	808	21841	27.03

Chapter 4

Results

4.1 Replicating Caliskan-Islam et al. 2015

To evaluate our experimental designs, and give us a point of comparison between an existing method and our new method, we evaluated our tool over a portion of Google Code Jam dataset, as Caliskan-Islam et al. did in 2015. This replication study was carried out using our attribution tool; we rely on Caliskan-Islam et al. for broad methodology only. We took a random sample of 250 authors from the 2012 Google Code Jam the with at least 9 submission files. We evaluate our methodology by carrying out 5 fold cross validation. Within each fold, we perform recursive feature elimination on the training data, then predict the author of the test portion of the fold to get our metrics. Since we had 9 files from 250 authors, this experiment included 2250 files in total.

Overall, our methodology performed well on this task, achieving a micro-averaged f1-score of .74. This is, however, some range away from Caliskan-Islam et al's 98% precedent on this dataset. The reasons for this discrepany are unclear, although we hypothesize that a number of factors may be at play. Our analysis makes some sacrifices for speed and RAM; also, our AST features are less granular than Calsiskan-Islam et al.ś, and our feature set therefore may contain less descriptive data. However, our results are still strong considering they were computed in about 5 minutes and given the small sample size of each author and general complexity of the task. This result indicated to us that although our methodology may not be "state-of-the-art" when it comes to this specific task, it is good enough to move forward with the remainder of the study to work towards practical applications of code stylometry. We can always expand and tweak our feature set once we have completed our aim of developing a practical programmer de-anonymization tool.

As a second point-of-comparison excercise, we re-ran the experiment using all of the data available for the most prolific 250 authors from the 2012 Google Code Jam. Our goal with this experiment was to see how much training set size influences our results. We collect the most recent version of every solution submitted by the available authors; this dataset contained 16,882 files in total from 250 authors as in our first replication trial. We found that given this extra volume training data, our models performed much better than with only 9 documents per author. A comparison can be found in Figure 4.1. It is important to note that when cross-validating with only 9 documents, only 1 or 2 documents from each author makes it into the test set, leaving only 7 or 8 to train with. Given the very large feature set, using such small training and test sets seems a somewhat problematic approach given the *Curse of Dimensionality*[31]. Indeed, when we train our models on this much larger training set, our results improve dramatically. On the larger Google Code Jam sample, our models achieve a micro-averaged f-1 score of .88, much closer to Caliskan-Islam's precedent, and much closer to the range we would like to see in a practical setting. This speaks to **RQ2**: *Can our modified approach work well on the Google Code Jam dataset?* Our answer is yes, but with some caveats. We were not able to replicate the 98% precedent on small training sets, but we were able to achieve strong classification rates when sufficient training data was collected. We would like to do more to determine what the cause of this discrepency may be, but for now we are satisfied that our models work for demonstrative purposes.

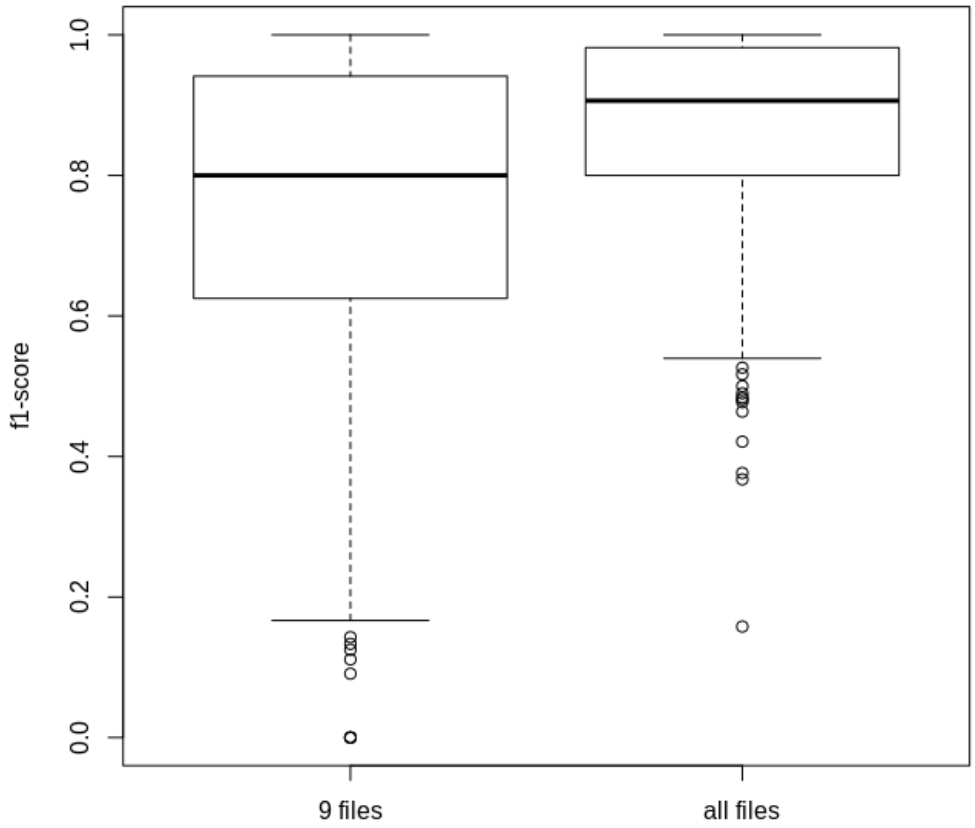
4.2 Intra-Repository Study

For each repository, we mine the full feature set and the features are reduced before training. For binary problems, feature selection occurs at the author level, i.e., each individual classifier in the ensemble has its own feature set. Hence, we had different features selected based on the experiment and model of choice.

4.2.1 Authentication

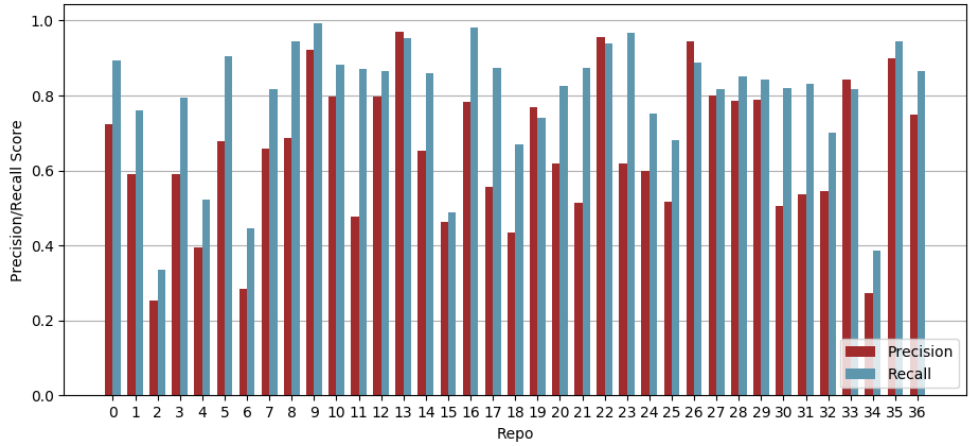
We chose to evaluate our methods for this task by precision/recall at the author level and to aggregate these results at the repository-level for presentation. Figure 3 shows precision/recall micro-averaged across authors in each repository If we let $q(a)$ equal the fraction of samples written by a given author, then the y-axis represents our expected precision/recall were we to select an author from a repository at random with probability distribution $p(a) = q(a)$. Our (macro) average precision/recall across all repositories for

Figure 4.1: Author F-1 Scores 9 files vs. all files



this task was .65 / .79. Therefore, if we chose a repository from our dataset uniformly at random, selected an author from it with probability $q(a)$ from said repository, and evaluated our model's strength for that author, we would expect to correctly identify 79% of the author's code samples in the repository as belonging to the author, and the set we identify as belonging to the author would contain around 65% true positives on average. These ratios may seem low, but it is important to contextualize the difficulty of this task. First, most individual authors do not own more than a third of the selected code in a given repository Although $q(a)$ will vary between repositories and authors, we should expect our precision to roughly equal this fraction in a random guessing scenario. We expect recall to equal 50% when guessing randomly, regardless of the fraction the questioned author makes up. This provides a good explanation for the fact that our models saw higher recall than precision in all but 4 of our repositories. Unfortunately, some repositories saw less than 50% average recall, which indicates that our model's recall on some authors was worse

Figure 4.2: Binary Classification metrics by Repository



than what should be expected when guessing randomly. We address the limitations of our model in certain cases in *discussion*.

4.2.2 Attribution

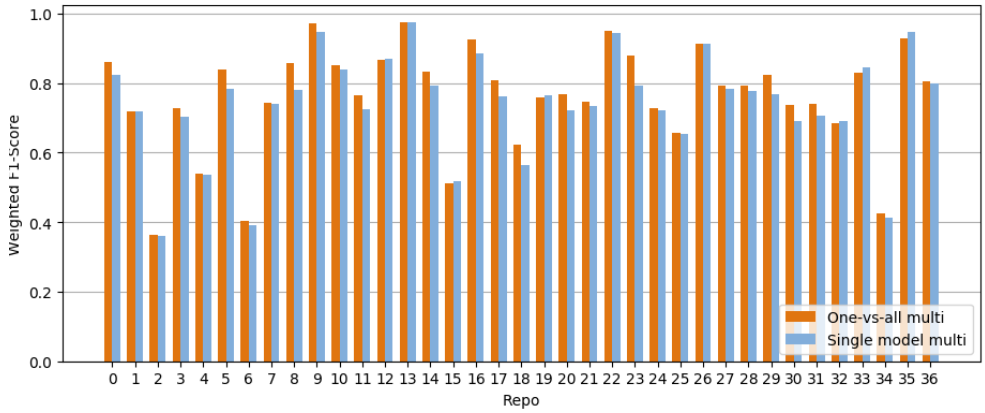
Precision/recall vary widely between authors in a given repository. When micro-averaged, these metrics approximately equal each other, so we use F1 score as a stand-in for both metrics. Figure 4 presents an overview of our results on this task. Macro-averaged over repositories, we achieved a 75% classification rate on this task. That is, if we choose a repository at random and select a function from it uniformly at random, our expected likelihood of guessing the correct author is around 75%.

In 28 out of 37 repositories, the one-versus-all approach performed better than the single model approach. However, the differences in F1-score between the models was very small, never exceeding .077 and averaging .019, an insignificant amount. These results suggest that in this problem, a one-versus-all model does not perform substantially better than a suitably complex single model.

4.3 Inter-Repository Study

To evaluate our modelling approach's effectiveness when employed across repositories, we applied our inter-repository training, modeling and evaluation procedure on 3 repositories.

Figure 4.3: Multi-Class Classification metrics by method and Repository



Author	Trained	Tested	Precision	Recall	F-1	AUC
JeremyRubin	67	20	0	0	0	0.4422
TheBlueMatt	210	59	0.1034	0.1017	0.1025	0.6263
jonasschnelli	102	60	0.75	0.55	0.6423	0.8021
laanwj	295	300	0.4392	0.2767	0.3486	0.5525
sipa	268	263	0.4567	0.5817	0.5154	0.6443
theuni	36	66	0	0	0.0000	0.5832
avg	163	128	0.2916	0.2517	0.2681	0.6084
med	156	63	0.2713	0.1892	0.2256	0.60475

Table 4.1: Classification Results for Bitcoin

We chose these repositories again because they are C++ repositories with a long development history and a number of significant contributors with many functions attributed to them, and a good variety of other projects from which to gather training data.

We report on the Precision, Recall, F-1, and AUC of each author included in our experiments, as well as the number of samples outside of the target repository we included for training and the number of samples from the target repository included for testing. Note that the vast majority of authors in the target repository were neither evaluated nor included in the candidate set as we either did not have enough data (10 functions) in the target repo to perform a good test, or we could not gather sufficient training samples (25) from the author's linked Github account.

