StyleCounsel: Seeing the (Random) Forest for the Trees in Adversarial Code Stylometry

by

Christopher McKnight

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, 2018

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

Authorship attribution has piqued the interest of scholars for centuries, but had historically remained a matter of subjective opinion, based upon examination of handwriting and the physical document. Midway through the 20th Century, a technique known as stylometry was developed, in which the content of a document is analyzed to extract the author's grammar use, preferred vocabulary, and other elements of compositional style. In parallel to this, programmers, and particularly those involved in education, were writing and testing systems designed to automate the analysis of good coding style and best practice, in order to assist with grading assignments. In the aftermath of the Morris Worm incident in 1988, researchers began to consider whether this automated analysis of program style could be combined with stylometry techniques and applied to source code, to identify the author of a program.

The results of recent experiments have suggested this code stylometry can successfully identify the author of short programs from among hundreds of candidates with up to 98% precision. This potential ability to discern the programmer of a sample of code from a large group of possible authors could have concerning consequences for the open-source community at large, particularly those contributors that may wish to remain anonymous. Recent international events have suggested the developers of certain anti-censorship and anti-surveillance tools are being targeted by their governments and forced to delete their repositories or face prosecution.

In light of this threat to the freedom and privacy of individual programmers around the world, and due to a dearth of published research into practical code stylometry at scale and its feasibility, we carried out a number of investigations looking into the difficulties of applying this technique in the real world, and how one might effect a robust defence against it. To this end, we devised a system to aid programmers in obfuscating their inherent style and imitating another, overt, author's style in order to protect their anonymity from this forensic technique. Our system utilizes the implicit rules encoded in the decision points of a random forest ensemble in order to derive a set of recommendations to present to the user detailing how to achieve this obfuscation and mimicry attack. In order to best test this system, and simultaneously assess the difficulties of performing practical stylometry at scale, we also gathered a large corpus of real open-source software and devised our own feature set including both novel attributes and those inspired or borrowed from other sources.

Our results indicate that attempting a mass analysis of publicly available source code is fraught with difficulties in ensuring the integrity of the data. Furthermore, we found ours and most other published feature sets do not sufficiently capture an author's style independently of the content to be very effective at scale, although its accuracy is significantly greater than a random guess. Evaluations of our tool indicate it can successfully extract a set of changes that would

result in a misclassification as another user if implemented. More importantly, this extraction was independent of the specifics of the feature set, and therefore would still work even with a more accurate model of style. We ran a limited user study to assess the usability of the tool, and found overall it was beneficial to our participants, and could be even more beneficial if the valuable feedback we received were implemented in future work.

Acknowledgements

This work benefitted from the use of the CrySP RIPPLE Facility at the University of Waterloo.

I would like to thank my supervisor Ian Goldberg for his guidance, encouragement, and resolute attention to detail. Our weekly meetings were a great help in nudging me toward the finish line, while the many opportunities for personal development offered throughout the duration of my programme were priceless. Truly my eyes have been opened to a world beyond that with which I was familiar, and I shall never look back.

To the members of my committee, Mike Godfrey and Yaoliang Yu, I am very grateful for your time, expertise and valuable feedback.

Being able to use the private study rooms at Waterloo Public Library while writing this thesis was priceless.

Finally, I would like to thank Radio X and all its DJs for keeping my sanity during long hours of coding and writing, particularly Johnny Vaughan and his "4til7 Thang" (even Little Si).

Dedication

For my loving and supportive wife Katya, whose tireless drive and work ethic was a constant inspiration, and our son James, whose companionship on many contemplative early morning walks helped get me though even the darkest hours of this arduous journey.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Authorship attribution is a topic that has been of interest to researchers and society in general for a considerable time. For example, in 1568, Mary, Queen of Scots, was believed guilty of murdering her second husband, Lord Darnley, largely due to evidence contained in the "Casket Letters", which naturally she claimed were forgeries [Bla34]. Despite considerable interest in the episode and attempts to analyze the content of the letters, there is no clear consensus and the destruction of the originals has left the true authorship of the letters an unanswered question. Furthermore, the authorship of twelve of the Federalist Papers [Ada74] previously claimed by Hamilton in 1810 were disputed by Madison in 1818, a claim which most researchers investigating the matter concur with [MW63, TSH96, HF95, BS98]. Finally, claims that Shakespeare did not really write his plays date back to the mid-19th century [EW13], although most experts do not subscribe to this idea and it is not generally taken seriously. Whether for the purposes of plagiarism detection, disputes over intellectual property rights, proving forgery of wills or other legal documents, criminal cases involving threatening letters or ransom notes or identifying cases of ghost writing or pen names, the use cases for authorship attribution are varied and diverse.

The dawn of electronic communications and the Internet precipitated a revolution in writing and self-publishing, greatly simplifying and vastly scaling the process of communicating between groups of people, allowing for the first time asynchronous communications on a massive scale almost instantaneously and at considerably lower cost. Whether sending email, blogging, participating in forums, IRC/chatrooms, instant messaging or social media, the possibilities for self-expression have been growing exponentially for the past 30–40 years. This explosion of data in the public domain, particularly written text, has resulted in even greater demand and interest in the identification of certain authors. The use cases for authorship attribution have grown to encompass new areas, such as cyber bullying [PH06, VVC08], fake news [CRC15, RCCC16] and more sinister purposes, such as the identification and subsequent persecution of political

bloggers [Boy05], among others. Additionally, new techniques have also been developed for analyzing this data with the assistance of computers, which is both a convenience and a necessity given the quantity of data involved.

Along with increasing the instances and quantity of written documents being published, the digital revolution has also changed what information can be inferred from writing samples. For example, there are no longer physical documents to examine, eliminating many of the cues used by forensic document examiners, such as handwriting analysis, type of paper, covert printer coding [Fou17], etc. That is not to say that less information can be inferred, however; on the contrary, in many cases more information is available for those with an advantageous position on the software platform, cloud, network, or physical computer to observe. Even those without privileged access can infer information that would not have been available before, such as dates and times of publication, document metadata, date of creation of the user account, as well as domain registration or possibly contact details for the author, such as email address or twitter handle. Moreover, there are many known techniques for the identification of users on a network that do not rely on the content of communications, even those taking measures to protect their identity, most of which are far beyond the scope of this work, however. One technique that is applicable to both traditional physical document analysis and modern, digital document analysis, is *stylometry*, a branch of which is the topic of this thesis.

Stylometry is defined as "the statistical analysis of literary style" [Hol98], and as such is concerned solely with the content of the document, rather than the method of delivery or the media containing it. Stylometry techniques tend to focus on syntactical features, such as choice of words, spelling, preference for certain grammatical structures and even layout. Stylometry has been the subject of a number of research studies looking at both closed- and open-class cases, real-world and fabricated data sets and use cases ranging from the canonical Federalist Papers disputed authorship [MW64] to more modern scenarios such as identifying the authors of tweets [BMA13]. While much research has been conducted on establishing stylometry as a viable method for the identification of authors, very little has been conducted into its resistance to conscious attempts at subverting it. Of the few studies that have been conducted, the results seem to suggest it is rather easy for an informed individual to thwart the analysis [KG06, BAG12], which may largely be a result of the underlying machine learning algorithms employed, that are known to be susceptible to evasion attacks [BNJT10], rather than a by-product of this particular problem domain.

Just as stylometry attempts to carry out authorship attribution through an analysis of the style in which a sample of writing has been composed, its equivalent in terms of software, *code stylometry* [CIHL+15], attempts to identify the programmer that wrote some sample of computer code through an analysis of their *programming* style. It achieves this by examining their source code or executable artifacts in order to discover common features that together may reveal a

"fingerprint" of the author's style, such as their preference for certain logical structures, data types, use of comments and naming conventions to name but a few. The origins of this idea date back to the early 80s when researchers and educators were interested in the analysis of coding style for the purposes of grading students' assignments or just assessing whether some program adheres to an agreed-upon standard of "good style" [Mee83, TC84]. Following the Internet/Morris Worm incident in 1988 [Orm03], a report was published that attempted to profile the author of the worm based on an examination of the reverse-engineered code [Spa88], casting style analysis as a forensic technique in addition to a code quality metric. This triggered a number of additional studies throughout the 1990s into using stylistic analysis as a forensic or deanonymization technique, even being dubbed "software forensics" at one point [Spa92].

Despite a substantial body of prior work on program authorship attribution, only one paper we are aware of [SZK18] (independent and concurrent work to ours) has yet investigated how robust the techniques are to adversarial modifications aimed at obfuscation of style or imitation of someone else's style, and how difficult or realistic this is. Our thesis statement is as follows:

Source code authorship attribution is fragile and prone to misclassification of files that have been deliberately altered to disguise one's identity. Furthermore, it is possible to deterministically derive a set of changes to elicit such a misclassification, but following this advice in a realistic setting can be difficult for users.

To that end, this thesis offers as contributions an examination of code stylometry techniques, along with the results of applying such techniques in a real-world setting. We developed a system named *Style Counsel* to assist programmers in obscuring their coding style and mimicking someone else's, achieved through an analysis of the decision trees contained in a random forest classifier. The name "Style Counsel" is derived from the intended behaviour of the system, to "counsel" users on their style.[1] We evaluate the effectiveness of our random forest analysis algorithm against a corpus of real-world data. Finally, we present the results from a pilot user study conducted to determine the ease with which a programmer can imitate someone else's coding style with and without the assistance of *Style Counsel*.

[1] *The Style Council* is also the name of a new wave pop band formed by Paul Weller in 1983 shortly after *The Jam* had split.

Chapter 2

Motivation

The threat to individuals' freedom and privacy online from both state and industry actors is growing year-on-year, resulting in an increasingly censored Internet and World-Wide Web. According to the Web Index 2014 report [JFF+14], 90% of the countries they surveyed became less free with regards to political journalism between 2007 and 2013. They stated:

"the overall environment for freedom of expression has deteriorated in the overwhelming majority of Web Index countries." [JFF+14]

They also highlighted the trend of declining press freedom in countries that had previously scored highly in this measure:

"in 14 countries, including the US, UK, Finland, New Zealand, and Denmark, scores fell by 20% or more."

While it should be pointed out that there is a limit to how low a country can score, which the worst scoring countries in previous years are probably approaching, it is still a concerning development that, rather than setting a positive example for others, the affluent West with its supposed ideals of democracy, liberty and egalitarianism seems to be in a race to the bottom with regards to freedom online with countries governed by oppressive regimes. Further evidence of a growing global censorship problem can be seen in Google's 2017 transparency report [Goo17], where the number of content removal requests received by Google from governments spiked to 22,515 in 2016 from 8,398 in 2015 and 6,845 in 2014, and over the same period user information requests from governments rose from 61,838 in 2014, to 76,042 in 2015 and 90,493 in 2016.

Looking in more detail at two specific countries, the US and UK, there have been some high-profile policy changes towards more surveillance, less corporate regulation and aggressive

denouncing of the use of strong encryption in certain applications. In December 2016, the UK Investigatory Powers Act 2016 came into effect [UK16a], which allows, among other provisions, for the warrantless seizure of Internet connection records from Connection Service Providers (CSPs) by a wide range of government departments, which includes the NHS, Ambulance Service and Food Standards Agency as well as various law enforcement and security/intelligence agencies among the roughly 50 departments listed. It is unclear to what ends these three agencies would need to access members of the public's browsing history and other metadata, nor why their needs would be so imperative and/or covert that they should bypass due process and judicial scrutiny. The act also compels CSPs to keep Internet connection records of their customers for one year and places a legal obligation on them, once served with a *"targeted equipment interference warrant"*, to *"take all steps for giving effect to the warrant"*[UK16b]. In March 2017, the US congress voted to repeal the Federal Communication Commission's (FCC) Broadband Consumer Privacy Proposal that prevented ISPs from storing their customers' metadata for the purposes of targeted advertising [Fun17]. This means that, as well as receiving a monthly subscription for Internet connectivity, ISPs will also be able to profit from harvesting and maintaining customer profiles containing personal and sensitive information, in order to target them with advertising. Furthermore, as there is a limited degree of competition amongst ISPs, consumers may have no alternative in their area but to use an ISP that monitors their activity. Following the Charlie Hebdo Paris attacks, in January 2015 then UK Prime Minister (PM) David Cameron called for *"the legal power to break into the encrypted communications of suspected terrorists"* [WMT15], then, in 2017, the Home Secretary Amber Rudd directly criticized WhatsApp for the use of strong encryption in its messaging app, saying:

"It is completely unacceptable. There should be no place for terrorists to hide. We need to make sure that organizations like WhatsApp, and there are plenty of others like that, don't provide a secret place for terrorists to communicate with each other." [Spa17]

Both these stances, along with the FBI's legal battle with Apple in 2015/16 [Gro16], failed to address either the privacy concerns of millions of innocent people, the technical and administrative infeasibility of implementing such a system and the security implications regarding criminals reverse-engineering and exploiting backdoors or gaining access to systems and databases containing private keys, certificates or other cryptographic artifacts used for access. Furthermore, the terms "criminal" and "terrorist" are subjective and dependent on whose laws and which perspective you hold, as well as changing over time. If one country is able to coerce companies into providing backdoor access or using deliberately weak encryption, regardless of how well intentioned their motives may be, it will set a precedent that other countries will follow, where being a human rights advocate or political opponent may be enough to be labelled a criminal or terrorist [GP16].

With these, and other, threats to individual freedom and privacy on the rise, it is no surprise

that interest in protecting their identity and circumventing censorship among both technical and non-technical users has risen markedly in recent years [Cam15]. A Pew Research Center report from 2013 found that *"86% of internet users have tried to use the internet in ways to minimize the visibility of their digital footprints"* and use of personal VPNs is at an all-time high [NI17], although it pays to carry out due diligence checks on the VPN provider before trusting it with your privacy and security [Rob17]. Depending on the threat model and with a responsible provider, VPNs are useful for preserving privacy against local adversaries, however. Anonymizing overlay networks such as Tor and I2P can provide for greater protections against more powerful adversaries. In addition to these well-known examples, a number of tools have been developed to meet the increased demand and changing threat landscape, in many cases published as open-source software by individuals who took little if any precautions to protect their identity. The proliferation and efficacy of such tools have prompted some authorities to clamp down on their usage and target the developers, who play a crucial role, often represent a single point of failure in the development and dissemination of the tool and, with few resources or legal backing, are easily intimidated.

A leaked Chinese police report, posted in a news article on the site globalvoices.org [Lam16] references the classification of certain circumvention tools by authorities in Xinjiang province as *"second class violent and terrorist software"*, in connection with the arrest of a citizen whose only crime was to download the software in question, believed to be a VPN. Compare the language used by the former Xinjiang Communist party chief Zhang Chunxian: *"90 percent of terrorism in Xinjiang comes from jumping the wall. Violence and terrorism keep happening due to the videos on the internet."* [Lam16], with the UK Home Secretary Amber Rudd's comments on WhatsApp, above. In the case of Chunxian a link is being made between the wider, uncensored Internet and radicalization, while Rudd links end-to-end encryption with helping terrorists to operate. In both cases, a clear link is being made between the use of anti-censorship and privacy-enhancing technologies and terrorism. This rhetoric helps to provide justification for legislation that targets the use of such technology and could position its developers as being part of a terrorist supply network, or even worse of collaborating or "arming" terrorists, which considering encryption used to be subject to arms export controls in the US [Cen92] is not as unlikely as it may seem.

There are several cases of developers being treated as individuals of suspicion, intimidated by authorities and/or coerced into removing their software from the Internet. In the US, Nadim Kobeissi, the Canadian creator of Cryptocat (an online secure messaging application) was stopped, searched and questioned by Department of Homeland Security officials on four separate occasions in 2012 about Cryptocat and the algorithms it employs [SOS12]. In November 2014, Chinese developer Xu Dong was arrested, primarily for political tweets supporting the occupy and umbrella movement in Hong Kong, but also because he allegedly *"committed crimes of de-*

veloping software to help Chinese Internet users scale the Great Fire Wall of China" [Cha14] in relation to software produced by his *"Maple Leaf and Banana"* brand, which includes a proxy for bypassing the Great Firewall. In August 2015, the Electronic Frontier Foundation (EFF) reported that Phus Lu, the developer of a popular proxy service hosted on Google's App Engine, called GoAgent, had been forced to remove all their code from GitHub and delete all their tweets on Twitter [O'B15]. This followed a similar incident reported on greatfire.org a few days earlier involving the creator of ShadowSocks, another popular proxy used in China to "scale the wall", known pseudonymously as clowwindy. According to the article reporting this incident, clowwindy posted a note afterwards that said: *"the police contacted him and asked him to stop working on the tool and to remove all of the code from GitHub"* [Per15], which was subsequently removed. The README file for the project now simply says *"Removed according to regulations"*. Earlier in March 2015, GitHub was subjected to *"the largest DDoS that they have ever dealt with"* [Bud15], which has been linked to the Chinese government [BLL+15] and has been suggested was in an attempt to bully the site into removing repositories that contravened their censorship regulations. As GitHub uses HTTPS, which encrypts the payload of the HTTP request, including the actual resources being requested, it is hard to discern which repositories are being requested and possibly easier to try to persuade GitHub to remove the offending material.

As the environment turns hostile towards the developers, many of them may opt to disguise their identity and authorship attribution techniques such as code stylometry could be deployed in order to identify them from other code they may have published using their real identity. Even the threat of such techniques could be enough to instill a chilling effect in open-source contributors who otherwise may have been willing to contribute their time and effort into assisting with censorship resistance tools and privacy-enhancing technologies.

Chapter 3

Background

Of the applications of authorship attribution, none has been as influential, yet so wildly different in nature, as the disputed Federalist Papers and the Internet worm incident. The disputed Federalist Papers have become something of a benchmark dataset in text-based authorship attribution, consisting of a large corpus of mostly single-author writing where most of the papers are known to have been written by one of three authors, with the authorship of twelve papers being disputed between two of the three authors. The Internet worm incident was a major computer security event in 1988 that brought the Internet to its knees and sparked a manhunt for the author(s) of the software responsible, prompting a forensic analysis of the program binary in which the behaviour of the worm was documented and characteristics of the author(s) were guessed at, giving rise to the practice of malware analysis and software authorship analysis.

3.1 The Federalist Papers

The Federalist Papers were written and published in 1787 and 1788 by three authors, Alexander Hamilton, James Madison and John Jay. Initially they went by the collective pseudonym of "Publius" (although, it should be pointed out, most of the papers were written singularly, not collectively, by each author) and the group's real identity was not revealed until a French language version named them in 1792 and subsequently an American edition named them in 1802 [Ada74]. The aim of the papers was to convince the American public to support the ratification of the US constitution. It is not clear whether the use of a pseudonym was due to security concerns by the authors or so as not to unduly influence the reader with familiar names that may impart either a positive or negative bias in their interpretation of the arguments set forth; i.e., to make the writing seem more "pure".

Of the 85 papers, 51 are now credited to Hamilton, 29 to Madison and five to Jay, although the authors themselves originally maintained that they were written collectively, until 1810 when a list compiled by Hamilton before his death in 1802 was used as the basis for crediting each paper with a (single) author. Originally, this list asserted that Hamilton was the sole author of 63 papers, however Madison disputed these assignations in 1818 with his own list that put him as the sole author of 29 and Hamilton 51 papers, respectively. Madison diplomatically suggested that Hamilton had made a mistake *"owing doubtless to the hurry in which [Hamilton's] memorandum was made out."* [Ada74].

While there were historical studies of these events in the following years that claimed either Hamilton or Madison as the authors of the disputed papers, it was not until 1963 that Mosteller and Wallace took a scientific approach to the problem [MW63]. This is perhaps the best-known and highly cited paper examining the Federalist Papers from an authorship attribution perspective, and since then the Federalist Papers have been used many times as the dataset for various authorship analysis studies using statistical methods.

There are several good reasons why this case has become the *de facto* dataset for authorship attribution research. First, the data is easily accessible, being publicly available online. Second, the writing is of historical and cultural significance in the US, where much of the research is conducted. Third, it involves a genuine case of disputed authorship, *which has never been conclusively resolved*, for while statistical studies such as these certainly lend weight to one side or the other they do not provide incontrovertible proof. Fourthly, there is a large corpus of ground truth data based on the undisputed papers and presumably other works these authors wrote during their lives. Finally, there are only three (or two if you assume Jay wrote none of the disputed papers based on his silence over the matter) authors to consider, greatly simplifying the task at hand.

One reason why this dataset may not be a good choice for testing new techniques of attribution is precisely because the authorship of the disputed papers is **not** known conclusively. Graham *et al.* [GHM05] claim that: *"Juola (1997) demonstrated a technique that, using samples as small as 500 characters, can correctly classify all the disputed Federalist Papers"*; however, without the original authors and no further evidence to settle the dispute, this is not a claim that should be made, especially when one considers that the authorship labels that have been assigned to the disputed papers in modern times are themselves based on statistical analyses of style. All that could be claimed is that this technique concurred with the other statistical studies for the disputed papers. An additional issue with basing a research study and statistical style analysis on the Federalist Papers alone is they were written during a specific period by three authors with similar educational backgrounds who were trying to write in a certain style, the *spectator* style, described by Mosteller and Wallace as *"complicated and oratorical"* [MW64]. In order to extract an essence or measure of writing style that is universal and applicable in practice, it would

be desirable to include samples of writing from many authors, on a variety of topics, over a period spanning decades, if not centuries. Many of the features discovered as being significant for discriminating between Hamilton, Madison and Jay, may not apply to more modern writing, or may not scale with the number of authors. For example, Mosteller and Wallace included variance between uses of the words *while* and *whilst*, but in modern usage *while* massively outnumbers *whilst*.

3.2 The Morris/Internet Worm

The Internet, or Morris, Worm as it is alternatively known, was a computer worm[1] launched in November 1988 by Robert T. Morris that shut down a large proportion of the Internet as it propagated, consuming bandwidth on the network and CPU time on any host it infected (often multiple times), preventing them from running any other processes. After systems were patched and the worm was purged from the Internet, the hunt for the author(s) of the worm began in earnest. One of the most prominent figures in the coordination effort, Eugene Spafford, published a report within weeks that carried out a technical analysis of the worm and a profile of its author(s) based on how they had written the code [Spa88]. The analysis was conducted on two decompiled and one disassembled version of the worm binary and is primarily concerned with the vulnerabilities the worm exploited as well as its structure and behaviour, however there is also a section towards the end that critiques the programming style the worm was written in and makes inferences about the author's level of expertise, their intentions, programming experience and even mental state. It should be pointed out that Morris was already a suspect by this time, having revealed one of the vulnerabilities (a buffer overflow[2] in the *fingerd* process) to staff at Carnegie Mellon University the previous year, and was eventually arrested, tried, convicted under the 1986 Computer Fraud and Abuse Act and sentenced to three years' probation in 1990 [Mar90].

The analysis and subsequent report about the Internet Worm was significant in two respects: First, it represents one of the first published malware analyses. At the time, antivirus software as a concept was still in its infancy, only just emerging from being a purely research topic to commercial endeavours [Coh87, Coh88] and the Common Vulnerabilities and Exposures (CVE)

[1] A worm is a type of malware that is able to spread from computer to computer, typically over a network, without requiring any action on the part of the user, making them far more virulent than other types of malware.

[2] A buffer overflow is caused when input data that is written to memory is too large for the data structure that has been allocated to contain it. In this case, without proper bounds checking, an attacker can overwrite other memory locations immediately following the buffer on the stack, including the return address pointer, effectively allowing them to control execution flow and redirect the instruction pointer to their own carefully constructed code contained in the input data.

database,[3] which systematized the dissemination of information on software vulnerabilities, was over ten years away from being realized. Second, it precipitated an interest in authorship attribution of software amongst the research community. Clearly, there was an appetite amongst policy makers and law enforcement to apprehend and punish computer criminals, including malware authors, to act as a deterrent to others, and academia, reliant as it is on sharing data and use of the Internet for collaboration, also had a vested interest in curbing behaviour such as this.

[3] https://cve.mitre.org

Chapter 4

Literature Review

As a research topic, authorship attribution has been, and continues to be, popular, with dozens of papers published each year since 2007. Indeed, rather than becoming exhausted, interest has in fact been growing year-on-year to 99 papers published in 2016.[1] As might be expected, most of the research has been from the field of Computer Science (40%), with Linguistics (17%) and Literature (15%) also featuring prominently.

In the sections that follow, we first look at relevant papers in the realm of authorship attribution of natural language texts, its applications and techniques. Following this, we will discuss papers in the area of plagiarism detection and the closely related problem of authorship validation that are relevant to authorship attribution. Next, we will review research conducted into software authorship attribution on both source code and binaries, before progressing to a discussion of defences, including adversarial approaches to defeating authorship attribution.

4.1 Authorship Attribution of Natural Language

4.1.1 Early Work

Before computers were available to researchers as a tool for processing large quantities of data and performing rapid calculations, studies in authorship attribution were characterized by their consideration of only a single metric/feature at a time. This is because the size of the datasets involved means a great deal of time has to be invested just to catalogue and categorize the data,

[1] Data obtained from Web of Science (https://webofknowledge.com)

leaving little appetite to calculate more than one measure. One of the earliest known scholarly works looking at statistical features for author identification was published in 1887 by Mendenhall [Men87]. The paper seeks to position average word length distributions as a distinguishing characteristic, crediting the mathematician Augustus De Morgan for inspiration, and analyzes the works of a number of authors to produce example distributions. The intuition behind this idea is that the distribution carries information as to the preferred vocabulary of the author. There are hundreds of thousands of words in the English language, but only 28 categories by word length[2], with the majority being between four and eight characters in length. It is hard to imagine a measure that compresses this much information into such a simple representation, even when presented as a distribution rather than just the raw mean value, could be very effective as a discriminator when taken over all but the most trivial of samples. The paper presents some example distributions (termed *characteristic curves of composition*) from the selected authors, however there are significant similarities between the distributions. Brinegar [Bri63] applies this same technique to the *"Quintus Curtius Snodgrass"* letters that are believed to have been written by Mark Twain under a pen name. The results of this study suggested that Twain was not the author of the ten letters in question. This result is cast into doubt by more recent research, however, as Holmes [Hol94] writes that Smith[3] [Smi83] found the average word length varied greatly depending on the genre and era in which it was written (and presumably the intended audience and topic also influenced the measure), by a far greater degree than it varied by author, rendering the approach ineffective for attribution.

A contemporaneous study to Brinegar's by Mosteller and Wallace [MW63] looked at other statistical measures that may capture consistent differences between authors, applied to the Federalist Papers case. The authors used Bayesian inference based on the usage of certain high-frequency function words, such as articles, prepositions and conjunctions, which they selected based on their non-contextuality and common usage, meaning that they would appear in all works by the authors and not be dependent on the subject of the paper. They also documented their efforts with using other sets of words, including those sourced from frequency lists of large, unrelated corpora and the federalist papers themselves. They found that the function words provided the best discriminatory power. As previously noted in Section 3.1, this work was seminal in the field of authorship attribution and represented a herculean effort by the researchers who, not having access to computers, had to perform their analysis by hand.

Holmes [Hol94] cites Bailey[4] [Bai79] as attempting to formalize the attribution process as a forensic discipline with conditions given for the candidate author set, size of samples and the

[2]The longest non-technical, uncoined word found in dictionaries is *antidisestablishmentarianism* at 28 characters.

[3]Paper could not be sourced

[4]Paper could not be sourced

type of features that should be considered. Holmes [Hol94] also conducts an exhaustive overview of the many metrics that were proposed in the early literature, including word lengths, number of syllables, sentence length, type of speech, function words, information entropy and so on. The general consensus with several of these features, such as word length (discussed above) and sentence length, is they are not representative of authorial style and should be discarded. But in many cases, there are conflicting results, depending on the text in question and the corpus being analyzed, its age, genre, etc. This disparity and lack of a clear consensus of features that produce consistent results is concerning for the discipline as a whole and suggests writing style may be too complex to be summarized in one or just a few features. Smith [Smi90] exposes some inconsistencies in four papers and offers the following cautious advice on the use of authorship attribution outside of a purely academic context:

1. The onus of proof lies entirely with the person making the ascription.
2. The argument for adding something to an author's canon has to be vastly more stringent than for keeping it there.
3. If doubt persists, an anonymous work must remain anonymous.
4. Avoidance of a false attribution is far more important than failing to recognize a correct one.
5. Only works of known authorship are suitable as a basis for attributing a disputed work.
6. There are no short-cuts in attribution studies.

4.1.2 Computer-Assisted Studies

With computers becoming more widely accessible to researchers, statistical analyses of large texts could more easily be accomplished, with a higher dimensionality in the features extracted, leading to the use of multivariate techniques, machine learning and AI. Holmes and Forsyth [HF95] revisited the Federalist Papers with three new stylistic measures combined with three modern analysis methods to compare their effectiveness with the original Mosteller and Wallace paper. The first technique looked at a six-variable system where each variable was a measure of vocabulary richness. This system was proposed in a previous paper [Hol92]. The six variables were:

1. Honoré's *R*-function [Hon79], calculated from the total number of words in a sample, how many words appeared just once (*hapax legomena*) and how many unique words a sample of text contains.

2. *Hapax Dislegomena* (the number of words appearing twice in a sample), divided by the total number of words, as suggested by Sichel's work [Sic86].

3. Yule's Characteristic *K* [Yul14], which assumes word selection in any given text sample are events that can be modelled with a Poisson distribution. Holmes points out that the efficacy of this measure for authorship attribution had previously been questioned by Tallentire [Tal72] and not recommended for univariate studies, but as this was a multivariate approach, decided to include it.

4. Brunet's W index [Bru78], which is simply $N^{V-\alpha}$, where *N* is the size of the sample, *V* is the number of unique words in the sample and α is a constant term $\alpha = 0.17$ (Brunet suggested $0.165 \leq \alpha \leq 0.172$).

5. & 6. The α and θ parameters of the Sichel Distribution [Sic75], a distribution based on observed word frequencies that estimates the probability that a word was observed exactly *r* times (in this case, where r is the actual observation) given the size of the sample, *N*. Pollatschek and Radday [PR81] showed that these parameters control the head and tail of the distribution, respectively.

These scale well with the size of the sample as the terms in the formulae are dependent on the total number of words *N*, and due to Zipf's law [Pow98] the proportion of hapax legomena/dislegomena remains approximately consistent with changing *N*. To confirm that this was the case, however, the authors calculated correlation coefficients for each variable with respect to the length of the corresponding text samples and found only one variable had a significant correlation, the *Hapax Dislegomena*, which they subsequently removed from the study. To these vocabulary richness variables they applied a principal component analysis, the results of which showed a clear divide between Hamilton and Madison, with Hamilton demonstrating a richer vocabulary in his papers and the jointly authored documents demonstrating the richest vocabulary of all, leading Holmes and Forsyth to wonder whether collaboratively written documents always demonstrated a richer vocabulary.

The second technique they evaluated used principal component analysis applied to word frequency lists of function words (a similar set of words as used by Mosteller and Wallace), a technique previously proposed by Burrows [Bur92]. The final technique used a genetic algorithm approach to automatically detect patterns, using the BEAGLE system [For81]. Holmes and Forsyth described this system as an early precursor to genetic programming [Koz92], as it is able to automatically derive rule sets, each rule of which may be equated to a Boolean expression in a larger program.

In all three approaches to the problem, Madison was indicated as being the most likely author of the disputed papers. They found the first two techniques produced clear clustering, that, being constructed from a principal component analysis, were easily discernible to the human eye. They found the second technique, the word frequency analysis, worked best with larger lists of the common function words, and gave worse results with the original 30 function words used by Mosteller and Wallace. In contrast, they concluded that the genetic algorithm approach worked better with the original 30 words. They also noted that evolutionary computing approaches to authorship attribution had been underutilized.

It does not appear that this comment regarding the underutilization of evolutionary computing in authorship attribution studies was heeded by the research community as no other papers appear to have investigated this approach, while machine learning techniques have come to dominate. Tweedie *et al.* [TSH96] evaluated the use of neural networks (NN) applied to the federalist papers. The inputs to their NN were word frequencies of a subset of eleven function words from the original 30 used by Mosteller and Wallace, chosen for having the most discriminatory power. They left considering other features as future work, choosing instead to carry out this initial study with the function words that were already proven to be effective, which also enables a better comparison to be drawn between the results of the two studies. Their results were consistent with other studies on the federalist papers with regards to attribution of the disputed papers.

Diederich *et al.* [DKLP03] investigated using Support Vector Machines (SVM) applied to a corpus of articles from the German newspaper Berliner Zeitung covering a three-month period from December 1998–February 1999 of politics, economics and local affairs categories with lengths exceeding 300 words. This gave them a corpus of 2,652 documents containing 1.9 million words, approximately 120,000 of which were unique. With this corpus, they then trialled two different feature extraction methods. The main difference between the first method they tried and any others hitherto used was the authors did not attempt to condense or summarize the information contained in the word frequencies to only a small subset of chosen words, word types, hapax (dis-)legomena, etc., but rather they used the frequencies for all the words found in the texts. This is an advantage of SVM and other classifiers that use kernel methods. Kernel methods allow for the use of raw feature values, without requiring them to first be converted to feature vectors, providing some form of similarity measure has been defined for evaluating the relative differences between values. This means feature vectors of very large dimensionality can be processed efficiently, eliminating the need for feature reduction. They experimented with several different frequency calculation methods, including relative frequency (the standard measure one associates with frequency) and *term frequency-inverse document frequency* (tf-idf) a measure used in information retrieval to rank documents relative to some search terms by their importance or relevance. This is achieved by first calculating the frequency of the term in the document (term frequency) and scaling it by the total number of documents divided by how many

documents contain that term, expressed logarithmically. The net result of this is that common words that appear in every document; e.g., "the", will have a tf-idf of 0 (because it appears in every document, hence the idf value will be $\log(1) = 0$), and terms that are rare overall but appear often in the current document will have high tf-idf values. Their second feature extraction methodology took over 800 function words appearing more than nine times in the corpus and tagged them according to their syntactical role and calculated the relative frequency of the tagged words and bigrams of the tagged words. Their conclusions were that SVMs were:

"especially suited for this task as the feature spaces have very high dimensions, most features carry important information and the data for specific instances is sparse." [DKLP03]

They highlighted the fact that when the feature space does not need to be compacted, as with SVMs, choosing a subset of words, such as function words, to focus on during feature extraction results in worse performance. This makes intuitive sense, as there will be a loss of information when considering only part of the content versus the entire document; however this is not true for non-kernel method classifiers where there is a trade-off between improvement in performance from the information gained with additional features and degradation in performance from high-dimensionality. As SVM does not suffer from the curse of dimensionality [RNI10] this trade-off does not need to be made, although caution should still be exercised here to avoid the classification being based too much on the topic rather than the author. Certain words are going to be more correlated with topic than others, particularly nouns and to a lesser extent verbs. As this study was focused specifically on articles about the same three topics, and journalists typically write articles about only one topic, this effect may not have been apparent with this particular dataset, but could become problematic with other datasets.

