ASEE 2022 ANNUAL CONFERENCE
Excellence Through Diversity
MINNEAPOLIS, MINNESOTA, JUNE 26TH-29TH, 2022 ASEE
Paper ID #36982

Sensitivity Preservation and Precision of Plagiarism Detection Engines for Modified Short Programs

Dylan Ryman

Dylan is currently an undergraduate studying computer science and mathematics at the University of Cincinnati. He is preparing to begin graduate studies in engineering education. His current research interests include source code plagiarism detection and computational thinking education with a focus on visual programming languages.

P.K. Imbrie (Head and Professor, Department of Engineering Education and Professor, Department of Aerospace Engin)

Head and Professor, Department of Engineering Education and Professor, Department of Aerospace Engineering and Engineering Mechanics University of Cincinnati P.K. Imbrie received his B.S., M.S. and Ph.D. degrees in Aerospace Engineering from Texas A&M University. He is an advocate for research-based approaches to engineering education, curricular reform, and student retention. Imbrie conducts both traditional, as well as educational research in experimental mechanics, piezospectroscopic techniques, epistemologies, assessment, and modeling of student learning, student success, student team effectiveness, and global competencies He helped establish the scholarly foundation for engineering education as an academic discipline through lead authorship of the landmark 2006 JEE special reports “The National Engineering Education Research Colloquies” and “The Research Agenda for the New Discipline of Engineering Education.” He has a passion for designing state-of-the-art learning spaces. While at Purdue University, Imbrie co-led the creation of the First-Year Engineering Program’s Ideas to Innovation (i2i) Learning Laboratory, a design-oriented facility that engages students in team-based, socially relevant projects. While at Texas A&M University Imbrie co-led the design of a 525,000 square foot state-of-the-art engineering education focused facility; the largest educational building in the state. Professor Imbrie’s expertise in educational pedagogy, student learning, and teaching has impacted thousands of students at the universities for which he has been associated. He is internationally recognized for his work in active/collaborative learning pedagogies and is a co-author of a text on teaming called Teamwork and Project Management. His engineering education leadership has produced fundamental changes in the way students are educated around the world. His current research interests include: epistemologies, assessment, and modeling of student learning, student success, student team effectiveness; experimental mechanics; and piezospectroscopic techniques.

Jeff Kastner (Professor Educator)

Sensitivity Preservation and Precision of Plagiarism Detection Engines for Modified Short Programs

Abstract

Source code plagiarism presents a continual threat to the integrity and effectiveness of engineering education, as habitual cheating often has devastating impacts on students' academic and professional careers. As programming becomes an increasingly central component of first-year engineering curricula, it is essential that instructors are able to uphold academic integrity by identifying students who engage in misconduct, either through direct plagiarism or excessive peer collaboration. Instructors have an arsenal of plagiarism detection tools at their disposal, and students are keenly aware of this. Consequently, in an attempt to evade detection, students routinely make superficial modifications to plagiarized work prior to submission. Effective plagiarism detection tools attempt to mitigate the effect of these alterations, however, the extent to which precision can be maintained for heavily modified code is limited. One aim of this paper is to quantify the effect of code modification strategies on a plagiarism detection tool's ability to preserve both sensitivity to plagiarism and precision of results. This paper will introduce a novel dimensionless metric apt for the evaluation and comparison of a plagiarism detection tool's robustness to code modification. The specific context of engineering education presents additional challenges, as research in plagiarism detection methods and performance is often not applicable to short programs in dynamically typed languages which constitute typical submissions in first-year engineering coursework. This paper will analyze the performance of relevant plagiarism detection tools on short Python programs, specifically those of fifty lines or fewer, that have been transformed by common code modification tactics, and evaluate which tools are most appropriate for use in this environment.

1 Introduction

Plagiarism is of grave concern to engineering educators, and the prevalence of plagiarism in engineering higher education is disturbingly high. In fact, 74% of engineering students self-report engaging in academic dishonesty [1]. Alarmingly, the frequency of plagiarism in engineering education continues to increase [2], perhaps because most students believe they have a right to plagiarize if they perceive their assigned workload to be unreasonable [3], or due to the rapid proliferation of contract cheating [4].