Author	Trained	Tested	Precision	Recall	F-1	AUC
chfast	46	43	0	0	0.0000	0.6239
chriseth	313	279	0.2667	0.9176	0.4947	0.7603
gavofyork	42	756	0.8571	0.0079	0.0823	0.596
gumb0	47	20	0	0	0.0000	0.7701
winsvega	110	148	0.0633	0.0946	0.0774	0.5474
avg	112	249	0.2374	0.2040	0.1309	0.6595
med	47	148	0.0633	0.0079	0.0774	0.6239

Table 4.2: Classification Results for Ethereum

Author	Trained	Tested	Precision	Recall	F-1	AUC
DDuarte	2246	200	0.3545	0.335	0.3446	0.4629
Naios	1133	139	0.2069	0.1295	0.1637	0.3239
Subv	624	59	0.4255	0.339	0.3798	0.6341
Traesh	924	19	0.032	0.2105	0.0821	0.5865
Warpten	571	48	0	0	0.0000	0.4074
gpascualg	1338	32	0.0317	0.0625	0.0445	0.6239
sirikfoll	449	29	0.0667	0.0345	0.0480	0.6152
avg	1040.7	75.1	0.1596	0.1587	0.1518	0.5220
med	924	48	0.0667	0.1295	0.0821	0.5865

Table 4.3: Classification Results for TrinityCore

Broadly, we can conclude that our models under-perform under these difficult experimental conditions, with some authors failing to be detected at all and others being detected over-zealously by our model. Just as in our intra-repository study however, the results vary dramatically between authors. In the Bitcoin repository for example, the model managed an respectable, if not impressive, .64 and .51 F-1 for authors jonasschnelli and sipa respectively, indicating a fairly detectable signature. However, in all three repositories there were authors with notably poor results. As is almost always the case with machine learning models, model performance was tightly coupled with the amount of available data. Overall attribution accuracies were 35% for Bitcoin, 22% for Ethereum, and 21% for TrinityCore. One can compare these scores to the expected attribution rate for a random guessing strategem: 16.6%, 20%, and 14.2% respectively. Bitcoin was the only repository for which our models performed significantly better than random guessing, with TrinityCore doing somewhat better and Ethereum doing no better at all.

Of the three repositories we tested, Ethereum proved the most difficult target. This is most likely due to this repository possessing the fewest training samples overall. Another problem with the Ethereum repository that likely contributed to the negative results is that 56% of samples in the training set were gathered for a single author, chriseth. This ultimately led to a landslide effect for this author. As we discuss in chapter 3, we took measures to prevent landslides, namely subsample-balanced class weighting. So, why did the landslide effect prevail even still? We believe that this likely occured because of feature sparsity in the training data. Recall that our feature set for each experiment is not constant, but is built from the tokens and AST node types that appear in the training set. Our resulting feature set can be imagined as the union of all features available for all authors in the candidate set. This, coupled with the overall sparsity of our feature vectors (most features for any given sample will be 0), leads to a large portion of our feature set containing non-zero instances only for a single sample or single author. Thus, this class imbalance produces a substantial subset of our features with high class entropy (discriminative power) as they occur in the prevalent author's work, but not in any other author's work. Such features may be beneficial in small numbers and when each author has their own distinct set of them, but when there is such a distinct class imbalance at play, if any such features occur in the training set, the model will almost always output that class, essentially overfitting to a single author.

As our classification metrics make evident, our results were mixed. Although we achieved rather strong classification metrics in the Google Code Jam problem and the Intra-Repository study, in many respects our approach failed to generalize to the inter-repository problem. However, authorship attribution is a complex problem with multiple objectives, so it is difficult to write an inclusive and informative report of findings based

on classification metrics alone. In 5, we present the details of our results, what they show, where the limitations of our approach lie, and what obstacles exist to making code stylometry applicable in practice.

Chapter 5

Discussion

Although we have made significant efforts to break down our results in an understandable and overarching way, we cannot stress enough how important the nuances in our results truly are. Classification rates varied widely by repository and within nearly all repositories, there were some authors for whom the model performed well and some for whom it performed poorly. To really understand these results, we look more closely at the results gathered from some specific repositories to gain insight into the cases where our model works well and where it does not.

5.1 Intra-Repository Study

Given that authentication is a positive/negative task and the costs of false negatives and false positive change depending on the application, basic classification rate and precision/recall metrics do not paint a complete picture of the model's performance. This is because a fixed threshold classification test does not show the classifier's true positive rate as a function of acceptable false positive rate, as is often the most useful metric in practical settings. For this, we carry out Receiver Operating Curve (ROC) analysis to get a more well-rounded and visualize-able metric for evaluating our authentication approach. An ROC curve is the result of plotting True Positive Rate (true positive rate) as a function of acceptable False Positive Rate (false positive rate) and is used to evaluate a test that outputs a number (in our case a probability). If the test was to be used to give a positive/negative result as in medicine or forensics, we must set this threshold somewhere. The ROC curve is the result of moving this threshold from the lowest possible value to the highest possible value by some small increment and at each threshold level plotting a point

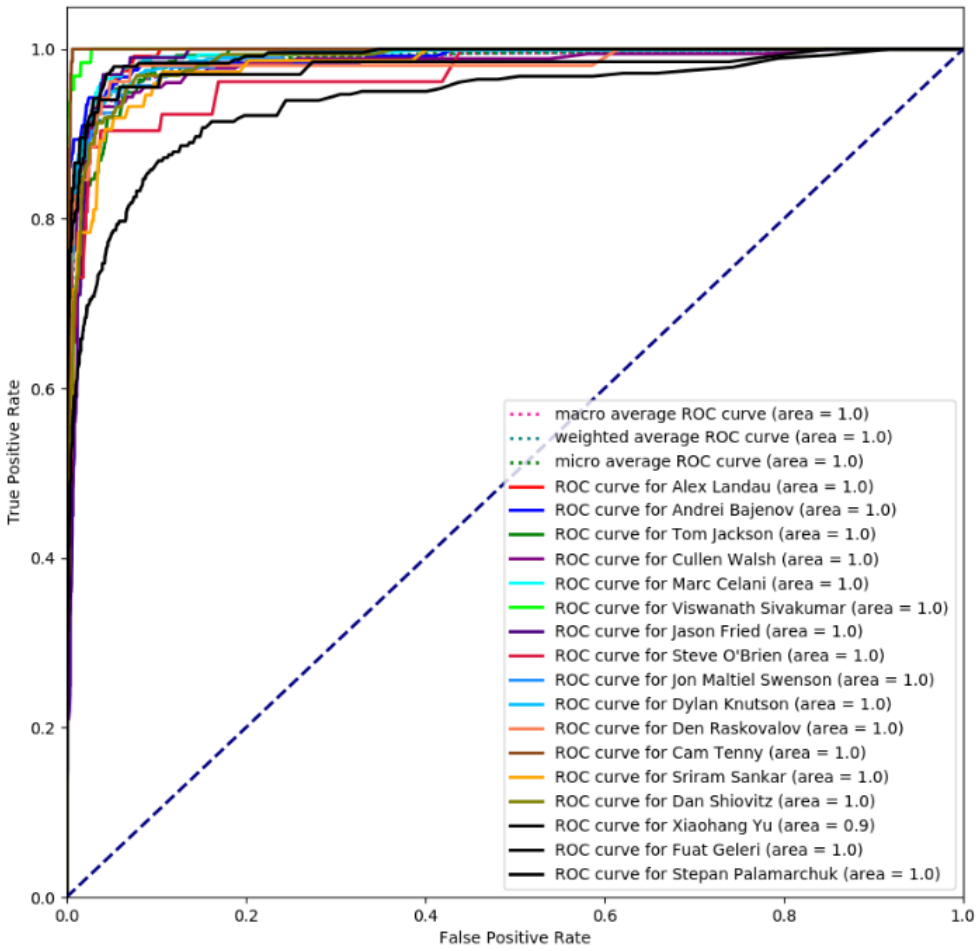


Figure 5.1: ROC Curves by Author for cpp-ethereum

showing false positive rate and true positive rate. Note that the curves are generated from the logistic regression output of a single experimental run. That is, the tests are not run repeatedly to generate the curve. Instead, the logistic regression outputs for each sample in the test set are generated once. We then iterate through the samples sorted increasing by confidence. We then move the confidence threshold from 0 to 1, plotting precision and recall as a point at each iteration.

If the test is very good, then the true positive rate will increase much more quickly than the false positive rate and be near 1 before false positive rate is high. This looks like a curve tightly hugging the left edge, then the top, coming close to the top left corner. If a test is no better than a random guess then the curve will approximate the diagonal. The

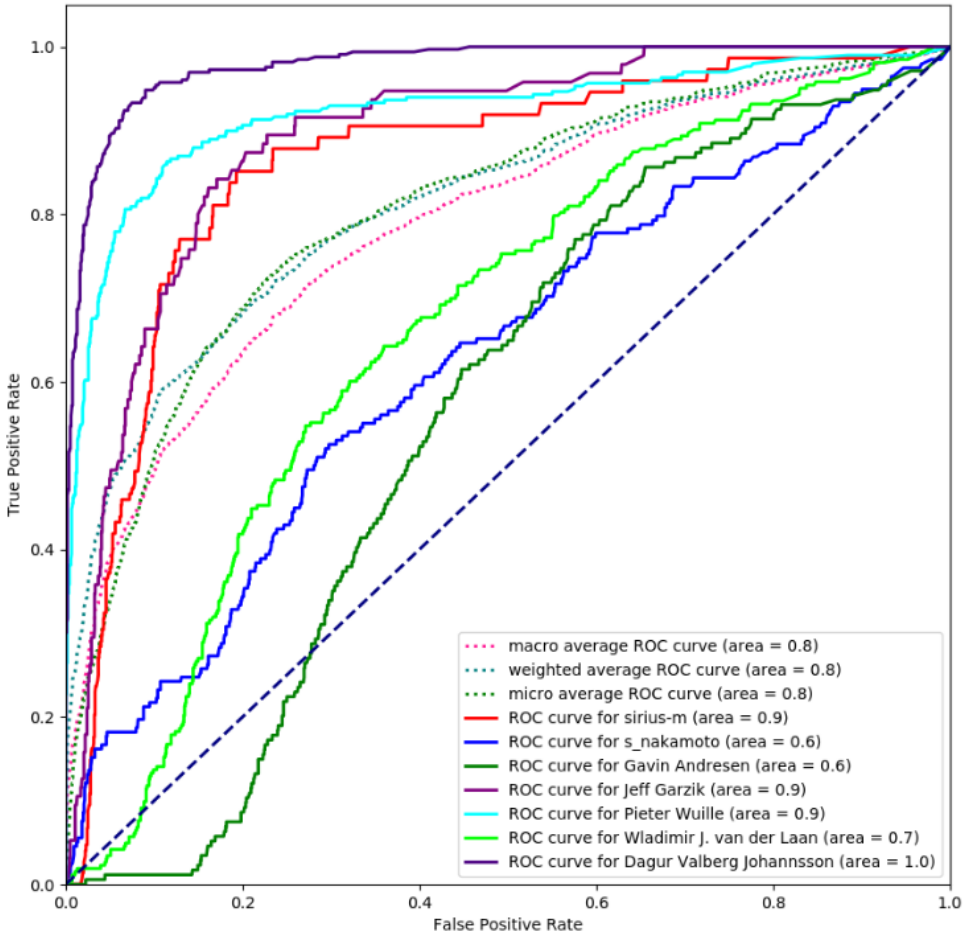


Figure 5.2: ROC Curves by Author for bitcoinxt

area under curve (AUC) metric is therefore a reasonable metric for measuring test efficacy across the threshold range. Note that unlike other metrics such as precision, an AUC of 0.5 is achievable through random guessing, so a good test is usually one with an AUC>0.8.

Figures 5 and 6 show ROC curves for each author in two similar repositories: ethereum and bitcoinxt. All such figures can be found in the appendix. The contrast between the model's handling of these two repositories is stark. This is especially interesting as these two repositories are similar in domain (crypto-currency), size (57 kLOC and 51 kLOC), and function length average (29.4 and 33.9 LOC). Indeed, by every descriptive metric we gathered, the two repositories are similar, so why did our methodology generate such disparate results in these two cases? To address this, we manually inspected the classifier

output and resulting confusion matrix. Almost all of our ROC curve plots look much like the ethereum plot with most authors exhibiting a fairly sharp curve and strong AUC. However, we observed in other repositories the existence of certain authors who are "hard to pin down". That is, some author's styles are so varied and difficult to model that we achieve very poor classification rates for them. We will refer to these authors as *indescribable*. Not surprisingly, simply being in a repository with an indescribable author makes an author more difficult to classify, as the indescribable author's examples are difficult to exclude. When one or more such individuals represent a significant fraction of the repository's code, classification rates repository-wide suffer. In the case of the bitcoinxt repository, three authors stand out as indescribable, with the other 4 authors conforming to the pattern of sharp curves observed in other repositories, albeit dragged down by the others. For reasons that are unclear, the model had a very difficult time distinguishing between the work of *s_nakamoto*, *Gavin Anderson*, and *Wladimir J. van der Laan*.