Luyckz and Daelemans [LD05] also investigated applying machine learning to a corpus of newspaper articles; this time the source was De Standaard, a Dutch-language Belgian newspaper, and two authors in particular. Their study differed from Diederich *et al.* in the method of feature extraction and classifiers used. For feature extraction they used *shallow text parsing* with the Memory-Based Shallow Parser [DVdB05], which involves chunking [Abn91] sentences into lexical units and labelling the tokens within each chunk according to their word type and syntactical purpose. With this parsed data, they created nine different feature sets based on frequency distributions of parts-of-speech, basic verb forms, verb forms, noun phrase patterns, function words and the 20 most informative[5] words, as well as a readability score, a combination of all the feature sets and a combination of the syntax-based features and readability score. Regarding the machine learning models they used Weka's[6] Neural Network classifier and TiMBL [DZVDSVDB04], a type of memory-based learning algorithm, similar to *k*-NN. Based on a three-class problem (au-

[5] Informative in this respect refers to a measure of *mutual information*. [Seb02]

[6] https://www.cs.waikato.ac.nz/ml/weka/

thors A, B and O for "other"), their results indicated that the combined feature set performed best with an average F_1 score of 71.3%, with the combination of syntax-based features and readability scoring 61.7%. Of the single syntax-based feature sets, the parts-of-speech frequency distribution scored best with 50.6%, and the function words performed best of the lexical features with 63.9%, which was also the second best overall.

With much of the work performed in isolation against disparate data, it is not easy to determine which features and which analysis technique is best suited to authorship attribution. Grieve [Gri07] attempted to resolve this question with regards to the best features by testing 39 different feature sets independently with the same dataset using the same classification algorithm. The 39 feature sets fall under the following categories:

- Average word length (2 sets)
- Average sentence length (4 sets)
- Vocabulary richness (11 sets)
- Character frequency (4 sets)
- Word frequency (3 sets)
- Punctuation frequency (5 sets)
- Word 2- and 3-grams (2 sets)
- Character n-grams (8 sets)

For the corpus, opinion columns in the British newspaper *The Telegraph* between 2000 and 2005 were used. This was to satisfy the requirement that the intended audience the author was writing for remain stable and the time frame is narrow enough that the author's style is unlikely to evolve much during the period. The researchers also attempted to control for the demographics of the authors by selecting columnists with similar age, political opinion, ethnicity, class, education and nationality. They noted that some of the authors had different social backgrounds even if overall the demographics of the group were consistent. In total, then, the corpus of newspaper columns represented the writings of 40 authors. For evaluation, a succession of leave-one-out tests were conducted where an author and one of their texts were selected at random, before carrying out a chi-squared analysis to determine the confidence that the text in question was drawn from each author's corpus. Then, the output values from the chi-squared test were ordered in ascending order and the lowest value was taken to be the most likely author according to the

feature being evaluated. The precision of this test (total successful predictions divided by total predictions) is then reported as the success rate of that particular feature. The evaluation is carried out on all forty authors, then on permutations of twenty, ten, five, four, three and two authors, the results of which are averaged for presentation. The results of the study suggest features based on punctuation have the most discriminatory power, with character n-grams also producing favourable results. Average word and sentence lengths performed poorly, which is consistent with other studies, but vocabulary richness measures also produced poor results, which contrasts with Holmes and Forsyth's earlier work [HF95] that had positive results, albeit when using the vocabulary richness measures in a multivariate, rather than a univariate system. It should be noted that all of the features tested produced better results than random guessing, and combining measures that represent different aspects of the content, such as punctuation with character n-grams, would almost certainly improve the predictive power. The authors of this study go on to perform a combination test where the best 16 features were selected and two voting mechanisms were implemented, the first where each feature had an equal vote and the second where features' votes were weighted depending on their performance in the individual evaluations. The results of these combination tests demonstrated that considering multiple features does indeed improve the precision overall, especially when weighting the votes of each individual feature by its own predictive power.

Jockers and Witten [JW10] also conducted a comparative study, this time however, they were interested in determining which machine learning algorithm was best suited for authorship attribution. For their tests, the feature sets and author data were kept consistent so a controlled comparison could be made between the classification algorithms. The corpus used in this paper was the federalist papers and two feature sets were evaluated, one containing all words and word bigrams common to all three authors (dimensionality 2,907) and the second containing words and word bigrams that were found with a minimum frequency in the texts (dimensionality 298). Five classification methods were tested: Delta [Bur02], k-NN, SVM, Nearest Shrunken Centroids (NSC) [THNC03] and Regularized Discriminant Analysis (RDA) [Fri89]. Delta is an algorithm that was designed expressly for the purpose of authorship attribution, while the other methods are general classifiers. The procedure for using Delta involves a feature selection step, therefore it was not run on the full 2,907- and 298-dimension feature matrix, but rather on a subset of lower dimensionality. NSC and RDA also perform dimension-reduction, but k-NN and SVM are run on the entire feature set. Their evaluation was carried out using 10-fold cross validation and came to the conclusion that nearest shrunken centroids gave the best overall performance as there were no classification errors in the cross validation for either the full or reduced dataset, whereas all the other classifiers experienced at least one misclassification. SVM performed particularly badly and gave the worst performance overall, which is in contrast to the conventional wisdom on authorship attribution (and other Natural language Processing—NLP—applications) where

SVM is considered to be a useful algorithm due to its kernel-method nature as discussed earlier in this section. This result should merit further investigation, possibly using a different evaluation method rather than cross validation or a different feature set, which could have a big impact on overall performance.

Overall, we can see that the use of computers enabled researchers to conduct studies involving far more complex statistical models, with multivariate systems and machine learning over sometimes thousands of variables, whereas before the advent of computers in this line of research, studies typically focused on just one or two features at a time. It is interesting to note, however, that the size of the datasets typically were no bigger than the earlier studies, despite the relative ease with which a computer would be able to process larger corpora compared with a human. Indeed, often the classic federalist papers corpus was used, cementing its position as the standard by which new features and algorithms would be benchmarked against. We will see in the next section, however, that more recent papers have begun exploring datasets of varying sizes, both in the quantity of samples, the number of authors and length of each sample.

4.1.3 Internet-Scale Authorship Attribution

In this Internet age, we see a trend in authorship attribution research for papers focusing on two factors that underline the challenges of the modern era: large quantities of data, particularly with respect to the number of authors, and smaller sizes of individual samples. There is also a shift in the applications, both implied and explicit, from authorship disputes; e.g., for copyright purposes, to criminal investigations; e.g., spam/phishing emails or cyberstalking/trolling.

Koppel *et al.* [KSAM06] first looked at this problem, using a corpus consisting of blogs (where each blog refers to all the posts in that blog, rather than a single blog entry) from 18,000 authors. In their first experiment, they took 10,000 blogs and their last n posts that together constitute at least 500 words (they referred to these as "snippets") were used as the test set and the remaining posts used for training. With these training/test sets, they applied several metrics from the information retrieval arena, which were:

- tf-idf over content words
- "binary" idf over content words[7]

[7] It is not clear what is meant by binary idf; no reference was provided and the term was not found in any other paper. It could be referencing a system where each term is either present (1) or not present (0) in the corpus, although in this case it would not be an inverse frequency.

- tf-idf on *"stylistic features (function words and strings of non-alphabetic, non-numeric characters)"*[8]

They evaluated these metrics by ranking the authors by how highly they scored for each snippet. The number one ranked author was taken to be that metric's classification. They reported their results as a type of cumulative distribution frequency where the x-axis represented the rank r and the y-axis the percentage of snippets for which the correct author was ranked in the top r ranks. They found "binary" idf performed best, with 42% of snippets placing the correct author as the number one rank, increasing to approximately 50% when $r = 10$. In their second experiment, they used a *meta-learning* approach [VD02] to enable an estimation of the degree of confidence in a particular classification. They achieved this through training an SVM classifier on various metadata about each snippet and the ranked authors, such as the absolute similarity between the snippet and the top ranked author, the degree of similarity between each of the top k authors and a comparison with the rankings produced by the other metrics (tf-idf on content and stylistic features). The output of this meta-learner was used by the main classifier to decide whether to attempt a classification or to return "Don't Know" [KSAM06]. They tested this meta-learning approach on the remaining 8,000 blogs that were not part of the first experiment and were able to report, for a limited subset of 31.3% of cases that it attempted a classification it correctly predicted the true author (number one rank) in 88.2% of cases.

Luyckx and Daelemans [LD08] conducted a study with 145 authors and (relatively) short passages of around 1,400 words sourced from student essays on a particular topic (a documentary on artificial life), which were subsequently split into ten equal parts. Their aim was to investigate the effect of increasing the number of authors being considered on the classification accuracy, to determine which classifiers perform well with limited sample sizes, experiment with different feature sets and compare authorship verification, which is the binary-class version of authorship attribution, with the standard multi-class scenario. They used a chi-squared analysis to perform feature selection/reduction (to 50 features) on word and part-of-speech *n*-grams that had been elicited by the Memory-Based Shallow Parser (see Section 4.1.2) [DVdB05], function word frequencies and two vocabulary richness measures in the Flesch-Kincaid readability score and type-token ratio. To evaluate the effect of increasing numbers of authors they randomly generated 100 permutation subsets of two, five and ten authors, then one random subset of 50 and 100 authors, before finally testing with all 145 authors using the TiMBL [DZVDSVDB04] memory-based learning algorithm. To evaluate which classifiers are best suited to limited sample sizes, they incrementally increased the size of the corpus, performing 5-fold cross validation at each increment with the following classifiers: TiMBL, Maxent [Le04] and SVO, a variant of

[8] Again, no further explanation is offered for which function words or how many they used, nor for how the strings of non-alphabetic and non-numeric characters were derived.

SVM implemented in Weka. All results were reported with respect to different feature sets, one containing the word and part-of-speech *n*-grams, one containing the function word frequencies and one containing the vocabulary richness measures, as well as combinations of these. For the authorship validation evaluation, they simply took each author in turn and labelled all other authors with the negative class, again performing 5-fold cross validation. Their results for the increasing number of authors experiment found that there was a sharp drop in the F_1 score once the number of authors being considered was greater than 20. They also found that as the number of authors increases, there is a greater performance improvement from including more features than with fewer authors, which would be expected, as the more classes there are to differentiate, the more information is required to differentiate between them. When experimenting with limiting data to different degrees, they did not find a clear difference between using the three different classifiers; they had theorized that the lazy memory-based learner TiMBL would outperform the eager Maxent and SVO classifiers with less training data. For the author verification problem, where the task is to determine if a sample being assessed was written by a single target author or by someone (anyone) else among the 144 other authors, i.e. not the author, they found memory-based learning gave the best precision, but a poor recall. Overall, this paper gives a thorough treatment to more realistic applied problems with authorship attribution and helps to highlight its limitations and possible overestimation of its accuracy in previous results. This study was followed up with another by the same authors in 2011 [LD11] with two larger corpora, the first in English taken from the Ad-Hoc Authorship Attribution competition [Juo04] and the second in Dutch taken from the Dutch Authorship Benchmark corpus [Hal07] and the same corpus as above [LD08]. Their results were much the same and served to reinforce the earlier conclusions.

Sanderson and Guenter [SG06] chose to focus on the relative performance of long and short samples, with a dataset of 50 authors consisting of newspaper journalists that had written articles on more than one topic and had at least 10,000 words to their name. They were interested in the two-class problem, where a text is classified as either being written by the target author or not. To this end, they chose ten authors to constitute the background dataset (the negative, or general, class) and used the remaining 40 authors as their evaluation set. This meant that each iteration of the experiment involved the writing of eleven authors in total: ten "background" authors representing the general class and one under evaluation), which is fairly limited. In terms of training data size they started with 28,000 characters (approximately 5,000 words) per author and then reduced this incrementally to 1,750 characters while simultaneously reducing the test size in line with this to determine the effect on accuracy of having smaller samples to work with. They also tested the effect of reducing the test size but maintaining the training size. For their first classification algorithm they utilized a Markov Chain approach that calculated the sum of the probabilities of seeing each token (character or word) given the previous sequence of m tokens, the product of these various sequence probabilities is then taken for the entire document,

smoothed to prevent divide by zero errors according to interpolated Moffat smoothing [CG96] then normalized by the size of the sample and finally taken as a log likelihood ratio. Their second classification algorithm was SVM customized with a kernel function that had been developed specifically to be used as a similarity measure between word and text sequences of varying lengths (a parameter, τ, is used to determine the maximum sequence length and all sequences of length $l \leq \tau$ are evaluated), weighted so longer sequences are more significant [CGGR03]. For the experiments with the Markov chain approach, they found that chains of order two gave the best results (meaning the current and previous two states were considered when determining the probability of the next transition) and decreasing both the training/test data together gave worse performance than just reducing the test data alone. In their experiments with SVM and the custom kernel function, they tried training/test data sizes of 7000, 14000, and 28000 and also varying sample sizes ("chunks"), with samples of 500 characters being most effective across all training/test data sizes. They also obtained the best results with $\tau = 4$ over the character-based kernel, which they found to be comparable to the Markov chain with order two. In a final experiment, they investigated a technique known as *unmasking*, first proposed by Koppel and Schler [KS04]. With unmasking, features are iteratively removed from a one-class learning model to assess the speed at which the classification degrades for a particular instance. The intuition is that the greater the accuracy drop with each feature removed, the more reliant the model is on just a few features for its classification. This lack of depth indicates that the instance may be a borderline case and the classification superficial. Note that this technique only works for the one-class problem, where "degrade" means that classification switches from positive to negative, or vice versa. For example, a text may be classified as not being written by the suspected author, but after excluding a relatively few features may classify positively. Their conclusions were that unmasking was significantly less effective for shorter texts and should not be used.

Iqbal *et al.* [IHFD08], Layton *et al.* [LWD10], Koppel *et al.* [KSA11], Bhargava *et al.* [BMA13] and Afroz *et al.* [ACIS+14] have all looked at authorship attribution from a realistic, applied perspective ("in the wild"), mostly focusing on the forensic use case, citing cybercrime as the main motivation. The common themes of these papers are short text samples and large numbers of potential authors. Iqbal *et al.* [IHFD08] were interested in identifying the authors of emails, and coined the term *"write-print"* to be the authorial version of a fingerprint. Just as a fingerprint contains features that are highly distinguishing (not truly unique, but close to), so the write-print would contain the features that were unique, or highly distinguishing, to an author out of the set of authors in the corpus. The write-print features for an author would be derived from a much larger set of potential features (the "universe" $\mathbb{U}$) that were scanned for in the author's emails, with only those present being retained. Then, features (or patterns of features) that were found to be common amongst all authors were also removed, leaving only the pattern of features that are singular to that author. In this way, the algorithm is searching for the

idiosyncratic aspects of an individual's writing. This is a very different approach to other works where variations in feature values that are common to all authors are preferred. This, they argued, would make the results more admissible in court as "forensic" evidence, due to its distinctive nature. Their features were represented as simple bit strings, where each bit encoded a Boolean indicating if that feature was present or not. The resulting sparse vector could then be translated into a set of nominal values using the feature names. It should be pointed out that the features themselves were not binary by nature, however, but were standard continuous values, such as relative frequencies, whose values had been discretized into partitions of configurable width. This discretized feature space is an advantage with training data consisting of smaller samples, as it requires less precision and so is less susceptible to noise. They used as their dataset the publicly available Enron emails,[9] which is almost certainly characterized by a great deal of similarity in terms of topic and content, making this approach even more desirable over statistical methods, which would no doubt be influenced by the use of common signatures or other boilerplate footer text. In their experiments, they were able to achieve a maximum "accuracy" (precision) of 90% when distinguishing between six authors and using 20 emails per author as training data.

Layton *et al.* [LWD10] and Bhargava *et al.* [BMA13] were interested in an even more extreme example of a corpus of short texts, *tweets*, again motivated by cybercrime investigations. Tweets are the messages users of the micro blogging platform Twitter post, and were at the time restricted to no more than 140 characters. In their case, Layton *et al.* wanted to determine whether a technique previously used for source code attribution, known as SCAP [FSG+07], would work in this domain. SCAP employs character n-grams, but taken at the byte level to also extract control characters and whitespace. Character n-grams is a metric often found in authorship attribution research, but instead of an exhaustive feature vector containing all possible combinations, resulting in a sparse matrix, SCAP takes just the top L occurring n-grams found in each author's training set as its model. To perform classification, it finds the author whose set of n-grams are closest to the set extracted from the document(s) being evaluated. The distance metric employed is simply the number of elements found in the intersection between the author's top L n-grams and the document(s) being evaluated. To reduce similarity between author models, the *silhouette coefficient* [Rou87] is used to determine the degree of overlap between authors' models. A model demonstrating a high degree of overlap with its neighbouring models may need to be reduced to one or more sub-models, by calculating the silhouette coefficient between subsets of its feature space and eliminating the sub-models with a high degree of internal overlap. Bhargava *et al.* [BMA13] used an SVM classifier and 22 features, combining lexical, syntactical, tweet-specific features (such as relative frequency of hashtags) and others, including use of emojis. They also combined up to ten tweets together to increase the amount of text per sample.

[9] https://www.cs.cmu.edu/~enron/

For their experiments, Layton *et al.* [LWD10] used 50 Twitter users and 200 tweets per author, randomly selected from a corpus of some 14,000 users that had been selected by searching Twitter for certain keywords over a four-day period, then reducing the resulting 56,000 users to 14,000 through a random selection. It is not clear what the keywords were, nor why these were chosen. Bhargava *et al.* [BMA13] used 10–20 authors and 200–300 tweets per author in their experiments, sourced by using the twitter-corpus tool[10] to collect 5,000 tweets, then randomly selecting users from this list and gathering additional tweets from their individual streams.

For the 50-author, 200-tweet dataset, Layton *et al.* [LWD10] used 10-fold cross validation for testing, and their best results were a 72.9% precision. They also experimented with removing "mention" (*@handle*) and "hashtag" (*#hashtag*) content from the tweets to see if this information improved or reduced the performance of classification. They found that including mentions improved performance but hashtags made little difference and attributed this to users interacting frequently with the same other users. The fatal flaw in these experiments and their conclusions, especially with regards to the mentions, is that the training and test data is drawn from the same corpus of accounts whose class labels are based on Twitter handle, rather than the account owner. Bhargava *et al.* [BMA13] reported results whose success depended on the number of users and tweets. They found for 10 users, the best results were with 200 samples, giving an accuracy of 81.42%, while for 20 users, 300 samples gave the best results with an accuracy of 64.54%.

Clearly, in practice law enforcement would face the scenario of attempting to link *different* accounts belonging to the same person (or bot). There is no way to tweet anonymously without using a separate account, so training and testing on the same data, even with cross validation, is meaningless. This also highlights the issue with Layton *et al.* [LWD10] concluding that including mentions increases performance because of frequent interactions with the same other users. A person with two different accounts, one of which is being used covertly, is highly unlikely to exhibit the same social graph in their covert account as their overt account. The only way to meaningfully conduct this study would be to take sets of multiple accounts that are known to belong to single individuals, if this is possible, and then to attempt authorship attribution where the training set contains their overt accounts and the test set their convert accounts.

Koppel *et al.* [KSA11] looked at blog posts from 10,000 authors on blogger.com during a single month. They once again used character *n*-grams (4-grams), first taking a "naïve" approach using a feature vector with all possible 4-grams, giving a dimensionality of just over 250,000 and calculating the cosine similarity [SB88] between the document being evaluated and each author's model to perform classification. Even this simple method achieved a precision accuracy of 46% for the 10,000 class problem, which is somewhat impressive, although the authors are quick to point out that this is insufficient for most applications.

[10] https://github.com/bwbaugh/twitter-corpus

Afroz *et al.* [ACIS+14] investigated the case of finding duplicate accounts in online forums, particularly criminal forums specializing in stolen credit card data. This study differs from previous studies in that it uses real data from multiple accounts, as opposed to simulated multiple account data created by artificially splitting up a single account's samples across training and test sets as with cross-validation, which we highlighted our concerns with above. They also describe additional practical considerations with users' extensive use of slang and "leetspeak" posing difficulties when using traditional language analysis, such as part-of-speech taggers. They approached this problem as an unsupervised learning task, obtaining the ground truth from data dumps that were leaked from four forums: AntiChat, BlackhatWorld, Carders, and L33tCrew. User accounts were linked between the forums by cross-referencing their registered email addresses. Within the Carders forum they were also able to link accounts by analyzing warning messages sent to users that logged in to a different account than that indicated by a persistent cookie on their machine (all these forums ban users with multiple accounts).

The feature set they used made use of part-of-speech taggers and function words in English, Russian and German. They also used frequencies of character *n*-grams, punctuation, special characters and leetspeak "words". The JStylo system [MAC+12] was used for feature extraction. To evaluate their corpus and feature set (single account attribution with ground truth within a single forum), they used an SVM classifier and ten-fold cross-validation, achieving up to 72% accuracy with 82 users on BlackhatWorld, and 44.4% accuracy over 1459 users on AntiChat.

For the multiple account detection, they start with a hold-one-out methodology, training with all other users than the one being evaluated, then test with the held out user's data to see which other user label the classifier assigns to them. This is repeated for every user and pairs of users whose combined pairwise probabilities were above some threshold are considered to belong to the same person. Because their forum corpus does not provide complete ground truth, they first evaluated using an independent dataset of blog authors that maintained multiple blogs, taken from Narayanan *et al.* [NPG+12], filtered by single-author blogs with a minimum of 4,500 words and finally restricted to 100 authors that wrote 200 blogs. On this dataset, they were able to achieve a precision of 90% and recall of 92%. Applying this method to verified multiple identities across different forums (Carders and L33tCrew), using the cross-referenced dataset outlined above, they were able to achieve 85% precision and 82% recall with 179 users. Next, they attempted to identify multiple accounts within the Carders forum only, using the private messages of 221 users that had written at least 4,500 words as their corpus. Verification was a manual step, involving comparing ICQ numbers, signatures, products traded, payment details, and contents of messages, as well as information derived from the data dumps, such as user creation date. This manual verification was carried out for 21 pairs that were assigned the highest probability by the classifier as being duplicate accounts. The categories assigned as a result of the analysis were: **true**, **probably true**, **unclear**, **probably false**, and **false**. Of the 21, ten were true, three were

probably true, two were unclear, three were probably false and six were false.

4.2 Plagiarism Detection

Detecting cases of plagiarism is of considerable importance in teaching, and due to the links between academic research and university teaching, it is no surprise that papers on plagiarism detection are common. Clough [Clo03] provides a good overview of plagiarism detection techniques, looking at both the software and natural language cases. Authorship attribution is discussed, but only insofar as it not being considered plagiarism detection. Later in the same paper, however, authorship attribution is given as an example of the approach taken by some authors towards plagiarism detection, but no papers are referenced in relation to this.

Plagiarism detection and authorship attribution are closely related, but distinct activities. With authorship attribution the aim is to take an artifact of unknown authorship and identify if it appears to have been written by an author from among a known set of authors. With plagiarism detection, the artifact being analyzed already has a named author and the aim is to decide if they actually wrote it or its content or ideas were copied from some other artifact that the purported author did not write. There are two main divisions in plagiarism detection: *external* plagiarism detection [Sta09] and *intrinsic* plagiarism detection [ZES06]. The external method attempts to identify a source document that was plagiarized, whereas intrinsic methods look for indicators of plagiarism without identifying the source. External plagiarism detection is a somewhat simpler problem because it typically involves searching for similar, or identical, content among some database of existing content and determining if the degree of similarity is above some threshold. The content may be broken down into chunks (by a sliding window [Sta09], paragraph, sentence [WC98], word [WC98, LMD01] or character n-grams [GHM05, Sta09]), or preprocessed in other ways but essentially most automated external plagiarism detection systems use some form of similarity measure between the content of different documents. This can be a weakness, as an astute individual simply has to change their plagiarized content sufficiently to fall below the threshold in order to avoid detection. One could argue that the effort required to modify the content sufficiently could act as a deterrent, as could simply making students aware of the existence of such automated tools; however, following this line of reasoning one could simply pretend to have such a capability. Intrinsic plagiarism detection has the advantage that no database or other source of content is required, as it focuses only on the document being analyzed. This drastically reduces performance and storage overheads, and does not require the corpus to be digitized, as well as resulting in a more flexible system that can also detect unauthorized collaborative efforts [Clo03]. The downside is the task is somewhat more complex and less tangible; it is no longer as clear what to search for and in some respects the task is the polar

opposite of external plagiarism detection as this time it is *dissimilarities* that are sought, rather than similarities.

In contrast to plagiarism detection, authorship attribution has to consider aspects that transcend the content of individual artifacts and are common or peculiar to the author themselves and therefore must be present in multiple unrelated documents. Two artifacts by the same author may vary considerably in content, and it is the task of authorship attribution to discover the common features inherent to authors that distinguish them from one another. Despite these differences, there are enough similarities in aims and methods to merit a discussion of plagiarism detection in the context of authorship attribution. Indeed, there are papers on intrinsic plagiarism detection that bridge the gap between it and authorship attribution by examining aspects of style rather than content, so simplistic rewriting or synonym replacement strategies that aim to defeat content matching would still be detected. These papers will be the focus of this section.

4.2.1 Intrinsic Plagiarism Detection

Eissen and Stein [ZES06] were the first to investigate intrinsic plagiarism detection, which they defined as looking for sections of a written document that may have been plagiarized, without necessarily identifying the source document. Their approach was to look for passages in the document that are written in a different style. This represents a stronger form of plagiarism detection than external plagiarism detection strategies as it would still work when the source document is unknown, as well as when it is known and the content has been copied verbatim. Such instances are fairly trivial for a human to detect, as the difference between a professionally edited, proof-read, published work and an assignment that has been through none of these quality assurance steps is usually glaringly obvious; however, automating a process that is trivial for a human, particularly one involving natural language, is far from simple.

Their paper outlines that the method they used was to calculate the "average word frequency class" in different passages of the text and look for divergences from the average. This average word frequency class for a corpus of text C and a word $w \in C$, is defined as $\log_2((\text{frequency of most common word})/(\text{frequency of } w))$ rounded down to the nearest integer (floor function). Therefore the more common a word is, the lower its frequency class. They claim that the average word frequency class *"tells us something about style complexity and the size of an author's vocabulary—both of which are highly individual characteristics"* [ZES06]. Furthermore, this method produces just a single metric as it is based on the average frequency class and does not code the specific words themselves. Such features as these would be insufficient for authorship analysis as the metric would probably vary too much between documents by the same author and may be dependent on the topic of the writing as well as style, but these are of no concern when

detecting differences within a single document.

A number of other papers have looked at intrinsic plagiarism detection. Stamatatos [Sta09] applied a distance metric based on the profile of character n-grams (3-grams were used) in passages of the text derived from a "sliding window" over the document and the profile of the entire document. Using a sliding window creates a continuous function and ensures there are no problems associated with boundary placing, as is the case when using discrete chunks. This was an adapted technique from an authorship attribution paper by the same author [Sta06]. The passages for which the distance metric peaked are the passages most likely to represent someone else's writing. Oberreuter *et al.* [OLRV11] also took the approach of a sliding window, this time using word frequency as the metric. Their idea was to count the occurrences of each word in the entire document and in the current window, then look for cases where there is a lower than average difference between the two. This identifies passages of text containing words that do not appear or are uncommon in any other passages, which they postulate is indicative of a change in style and hence author, as two authors will have different vocabularies they draw on when writing.

4.2.2 Authorship Verification

Stein et al. [SLP11] highlight the link between intrinsic plagiarism detection and authorship "verification", which is a one-class form of authorship attribution where a specific author is believed to have authored a document and the classifier must decide whether the document was indeed authored by that person or not. They present a table of stylometric techniques shown to have "discriminatory power" for these problems and use a number of lexical and syntactic features for their study, which applies Bayes' theorem and the *unmasking* approach proposed by Koppel and Schler [KS04], and described in Section 4.1.3.

4.3 Authorship Attribution of Software[11]

Donald Knuth and Edsger Dijkstra were well known for considering computer programming an aesthetic discipline, with Knuth's book series titled *The Art of Computer Programming* [Knu73] and Dijkstra's *A Short Introduction to the Art of Programming* [Dij71]. As a human endeavour, there will always be considerable variation in how the task of writing a program to solve a problem is undertaken, and programming languages, indeed even the very architecture of computers, enables and even encourages this variation. With this variation comes choice, then, both

[11] Some of the text in this section has been partially adapted from a paper written for a class project (CS858), by Iyer and McKnight.

conscious and unconscious, and preference in how to write code, influenced by education, experience and personality. As creatures of habit, we tend to follow familiar paths rather than striking off into the unknown, even if the familiar path is not the best or shortest route to our destination; changing these habits is a relatively slow process. Considering this, it is likely, inevitable even, that programmers will be identifiable to some degree by the way their code is written, presented, and the methods they employ to solve the problem at hand.

Hayes and Offutt [HO10] examined the *consistent programmer* hypothesis, which states that a programmer's coding style (or *voice*—a term drawn from natural language writing style) is consistent over time. The truth of this statement is important for any research examining author attribution. If it is false, and programmers are inconsistent and continually changing their style, then identifying them from their source code would be impossible. They based their initial assertion on anecdotal evidence from colleagues and acquaintances that they were able to recognize first notes, then typed notes and finally code written by a colleague that they had worked with for some years. The goals of their study were to show that, due to consistency of style, individual programmers would be susceptible to introducing the same bugs into their code so testing of a component could be more directed depending on the contributor(s). They focused primarily on frequency analysis of operators, operands, syntactical features and, uniquely, warnings raised by the lint static analysis tool. They found that the number of lint warnings was a distinguishing factor, but the *testability* of code was not correlated to the individual. Testability was defined through three measures that were designed to represent the probability that defects would be detected during testing. For our purposes, the lack of an observed correlation between programmer and the testability of their code is not significant, however the correlation between syntactic feature frequency and author is.

4.3.1 Source Code Attribution

As mentioned in Chapter 1, the earliest work attempting to analyze the style of programmers did so with the intention of assessing, or grading, a program based on whether it followed accepted style guidelines and best practice [Mee83, BM85]. It was assumed the likely users of such tools would be academics grading students' assignments, with the automated analysis serving as an objective marking aid for that portion of the marks, to relieve the instructor (or teaching assistant) from this tedious task so they could focus on the behaviour and functionality of the program instead. Oman and Cook [OC89] were the first to link program style with programmer identity, with the intended application being plagiarism detection and copyright infringement. They disregarded metrics such as complexity analysis, based on Berghel and Sallach's work [BS84] that had concluded these were not useful in detecting cases of plagiarism. Instead, they chose to use style "markers" in Pascal, aspects that were most under the control of the author and less

rigorously defined in the language, allowing for greater freedom of expression. Such markers were the layout of the code and use of indentation and whitespace, comments (frequency, type, length) and naming conventions for variables, procedures/functions, constants. They also took the novel approach of first conducting a user study in which they evaluated the ability of humans to distinguish between a small set of programs with the same functionality, written by three different authors, taken from a corpus of computer science programming text books. Each text book contained implementations of several common algorithms, such as bubble sort, quick sort and tree traversal. They showed these samples to eleven experienced programmers and tasked them with grouping the implementations by author. In all but one case, the example code was correctly grouped by the humans, lending credence to their hypothesis that style is consistent and individualistic. Their automated style analysis was carried out on the same set of example algorithms from the text books and consisted of a vector of Boolean values, which they then used to calculate the inconsistency of each book with respect to its feature vector and performed cluster analysis to try and determine which features were connected to author and which to behaviour. After this, they applied the style checker to industrial code, carrying out a similar cluster analysis, finding a consistent level of similarity between code written by the same organization.

This work predated that of Spafford [Spa88], discussed in Section 3.2, which was primarily an analysis of the behaviour of the Morris worm, with the authorship analysis being a result of manual review, rather than the output from an automated tool. Spafford and Weeber followed this earlier report up, however, with a position paper that discussed potential automated authorship attribution methods [SW93], coining the term "software forensics" to describe this approach. Unlike Oman and Cook [OC89], Spafford and Weeber also considered the analysis of binary executables as well as source code as part of software forensics, enabling the technique to be applied in situations where the source code was not available, such as for malware samples seen "in the wild". This was the first occurrence in the literature of discussing authorship attribution with respect to executable code, which, given the author's earlier experiences in dealing with the aftermath of the Morris Worm, is not surprising. The authors go on to describe what they term as *"code clichés"*, idiosyncratic snippets that are used habitually by a programmer and can be used to build a profile of and identify them. They compared this to handwriting analysis ("graphology") where features of handwriting are sought that vary significantly over the population as a whole, but tend to be consistent or vary little for a particular writer. In addition to these idiosyncratic "clichés", statistical features were also mentioned, whose identifying characteristics would become apparent only after examining a large corpus of the author's source code. Such metrics as code complexity, program size, function size and comment frequency were given as examples. With executable code, the researchers pointed out that compilers often destroy many potentially identifying features, particularly whitespace and comments, which are never retained, and identifier names, if debug symbols are not included with the executable file. In addition, the

paper mentions some alternative methods of achieving the same end result are mapped to the same structure during compiler optimization, such as different looping constructs, the choice of which may say much about the programmer's style. The degree to which this mapping occurs and the aggression with which the compiler carries it out will depend in large part on the level of optimization it is invoked with. They did offer some suggestions of features that would survive the compilation process, however. These were:

- **Choice of data structure**: It should be possible to derive from the executable whether the programmer chose to use a linked list or hash table, for example.
- **Compiler and system information**: The executable file produced by the compiler should contain indications of which compiler, toolchain and operating system was used by the programmer, and the original source language it was written in.
- **Choice of system and library calls**: Programmers may prefer certain libraries or particular system functions.
- **Bugs and vulnerabilities**

In terms of source code analysis, the suggested features that could be extracted were:

- **Choice of language**
- **Formatting**: Whitespace, indentation, placing of braces, etc.
- **Special features**: Use of compiler-specific directives.
- **Comment styles**: Different ways of presenting comments, frequency, verbosity, etc.
- **Variable names**: A variety of preferences and best practices for naming schemes exist.
- **Spelling and grammar**: Particularly mistakes.
- **Language features**: Most languages provide a plethora of ways to achieve the same end result.
- **Scoping**: Preference for global vs. local for instance.
- **Execution path**: Particularly unreachable code, such as debugging statements remaining after the development phase.
- **Bugs**

- **Metrics**: Cyclomatic complexity and other metrics that can be seen as equivalent to such measures as vocabulary richness or readability scores in natural language.
- **Clichés**: Referring to idioms; i.e., small, but reusable patterns for solving simple problems of the sort one encounters commonly (so a more elementary level than a design pattern).

The problems they foresaw with attribution were the amount of code available to analyze, whether the programmer had copied blocks of their code from elsewhere, possibly written by someone else, and collaborative coding, where different parts of their program had been written by different authors, especially if contained in the same file. Nowadays it is very common for programmers to copy large amounts of code from online forums, such as StackOverflow,[12] with a minimum of editing. Additionally, much code is written, revised and edited at multiple times by multiple authors. Over time there is naturally going to be a decline in coherent style exhibited by a program that is the work of a team of contributors. Modern version control systems, such as Git, in conjunction with the web enable easy and seamless team coding, encouraging greater collaboration.

Sallis, Aakjaer and MacDonell [SAM96] also wrote a position paper on this topic, touching on many of the same points as Spafford and Weeber, but with the following additional suggestions for features: *control flow graph*, *data dependency*, *nesting depth*, and complexity measures. Their conclusions were that a combination of different features would be necessary for robust identification and the challenge would be to obtain sufficient source material to perform an adequate analysis. A later paper by Gray, MacDonell and Sallis [GMS97] presents these same ideas, along with stating their intention to write a system for performing authorship attribution of software and summarizing the analyses carried out by Spafford on the Morris Worm [Spa88] and Longstaff and Schultz [LS93] on the WANK and OILZ worms, which like Spafford's analysis, also included manual authorship profiling.