Source code plagiarism, while a narrow subset of plagiarism in general, is particularly well-suited for accurate automatic detection. Source code plagiarism detection or similarity analysis tools, which will be known as "similarity engines" for the remainder of this paper, are an integral

component of the assessment pipeline for assignments involving student source code submissions. Many distinct theoretical developments have been applied to similarity engines, resulting in a wide array of different technologies available for use.

Student attempts to modify plagiarized work in an effort to evade detection by similarity engines, which will be known as "mutations" for the remainder of this paper, are of substantial concern to engineering educators as they threaten the ability of the assessment process to accurately identify which students behaved ethically and which students engaged in academic misconduct. Therefore, it is essential that similarity engines are as well-equipped as possible for mitigating the impact of these attempts. The ability of a similarity engine to retain accurate and precise detection of plagiarized source code files in spite of the application of mutations is an important factor to consider in an evaluation of its effectiveness, and attempts to improve this aspect of an engine's performance are often the source of entirely new similarity engines and corresponding theoretical innovation [5, 6].

The increasing prevalence of programming assignments in first-year engineering coursework marks a corresponding increase in source code plagiarism outside the confines of computer science departments. The nature of programming assignments in the context of engineering education presents unique challenges for similarity engines developed with lengthy, statically typed programs in mind. This paper is interested in the application of its results in the first-year engineering education setting, so it will focus on Python, a rapidly growing language that continues to see new uses in the first-year setting [7], and on the short submission files resulting from the comparatively low complexity of programming assignments given to first-year engineering students. Specifically, this paper will analyze Python programs of 50 or fewer lines.

This paper introduces three metrics employed over two distinct experiments as a means to quantify the ability of a similarity engine to maintain accuracy in spite of code mutations. The first of these is called the sensitivity preservation metric. This metric indicates the robustness of an engine's similarity score to a specific mutation, by considering only similarity that is a result of actual plagiarism in a sense that will be rigorously defined later in this paper. Further, the sensitivity preservation metric is dimensionless and has natural interpretations independent of the specific similarity engine or mutation that it was computed for. In addition, it is normalized in such a way that enables cross-engine and cross-mutation comparisons. The second metric used in this paper's experiment is called detection precision. It measures the indices of each true-positive detection in the context of the similarity engine's well-ordered list of matches. One final metric, called the result incompleteness metric, is used to add additional context to precision results by indicating if a similarity engine failed to identify all instances of plagiarism for a given mutation.

Combining these factors naturally leads to this paper's research question: Which source code similarity engine is best equipped for analysis of short Python programs in terms of sensitivity preservation and precision over source code mutation?

2 Background

Source code similarity engines have been in active use for decades, with mutation resistance being a subject of research for just as long [8]. The number of available similarity engines has

increased considerably [9], solidifying the need for performance metrics that are both objective and effective.

Both qualitative and quantitative analyses of similarity engine performance have been thoroughly explored in literature. Many studies have computed the well known F-Measure [10, 11], or precision-recall metrics [12, 11], for various similarity engines. While these experiments employ well-established metrics to provide valuable insight on similarity engine performance with respect to generalized data, either organic or constructed, this paper is focused on the specific performance characteristic of mutation robustness. Little information about this performance characteristic can be gained from studies that process existing datasets with unknown mutation frequency and topology.

Little is known about similarity engine mutation resistance in the narrow scope of engineering education. Previously, we presented *Application of Source Code Plagiarism Detection and Grouping Techniques for Short Programs* [13], which for the remainder of this paper will be known as SPPD (Short Program Plagiarism Detection). Several design decisions for SPPD were made with the goal of both increasing detection sensitivity and increasing mutation robustness specifically for the types of short source-code submissions of interest to this study. Preliminary qualitative results comparing the mutation robustness of SPPD and MOSS were presented. These initial results were positive for SPPD, but their scope and room for interpretation were limited. One aim of this paper is to quantify these findings, and to provide new results comparing the performance of SPPD to other similarity engines.

