Style Counsel: Seeing the (Random) Forest for the Trees in Adversarial Code Stylometry*

Christopher McKnight
Ian Goldberg

ABSTRACT

The results of recent experiments have suggested that *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 code sample 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, we devised a tool, *Style Counsel*, 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.

Our tool 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 more accurate models of style. We ran a pilot 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.

1 INTRODUCTION

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. In response to this, the use cases for authorship attribution have grown to encompass new areas, such as cyber bullying [31, 40], fake news [12, 34] and more sinister purposes, such as the identification and subsequent persecution of political bloggers [5], 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.

One particular technique for authorship attribution, known as *stylometry*, is defined as "the statistical analysis of literary style" [14], and as such is concerned solely with the content of the document, rather than the method or medium of its delivery. Stylometry techniques tend to focus on syntactical features, such as choice of words, spelling, preference for certain grammatical structures and even layout; these characteristics make it ideal for use in digital analysis. 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 [27] to more modern scenarios such as identifying the authors of tweets [3]. 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 [7, 18], which may largely be a result of the underlying machine learning algorithms employed, that are known to be susceptible to evasion attacks [2], 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* [9], 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" [26, 38]. Following the Internet/Morris Worm incident in 1988 [30], a report was published that attempted to profile the author of the worm based on an examination of the reverse-engineered code [37], casting style analysis as a forensic technique in addition to a code quality metric.

Despite a substantial body of prior work on program authorship attribution, only one paper we are aware of [35] (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. We go beyond prior work by developing a tool (a plugin for the Eclipse IDE) named *Style Counsel* to assist programmers in obscuring their coding style by 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

*This is an extended version of our WPES 2018 paper [25]. Even more detail can be found in the first author's Master's thesis [24].

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*.

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 [17], 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."* [17] 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."*

There are several cases of *software 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 [36]. 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 developing software to help Chinese Internet users scale the Great Fire Wall of China"* [10] 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 [29]. 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"* [32], 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"* [8], which has been linked to the Chinese government [21] and has been suggested was in an attempt to bully the site into removing repositories that contravened their censorship regulations.

As the environment turns hostile towards the developers, many of them may opt to disguise their identities 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 identities. 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.

3 RELATED WORK

As a research topic, authorship attribution has been, and continues to be, popular, with dozens of papers published each year. The first author's Master's thesis [24, Ch. 4] features an extensive discussion of stylometric analysis, of natural language, source code, and even executable binaries. In this paper, we will focus on *defences* against stylometric analysis.

Kacmarcik and Gamon [18] 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. For the modification list, they first calculated the relative frequencies of all 8,674 words found in the papers. They then applied a novel feature selection process using decision trees to establish which words to focus on. This process began by generating a decision tree from all the features, 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 either 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 as 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.

They then trained SVM classifiers on feature sets of varying dimensions (from three to 70), taken from other studies that used the same corpus [4, 15, 28, 39], which assigned all but essay 55 to Madison (in agreement with the consensus 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 documents were assigned to him. 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. 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 [20] demonstrated that the change in classification may be somewhat superficial.

[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.

Brennan, Afroz, and Greenstadt [7] 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, they tried three techniques with the participants: manual obfuscation, manual imitation, and 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 of his book *The Road* [22]. For the final technique, the participants' sample documents and excerpts from *The Road* were subjected to automated translation by the services provided by Bing[2] and Google[3] 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[4] 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 [11], 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.
(3) SVM with the write-prints [16] 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. 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, which was marginally better. 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. A paraphrasing tool, such as Barreiro's SPIDER [1] would clearly be a better option for automated obfuscation of text and an abundance of these tools are available online [33].

In a follow-up paper, McDonald *et al.* [23] 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.

In parallel with our own work, another paper has been published that investigates adversarial source code stylometry, by Simko *et al.* [35]. This work carries out two significant user studies looking into the robustness of a state-of-the-art source code stylometry system [9] 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, data from the Google Code Jam competition[5] were used as the corpus, taken from the dataset used by Caliskan-Islam *et al.* [9].