Krsul and Spafford were the first to attempt attribution experimentally [KS97], taking 88 programs written by a total of 29 participants. They dismissed using spelling errors as a feature as there were too many non-dictionary words that were flagged as spelling errors by the spellchecker they used and it was too difficult to differentiate genuine spelling mistakes from the intentional use of non-existent words. The features they used were largely statistical in nature and spanned categories including whitespace, indentation, program and function structure, naming conventions, comments, scope, function signature, branching and code complexity. For classification methods, they tried discriminant analysis, neural network and the likelihood (Gaussian) classifier provided by software that had been developed at MIT's Lincoln Laboratory. For discriminant analysis, they were able to achieve a precision of 73%, the results for the neural

[12] https://www.stackoverflow.com

network (a multi-layer perceptron) were given in terms of error rate, and the best result had an error rate of 2% when using 4-fold cross validation, after linear discriminant analysis normalization of the feature vector and reducing the dimensionality of the feature vectors to 15 (using a forward and backward search to find the optimal features to include). The Gaussian classifier was able to classify with 100% precision using just six features. They concluded that aspects of programming style are consistent, however they acknowledged their sample size was limited and more experiments would be required with a larger number of classes. They were also of the opinion that style changes over time therefore the training data would need to be chosen to correlate temporally with the data being evaluated.

Kilgour *et al.* [KGSM98] argued that features based on qualitative values that were assigned through manual examination by a human judge could complement more quantitative features calculated through automated means. Their reasoning for this line of argument was that some aspects of style are difficult to quantify and extract through mechanical means, but require some degree of intuition by a subjective judge. If the same judge examines all the artifacts used in the training and evaluation phases these judgments should be consistent. The obvious issue with this sort of approach is scalability.

MacDonell, Gray and Sallis [MGS99] investigate using case-based reasoning (CBR), neural networks and multiple discriminant analysis along with a framework they had written called IDENTIFIED, for the extraction of features and classification with the ability to configure which learning algorithm is used. Their study was rather small, involving just seven authors, but gave promising results for CBR. Case-based reasoning is a form of memory-based learning, where some form of distance metric is used to find past examples from the training set that are most similar to the example to be classified, and extrapolating the answer based on these past examples. Formulated in this way, it can be seen as very similar to k-NN, however with CBR there is an attempt to perform extrapolation based on the current example's position in the feature space relative to its nearest neighbours.

While most studies investigating this problem had focused on the C or C++ languages, Ding and Samadzadeh [DS04] decided to determine if there were any notable differences between languages, by looking at Java instead. As Java is compiled to intermediate bytecode, rather than machine code, it is far easier to decompile Java and extract meaningful source code. In general, they did not find any major differences in the approach or performance of their classifier with Java, other than some features being necessarily different due to using different keywords and syntax. This result mirrors the findings of natural language authorship analysis where the technique and approach is broadly applicable not only to English but also other languages, and there is no reason to believe one language would be far easier or more difficult to analyze stylometrically or perform attribution with than another. In theory, however, if we view style as a manifestation of choice between two or more equally valid options, then it follows that the more

options there are, i.e. the more choices one has in order to produce some desired effect, the more style should be apparent. In natural language, there are far more ways to express oneself and deliver the meaning of a sentence than in computer programming languages, which are more restricted. It follows, therefore, that authorship attribution should be easier for natural language than computer code. The differences between computer languages is far closer and in general they all follow strict syntactical rules, but there is some definite variance. JavaScript, for example has a more relaxed syntax than Python, so it should be the case that it is easier to identify JavaScript programmers from samples of their code than Python programmers. C++ is known for supporting many programming paradigms, either natively or through library support, whereas C is a procedural language only.

Kothari, Shevertalov and Stehle [KSSM07] took a novel approach to feature selection by utilizing entropy both at the population level and the individual programmer level. Most feature selection criteria only take into account the population level discriminatory value of a feature, but by also considering its information entropy at the individual level, they were able to tailor the feature set programmer-by-programmer. This worked by selecting features that had low entropy individually, meaning it was hard to distinguish between files that developer had written based on the metric, indicating the feature's value was predictable among their files, and high entropy collectively, meaning the values the feature assumed had high variance and low predictability (without the class label being taken into account). Based on this evaluation, each programmer is assigned a profile containing the best k out of n features. When performing classification, all potential n feature values are extracted and matched to the programmers' profiles, with the best match being the assigned class. This approach is very similar to that taken by Iqbal *et al.* [IHFD08] for natural language authorship attribution with their "write-prints" analysis of emails, although this paper was published first. Kothari *et al.* also proposed using character *n*-grams, a long-held strongly identifying feature used in authorship attribution of natural language. They used $n = 4$, which interestingly also tends to produce the best results in natural language applications. In their experiments, despite using a relatively small set of authors, they took a novel approach by only considering developers that had written code across multiple projects and performing holdout evaluation where an entire project was held out. In previous studies, typically code was gathered without considering projects, repositories or software products, and for evaluation purposes either *k*-fold cross validation was used or the data was split without taking different projects into account. As mentioned in Section 4.1.3, Frantzeskou *et al.* [FMSG08] also utilized *n*-grams in their SCAP tool, with the exception being theirs were byte-level rather than strictly character-level, but otherwise it is the same technique. Frantzeskou *et al.* followed up their study with an investigation into what high-level features their SCAP tool was picking out in two languages, Java and Common Lisp, and which of these high-level features were most important in assigning authorship. In order to determine the most important features, they applied

an *unmasking* approach similar to Koppel and Schler [KS04], described in Section 4.2.2, in which they iteratively removed the high-level features to compare the effect on classification performance. They discovered that comments, layout and naming were the most relevant, which is perhaps not surprising given that these aspects are most under the programmer's control.

Caliskan-Islam *et al.* [CIHL+15] investigated a vast battery of features in their 2015 paper, referring to the practice as *"code stylometry"* rather than software forensics. In what could be regarded as a watershed moment in source code authorship attribution, they were able to achieve very high levels of accuracy (over 90%) even when considering a huge dataset containing code from 1,600 authors and limited file size. Their unique approach to the problem involved parsing an Abstract Syntax Tree (AST) representation of the code, in addition to the typical flat file analysis. Their initial feature set contained some 120,000 dimensions, the majority of which were the result of dynamic features that included combinations of elements. The features considered fell into lexical, layout and syntactic categories, with the lexical and syntactic categories accounting for the majority. The size of the lexical category was in large part due to a bag-of-words feature which calculated the tf-idf for every single word seen in the corpus, including in comments, names of variables, functions, structs, etc. and string literals; this feature alone accounted for 55,000 dimensions. The remainder of the lexical features consisted of standard statistical measures at the comment, line, function and file level. Layout features represented the smallest category, with standard measures of such aspects as indentation and whitespace. Finally, the syntactic category consisted of the AST-based features, representing over 58,000 dimensions. 45,000 dimensions in this category were made up of the tf-idf calculated for all possible AST node bigrams, which the authors reported were the most identifying features. All their experiments were run with 10-fold cross validation, using the *random forest* classifier [Bre01]. The random forest algorithm is an ensemble classifier, meaning it is made up of multiple simpler classifiers, who each provide their prediction and "vote" on the class. In this case, the ensemble is made up of random decision trees. The class that received the most votes is taken to be the overall prediction. Random Forests are discussed in more detail in Section 5.2.4. Due to the extreme dimensionality of their feature vectors, they performed feature reduction using information gain, to prevent the decision trees from becoming too large and reducing the effect of overfitting. Information gain is a univariate measure of a variable's influence by taking the information entropy of the distribution of classes in the dataset, then deducting the entropy of the classes given the values of the feature, providing the extent with which the feature reduces entropy, or how much information is *gained* about the class by considering the feature. Using information gain, they were able to reduce the dimensions to a "few hundred" (the exact amount was not given) without a loss in classification accuracy, but with a far shorter response time. To test the generality of the information gain feature selection, they tried it on two distinct subsets of their training data and reported that the features selected and their ordering were more or less consistent. The

data were obtained from the Google Code Jam[13] (GCJ) programming competition stats site,[14] between 2008 and 2014. Their main experiment was conducted with 250 authors from the 2014 competition that had submitted solutions in C/C++, that had all submitted solutions to the same nine problems. This experiment, using the smaller feature set, had a precision of 95.08%. The experiment was repeated with different combinations of years and problems, producing comparable results. In addition, increasing numbers of authors were considered by including previous years' competitors and solutions, up to 1,600 in total. This largest class category was still able to predict the true class with 92.83% accuracy. They also discussed the open-world situation where the true author may not be known, the two-class problem and the binary classification problem (author verification), which often arises in cases of plagiarism detection. For possible defences, they discussed using obfuscation and ran experiments with two off-the-shelf obfuscators against a 20-programmer subset of their main dataset. They found one obfuscator, Stunnix,[15] which performs basic obfuscation, barely reduced the accuracy at all, while the other, Tigress,[16] which performs a more fundamental obfuscation at the cost of readability and performance, reduced the accuracy significantly, from 95.91% to 67.22%. Overall, this paper offers a comprehensive treatment of source code authorship attribution, virtually exhausting all possible features rather than focusing on just one category and experimenting with a huge corpus of code. We conclude from their results that n-gram based features once again produce strong classification accuracies (in this case the items were AST nodes). It is possible that higher-order n-grams may produce even stronger results, however in this case as the number of distinct node types is far greater than the number of alphabetic characters, the dimensionality would grow at an unfeasible rate.

One potential pitfall with this study is the bias introduced by examining only code submitted for programming competitions, which bears little similarity with code that is written for real software, the likely target for actual authorship attribution. While it may seem superficially that this task is harder, as much of the behaviour-specific content is the same, the reality of competitive coding involves a significant proportion of boilerplate code that is typically copied from one submission to another. For example, each online competition will have a standard for how the input data is presented and how the output should be formatted and returned. This leads competitors to reuse the same lines of code to both read and process input data and construct and return output data, giving them more time to focus on the actual problem. For programs averaging just 70 lines, this reproduced code could represent a significant fraction of those lines. Still, the results are a great deal more impressive than any previous study on source code stylometry, due in large part to the use of the AST-based features. It would be informative to experiment with such

[13] https://code.google.com/codejam/
[14] https://www.go-hero.net/jam/
[15] http://stunnix.com/prod/cxxo/
[16] http://tigress.cs.arizona.edu

features on real code, sourced from publicly available repositories, to evaluate the performance "in the wild".

In a recent paper, Dauber *et al.* [DCHG17] looked at attribution of incomplete code *snippets* taken from GitHub. The samples were derived by using the `git blame` command on the files contained within the repositories in their corpus. They gathered their corpus by taking 14 *seed* authors and enumerating the collaborators' data associated with those user accounts, which is returned by the GitHub API, before taking the C++ repositories those one-hop collaborators had contributed to. This initial corpus consisted of 1,649 repositories and 1,178 programmers, but was eventually reduced to 106 programmers after the following filters had been applied: repositories of insufficient size, user accounts representing more than one individual and at least 150 samples of C++ code per author (with a minimum of one line of actual code per sample, discounting whitespace and comments). The samples were grouped by author and line number (rather than by author and commit), so consecutive lines blamed to the same author were grouped into one sample, regardless of the commits. In their experiments, they worked with variously sized subsets, including a smaller set of 15 programmers with a greater than average number of samples (385) and a larger set of 96 programmers with fewer samples (90) and a set of 90 programmers with variable numbers of samples. The feature set they applied was that used by Caliskan-Islam *et al.* [CIHL+15], described above. They pointed out that feature reduction using information gain was not possible, however. This was because the feature vectors on the whole were far more sparse when extracted from these samples than from entire files, rendering the information gain calculation unreliable. Therefore, their feature set had a very high dimensionality (62,561 for the 15-programmer set and 451,368 for the 106-programmer set). When combined with a forest consisting of 500 trees, each experiment with the 106-programmer set took 20 hours on a 32-core 240 GB RAM machine, which is considerably slower than Caliskan-Islam *et al.* experienced with their reduced feature set and 300-tree forest, where an experiment involving 1,600 programmers and 32 GB RAM took less than one hour. The authors investigated two main use cases, and several additional use cases. Their main use cases were single sample attribution and multiple sample attribution. Single sample attribution was, as its name implies, classifying a single sample consisting of an average of 4.9 lines of code per sample (for training, there were of course still multiple samples). With multiple sample attribution, they took an aggregation approach to the problem, taking all the samples known to be authored by the same (as yet, unlabelled) user and aggregating the confidences returned by the classifier for each sample to reach an overall classification for the entire sample set. This approach was also proposed by Overdorf and Greenstadt [OG16] for application to cross-domain authorship attribution of short texts. They compared the results of using this technique with another that combines the resulting feature vectors, and noted that the results were better overall when applying this form of *stacked* ensemble, than combining the feature vectors.

For the single sample attribution case, with 106 programmers they were able to achieve an accuracy of 73% using a 500-tree forest. For the multiple sample case, they began with a 15-programmer set and experimented with combinations of using aggregated, averaged and individual samples during training and evaluation. The results of these experiments suggested that limited merging of training samples by averaging their feature vectors and aggregating the evaluation of individual samples gave the best results, therefore this approach was taken with later experiments using larger numbers of programmers. With the 106-programmer set, when aggregating results across 15 samples and using 500-tree forests, they were able to achieve an accuracy of 99%. One final experiment they ran which has significance for our research is the effect corruption in the dataset has on the performance of the classifier. This was carried out by taking the GCJ dataset as used by Caliskan-Islam *et al.* [CIHL+15, CYD+15], which is known to be of high purity with respect to its ground truth, and swapping individual files' labels in the training set. The results of this experiment showed that increasing proportions of corrupt data caused decreasing accuracies, but without a strong drop-off. This led the researchers to conclude:

"It would take serious systemic ground truth problems to cause extreme classification problems."

4.3.2 Executable Code Attribution

Rosenblum *et al.* [RZM11] were able to demonstrate that aspects of programmer style are preserved through the compilation process. In fact, they were able to identify authors with a high degree of accuracy (81% from 10 classes, falling to 51% from 191). The technique they employed made use of ParseAPI [Pro11], which is able to produce instructions and control flow graph (CFG) representations of an executable. They then extracted a large feature set from the code at varying levels of abstraction, which they hypothesized would contain the stylistic remnants of the source code. This feature set consisted of *n-grams, idioms, graphlets, supergraphlets* and *call graphlets*. Their *n*-grams are based on 3- and 4-gram bytes found in the binary file. *Idioms* are essentially *n*-grams of assembly instructions, with an n of 1, 2 or 3. *Graphlets* are trigrams of CFG nodes, while *supergraphlets* are trigrams of nodes in a collapsed CFG, where the collapsing algorithm merges each node with a random neighbour. *Call graphlets* are likewise trigrams of nodes, taken from a reduced representation of the CFG, where only nodes consisting of a call instruction are retained. The edges in this reduced CFG represent an entire path of vertices and edges in the original CFG. The combination of these feature types gave them a total of over 609,000 features. For their corpora, they chose to use Google Code Jam competition entrants[17] from 2009 and 2010 who used C/C++ and submitted at least eight solutions and student

[17]This appears to have been the first time GCJ data were used in authorship attribution research.

assignments from an operating systems course. The GCJ corpus consisted of 284 authors from 2009 and 2010 (93 and 191, respectively) with a total of 2,581 files while the student corpus consisted of 32 authors and 203 files. Using a measure of mutual information to rank their features, they then tested various combinations of the highly ranked features with ten-fold cross-validation to select the top 1,900 features for the GCJ corpus and the top 1,700 for the student corpus. For classification, they used SVM, again with ten-fold cross-validation achieving an average accuracy of 81% over 20 separate experiments with 10 randomly chosen programmers. This accuracy fell to 51% from the full set of 191 programmers from the GCJ 2010 data. They were also able to place the correct author in the top five predictions 95% of the time for 20 authors, and 81% for 100 authors.[18] Their work was able to answer in the positive the question of whether stylistic features are preserved and if they exist in large enough quantities to identify authors. The study was unable to specify what exactly those features represented, however, and how they relate to high-level source code; it answered the if and how, but not the why.

Alrabaee *et al.* [ASP+14] improved on the work of Rosenblum *et al.*, taking an "onion" model approach to the problem, with different layers of filters and feature set extractors. Their filtering layer, referred to as the *stuttering layer* first removes those parts of the binaries that may be unique to a given compiled program but are not indicative of programmer style, such as unique IDs or names that a compiler may generate internally during the compilation process. Then it identifies the parts that represent library code using a feature of *IDA Pro*[19], a powerful commercial debugging platform, hex editor and decompiler. Once the binary has been filtered in this way, they perform pattern matching in their *code analysis layer*, which attempts to match low-level instruction patterns to high-level programming constructs using a set of templates and a form of fuzzy matching. This layer shares many of the challenges a decompiler faces when reversing a binary program and the relevance of taking this approach as opposed to using a purposeful decompiler is not clear. The final layer in their onion model performs *register flow analysis*. This novel approach to binary program analysis creates a *register flow graph* to chart the use of registers in the program as a way of fingerprinting the data flow. Using these techniques they were able to reduce the number of false positives seen over two, four and six authors from approximately 13–65% to 3–12% in comparison with Rosenblum *et al.* [RZM11].

Caliskan-Islam *et al.* [CYD+15] took a hybrid approach to the problem of identifying the authors of program binaries, combining the feature set used by Rosenblum *et al.* [RZM11] with a subset of the features used in their previous paper on source code stylometry [CIHL+15]. More specifically, they made use of assembly code *n*-grams, and control flow graph block unigrams and bigrams, as well as word unigrams, library and internal function names and AST-based features, such as node type unigrams. The assembly code and control flow graph features were

[18] Note this "top-*n*" test is often used in identification and attribution research, where false positives are common.
[19] https://www.hex-rays.com/products/ida/

extracted from disassembled representations of the executable files produced by the Netwide[20] and Radare2[21] disassemblers. The word unigram and function name features were extracted from decompiled versions of the binaries, using IDA Pro and the Hex-Rays Decompiler plugin.[22] The AST-based features were extracted after the decompiled source code had been parsed by Joern[23] [YGAR14]. For their main dataset, they once again used the Google Code Jam entries, this time selecting 100 programmers that each submitted C++ solutions to the same nine problems, providing a very controlled, consistent dataset, in order to reduce the influence that functionality had on the outcome and try to isolate style. Their unrefined feature set had very large dimensionality with 750,000 features and was also mostly sparse. As they had chosen to use random forest classifiers, the sparseness of the feature vectors were a concern, as only a random subset of the features are compared at each node, and with such a sparse set of feature values it was probable that a large proportion of the selected features would have 0 values. To address this concern, they used information gain for feature selection, much as with their earlier paper on source code stylometry [CIHL+15], to reduce the dimensionality to less than 2,000. In addition to speed and memory gains with this feature reduction, they also saw a significant increase in successful classifications, from 30% with the unrefined feature set to 90% after reduction. As with their earlier work [CIHL+15], they trained a 500-tree forest with $\log(M) + 1$ attributes selected at each node (where M refers to the total number of attributes), and evaluated with nine-fold cross-validation. This was instead of the more typical $\sqrt{M} + 1$ attributes selected at each node. The cross-validation was stratified in such a way that the training set contained the *same* eight problems for each author, and each author was present, effectively becoming a hold-one-out test. The accuracy of this cross-validation was 89.8% for the 100-programmer set with the reduced feature set. When this same feature set was used with a different, non-overlapping, set of 100 programmers, the accuracy was 92.8%, which demonstrates the feature set is applicable beyond the original sample and may be representative of style in general. When increasing the number of authors from 100 to 600, a modest decrease in accuracy was observed, to 78.1%. Experiments were also performed with compiler optimizations at levels 1, 2 and 3 and 100 programmers, resulting in a decrease in accuracy to 85.7% with level 3 optimization, while fully stripping symbol information reduced accuracy "by 23%" (so presumably from 89.8% to approximately 66.8%). Using Obfuscator-LLVM [JRWM15] reduced the accuracy by only 3.6%. In addition to their experiments with the GCJ dataset, the authors also wanted to approach the problem of attribution "in the wild", using repositories found on GitHub. To this end, they selected C/C++ repositories that only a single user had committed to, with at least five stars, a minimum of 200 lines of code and had not been forked from another repository. Furthermore, they filtered repositories

[20] http://www.nasm.us/doc/nasmdoca.html
[21] http://www.radare.org/
[22] https://www.hex-rays.com/index.shtml
[23] http://mlsec.org/joern/

with some common names (“linux”, “kernel”, “osx”, “gcc”, “llvm” or “next”) or the comment “signed-off” (as this was taken as an indicator of someone else’s work). As a final step, they manually checked each remaining repository to ensure it appeared to be the work of the owner, reducing the dataset to 161 authors and 439 repositories. Despite all these steps to clean the data, they still had concerns about noise, noting:

“This data presents difficulties, particularly noise in ground truth because of library and code reuse.”

In their experiments, they selected 50 programmers that had 6–15 files, to ensure a more even class distribution, and reported an accuracy of 60.1%.

4.4 Defences Against Authorship Attribution

As can be seen from the previous sections’ discussions, there have been a great deal of papers published on performing authorship attribution, plagiarism detection and authorship verification. In contrast, very little research has been dedicated to critically evaluating the techniques presented to establish their feasibility in realistic settings, nor to how robust they are in the face of an informed adversary. While performing identification is possibly a more natural and intuitive action to take, it is still important to consider alternative perspectives and test assumptions made, as this will lead to a greater understanding of the field and its limitations, and may help to identify flaws. Natural language authorship attribution has been used in legal cases, so it is crucial to subject it to tests where the desired outcome is negative, rather than positive, to avoid a bias in the literature. Furthermore, as discussed in Chapter 2, there is a real risk that these techniques could be used for nefarious purposes and so offering a defence for those that might be the targets of such operations is also important.

Kacmarcik and Gamon [KG06] were the first to consider how robust known stylometry techniques were to adversarial modifications. Using the federalist papers as their corpus, they decided to select the features to modify independently of selecting the features to use for classification. The features for both modifying and classifying with were all restricted to be frequency-based, however. For the modification list, they first calculated the relative frequencies of all 8,674 words found in the papers. Then a novel feature selection process using decision trees was applied to establish which words to focus on. This process began by generating a decision tree from all the features (note that this does not mean the decision tree with have a depth equal to the size of the feature set; decision tree generation will halt once classification accuracy cannot be improved any further, or all instances have been classified, whichever comes first), then extracting the word at the root node along with its threshold value, taking this to be the most influential word. Decision

trees are normally generated with a greedy algorithm that chooses as the root node the feature and threshold value that produces subsets with the highest information gain, or purity measure relative to the parent set. This process was then repeated iteratively, removing the root node word each time to force the decision tree to select a new feature for the root node. This iteration continued until the classification accuracy dropped below the level of a random guess, producing a list of 2,477 ranked words. SVM classifiers were trained on feature sets of varying dimensions (from three to 70), taken from various other federalist papers studies [MW63, TSH96, HF95, BS98], assigning all but paper 55 to Madison (in agreement with the original studies and others on this problem). Then, taking the top ten ranked words according to their feature selection process and the related threshold values, the researchers modified the feature vectors of the disputed papers to change the values for these words to favour Hamilton, rather than Madison. It was a partial success, with half of the papers being assigned to Hamilton and half still to Madison. When the top ten ranked features were limited to only those words appearing at least once per thousand words, however, the ability to successfully attribute the disputed papers to Hamilton rose so that all the documents were assigned to him and the average reduction in the confidence that Madison wrote each paper was 84.42%. Limiting the words whose values were perturbed to only the more frequent words increased the chances that that word formed part of the feature set used in each SVM classifier. One might ask why not simply choose the frequencies of the words known to be used by the classifiers to modify, but this would not demonstrate that the technique is generalizable. The classifiers with higher dimensionality were found to be more resistant to these modifications, because they are able to base their classification on a wider selection of features; it is less likely that all the features used would be modified by any general approach such as this. Thus, classifiers that are less fine tuned to the particulars of their training data are likely to contain more redundancy than feature sets that have been reduced to only key features. This is an argument for not over-optimizing feature sets, particularly when features are excluded that are seen to be similar to another feature that has a higher impact/influence. In total, only 14.2 changes on average per 1000 words were required to invert the classification from Madison to Hamilton, although applying the *unmasking* approach proposed by Koppel and Schler [KS04] demonstrated that the change in classification may be somewhat superficial. This is to be expected, however, when only the minimal set of modifications are sought, and either including more features or perturbing the chosen features by more than the minimum imposed by the threshold value would serve to deepen the effect, at the cost of more effort.

Brennan, Afroz and Greenstadt [BAG12] sought to build on this simulation of adversarial modifications with empirical results, by running a user study to ascertain the feasibility of such modifications on real documents by human participants. During the user study, three techniques were tried with the participants:

- Manual obfuscation.

- Manual imitation.
- Automated obfuscation (using a translation service).

Participants were asked to submit samples of their formal writing (assignments, essays, theses, professional correspondence, etc.) totalling at least 6,500 words, then asked to write a new passage on a given topic with the intent of obfuscating their writing style and a second passage imitating the style of a "target" author, chosen in advance to be Cormac McCarthy, in particular the style in which his book *The Road* [McC09] was written. This author was chosen because he has a distinctive writing style and *The Road* is well known, having won a Pulitzer prize and a Hollywood movie adaptation. The reason why a distinctive writer was chosen is because the researchers reasoned that the participants would not be skilled or experienced in writing in different styles and therefore selecting an author that may be easier to imitate would compensate for this inexperience and simulate the abilities of a writer who was skilled in switching their style, imitating a less distinguishable author. For the final technique, the participants' sample documents and excerpts from *The Road* were subjected to automated translation by the services provided by Bing[24] and Google[25] between the following languages:

- English → German → English
- English → Japanese → English
- English → German → Japanese → English

For background training data, two corpora were derived, one from student submissions and the other from Amazon Mechanical Turk[26] comprising twelve and 45 authors, respectively. In total, three classification systems were tested:

1. A Neural Network trained with the "basic-9" feature set—this feature set consisted of nine simple statistical and readability features.
2. A synonym-based approach [CH07], where shared words between texts are counted and weighted by the number of synonyms that word has—in other words it is a tf-idf with synonym weighting. The rationale is that the selection of a particular word from among many synonyms is more telling than a word that has few alternatives, therefore it measures the degree of choice exhibited by each word present in both texts.

[24] https://www.bing.com/translator
[25] https://translate.google.com/
[26] https://www.mturk.com/mturk/welcome

3. SVM with the write-prints [IHFD08] feature set.

In their experiments, the authors were split into 1000 sets containing unique combinations of 5, 10, 15, 20, 25, 30, 35, and 40 authors before 10-fold cross validation was used and the mean precision was taken as the overall accuracy. Initially, a baseline test was run to assess the performance of the classifiers on the unmodified documents. The synonym-based approach and write-prints had precisions over 90% on average for five authors, falling to approximately 75% when considering 40 authors. The NN using basic-9 feature set had a relatively poor performance, but the authors wanted to show that even this simple system was still far better than a random guess. When evaluating the success of the participants at obfuscating their style, all three classifier systems were reduced to no better than a random guess, except for write-prints, although even this was barely any better than that. The imitation attack successfully fooled the basic-9 and synonym-based systems, returning 43–68% confidence that the passage had been written by Cormac McCarthy, while write-prints proved to be somewhat more resistant at 18–44%. The machine translation obfuscation experiment demonstrated that this approach would not be effective, for two reasons. First and foremost, it reduced the accuracy, but not by enough to misclassify as another author. Secondly, some sentences of the obfuscated text were often difficult to read or had their meanings altered, while others were completely unchanged. This was not the desired effect, which was that most, if not all, sentences would be altered in subtle ways that did not change the meaning, however the purpose of a translation service is not to paraphrase the original text, but to recreate its meaning as closely as possible in another language. A paraphrasing tool, such as Barreiro's SPIDER [Bar11] would clearly be a better option for automated obfuscation of text; an abundance of these tools are available online [RM17].

In a follow-up paper, McDonald *et al.* [MAC+12] described a system they had written to assist authors in anonymizing their documents, called Anonymouth. This consisted of two subsystems, the first, JStylo, performed feature extraction and could be configured with a choice of feature sets and classifier. The second subsystem, Anonymouth, produced recommendations for the user by leveraging a clustering algorithm and information gain to select features and then advised the users to either move toward or away from the centroid of a cluster depending on whether they should increase or decrease a particular feature. These were shown as markers in an editor window. They carried out a user study involving ten participants, in which they were asked to write a new document, following the recommendations given by the tool. Eight of the ten users were able to do so successfully; i.e., obfuscate their style. The authors noted that attempting to carry out this task on an existing document was very hard and so did not attempt this, focusing instead on the likely use case where a user would be guided by the tool in creating new documents.

Since the culmination of our own research, another paper has been published that investi-

gates adversarial source code stylometry, by Simko *et al.* [SZK18]. This work carries out two significant user studies looking into the robustness of a state-of-the-art source code stylometry system [CIHL+15] to human adversaries attempting to perform targeted mimicry attacks and non-targeted obfuscation of style. The first user study had 28 participants who were asked to modify someone else's code with the express intent of imitating a third person's style. The second study, involving 21 participants (none of whom had taken part in the first study), examined their ability to attribute forged samples to their true author, initially without knowledge of forgery and then again after being informed of the potential forgeries.

In both studies, GCJ competition data is once again used as the corpus, taken from the dataset used by Caliskan-Islam *et al.* [CIHL+15]. The five programmers with the most files (42.8 average) from this dataset were selected as the main subjects of the studies, while variable numbers of authors were chosen as background data to augment the training data from these targets: 0, 15, and 45 (making the total number of classes including the subjects in these training sets 5, 20, and 50, respectively). The additional authors used for background data were chosen randomly from the set of programmers that had submitted at least five files in the main dataset. The baseline precision/recall achieved by the classifier on these datasets was 100/100%, 87.6/88.2% and 82.3/84.5%, respectively.

In terms of setup, the learning algorithm used and its implementation were identical, as was the feature set and data; the only differences were the whitespace and indentation of the files being normalized by Simko *et al.* (see below) and the training set containing variable numbers of instances, whereas Caliskan-Islam *et al.* ensured their training sets had uniform numbers of instances.

The first study, investigating mimicry, was comprised of the following steps, once participants had been briefed on the objective:

1. **Train**: participants were shown files written by two out of the five subjects (labelled X and Y) that had been submitted to solve the same problems, in order to learn their respective styles.

2. **Attribute**: given ten unseen samples, classify them as either X or Y.

3. **Forge**: participants were given the choice of forging either X or Y's style, and were then presented with a file written by the other author to modify. Participants were not given feedback on their level of success, but rather were asked to continue until they believed they had done enough.

4. **Forge with oracle**: taking the forgeries from the previous task, and providing feedback on classification confidences, to encourage them to keep trying to achieve higher confidences.

It is important to note that all code was formatted prior to both studies, using an open-source tool.[27] By normalizing the formatting of all files, Simko *et al.* wanted to focus the minds of participants on the syntactic aspects of style present in the files, rather than the minutiae issues associated with indentation and whitespace, saving them from the tedium of making such changes. The second study comprised the following steps:

1. **Train**: participants were shown files written by one author (labelled X) and the four other subject authors (labelled "notX") in order to learn X's style.
2. **Simple attribution**: given 12 unseen samples, classify them as "X" or "notX". Four of these samples were forgeries, taken from the first study.
3. **Attribution with knowledge of forgery**: after being informed that some samples may be forgeries, reclassify the 12 samples from the previous task.
4. **Find the forgery**: participants were shown pairs of files where one was genuine and the other a forgery and had to decide which was the forgery.

The main finding from the first study was that the classifier they used was not robust against the adversarial attacks. This is significant for our thesis, because one of our goals, as well as forming part of our thesis statement, was to ascertain the ability of current code stylometry techniques to resist adversarial modifications. With feedback on their success (forge with oracle task), participants were able to successfully forge the target's style 66.6, 70, and 73.3% of the time for the 5-, 20- and 50-class datasets. Without feedback, the average success rate was 61.1%. These results do not indicate the level of confidence achieved by the participants; an attack was considered successful if the most numerous class returned by the random forest was the target, which could be as low as 3% for the 50-programmer test set. This in part explains the greater success achieved by the participants against the larger datasets, as the bar is set comparatively lower. Additionally, because machine learning tasks with larger numbers of classes typically result in lower success rates, so it follows that the harder the task of classification, the easier the task of subversion. Unfortunately, the average length of time spent by each participant on the task is not given, as this would indicate first, how difficult the task was for participants and second, the level of tolerance and patience we can expect users of our own proposed system to have for making such changes to their own code. The obfuscation success rate (which they term as "masking") was comparatively higher, at 76.6, 76.6 and 80% for the 5-, 20- and 50-class datasets. Note that masking was not given as an explicit task by the researchers, but rather is

[27] http://astyle.sourceforge.net

what an attempt at forgery that was unsuccessful at being classified as Y was termed, if it was also not classified as X.

Simko *et al.* reported the following as their main finding from the second study:

"While our participants do not spot forgeries when given no information at all, they can develop successful forgery detection strategies without examples of forgeries or instructions about forgery creation." [SZK18]

When performing the simple attribution task; i.e., classification without knowledge of forgeries, the average attack success rate was 56.6%. This means 56.6% of the time, the participants labelled a sample that had been modified to imitate programmer X's style, say, as X. It is not clear whether only the successful forgeries (in terms of fooling the machine classifier) were shown to the participants, or both successful and unsuccessful attempts were shown, nor is it clear whether in the other 43.4% of cases the forged sample was labelled as the true author, or some other author from the five subjects (which would be a successful *masking* attack). After being informed of the potential forgeries in the dataset, participants were only fooled 23.7% of the time, although Simko *et al.* noted this coincided with an increase in the number of false positives; i.e., where an unmodified sample was mistakenly believed to be a forgery. This translated to a success rate in identifying non-forged samples of 76.2%, versus 92.7% when the participants were not aware of the presence of forgeries.

In terms of the types of changes made, the researchers found these were *"overwhelmingly local control flow, local information structure changes, and typographical changes." Local control flow* changes refer to such aspects as the type of loop employed, or nested conditionals instead of guard clauses. *Local information structure* refers to variable naming, declaration location or use of macros. *Typographical* changes are modifying whitespace or placement of braces, etc. It is not clear whether participants were informed the code had been formatted prior to the study. Simko *et al.* commented that the typographical changes may have been due to using an IDE that automatically applied its own formatter upon saving, for example. If these changes were deliberate, rather than automated IDE operations, it would have been pertinent to ask the participants what typographical styles they were identifying within the formatted code. It is possible the automated formatter was not fully normalizing all layout styles due to default settings, etc. This is an important consideration, because even if the machine classifiers do not include whitespace and indentation in their feature sets, the human analysts in the second study may have been including these factors in their decision making process. Indeed, some of the participants did report this as being a factor for them in detecting possible forgeries. It may also be the case that simple attribution without knowledge of potential forgeries was influenced by this too, although participants may not have been aware of the influence it had on their decisions. Local changes such as described above are in contrast to *algorithmic changes*, which are defined as a change in

the method of solving the problem, such as using dynamic programming instead of recursion, or more subtly, refactoring a block of code into a helper function.