Similarity engine mutation robustness is not an entirely unexplored domain, and while results are limited, multiple relevant works were identified. First is the work of Hage *et al.* [14], which provided the initial inspiration for the experiment conducted in this paper. Just as in this paper, Hage *et al.* perform a quantitative analysis on similarity engine robustness to single mutations, but their results do not address this paper's research question for two reasons. The dataset consists of pairs of two Java source files per submission [14, Sec. 5.1], rather than a single short Python file. Additionally, the metric used to measure mutation robustness is raw score produced by the target similarity engine. That metric is insufficient for addressing this paper's research question because this paper aims to make comparisons across various similarity engines. Specifically, Hage notes: "... when tool X scores higher than tool Y for a certain version, this does not necessarily mean that tool Y is more sensitive to this kind of attack than tool X [14, Sec 5.1, pp. 9]". The metric developed in this paper is more sophisticated than the one used by Hage *et al.*, and does capture cross-engine relative performance. In Hage *et al.*, an analysis of the accuracy of top matches is performed. This paper expands on that type of experiment and adjusts it for the research question at hand by instead investigating the plagiarism detection precision of the top results with respect to mutation. Mutations were not considered for the "Top-N" experiment in Hage *et al.* [14].

Second is the work of Cheers *et al.* [15], another quantitative analysis of similarity engine mutation robustness. The use of automated tools enables Cheers *et al.* to test a substantially greater number of mutations and samples when compared to this work. However, the results in Cheers *et al.* are unable to answer this paper's research question because the purpose of Cheers *et al.*'s work is to measure similarity robustness in isolation, and does not attempt to evaluate detection correctness [15, Sec. 4.1]. The gap between the work of Cheers *et al.* and this paper's research question is filled by the development of the sensitivity preservation metric that considers

only the similarity robustness that is a result of correctly detected plagiarism. Additionally, Cheers *et al.* note that "...the interpretation of the second comparative metric poses a threat to the validity of results in this work. As discussed, it is not a normalized measure that can be used to compare SCPDTs [similarity engines] on different data sets. Identifying a 'universal' normalized comparison metric is outside of the scope of this work [15, Sec. 8.3, pp. 57]". This paper contributes such a universal normalized comparison metric. One major focus of Cheers *et al.* is the layering of multiple mutations, often by a nondeterministic method [15, Sec. 5.0], which is beyond the scope of this work.

While these works present insightful results regarding similarity engine mutation robustness, they were insufficient for addressing this paper's research question. Therefore, the development and execution of a new experimental procedure was required.

3 Methodology

The following procedure should enable replication of the results discussed later in this paper, with the exception of the results for SPPD. Unfortunately, the reference implementation of SPPD used for this experiment is currently unpublished, and an alternative implementation based solely on SPPD's publication will likely be different enough from the reference implementation to produce notably disparate results in this highly sensitive experiment. The authors continue to investigate means to make SPPD available to those who need it. On account of copyright restrictions, the dataset used in this paper cannot be published, so replication will require the generation of a new dataset using the procedure described in Section 3.2.

3.1 Similarity Engines

First, the similarity engines for comparison, other than SPPD, were selected. In order to limit the complexity of the experiment, the number of alternate similarity engines for comparison was limited to five. Future work may introduce automated tooling aimed at reducing the complexity of conducting this experiment, enabling the comparison of additional similarity engines. Inclusion criteria for similarity engines consisted of support for the Python programming language, support for direct file-to-file comparisons, and relevance. Programming language support was easily discernible from tool descriptions. Relevant similarity engines were selected by two metrics: the most commonly cited engines in comparison and usage reports, as well as the engines most easily accessible by internet search. These two metrics capture both historically important engines and engines that currently have high accessibility. It is important to consider both of these relevance indicators, because historically important engines are likely still in common use at academic institutions, while highly accessible engines are likely to increase in populatity by acquiring a userbase of institutions looking to get started with plagiarism detection. The top-five similarity engines by number of citations in published works are JPlag, MOSS, Plaggie, and SIM-Grune [9]. As of February 2022, performing an internet search using the Google search engine for the term "source code plagiarism detection" yields many results that point to specific similarity engines. The first of these results is for Codequiry and the second is for AC2, taking third and fifth place in the list of all results respectively. Performing the same search with the Bing search engine yields Codequiry again, this time as the top result overall, and Copyleaks as the second result. The

Engine	Resolution	Exclusion Reason
JPlag [17]	Included	
MOSS [16]	Included	
Sherlock-Warwick [18]	Included	
Codequiry [19]	Included	
AC2	Included	
Plaggie [20]	Excluded	Does not support Python language.
SIM-Grune [21]	Excluded	Does not support Python language.
Copyleaks	Excluded	Does not support file-to-file comparisons.