It is important to note that all code was formatted prior to both studies, using an open-source tool.[6] 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 main finding from the first study was that the classifier used was not robust against the adversarial attacks. With feedback on their success, participants were able to successfully forge the target's style 66.6, 70, and 73.3% of the time for 5-, 20- and 50-class datasets. Without feedback, the average success rate was 61.1%. 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 what they termed an attempt at forgery that was unsuccessful at being classified as Y, 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*

[2]https://www.bing.com/translator
[3]https://translate.google.com/
[4]https://www.mturk.com/mturk/welcome
[5]https://code.google.com/codejam/
[6]http://astyle.sourceforge.net

given no information at all, they can develop successful forgery detection strategies without examples of forgeries or instructions about forgery creation." [35] When performing a simple attribution task; i.e., classification without knowledge of forgeries, the average attack success rate was 56.6%. 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.

In terms of the types of changes made, the researchers found these were *"overwhelmingly local control flow, local information structure changes, and typographical changes."* These local changes are in contrast to *algorithmic changes*, such as using dynamic programming instead of recursion, or refactoring a block of code into a helper function. 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, their paper offers 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 code style, much in the vein of Anonymouth [23] by McDonald *et al.* for disguising one's natural-language writing style.

4 STYLE COUNSEL

4.1 Contributions

While there are some papers [7, 18, 23] investigating natural language stylometry from an adversarial perspective, and one [35] about the source code equivalent, we look at *automating* the process of making suggestions for altering source code to imitate the style of another author. If code stylometry 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 identities against such a threat. To this end, this work offers the following contributions:

(1) A new set of features for capturing elements of programming style.
(2) 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.
(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 pilot user study evaluating the usability of the tool, the feasibility of manually imitating another's style, and the practicalities of using the tool for this task.

4.2 Data Collection and Feature Extraction

One of the aims of our work was to perform a realistic authorship attribution study, to discover, highlight and hopefully overcome some of the practical challenges 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, textbooks 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 (particularly with regards to their education). Code from textbooks is likely to be proofread 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,[7] with the caveat that the code belong to a single author (and truly 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 has 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 intended beneficiaries of our tool. 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.

We enumerated 66,619 C repositories on GitHub, collecting data on 11,164 that GitHub reported as containing 32 KB or more of C code with a single contributor. After cloning, we filtered the set of repositories as follows (see the thesis version of this work [24, §5.3] for more details):

(1) Removing repositories whose commit logs contained names or email addresses of different people.

[7] https://github.com

Table 1: Distribution of repositories per author

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

(2) Removing code from "lib" and "ext" folders.
(3) Excluding repositories belonging to different owners that contained identical files.
(4) Removing duplicate files by the same author.
(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.

Our evaluations were carried out on a final tally of 1,261 repositories from 525 authors. Table 1 gives the distribution of repositories per author in our dataset.

Our target platform is the Eclipse IDE, so 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. The Eclipse C Development Tools (CDT) provides a convenient mechanism for traversing the AST it constructs internally, with an abstract class (ASTVisitor) containing callback methods one can implement and pass as an argument to the AST interface (IASTTranslationUnit). Our feature set is constructed largely from this tree traversal, while specialized feature extractors are used to parse comments and preprocessor directives, which are not present in the AST.

We extracted features in the following categories:

- **Node Frequencies**—The relative frequency of AST node type unigrams in the AST (detailed in Appendix A.1).
- **Node Attributes**—The relative 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. (detailed in Appendix A.2).
- **Identifiers**—Naming conventions, average length, etc. (detailed in Appendix A.3).
- **Comments**—Use of comments, average length, ratio of comments to other structures, etc. (detailed in Appendix A.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.* [35] 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 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 counting node bigrams as used by Caliskan-Islam *et al.* [9], or character *n*-grams, as implemented by Frantzeskou *et al.* [13]. Node bigram-based features result in extremely high-dimensional feature vectors, while character *n*-grams 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 its main purpose 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.