Regarding specific changes, 25 out of 28 participants changed variable names, 23 copied entire lines of code from the target, 22 changed the libraries that were imported (and often the specific API calls were swapped for alternative, functionally equivalent calls), 19 changed the indentation scheme and 19 moved variable declarations. The relative ease with which participants were able to successfully fool the machine classifier by making these superficial modifications appears to contradict the findings of Caliskan-Islam *et al.* [CIHL+15], who found running code through the commercial obfuscator Stunnix had little impact on the classification accuracy (dropping from 100% to 98.89% for a 20-programmer dataset, just 1.11%); however, the changes Stunnix makes to perform its obfuscation include variable renaming and removal of whitespace, indentation, and comments. For an explanation of the participants' success, Simko *et al.* offer the following:

"By making small, local changes to only variable names, macros, literals, or API calls, forgers had access to over half of the features"

This suggests the feature set devised by Caliskan-Islam *et al.* may be overly reliant on such content-specific attributes, rather than structural features; however, in their paper, Caliskan-Islam *et al.* stressed the importance of the AST-based features, which are more structural in nature, in producing their high accuracies. Note that to achieve a classification as a particular class in a decision tree, *all* decision points on paths leading to that class must be satisfied. This implies there were sufficient numbers of trees contained within the random forest inducted by Simko *et al.* that were classifying instances based solely on these local features, such as variable names, without including any AST-based features.

For recommendations in relation to local, content-specific features, Simko *et al.* suggested:

"future classifiers should consider fewer of these features, or that these features could be contextualized with their usage in the program."

They also suggested including more features in future classifiers that are harder to forge, such as more complex AST features, or higher-level algorithmic characteristics, as well as including adversarial examples in training sets.

Overall, this paper offers some deep insights into the vulnerabilities of even state-of-the-art classifiers, and highlights the problems that can arise by only evaluating classification systems (of any sort, not merely authorship attribution systems) in terms of their accuracy under ordinary conditions, assuming honest actors. It is complementary to our own research, as we are interested in establishing a method for automated extraction of adversarial modifications, with a developer tool that assists users and counsels them on their use of style, whereas this work explores instead

non-automated means of both forgery and detection with human adversaries. In terms of our thesis statement, it also supports our claim that attribution systems are brittle and prone to misclassifications, providing further justification for our own research. Just as with software, testing machine learning models under only favourable conditions will result in brittle systems that are vulnerable to being exploited by carefully crafted inputs. An effective model would maximize both accuracy and resistance to adversarial inputs, and moreover these may not necessarily be orthogonal to one another.

Chapter 5

Implementation and Methodology

In this chapter, we will discuss the main contributions of the thesis, along with associated implementation details and user study methodology. We will begin in Section 5.1 with a discussion of the contributions this thesis offers to the field. Section 5.2 covers requisite background information necessary for an understanding of the rest of the chapter, including the source of our data, the tools we used and the classification algorithm that forms the core of our recommendation system. Obtaining the data was a necessary task with its own unique challenges, not the least of which was ensuring its integrity and independence—these aspects are discussed in Section 5.3. Following this, in Section 5.4 we discuss the feature extraction phase, including details of the various features we used in our experiments. In Section 5.5, we describe the methods we employed to evaluate the feature set and learning algorithm in their ability to predict the author of the files and repositories in our corpus. Section 5.6 introduces the algorithm developed to produce a set of changes to present to the user in order to change the assigned class of the file under evaluation to some predetermined other class. We provide an overview of how the plugin is used in Section 5.7 and finally, in Section 5.8 we discuss details of the pilot user study, including its aims and objectives and research ethics clearance we received.

5.1 Contributions

It is clear from reviewing the literature that more work needs to be done evaluating the robustness of existing authorship attribution techniques and casting a critical eye over the claims that have been made with regards to its accuracy and scalability. In particular, before being used in any kind of legal setting, whether criminal or civil proceedings, it should be demonstrated beyond

any reasonable doubt that it can truly classify the authors of documents based on their style, independently of content, and that this style is at least resistant to conscious attempts at obfuscation or imitation. Furthermore, there must be a "smoking gun"—some specific and meaningful characteristics that can clearly be linked to one's style. While *n*-grams appear to offer superior accuracy, they are inextricably intertwined with content and the division between style and content at this level of granularity is very hard to discern. The write-prints [IHFD08] work is a step in the right direction in this regard, but it appears to be incredibly brittle at even trivial attempts to defeat it by non-expert adversaries [BAG12]. This is a far cry from its namesake forensic discipline of fingerprint analysis, which, while being far from infallible, certainly provides much stronger evidence and resistance to forgery. Stylometry and other identification techniques often hide behind the term "forensic science" in order to appear to offer undeniable, objective *truth*, cashing in on the brand equity provided by such methods as DNA fingerprinting, but the reality is often less rigourous than we should demand of any technique claiming to be a science or proving the innocence or guilt of an individual.

While there are, at least, some papers investigating natural language stylometry from an adversarial perspective, there are none that look at the source code equivalent. Moreover, no research has yet been conducted in to applying this technique on a large scale in a realistic setting. Finally, if this technique is truly feasible *en masse* against real world data, it represents a threat to the safety of individuals online and therefore defences ought to be developed to assist programmers in protecting their identity against such a threat. To this end, this thesis offers the following contributions:

1. A study into the feasibility of conducting source code authorship attribution at the Internet scale, along with a discussion of the challenges inherent in such a feat.

2. A new set of features for capturing elements of programming style.

3. A tool to assist developers in protecting their anonymity that integrates with a popular IDE and is able to perform feature extraction on their source code and recommend changes to both obfuscate their style and imitate the style of another, specific individual.

4. A novel, practical, algorithm for extracting a change set from a Random Forest classifier in order to produce a misclassification of a particular feature vector as an alternative, known class.

5. A pilot user study evaluating the usability of the tool, the feasibility of manually imitating another's style, and the feasibility of using the tool for this task.

5.2 Background

This section introduces several tools, technologies and algorithms that are key to this thesis, but have so far not been covered in any of the previous chapters. We start with a short summary of the reasoning behind our choice of programming language for our research. We then provide a brief overview of the Eclipse IDE, the development environment that hosted our plugin, and the API we integrated with in order to achieve this. Next, we move on to summarize the Weka machine learning environment, which was used to run our experiments and forms the machine learning core of the plugin. Following this, the random forest classifier is discussed, which our recommendation module utilizes in order to derive its change set. Finally, GitHub and its data API are summarized.

5.2.1 Choice of Programming Language

While there are many similarities between programming languages in terms of style, there are enough differences that trying to extract common features from multiple languages and identifying programmers based on them would be counterproductive. Each language has its own idiomatic usage (e.g., the "Pythonic" way) and the same programmer would probably evince different styles when working with different languages. Furthermore, due to differences in syntax, by attempting to use a common feature set we would lose much potential identifying information. Therefore, it made sense to focus on just one language for our research, which we chose to be C.[1] C is a venerable language and has influenced the syntax of many other languages created since. It is also a subset of the C++ programming language, and therefore many features and style markers that can be used in C would be applicable to other languages. Additionally, C is ubiquitous, found in embedded systems, computer operating systems and mobile devices, as well as application software, particularly when performance is a primary concern, such as for web servers and databases. Lastly, due to is longevity, several generations of computer programmers have learned and worked with C, which should manifest itself in a rich style variance, as good style, best practices and teaching methods tend to change over time. Considering all the above, selecting C as our primary language was a natural choice.

[1] The latest standard of the C programming language can be found at https://www.iso.org/standard/57853.html.

5.2.2 Eclipse IDE

The Eclipse IDE[2] is a very popular open-source Integrated Development Environment (IDE) for developing software in multiple languages. Written in Java, it is also currently the most popular IDE for Java development [Reb14] and enjoys regular stable releases, the current of which, 4.7, was in June 2017 under its bespoke Eclipse Public License (EPL). Support for multiple languages is provided through the Eclipse platform and associated runtime, which consists of a container providing basic services and a flexible plugin architecture to build application-specific services. This container and plugin architecture allows the Eclipse platform to be extended in multiple ways, even beyond software development, as a general editor and GUI with the Rich Client Platform (RCP).[3] Given Eclipse's position as market leader, combined with its open-source license and extensible architecture, it is the natural choice to develop our privacy-enhancing programmer tool.

Plugin Development Environment

The Plugin Development Environment (PDE) provides the plugins and tools required to develop other plugins, with an API and associated documentation. There are wizards for creating new plugins and support services to allow other plugins to be accessed. A workbench UI is provided and a set of editors for a variety of file types. A set of fundamental plugins are available for interacting with the filesystem, editor, workbench, preferences and other basic features of the platform. A manifest editor allows for additional plugins to be defined as dependencies and their associated JAR files added to the classpath. The dependent plugins are loaded at runtime and made available via the platform container. There are a wealth of additional plugins to make available functionality such as preferences, dialog boxes and wizards, asynchronous task execution, progress bars, compilation, code preprocessing and more.

C/C++ Development Tools

The C/C++ Development Tools (CDT) project provides an integrated IDE for C/C++ development. Built on top of the Eclipse platform, it retains its look and feel, but internally defines its own API and extends much of the core Eclipse functionality with features specific to C/C++ development. CDT provides two interfaces for interacting with the internal representation of the code in a compilation unit. One interface represents the semantic, or logical, view of the code,

[2] http://eclipse.org
[3] https://wiki.eclipse.org/Rich_Client_Platform

mapping tokens in the code with their high-level meaning, whereas the other interface represents the syntactic, or physical, view, mapping the code to an Abstract Syntax Tree (AST) representation after preprocessing, so comments and directives are not included by default nor macros expanded in this representation; a separate method call must be made to access these elements.

5.2.3 Weka Machine Learning

Developed at the University of Waikato and released as open-source software under the GNU General Public License (GPL), Weka[4] is a very popular, general purpose machine learning research tool, including data exploration and experimentation features accessible from both a GUI and CLI. Weka supports a wide variety of algorithms for filtering, classification, feature selection and visualization of data [EHW16], all accessible from a single application with a common file type called the Attribute-Relation File Format (ARFF) and several converters that can read XML or CSV structured data, for example, and output ARFF for internal processing. Its broad scope naturally leads to a lack of depth in specific learning algorithms, but this is compensated for by its flexibility, making it ideal for exploratory research and proof-of-concept work. Like Eclipse, it is also written in Java making it easily runnable on a variety of host systems and simple to integrate into an Eclipse plugin, which is the primary reason for its inclusion in our research tool.

5.2.4 Random Forest

The random forest (RF) learning algorithm, first proposed by Breiman [Bre01], is an ensemble classifier (meaning an aggregate of other classifiers) that is constructed from decision trees (DT). Decision trees can grow to be very deep during induction and can suffer from overfitting to the training data, but RFs compensate for this through a process known as *bagging* [Bre96], explained in the next section. RF is known to perform very well on classification tasks, bettering most other algorithms on benchmark tests, according to Caruana and Niculescu-Mizil[5] [CNM06], and once trained is fast at evaluating new instances.

Ensemble Learning

Ensemble learning classifiers work by aggregating the outputs of multiple other classifiers that (should) have been trained on their own subsamples of the training data, possibly with their own

[4] https://www.cs.waikato.ac.nz/ml/weka/

[5] The tests performed by Caruana and Niculescu-Mizil used an unusually large forest of 1024 trees.

subset of the feature set, so each individual classifier is created and trained with its own unique combination of training data and features, while maintaining a statistical correlation with the full dataset and features. The idea is that variance in how each classifier is built, the inferences it makes and noise in the training set are smoothed out across the population of subclassifiers, so that the "wisdom of the crowd" effect is observed and the consensus opinion is likely to be more statistically accurate than any one individual.

Bagging

Bagging is a technique used in ensemble learning, also developed by Breiman [Bre96], and stands for "Bootstrap AGGregatING", as coined by its creator. These two constituent terms refer to the way in which data is split among subclassifiers in the ensemble for training and how the consensus opinion is derived:

- **Boostrapping** refers to how each of the constituent classifiers is presented with its own individual training data, that is still statistically correlated with the overall data set. It makes use of sampling with replacement, meaning a training instance is sampled (drawn) at random from the data set, then replaced, before another is sampled and so on until the desired number of training instances has been reached (typically the same size as the original data set). In this way, each of the aggregate classifiers receives the same amount of training data with a distribution of classes that is still representative of the original class distribution. Furthermore, each of the sampled training sets is almost certain to contain duplicate instances, which will result in its classifier potentially reaching an alternative learning outcome than another identical classifier training with different duplicated instances and class distribution.

- **Aggregating** refers to the manner in which classification consensus is reached by the ensemble. Typically (and in the case of RF) this is achieved with a voting mechanism, where each of the aggregate classifiers is given an equal vote and the overall consensus is simply the class that received the most "votes", with a confidence that is proportional to the number of aggregate classifiers that voted for it. For the regression case, typically the mean of the aggregated classifiers' outputs is taken as the output of the ensemble.

Breiman demonstrated that this technique can improve the performance of unstable learning algorithms, due to the smoothing effect of aggregating the outputs. In this way, we can view the output of any single classifier as a random variable drawn from some distribution whose centre is the optimal output given the training data; therefore the more variables sampled, the closer the

overall output is to the optimum. With bootstrapping, there are instances that were not selected and so did not form part of that tree's training data. These *"out of bag"* instances can then be used to evaluate the accuracy of the tree by classifying them and recording how many were incorrectly labelled. This *"out of bag error"* estimate eliminates the need to run cross validation.

Decision Trees

A decision tree is a simple and intuitive type of classification procedure, based on the binary tree data structure. They are closely related to the classification keys typical to, for example, botany where plant species can be identified based on characteristic features of their anatomy. To perform classification, at each node the input data is tested and depending on the result, either the left or right branch is taken. The leaf nodes contain the class (or numerical value in the regression case) that the input vector has been assigned. The exact structure a decision tree will assume in terms of its depth and the number of branch and leaf nodes it contains primarily depends on the structure of the data—how many classes there are, the clustering of data points, the number of training instances seen and dimensionality of the features.

Training Decision Trees

In the case of machine learning, DTs are inducted, i.e. "grown", automatically using an algorithm that decides, at each node, which feature to test and which value to split on, based on the training data [Qui86]. Most DTs are inducted using a greedy algorithm that selects the best feature/value to split on at each node based on a measure of how well that combination of feature/value divides the training data into subsets. The measures typically used are either information gain (discussed in Section 4.3) or Gini impurity. Using Gini impurity can be thought of as measuring the degree to which instances of the training data are in the "wrong" split according to the distribution of classes in that split, while using information gain can be thought of as measuring the degree with which a feature/value split decreases the "element of surprise" by making an informed guess as to a random training instance's class based only on the class distribution in that split more likely to succeed. This induction process continues until either the measure used cannot be improved any further, or all instances are contained in their own leaf node. A **random** decision tree is inducted in the same manner, except at each node instead of considering all possible features, the induction algorithm is restricted to choosing from a random subset of the features, typically the number of which is fixed at $\sqrt{|X|}$, where X is the set of features.

Putting it all Together

Using a random forest can be summarized as follows:

1. For each tree in the forest, create a training set by bootstrapping the original training data.
2. Induct each tree from its training set using the method outlined above, where each tree is a random decision tree.
3. Present each instance to be evaluated to each random tree in the forest, count the votes received for each class and output the class receiving the most votes as the output of the ensemble.

5.2.5 GitHub

GitHub[6] provides free and public (private repositories are also available for a fee) Git-based repository management on the Internet. It is used by individuals for personal projects and entire teams working on large open-source software projects and can provide anything from a cloud-based backup solution to a complete version control and collaborative programming platform, with usage statistics, defect management and project blogging. As of April 2017, GitHub reported that there were 57 million repositories,[7] while our own research indicated 89 million repositories had been created, which includes public, private and removed repositories. GitHub also provides a data API that can be used to make a limited number of queries anonymously or a greater (but still rate-limited) number of queries upon registration. Their terms of service[8] allows for the use of the API for research purposes, providing the research is not of a commercial nature and is in keeping with their privacy policy.[9] This huge source of software code that is publicly available and has a well documented API for obtaining data is a perfect choice to use in our experiments for gathering real-world training data.

5.3 Obtaining Data

As previously noted, one of the aims of our thesis was to perform a realistic authorship attribution study, to discover, highlight and hopefully overcome some of the practical challenges

[6] https://github.com/
[7] https://github.com/blog/2345-celebrating-nine-years-of-github-with-an-anniversary-sale
[8] https://help.github.com/articles/github-terms-of-service
[9] https://help.github.com/articles/github-privacy-statement/

associated with carrying out a study such as this "in the wild". All prior studies into source code attribution have used corpora derived from student assignments, text books and programming competitions—but none of these sources presents a corpus such as one would encounter in a real attempt at performing large-scale deanonymization. Student assignments are often relatively short, all trying to achieve the same end result, and written by individuals from very similar backgrounds and demographics (particularly with regards to their education). Code from text books is likely to be proof-read and edited, over-commented and, from the author's perspective, a model of perfection. Code from programming competitions is likely to contain much copy and pasted boilerplate code, taken from their other submissions, as well as being short, uncommented and probably not following the competitor's usual style—its purpose is to solve the problem as quickly as possible, it is not intended to be production quality, readable or maintainable.

To this end, we chose to obtain a large corpus of source code from real projects that had been published on GitHub, with the caveat that the code belong to a single author (and truly be written by that person), to ensure purity of style. The multiple-author case is considerably harder, as this obviously introduces multiple styles, and to varying degrees depending on how many lines each author had contributed, whether code reviews were conducted and who by, and so on. Trying to solve the multiple-author case may also not be necessary, as in most cases lone authors are more likely to be the targets of intimidation, as they are more vulnerable and to remain anonymous it is more likely one would choose to work alone. Selecting a popular and public source for our data ensures a wide diversity of both developers, in terms of their background and demographic, and projects, in terms of purpose and size.

5.3.1 The GitHub Data API

The GitHub data API[10] is a RESTful web service returning JSON data that allows, among other things, for data to be retrieved about specific repositories, including activity, history and owner and searches to be run. The search method[11] uses a rich syntax and allows various fields to be searched in and filters to be applied to results. Fields that can be filtered on include various attributes of the repository, such as the creation date, topic and language. The search method appears to be an ideal choice for our purposes, as we can simply choose to search for all repositories, but filter it by language to just C. There is one issue, however, which is a hard limit imposed on searches to only return 1,000 results. This limit means that the same search will always return 1,000 results; it is not a rate limiting feature, but a total limit. It is possible to search by creation date, so in theory one could take the most recent creation date in the search

[10] https://developer.github.com/v3/
[11] https://developer.github.com/v3/search/

results and use that in the following query to only return results greater than or equal to that date. This is made more difficult because searches cannot be sorted on creation date, so we would have to "remember" the most recent date seen in our results. There is still a problem, however, which is that if we can't impose an ordering on our results by creation date, there is no guarantee the results returned will be the 1,000 repositories whose creation dates are immediately after the date we specified in our query. For example, if our initial search is for any repositories written in C, we cannot guarantee that GitHub will return the 1,000 oldest repositories, so that when we take the most recent date seen in those results and apply that to our next query we will get the next 1,000 repositories according to chronological order. On the contrary, it is likely GitHub will instead by default return results ranked according to their popularity, reputation, activity or some other measure of relevance.

As a consequence of this restriction in the standard search operation, we chose to use the "repositories" method, rather than search, which provides the ability to enumerate all public repositories on GitHub.[12] When called with no parameters, this method returns the first 100 repositories according to their ID, with IDs assigned sequentially according to creation date. It is guaranteed that no two repositories will have the same ID and if $ID_i < ID_j$ for some repositories i and j, then repository i was created before repository j. When this enumeration method returns, it includes a HTTP header (`next`) containing the URL of the next page, which includes a parameter (`since`) that restricts the query to only repositories with IDs greater than the parameter; i.e., the 100 repositories with the lowest IDs that are greater than the parameter. It is recommended by GitHub that this header is used for paging through results.

5.3.2 Rate Limiting

A consequence of using the enumeration method rather than search is that all repositories are returned, regardless of the language in which they were written, if any (a repository does not have to contain program source code); this means we must perform our own filtering. GitHub imposes a rate limit of 5000 requests per hour for registered users, so the theoretical maximum number of repositories we can enumerate per hour is the number of requests permitted multiplied by the number of repositories returned per request, which is $5000 \times 100 = 500,000$. The data returned by the enumeration method mostly consists of links to other REST methods for retrieving additional data about the repository and only the id, name, private (Boolean) and forked (Boolean) fields contain non-URL values. Therefore, to decide whether to exclude a repository based on its language, or any other criteria not related to its name or forked status, we must make an additional HTTP request. This requirement makes rate limiting a significant factor, because

[12] https://developer.github.com/v3/repos/#list-all-public-repositories

now for each request to the repository enumeration method, we are forced to make 100 additional requests. This means we can only enumerate a maximum of $5000/1.01 = 4950$ repositories per hour, if we limit by language only. As the ratio of C repositories will likely be small, the number that will be potentially usable will obviously be far lower than 4,950. This limiting factor is further compounded by additional criteria we have to filter our results on, namely that repositories should be single author and contain enough code to be able to train a classifier with. Luckily, the size of the repository in terms of bytes is given in the language method's return data, and is grouped by language (a repository can contain more than one language), but determining single authorship entails making another HTTP request to retrieve data relating to contributors, as a first step (additional methods to ensure sole authorship are discussed in the next sections).

We can see that before a repository can possibly be added to our list, we must make two requests, giving us an upper limit of $5000/2.01 = 2487$ repositories per hour that can be added, if every repository in that hour is a C repository with a size greater than our minimum acceptable size in bytes (we chose 32 KB as our minimum threshold) and appears to be singly authored. The true number will clearly be far lower than this as the proportion of C repositories fitting our criteria is likely to be fairly small.[13]

If there were some way we could filter bad repositories based solely on the data returned by the initial enumeration method, without having to make additional requests, we could increase the number of repositories we are able to scan per hour. To address this, after running our enumeration script for several hours we generated statistics of the names of repositories that were rejected by the script for not containing C, and were able to infer some filtering rules:

1. Names containing other languages' names—it is common in IT to name software projects after the language used to write it, particularly for libraries as these are the most language sensitive; e.g., if looking for a library to use for your Perl program, you ideally want to use a library that was also written in Perl. Including the language in the name makes it easy for others to see if a library is going to be usable by their code without digging around in the documentation. This means we can, with high probability, rule out repositories whose names start or end with words such as "python", "ruby" or "JS" (case-sensitive), as these are highly unlikely to be written in C. There are surely exceptions to this rule, but we are content to trade a small number of false negatives for the increase in processing rates—we have plenty of potential repositories from which to derive our corpus.

[13] In fact, running the query https://api.github.com/search/repositories?q=language:C returns a total count, even though the results that can actually be viewed are limited by paging and the hard limit of 1000. The total count returned is 774,763 out of approximately 57 million (as of April 2017—probably higher now), which is 1.36%, with some of these being too small and many involving multiple authors.

2. A list of common language-independent names that had far higher occurrences of rejected than accepted repositories. Examples of these are "try_git",[14] "test", "android" and course codes for certain Massive Open Online Courses (MOOCs) and even individual universities. This list was dynamic and changed over time, therefore it was necessary to gather the statistics every day to update it. "Android" was an interesting case, which we will discuss next.

3. Names that indicated a repository was being used as a backup for some specific version of third-party software. In this case the owner may be either investigating defects/vulnerabilities they had encountered, or were adding some new functionality to the software, but did not want or know how to fork it or initiate a pull request on the original repository (or the original was hosted elsewhere). This category included common software, such as Android, MySQL and Linux. The determining factor in these cases, and upon which the rule was based, was the presence of a version number in the repository name; e.g., linux-3.4 or htc8960-3.4 (taken from a HTC handset). The reasoning behind this rule was that developers using git to manage their own software would not include a version number in the repository name, after all Git *is* a version control system, it is not necessary and completely nonsensical to create new repositories each time a new version is desired. This should be, and in the majority of cases is, managed in Git itself with the use of "tags". Therefore, only repositories falling into the examples given above would have names containing these version numbers. To detect these cases, we created a regular expression to look for "`-d+\.d+`" at the end of repository names. The one exception to this rule was if the repository in question had a number of "stars"[15] greater than some threshold. This threshold varied depending on the size of the repository: repositories 32 KB–1 MB did not need any stars, 1–3 MB needed more than ten stars, 3–10 MB needed more than 100 stars, 10–30 MB needed more than 500 stars and 30–100 MB needed more than 1,000 stars. Repositories with less than 32 KB or greater than 100 MB of data were rejected regardless of the number of stars. It was reasoned a repository that existed solely as a place to back up someone else's copied code would not receive many stars, but also the smaller a repository was, the less likely it was to be a copy of a popular third-party library or product, as these tend to be larger repositories; hence the requirement for more stars the larger the suspect repository was. Clearly there is a trade-off here and these threshold values were chosen subjectively, but with this consideration in mind.

Implementing these filtering rules based on the repository name enabled us to enumerate far

[14] Due to a tutorial at https://try.github.io

[15] A user can star a repository they find interesting, similarly to bookmarking. GitHub reports on the number of times a repository has been starred as a popularity measure.

more repositories than would otherwise be possible using the naïve approach of checking the implemented languages for every single repository. The trade-off in terms of a small number of false negatives was acceptable given the increase in performance, which at times was as high as 20,000 repositories per hour enumerated.

5.3.3 Ensuring Sole and True Authorship

For our feature extraction to be at all effective, it was of the utmost importance to ensure whenever possible that each repository we included in our corpus represented the sole work of the given owner/contributor. Note that this is far more important in our case than for Caliskan-Islam *et al.* [CIHL+15], for example, as we are training and evaluating on completely independent repositories by the same author. Our classification will almost certainly fail if an author owns two repositories and one of those repositories were written by a different author; i.e., copied. Even if all the repositories for a given author were written by others, it is unlikely that the copied code belonged to the same other author, so there would still be an inconsistency in style. In the case that the copied code did belong to the same other author, it would still be undesirable, because the other author is also likely to be in our corpus, so there would be an increase in noise. While we were able to reduce such noise to a large extent, it is impossible to fully verify that each and every repository contains code that was written by the owner/contributor, so noise remains an issue, probably to a greater degree than other studies that had more controlled corpora, such as student submissions. The reason why verifying the owner is the sole and true author of the code contained in the repository is impossible is because we rely on the author to leave some tell-tale sign that the work was truly theirs or not. Tell-tale signs include naming their repository in a predictable manner that indicates it was copied, or multiple contributors pushing with distinct user accounts, or, as we will see in the next section, the same code existing elsewhere in our corpus.

To this end, we began by filtering repositories that were marked as forks, as these are typically copies of other authors' repositories and so do not constitute original work. Even if the owner of the forked repository were the same as the owner of the original, the original would be enumerated in due course, so there was no need to include both. Our next method for determining sole authorship was simply to check the contributors' data provided by the relevant method in the GitHub API,[16] the URL of which is returned for each repository by the enumeration method. If there is more than one contributor in the returned data, we can safely filter the result. Following this, once cloned, repositories were checked again for the amount of source code they contained, as the size in bytes indicated by the web service was not completely accurate—some

[16] https://api.github.com/repos/username/reponame/contributors

non-C files were marked as C source, as were header files, which we did not use for feature extraction. We used the `file` Linux utility and a check for .c file endings. We also excluded repositories containing fewer than ten and greater than 100 individual files. Requiring at least ten files ensured we had sufficient training material, while repositories containing greater than 100 files were found to be more likely to be copies, rather than true single-author projects.

An additional check for multiple-author repositories was carried out by examining the commit logs; if there was more than one unique name and email address combination, the repository was marked for manual inspection. 635 repositories fell into this category, taking several hours to check. This involved comparing the different names/email addresses seen in the logs to determine if they probably belonged to the same person or different people. The reason these were manually checked, rather than being automatically excluded, was because they were often the same person committing from either a different location with a different SSH key, or they had changed the name or email address associated with their account, which does not automatically update the historical commit data. A manual check was necessary, because the clue that reveals it is the same person varies; trying to define an exhaustive set of rules for these was not practical, at least not in the time it took to carry out the manual check. Some examples of variations seen in names and emails were:

- Axel ”0vercl0k” Souchet and Axel Souchet
- Jens John and Jens Oliver John
- adamsch1 and Shane Adams—this one is more subtle, but as both contain the string “adams” it was enough to suggest this was probably the same person
- amotta91@gmail.com and motta@ubuntu.epfl.ch
- angel.d.death@gmail.com and angeldeath29@gmail.com

5.3.4 Removing Duplicate and Common Files

Another source of noise within our dataset was from duplicate files between repositories belonging to different authors and the same author, and third-party library code included in subfolders of a repository. It is common practice to put code from third-party libraries into “lib” or “ext” folders, so these were removed if present. Following this, md5 hashes were taken of all source code files in our corpus, and if duplicate files were found in two different repositories (belonging to different authors), this was taken to be an indication the repositories contained copied code and they were both excluded from our dataset. Note that if this meant we only had one repository

left for an author, we also excluded that as we require at least two repositories per author in our dataset. It may have been the case that only some files were copied, but we considered the task of determining what code probably belonged to the repository owner and what did not to be too complex for automation and too time-consuming for manual checking. Furthermore, we did not attempt to determine if one of the authors had copied from the other, or both were copied from a third source for much the same reasons.

The final step we took was to also remove duplicate files between repositories belonging to the same author, ensuring that one copy of that file remained. Obviously, there is nothing suspicious about an author reusing the same code for several of their projects, but for our evaluation purposes, we want to remove such code to avoid overfitting and to make the evaluation strenuous for our learning system. If, after this removal, a repository was found to contain fewer than ten source code files, it was excluded and once again authors were excluded if they were represented with only one repository after this removal.

5.3.5 Summary of Data Collection

In total, then, we enumerated 66,619 C repositories on GitHub, collecting data on 11,164 that GitHub reported as containing 32 KB or more C code with a single contributor. After cloning, this number was reduced further after application of our filtering criteria to 1,618 repositories representing the work of 669 authors. Once duplicate files, third-party libraries and copied repositories had been removed, 1,381 repositories from 645 authors still retained sufficient numbers of files to meet our inclusion criteria. Out of these, 120 authors had only a single repository left in the dataset; i.e., their other repositories had been excluded by the filtering criteria. Therefore, our evaluations were carried out on a final tally of 1,261 repositories from 525 authors. Table 5.1 gives the distribution of the number of repositories per author in our dataset, while our filters are summarized below:

1. Removing repositories whose commit logs contained names/email addresses of different people.
2. Removing code from “lib” and “ext” folders.
3. Excluding repositories with different owners that contain identical files, checked by hash code.
4. Removing duplicate files from the same author, checked by hash code.

Table 5.1: Distribution of repositories per author

Repositories	Authors
2	396
3	88
4	25
5	8
6	2
7	2
8	0
9	2
10	1
11	1

5. Excluding repositories not containing a minimum of 32 KB of C source code (using a more accurate measure than provided by GitHub) spread over 10–100 files, after the preceding filters had been applied.

6. Excluding authors that were only represented by a single repository, after all preceding filters had been applied.

5.4 Feature Extraction

As our tool is an Eclipse plugin, we wanted to integrate the task of feature extraction within the plugin as much as possible and take advantage of the rich services provided by the IDE for code parsing. Eclipse CDT provides a convenient mechanism for traversing the AST it constructs internally, with an abstract class (ASTVisitor) one can implement and pass as an argument to the object implementing the base interface of the AST (IASTTranslationUnit). Our feature set is constructed largely from this tree traversal, where the Observer pattern [JGVH95] is used to notify special classes that contain the logic to extract a particular feature category. Features that are based on comments and preprocessor directives, such as macro definitions, are extracted with a different mechanism as described in Section 5.2.2. In this case, comments and directives are parsed with their own specialized feature classes.

We decided to extract features in the following categories:

- **Node Frequencies**—The frequency of AST node type unigrams in the AST (discussed in Section 5.4.1).
- **Node Attributes**—The frequency of AST node attributes. These are dependent on the node type and provide more detail on the content of that node; e.g., for the node type IASTBinaryExpression, there is an attribute "type" that defines whether the expression is addition, subtraction, etc. (discussed in Section 5.4.2).
- **Identifiers**—Naming conventions, average length, etc. (discussed in Section 5.4.3).
- **Comments**—Use of comments, average length, ratio of comments to other structures, etc. (discussed in Section 5.4.4).

These categories combined to give us a total of 265 features. We purposefully exclude typographical features, such as indentation and whitespace, as these inflate the accuracy of a classifier at the cost of susceptibility to trivial attacks. Furthermore, as Simko *et al.* [SZK18] alluded to, asking users to make many minor typographical modifications is tedious and frustrating, while there would be little research novelty in automating such changes within our tool as code formatters are already very common and it would make our adversarial attacks less compelling. Instead, we invoke Eclipse's built-in code formatter in order to provide default protection for our users against the weakest attribution systems, without considering such modifications as being successful defences. We also decided against implementing node bigrams as used by Caliskan-Islam *et al.* [CIHL+15], as this results in extremely high-dimensional feature vectors, or character *n*-grams, as implemented by Frantzeskou *et al.* [FSG+07], as these would be completely impractical for producing recommendations to the user, being made up of combinations of partial words, tokens and whitespace. As this is exploratory work and the main purpose of our thesis is to explore defences against attribution, rather than performing attribution itself, this comprehensive but not exhaustive set of features was chosen to be representative of the features one might employ if wishing to perform authorship attribution while simultaneously being of a high enough level to be the source of meaningful advice to present to the user. Our aim is to demonstrate the feasibility of an approach to parse the model generated by a learning algorithm to automatically produce change sets for misclassification. Because of the generality of this goal, we have provided a flexible framework that can accommodate varying feature sets; exploring such alternative feature sets to discover those that succinctly capture an author's style, while being amenable to producing actionable advice, would be an excellent avenue for future work.

5.4.1 Node Frequencies

Counting Nodes The AST class hierarchy used by Eclipse is not a one-to-one mapping between tokens seen in the code and classes/interfaces in the hierarchy. Each node typically implements multiple AST interfaces depending on its syntactical role, making the AST, when viewed from a class hierarchy perspective, somewhat "hairy" (see Figure 5.1). The concrete class each node object instantiates is not a suitable level of abstraction to count occurrences for our unigram frequencies, because the Eclipse plugins involved have placed restrictions on referencing these classes from external plugins, which are enforced to discourage their use. This is to protect calling code from becoming too tightly coupled to a particular version of the Eclipse platform; classes are not documented and may be subject to change at any time between even minor version releases. Therefore, it is not advisable to assume beforehand which classes are going to be present in the AST if we wish our plugin to be at least somewhat portable between different versions of Eclipse; however, at the same time we must predefine our feature set as it is also not practical to include within a plugin JAR file all the training data in its raw, unextracted form (e.g., source and associated files). Therefore, we must include the background training data in an extracted, normalized form of feature vectors ready to be imported directly into our learning system (Weka). This means for our unigram node frequency features, we must focus on the interfaces, rather than the implementing classes, which presents us with the problem of which of the multiple AST interfaces that a class implements should it be counted against? The solution we chose for this problem was to derive a list/array of all *relevant* interfaces (i.e., extensions of the root ASTNode) that a node class implements, then record their counts against *each* of their parent interfaces. In Java, classes can implement multiple interfaces and interfaces can extend multiple parent interfaces. This can cause similar issues to multiple class inheritance when trying to carry out reflection/introspection operations or navigate a class hierarchy. We decided to use a list/array structure rather than a set, because an interface may be implemented or extended twice in the same class hierarchy. In order to keep child counts in the correct proportion to their parents, i.e. the sum of the occurrences of all child nodes equal to the occurrences of their parent, it was necessary to count interfaces each time they were encountered, even if this meant double counting. Indeed, in this case multiple counting is unavoidable if we wish our relative frequencies to be meaningful—it is no use simply taking the interface that appears lowest in the class hierarchy, immediately above the concrete class, because the class may directly implement multiple relevant interfaces and moreover a class may directly implement an interface that is not a "leaf" interface. In short, the problem we are facing is that each node in the AST is not a single entity, but due to polymorphism, has its own class hierarchy, which is a graph, not a tree. Mapping from one to the other to count occurrences for the purposes of frequency derivation requires that each node in the graph be counted, regardless of its type. The counts for the two nodes featured in Figure 5.1 are summarized in Table 5.2.