What causes an author's style to be indescribable? This is difficult to quantify in an independent way, but we conjecture that multiple factors may be responsible. A contributor who focuses solely on GUI code will inevitably be distinguishable from a contributor who focuses on low-level engineering, but when two developers have highly overlapping areas of interest, our model will not benefit from this effect. Another prior that could lead to an indescribable style is multiple individuals committing under the same name. If this occurs, the individual's style signature will exhibit elements of multiple authors, diluting the "author"'s habits and skills. These effects are, at this time, hypothetical.

On the attribution problem, our multi-model approach did marginally better than the single model approach in 28 out of 37 repositories, but overall they performed similarly. They mostly misclassify the same points and achieve nearly the same results on each author. Considering the additional computation costs of the multi-model approach, it does not seem to be an improvement.

Multi-class classification did not prove to be significantly more difficult than authentication. Further, binary precision and recall were strongly correlated with multi-class precision and recall down to the author level. That is, if we can authenticate an author well, we can also attribute their work well. This is actually not too surprising. In all of our experiments, the underlying model is identical. We are training the same model with same (unselected) feature set, but in the binary problem, we merely reduce the output space to two classes. Clearly, we could build a reasonable authentication model if we trained a multi-class model and then simply considered any output not equal to the author in question as a "negative". Although we did not evaluate this approach, our evidence suggests this will work approximately as well as a purpose built binary classifier.

5.1.1 Feature Selection

The results of our feature selection also indicate some simple methods that may make de-anonymization less likely. We found that whitespace features such as the frequency of empty lines and spaces were consistently important features in our models. This suggests that the first defense against a de-anonymization attack should be to remember to format all source code whitespace in a way consistent with the broader community. Better yet, use an auto-formatting tool with the default parameters set to eliminate any personal habits coming through in whitespace.

Another consistently strong subset of our feature set were the frequency of various operators and keywords. In particular: `const`, `static`, `this`, `new`, and `return`. One of the downsides to our approach is that although our model can give us a ranking of feature importance, it is up to us to speculate as to *why* a feature should indicate a particular habit or style. As an example, consider the keyword `return`, one of our top scoring features. Since everyone uses it, it is not obvious why frequency of usage will differ between authors. Nevertheless, one can imagine two developers solving the same problem in two different ways as follows:

Listing 5.1: Two ways to solve the problem

```
int isEvenA(int k)
{
    return k%2 == 0;
}

int isEvenB(int k)
{
    if(k%2 == 0)
        return true;
    else
        return false;
}
```

It stands to reason that mere cognisance of the risk involved in publishing work that may carry one's style may allow one to reduce their usage of unusual syntactic patterns, but some more subtle patterns like the one presented above will be impossible to obscure with mere code formatting.

Obfuscation will be effective at reducing ones ability to de-anonymize code in this way. However, Caliskan-Islam et al. showed that certain style features can survive compilation and obfuscation processes, defying intuition [6]. Further, obfuscation is certainly not an option for open-source contributions as code should be legible to be committed to a project.

AST features have less weight on the classifier than any other class. The ASTs used for this problem are generated at the function level instead of at the file level. In addition, the fuzzy parser that generates the ASTs does not capture statement-specific details about the function. In future work, we can increase the importance of the AST by improving the functionality of the fuzzy parser and further adjusting the AST features to take on more reasonable values at the function-level. This also warrants further research into the relationship between the structure of an AST and the structure and components of its generative function, and how this relationship can be used to create new, indepedent AST-related features that can impact the model's classification capability.

5.2 Inter-Repository Study

Just as in the intra-repository study, we produce ROC curve plots for the 3 repositories we studied. These plots give some important information that our performance metrics do not.

The ROC curve plot from the Bitcoin experiment shows that all but one author have an AUC above .5 and all but the same one have ROC curves above the diagonal. This indicates that at each probability threshold, our binary model of each author performs better than random guessing to varying degrees. This is a somewhat reassuring indicator that our model is indeed picking up some signal from the majority of authors in this case. Two authors stand out as being particularly detectable: *jonaasschnelli* and *sipa*. Taking a closer look at the results, we see that this is suprising. While *sipa* is the second most prevalent author in the training set, our model performed better on *jonaasschnelli* with fewer than half as many samples. However, upon investigation, we have made some observations that suggest an explanation for *jonaasschnelli*'s characteristic "style". Paging through his commit history, we see a prevalence of commits address GUI issues and code related to Qt, a cross platform development framework. Therefore, we should not be overly surprised that he stands opposed to the other contributors to this project. *He frequently works on a fundamentally different portion of the project than the other principle developers.* This gives a good demonstration of a fundamental issue with Code Stylometry as a tool that is present in all of our experiments and, it could be argued, is a significant factor in virtually all prior work. We will discuss what we've learned about this issue further in 5.3.

The ROC curve plot for Ethereum paint a more optimistic picture of our results than does the chart of results. In this case, all authors had an AUC > 0.5. However, none exceeded our "good test" standard of 0.8. By AUC, the best performing authors were *chriseth* and *gumb0*. This is interesting as *gumb0* was never predicted as the author

of any of our test samples. This most likely occured due to the landslide effect which caused *chriseth* to dominate the other candidates in the predictions. What this says is that although our predictive model never detected *gumb0* above the other authors, in a validation scenario, our model performed rather well for this author. That is to say, if we phrased the question as "did *gumb0* write this function?", we achieved a recall of .7 with a false positive rate of only .2 (letting the threshold equal its optimal value). This is certainly better than the 0 recall we achieved in the attribution context, but we must remember that a false positive rate of .2 when the author only makes up a small fraction of samples means that the majority of positives will be false.

TrinityCore's ROC curve plot stands in opposition to the crypto-currency repositories. In this case, 3 of the 7 authors had AUC $< .5$ meaning that in a validation setting, we are more likely to indicate positive on a true negative than a true positive. This is a confounding result that held true despite tweaks to model parameters and indicates a very poor model. Even the best performing author, *Subv*, had an AUC of only .63 and only performs well at the low end of the confidence spectrum. If we let *Subv*'s false positive rate equal .1, we are able to indicate positive on on 40% of this author's samples, which is better than we can do for almost all other authors in this study, but still far from impressive. We do not have a great explanation for why this repository proved so much more difficult to model than the others we studied, but we can speculate that it has to do with the domain of the project. Game Engines are complex systems with code involving graphics APIs, multi-threading issues, character and item behavior and more. It is likely that this leads to each other working on more diverse problem sets in this domain than in others. It is also possible that since game engines are so complex, they are generally written with highly abstract structures involving inheritance, funneling programmers into stricter design patterns than smaller, less fault-tolerant systems like crypto-currencies.

5.3 The Genre Problem

We've alluded to *domain* being an obstacle to building and evaluating stylometric models before. Here, we will expand on this topic and illustrate why we believe it to be pervasive in our experiments and how it may undermine much recent work in this field.

Consider this problem in natural language stylometry: "Classify a chapter of a novel as either written by H.G. Wells or Agatha Christie." In many cases, this may be a trivial problem. We *could* develop an interesting algorithm that takes into account esoteric details of sentence structure, but in most cases, we would not need to. A simple term-frequency based feature set would work well on most examples as Wells mostly wrote science fiction

while Christie wrote mostly mystery novels. This raises a few interesting points. First, could we really call the trivial term-frequency based approach a model of "authorship"? If we introduced a novel by Isaac Asimov, it would most likely confidently bin it with Wells' work. What stylometric models are really doing, if we are not careful, is classifying based on *genre*, not writing style. Additionally, there are some hard cases that could not be so trivially solved. Both authors also wrote works of horror, for which our imaginary model may fail. For these tough cases, we need to focus on features that are *subject-matter invariant*.

In natural language, when the subject changes, the syntax does not. We use the same syntactical forms writing a mystery novel as we do novel. We use different nouns and verbs and adverbs, but the syntactic structures demanded by each genre are consistent. At the risk of reading utterly non-rigorous, we would say that the meaning of a sentence comes from the choice of words but the structure is somewhat imposed upon us by language rules. Of course, the rules of the language do give us some options, which is why we can do natural language stylometry well in a lot of cases. We have options as to what pronouns we use more often, how many words are in our sentences, the frequencies with which we use stop words, and many other features, but we are always restricted to relatively few high-level syntactic forms. In code, we would argue that the process is very different. One begins their process with a problem to solve, then they work backwards from the goal to achieve a structure which solves that problem. This is not to say that when two people write the same program, they will necessarily create very similar structures. Rather, the difficulty for stylometry is in the fact that *the task defines the required syntactic structure*. 100 programmers will likely produce a large variety of solutions to a sufficiently complex problem, but the code that they produce is always in response to problem with which they are presented. The intuition of the code stylometrist however is that the decisions the programmer makes along the way are what may make him identifiable, just as in natural language.

The problem with this intuition is subtle. The first troubles come when we consider just how many reasonable ways there are to accomplish a programming task and how difficult it is to characterize them in any meaningful way. In natural language, we have a finite set of stop words, contractions, "to be" verbs and other subject-matter invariant linguistic constructs. We would argue that the stylistic choices we make in writing natural language all pull from a limited, although large, set of linguistic tools. These sets are small enough to enumerate for any given category and we can be sure that no novelist will write a chapter without using a function word and, therefore, we can measure these choices with some certainty that the variable we're measuring will apply to the questioned document. In source code we have no such guarantee. Frequently used elements of one

project may be vanishingly rare in another. Certainly, we can limit our analysis only to the most ubiquitous and unavoidable constructs of language like white-space ratios and return statement usage, but how likely is it that such shallow feature sets will be able to do the job? The fundamental semantic elements of a program are hugely diverse, ever changing, and difficult to characterize. Style means one thing in writing GUI code and means another in writing back-end code. Since the set of common constructs and tools will vary so much, any strong descriptor of code style in one genre is likely to be a poor descriptor of style in the other, and any succinct description of style equally applicable to both will be a poor descriptor of both. Our fundamental argument to this point, synthesized, is this: *Any feature set that can claim to succinctly describe the style of a given program will fail to give an informative description of the style of most other programs.* It is hard for us to imagine a method of empirically justifying this claim, but it could certainly be challenged by a sufficiently strong cross-genre model of coding style.

5.4 The Dimensionality Problem

We qualified our previous claim by requiring a description of style to be succinct. Why should this necessarily be a requirement? We have ever increasing computational might, why not build ever more inclusive feature sets? Simply, as we found throughout the process of our studies, more inclusive feature sets do not generally lead to more powerful models. This is not a new discovery, nor is it unique to authorship attribution problems in general. This problem is know in the statistical learning community as *The Curse of Dimensionality*. Simply, though unrigourously stated: "For any given learning algorithm and training set size, there exists some optimal number of features up to which generalization error will decrease and beyond which generalization error will increase." In some of the very earliest work on this topic, Gordon P. Hughes[12] explores this idea from first principles . This work convincingly demonstrates the existence of what the author calls n_{opt}, the optimal dimensionality for a data set of any fixed size. This work discusses Bayesian classifiers and assumes binary classification with $p(c_1) = p(c_2)$ and therefore the numerical results he achieves won't apply to our specific problem (or any non-ideal problem for that matter), but he arrives empirically at $n_{opt} = 23$ for $m = 1000$ for a binary classification problem with equal class likelihood (he uses n for dimensionality and m for sample size, the inverse of common practice today). Rules of thumb abound in the field for this problem. The recommended minimum number of samples per features ranges from 10 in the widely referenced Machine Learning Textbook textbook *Learning from Data*[5] to 5 samples in *Pattern Recognition*[30], a widely utilized data processing reference. These values are

given for binary classification problems without class imbalance, so for multiclass problems with imbalanced classes, the picture becomes murkier. With any problem and model, the optimal number is an unknowable. We can empirically estimate it with feature reduction, as we have done, but this process has caveats, especially when training and test sets are not drawn from the same population. In short, there is no way to compute this number for any real-world problem, but the general consensus in the community is unarguably that for any given problem, $m >> n_{opt}$. This relationship enforces the conciseness constraint we mentioned previously. Feature set conciseness is not merely a desirable property of any statistical learning method, it is provably a fundamental requirement.