4.3 Training and Making Predictions

Once 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 (using the popular open-source Weka platform[8]) to make predictions about the author of a file, and subsequently of an entire repository. The random forest algorithm, first proposed by Breiman [6], 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 receiving the most votes is taken to be the overall prediction. Random forests are discussed in more detail in the thesis version of this work [24, §5.2.4]. We decided to use a similar training and evaluation methodology to hold-one-out, 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. 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. The thesis version of this work [24, §5.5] gives the details of our hold-one-out evaluation against the entire corpus, with and without character unigram frequencies.

4.4 Making Recommendations

A significant part of our system, and crucial to its effectiveness, is 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

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

individual, rather than just "any" author, or "no" author (obfuscation) are as follows: first, with obfuscation the aim is 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 [19].

4.4.1 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. 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. 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. 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. 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, which we can then use 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. 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.

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.

4.4.2 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 tree traversal. 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.

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 any child nodes returned a list indicating one or more descendant leaf nodes classify as the target. If so, insert the current node to the start of the list, 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;

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 forests, for reasons that will be elaborated in the next section.

4.4.3 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 $[0, x), [x, +\infty)$.

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, Λ and Γ, containing all the split points where the path we are interested in follows the less than or greater than/equal side of the 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 $peek(\Gamma') < peek(\Lambda')$. We detail this approach in Algorithm 1 in Appendix D.

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.

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 1.

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 represents a feasible combination of source code elements. 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. Algorithms 2 and 3 in Appendix D are extensions to Algorithm 1 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 we have not suffered from overfitting. As future work, 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 evaluations.

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 confidence is with some form of metaheuristic algorithm, such as simulated annealing, memetic search, hill climbing or genetic algorithms. 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.

4.4.4 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. 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 in the Eclipse AST 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 in determining what to change.

4.5 Using the Plugin

In this section, we 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 1).

Setup. Before using the plugin, the user should have a corpus of publicly available and attributable source code they have authored, and a second corpus of unpublished source code, for which they wish to disguise their authorship. In this case, their public source

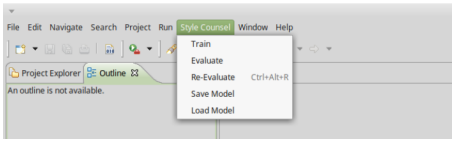

Figure 1: Plugin menu

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, 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 resources can include files that do not contain source code, as the plugin filters out such files. When training a model, the user should select resources other than those they wish to modify. Once the plugin has the list of source code files, it proceeds to extract the features described in Section 4.2 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 4.4.4.

Evaluation. Upon choosing this command, the user is once again presented with a resource selection dialog 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, as well as an aggregate classification/confidence value for the set of files (see Figure 2), 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 4.4, the differences are calculated and finally recommendations are generated and placed into template messages as described in Section 4.4.4. The recommendations are given as warnings in the "problems" view in the workspace (see Figure 3).

4.6 Pilot User Study

In order to help assess the usability and feasibility of our plugin, we conducted a small pilot user study in order to receive feedback from real users—extremely valuable for developing an effective tool.

For our pilot study, we chose participants that had C programming experience and a corpus of source code files they had authored.