Table 1: Similarity engines evaluated for inclusion.

considered similarity engines and their resolutions are given in Table 1. The following is a brief description of each included engine.

1. SPPD: This similarity engine was designed specifically for high sensitivity and mutation resistance on short code samples like the subjects of this paper's experiment. Its core technology is closely related to MOSS, but it greatly improves over simple *n-gram* hash counting through the use of frequency tables and multivariate methods [13].
2. MOSS: This similarity engine is based on *n-gram* hash matching and counting, and was novel in its introduction of a robust hash selection algorithm, known as Winnowing, with theoretical lower bounds on sensitivity [16].
3. JPlag: This engine leverages the Greedy String Tiling algorithm for measuring similarity by searching for the longest contiguous matches, marking them, and iteratively repeating to identify shorter, less important matches [17].
4. Sherlock-Warwick: This similarity engine traverses a pair of files and records "runs", or lines that are common between the two files with some acceptable number of anomalies. Files are compared by identifying runs with maximum length [18].
5. Codequiry: Codequiry is a commercial similarity engine that purports to be "the most advanced plagiarism detection solution" [19]. Despite it being the most accessible engine by internet search, it has been scarcely mentioned in published works. No quantitative analyses of similarity engine performance that included Codequiry were identified.
6. AC2: No published works mentioning AC2 were identified, however, this similarity engine generates similarity scores by analyzing the compressibility of each pair of files. The effectiveness of a compression algorithm is dependent on the entropy of the data to be compressed, so the actual similarity of files in a pair is positively correlated with that pair's compressibility.

3.2 Dataset

One requirement for the experimental procedure was the manual introduction of plagiarized pairs into a source population that otherwise contained no plagiarism. Granular control over the precise

Criterion	Justification
Python programming language.	Study criteria.
Length between 1 and 50 lines.	Study criteria.
No non-standard module imports.	Leads to uninteresting methods of solution.
No additional non-sorting functionality.	Assume assignment is to implement sorting only.
Top level members only.	Typical first-year curricula lack OO Python.
English identifiers and strings.	Dataset uniformity.

Table 2: Dataset inclusion member criteria.

number of plagiarized pairs and mutations applied to those pairs was necessary to ensure that the results could not be contaminated by unnoticed plagiarism in the source dataset.

This constraint immediately eliminated the possibility of using an organic dataset, such as the submission sets from first-year engineering courses readily available at the authors' institution, or public datasets that have been manually reviewed and tagged for plagiarism. Although datasets such as these have been historically used for evaluating similarity engine performance, it is not possible to ensure that every true positive instance of plagiarism in such uncontrolled datasets has been accurately identified. Rather, a procedure was developed for constructing a synthetic dataset.

First, the type of program to be included in the dataset was selected. It was determined that all selected source code files should attempt to implement the same functionality, simulating the type of per-assignment uniformity observed in real submission sets. The quicksort sorting algorithm was selected for this purpose on account of desirable properties:

1. The quicksort algorithm is popular, and Python implementations are plentiful.
2. When implemented concisely, the algorithm requires very little code. One correct implementation sampled in this experiment consisted of only seven lines.
3. Possible implementation strategies are diverse enough to enable interesting similarity engine results.

GitHub, an internet provider of source code repository hosting and collaboration tools, was selected as the source of the quicksort implementations for the population dataset. GitHub's "advanced search" functionality was utilized to search for code matches including the target keywords and filtered to the Python language. A number of inclusion criteria were created to encourage a similar level of uniformity in the constructed dataset as what would be expected in typical assignment submissions. The inclusion criteria and associated justifications are given in Table 2.

Each search result was sequentially evaluated on the inclusion criteria, and discarded if any criterion was unsatisfied. This process was continued until the source population of thirty files was constructed.