Figure 5.1: Two AST nodes with their respective class hierarchies. The pale green ellipses are the nodes, while orange labels represent classes (both concrete and abstract) and turquoise labels represent interfaces.

Table 5.2: Node counts for the two AST nodes depicted in Figure 5.1. An italicized name is the concrete class. Neither these nor abstract classes are counted, only interfaces, hence the 0 counts for rows whose names do not begin with an 'I'.

Class/Interface	**Count**
CASTFunctionDefinition	0
ASTNode	0
IASTFunctionDefinition	1
IASTDeclaration	1
IASTNode	1
CASTSimpleDeclSpecifier	0
CASTBaseDeclSpecifier	0
ASTNode	0
ICASTDeclSpecifier	1
ICASTSimpleDeclSpecifier	2
IASTSimpleDeclSpecifier	1
IASTDeclSpecifier	3
IASTNode	3

Feature Representation In total, there are 89 AST interfaces we are interested in for the purposes of frequency derivation. As discussed above, it was necessary to count the occurrences of a relevant interface for each of its parent interfaces, up to the root ASTNode interface. Each count is relative to its parent, so this means there are multiple independent counts of the same interfaces in our feature set, named *Child-ParentNodeFrequency* to avoid collisions with other features based on the same child interface. This resulted in a total of 95 AST frequency features, representing the frequencies of the 89 interfaces.

5.4.2 Node Attributes

Many of the interfaces encountered define attributes that provide additional information about the particular node type it represents. Table 5.3 provides some examples of this. The information provided by these attributes is relevant to us, therefore we created features representing, for example, the ratio of each unary expression type to the total number of unary expressions, which would vary between authors that had a preference for prefix instead of postfix increment/decrement operators.

Table 5.3: Examples of node attributes

Interface	Attributes
IASTBinaryExpression, IASTUnaryExpression	int getOperator()—returns a flag indicating the type of operator; e.g., addition, postfix increment
IASTDeclSpecifier	int getStorageClass()—returns a flag indicating the storage class of this declaration; e.g., extern
IASTLiteralExpression	int getKind()—returns a flag indicating the type of the literal; e.g., character

5.4.3 Identifiers

Identifiers can provide a rich source of style information, demonstrating the programmer's preference for certain naming conventions, for example, that can differ depending on educational background, experience and native language. Depending on the method by which a programmer learned their craft, they may have been taught or read about a variety of "best" practices regarding naming of variables, functions, structures and so on. This can also indicate their prior experience with other languages, as different languages sometimes have their own idiomatic preferred naming conventions. Their experiences working in teams either professionally or on open-source

projects will also have moulded their preferences, with different organizations promoting different schemes amongst their team members; e.g., Microsoft's use of Hungarian notation in its Win32 API.[17] Finally, programmers often use their native language when naming elements of their code, particularly if they are accustomed to working in teams where their native language is spoken. This could have an impact on average name length, or character frequencies, for example.

To try to capture some of these preferences in our feature set, we labelled identifiers according to the following characteristics (maintaining separate lists for the names of variables, functions and structs/unions):

- **Title Case**—First and subsequent words within the identifier are capitalized; e.g., TitleCase.
- **Camel Case**—First word is lower case, but subsequent words are capitalized; e.g., camelCase.
- **All Caps**—All characters are in uppercase; e.g., ALLCAPS.
- **Underscore Delimited**—Words are separated with an underscore; e.g., underscore_delimited, or ALL_CAPS.
- **Underscore prefix**—The identifier begins with an underscore; e.g., _identifier.
- **Single char**—The identifier is made up of a single character only; e.g., x.
- **Hungarian notation**—The identifier uses the so called Hungarian notation, which prefixes the name with the type of the variable, or return type of the function, in lower case before reverting to title case for the remainder of the identifier. This is harder to determine, as the prefixes are often a single character, which could be part of a non-Hungarian notation identifier. For example 'a' is used to denote an array, but an identifier may be called "aCar", for example, using camel case notation. These false positives are unavoidable, but in code that is not actually using Hungarian notation, they would make up a small number, so a high threshold could be used to separate the programs that genuinely use Hungarian notation from those that returned a false positive result.

We refrained from using character frequencies or *n*-grams, with one exception described below, as the sample would be far too small in any given file. Also, as we are precomputing

[17] https://msdn.microsoft.com/en-us/library/windows/desktop/aa378932(v=vs.85).aspx

our background training data, before the user's training data can be known, comparing files with one another is not possible, therefore shared names or words cannot be utilized. In any case, shared words are likely to be highly identifying when training/evaluating from the same corpus, i.e. when using cross validation, due to the shared functionality between files from the same software repository, but would be far less so when evaluating files from different projects by the same author, as would be the case for our tests. Unigram frequencies of all possible words is similarly impractical, due to the high number of non-dictionary words that would be present, and as we do not know what data the user will want to train and evaluate on in advance, we cannot precompute the words that will be present in our corpus.

In addition to the above features based on labelling identifiers, we also extracted the following:

- The character frequencies of single-char identifiers, as it was postulated programmers would demonstrate a preference for particular characters when using single-char identifiers. While many programmers may favour i, j and k for their control variables in for loop constructs, others may favour a, b and c. Other programmers may habitually use single-char identifiers in regular code.
- Average length of identifiers.

5.4.4 Comments

Similarly to identifiers, comments can be highly individualistic as they represent the least restricted aspect of a program's structure. There is no strict syntax within a comment and styles vary greatly both in terms of number and length of comments, as well as their contents. Comment contents range from simple notes on the intention of one or more statements, details of bug fixes, TODOs (tasks) and code fragments, to forming part of formal API documentation. In our feature set, we catalogue the following features related to comments:

- **Comment frequency**—As a ratio of total nodes.
- **Single vs. multi-line preference**—i.e., // or /* */
- **Presence of header or footer comments**—Before the first non-comment token, or after the last.
- **Character type ratio**—ASCII to non-ASCII, letters to non-letters and control chars.

- **Use of frames**—When non-letter chars are used to place borders around comments and between sections; e.g., ********

- **Use of tags**—Providing metadata attributes, such as author, date and version.

In an earlier version of our feature set, character unigram frequencies were extracted; however, after testing it was decided to remove these as it was entirely impractical to implement recommendations based on altering such frequencies; e.g., increasing the occurrences of one or more letters and decreasing others. Furthermore, our experiments revealed only a modest drop in classification accuracy after removing these features, illustrated in Section 6.1.

Once again, despite the actual words used in the comments probably revealing much about authorship, creating features based on bag-of-words or word unigrams would be corpus-specific, and as the user's training data cannot be known in advance, it would be of little use without including either the full representation of the background corpus, or at least the set of words and their counts/files/author details found in the background corpus, so that the full training data set including the user's could be dynamically created. This latter option is feasible, although would make the plugin larger and somewhat more complex, and is a possibility for future work. There is also the question of *realism*, because although finding common words, such as names, within multiple repositories by the same author would indeed be an identifying feature, we should remind ourselves that the authors of these repositories are not *trying* to disguise their identity—they have openly linked these repositories together to one account. In the case where a programmer is concerned about remaining pseudonymous, it is unlikely they would include their real name, or a user name that links them to an overt account they also use. It could also be the case, however, that they have copied and pasted code they wrote in another repository into their covert repository and this copied code includes their other username, so this could actually be a realistic consideration. These, and other related questions, remain open problems in authorship attribution in both the natural language and source code settings.

5.4.5 Other

Other features that do not fall into any of the categories mentioned above are:

- **Size of AST**—Total number of nodes (objects only), reveals more about the complexity than a simple line count.

- **Max breadth and depth of the AST**—Can reveal something about the nesting habits of the developer, i.e. a narrower, deeper tree indicates preference for more deeply nested code.

- **Ratio of leaf to branch nodes**—Also indicates a preference for shallow or deep nesting.
- **Fraction of if statements with an else clause**
- **Average number of parameters to functions**

5.5 Training and Making Predictions

Now that we had our background data corpus and a set of features we could extract from it, we wanted to ascertain the ability of our feature set and chosen random forest classifier to make predictions about the author of a file, and subsequently of an entire repository. We decided to use a similar training and evaluation methodology to hold-one-out (see Algorithm 1), meaning, for each repository in our dataset, we trained our classifier on all repositories except for the one under evaluation, then we classified each file in the repository being evaluated and recorded the result. The overall prediction of the author of an entire repository was simply the most numerous author class assigned to each file in the repository. In the case multiple authors tied for the plurality, the repository classification was deemed unsuccessful, as was obviously the case when the most numerous class was incorrect. In general, if the number of classes is greater than the number of files, as in our case, the minimum number of successfully classified files necessary for a successful repository classification is two. In our case, the number of classes will always be greater than the number of files, as we have restricted the number of files to less than or equal to 100, but our class set has cardinality greater than 100. Another way to derive the repository prediction using aggregation would be to sum the individual trees' output predictions for each class and file, then take the class that received the most votes from individual trees as the repository prediction, or use this figure to produce a "top n" list of possible authors. It would be worth exploring the differences this makes in future work. We present the results of our hold-one-out evaluation against the entire corpus, with and without character unigram frequencies, in Section 6.1.

5.6 Making Recommendations

A significant part of our system, and crucial to its effectiveness, was the ability to make recommendations to the user on what aspects of their code they should change in order to disguise their identity as a particular target author. Our reasons for imitating a specific individual, rather than just "any" author, or "no" author (obfuscation) are as follows: first, with obfuscation the aim is

Algorithm 1 Method for performing hold-one-out evaluation

```
for all r ∈ repositories do
    training_data ← repositories \ r
    evaluation_data ← {f | f ∈ r}
    classifier ← train(training_data)
    results ← ∅
    for all f ∈ evaluation_data do
        prediction ← evaluate(classifier, f)
        result ← ⟨f, prediction⟩
        results ← results ∪ result
    end for
end for
// Find the most numerous predicted class
return max_prediction(results)
```

to reduce the classification confidence to some target value, preferably to that of a random guess or below that of some other author. This typically would involve perturbing the feature vector to a position just outside the boundaries of that class in the feature space. A second classifier trained on the same data, or with alternative background data, may derive a different boundary that places the perturbed feature vector within the bounds of its original class. Furthermore, there is the problem of selecting which features to perturb, and by how much. Imitation of "any" author suffers from many of the same drawbacks. Granted, the direction and magnitude of perturbations is now more clearly defined (toward the nearest other author in the feature space), but if it is known that the feature vector has been perturbed, the original author could be determined by finding what classes are nearest to the feature vector's position other than the given class. Indeed, we must assume everything is known about our defences, and design a system that is secure despite this knowledge; this design requirement follows from Kerckhoffs' Principle [Ker83].

5.6.1 Requirements

We have the following requirements for our system:

1. The advice should relate to something that is possible for the programmer to change, so not refer to something that is inherent to the programming language itself, or violate syntactical rules of the language.

2. The recommendations should not contradict one another, so not advising the user to increase one feature while simultaneously decreasing another that is strongly positively correlated.

3. The user should be presented only with changes that contribute to the desired misclassification—either reducing confidence in their classification or increasing it in the target author.

4. There should be a minimum of effort on the part of the user; they should be presented with the minimum set of changes required to effect a misclassification as the target.

5. The recommendations should make sense to the user; they should be able to understand what is required.

6. Similarly, the advice should not be too vague; there should be a clear connection between the recommendation and the content of each file.

7. As our tool is aimed at open-source developers, we want them to be able to implement the changes without having a large negative impact on readability of the code.

Of these requirements, the first two are the most important and possibly easiest to ensure. The first equates to *correctness* and is mostly a requirement of feature selection, extraction and representation. If our feature set captured elements of the code that were beyond the control of the programmer and simply a result of the programming language syntax, clearly this would be pointless and not contribute to the overall classification, and in fact would probably be a hindrance. The second requirement equates to *consistency* and refers to our ability to analyze the dataset and the relationships between features; we cannot simply derive recommendations from the features in isolation, but must take into consideration how features are related and what impact a recommended change has on the rest of the features.

The third and fourth requirements relate to how we extract meaning from the classification model itself. The third equates to *relevance* and can be met by only considering features that are actually used by the learning algorithm. With random forests, a form of feature selection occurs during induction, due to the algorithm selecting the best feature/value split at each node from among a random subset of the total features. Therefore, the more important and influential features will be seen with greater probability in each tree. With a sizable forest, the inherent variation introduced with the randomness of bootstrapping and the selection of the feature subset at each node will be smoothed out. This should result in the most relevant and strongly correlated features appearing most often, while the probability that a unimportant or irrelevant feature would appear with any great frequency is vanishingly small. This gives us the ability to "rank"

recommendations according to their influence on the overall (mis)classification, which is governed by how many paths within the decision trees contain the feature. If every path in every tree in the forest that leads to a certain class contains the same feature, that feature has maximum influence on that classification. If every path in every tree contains the same feature, across all classifications, that feature has maximum influence on the population as a whole. Conversely, if a feature does not appear in any path leading to a certain class, that feature can be ignored or assigned any arbitrary value, as it does not contribute to the classification. In some situations, it can be beneficial to know which features fall into this category, as being able to assign arbitrary values to a feature can help when needing to adjust values in another, related feature that does contribute to the classification, without directly modifying the underlying content relating to that other feature (more on this later). By only making recommendations to the user that will actually affect their classification, we can maximize the effectiveness of the plugin, and reduce the impact on the original code. The fourth requirement equates to *efficiency* and can be met by calculating some form of *effort requirement* to transform the user's feature vector into one that elicits a classification as the target. If our analysis presents us with a set of feature vectors that each produce a classification as the target, then we can use this measure of effort between the user's vector and each of the adversarial vectors to select the one requiring the least effort.

The fifth and sixth requirements are related to the tool's communications with the user. The fifth equates to *simplicity* of communication, and can be met by using language that is familiar to programmers, but without introducing too much jargon. A certain level of familiarity can be assumed with technical terminology, but where applicable, alternative terms should be used for the same concepts, as not everyone will use the same lexicon. The sixth requirement equates to *clarity* and is mostly related to the features used. Features based on vague patterns found in the file contents that are not tied to discrete semantic objects, such as character *n*-grams rather than words, are going to be hard to relate to real content. Any feature that derives its values from the contents of a file must be based on semantic structures found in that content, therefore more concrete features must exist that relay the same information.

The final requirement equates to *non-intrusiveness*, and is the most difficult of the requirements to meet. It is dependent, to a large extent, on the person implementing the change, and how exactly they choose to do it. However, it is also dependent on the feature set and the interpretation of the classification model. As mentioned above, vague recommendations are hard to relate to real content and can result in highly intrusive changes that affect readability and other desirable aspects of the code, possibly even to the detriment of performance and correctness. If the classification model is interpreted incorrectly, superfluous or over-zealous changes may be recommended and the more changes made, the more chance there is of one having a negative impact. As automated program comprehension is far beyond the scope of this thesis, we cannot expect our plugin to "know" what recommendations it makes will maintain the readability and

desirable behavioural aspects of a program—this must be the job of the user in their interpretation of the advice. All we can do, then, is try to ensure the features are at a level of abstraction that is not too vague, nor too far reaching and fundamental; while features at the lowest level, such as character or byte *n*-grams, make for vague and confusing recommendations, features at the highest level, such as design patterns or overall architecture, make for intrusive and time-consuming alterations, leading to extensive refactoring.

Summarizing these requirements as a set of desirable properties for our system, we have that it should be *correct* and *consistent* with regard to feature selection and analysis, while being *relevant* and *efficient* in selecting changes. These changes should be *non-intrusive* and communicated with *simplicity* and *clarity* to the users.

5.6.2 Parsing the Random Forest

A random forest contains a great deal of information about its training dataset. As discussed above, Requirement 3 states that the user should only be presented with changes that will affect their classification. We described an approach to solving this by only considering features that are present on paths leading to our user's present classification, and on paths leading to leaf nodes that classify as our target. Finding the set of features and their split points that contribute to our current classification is relatively straightforward, requiring only a minor modification to Weka to add a method that can return this information. As we already have our user's feature vector, we simply need to evaluate that vector for each tree in our forest, recording the feature splits found at each node along the way. As data structures, trees have the property that each node is itself the root of its own subtree. This recursive property lends itself to recursive algorithms for traversing the tree structure quite elegantly. Our algorithm for deriving the set of feature splits for a particular feature vector is also recursive in nature and is given in Algorithm 2.

Finding the set of features and their splits points that exist on paths leading to our target's classes is a little more complex, as we do not have a feature vector to traverse the tree with, however a recursive solution is once again our preferred approach, using a post-order traversal, and can be summarized as follows:

1. Traverse each path in the tree down to the leaf nodes;
2. Check the majority class at each leaf node—if it is our target, start a list whose first element is the leaf node;
3. At each branch node, check whether its child nodes eventually lead to a leaf node whose class is the target. If so, add the current node to the start of each list returned by the

Algorithm 2 Method for retrieving feature splits for a given vector from a random forest

function GETSPLITS($\vec{x} : (x_1, x_2, \ldots, x_n), forest : \{tree_1, tree_2, \ldots, tree_n\}$)
  $allSplits \leftarrow \emptyset$
  for $i \in \{1, \ldots, |forest|\}$ **do**
    $allSplits \leftarrow allSplits \cup$ GETSPLITSFORTREE($tree_i, \vec{x}$)
  end for
  return $allSplits$
end function

function GETSPLITSFORTREE($tree : (node, tree_{left}, tree_{right}), \vec{x}$)
  if $node = null \vee (tree_{left} = null \wedge tree_{right} = null)$ **then**
    return $\emptyset$
  else
    // $node : (i, feature_i, split \in \mathbb{R})$
    $splits \leftarrow splits \cup \{node\}$
    if $x_i < split$ **then**
      $splits \leftarrow splits \cup$ GETSPLITSFORTREE($tree_{left}, \vec{x}$)
    else
      $splits \leftarrow splits \cup$ GETSPLITSFORTREE($tree_{right}, \vec{x}$)
    end if
  end if
  return $splits$
end function

child node, additionally indicating whether the split is left or right (less than or greater than/equal, respectively);

4. Once the root node has returned, there will be n lists, one for each leaf node that classifies as the target, containing all the feature/value splits and their left/right direction for the nodes on the paths leading to the leaf nodes;

This is presented more formally in Algorithm 3. Note that this algorithm returns nested data structures, organized by tree, path and node, in that order. Most nodes will appear in more than one path, as multiple leaf nodes may be reachable from each branch node, and each leaf node is the endpoint of a distinct path.

Note also that this algorithm can be modified to additionally return the feature vector of each training instance of our target class, if the random forest is adapted to have a "memory" of its training instances. We made this adaptation to Weka's implementation of Random Forest, for reasons that will be elaborated in the next section.

5.6.3 Analyzing the Split Points

Now that we have a set of feature/value splits that lead to our current classification and our target's, we can find a set of changes that, if implemented, will lead to us being classified as the target. We can actually reduce this to simply finding a set of feature value *intervals* that will be classified as the target, then perform a difference calculation with our current feature values and present to the user as recommendations the features that are outside this interval, with information on exactly how much to perturb each feature in order to move within the range. Intervals are the natural representation when dealing with decision trees, because each split point in a tree defines two intervals; e.g., if the split point is x, then the two ranges are $[0, x), [x, +\infty)$. Two split points define three ranges; e.g., $x, y \in \mathbb{R} : x < y \rightarrow [0, x), [x, y), [y, +\infty)$. In general, n split points define $n + 1$ intervals. If we include the implicit end points 0 and $+\infty$, then there is one fewer interval than there are split points. We now have the following representation for multiple split points extracted from a random forest:

$$splits_f = \{s_1 = 0, s_2, s_3, \ldots, s_{n-1}, s_n = +\infty \mid s_i < s_j\}$$

Representing the following intervals:

$$intervals_f = \{[s_1, s_2), [s_2, s_3), \ldots, [s_{n-1}, s_n)\}$$

Algorithm 3 Method for retrieving paths for a given class from a random forest

```
function GETSPLITSFORFOREST(forest, targetClass)
    paths ← ∅
    for all tree ∈ forest do
        paths ← paths ∪ GETSPLITSFORTREE(tree, targetClass)
    end for
    return paths
end function

function GETSPLITSFORTREE(tree, targetClass)
    paths ← ∅
    if tree is a leafNode then
        if nodeClass = targetClass then
            paths ← paths ∪ {()}
        end if
    else
        for all child ∈ tree.children do
            childPaths ← GETSPLITSFORTREE(child, targetClass)
            for all childPath ∈ childPaths do
                childPath ← (feature, split, direction ∈ {left, right}) ∪ childPath
            end for
            if childPaths ≠ ∅ then
                paths ← paths ∪ childPaths
            end if
        end for
    end if
    return paths
end function
```

Where $|intervals_f| = |splits_f| - 1$. In our case, we are also concerned with the *direction* of the split; i.e., whether the path leading to our target class requires that the value of the feature be greater than/equal or less than the split point, or both (splits found higher in the tree, i.e. at lower depths, often lead to leaves of a given classification in either direction). We can construct two sets, Γ and Λ, containing all the split points where the path we are interested in follows the greater than/equal or less than split, respectively:

$$\Lambda = \{\lambda_1, \lambda_2, \ldots, \lambda_{n-1}, \lambda_n = +\infty\}, \Gamma = \{\gamma_1 = 0, \gamma_2, \gamma_3, \ldots, \gamma_m\}$$

We can place an element in both sets to represent the case where paths exist following both sides of the split. If we order the two sets, such that:

$$\forall i \in \{1, \ldots, |\Lambda|\}, j \in \{1, \ldots, |\Gamma|\} : \lambda_{i-1} < \lambda_i \wedge \gamma_{j-1} > \gamma_j$$

And: $\lambda_0 = minval(\Lambda)$ and $\gamma_0 = maxval(\Gamma)$, we can construct an interval for each:

$$[0, \lambda_0), [\gamma_0, +\infty)$$

Such that values in the intervals are guaranteed to satisfy all the split points in their relative sets. Furthermore, if $\lambda_0 > \gamma_0$, we can define an interval: $[\gamma_0, \lambda_0)$ that is guaranteed to satisfy all split points in both Γ *and* Λ:

$$\gamma_0 \leq x < \lambda_0 \implies \forall \gamma_i \in \Gamma, \lambda_j \in \Lambda : \gamma_i \leq x < \lambda_j$$

As our decision trees are inducted from multiple training instances, it can easily be the case that our constructed sets Γ and Λ contain split points that are less harmonized, causing an overlap between elements' split points. If $\lambda_0 \leq \gamma_0$, we can construct two subsets, $\Gamma' = \{\gamma \in \Gamma \mid \exists \lambda_i \in \Lambda : \gamma \geq \lambda_i\}$ and $\Lambda' = \{\lambda \in \Lambda \mid \exists \gamma_i \in \Gamma : \lambda \leq \gamma_i\}$, then our task is to remove the least number of elements from either Γ or Λ to reduce both Γ' and Λ' to $\emptyset$. If we once again order the two sets, such that the elements in Γ' are in ascending order and Λ' are in descending order, we can store our values into a stack, pushing the elements in order so the top of the stack for Γ' is the highest numbered split and for Λ' it is the lowest. Now, we only need to compare the top of each stack, and if $peek(\Gamma') \geq peek(\Lambda')$, we choose one side to pop according to some rule, and repeat until $peek(\Gamma') < peek(\Lambda')$. This approach is summarized in Algorithm 4.

Algorithm 4 Overall structure of an algorithm for deriving an interval satisfying the split points of a feature seen on paths in the forest

Let $splits$ be a set of decision point tuples, containing non-negative split value and direction (left, right) for a particular feature.

function GETINTERVAL($splits$)
    $\Gamma, \Lambda \leftarrow$ empty list
    for all $split \in splits$ **do**
        if $split.direction = right$ **then**
            add $split.value$ **to** Γ
        else
            add $split.value$ **to** Λ
        end if
    end for
    SORT(Γ, desc)
    SORT(Λ, asc)
    if $\gamma_0 \geq \lambda_0$ **then**
        RESOLVEOVERLAP(Γ, Λ)
    end if
    return (γ_0, λ_0)
end function

function RESOLVEOVERLAP(Γ, Λ)
    while $\gamma_0 \geq \lambda_0$ **do**
        $\varphi \leftarrow$ CHOOSE($\Gamma, \Lambda, paths$) // rules used here are described later
        if $\varphi = \gamma_0$ **then**
            remove γ_0 **from** Γ
        else
            remove λ_0 **from** Λ
        end if
    end while
end function

Path Aware Taking this view of our decision trees and hence random forest, as being a set of decision points made up of a feature, value and direction ($<, \geq$), it is easy to forget the context within which these decision points lie—to not see the forest for the trees (or the trees for the paths (or the paths for the steps)). A *path* is a sequence of steps (decision points) leading to a classification:

$$\Pi = \{\pi_0 = (\phi_0, x_0, \psi_0), \ldots, \pi_{n-1} = (\phi_{n-1}, x_{n-1}, \psi_{n-1}), \pi_n = (\theta)\}$$

Where ϕ_i is the feature, x_j is the split point, $\psi_k \in \{<, \geq\}$ and θ is the class. Paths have the property that **all** decision points must be satisfied to reach the classification. In fact, a path is uniquely defined by its decision points; if a condition is not met by a certain value, then there must exist another path in which that condition is met by the value. By taking this conceptual view of a decision tree, and in turn a random forest, as being a set of *paths*, it becomes clear that if we eliminate a particular split from either of our sets Λ or Γ, we have effectively eliminated *the entire path* that contains the decision point represented by that split, and we should therefore remove all the other splits on that path from our consideration in their respective Λ or Γ too. If we fail to remove these "broken" paths from our consideration, we may later make decisions to disregard other nodes on unbroken paths, due to the presence of nodes on broken paths. As we consider features one by one, slowly our trees become more "pruned", resulting in less overlap between the respective Λ and Γ sets.

As the result of parsing our random forest returns data for all paths leading to our target class, if the result of not meeting a given condition is a path that still leads to our target class, then we know that path will exist in our set of paths and so that node will still form part of our change set for making recommendations, therefore we need not be concerned. Conversely, if the result is a path that does not lead to our target classification, we know that path will not exist in our set of paths and the node will truly be excluded. We took this approach in our system, modifying Algorithm 4 so all nodes in a given path are disregarded if any one of the nodes in the path is disregarded.

One difficulty in taking this path elimination approach is deciding which feature to consider first. The order in which features are chosen can have a significant impact—if the first feature's set of conditions happens to contain a large degree of overlap, a lot of paths would be eliminated, but this may not necessarily be optimal. Calculating all possible combinations of feature ordering in this respect, to decide which is optimal would be expensive computationally. As a path is a sequence of conditional steps towards a given classification, it is natural to consider conditions in ascending order of depth. Therefore, in our implementation, we took the greedy approach of deriving intervals for all the conditions at a particular depth, d_i, before d_{i+1}. Once an optimal interval has been derived for the features found at depth d_i, these form the endpoints for the

ordered sets/stacks Γ and Λ for that feature at depths $d_j \mid j > i$, meaning any conditions at d_j with splits lower or higher than their respective endpoints are dropped automatically. This favours the conditions at lower depths, which is a reasonable assumption, as these conditions have the greatest influence on classification, and are likely to be present on multiple paths.

Tree Aware In addition to being more path aware when choosing to disregard a certain node, it is also necessary to be aware of the individual trees the paths exist within. The overall output of our ensemble classifier is determined by the number of trees that voted for each class, therefore it is beneficial to maximize the number of different trees represented in a set of recommended changes, to increase our potential votes, as each tree can vote only once on the output classification. When selecting a decision point to exclude, it is therefore advisable to select one from a tree that includes other potential paths to our target class rather than one from the only path present in a tree. We also included this heuristic in our implementation of Algorithm 4.

Cluster Aware Finally, in order for our calculated intervals to produce feature vectors that will actually classify successfully as our target, it is necessary to take a *holistic* approach. Hitherto, we have considered our features as isolated variables that can be optimized independently of one another to satisfy the maximum number of decision points, only referencing other features when deselecting entire paths. It is important to note, however, that paths are constructed during induction from real training instances, and as such, each path represents a set of features that are in synergy with one another, representing some configuration of the system we are modelling (the source code) that is feasible and *makes sense* in the rules of the system. So, for our model, each path represents a combination of elements in a source code file that can co-exist in the definition of the language syntax and produce features whose values satisfy the conditions of the path's decision points. We can extrapolate this argument further, by noting that any paths that were constructed from the same cluster (sets of training instances that can be grouped together in the feature space), if combined, must produce a set of decision points that some concrete instances can satisfy. Note that this does not mean any feature vector that can satisfy such a set is a feasible combination of real elements that could be found in a source code file. The decision points created in a decision tree represent open-ended intervals—they divide sets of training instances, but do not bound them. However, we can guarantee that it must be possible for at least k real instances to exist that satisfy the decision points, where k is equal to the cluster size, because those decision points are *based* on the same k training instances. If we take different paths leading to the same classification, but that were constructed from different clusters, we cannot just arbitrarily combine the decision points from these two paths to construct a set of recommendations, because *unless we know these paths led to the same cluster of training*

instances, we do not know if the features found can be combined to represent a source code file that is possible to exist.

As each tree is presented with a slightly different training set and selects from a random subset of features at each node during induction, we cannot guarantee that the same clusters will be present throughout our forest. This is particularly true for larger clusters, where the probability that one of the training instances was not present in the training set is higher. If two clusters are not exactly the same, we cannot guarantee that combinations of decision points derived from paths leading to either of the two clusters will be satisfiable. Previously, we noted that maximizing the number of trees increases the potential classification confidence, or number of votes, of our target class by the ensemble. Therefore, rather than combining paths for a particular cluster, it is better to combine paths for a particular *training instance*. The rationale behind this is simple: firstly, by selecting only paths leading to a particular training instance, we guarantee to find a set of decision points that are satisfiable by files that can exist. Secondly, we will also automatically find a set of decision points that are satisfiable by any other files in the same cluster. This is true whether the size of the cluster is one or 100. Algorithm 5 enhances Algorithm 3 with modifications for retrieving nodes by training instance, while Algorithms 6 and 7 are an update to Algorithm 4 with tree- and path-aware behaviour.

Limitations While the approach presented here is a technique for deriving a set of feature value ranges that are guaranteed to be satisfiable by real source code files, and hence is applicable in practice, there are certain limitations which should be noted.

The first limitation is that finding sets of conditions for a particular training instance is only possible for trees that were inducted on training sets containing that instance. As bootstrapping employs sampling with replacement to produce an alternative training set the same size as the original, any particular training instance is likely to occur in only $1-\frac{1}{e} \approx 63.2\%$ of decision trees, which means we are only able to gather information from that proportion of our forest to aid us. In reality, we are likely to accumulate some votes from trees that we did not parse, as a feature vector that has already been perturbed to resemble our target class is likely to also resemble it in some unparsed trees, if the out-of-bag error is sufficiently low, the data set exhibits a degree of clustering and the we have not suffered from overfitting. We could improve this situation further by evaluating the training instance against the decision trees inducted without it, noting when it was classified correctly, and also including these paths in our $paths$ set. This would be a suitable avenue for future work.

The second limitation, also in relation to solving for a particular training instance, is that a feature vector perturbed according to our recommendations is not guaranteed to return an optimal classification confidence. One way we could optimize our feature vector towards higher

Algorithm 5 Method for retrieving paths for a given class by training instance from a random forest

Let $forest$ be a collection of decision trees
Let $tree$ be the root of a decision tree, consisting of a feature ID ($\in \mathbb{N}$), split value ($\in \mathbb{R}$), class distribution ($\in \mathbb{R}^N$) and pointers to two child trees, left and right
Let *leafNode* be a special $tree$ with no children, which contains training instances
function GETSPLITSFORFOREST($forest$)
    $paths \leftarrow \emptyset$
    for all $tree \in forest$ **do**
        $treePaths \leftarrow$ GETSPLITSFORTREE($tree, targetClass$)
        Add $treePaths$ to $paths$
    end for
    return $paths$
end function
function GETSPLITSFORTREE($tree, targetClass$)
    $paths \leftarrow \emptyset$
    if $tree$ **is a** $leafNode$ **then**
        if most numerous class = $targetClass$ **then**
            for all $trainingInstance \in leafNode$ **do**
                Initialize $path$ structure, containing the $trainingInstance$ and an empty tuple
                Add $path$ to $paths$
            end for
        end if
    else
        for all $child \in children$ **do**
            $childPaths \leftarrow$ GETSPLITSFORTREE($child, targetClass$)
            for all $childPath \in childPaths$ **do**
                // Direction is determined by checking if $child$ is the left or right child
                Add $(feature, split, direction)$ to start of tuple contained in $childPath$
            end for
            if $childPaths \neq \emptyset$ **then**
                Add $childPaths$ to $paths$
            end if
        end for
    end if
    return $paths$
end function

Algorithm 6 Method for deriving intervals for a set of paths

Let $paths$ be a set of $path$ items from a random forest for a specific training instance.
Let $path$ be a sequence of $conditions$ in a decision tree
Let $condition$ be the tuple $(tree, path, feature, split, direction)$, where $tree \in \mathbb{N}$, $path \in \mathbb{N}$, $feature \in \mathbb{N}$, $split \in \mathbb{R}$ and $direction \in \{left, right\}$.

function GETINTERVALS($paths$)
    $intervals \leftarrow$ empty hashtable
    for $j \in \{0, 1, \dots\}$ **do**
        // Build a hashtable of conditions found at depth j in each path, indexed by feature
        $conditions_j \leftarrow$ BUILDFEATURESPLITS($paths, j$)
        if $conditions_j$ **is empty then**
            break // Max depth reached
        else
            for all $conditionsEntry \in conditions_j.entries$ **do**
                if $conditionsEntry.key \in intervals.keys$ **then**
                    $interval \leftarrow intervals.get(conditionsEntry.key)$
                    $featureConditions \leftarrow conditionsEntry.value$
                    set $featureConditions$ endpoints to min and max from interval
                else
                    set $featureConditions$ endpoints to 0 and ∞
                end if
                // Defined in Algorithm 4
                $interval \leftarrow$ GETINTERVAL($featureConditions$, $paths$)
                $intervals.put(conditionsEntry.key, interval)$
            end for
        end if
    end for
    return $intervals$
end function

Algorithm 7 Method for deriving an interval satisfying the split points, with tree- and path-aware modifications

Let $conditions$ be a set of $condition$ tuples $(tree, path, feature, split, direction)$, where $tree \in \mathbb{N}$, $path \in \mathbb{N}$, $feature \in \mathbb{N}$, $split \in \mathbb{R}$ and $direction \in \{left, right\}$.
Let $paths$ be a set of $path$ items from a random forest for a training instance τ.

function GETINTERVAL($conditions, paths$)
    $\Gamma, \Lambda \leftarrow$ empty list
    for all $condition \in conditions$ **do**
        // It is assumed conditions with left **and** right directions exist twice in $conditions$
        if $condition.direction = right$ **then**
            add $condition$ **to** Γ
        else
            add $condition$ **to** Λ
        end if
    end for
    SORT(Γ, desc)
    SORT(Λ, asc)
    if $\gamma_0 \geq \lambda_0$ **then**
        RESOLVEOVERLAP($\Gamma, \Lambda, paths$)
    end if
    return (γ_0, λ_0)
end function

function RESOLVEOVERLAP($\Gamma, \Lambda, paths$)
    while $\gamma_0 \geq \lambda_0$ **do**
        $\varphi \leftarrow$ CHOOSE($\Gamma, \Lambda, paths$)
        if $\varphi = \gamma_0$ **then**
            remove γ_0 **from** Γ
        else
            remove λ_0 **from** Λ
        end if
        remove $path$ **from** $paths$
    end while
end function

Algorithm 7 continued

```
function CHOOSE(Γ, Λ, paths)
    numPaths_γ ← GETNUMPATHSINTREE(γ_0.tree, paths)
    numPaths_λ ← GETNUMPATHSINTREE(λ_0.tree, paths)
    if numPaths_γ = 1 ∧ numPaths_λ > 1 then
        return λ_0
    else if numPaths_λ = 1 ∧ numPaths_γ > 1 then
        return γ_0
    else
        // Check which side has the greatest gap to its successor
        if γ_1 − γ_0 > λ_0 − λ_1 then
            return γ_0
        else if γ_1 − γ_0 < λ_0 − λ_1 then
            return λ_0
        else
            return Choose randomly ∈ {γ_0, λ_0}
        end if
    end if
end function

function GETNUMPATHSINTREE(treeId, paths)
    numPaths ← 0
    for all path ∈ paths do
        if path.tree = treeId then
            numPaths++
        end if
    end for
    return numPaths
end function
```

confidence is with some form of metaheuristic algorithm, such as simulated annealing, memetic search, hill climbing or genetic algorithms. These all feature randomly initialized populations of potential solutions and some form of objective function determining "fitness" of a solution, with fitter solutions propagating into later iterations of the algorithm. In our case, the objective function would, of course, be an evaluation by the random forest, returning the confidence value of our target as the primary measure of fitness (and possibly the confidence in our user as an inverse secondary measure). The open question in this case would be whether discovered solutions can exist in terms of the original source code. Furthermore, is it the case that a solution reaching high confidence must represent a feasible solution; i.e., would the random forest encode in its multitude of nodes a definition of what is possible to exist? By extension, could we assume that such solutions are feasible and safely present them to our users? Another potential issue with this form of search through optimization is it does not naturally return intervals, but rather discrete points in the feature space. Deriving a set of intervals to use for making recommendations from a set of discrete points in our feature space would be a difficult problem, possibly reducing to our original problem, albeit with clearer clustering.