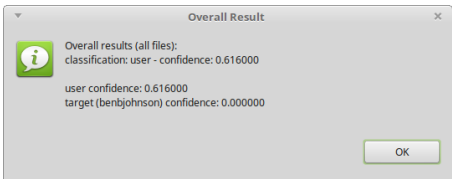

Figure 2: Aggregate output

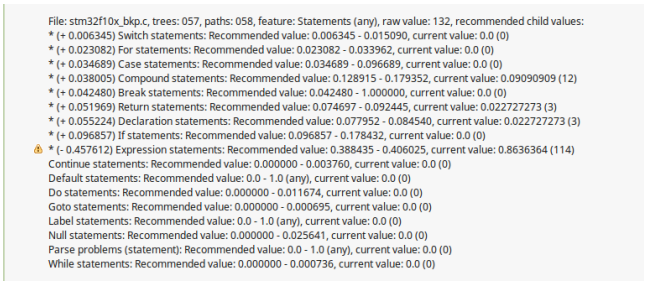

Figure 3: Sample recommendations

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 from carrying 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).

5 RESULTS

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

We evaluated the recommendation algorithm described in Section 4.4. 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.

We carried out this evaluation by generating recommendations for each file in our corpus, as though the author of that file were the *target*. The min, max and mid values of each recommendation were used to perturb that file's existing feature vector. 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. Two versions of these min-, mid-, and max-modified vectors

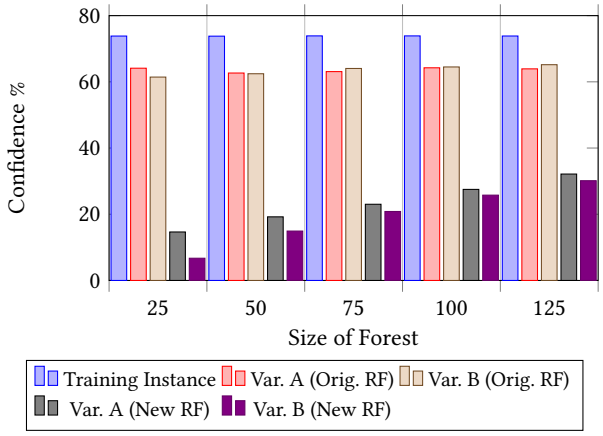

Figure 4: Results of evaluating variation A and B feature vectors generated from recommendations. The fabricated feature vectors were evaluated by the random forest that produced them and an alternate random forest trained on the same data. The confidences reported here indicate how successfully the extracted intervals model the volume of feature space occupied by the training instance, with higher confidences being more successful.

were then produced, according to how the features with no recommendations were treated. In variation A, those features were left as their original values, and in variation B they were set to 0. If the variation A and B feature vectors return similar confidence labels, then we can say the recommendations only included the features that actually contribute to the classification. This 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 (representing the fact that the defender aiming to hide her identity will not have access to the exact trained classifier being used by the stylometric analyst). The results of these evaluations are given in Figure 4. The confidence returned for the original training instances are averaged across both random forests, as the differences are negligible, while the confidence for the variation A and B 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 accuracy in our context is always strictly greater than the confidence.

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 perturbed feature vectors on the second random forest also increases. The performance with the same feature vectors and the original classifier

increases only marginally, and for the original 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 variation A vectors give significantly better confidences in the smallest retrained forests due to the compensating effects of defaulting to the original 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 variation A and B vectors, we can see there is very little difference, which demonstrates that the subset of features used in deriving the recommendations for 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.

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, once using the middle values and once using the maximum values in the interval. The output of these three samples are averaged in Figure 4 for clarity, but note that there are minor differences in their performance. For evaluations on the original forest, the feature vectors using the maximum values from the intervals gave the best average performance. 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.

5.2 Pilot User Study

Our next 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 B. 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 answer a short written 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 workstation 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 4.4.4 and 4.5 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.

5.2.1 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 modifying 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 from task one. 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. On completing the assisted task with Target Y, 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 yielded 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.

For detailed discussion of the participant experiences in the pilot user study, see Appendix C.

5.2.2 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 harder to alter.

6 CONCLUSIONS

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 combine 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 developed an algorithm for providing source code modification recommendations that will result in a successful imitation if followed, using a "human-in-the-loop" model, where it is down to the user's discretion whether and how to implement said recommendations. We presented our solution as a plugin called *Style Counsel* for the popular open-source IDE Eclipse.