Metric	Population (n=30)	Sample (n=5)
Mean Lines	23.06	18.20
Median Lines	21	18
Mean Complexity	4.80	5.25
Median Complexity	4.83	5.61

Table 3: Representative sample metric preservation.

Mutation	Resolution	Novak *et al.* Ranking [9]
Identifier Modification	Included	1st
Block Reorder	Included	2nd
Operator Reorder	Included	7th
Code Insertion	Included	5th
Comment Modification	Excluded	3rd
Code Formatting	Excluded	4th
Control Structure Replacement	Excluded	6th

Table 4: Mutation reference frequency.

3.3 Sampling and Mutation

In order to limit the complexity of the experiment, a subset of files from the dataset was selected to be the target of mutation and comparison. A representative sample of five target files was constructed from the population of thirty initial files. To prevent the introduction bias favoring specific similarity engines, the sample was composed such that each of four metrics, mean line count, median line count, mean complexity, and median complexity, were approximately conserved. The complexity metric was given by the Halstead difficulty measure [22]. A rigorous optimization procedure was not used for this process. Rather, files were added and removed from the sample until further changes yielded no improvement. The results of the metric preservation between the population and representative sample are shown in Table 3. Future work could develop an automated tool for rapidly generating appropriate samples.

A representative set of five mutations was selected for testing. As discussed previously, these constitute potential strategies a dishonest student might employ in an attempt to evade detection by similarity engines. The factors for selecting mutations included a preference for a high frequency of references to the mutation in literature, and a preference for mutations that were applicable to the dataset. Mutation reference frequency rankings are given in Table 4. The following is a list of the selected mutations accompanied by the procedure used to mutate the sample:

1. Identical Copy: In addition to serving as a baseline measurement for the sensitivity preservation analysis that will be described in the next section, some students submit plagiarized work without any modifications. Therefore, evaluating performance in this scenario should not be overlooked. On account of the fact that some similarity engines refuse to process byte-for-byte identical files, a minimally invasive modification was made to each file in the sample: the addition of a single whitespace character at the file's tail. For

the engines selected above, only AC2 has this property. While not a traditional mutation, the identical copy case will be evaluated with the same methods as other mutations, so it is included here for clarity. For the remainder of the paper, "non-substantive mutation" will refer to this mutation, while "substantive mutation" will refer to any other mutation.

2. Identifier Modification: In the context of this paper, an identifier is any source code component that is the target name of a "name binding operation", as defined in the Python Language Reference section 4.2.1. These include variable assignments, function definitions, function parameter definitions, for loop header definitions, and many other constructs not relevant to this dataset. For each file in the sample, the name referenced in the name binding operation was replaced with the output of an injective function applied to the initial name. The codomain of this function consisted of two-character alphanumeric codes.

3. Block Reorder: Code reordering involves changing the order of components in the source code file in a way that does not affect program behavior. Candidates for reordering are logically independent lines, such as two unrelated variable assignments with no side effects, or code rendered independent by the parser, such as the order of function declarations. This mutation was applied to each sampled file in a two-phase process: a major phase and a minor phase. In the major phase, an attempt was made to reorder substantial blocks of code, typically entire functions. This was possible for three of the five sampled files, but impossible for the two shortest files that contained only one function each. In the minor phase, logically independent lines of code were reordered at every possible opportunity, but no modification to any program logic was made to facilitate reordering. It is important to note that only truly independent lines were reordered such that the program behavior remained completely unmodified. However, it is perhaps unreasonable to assume that typical students will make those evaluations correctly, and may inadvertently reorder a greater number of lines than this procedure would for the same source file.

4. Operator Reorder: In this common mutation, the order of operands in arithmetic and boolean expressions are changed, and the corresponding operator is changed if appropriate. For the sampled files, only commutative and associative arithmetic operations were considered for modification, in which case the order of the operands was changed but the operator itself was not modified. For boolean expressions, the order of operands was changed to the greatest extent possible without adding additional tokens, including parentheses, such that the meaning of the expression remained unchanged. In cases where a less than or greater than operator was used, and the operands were swapped, the operator was correspondingly changed to maintain the same logical expression.