5.6.4 Presenting to the User

Once we have a set of intervals, grouped by training instance, that can be used to achieve a certain classification, our next task is to derive concrete recommendations for the user. Initially, we must decide which training instance we want to use the set of intervals for. We decided to base this decision on the degree of change required, primarily how many features to change and secondarily how many edits within each change. For features based on relative frequencies and arithmetic means, the difference between the current and recommended values is first calculated, then this is multiplied by the denominator value that was used to derive the frequency or mean. If the current value is 0, the recommended value can be inverted and rounded to the nearest integer to give an approximation of the number of changes; e.g., to change the frequency from 0 to 0.5, at least 2 changes would need to be made. For Boolean-valued feature changes, the number of edits is counted as one. We also chose to use the closest endpoint of the recommended interval when presenting the suggested changes to the user, although they are also advised where the opposite extremum lies to avoid over-editing. If the user edits their source file, the degree of change metric will change for each of the target instance intervals. It is possible that these edits may result in a different set of target intervals returning a lower number for the degree of change, in which case all the suggested feature values would be updated. Further edits may cause the target to change repeatedly, particularly if the distances between the user's feature vector and the targets were high—a consequence that would be rather disconcerting for the user. In order to maintain consistency in our recommendations and avoid this "moving target" problem, the plugin records

the chosen training instance and its intervals, so this selection process is only necessary the first time the user's file is evaluated.

Lists of recommended changes are presented to the user and grouped according to their relationship with other features. In the case of node type unigrams, they were grouped with their siblings according to the extended interface they shared. Features that were already within their recommended intervals were given, along with features that did not appear in any trees and paths for that training instance. This is because knowing these values can be incredibly useful when planning edits to a file, as it informs the user which features can safely be added to or removed in order to satisfy a related feature's recommendation. Recommendations that are not currently met by their respective features are marked for clarity.

Each recommended change is formatted according to a template, indicating the potential influence of the change (according to how many trees and paths it was seen on in the forest), the direction (increase/decrease), and magnitude. Additionally, the feature's name is mapped to a more descriptive term to aid the user determine what aspect of their file is required to be changed. The next section provides details on using the plugin with screenshots illustrating the aspects described here.

5.7 Using the Plugin

In this section, we will describe the plugin and its workflow from the user's perspective. There are three main functional components to the system that are accessible to the user: training a model, evaluating one or more files and making recommendations, and saving/loading trained models. These actions are presented as commands in a menu accessible from the main toolbar in the Eclipse editor (see Figure 5.2).

Setup Before using the plugin, the user should have a corpus of publicly available source code they have authored that is attributable to them, and a second corpus of source code that has not yet been published, for which they wish to disguise their authorship. In this case, their public source code will form part of the training data and be combined with the background data included with the plugin.

Training After selecting this option, the user is presented with a resource selection dialog (see Figure 5.3), which they can use to select either individual files or entire folders and projects that are present in the Eclipse workspace (collectively known in Eclipse as *resources*). The selected

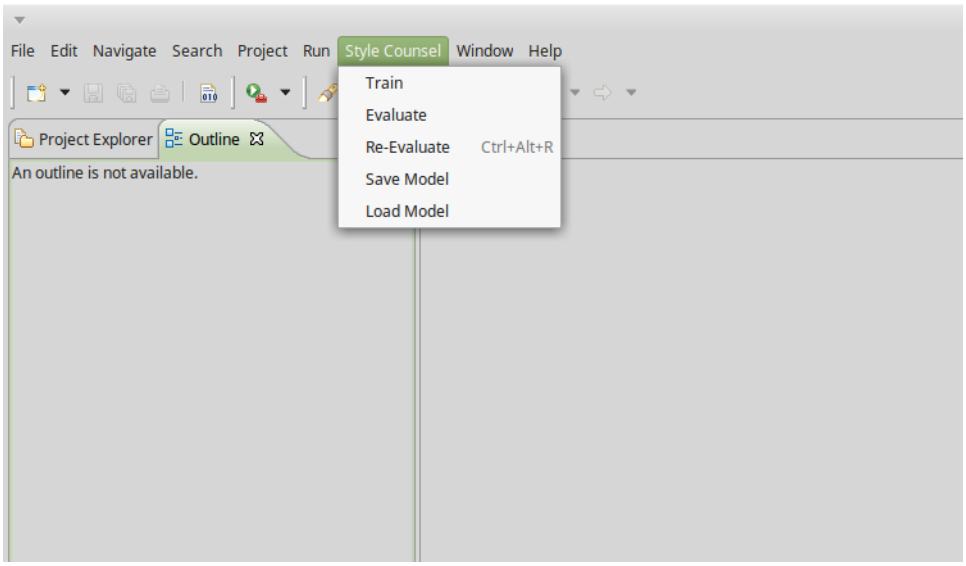

Figure 5.2: Plugin menu

resources can include files that are not source code files, as the plugin filters out files that do not contain C code. When training a model, the user should select resources that do not include the file(s) they wish to modify. Once the plugin has the list of source code files, it proceeds to extract the features described in Section 5.4 to produce a feature vector for each file, informing the user once this process is complete. These feature vectors are combined with the background training data and used to train a random forest classifier from Weka by calling the embedded weka.jar file, which produces a model.

Saving and Loading To enable users to work across multiple sessions, being able to save the current model and load it at a later time is vital. Due to the inherent randomness in the random forest algorithm, there can be a fair degree of variation between models trained on the same data, causing significant extra work for the user if they must generate a new model in each session. This would be compounded by the plugin potentially choosing a different training instance's set of conditions each time, according to the degree of change metric discussed in Section 5.6.4. Upon selecting the save action, the user is asked to choose a destination and the model is serialized to that location and given a standard name ("stylecounsel.model"). When loading, the model file is also assumed to have this standard name.

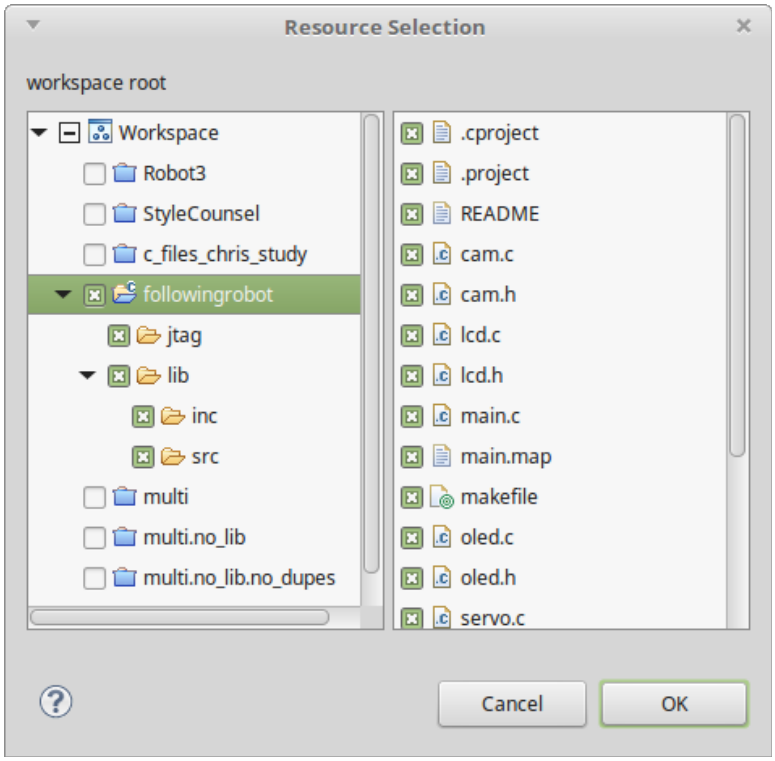

Figure 5.3: Resource selection dialog for training

Evaluation Upon choosing this command, the user is once again presented with a resource selection dialog (see Figure 5.4) from which they can select one or more resources they wish to evaluate and generate recommendations for. If no trained model exists at this point, the command will exit without taking any action. Assuming a model does exist, the plugin extracts features for the files being evaluated, classifies them using the trained model and outputs messages indicating the classification and confidence of each file (see Figure 5.5), as well as an aggregate classification/confidence value for the set of files (see Figure 5.6), if more than one file was selected. Following this, the recommendations are generated by initially computing all paths in the forest leading to each training instance of the target class (defined *a priori*). These paths are processed according to the algorithms presented in Section 5.6, the differences are calculated and finally recommendations are generated and placed into template messages as described in Section 5.6.4. The recommendations are given as warnings in the "problems" view in the workspace (see Figures 5.7, 5.8 and 5.9).

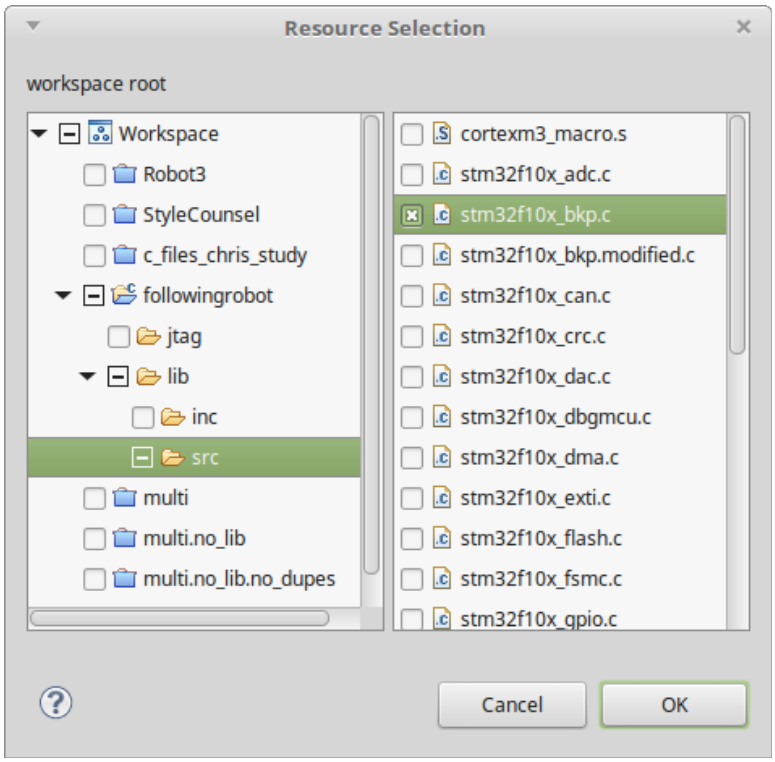

Figure 5.4: Resource selection dialog for evaluation

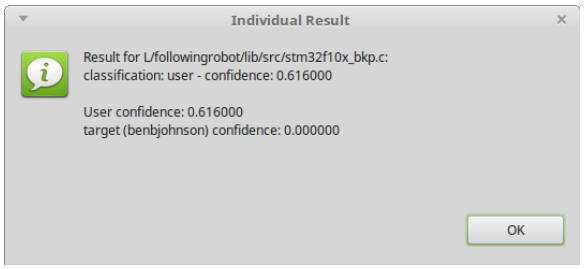

Figure 5.5: Individual file output

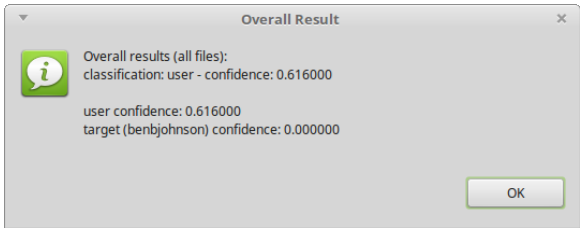

Figure 5.6: Aggregate output

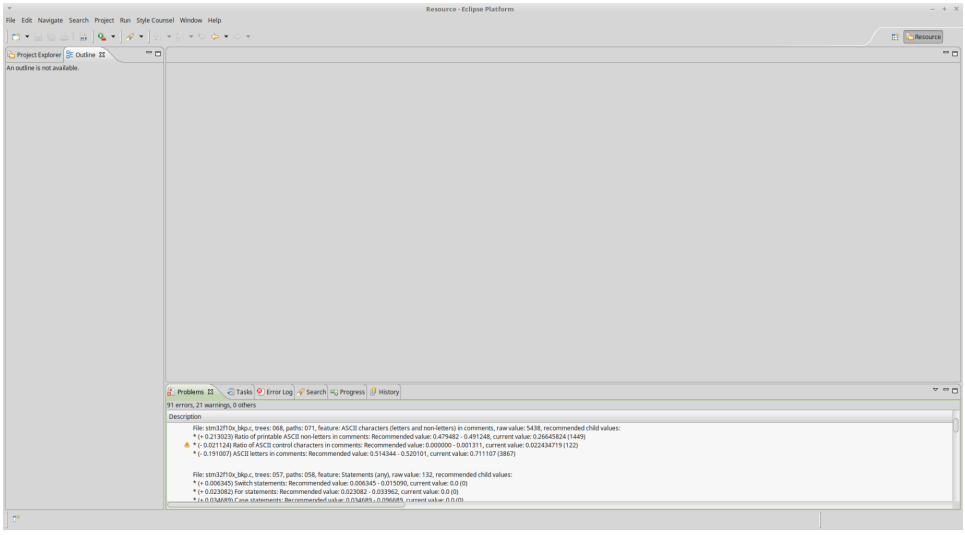

Figure 5.7: Overall recommendations view

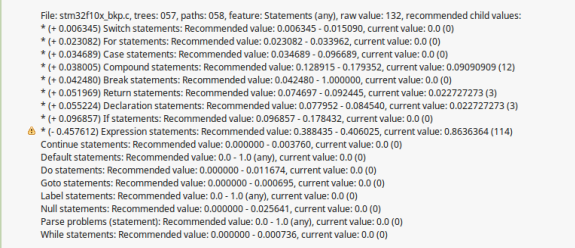

Figure 5.8: Sample recommendations—statements

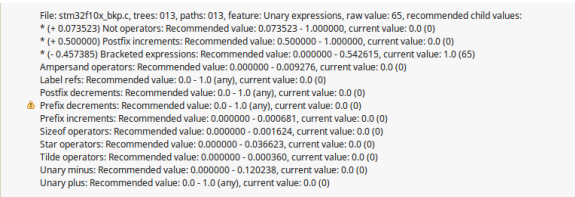

Figure 5.9: Sample recommendations—unary expressions

5.8 Pilot User Study

In order to help assess the usability and feasibility of our plugin, we conducted a small pilot user study. It was felt receiving feedback from real users would be invaluable in developing an effective tool, particularly as there is no way to automate the evaluation of these aspects of the system. We can automate the evaluation of our feature set and recommendation extraction algorithm, but not its usability.

5.8.1 Study Details

For our pilot study, we chose participants that had C programming experience and a corpus of source code files they had authored. Three members of the CrySP (Cryptography, Security, and Privacy) lab at the University of Waterloo who satisfied these criteria volunteered for the study. Each participant was given two tasks; the first was to manually analyze another author's source code with the aim of identifying elements of their style and reproducing those elements in one of the participant's own files. The second task was to use our plugin to achieve the same goal, with a different author so as not to confer an advantage by already having carried out the first task. The

tasks were chosen in this order so that completion of the assisted task would not provide the user with insights into the feature set for the unassisted task. Our Office of Research Ethics approved our study (reference number ORE#22378). We present the results of this study in Chapter 6, next.

Chapter 6

Results

In this chapter, we review the results from the series of evaluations we made against the various aspects of our system. These evaluations are described in Sections 5.5 and 5.8.

6.1 Conducting Source Code Authorship Attribution at the Internet Scale

Our first evaluation was to empirically test our feature set and random forest combination against the entire corpus we obtained, as described in Section 5.3. This evaluation was conducted on an 80-core, 2 TB server in the CrySP RIPPLE Facility,[1] of which 512 GB was used per test run. We initially evaluated with two different feature sets: one containing character frequencies in comments and string literals, and one without. These features and our reasons for excluding them from the final system were previously discussed in Section 5.4.4. We include the results of evaluating with and without them to assess the cost of removing them compared to the benefits. Figure 6.1 gives the results of evaluating for individual files, while Figure 6.2 gives the repository-level results.

Using the hold-one-out methodology required us to train a new classifier for each repository, as the background and evaluation data sets changed with each evaluation. In fact, our methodology involved more than just hold-one-out (for evaluation), as it entailed, for each repository being held out, training on just one of the author's repositories at a time, so the hold-one-out was also enacted on the training data as well as the test data. For authors that had exactly two

[1] https://ripple.uwaterloo.ca/

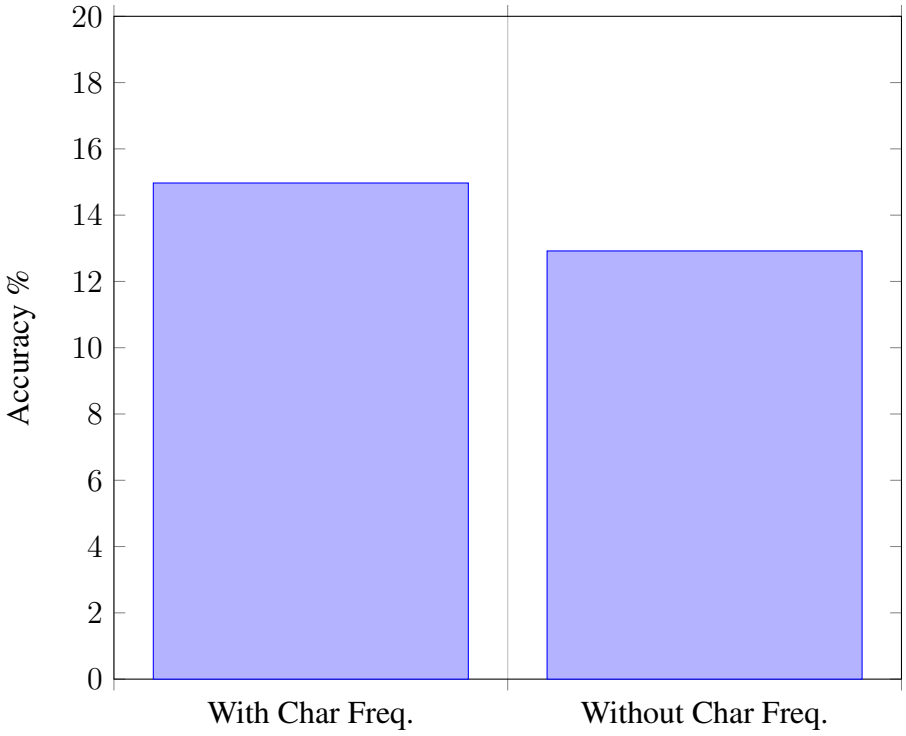

Figure 6.1: Overall accuracy of identifying the author of each file in the complete data corpus using feature sets with character frequencies and without

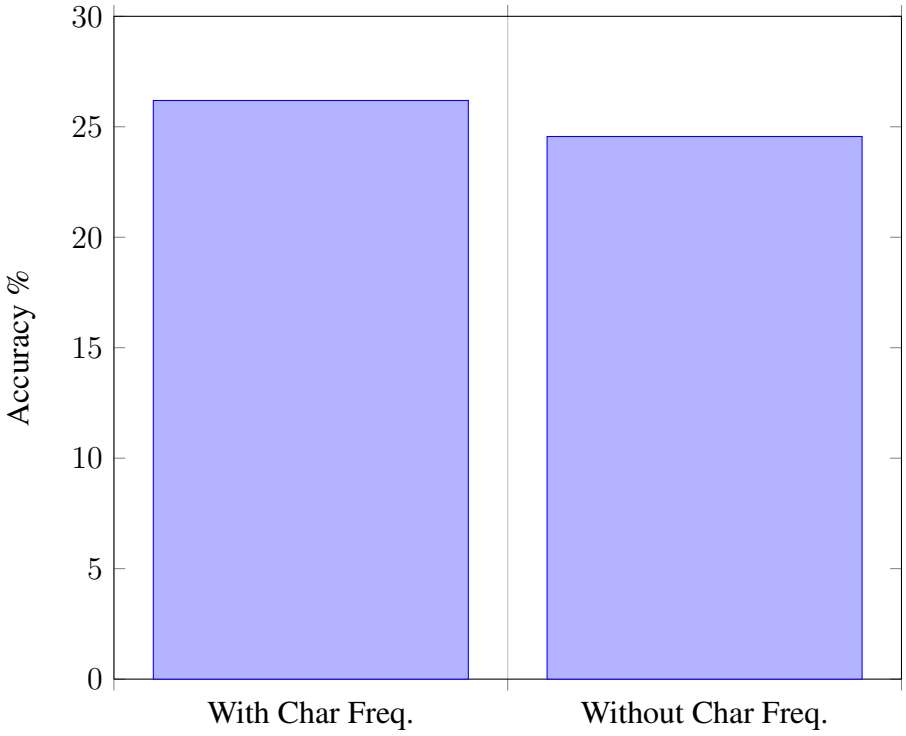

Figure 6.2: Overall accuracy of identifying the author of each repository in the complete data corpus using feature sets with character frequencies and without

repositories this had no effect, but each repository for authors with more than two was evaluated multiple times, $n(n-1)$ for n author repositories to be exact. The reason for this was because we wanted to reduce the effect of one copied repository "misleading" the learning algorithm and having a negative effect on the outcomes of the author's true repositories. Therefore in total, we ran our experiment with 1,261 repositories, but performed 2268 evaluations, which took a total of 36:17 hours, averaging 57.59 seconds per repository evaluation.

Our initial test run with 1,261 repositories and 525 author classes returned a successful classification for 14.97% of files when using the feature set that included character frequencies and 12.92% without these features. For the repository level, this translates to 26.19% and 24.56%, respectively. The difference excluding the character frequencies makes is fairly small, but not trivial. Looking at Figure 6.3, which breaks the success/failure down by how many repositories were completely misclassified (all fail), failed but with some successful, successful but with some failures, and completely successful (all correctly classified), the main difference from removing the character frequencies is in the marginal categories of partial success/fail, while the total fail and total success categories are largely unaffected. While removing character frequency features clearly has a negative overall effect on classification ability, we felt the difference was not great enough to justify including such low-level features that would certainly impact the usability of the tool and render advice based on these features highly impractical.

While the accuracy of our classifier and feature set may not be very high (although far higher than a random guess, which would be less than 0.2%), we would like to stress that we do not rely on typographical features, such as indentation, or presence of word unigrams, which capture specific variable, function and macro names. Such features are known to be highly influential in performing classification, but the resulting models have been demonstrated to be vulnerable to even trivial attacks [SZK18]. Also, as discussed in Section 5.4, word unigrams require an analysis of the entire corpus; however, in our case there are practical limitations to including the entire corpus within the plugin at training time, and the user's own files, which are the most crucial to accurately represent in the training data, cannot be known in advance. A small number of our features are based on the apparent naming *scheme* employed, which we argue is more representative of overall style than matching specific names, and is harder for an adversary to modify—specific names and word unigram features can be modified with a single character edit. We do not use any typographical features in our system either, but do invoke Eclipse's built-in code formatter to provide protection against weaker attribution systems, without including the effect of this type of modification in our results. Furthermore, we employ a modified hold-one-out methodology at the repository level, which is a considerably more strenuous test of our feature set and classifier than cross-validation. Performing cross-validation would mix data from all of a user's repositories in both training and test sets, resulting in similarities that are influenced more by the purpose of the repository than the author's style. Finally, we are more concerned

with exploring the challenges faced by those attempting to avoid stylometric identification, and demonstrating general techniques for achieving this, rather than performing attribution itself. For future work, it would be useful to reduce our feature set using information gain or principal components analysis, and evaluate our proposed method on established attribution systems to compare its effectiveness and generalizability with other feature sets.

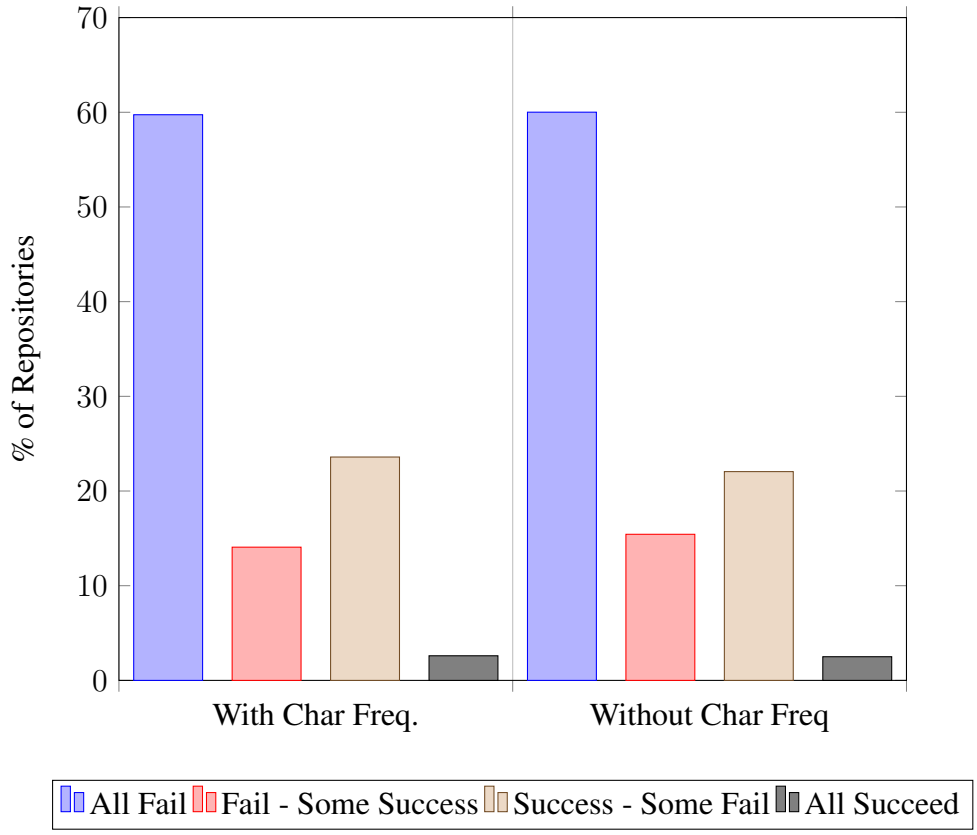

Figure 6.3: Results of identifying the author of each repository in the complete data corpus using feature sets with character frequencies and without

Regarding the effect of noise, it is clear from looking at Figure 6.3 that the majority of repositories in our dataset were completely misclassified; i.e., every single file. We decided to investigate further, and by manually inspecting a random sample of 25 of these repositories, we found that only two of the 25 were definitely the work of a single author, with a further one that was probably single author. The remaining 22 were either multi-authored works, or different single authors between the training and test repository, or contained third-party library code that had not successfully been removed by our filters. These results are summarized in Table 6.1,

which provides the author name, the training and test repository and the result of the manual inspection. The result falls into the following categories:

- Single-Author—Only one (or no) author's name appears in comments, README files or other artifacts. No evidence of multiple author involvement or someone else's work.
- Multi-Author—Either multiple author names appear in the code comments or README files, or there is some other evidence of multiple-author involvement, e.g. abhishek-Shukla's System-Call-Inherit repository included a PDF file that appeared to be an academic paper indicating three authors were involved, or patback66's Team-254B repository that linked to a team entry into an academic robotics competition.
- Different Authors—Either two single-author repositories clearly written by two different people, or two multi-author repositories written by different groups of authors, or some combination of these.
- 3rd-Party Lib Code—Indicates the presence of library code that had not been successfully removed by our filters, typically because the directory containing the code was not called "lib" or "ext", such as "libavl" in matthewjmiller/mkavl.
- Ported—Indicates the presence of code that has been ported from elsewhere, either to a different hardware platform or programming language. Ported code is typically heavily influenced by the original code's style and behaviour, if not directly copied with the minimal changes required to effect the port.
- Generated—Code that has been automatically generated by some tool was present in the repository.
- Copied—Entire repository was written by a different author.

These categories represent different degrees of noise in our dataset, some worse than others. Clearly, if two repositories owned by the same user have been written by two completely different authors (group or single), then any similarity in style will be purely coincidental. This type of noise can seriously mislead a learning algorithm leading to poor performance if present in significant numbers. Third-party library code is typically less of a problem, depending on what proportion of the total files are from the third-party. Ported code is usually heavily influenced by the original code and can also represent significant noise, depending again on how the port was carried out and the proportion of directly ported code to novel code written by the porting author. It also depends, of course, on whether the author writing the port is the same as the original

Table 6.1: Result of investigation into random 25 repositories in "all fail" category

Author	**Training Repo**	**Test Repo**	**Result**
abhishekShukla	System-Call-Inherit	Linux-Stackable-File-System	Multi-Author
bruceg	barch	journald	Single-Author
BuckRogers1965	DataObj	SDLtut	3rd-Party Lib Code (DataObj)
dougszumski	nRF24L01	docLamp	Multi-Author, Ported
emmadoraruth	MDL	Boggle	Multi-Author, Generated
jawnsy	Alien-Libjio	libdebctrl	Different Authors
jimwise	cempire	shared	Multi-Author, Different Authors
jmesmon	vex-cortex	cadaver	Different Authors
Kanma	sip	zziplib	Multi-Author, Different Author, 3rd-Party Lib Code
Ludo6431	DSerial-firmware	iptk	Multi-Author, Different Authors
lumag	mmtrace	emv-tools	Probable Multi-Author (mmtrace)
matteobertozzi	GDEngine	MingChenSlot	Different Authors
matthewjmiller	C-Interface-Generator	mkavl	3rd-Party Lib Code
msantos	rst	sods	Single-Author
noname22	makeadf	megadrive-gcc	Multi-Author
patback66	Team-254B	CS-49C	Multi-Author (Team-254B)
robotang	academic	player_plugins	Multi-Author (player_plugins)
rofl0r	gnuboy	hexedit0r	Multi-Author, Different Authors
siddesh	tinycdb	cdb	Different Authors
svk	lib1tquery	ritual-scheme	3rd-Party Lib Code, Copied (ritual-scheme)
tpenguin	solarisvoip-asterisk-zaptel	solarisvoip-asterisk-addons	Multi-Author, Different Authors
trasz	ofx	libsmb2	Probably Single-Author
troglobit	gul	advent4	Multi-Author (gul)
XVilka	bvim	2ndboot-ng	Multi-Author, Different Authors
yaoweibin	libcharguess	mod_tcache	Copied (libcharguess)

author, in which case it should not be counted as noise. Generated code has been produced by an automated tool and not a human. Typically, the code is not particularly readable, with minimal comments, except for a header indicating the tool that generated it.

The results of this manual inspection indicate a high degree of noise in the repositories that completely failed to be classified correctly. Dauber *et al.* [DCHG17] found a 15% degree of corruption in the training set resulted in a greater than 15% reduction in accuracy; in our case, we found an 88% degree of corruption. Considering this, we re-ran the evaluation twice more, once including all the repositories that classified at least one file correctly, and once including only the repositories that had a plurality of correctly classified files. This is summarized in Figures 6.4 and 6.5. In total, we ran our experiment with 567 and 392 repositories, performing 1096 and 722 evaluations, which took a total of 7:25, and 3:56 hours, averaging 24.36 and 19.61 seconds per repository evaluation, respectively. This time we were able to successfully identify the owner in 28.64% and 43.02% of files, which translated to a success rate when aggregated over the entire repositories of 49.27% and 66.76%, respectively.

6.2 Extracting a Class of Feature Vectors That Can Systematically Effect a Classification as Any Given Target

Our second evaluation was to test the recommendation algorithm described in Section 5.6. The purpose of this evaluation is to demonstrate that the recommendation algorithm produces correct recommendations, in terms of eliciting a misclassification as a target author. We also wish to show that features not included in the recommendations do not contribute to the overall classification for that target, and can safely be ignored. These tests were all conducted on an Intel® Core™i7 home PC with six cores and 8 GB RAM.