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, and are ripe targets for future work.

ACKNOWLEDGEMENTS

We thank NSERC for grants STPGP-463324 and RGPIN-03858. This work benefitted from the use of the CrySP RIPPLE Facility at the University of Waterloo.

REFERENCES

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

[2] Marco Barreno, Blaine Nelson, Anthony D. Joseph, and J. D. Tygar. 2010. The Security of Machine Learning. *Machine Learning* 81, 2 (2010), 121–148. https://doi.org/10.1007/s10994-010-5188-5

[3] Mudit Bhargava, Pulkit Mehndiratta, and Krishna Asawa. 2013. 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)*, Vol. 8302 LNCS. 37–47. https://doi.org/10.1007/978-3-319-03689-2_3

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

[5] Clark Boyd. 2005. The Price Paid for Blogging Iran. http://news.bbc.co.uk/2/hi/technology/4283231.stm [Online; Accessed July 2018].

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

[7] Michael Brennan, Sadia Afroz, and Rachel Greenstadt. 2012. Adversarial Stylometry: Circumventing Authorship Recognition to Preserve Privacy and Anonymity. *ACM Transactions on Information and System Security* 15, 3 (2012), 1–22. https://doi.org/10.1145/2382448.2382450
[8] Bill Budington. 2015. China Uses Unencrypted Websites to Hijack Browsers in GitHub Attack. https://www.eff.org/deeplinks/2015/04/china-Uses-Unencrypted-Websites-to-Hijack-Browsers-in-Github-Attack [Online; Accessed July 2018].
[9] Aylin Caliskan-Islam, Richard Harang, Andrew Liu, Arvind Narayanan, Clare Voss, Fabian Yamaguchi, and Rachel Greenstadt. 2015. De-Anonymizing Programmers via Code Stylometry. *24th USENIX Security Symposium (USENIX Security 15)* (2015), 255–270. https://doi.org/10.1145/2665943.2665958 arXiv:1512.08546
[10] China Change. 2014. Young IT Professional Detained for Developing Software to Scale GFW of China. https://chinachange.org/2014/11/12/young-It-Professional-Detained-for-Developing-Software-to-Scale-Gfw-of-China/ [Online; Accessed July 2018].
[11] Jonathan H. Clark and Charles J. Hannon. 2007. A Classifier System for Author Recognition Using Synonym-Based Features. In *Mexican International Conference on Artificial Intelligence*. Springer, 839–849.
[12] Niall J. Conroy, Victoria L. Rubin, and Yimin Chen. 2015. Automatic Deception Detection: Methods for Finding Fake News. *Proceedings of the Association for Information Science and Technology* 52, 1 (2015), 1–4.
[13] Georgia Frantzeskou, Efstathios Stamatatos, Stefanos Gritzalis, Carole E. Chaski, and Blake Stephen Howald. 2007. Identifying Authorship by Byte-Level N-Grams: The Source Code Author Profile (SCAP) Method. *International Journal of Digital Evidence* 6, 1 (2007), 1–18.
[14] David I. Holmes. 1998. The Evolution of Stylometry in Humanities Scholarship. *Literary and Linguistic Computing* 13, 3 (1998), 111 –117. https://doi.org/10.1093/llc/13.3.111
[15] David I. Holmes and Richard S. Forsyth. 1995. The Federalist Revisited: New Directions in Authorship Attribution. *Literary and Linguistic Computing* 10, 2 (1995), 111 –127. https://doi.org/10.1093/llc/10.2.111
[16] Farkhund Iqbal, Rachid Hadjidj, Benjamin C M Fung, and Mourad Debbabi. 2008. A Novel Approach of Mining Write-Prints for Authorship Attribution in E-Mail Forensics. *Digital Investigation* 5, SUPPL. (2008), 42–51. https://doi.org/10.1016/j.diin.2008.05.001
[17] Anne Jellema, Hania Farhan, Khaled Fourati, Siaka Lougue, Dillon Mann, and Gabe Trodd. 2014. Web Index Report. [Online; http://thewebindex.org/wp-content/uploads/2014/12/Web_Index_24pp_November2014.pdf; Accessed August 2018].
[18] Gary Kacmarcik and Michael Gamon. 2006. Obfuscating Document Stylometry to Preserve Author Anonymity. *Proceedings of the COLING/ACL on Main Conference Poster Sessions* - (2006), 444–451. https://doi.org/10.3115/1273073.1273131
[19] Auguste Kerckhoffs. 1883. La Cryptographie Militaire. *Journal des Sciences Militaires* IX (1883), 5–83.