This brings us to our last hope of building a strong, general purpose model of authorship: an abundance of data. The subject matter of source code authorship attribution however, dashes these hopes. In our inter-repository study, the most prevalent author we had was DDuarte ($m = 2246$). Even if we considered lines of code samples, instead of functions, we are left with less than 10,000 samples for all but the most prolific open source engineers in the world and lines make up a much less information rich sample. Taking 10,000 LOC (naively) as a typical sample size, this leaves us with n_{opt} likely being <5000 (generously), far less than the >100,000 (post-selection) employed by Dauber et al. [10] and Caliskan-Islam et al. [8] for their experiments. How then do we reconcile their impressive classification rates with the concensus of the machine learning community? Based on an examination of their tools [1], the setup of their problems[10], we believe that a form of overfitting is the likely culprit. Specifically, over-fitting to genre specific features present in the distinct sets of software projects gathered for each author. Although we did not employ an identical approach, we feel confident that given the strict project separation criterion of our experiments, the aforementioned authors' approaches would do little better than ours. We have reached out to the authors of Caliskan-Islam et al. for comment, but did not receive a response. Since this genre issue was mentioned as a threat in Dauber et al. it seems the authors are aware of this problem and may address it in future work.

The results of our experiments show how sensitive these methods are to problem definition. A model that works in one experiment may not generalize well to different sets of constraints. This establishes the level of care that must be taken when carrying out such experiments and the nuance with which researchers and practitioners must interpret their results. It cannot be overstated how important it is that readers look past classification metrics and examine the constraints to which the experimenters adhered and consider how those constraints simulate problems one would wish to solve in practice.

[1] Available here: github.com/calaylin/CodeStylometry

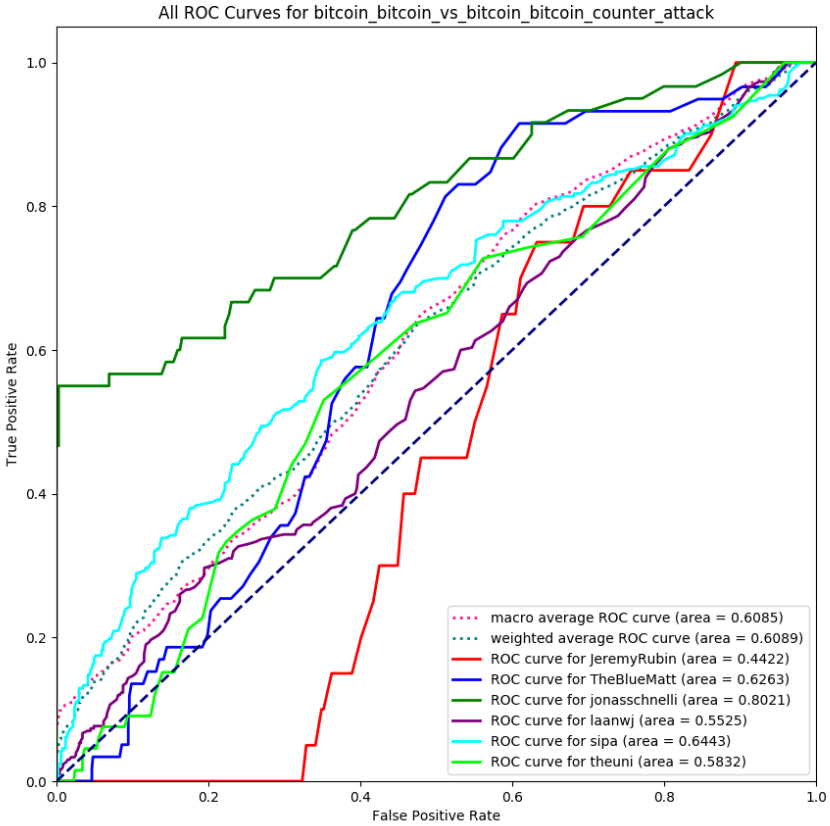


Figure 5.3: ROC Curves by Author for Bitcoin

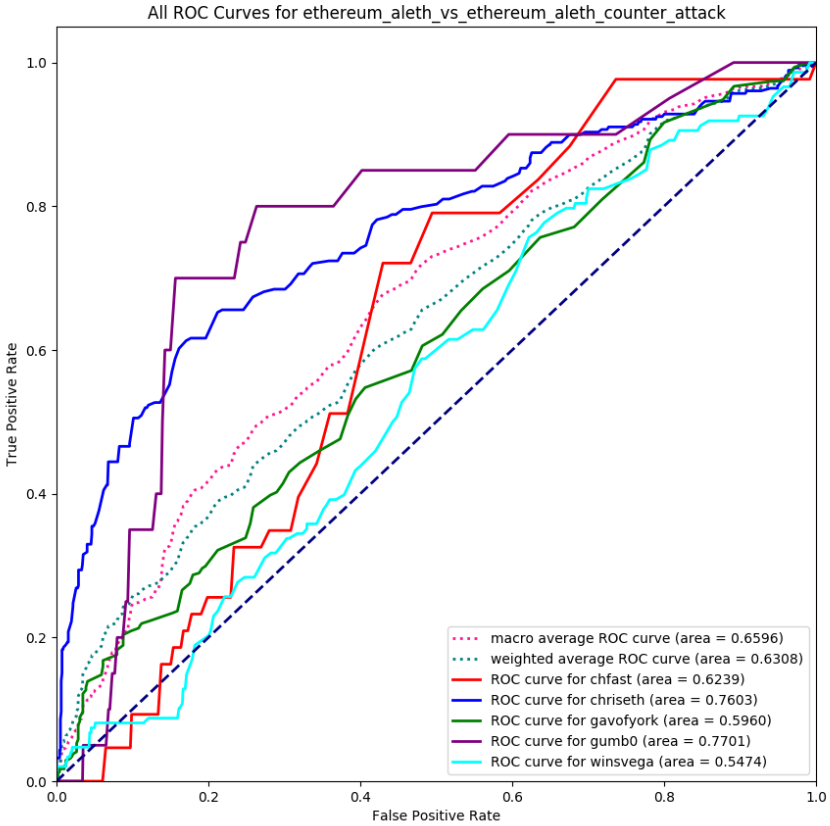


Figure 5.4: ROC Curves by Author for Ethereum

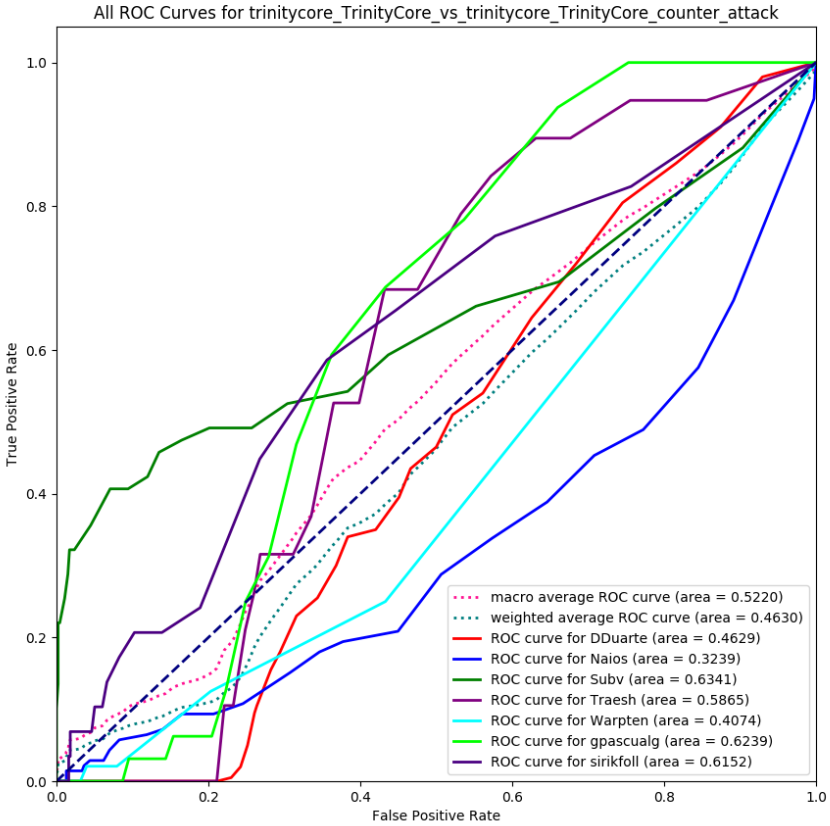


Figure 5.5: ROC Curves by Author for TrinityCore

Chapter 6

Conclusions

6.1 Future Work

Although we were able to produce some strong discriminative models of authorship between authors within a repository, to make this technology useful to forensic practitioners, it must scale well to the problem of internet-wide authorship attribution. If we want to be able to examine a sample of code and ask questions about who we believe the author to be without a given set of suspects, we must assume our suspect pool to be the broadest possible development community. Our methods prove effective when the pool of suspects is relatively small, a few dozen or so. However, the computational requirements for even this medium-scale study were substantial and carrying out the attribution task at a very large scale would be infeasible with the methods presented here. In order to allow for generalized de-anonymization over very large sets of repositories, we must first develop a scalable protocol to reduce the initial suspect pool for a query from the hundreds of thousands of open-source contributors on Github and other platforms, to a smaller pool of reasonable candidates. We intend to employ information retrieval techniques to allow us to scale our approach. It is our belief that a combination of high performance computing resources and a layered filtering approach to candidate set reduction may be an effective avenue towards creating a general purpose, ultra-large scale authorship attribution tool. Stomatos[29] has demonstrated applications of information retrieval techniques to source code authorship attribution in the past. Narayanan et al. have explored large scale cases of authorship attribution and found that their techniques tended to scale rather well to cases with thousands of authors [23]. This is our intention for the near future. Our next step will be to re-implement our authorship attribution workflow in Apache Spark [1], a big-

data analytics platform, to allow interactive work sessions to leverage the high-performance computing infrastructure available to us to give timely results to queries involving ultra-large sets of authors.

We would like to note that the models applied throughout our effort were Random-Forest classifiers. This model was selected for its variance reduction capability, interpretability, and because Random-Forests had been demonstrated by others [7] to be effective in attribution tasks. However, there are other models that have been empirically shown to be effective for text classification problems that may produce better results with lower computational cost than Random-Forests. Similar to Random-Forests, boosting methods produce a strong classifier from multiple weak classifiers. However, the weak classifiers in boosting methods are dependent and sequentially trained as with each classifier, incorrectly-classified observations are penalized with higher weights. This could be especially useful for our problem, as authors with less contributions will likely be prioritized during training, resulting in more accurate results for such authors.

Finally, we would like to explore empirical methods in attribution problems that are more limited in scope. Single author detection problems where we would like to determine whether a specific author contributed to a work are the most likely to find application in forensic practice. We note that this is similar to the authentication problem presented here, but is less concerned with individual samples and more interested in the presence of an author in a large set of samples. This appears to be a more tractable problem and our results indicate that this may be doable in the near future with simplified versions of present models.

6.2 Final Remarks

Code Stylometry is a relatively nascent and as yet, obscure discipline. Our experience working in this area simultaneously evidences two standpoints.

First, our work does provide some evidence that with advancements in NLP, machine learning, and high performance computing, authorship attribution is becoming a tractable problem in some limited respects. Given the right limited problem space, small candidate set, and consistent domain, it is possible to build a model which to some degree characterises the style of an author in that domain, with respect to the others they have worked with. Whether or not such techniques could be employed or have already been employed by governments or private entities for information gathering activities is currently unclear. With the recent global turn away from internet freedom and privacy, it seems likely that

there will be more rogue engineers building technologies that their own state may consider "subversive"; any such individuals should be aware that avoiding digital traces alone may not be enough, as researchers may develop ever more sophisticated means of fingerprinting authors by their work.

However, we feel that it is important for any researcher or practitioner, dabbling in this field to know that significant theoretical work in statistical learning predicts that this will be a *very* hard problem to solve with any empirical validity. We are always forced to choose between an extremely descriptive model, for which we will not have nearly the volume and variety of training data we would need to plausibly claim that we have avoided over-fitting, or a simpler model, which will generalize better, but won't be complex enough to describe style deeply enough to transition well between genres.

When the only tool you have is a hammer, every problem looks like a nail. The allure of ever more complex models like deep neural networks will draw in practitioners who have faith that such models can be taught to solve the most complex of problems. Source code authorship attribution is indeed a complex problem and so lends it self to such ideas intuitively. However, the complexity of this problem lies in the empirical details of gathering clean training sets and concise, informative, and subject-matter-invariant feature sets or models that can somehow identify them. Our standpoint is not that code stylometry is an outright impossible task, but that designing an automated pipeline that can sensibly get from question to answer for any given problem is not a good near-term goal. Researchers' time would be better spent focusing on human-in-the-loop empirical methods for small subsets of the authorship attribution problem space, for it is in limited, carefully validated studies of programming style where we can see advancements in the field in the near future.