We carried out this evaluation by extracting the feature value intervals for each file in our corpus, as though the author of that file were the *target*, and the training instance representing the file had been chosen as the basis for the recommendations. Min, max and mid values from these intervals are used to perturb that file's existing feature vector and generate sparse feature vectors. If a feature vector is said to be perturbed, it means the features that were not part of the recommendations were left as their original values, sparse means they were set to 0. For example, if a particular feature in the targeted instance had a value of 0.44 and the extracted interval indicated a perturbation in the range of $[0.4, 0.5)$ was possible, we evaluate the feature vector with this value set to 0.4, 0.45 and $0.5 - \epsilon$, where ϵ is some suitably small value. If the perturbed and sparse feature vectors return similar confidence labels, then we can say the recommendations were only for the features that actually contribute to the classification. This

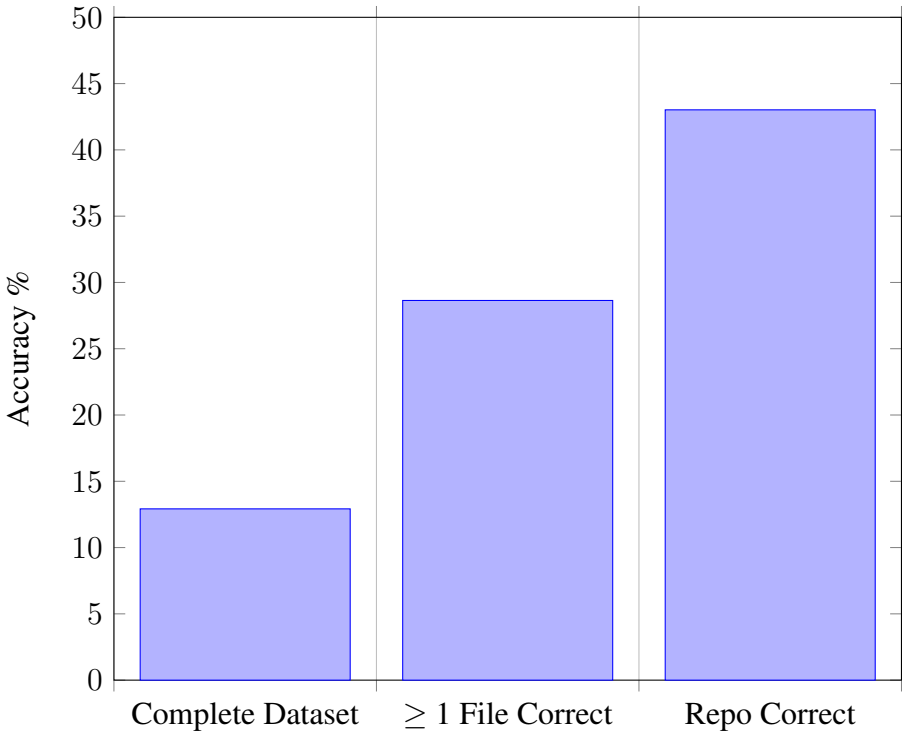

Figure 6.4: Overall accuracy of identifying the author of each file in the complete dataset, the dataset limited to at least one file correct and the dataset limited to the entire repository correctly classified.

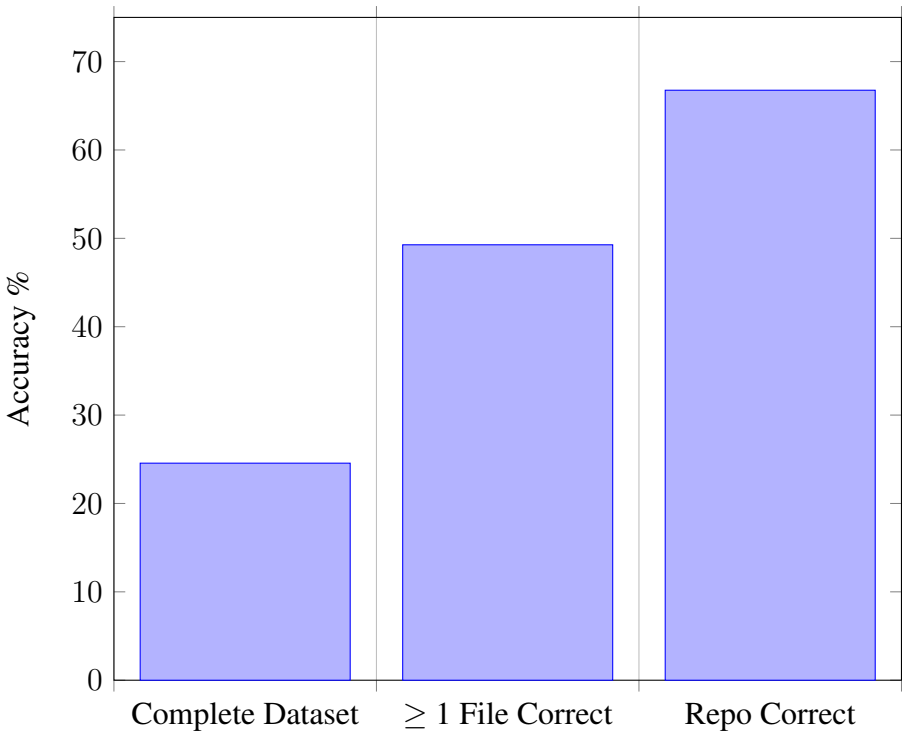

Figure 6.5: Overall accuracy of identifying the author of each repository in the complete dataset, the dataset limited to at least one file correct and the dataset limited to the entire repository correctly classified.

is a desirable characteristic for our system as we want to minimize the changes we ask users to make. We then evaluated these feature vectors with both the random forest they were derived from and a second random forest trained on the same data. The results of these evaluations are given in Figure 6.6. The confidence returned for the training instances used as the target are averaged across both random forests, as the differences are negligible, while the confidence for the sparse and perturbed feature vectors are averaged separately over the respective forests, as their differences are more significant. We report the *confidences*, rather than accuracy, because this more accurately reflects the closeness with which we are able to imitate the target, but note that with the relatively large number of classes in our corpus, any class receiving a confidence (i.e., votes) greater than 2% will typically become the overall label attached to that instance; a class receiving a confidence greater than 50% is guaranteed to become the overall class label. Therefore, the confidence in our context is always strictly less than the accuracy.

We tested our algorithm with varying numbers of decision trees to see the effect this had on confidence. It is clear that as the number of trees increases, the classification confidence of the sparse and perturbed feature vectors on the second random forest also increases. The performance with the same feature vectors and the original random forest increases only marginally, and for the target training instance, the confidence remains almost constant. This increase is explained by the additional information provided by larger forests, as more combinations of features are compared in individual nodes at differing depths and with subtly varied training data, so the recommendations become more robust and more likely to still give the desired result even with different classifiers. The perturbed vectors give significantly better confidences in the smallest forests due to the compensating effects of defaulting to the original training instance's values rather than 0. The additional information available in larger forests all but cancels this effect out with more than 100 trees, however. This additional depth comes at the cost of more recommendations for the user to implement, which is a tradeoff that could be configured according to each user's preference.

Comparing the performance of the perturbed and sparse vectors, we can see there is very little difference, which demonstrates that the subset of features used in deriving the intervals, and hence the recommendations to the user, are the only features contributing to the classification. The differences between the target and derived feature vectors' performances on the original random forest are a result of the relaxing of the feature vector values from the original single point to a *volume* of the feature space encompassing a much greater number of potential feature vectors, each of which can expect to elicit a similar classification confidence from the random forest in question as its peers. Having a target volume to guide users toward instead of a single point in the feature space is far more flexible, providing our users with more options when it comes to deciding how to implement the suggestions offered by the tool, improving its usability. This increase in flexibility and usability comes at a cost of lower overall classification confidence,

however.

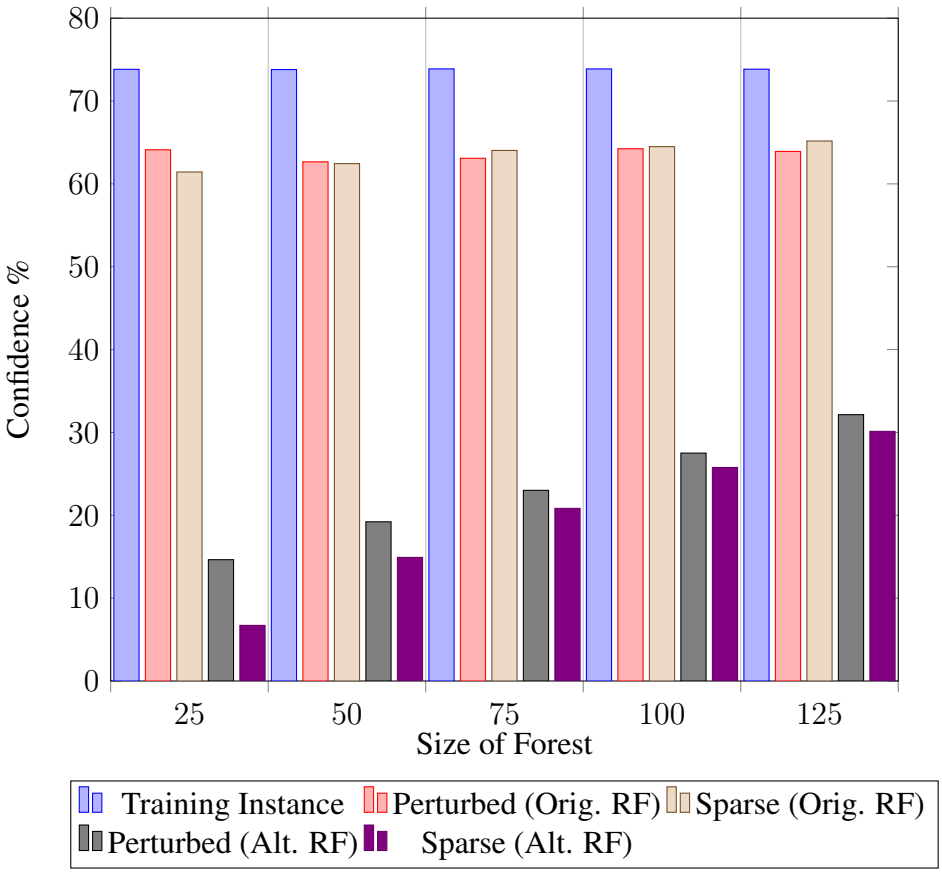

Figure 6.6: Results of evaluating sparse and perturbed feature vectors generated from extracted intervals used for making recommendations. The fabricated feature vectors were evaluated by the random forest that produced them and a new random forest trained on the same data. The confidences reported here indicate how successfully the extracted intervals model the volume of feature space occupied by the training instance, with higher confidences being more successful.

As previously noted, the set of intervals extracted for each training instance encountered while traversing the random forest was sampled three times during evaluation: once using the minimum values, one using the middle values and one using the maximum values in the interval. The output of these three samples are averaged in Figure 6.6 for clarity, but note that there are minor differences in their performance. Figure 6.7 gives these differences for the 100 tree forest, which is the default size used by Weka. For evaluations on the original forest, the feature vectors using the maximum values from the intervals gave the best average performance. The exact

reasons for this are unknown, but it would certainly merit further investigation. The results for the new forest are expected, with the mid-range values providing the best performance. This is because values at the fringes of the derived intervals are more likely to fall on the wrong side of a split when a new forest is inducted on a different bootstrapped dataset. There is a certain amount of *fuzziness* in the positions of boundaries for the same feature between two independent forests, therefore it follows that values falling well within these boundaries, rather than near the end points, would be more resistant to such uncertainty.

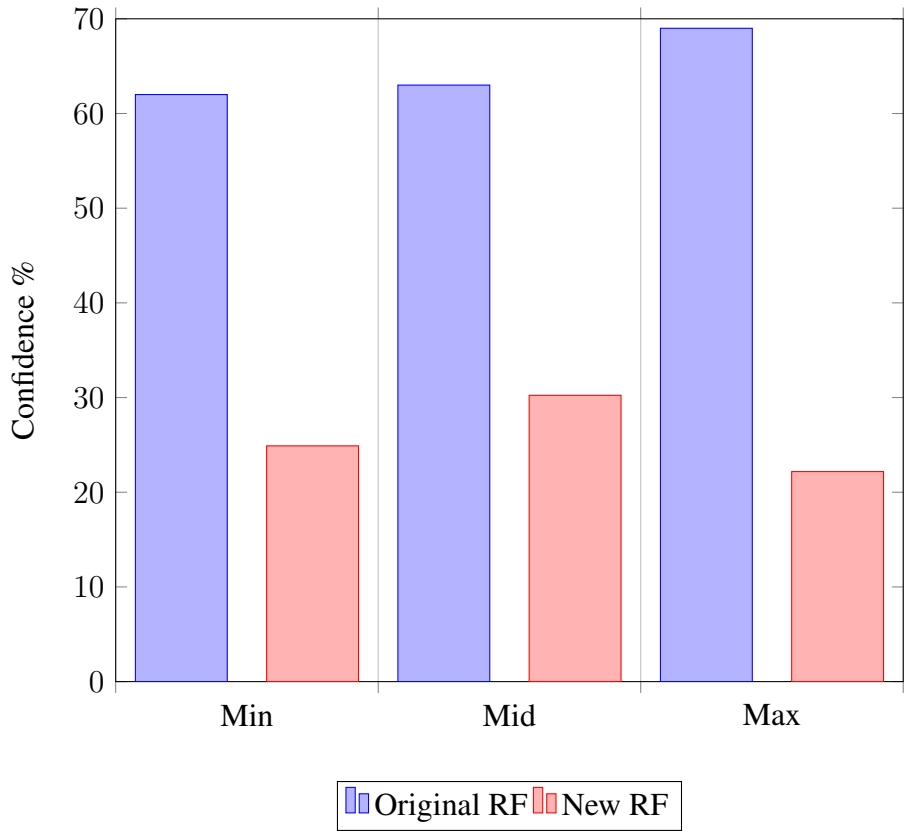

Figure 6.7: Results of evaluating sparse feature vectors against a 100-tree forest generated from the min, mid and max values of the derived intervals.

6.3 Pilot User Study

Our final evaluation was to elicit some valuable initial feedback on the usability aspects of our plugin, and how it compared in the participants' perspectives to a manual attempt at the same feat. This evaluation was of a more qualitative nature than our previous assessments, consisting of far fewer samples, and largely based on feedback responses to a questionnaire, which is provided in Appendix A. Still, quantitative results were also obtained as part of this process, and these are discussed alongside the participants' responses below.

We refer to our three participants as P1, P2 and P3 respectively. Each participant provided a number of C source files they had written and were tasked with a manual attempt at mimicking another programmer's style (Target X) after being given access to a selection of their source files and an assisted mimicry attempt of a different programmer (Target Y) using our tool. Upon completion, the participants were asked to complete and return a short questionnaire asking about their experiences using the plugin and comparing it to their manual attempt, as well as how they thought the plugin could be improved. Programmers P1 and P2 returned their responses, however Programmer P3 chose not to complete the questionnaire. Each participant used the same eight-core AMD desktop PC with 16 GB RAM to carry out their tasks. The workflow during the assisted attempt was identical to the workflow a real user would follow when using the plugin. This involved initially training the classifier on the participant's provided source code plus the background dataset, then for the file(s) they wish to modify as part of the task, performing an evaluation and following the recommendations as described in Sections 5.6.4 and 5.7 until a desired classification confidence was reached, or the time limit of one hour expired. Note that in a real user session, ideally the classifier would be trained on completely independent repositories of code to the one being modified, preferably public repositories, if any exist. In our limited user study, the files provided by the participants did not constitute entire repositories, nor where they in great enough numbers to split into separate training/test sets, therefore all files were used for training. Note that this has no bearing on the outcome of the study, as the recommendations are based on the target programmer, whose code is part of the background dataset. The user's files are used to calculate the differences between their current values and the end points of each interval, regardless of whether that file's data formed part of the background dataset or not. Indeed, including all the user's files in the training data makes the task harder, as the initial classification confidence will be higher than if it were not included.

6.3.1 Results

Programmer P1 furnished us with 22 files, with an average size of 1.45 KB. After training, their files were classified with an average confidence of 71.95%. They selected two files for modi-

fying in the first task, one of which was 4.1 KB and the other 4.6 KB. These files were initially classified as them with a confidence of 66% and 67%, respectively and the target with 0% (i.e., that percentage of trees output those predictions). For task two, they modified the same 4.6 KB file. After their manual mimicry attempt of Target X, Programmer P1's classification confidence had been reduced to 64% for the first file and 30% for the second, while the target's was still 0% for both. Moving on to the assisted task with Target Y, upon completion Programmer P1's classification confidence had been reduced to 6%, while that of the target was still 0%.

Programmer P2 provided ten files, with an average size of 20.06 KB. After training, their files were classified with an average confidence of 63.3%. They selected one file for modifying in both tasks, with a size of 14.59 KB, that was classified as them with a confidence of 65% and the target with 0%. Their manual attempt with Target X resulted in a reduction in confidence to 31% and 0% as the target. The assisted attempt resulted in 5% as themselves and 2% as Target Y.

Programmer P3 provided 32 files, with an average size of 14.7 KB and classification confidence of 74.78%. The file they selected for both tasks was 4.5 KB in size and had a classification confidence of 66%, with the target 0%. Upon completion of the first task, they managed to reduce this to just 4% (0% Target X), while the second task saw a reduction to 11% for them and 1% as Target Y.

6.3.2 Experiences with Manual Task

P1 reported that they found the manual task easy, while P2 thought it only *"seemed easy at first"*, but they were *"only able to find distinguishing features that were small in scope"* and were unsure if these were actually useful in identifying authors. On reflection, they stated that they probably *"didn't imitate them as well as I originally thought"*.

Some of the observations made by P1 were that the target often used `static` functions, `do/while` instead of `while` loops, `goto` statements, nested `if/else` conditionals rather than compound structures and extensive macro definitions. P2 picked out the choice of error codes returned by functions and use of whitespace as distinctive aspects. In Section 5.4, we outlined our reasons for *not* including whitespace or any typographical-based features in our feature set. This was also communicated to our participants; however, it is easy to forget such details. By automatically formatting users' code with Eclipse's built-in formatter, despite these features not forming part of our feature set, all users are able to benefit from this basic protection without having to think about such minutiae. P1 was able to adapt their code to the differences they noticed to varying degrees, although they reported that they believed they were only *"moderately successful"* in achieving the aims, citing the time restriction and the challenge in *"identifying the*

changes to be made" as the main limitations. Expanding on this last statement, they described the task of *"*[determining] *the differences between my own style of writing and the target user's"* as difficult, because *"stylistic aspects of the target user could be present in a multitude of different files and the target user may use features of the language on an ad-hoc basis"*. P2 made a similar observation, noting that *"it was challenging to find distinctive features for a programmer that spanned multiple files"*.

Overall, the manual task gave the participants complete freedom in how they chose to interpret the target's style and adapt their own files to mimic this style, leading to changes that were certain not to have a negative impact on the integrity or readability of the code. The downside to this freedom is it causes uncertainty about what aspects of the target they should try to mimic, what was significant and to what degree they were successful in their imitation. Having to carry out manual analysis also presented difficulties when it came to identifying the common features—without having access to quantifiable measures statistically significant aspects can easily be overlooked.

6.3.3 Experiences with Assisted Task

With the assisted task, P1 reported that they found it a bit tedious and frustrating at times, while P2 found it to be fun and *"almost like a game"*. Regarding the clarity of the recommendations, P1 stated that *"it was relatively difficult to implement the plug-in's suggestions"*. They went on to elucidate: *"some terms used to describe syntactical features, in the suggestions, were hard to understand"*. P2 was similarly confused with the recommendations, finding that *"at first it was difficult to interpret which changes I was supposed to make"*. This indicates some effort is required to improve the feature descriptions that are mapped during generation of the recommendation text. In some cases, the node frequencies are based on highly abstracted interfaces from the Eclipse AST API, which prove to be very difficult to describe in terms of tangible aspects of the code. A review of these node types, and whether they should in fact be included, would be prudent in future versions of the plugin.

Both respondents also found some suggestions were hard to implement without negatively affecting the behaviour and/or performance of the code in question, and were concerned about maintaining readability while adding redundant code to meet certain suggestions. Comparing the suggestions to the changes they had identified in the first task, P1 commented:

"in contrast to the features that I identified in the first task, all of which were actionable, not all of the suggestions provided by the plug-in were stable and implementable. Therefore, I had to spend time identifying which ones were or were not actionable."

P2 mostly implemented recommendations related to comments and string literals, because:

"I was afraid that other changes in the code would alter the behaviour of my program and would be difficult to manipulate in a functionally correct way."

However, even this strategy had undesirable consequences, as they noted *"my comments ended up looking very strange"*.

This highlights two problems, the first is related to using low-level features while the second is related to automated generation of recommendations. Using low-level features can present a problem when linking features to the original phenomena. In most cases, such features represent some normalization of a real characteristic that may not be a one-to-one mapping, in which case either assumptions must be made or human interpretation must be employed to ascertain the origin. In either case, the feature values themselves or any feedback derived from them, do not represent consumable, "actionable" suggestions. Indeed, the consumer of such advice must carry out two cognitive tasks: one to determine what underlying characteristics the low-level feature is derived from, and another to determine if and how those characteristics can be changed without adversely affecting either the essential or desirable qualities of the artifact. As an analogy, take character 4-grams in natural language stylometry. It might be the case that a particular author uses more instances of "tion" than another. For the author wishing to disguise their style and effect a reduction in this feature, they must firstly find all the words containing this 4-gram, then decide which of them can be changed and what to change them to. Alternatively, by presenting them with feedback indicating that a whole word, such as "obstruction", occurs too frequently, a tool assisting them could offer suggestions for alternatives, such as "barrier" or "hindrance". This reduces the cognitive load on the user, making the tool more intuitive and its results easier to interpret. While such an approach also improves the automation capability of such a tool (i.e., by making concrete suggestions), the second problem with any tool is that it can never decide on behalf of the user whether a suggestion will irrevocably change some desirable characteristic of the artifact. Such characteristics are, for the most part, subjective, and extremely difficult to quantify. By automating as much as can reasonably be automated with regards to the suggestions, however, we can at least reduce the decisions that must be made by the user to only those that are infeasible for a computer to calculate.

In terms of the additional information provided by the plugin's evaluations, P1 said:

"The continuous feedback to me, through the updated textual output to improve my (mis)-classification and the dialog box indicating the current classification, was very helpful."

They also liked that the suggestions were grouped according to their sibling and parent features, when presenting node frequency suggestions. The most significant benefit P1 found to using the tool over manual attempts, however, was the automation of the code analysis, which had proved to be one of the most difficult aspects of task one. They found they *"did not need to spend time identifying stylistic features in my or the target user's code"*. This was confirmed by

P2, who stated *"the advice the tool gave was very useful in pointing out features of my code that deviated from another author's programming style."*. P1 also commented that:

"the tool successfully did identify some stylistic features which were missed through the manual observation process in task one. For example, the tool indicated that I used a much higher frequency of integer type declarations and much lower frequencies for almost all other data types."

P2 found a similar benefit, saying *"it pointed out features that I didn't even think of when I was trying to imitate someone manually"*.

For potential improvements, both respondents commented that having examples of feature recommendations would have helped them complete the task, with P2 suggesting *"the task would have been a lot easier if I had concrete examples from the target's code"*. They also both wanted to see in future versions suggestions that would not alter the behaviour of the code, with P2 offering as a potential solution formal verification methods.

6.3.4 Summary of User Study

Overall, we can see there were both significant benefits and drawbacks to using the tool to assist with the mimicry attempt. First, the results for both Programmers P1 and P2 were significantly stronger in terms of reducing their own classification confidence with assistance than without, while Programmer P3 was able to achieve a lower confidence in task one than task two. Moreover, all three Programmers' files were classified as a different author after completion of task two, even if they did not manage to achieve a classification as the target, whereas only P3 managed to achieve this in task one. From the participants' responses to the questionnaire, they thought one of the most important benefits was that analysis of both theirs and the target's code was automated. This saves a great deal of time over manually inspecting the files and allows for objective and comprehensive comparison between features present in both sets of files. Of course, the automated approach is able to go much farther than that even, as many thousands of files and authors can all be compared within seconds in order to find the features that are potentially most significant, which would be beyond the capabilities of a single person carrying out a manual analysis. An additional benefit was that, with frequent feedback on progress, the participants were able to keep track of how much of an effect their changes were having and when they had made sufficient modifications to a particular feature to produce the desired effect. This guidance allowed them to focus on the important aspects and ignore the rest. The downsides were that it was often difficult to understand what they were being asked to do by the plugin and the suggestions often represented changes that were not conducive to maintaining functionality or readability of the code. The first of these downsides is entirely preventable in future versions if

more effort is put into improving the wording of the recommendations and the mapping between feature names and user-friendly descriptions. This problem would also be greatly improved with a better selection of features, utilizing fewer low-level and ambiguous representations and more higher-level characteristics that have a one-to-one mapping to recognizable elements in the code itself. The second drawback, relating to "correctness" of suggested changes, is much harder, and possibly touches on some unsolved problems related to automated program synthesis and analysis, which is an area of active research in its own right and far beyond the scope of this work. This drawback could be alleviated, however, with the same careful selection of features that would solve some of the issues related to clarity. Having more concrete recommendations, truly representative of style rather than content, would be easier for the user to incorporate and could be combined with examples of how to achieve the suggestion. A solution involving weighting of features when determining the degree of change could also be incorporated, so features known to be less intrusive and easier to implement without affecting the program behaviour, could be favoured over other features that are less easy to alter.

Chapter 7

Conclusions

Authorship attribution has captured the imaginations of researchers for well over a century, and applications in legal proceedings date back even farther. Initially concerned with physical document scrutiny and handwriting analysis, it has since progressed to examining the content of typeset digital documents in order to identify similarities in writing composition style—a type of authorship attribution known as stylometry. Furthermore, there has been a shift from subjective, qualitative analysis by human experts to quantitative, statistical analysis by computers. This shift towards computational stylometry has also precipitated an automated approach in the analysis and statistical models made up of multiple variables and very large corpora that would be far too complex and time consuming for human analysis. We can also see the progression of stylometry applications, from civil matters, such as disputed authorship, to criminal matters, such as evidence in legal proceedings, to societal matters, such as identifying rogue accounts in social networks and other, more Kafkaesque situations, where people are accused of, and punished for, breaking vague laws they did not know existed and without being told what those rules are.

Despite much published research into authorship attribution of natural language and more specifically, stylometry, there is still a lack of clear consensus on the precise attributes, their method of extraction and techniques for analyzing them that produce consistent, and significant, results over all but the most trivial of author sets. Moreover, there is yet to be published a study that conclusively demonstrates where the dividing line is between style and content of a document. There has been some limited research investigating the robustness of stylometry in adversarial situations, revealing a distinct brittleness in the state of the art to both obfuscation and mimicry attacks. This brittleness is suggestive of attributes that capture only superficial similarities between documents by the same author, which may in turn indicate more content-specific characteristics are being found, rather than true style, which should be present at a more

fundamental level. These adversarial studies are far from comprehensive, however, and there is still much more that can, and should, be done.

A closely related field to authorship attribution in the natural language setting, is code stylometry, which seeks to perform authorship attribution on program source code. Computer programming languages have much in common with natural languages, but with a stricter, and simpler, syntax. This is no accident, as they were designed to be more intuitive and easier to read and follow for humans than machine code. With the addition of comments in particular, which are free-form text sections in a program source file, many of the same or similar attributes can be found in source code stylometry as for natural language stylometry. Interestingly, despite a close kinship with its natural language counterpart, code stylometry arose completely independently, with its origins in teaching, and a desire for aesthetically pleasing programs that adhere to accepted best practices. Unsurprisingly, source code stylometry is firmly rooted in automated, computer-based analyses, using statistical and machine learning models.

Being a more recent field, and considering it as a subset of the natural language setting, source code stylometry has naturally been the subject of fewer published research papers. This correlation extends to the adversarial setting as well, of which there have been no published studies to date. Source code stylometry has been identified as a potential threat to the privacy of software developers, particularly those working in the open-source community. In addition, several recent cases have highlighted a worrying trend of governments targeting the developers of tools deemed to be used primarily for bypassing Internet censorship and surveillance. It is easy to see how these two separate phenomena could be combined to threaten the safety and anonymity of current contributors, as well as push would-be contributors into silence. Alternatively, using authorship attribution has also been proposed as a means of identifying computer criminals and malware developers. Before we can reach any meaningful conclusions about its applications, however, it is important to understand its limitations with more research into its feasibility in real-world settings, its robustness in adversarial settings and its ability to discern style from content.

To this end, we evaluated code stylometry on real open-source repositories on GitHub, to identify some of the difficulties of performing such an operation, and offer some potential solutions. Our aims here were to establish how much of a practical threat this technique poses, and develop robust defences against it. We decided to implement our defence as an adversarial imitation of another author's style, utilizing machine learning as a medium by which to extract the aspects of the code to modify and assess the degree of success of the imitation attempt. Random forests were chosen as the learning algorithm to achieve this, due to their speed, accuracy and conduciveness to rule extraction and parsing. We chose to use a human-in-the-loop model for our system, where the tool provides recommendations for the user that will result in a successful imitation if followed, and it is down to the user's discretion whether and how to implement

said recommendations. We presented our solution as a plugin for the popular open-source IDE Eclipse, embedding the Weka machine learning system as the provider of the learning algorithm.

We used the GitHub data API to enumerate C repositories, filtering those with self-declared multiple authors or obviously copied code. We further filtered repositories by size and content, looking for evidence of multiple author involvement in the commit history, common files by hash and third-party code in specific subdirectories. Despite these precautions, the dataset we ended up with contained significant amounts of noise in the form of multiple authorship, copied code, third-party libraries and auto-generated code by tools. This noise became apparent during evaluation of our dataset with our chosen features, when a random selection of 25 repositories that had performed the worst during evaluation was manually inspected and found to contain 22 "noisy" repositories. Re-evaluating our dataset while excluding the worst-performing repositories significantly improved the accuracy of our classifications. We chose this cleaner dataset for the background training data used in the plugin, for its improved accuracies, as well as compactness and faster/more responsive training times.

Our feature set consisted of a mixture of AST node type unigram frequencies, preferences for operator types, storage classes, macros and data types, naming conventions, and comment and string literal contents. In order to realize the rule extraction and parsing aspect of our system, we modified Weka and the random forest algorithm to provide it with a "memory" of its training instances. We also incorporated a novel algorithm we devised to traverse the trees in the forest, build a set of conditions and produce recommendations based on a specific training instance that was closest in terms of a degree of change metric to the file to be modified. These recommendations were therefore guaranteed to result in a classification as the target user by at least one real file that is possible to exist.

We tested our recommendation algorithm by producing change sets for every file contained in our background training data, as though that file were the *target* we wish to imitate. We then evaluated the change sets by producing sparse feature vectors containing only feature values drawn from the recommended intervals and classifying them with the random forest they were derived from, and a new random forest trained on the same data. Our results showed that the confidence on the original random forest was close to the confidence for the actual training instance, demonstrating a very successful imitation, while the confidence for the new random forest was significantly lower; however, this confidence increased with the size of the forest. Furthermore, classifier confidence is typically far lower than accuracy, and even relatively low confidences can result in a successful classification, depending on the number of classes involved.

Finally, we ran a pilot user study to gain feedback and assess the usability of the plugin with a small number of participants. The results showed that two of the three participants performed far better in the task of imitating another user with the assistance of the tool than without. The

participants that returned responses to a questionnaire about their experiences highlighted the ability of the tool to perform a mass analysis of their and the target's source code and continual feedback on progress as the main benefits. The clarity of recommendations and difficulty implementing them without negatively impacting the code's readability or behaviour were given as the main drawbacks.

7.1 Future Work

We identified a number of future directions and avenues for this line of research, as well as improvements to be made to our system.

Starting with data collection, in future work more effort should be invested in ensuring high-quality data is gathered, with as little noise as possible. In addition to the steps we already took that were described in Section 5.3, some suggestions include performing searches within the code itself, looking for strings such as "copyright", "written by", "author" or even email addresses to extract names and compare them to the repository owner, as well as each other; two distinct names or email addresses in a single repository should be viewed as suspicious, as should distinct names in two repositories owned by the same user. Furthermore, building a database of the names of well-known open-source software to use as a blacklist would be useful (unless that repository *is* the original repository and happens to be singly authored; e.g., Vim, written/maintained by Bram Moolenaar). Regarding auto-generated code, typically a comment is placed in the file by the tool indicating that it was generated, so this could also be searched for and the file(s) filtered. Finally, in addition to checking file hashes for duplicates, files by different authors should be flagged that are closer than some threshold edit distance. Clearly, with greater scrutiny there will be fewer files, repositories and authors in the data set, so the data collection process should be run for longer, possibly even until all C repositories on GitHub have been enumerated. This process would take several months at the rates our scripts were limited to, so some consideration should be given to improving its efficiency.

Moving on to feature extraction, despite the main aim of our work being to devise defences against stylometry that can work with any feature set, having a better feature set that more accurately captures individual style would be beneficial, for three reasons. First, if this tool is to be released for public use, we should make sure we have done all we can to produce a strong classification as a starting point that is not too easy to defeat, else the user may be imbued with a false sense of security regarding their anonymity. Second, one of our system's design goals was to not be too intrusive with recommendations, which is largely dependent on to what degree the features represent style rather than content. If there is too much correlation with content, the changes are going to be difficult for the user to implement without damaging readability,

maintainability or behaviour, possibly even introducing errors or defects. If the features are correlated with style, however, there should always be an alternative way of achieving the same end result that does not have these same negative effects. Therefore, it is crucial that any future work addresses this point and strives to separate style from content. Third, a better feature set would also be more informative to the research community in general about how feasible this sort of forensic analysis is in practice, and hence how serious of a threat it should be considered.

Improving our recommendation algorithm could form part of future work, in particular looking at ways to increase the classification accuracy of feature vectors drawn from the distribution described by the derived intervals, without sacrificing the flexibility of having relatively wide ranges. Possibly a very different approach could be taken, using some form of stochastic process to search the feature space and optimize toward high classification confidences, although as pointed out in Section 5.6.3, we would need to demonstrate that this approach generated feature vectors that valid source files according to the syntax of the language could map to.

In terms of the plugin, the main area requiring improvement is in the wording of the recommendations so they make sense to users. This is also tied in to the feature set and to what degree it maps to recognizable aspects of the code itself, so the user need not discern the relationship between a recommendation and its originating location in the code. Using low-level features means a higher degree of correlation and dependence between features, which makes it harder for users to implement one recommendation without affecting other feature values and potentially their suggestions too. This makes using the tool much more complex for the user and increases their frustration. Features of a more binary nature, such as capturing a preference for using one language construct over another, would be much easier for users to comprehend than asking them to reduce the relative frequency of a language construct to some arbitrary value, while increasing another by an equivalent amount. We must strike an appropriate balance between accuracy and effectiveness in obfuscation/mimicry and usability. These two qualities may not be entirely opposed; it is possible one may find an optimum set of features and recommendation technique that succinctly captures a user's style, consistent across the majority of their work, that can also be changed easily and unobtrusively.

Furthermore, future work should expand on our pilot user study to incorporate more participants. Ours was limited to graduate students from the CrySP research group at the University of Waterloo, but a future study could instead invite participants from among the background data extracted from GitHub; many owners include their real names and email addresses in their GitHub profiles. Indeed we found a number of "@uwaterloo.ca" email address among the owners in our dataset, either students or faculty, that would be ideal candidates for an initial recruitment campaign. If their repositories are in our dataset, they are suitable for inclusion in our study, so no selection process would be necessary. Any future user study should only be conducted after the issues highlighted by the respondents to our questionnaire have been addressed, however. We

would like to see a future study investigate how programmers write code when trying to publish covertly versus overtly, whether they do in fact take steps to disguise their style or other markers of identity from the source, and what those steps are. It would be difficult to recruit potential participants for such a study for two reasons: first, the challenges of identifying anonymous publication—many user accounts use pseudonyms and do not include contact details, but are not necessarily trying to remain anonymous. Second, anonymous publishers would probably be reluctant to participate, and for good reason; such a study would undoubtedly pose a risk to their continuing anonymity. Therefore, a more sensible approach would be to *simulate* this scenario with participants.