[20] Moshe Koppel and Jonathan Schler. 2004. Authorship Verification as a One-Class Classification Problem. *Twenty-First International Conference on Machine Learning - ICML '04* (2004), 62. https://doi.org/10.1145/1015330.1015448
[21] Bill Marczak, Nicholas Weaver, Jakub Dalek, Roya Ensafi, David Fiflield, Sarah Mckune, Arn Rey, John Scott-Railton, Ronald Deibert, and Vern Paxson. 2015. *China's Great Cannon.* Technical Report. Citizen Lab. https://citizenlab.ca/2015/04/chinas-Great-Cannon/, [Accessed Aug 2018].
[22] Cormac McCarthy. 2009. *The Road.* Pan Macmillan.
[23] Andrew W.E. McDonald, Sadia Afroz, Aylin Caliskan, Ariel Stolerman, and Rachel Greenstadt. 2012. Use Fewer Instances of the Letter "i": Toward Writing Style Anonymization. In *Privacy Enhancing Technologies*, Vol. 7384. Springer, 299–318.
[24] Christopher McKnight. 2018. *StyleCounsel: Seeing the (Random) Forest for the Trees in Adversarial Code Stylometry.* Master's thesis. University of Waterloo. https://uwspace.uwaterloo.ca/handle/10012/12856.
[25] Christopher McKnight and Ian Goldberg. 2018. Style Counsel: Seeing the (Random) Forest for the Trees in Adversarial Code Stylometry. In *17th ACM Workshop on Privacy in the Electronic Society (WPES 2018).*
[26] Brian A.E. Meekings. 1983. Style Analysis of Pascal Programs. *SIGPLAN Notices* 18, September (1983), 45–54.
[27] Frederick Mosteller and David Wallace. 1964. Inference and Disputed Authorship: The Federalist. (1964).
[28] Frederick Mosteller and David L. Wallace. 1963. Inference in an Authorship Problem. , 275–309 pages. https://doi.org/10.1080/01621459.1963.10500849
[29] Danny O'Brien. 2015. Speech That Enables Speech: China Takes Aim at Its Coders. https://www.eff.org/deeplinks/2015/08/speech-Enables-Speech-China-Takes-Aim-Its-Coders [Online; Accessed July 2018].
[30] Hilarie Orman. 2003. The Morris Worm: A Fifteen-Year Perspective. *IEEE Security & Privacy* 99, 5 (2003), 35–43.
[31] Justin W. Patchin and Sameer Hinduja. 2006. Bullies Move Beyond the Schoolyard: A Preliminary Look at Cyberbullying. *Youth Violence and Juvenile Justice* 4, 2 (2006), 148–169.
[32] Percy. 2015. Chinese Developers Forced to Delete Softwares by Police. https://en.greatfire.org/blog/2015/aug/chinese-Developers-Forced-Delete-Softwares-Police [Online; Accessed July 2018].
[33] Ann M Rogerson and Grace McCarthy. 2017. Using Internet-Based Paraphrasing Tools: Original Work, Patchwriting or Facilitated Plagiarism? *International Journal for Educational Integrity* 13, 1 (2017), 2.
[34] Victoria L. Rubin, Niall J. Conroy, Yimin Chen, and Sarah Cornwell. 2016. Fake News or Truth? Using Satirical Cues to Detect Potentially Misleading News. In *Proceedings of NAACL-HLT*. 7–17.
[35] Lucy Simko, Luke Zettlemoyer, and Tadayoshi Kohno. 2018. Recognizing and Imitating Programmer Style: Adversaries in Program Authorship Attribution. *PoPETs* 2018, 1 (2018), 127–144.
[36] Privacy SOS. 2012. Programmer and Activist Interrogated at the Border. https://privacysos.org/blog/programmer-and-Activist-Interrogated-at-the-Border/ [Online; Accessed July 2018].
[37] Eugene H. Spafford. 1989. The Internet Worm Program: An Analysis. *ACM SIGCOMM Computer Communication Review* 19, 1 (1989), 17–57.
[38] David R. Tobergte and Shirley Curtis. 1984. Program Complexity and Programming Style. *Data Engineering, 1984 IEEE First International Conference On* (1984), 534 – 541. https://doi.org/10.1109/ICDE.1984.7271316
[39] Fiona J. Tweedie, Sameer Singh, and David I. Holmes. 1996. Neural Network Applications in Stylometry: The Federalist Papers. *Computers and the Humanities* 30, 1 (1996), 1–10. https://doi.org/10.1007/BF00054024
[40] Heidi Vandebosch and Katrien Van Cleemput. 2008. Defining Cyberbullying: A Qualitative Research Into the Perceptions of Youngsters. *CyberPsychology & Behavior* 11, 4 (2008), 499–503.