References

[1] Apache Spark - Unified Analytics Engine for Big Data.

[2] Joern - A Robust Code Analysis Platform for C/C++.

[3] Xu Dong develops software for circumventing the Great Firewall, arrested for crime of creating a disturbance. — Boxun News, 2014.

[4] Shadowsocks: A secure SOCKS5 proxy. Technical report, 2019.

[5] Yaser S Abu-Mostafa, Malik Magdon-Ismail, and Hsuan-Tien Lin. *Learning From Data*. AMLBook, 2012.

[6] Aylin Caliskan, Fabian Yamaguchi, Edwin Dauber, Richard Harang, Konrad Rieck, Rachel Greenstadt, and Arvind Narayanan. When Coding Style Survives Compilation: De-anonymizing Programmers from Executable Binaries. (February), 2015.

[7] Aylin Caliskan-Islam. Stylometric Fingerprints and Privacy Behavior in Textual Data. *ProQuest Dissertations and Theses*, 2015.

[8] Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt. De-anonymizing Programmers via Code Stylometry. *USENIX sec*, pages 255–270, 2015.

[9] Nitesh V. Chawla, Nathalie Japkowicz, and Aleksander Kotcz. Editorial. *ACM SIGKDD Explorations Newsletter*, 6(1):1, 2004.

[10] Edwin Dauber, Aylin Caliskan, Richard Harang, and Rachel Greenstadt. Git blame who?: Stylistic authorship attribution of small, incomplete source code fragments. *Proceedings - International Conference on Software Engineering*, 2019(3):356–357, 2018.

[11] Roger Dingledine. Tor: The Second-Generation Onion Router. Technical report.

[12] Hughes G F. On the mean accuracy of statistical pattern recognizers. *IEEE Transactions on Information Theory*, 14:55, 1968.

[13] Mathieu Goeminne and Tom Mens. A comparison of identity merge algorithms for software repositories, 8 2013.

[14] Pablo M Granitto, Cesare Furlanello, Franco Biasioli, and Flavia Gasperi. Recursive feature elimination with random forest for PTR-MS analysis of agroindustrial products. 2006.

[15] Rachel Greenstadt. Code Stylometry and Programmer De-anonymization, 2016.

[16] Lile P Hattori and Michele Lanza. On the Nature of Commits. Technical report.

[17] Moshe Koppel and Jonathan Schler. Authorship verification as a one-class classification problem. page 62, 2004.

[18] Erik Kouters, Bogdan Vasilescu, Alexander Serebrenik, and Mark G J Van Den Brand. Who's who in Gnome: Using LSA to merge software repository identities. *IEEE International Conference on Software Maintenance, ICSM*, pages 592–595, 2012.

[19] Chris Lattner and Vikram Adve. LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[20] TingPeng Liang, ChihChung Liu, TseMin Lin, and Binshan Lin. Effect of team diversity on software project performance. *Industrial Management & Data Systems*, 107(5):636–653, 2007.

[21] B. McKeeman. The Grammar for ARM C++ with _opt factored out. 1993.

[22] Satoshi Nakamoto. Bitcoin: A Peer-to-Peer Electronic Cash System. Technical report.

[23] Arvind Narayanan, Hristo Paskov, Neil Zhenqiang Gong, John Bethencourt, Emil Stefanov, Eui Chul, Richard Shin, and Dawn Song. On the Feasibility of Internet-Scale Author Identification. Technical report, 2012.

[24] Paul Oman. Typographic Style is More than Cosmetic. *Article in Communications of the ACM*, 1990.

[25] Paul W Oman and Curtis R Cook. A paradigm for programming style research. *ACM SIGPLAN Notices*, 23(12):69–78, 1988.

[26] F Pedregosa, G Varoquaux, A Gramfort, V Michel, B Thirion, O Grisel, M Blondel, P Prettenhofer, R Weiss, V Dubourg, J Vanderplas, A Passos, D Cournapeau, M Brucher, M Perrot, and E Duchesnay. Scikit-learn: Machine Learning in {P}ython. *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[27] Andrea Dal Pozzolo, Olivier Caelen, Reid A. Johnson, and Gianluca Bontempi. Calibrating probability with undersampling for unbalanced classification. *Proceedings - 2015 IEEE Symposium Series on Computational Intelligence, SSCI 2015*, (November):159–166, 2015.

[28] Ryan Rifkin and Aldebaro Klautau. In Defense of One-Vs-All Classification. Technical report, 2004.

[29] E. Stamatatos, N. Fakotakis, and G. Kokkinakis. Computer-based authorship attribution without lexical measures. *Computers and the Humanities*, 35(2):193–214, 2001.

[30] Sergios Theodoridis and Konstantinos Koutroumbas. *Pattern recognition*. Academic Press, 2009.

[31] G. V. Trunk. A Problem of Dimensionality: A Simple Example. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-1(3):306–307, 1979.

APPENDICES

.1 Our Tool: CoderID

One of our goals for this project was not only to propose novel approaches for source code stylometry, but to implement them as a research tool that is easy to use, to help with replication efforts and to allow others to improve upon our work. To that end, we have assembled all routines used to mine, extract, and classify source code into an open-source software package called coderID.[1] This is a Python CLI tool that includes a module implementing the common subroutines needed for this work, along with a command line interpreter to allow relatively easy replication and enhancement. Our methodology has been automated down to running 1 driver script, which should produce the graphs we displayed as Figures ??, ??, ??, and ??. It also produces all results in raw CSV format for deeper analysis.

CoderID includes commands for cloning, mining, and classifying Github repos using the Github API. This allows cursory analysis and testing to be carried very quickly and easily after setup and installation is complete. Our package also includes routines for building training sets from arbitrary sets of repositories and performing de-anonymizing routines without programming on the user's part. The modules are designed to be extensible and configurable, so researchers can apply different models in the SciKit-Learn Machine Learning toolkit and tweak model parameters using a configuration file. All figures presented in this work were created either directly using the tool, or generated via the CSV files generated after running attribution routines.

The operating idea of the tool is to select a repository of interest and command the CLI to perform various operations on it from mining it for commit information, outputing summaries of this information, training and evaluating and intra-repository model, or assembling, training, and evaluating an inter-repository model.

[1] Available here: github.com/danielWatson3141/coderID

The tool is capable of saving and loading files containing extracted repository information to save progress. This is accomplished using Python's Pickle module. A repository's feature set can also be written to the same file in compressed sparse matrix format allowing a previously mined and feature detected repository to be quickly loaded and analyzed. Multiple repositories can also be joined together to form a single profile for when we would like our training or test set to consist of multiple repositories. We can also limit our mining to mine only certain authors, avoid certain commits, or mine only specific commits, although these restrictions are managed automatically when the user attempts an inter-repository de-anonymization routine.

The design goals of the interface were:

1. Make the process abstract. We wanted to avoid the user needing to type in URL's, file paths and the like. Let the practitioner focus on the task, not the details.

2. Make the process checkpoint capable. If the program fails or needs to be interrupted, it is very inconvenient to restart the entire pipeline from square one.

3. Make the process transparent. Give the user information about the progress of things without polluting the prompt. We found when using the tool it was important to us to have some picture of the pace of things to plan our experiments.

4. Make the tool configureable. Provide a configuration file wherein the user can set their desired classifier and data mining parameters.

To keep the process as abstract as possible, certain commands have in-built polymorphism which helps to isolate the user from the complexities of the backend. For example, the load [repoName] command has 3 separate effects in the background, but the end result is always the same. The whatever github repo is specified by repoName becomes the active repo. The background operations of this command are this:

1. If a saved profile with under the same name is in /savedSets, then load it from the file and set it as the active repo.

2. If the repo is not in the /repos directory already, coderID searches Github for a matching repo and clones it, then sets it as the active repo.

3. If a matching repo has already been downloaded to the /repos directory, then set it as the active repo. This allows users to mine repos not available on Github.

There are other instances of such command polymorphism, and the goal is for the end result to be the same no matter the context. Loading a repo always sets the desired repo as the active one (assuming it exists), and the result does not depend on what stages of the pipeline have already been carried out.

After a repository is loaded, it can be mined, or "compiled". This is the part of the process that traverses the commits in the repository using Pydriller and extracts functions using Lizard. This is typically the most time consuming portion of the pipeline as the process is a disk-bound one. This can take anywhere from a few minutes for very small repositories to about 12 hours for the largest of them, but the time requirements of this phase are rather hardware dependant.

This is where the checkpoint capability of coderID becomes critical. If we spend hours mining a repository only to make a mistake later on, we do not want to lose our progress, so the tool automatically writes the extracted source code to the file associated with that repository so it can be loaded quickly later. The same is true of the feature extraction step which also writes its results to a file after completion to avoid re-doing work. See figure ?? for an example of typical operation at this phase. The velocity and progress indicators are from the extremely elegant TQDM module which gives active updates on progress and time remaining.

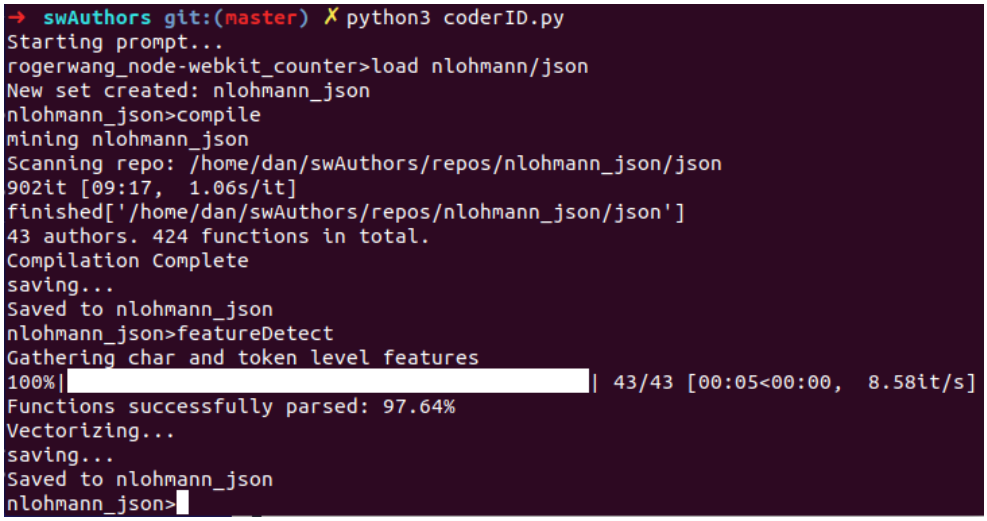


Figure 1: Commands for downloading and mining repositories

Once the repository of interest has been mined, the user can attempt any of the de-anonymization procedures presented in this work. For intra-repository tasks the user has

the option between "multiclassTest" and "multiClassSingleModelTest" which together were used to produce the results in section 4.2. The user may also invoke "deAnonymize" to attempt the inter-repository attack described in section 3.4.5.

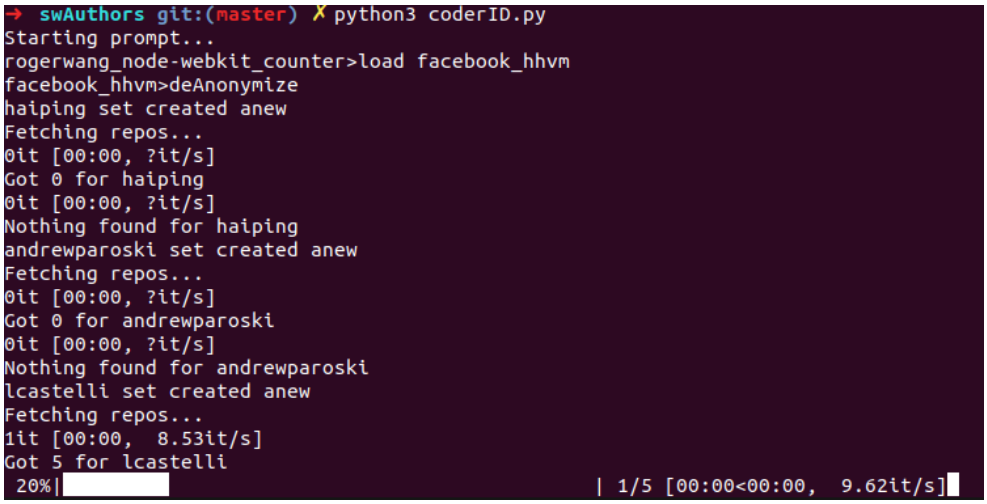


Figure 2: Process of constructing counter set to target repository

This process begins with constructing a "counter set" consisting of all available data from the authors contained in the target repository. A list of the authors' repositories and those to which they have contributed is obtained via the Github API, then each is cloned and mined. If we mine indiscriminately, we can end up downloading and scanning a large volume of other authors' commits and non-C++ code, so the tool takes multiple measures to minimize this. When mining, we only mine commits belonging to the authors in question as informed by the Github API. Since our tool is at this time limited to C++, we only scan repositories that have at least 1KB of C++ source code. Figure ?? shows the beginning of this process when running on a repository that has already been mined. Once this process is complete, the counter set is saved and can be quickly loaded again to re-run analysis with different parameters.