Finally, in addition to comprehensive systems such as those developed during this thesis, it would be of considerable benefit to furnish the community with a set of recommendations, or manual steps, one might take to provide a minimum level of protection. These steps could be based on knowledge of the state of the art with regards to attacks, but also sensible measures, such as checking for personally identifiable information in source code comments, not copying and pasting code from overt repositories and purging metadata from documents or images.

7.2 Final Remarks

What *is* style? This word is often used in authorship attribution, but with no attempt to actually define it. The meaning of the word, however, is far from concrete and absolute. Like the word *art*, or *humour*, for example, it is a rather vague and ambiguous term. Using such a word as this so often in research papers surely demands a definition of some kind, or more specifically, some formal description. Many times, the term *style* is used to describe any characteristic of a document, file, etc., that can be statistically correlated to a certain label, with no attempt to critically evaluate whether that truly represents style, or is simply a manifestation of the content, topic or functionality of the artifact in question. Just because a characteristic of an artifact is present in more than one document by the same author and seems to be correlated with that author, does not mean it is independent of content. Note it is far easier to mix these concepts than it may appear, especially if one is optimizing according to how well the learning algorithm classifies samples drawn from the same population as the training data. Files in the same repository often contain copied and pasted code, as well as comments, from other files. It is our belief that truly stylistic features should be defined and justified *a priori*, based on the formal language syntax itself, rather than its discriminatory power. It must be demonstrated that the feature represents a choice between (superficially) equally valid options; e.g., choosing whether to implement a loop as a `do/while` rather than `while`. Even a feature that differentiates between `do/while` and `while` loop constructs can pose a problem, as there are legitimate reasons for using one

rather than the other that have nothing to do with style. A metaphor in everyday life would be the choice of clothes two people make. If weather is not taken into account, a learning algorithm may learn that one person's "style" is to prefer sweaters to t-shirts, or umbrellas to sunglasses. Granted, there may have been a choice made between these options, but the decision was heavily influenced by the fact it was cold or raining on the day one person's choice of clothes was sampled and warm and sunny on the day the other person's choice was sampled.

Our definition of the word *style* is that it represents a choice between equally valid options for achieving some objective, be that summing the results of a number of expressions, or keeping warm on a cold day. Whenever one is presented with a choice, and the options from which to decide between are relatively equal in terms of merit, personal preference must come into play. When a set of preferences are grouped according to the category of choices they express, such as programming language constructs, we refer to that as a person's style. Using this definition, we can assess the claim that a certain feature of an artifact is a matter of style or not, by asking what other equally valid option did they have for achieving the outcome produced by that feature? Taking natural language as an example, synonyms can represent style. These were used by Clark and Hannon [CH07] and were found by Brennan *et al.* [BAG12] to be the most robust of the three systems they tested to adversarial modifications. It should be noted, however, that dangers are present even within this representation of style, as synonyms often have subtly (sometimes not so subtly, depending on context) different meanings that may favour selecting one word over its synonyms to convey a particular nuance. In computer programming we can actually assess this in a far more definite way, due to the fact high-level languages are compiled to machine code. Clearly, if two different statements compile to the same machine code, then selecting one over the other was a matter of style, by our very definition of style being a choice between two or more equally valid options. If the two statements produce exactly the same outcome, they are equivalent. This could even be expanded to include statements that compile to similar sequences of machine code, within some limit. In a less exact manner, we could define our assessment as blocks of statements that have the same effect on program state, and further relaxing the requirement we could use unit test code written by the author to define the behaviour of a method or function to define style over a larger structure.

Whatever our definition of *style* is, we must not lose sight of the original motivations for this work, namely to assist open-source contributors protect their identity in order to avoid persecution. In this respect, what matters most are results, regardless of the method or theory; a deployed system attempting to perform attribution at scale would likewise be results-oriented, and it is *that* we are trying to defend against. Therefore, any defences that are developed should be required to be tested against all known attacks as a minimum, including those not considered to be effective, before it is released; if a defence is truly robust, the ineffective attacks should be trivial to nullify. In the context of website fingerprinting, Wang and Goldberg [WG17] discuss

defences that are effective against all *possible* attacks, not just all known attacks. In our case, such a theoretical defence would be one that, given two programs written by different authors with the same functionality, could find a set of changes that would transform one into the other, creating two identical programs. In practice, it is unlikely any human programmers would be so consistent that their style choices could be predicted so perfectly, but a system that tried to model the thought processes of a programmer when presented with different problems could bridge this gap. We have taken the first tentative steps toward the ultimate goal of an effective and usable advisor, demonstrating the techniques one might employ, the challenges faced, and potential solutions. There is still much more to be done, however, and it is our hope that the challenges will be met, and overcome, in due course.

References

[Abn91] Steven P. Abney. Parsing by Chunks. In *Principle-Based Parsing*, pages 257–278. Springer, 1991.

[ACIS+14] Sadia Afroz, Aylin Caliskan-Islam, Ariel Stolerman, Rachel Greenstadt, and Damon McCoy. Doppelganger Finder: Taking Stylometry to the Underground. In *Proceedings - IEEE Symposium on Security and Privacy*, pages 212–226, 2014.

[Ada74] Douglass Adair. *Fame and the Founding Fathers: Essays*. Institute of Early American History and Culture at Williamsburg, Va., 1974.

[ASP+14] Saed Alrabaee, Noman Saleem, Stere Preda, Lingyu Wang, and Mourad Debbabi. OBA2: An Onion Approach to Binary Code Authorship Attribution. *Digital Investigation*, 11, Supplement 1:S94 – S103, 2014. Proceedings of the First Annual {DFRWS} Europe.

[BAG12] Michael Brennan, Sadia Afroz, and Rachel Greenstadt. Adversarial Stylometry: Circumventing Authorship Recognition to Preserve Privacy and Anonymity. *ACM Transactions on Information and System Security*, 15(3):1–22, 2012.

[Bai79] Richard W. Bailey. Authorship Attribution in a Forensic Setting, 1979.

[Bar11] Anabela Barreiro. SPIDER: A System for Paraphrasing in Document Editing and Revision-Applicability in Machine Translation Pre-Editing. *Computational Linguistics and Intelligent Text Processing*, pages 365–376, 2011.

[Bla34] Adam Blackwood. *History of Mary Queen of Scots*. Number 31. Maitland Club, 1834.

[BLL+15] Authors Bill, Marczak Lead, Nicholas Weaver Lead, Jakub Dalek, Roya Ensafi, David Fiflield, Sarah Mckune, Arn Rey, John Scott-Railton, Ronald Deibert, and Vern Paxson. China's Great Cannon. (April):1–19, 2015.

[BM85] Robert E. Berry and Brian A.E. Meekings. A Style Analysis of C Programs. *Communications of the ACM*, 28(1):80–88, 1985.

[BMA13] Mudit Bhargava, Pulkit Mehndiratta, and Krishna Asawa. Stylometric Analysis for Authorship Attribution on Twitter. In *Lecture Notes in Computer Science (Including Subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, volume 8302 LNCS, pages 37–47, 2013.

[BNJT10] Marco Barreno, Blaine Nelson, Anthony D. Joseph, and J. D. Tygar. The Security of Machine Learning. *Machine Learning*, 81(2):121–148, 2010.

[Boy05] Clark Boyd. The Price Paid for Blogging Iran, 2005. [Online; Accessed 21-September-2017; http://news.bbc.co.uk/2/hi/technology/4283231.stm].

[Bre96] Leo Breiman. Bagging Predictors. *Machine Learning*, 24(2):123–140, 1996.

[Bre01] Leo Breiman. Random Forests. *Machine Learning*, 45(1):5–32, 2001.

[Bri63] Claude S. Brinegar. Mark Twain and the Quintus Curtius Snodgrass Letters : A Statistical Test of Authorship. *Journal of the American Statistical Association*, 58(301):85–96, 1963.

[Bru78] Etienne Brunet. *Le Vocabulaire De Jean Giraudoux. Structure Et Évolution*. Number 1. Slatkine, 1978.

[BS84] Hal L. Berghel and David L. Sallach. Measurements of Program Similarity in Identical Task Environments. *ACM SIGPLAN Notices*, 19(8):65–76, 1984.

[BS98] Robert A. Bosch and Jason A. Smith. Separating Hyperplanes and the Authorship of the Disputed Federalist Papers. *The American Mathematical Monthly*, 105(7):601–608, 1998.

[Bud15] Bill Budington. China Uses Unencrypted Websites to Hijack Browsers in GitHub Attack, 2015. [Online; Accessed 5-October-2017; https://www.eff.org/deeplinks/2015/04/china-Uses-Unencrypted-Websites-to-Hijack-Browsers-in-Github-Attack].

[Bur92] John F. Burrows. Not Unless You Ask Nicely: The Interpretative Nexus Between Analysis and Information. *Literary and Linguistic Computing*, 7(2):91–109, 1992.

[Bur02] John Burrows. Delta: A Measure of Stylistic Difference and a Guide to Likely Authorship. *Literary and Linguistic Computing*, 17(3):267–287, 2002.

[Cam15] Jamie Campbell. Young People Going to Increasing Lengths to Protect Online Privacy, 2015. [Online; Accessed 25-September-2017; https://www.independent.co.uk/news/world/young-People-Going-to-Increasing-Lengths-to-Protect-Online-Privacy-10108955.html].

[Cen92] Electronic Privacy Information Center. International Traffic in Arms Regulations, 1992. [Online; Accessed 14-Nov-2017; https://epic.org/crypto/export_controls/itar.html].

[CG96] Stanley F. Chen and Joshua Goodman. An Empirical Study of Smoothing Techniques for Language Modeling. In *Proceedings of the 34th Annual Meeting on Association for Computational Linguistics*, pages 310–318. Association for Computational Linguistics, 1996.

[CGGR03] Nicola Cancedda, Eric Gaussier, Cyril Goutte, and Jean-Michel Renders. Word-Sequence Kernels. *Journal of Machine Learning Research*, 3(Feb):1059–1082, 2003.

[CH07] Jonathan H. Clark and Charles J. Hannon. A Classifier System for Author Recognition Using Synonym-Based Features. In *Mexican International Conference on Artificial Intelligence*, pages 839–849. Springer, 2007.

[Cha14] China Change. Young IT Professional Detained for Developing Software to Scale GFW of China, 2014. [Online; Accessed 5-October-2017; https://chinachange.org/2014/11/12/young-It-Professional-Detained-for-Developing-Software-to-Scale-Gfw-of-China].

[CIHL+15] Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt. De-Anonymizing Programmers via Code Stylometry. *24th USENIX Security Symposium (USENIX Security 15)*, pages 255–270, 2015.

[Clo03] Paul Clough. Old and New Challenges in Automatic Plagiarism Detection. *National Plagiarism Advisory Service*, (February):14, 2003.

[CNM06] Rich Caruana and Alexandru Niculescu-Mizil. An Empirical Comparison of Supervised Learning Algorithms. In *Proceedings of the 23rd International Conference on Machine Learning*, ICML '06, pages 161–168, New York, NY, USA, 2006. ACM.

[Coh87] Fred Cohen. Computer Viruses. Theory and Experiments. *Computers and Security*, 6(1):22–35, 1987.

[Coh88] Fred Cohen. On the Implications of Computer Viruses and Methods of Defense. *Computers & Security*, 7(2):167–184, 1988.

[CRC15] Niall J. Conroy, Victoria L. Rubin, and Yimin Chen. Automatic Deception Detection: Methods for Finding Fake News. *Proceedings of the Association for Information Science and Technology*, 52(1):1–4, 2015.

[CYD+15] Aylin Caliskan-Islam, Fabian Yamaguchi, Edwin Dauber, Richard Harang, Konrad Rieck, Rachel Greenstadt, and Arvind Narayanan. When Coding Style Survives Compilation: De-anonymizing Programmers from Executable Binaries. *ArXiv e-prints*, December 2015.

[DCHG17] Edwin Dauber, Aylin Caliskan, Richard Harang, and Rachel Greenstadt. Git Blame Who?: Stylistic Authorship Attribution of Small, Incomplete Source Code Fragments. *ArXiv e-prints*, January 2017.

[Dij71] Edsger Wybe Dijkstra. *A Short Introduction to the Art of Programming*, volume 4. Technische Hogeschool Eindhoven Eindhoven, 1971.

[DKLP03] Joachim Diederich, Jörg Kindermann, Edda Leopold, and Gerhard Paass. Authorship Attribution With Support Vector Machines. *APPLIED INTELLIGENCE*, 19(1-2):109–123, 2003.

[DS04] Haibiao Ding and Mansur H. Samadzadeh. Extraction of Java Program Fingerprints for Software Authorship Identification. *Journal of Systems and Software*, 72(1):49–57, 2004.

[DVdB05] Walter Daelemans and Antal Van den Bosch. *Memory-Based Language Processing*. Cambridge University Press, 2005.

[DZVDSVDB04] Walter Daelemans, Jakub Zavrel, Kurt Van Der Sloot, and Antal Van Den Bosch. Timbl: Tilburg Memory-Based Learner. *Tilburg University*, 2004.

[EHW16] Frank Eibe, Mark A. Hall, and Ian H. Witten. *The WEKA Workbench. Online Appendix for "Data Mining: Practical Machine Learning Tools and Techniques"*. Morgan Kaufmann, fourth edition, 2016. [Online (https://www.cs.waikato.ac.nz/ml/weka/Witten_et_al_2016_appendix.pdf); Accessed 22-October-2017].

[EW13] Paul Edmondson and Stanley Wells. *Shakespeare Beyond Doubt: Evidence, Argument, Controversy*. Cambridge University Press, 2013.

[FMSG08] Georgia Frantzeskou, Stephen MacDonell, Efstathios Stamatatos, and Stefanos Gritzalis. Examining the Significance of High-Level Programming Features in Source Code Author Classification. *Journal of Systems and Software*, 81(3):447–460, 2008.

[For81] Richard Forsyth. BEAGLE-A Darwinian Approach to Pattern Recognition. *Kybernetes*, 10(3):159–166, 1981.

[Fou17] Electronic Frontier Foundation. Printer Tracking: Is Your Printer Spying on You?, 2017. [Online; Accessed 21-September-2017; https://www.eff.org/issues/printers].

[Fri89] Jerome H. Friedman. Regularized Discriminant Analysis. *Journal of the American Statistical Association*, 84(405):165–175, 1989.

[FSG+07] Georgia Frantzeskou, Efstathios Stamatatos, Stefanos Gritzalis, Carole E. Chaski, and Blake Stephen Howald. Identifying Authorship by Byte-Level N-Grams: The Source Code Author Profile (SCAP) Method. *International Journal of Digital Evidence*, 6(1):1–18, 2007.

[Fun17] Brian Fung. The House Just Voted to Wipe Away the FCCs Landmark Internet Privacy Protections, 2017. [Online; Accessed 14-Nov-2017; https://www.washingtonpost.com/news/the-Switch/wp/2017/03/28/the-House-Just-Voted-to-Wipe-Out-the-Fccs-Landmark-Internet-Privacy-Protections].

[GHM05] Neil Graham, Graeme Hirst, and Bhaskara Marthi. Segmenting Documents by Stylistic Character . 11(4):397–415, 2005.

[GMS97] Andrew Gray, Stephen MacDonell, and Philip Sallis. Software Forensics : Extending Authorship Analysis Techniques to Computer Programs (The Information Science Discussion Papers Series). (97/14), 1997.

[Goo17] Google. Transparency Report, 2017. [Online; Accessed 23-September-2017; https://transparencyreport.google.com].

[GP16] Jennifer Stisa Granick and Riana Pfefferkorn. Brief of Amici Curiae iPhone Security and Applied Cryptography Experts in Support of Apple Inc.s Motion to Vacate Order Compelling Apple Inc. to Assist Agents in Search, and Opposition to Governments Motion to Compel Assistance, 2016.

[Gri07] Jack Grieve. Quantitative Authorship Attribution: An Evaluation of Techniques. *Literary and Linguistic Computing*, 22(3):251–270, 2007.

[Gro16] Lev Grossman. Inside Apple CEO Tim Cooks Fight With the FBI, 2016. [Online; Accessed 14-Nov-2017; http://time.com/4262480/tim-Cook-Apple-Fbi-2/].

[Hal07] Hans Van Halteren. Author Verification by Linguistic Profiling: An Exploration of the Parameter Space. *ACM Transactions on Speech and Language Processing (TSLP)*, 4(1):1, 2007.

[HF95] David I. Holmes and Richard S. Forsyth. The Federalist Revisited: New Directions in Authorship Attribution. *Literary and Linguistic Computing*, 10(2):111 –127, 1995.

[HO10] Jane Huffman Hayes and Jeff Offutt. Recognizing Authors: an Examination of the Consistent Programmer Hypothesis. *Software Testing, Verification and Reliability*, 20(4):329–356, 2010.

[Hol92] David I. Holmes. A Stylometric Analysis of Mormon Scripture and Related Texts. *Journal of the Royal Statistical Society. Series a (Statistics in Society)*, 155(1):91–120, 1992.

[Hol94] David I. Holmes. Authorship Attribution. *Computers and the Humanities*, 28(2):87–106, 1994.

[Hol98] David I. Holmes. The Evolution of Stylometry in Humanities Scholarship. *Literary and Linguistic Computing*, 13(3):111 –117, 1998.

[Hon79] Antony Honoré. Some Simple Measures of Richness of Vocabulary. *Association for Literary and Linguistic Computing Bulletin*, 7(2):172–177, 1979.

[IHFD08] Farkhund Iqbal, Rachid Hadjidj, Benjamin C M Fung, and Mourad Debbabi. A Novel Approach of Mining Write-Prints for Authorship Attribution in E-Mail Forensics. *Digital Investigation*, 5(SUPPL.):42–51, 2008.

[JFF+14] Anne Jellema, Hania Farhan, Khaled Fourati, Siaka Lougue, Dillon Mann, and Gabe Trodd. Web Index Report, 2014. [Online; http://thewebindex.org/wp-content/uploads/2014/12/Web_Index_24pp_November2014.pdf].

[JGVH95] Ralph Johnson, Erich Gamma, John Vlissides, and Richard Helm. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[JRWM15] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. Obfuscator-LLVM–Software Protection for the Masses. In *Software Protection (SPRO), 2015 IEEE/ACM 1st International Workshop on*, pages 3–9. IEEE, 2015.

[Juo04] Patrick Juola. Ad-Hoc Authorship Attribution Competition. In *Proceedings of the Joint Conference of the Association for Computers and the Humanities and the Association for Literary and Linguistic Computing*, pages 175–176. Goteborg, Sweden, 2004.

[JW10] Matthew Jockers and Daniela Witten. A Comparative Study of Machine Learning Methods for Authorship Attribution. *Literary and Linguistic Computing*, 25(2):215–223, 2010.

[Ker83] Auguste Kerckhoffs. La Cryptographie Militaire. *Journal des Sciences Militaires*, IX:5–83, 1883.

[KG06] Gary Kacmarcik and Michael Gamon. Obfuscating Document Stylometry to Preserve Author Anonymity. *Proceedings of the COLING/ACL on Main Conference Poster Sessions* -, pages 444–451, 2006.

[KGSM98] Richard I. Kilgour, Andrew R. Gray, Philip J. Sallis, and Stephen G. Macdonell. A Fuzzy Logic Approach to Computer Software Source Code Authorship Analysis. *Fourth International Conference on Natural Processing*, pages 865–868, 1998.

[Knu73] Donald E. Knuth. *Fundamental Algorithms: The Art of Computer Programming*. 1973.

[Koz92] John R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*, volume 1. MIT Press, 1992.

[KS97] Ivan Krsul and Eugene H. Spafford. Authorship Analysis: Identifying the Author of a Program. *Computers & Security*, 16(3):233–257, 1997.

[KS04] Moshe Koppel and Jonathan Schler. Authorship Verification as a One-Class Classification Problem. *Twenty-First International Conference on Machine Learning - ICML '04*, page 62, 2004.

[KSA11] Moshe Koppel, Jonathan Schler, and Shlomo Argamon. Authorship Attribution in the Wild. *Language Resources and Evaluation*, 45(1):83–94, 2011.

[KSAM06] Moshe Koppel, Jonathan Schler, Shlomo Argamon, and Eran Messeri. Authorship Attribution With Thousands of Candidate Authors. *Sigir*, pages 659–660, 2006.

[KSSM07] Jay Kothari, Maxim Shevertalov, Edward Stehle, and Spiros Mancoridis. A Probabilistic Approach to Source Code Authorship Identification. *Proceedings - International Conference on Information Technology-New Generations, ITNG 2007*, pages 243–248, 2007.

[Lam16] Oiwan Lam. Leaked Xinjiang Police Report Describes Circumvention Tools as 'Terrorist Software'. 2016. Online; Accessed 2-October-2017; https://globalvoices.org/2016/10/26/leaked-Xinjiang-Police-Report-Describes-Circumvention-Tools-as-Terrorist-Software.

[LD05] Kim Luyckx and Walter Daelemans. Shallow Text Analysis and Machine Learning for Authorship Attribution. In *Proceedings of the Fifteenth Meeting of Computational Linguistics in the Netherlands CLIN 2004*, pages 149–160. 2005.

[LD08] Kim Luyckx and Walter Daelemans. Authorship Attribution and Verification With Many Authors and Limited Data. In *Belgian/Netherlands Artificial Intelligence Conference*, pages 335–336, 2008.

[LD11] Kim Luyckx and Walter Daelemans. The Effect of Author Set Size and Data Size in Authorship Attribution. *Literary and Linguistic Computing*, 26(1):35–55, 2011.

[Le04] Zhang Le. Maximum Entropy Modeling Toolkit for Python and C++. *Natural Language Processing Lab, Northeastern University, China*, 2004.

[LMD01] Caroline Lyon, James Malcolm, and Bob Dickerson. Detecting Short Passages of Similar Text in Large Document Collections. *Proceedings of the 2011 Conference on Empirical Methods in Natural Language Processing (EMNLP 2001)*, pages 118–125, 2001.

[LS93] Thomas A. Longstaff and E. Eugene Schultz. Beyond Preliminary Analysis of the WANK and OILZ Worms: A Case Study of Malicious Code. *Computers & Security*, 12(1):61–77, 1993.

[LWD10] Robert Layton, Paul Watters, and Richard Dazeley. Authorship Attribution for Twitter in 140 Characters or Less. *Proceedings - 2nd Cybercrime and Trustworthy Computing Workshop, CTC 2010*, pages 1–8, 2010.

[MAC+12] Andrew W.E. McDonald, Sadia Afroz, Aylin Caliskan, Ariel Stolerman, and Rachel Greenstadt. Use Fewer Instances of the Letter "i": Toward Writing Style Anonymization. In *Privacy Enhancing Technologies*, volume 7384, pages 299–318. Springer, 2012.

[Mar90] John Markoff. Computer Intruder Is Put on Probation and Fined 10,000, 1990. [Online; Accessed 30-September-2017; http://www.nytimes.com/1990/05/05/us/computer-Intruder-Is-Put-on-Probation-and-Fined-10000.html].

[McC09] Cormac McCarthy. *The Road*. Pan Macmillan, 2009.

[Mee83] Brian A.E. Meekings. Style Analysis of Pascal Programs. *SIGPLAN Notices*, 18(September):45–54, 1983.

[Men87] Thomas Corwin Mendenhall. The Characteristic Curves of Composition. *Science*, 9(214):237–249, 1887.

[MGS99] Stephen G. Macdonell, Andrew R. Gray, and Philip J. Sallis. Software Forensics for Discriminating Between Program Authors Using Case-Based Reasoning , Feed-Forward Neural Networks and Multiple Discriminant Analysis

1 Introduction 2 Techniques for Authorship Dis- Crimination 3 Authorship Data Set. pages 66–71, 1999.

[MW63] Frederick Mosteller and David L. Wallace. Inference in an Authorship Problem, 1963.

[MW64] Frederick Mosteller and David Wallace. Inference and Disputed Authorship: The Federalist. 1964.

[NI17] Stephen Nellis and David Ingram. Vote to Repeal U.S. Broadband Privacy Rules Sparks Interest in VPNs, 2017. [Online; Accessed 25-September-2017; https://www.reuters.com/article/us-Usa-Internet-Privacy/vote-to-Repeal-U-S-Broadband-Privacy-Rules-Sparks-Interest-in-Vpns-idUSKBN17005Y].

[NPG+12] 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. In *Security and Privacy (SP), 2012 IEEE Symposium on*, pages 300–314. IEEE, 2012.

[O'B15] Danny O'Brien. Speech That Enables Speech: China Takes Aim at Its Coders, 2015. [Online; Accessed 5-October-2017; https://www.eff.org/deeplinks/2015/08/speech-Enables-Speech-China-Takes-Aim-Its-Coders].

[OC89] Paul Oman and Curt Cook. Programming Style Authorship Analysis. *Proceedings of the 17th Conference on ACM . . .*, pages 320–326, 1989.

[OG16] Rebekah Overdorf and Rachel Greenstadt. Blogs, Twitter Feeds, and Reddit Comments: Cross-domain Authorship Attribution. *PoPETs*, 2016(3):155–171, 2016.

[OLRV11] Gabriel Oberreuter, Gaston L'Huillier, Sebastián A. Rıos, and Juan D. Velásquez. Approaches for Intrinsic and External Plagiarism Detection. *Proceedings of the PAN*, 2011.

[Orm03] Hilarie Orman. The Morris Worm: A Fifteen-Year Perspective. *IEEE Security & Privacy*, 99(5):35–43, 2003.

[Per15] Percy. Chinese Developers Forced to Delete Softwares by Police, 2015. [Online; Accessed 5-October-2017; https://en.greatfire.org/

blog/2015/aug/chinese-Developers-Forced-Delete-Softwares-Police].

[PH06] Justin W. Patchin and Sameer Hinduja. Bullies Move Beyond the Schoolyard: A Preliminary Look at Cyberbullying. *Youth Violence and Juvenile Justice*, 4(2):148–169, 2006.

[Pow98] David M.W. Powers. Applications and Explanations of Zipf's Law. In *Proceedings of the Joint Conferences on New Methods in Language Processing and Computational Natural Language Learning*, pages 151–160. Association for Computational Linguistics, 1998.

[PR81] Moshe Pollatschek and Yehuda Thomas Radday. Vocabulary Richness and Concentration in Hebrew Biblical Literature. *ALLC BULL.*, 8(3):217–231, 1981.

[Pro11] Paradyn Project. ParseAPI. http://www.paradyn.org/html/parse9.0.3-features.html, 2011. A platform-independent API for parsing and extracting control flow graphs from binary code.

[Qui86] Ross Quinlan. Induction of Decision Trees. *Machine Learning*, 1(1):81–106, 1986.

[RCCC16] Victoria L. Rubin, Niall J. Conroy, Yimin Chen, and Sarah Cornwell. Fake News or Truth? Using Satirical Cues to Detect Potentially Misleading News. In *Proceedings of NAACL-HLT*, pages 7–17, 2016.

[Reb14] Rebel Labs. Java Tools and Technologies Landscape for 2014, 2014.

[RM17] Ann M Rogerson and Grace McCarthy. Using Internet-Based Paraphrasing Tools: Original Work, Patchwriting or Facilitated Plagiarism? *International Journal for Educational Integrity*, 13(1):2, 2017.

[RNI10] Miloš Radovanović, Alexandros Nanopoulos, and Mirjana Ivanović. Hubs in space: Popular nearest neighbors in high-dimensional data. *Journal of Machine Learning Research*, 11(Sep):2487–2531, 2010.

[Rob17] Adi Robertson. A VPN Can Stop Internet Companies From Selling Your Data—but It's Not a Magic Bullet, 2017. Online; https://www.theverge.com/2017/3/25/15056290/vpn-Isp-Internet-Privacy-Security-Fcc-Repeal.

[Rou87] Peter J. Rousseeuw. Silhouettes: A Graphical Aid to the Interpretation and Validation of Cluster Analysis. *Journal of Computational and Applied Mathematics*, 20:53–65, 1987.

[RZM11] Nathan Rosenblum, Xiaojin Zhu, and Barton P. Miller. Who Wrote This Code? Identifying the Authors of Program Binaries. In *Proceedings of the 16th European Conference on Research in Computer Security*, ESORICS'11, pages 172–189, Berlin, Heidelberg, 2011. Springer-Verlag.

[SAM96] Philip Sallis, Asbjorn Aakjaer, and Stephen MacDonell. Software Forensics: Old Methods for a New Science. *Software Engineering: Education and Practice, 1996. Proceedings. International Conference*, pages 481–485, 1996.

[SB88] Gerard Salton and Christopher Buckley. Term-Weighting Approaches in Automatic Text Retrieval. *Information Processing & Management*, 24(5):513–523, 1988.

[Seb02] Fabrizio Sebastiani. Machine Learning in Automated Text Categorization. *ACM Computing Surveys (CSUR)*, 34(1):1–47, 2002.

[SG06] Conrad Sanderson and Simon Guenter. Short Text Authorship Attribution via Sequence Kernels, Markov Chains and Author Unmasking: An Investigation. *Computational Linguistics*, (July):482–491, 2006.

[Sic75] Herbert S. Sichel. On a Distribution Law for Word Frequencies. *Journal of the American Statistical Association*, 70(351a):542–547, 1975.

[Sic86] Herbert S. Sichel. Word Frequency Distributions and Type-Token Characteristics. *Mathematical Scientist*, 11(1):45–72, 1986.

[SLP11] Benno Stein, Nedim Lipka, and Peter Prettenhofer. Intrinsic plagiarism analysis. *Language Resources and Evaluation*, 45(1):63–82, 2011.

[Smi83] Michael William Smith. Recent Experience and New Developments of Methods for the Determination of authorship. *ALLC BULL.*, 11(3):73–82, 1983.

[Smi90] M. Wilfrid A. Smith. Attribution by Statistics: A Critique of Four Recent Studies. *Revue, Informatique Et Statistique Dans Les Sciences Humaines*, 26:233–251, 1990.

[SOS12] Privacy SOS. Programmer and Activist Interrogated at the Border, 2012. [Online; Accessed 5-October-2017; https://privacysos.org/blog/programmer-and-Activist-Interrogated-at-the-Border].

[Spa88] Eugene H. Spafford. The Internet Worm Program : An Analysis. 1988.

[Spa92] Eugene H. Spafford. Software Forensics : Can We Track Code to Its Authors ? 12:585–594, 1992.

[Spa17] Andrew Sparrow. WhatsApp Must Be Accessible to Authorities, Says Amber Rudd, 2017. [Online; Accessed 24-September-2017; https://www.theguardian.com/technology/2017/mar/26/intelligence-Services-Access-Whatsapp-Amber-Rudd-Westminster-Attack-Encrypted-Messaging].

[Sta06] Efstathios Stamatatos. Ensemble-Based Author Identification Using Character N-Grams. 2006.

[Sta09] Efstathios Stamatatos. Intrinsic Plagiarism Detection Using Character N-Gram Profiles. *threshold*, 2(1,500), 2009.

[SW93] Eugene H Spafford and Stephen A Weeber. Software Forensics: Can We Track Code to Its Authors? *Computers & Security*, 12(6):585–595, 1993.

[SZK18] Lucy Simko, Luke Zettlemoyer, and Tadayoshi Kohno. Recognizing and Imitating Programmer Style: Adversaries in Program Authorship Attribution. *PoPETs*, 2018(1):127–144, 2018.

[Tal72] D. Roger Tallentire. *An Appraisal of Methods and Models in Computational Stylistics, With Particular Reference to Author Attribution*. PhD thesis, University of Cambridge, 1972.

[TC84] David R. Tobergte and Shirley Curtis. Program Complexity and Programming Style. *Data Engineering, 1984 IEEE First International Conference On*, pages 534 – 541, 1984.

[THNC03] Robert Tibshirani, Trevor Hastie, Balasubramanian Narasimhan, and Gilbert Chu. Class Prediction by Nearest Shrunken Centroids, With Applications to DNA Microarrays. *Statistical Science*, pages 104–117, 2003.

[TSH96] Fiona J. Tweedie, Sameer Singh, and David I. Holmes. Neural Network Applications in Stylometry: The Federalist Papers. *Computers and the Humanities*, 30(1):1–10, 1996.

[UK16a] HM Government UK. Investigatory Powers Act 2016, 2016. [Online; Accessed 23-September-2017; http://www.legislation.gov.uk/id?title=Investigatory+Powers+Act+2016].

[UK16b] HM Government UK. Investigatory Powers Act 2016, 2016. [Online; Accessed 23-September-2017; http://www.legislation.gov.uk/ukpga/2016/25/section/128].

[VD02] Ricardo Vilalta and Youssef Drissi. A Perspective View and Survey of Meta-Learning. *Artificial Intelligence Review*, 18(2):77–95, 2002.

[VVC08] Heidi Vandebosch and Katrien Van Cleemput. Defining Cyberbullying: A Qualitative Research Into the Perceptions of Youngsters. *CyberPsychology & Behavior*, 11(4):499–503, 2008.

[WC98] David Woolls and Malcolm Coulthard. Tools for the Trade. *Forensic Linguistics*, 5(2):33–57, 1998.

[WG17] Tao Wang and Ian Goldberg. Walkie-Talkie: An Efficient Defense Against Passive Website Fingerprinting Attacks. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1375–1390, 2017.

[WMT15] Nicholas Watt, Rowena Mason, and Ian Traynor. David Cameron Pledges Anti-Terror Law for Internet After Paris Attacks, 2015. [Online; Accessed 24-September-2017; https://www.theguardian.com/uk-News/2015/jan/12/david-Cameron-Pledges-Anti-Terror-Law-Internet-Paris-Attacks-Nick-Clegg].

[YGAR14] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and Discovering Vulnerabilities with Code Property Graphs. In *Security and Privacy (SP), 2014 IEEE Symposium on*, pages 590–604. IEEE, 2014.

[Yul14] George Udny Yule. *The Statistical Study of Literary Vocabulary*. Cambridge University Press, 2014.

[ZES06] Sven Meyer Zu Eissen and Benno Stein. Intrinsic Plagiarism Detection. *Advances in Information Retrieval*, pages 565–569, 2006.

APPENDICES

Appendix A

User Study Questionnaire

Thank you for agreeing to participate in this study. As previously mentioned, I am interested in exploring your experiences, thoughts and feedback regarding the tasks you were just asked to complete. I would like to remind you that you are not obligated to participate in the study or respond to any questions in the questionnaire if you do not wish to. You may choose to end your participation in this study at any time without repercussions.

1. How would you describe your experience of completing the first task of imitating the other author's style without guidance or assistance? How easy/difficult did you find the task? Did you feel you were able to make sufficient changes to successfully imitate their style of programming?

2. What aspects of the task did you find particularly challenging, and why?

3. What aspects of the task did you find particularly easy, and why?

4. How would you describe your experience of completing the second task of imitating another author's style with the guidance of the tool? How easy/difficult did you find the task? Do you feel the changes you were able to make were sufficient to successfully imitate their style of programming?

5. What aspects of this task did you find particularly challenging, and why?

6. What aspects of this task did you find particularly easy, and why?

7. Overall, how would you compare the difficulty of completing the task with and without the assistance of the tool? Did you find the advice provided by the tool to be useful?

8. Would you recommend this tool to others who might benefit from its application?
9. What recommendations do you have for how the tool might be more useful or effective in carrying out the tasks?
10. Do you have any other comments you'd like to add about your participation in the study today?