A DETAILS OF OUR FEATURE SET

A.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. 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.

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.

A.2 Node Attributes

Many of the interfaces encountered define attributes that provide additional information about the particular node type it represents. 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, for example.

A.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.[9] 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.

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

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.

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 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.

A.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.

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.

A.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**

B 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?

C PARTICIPANT EXPERIENCES IN THE PILOT USER STUDY

C.1 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 4.2, 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.

C.2 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.

D DETAILS OF THE ALGORITHMS

Algorithm 1 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

Algorithm 2 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 1
        $interval \leftarrow$ GETINTERVAL($featureConditions$, *paths*)
        $intervals.put(conditionsEntry.key, interval)$
      end for
    end if
  end for
  return *intervals*
end function

Algorithm 3 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(Γ, Λ, *paths*)
  end if
  return (γ_0, λ_0)
end function

function RESOLVEOVERLAP(Γ, Λ, *paths*)
  while $\gamma_0 \geq \lambda_0$ **do**
    $\varphi \leftarrow$ CHOOSE(Γ, Λ, *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 3 continued

function CHOOSE(Γ, Λ, *paths*)
  $numPaths_\gamma \leftarrow$ GETNUMPATHSINTREE($\gamma_0.tree$, *paths*)
  $numPaths_\lambda \leftarrow$ GETNUMPATHSINTREE($\lambda_0.tree$, *paths*)
  if $numPaths_\gamma = 1 \wedge numPaths_\lambda > 1$ **then**
    return λ_0
  else if $numPaths_\lambda = 1 \wedge numPaths_\gamma > 1$ **then**
    return γ_0
  else
    // Check which side has the greatest gap to its successor
    if $\gamma_1 - \gamma_0 > \lambda_0 - \lambda_1$ **then**
      return γ_0
    else if $\gamma_1 - \gamma_0 < \lambda_0 - \lambda_1$ **then**
      return λ_0
    else
      return Choose randomly $\in \{\gamma_0, \lambda_0\}$
    end if
  end if
end function

function GETNUMPATHSINTREE(*treeId*, *paths*)
  numPaths $\leftarrow 0$
  for all *path* $\in$ *paths* **do**
    if *path.tree* = *treeId* **then**
      numPaths++
    end if
  end for
  return *numPaths*
end function