Once the counter set is constructed, we have our training and test sets to evaluate. Figure ?? shows the output of this process. First, any authors with fewer samples than a configurable minimum are removed from both the training and test set. Then, the feature sets of the training and target sets are unified, that is, we take their union. Our random forest model is then contstucted over the training set using the parameters set in the configuration settings. Finally, the model is evaluated over the test set (the target repository). Metrics on the results are output to the terminal as well as written to a csv

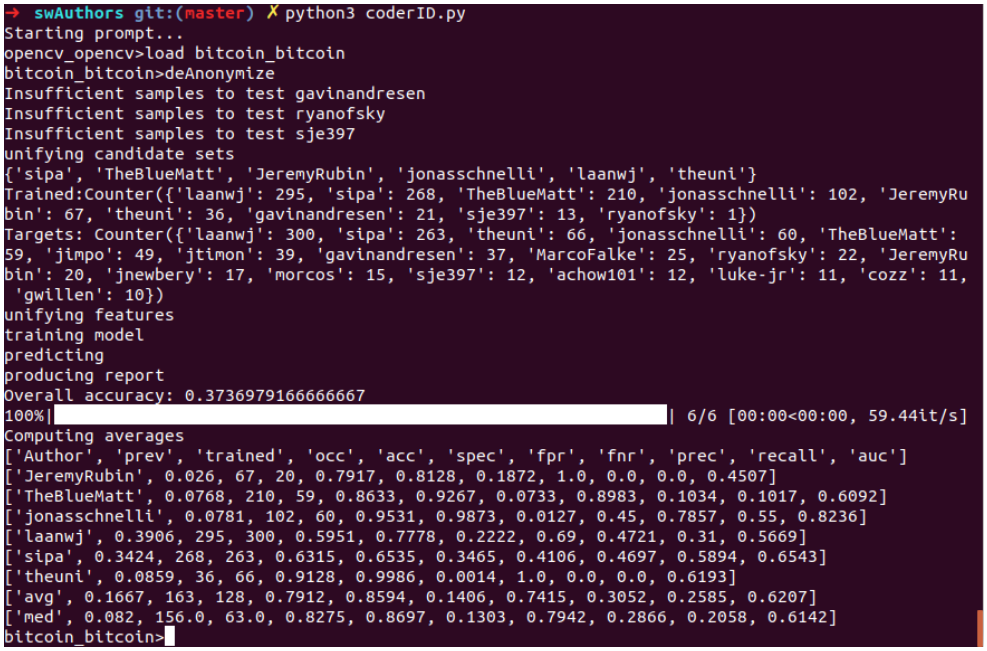


Figure 3: Result of de-anonymizing with counter set

file in /classResults. Plots showing ROC curves for each author are also generated and stored in /plots. The full process is automated. One simply needs to load the repository they are interested in and invoke deAnonymize. If the target repository is very large, this process may take hours and use a large amount of RAM, so we recommend running it on a capable server and using a Tmux window to prevent network interruptions from halting the process.

.2 Intra-Repository Study ROC Plots

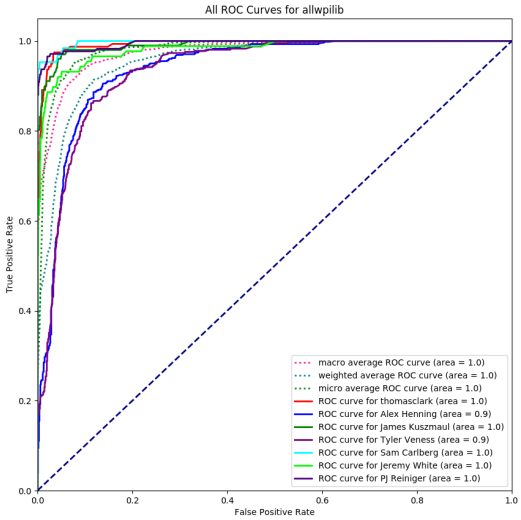


Figure 4: ROC Curves by Author for allwpilib

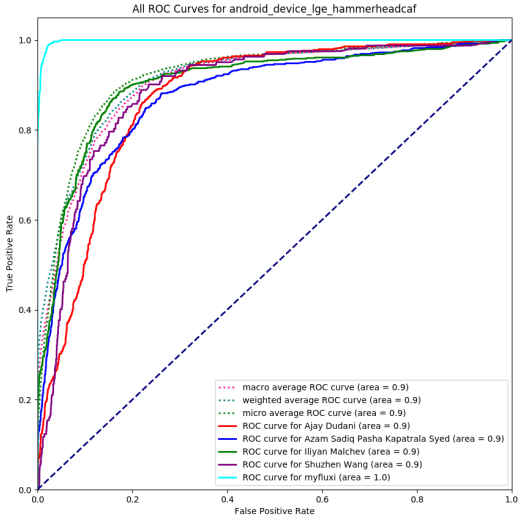


Figure 5: ROC Curves by Author for android_device_lge_hammerheadcaf

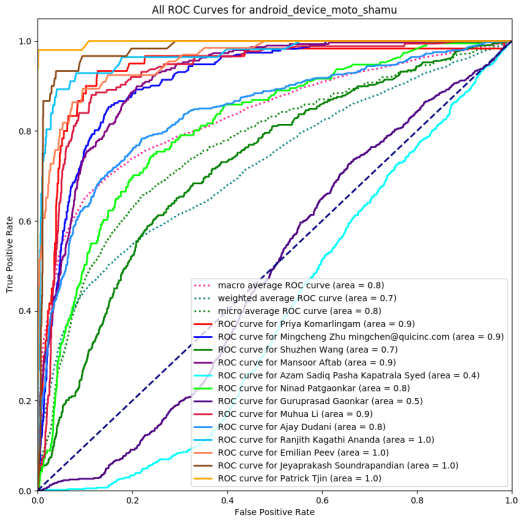


Figure 6: ROC Curves by Author for android_device_moto_shamu

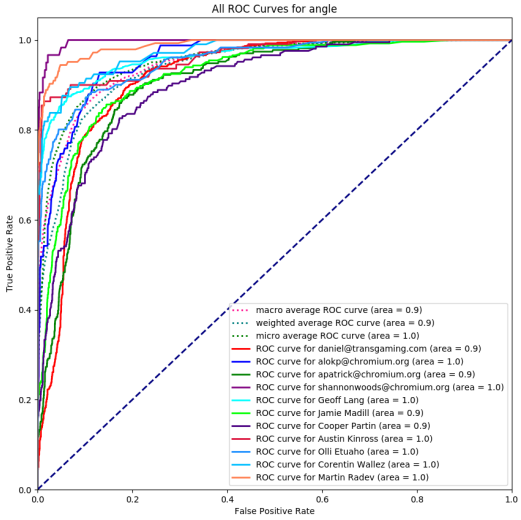


Figure 7: ROC Curves by Author for angle

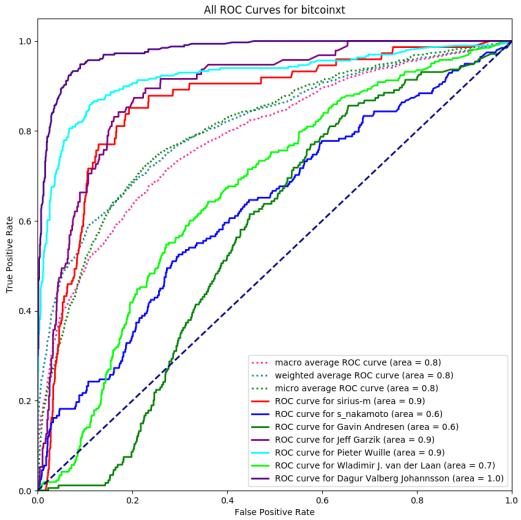


Figure 8: ROC Curves by Author for bitcoinxt

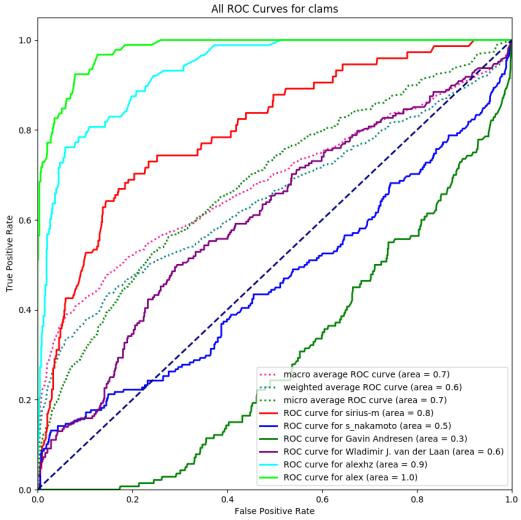


Figure 9: ROC Curves by Author for clams

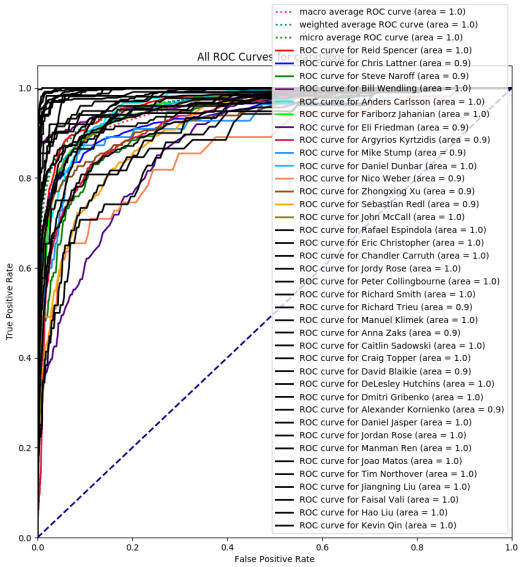


Figure 10: ROC Curves by Author for clang-logos

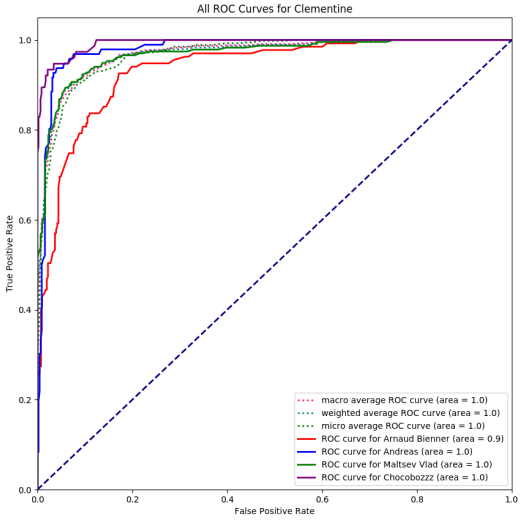


Figure 11: ROC Curves by Author for Clementine

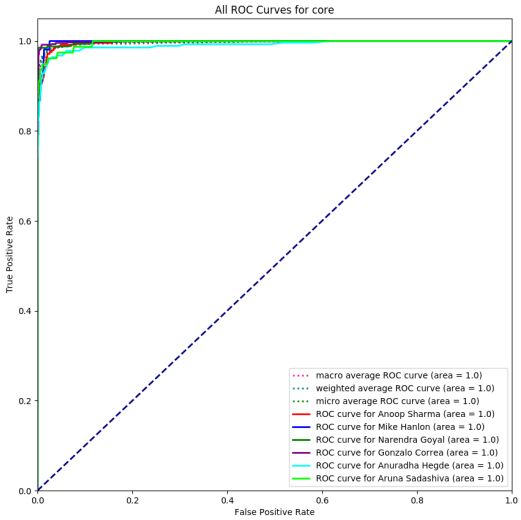


Figure 12: ROC Curves by Author for core

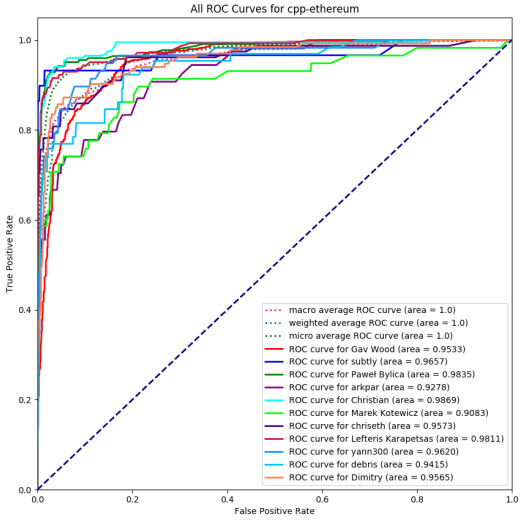


Figure 13: ROC Curves by Author for cpp-ethereum

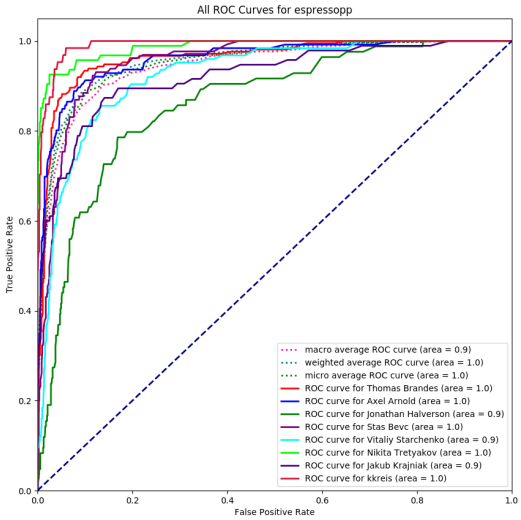


Figure 14: ROC Curves by Author for espressopp

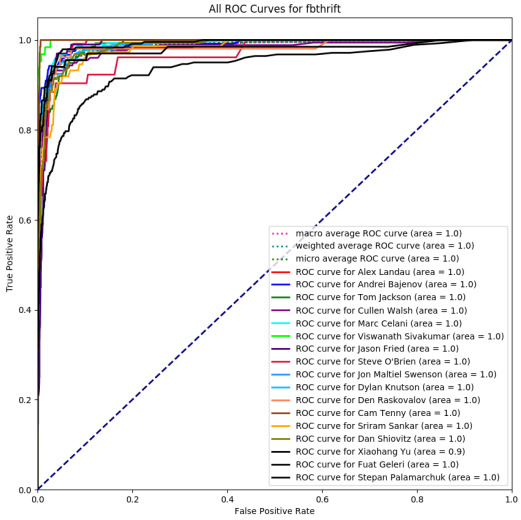


Figure 15: ROC Curves by Author for fbthrift

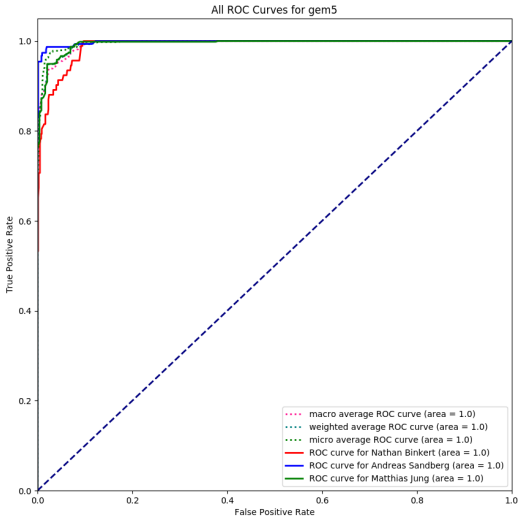


Figure 16: ROC Curves by Author for gem5

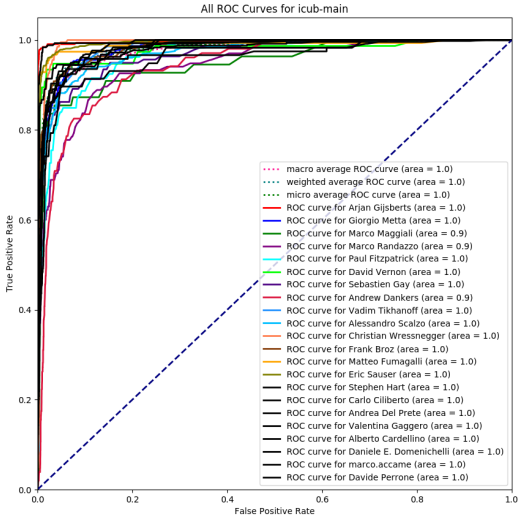


Figure 17: ROC Curves by Author for icub-main

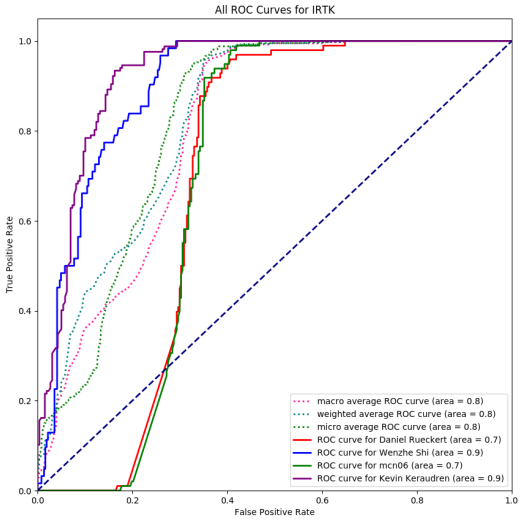


Figure 18: ROC Curves by Author for IRTK

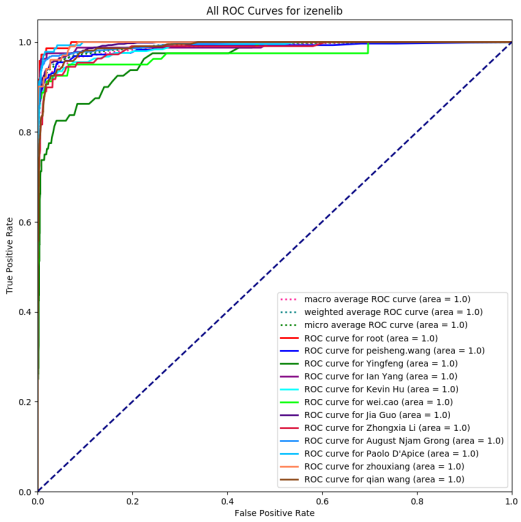


Figure 19: ROC Curves by Author for izenelib

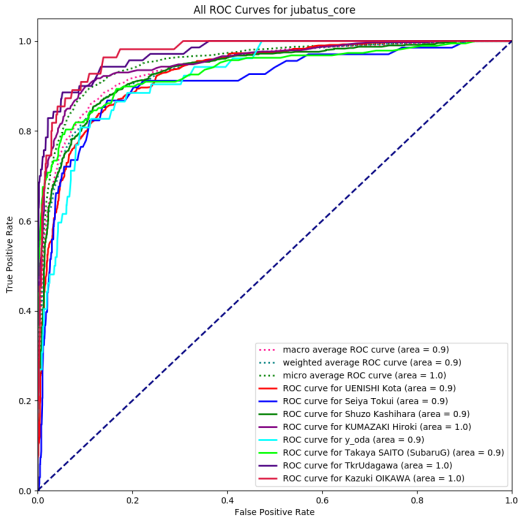


Figure 20: ROC Curves by Author for jubatus_core

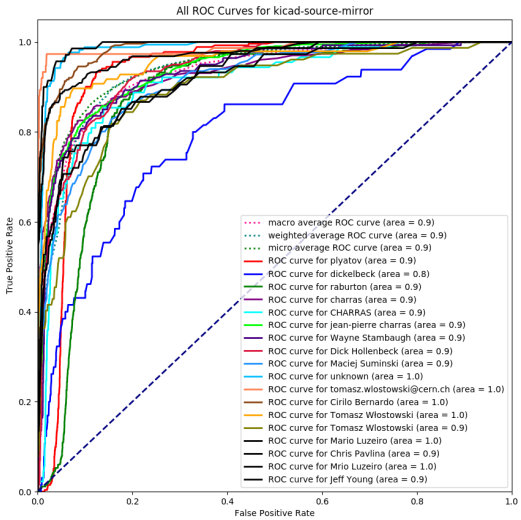


Figure 21: ROC Curves by Author for kicad-source-mirror

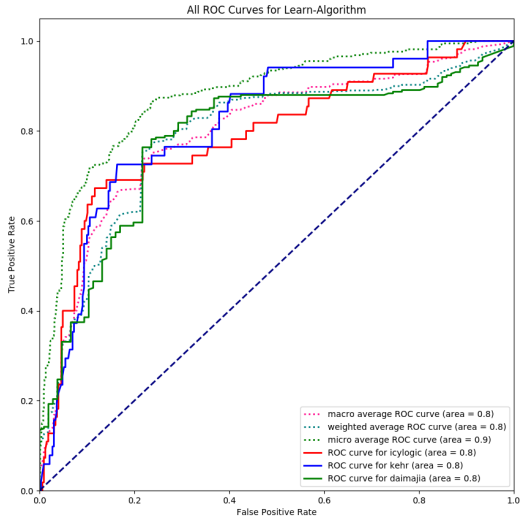


Figure 22: ROC Curves by Author for Learn-Algorithm

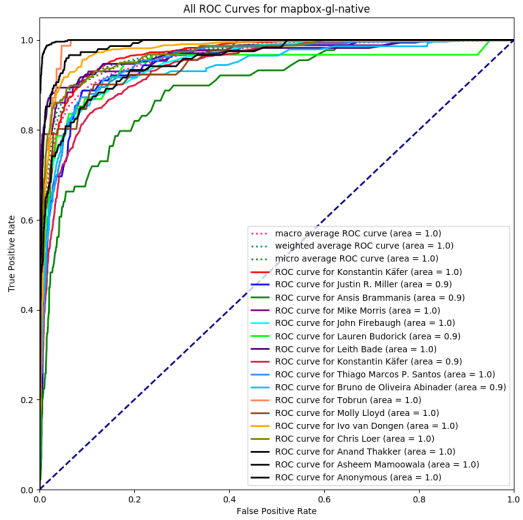


Figure 23: ROC Curves by Author for mapbox-gl-native

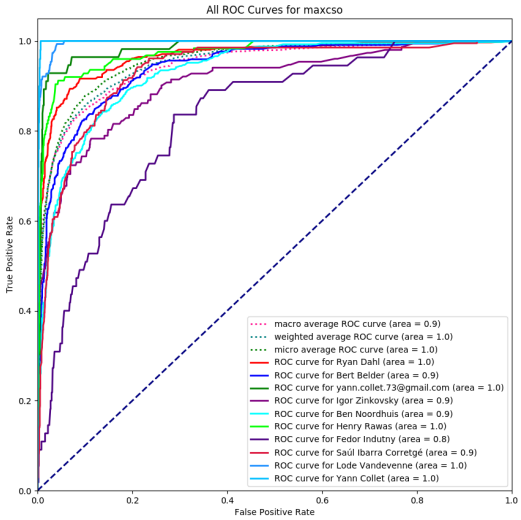


Figure 24: ROC Curves by Author for maxcso

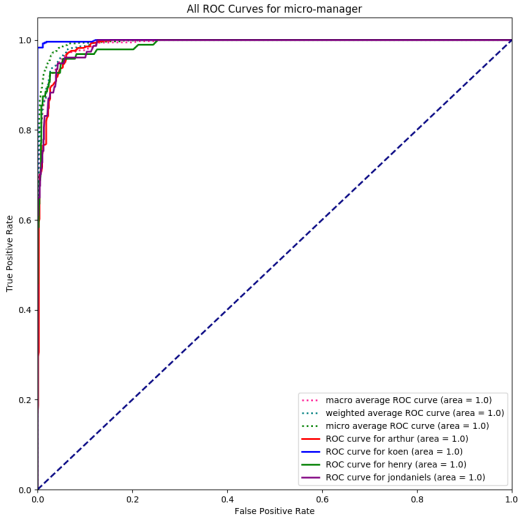


Figure 25: ROC Curves by Author for micro-manager

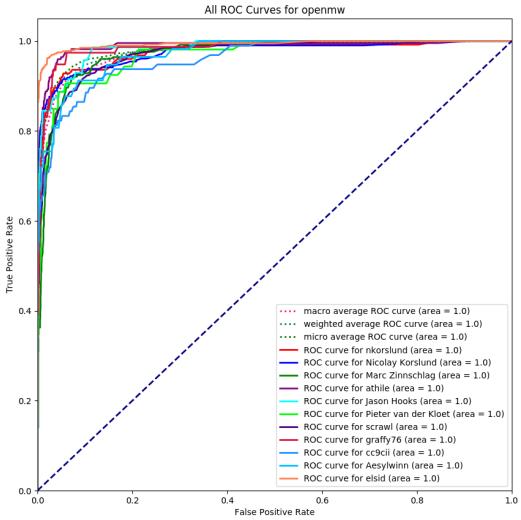


Figure 26: ROC Curves by Author for openmw

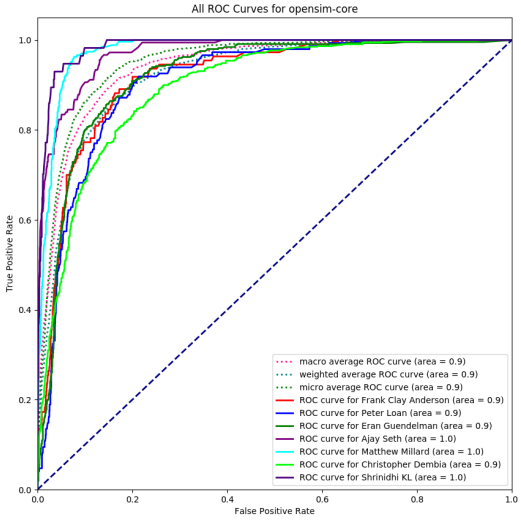


Figure 27: ROC Curves by Author for opensim-core

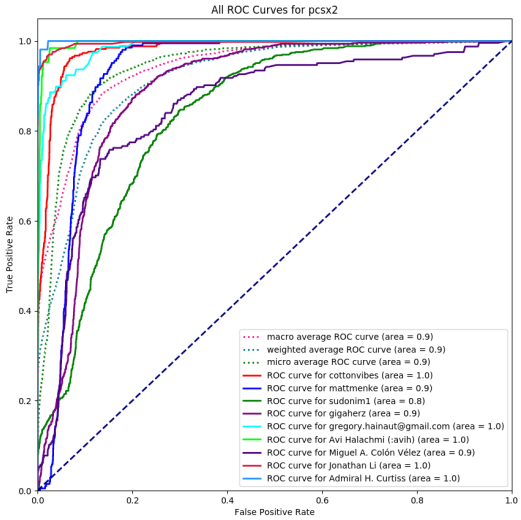


Figure 28: ROC Curves by Author for pcsx2

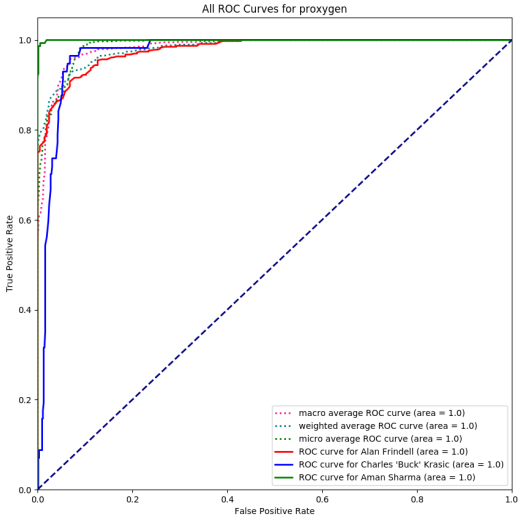


Figure 29: ROC Curves by Author for proxygen

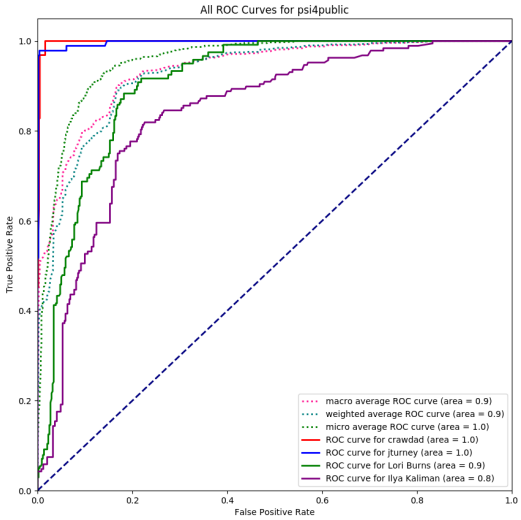


Figure 30: ROC Curves by Author for psi4public

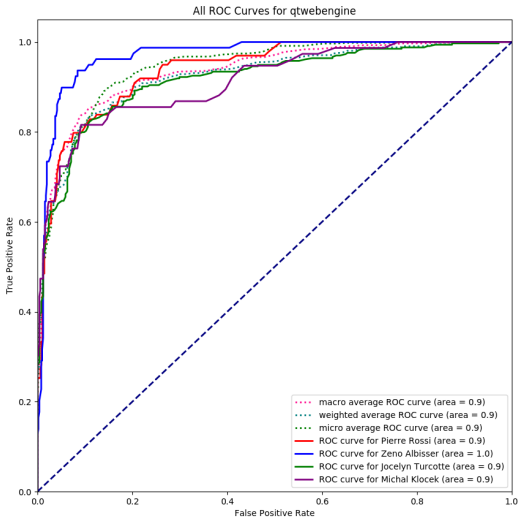


Figure 31: ROC Curves by Author for qtwebengine

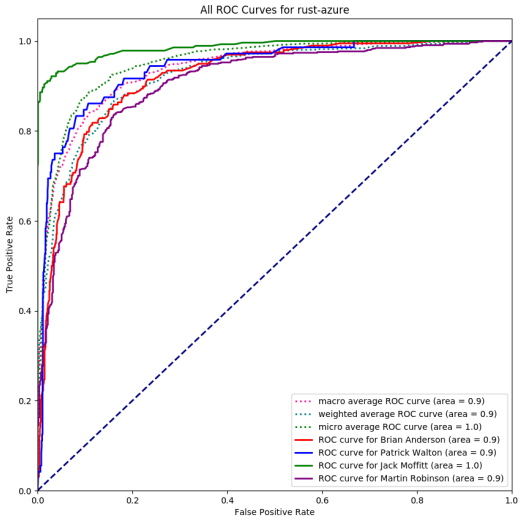


Figure 32: ROC Curves by Author for rust-azure

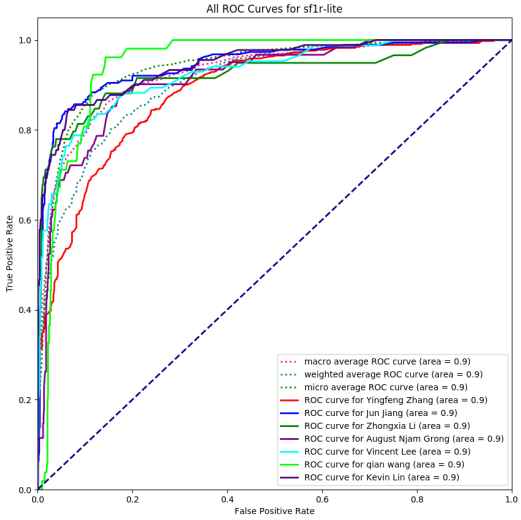


Figure 33: ROC Curves by Author for sf1r-lite

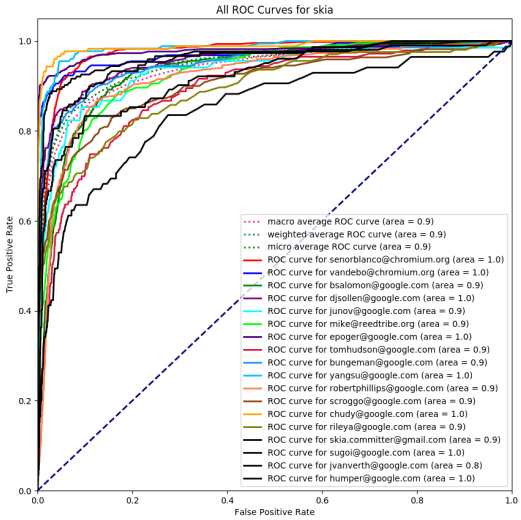


Figure 34: ROC Curves by Author for skia

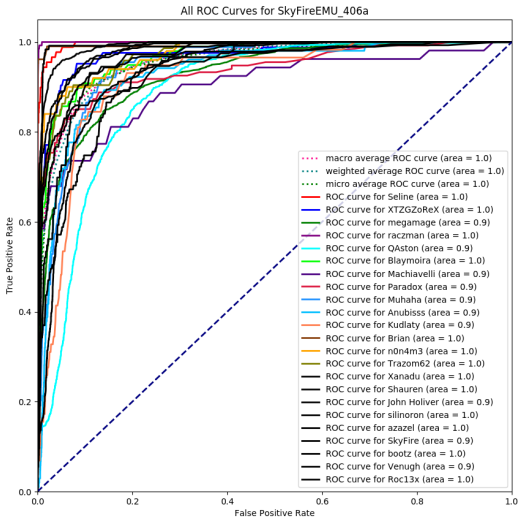


Figure 35: ROC Curves by Author for SkyFireEMU_406a

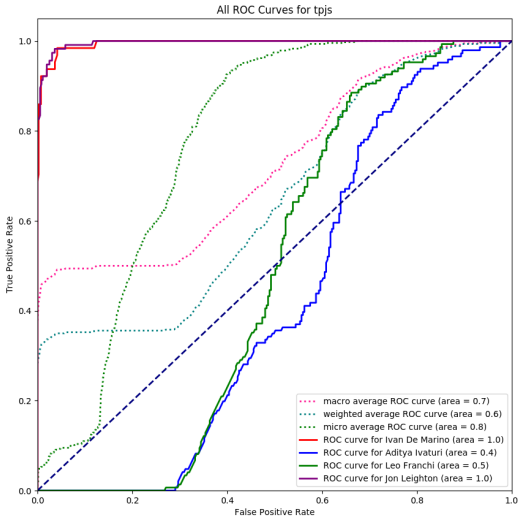


Figure 36: ROC Curves by Author for tpjs

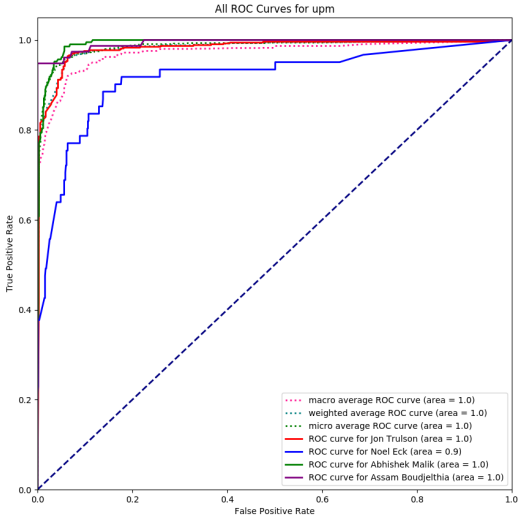


Figure 37: ROC Curves by Author for upm

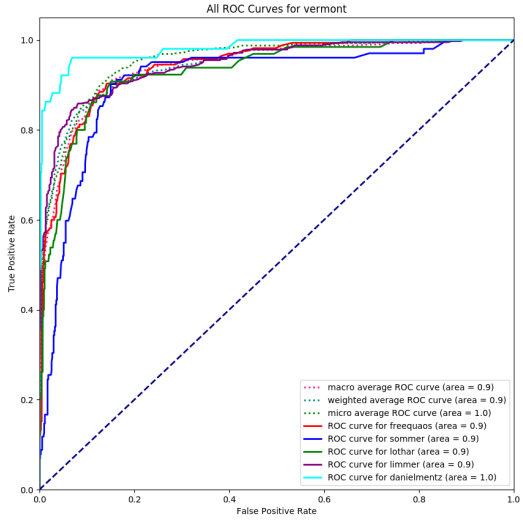


Figure 38: ROC Curves by Author for vermont