5. *MOSSAD* Code Insertion: This mutation involves the insertion of small amounts of "dead code", or syntactically valid code that is interpreted at runtime but has no effect on the behavior of the program. This paper refers to this technique as the *MOSSAD* Code Insertion mutation because the procedure for the technique is derived from a paper by the same name [23]. This technique was developed specifically to evade detection by MOSS, and several design decisions for SPPD were made with the precise intention of mitigating the effectiveness of this mutation. Note that while the specific implementation described in the *MOSSAD* paper only supports statically compiled languages, as is necessary for the semantic preservation test, the technique itself is generally applicable if constructs that alter

program semantics can be identified. The mutation was applied to each sampled file by inserting a randomly selected dead code snippet, including constructs such as unused variable assignment, mutation of previously defined unused variables, and arithmetic or logic operations on the same, at a randomly selected interval between zero and six lines. The interval was randomly selected again after each insertion. Overall, this mutation most closely resembles an enhanced version of the $MOSSAD_{NONDET}$ variant [23].

Several mutations with a frequency higher than some of those selected were rejected because they lack applicability to the constructed dataset:

1. Comment Modification: Comments were not included in the dataset.
2. Code Formatting: Due to the restrictive nature of whitespace usage in the Python language, this mutation is uninteresting.
3. Control Structure Replacement: The short nature of the programs in the dataset caused the number of distinct control structure replacements that could be made for any given file to be relatively limited, reducing the effectiveness and applicability of this method.

The five sampled files were identically duplicated five times. Each accepted mutation listed above was applied to each of the files in one of the replicas. Note that no replica was subjected to multiple mutations, however, this possibility presents an opportunity for future work.

3.4 Data Collection

After the completion of the mutation step, it was necessary to reintroduce the mutated files into the population from which they were sampled. The initial population of thirty files was replicated five times, and the input file set for similarity engine evaluation was created by combining each of the five mutated samples with a copy of the unmodified original thirty files, creating the five final similarity engine input sets of 35 files each (n=35), for which similarity data would be collected. The precise procedure for gathering similarity results from each engine is beyond the scope of this paper, however, the following is a list of non-default settings applied to each similarity engine and the version of the engine tested, if applicable, that should be used when replicating this paper's results.

1. SPPD: The Python language setting was selected. This implementation is currently unpublished and does not have version information.
2. MOSS: The Python language setting was selected. No version information for MOSS was identified.
3. JPlag: The Python language setting was selected. Version v3.0.0 released on 4 December 2021 was tested.
4. Sherlock-Warwick: The "No whitespace (For other syntax families)" setting was enabled. The "other" in this setting refers to a previous setting specific to languages in the Java/C/C++ syntax family. Therefore, "other" should be interpreted to mean "non-Java/C/C++ family languages", of which Python is a member. Version 5 released in 2003 was tested.

5. Codequiry: The Python language setting was selected. The "Group Similarity Only" mode was selected, which disables web searching functionality and relies only on direct file-to-file comparisons. Version v1.7 released on 20 March 2021 was tested.
6. AC2: Default settings were used. Version 2.1.5 released on 1 February 2022 was tested.

Using the above settings, similarity data were gathered for each combination of similarity engine and input dataset. The set of all pairs of files identified as matches, as well as all associated similarity scores, were recorded for use in the two analysis procedures described in the following sections.

3.5 Sensitivity Preservation Analysis

As previously discussed, sensitivity preservation refers to the proportion of similarity remaining after the application of a mutation, considering only similarity that can be explained by actual plagiarism. This section will first formalize the definition of the sensitivity preservation metric, and subsequently, explain its properties and justify the utility of those properties. Finally, the specific procedure used for computing the metric in this experiment will be given. Note that, while the following definition is included for completeness, it is not necessary for computing the metric, interpreting the results, or understanding the rest of the paper.

Definition 1. (Sensitivity Preservation Function). *Let S be the set of all files in a target sample and $T \subseteq S \times S$ the set of all plagiarized pairs. Given a match scoring function $s : S \times S \to \mathbb{Q}$ and an identical similarity function $i : S \to \mathbb{Q}$, the sensitivity preservation function on a similarity engine result set, $p : \mathscr{P}(S \times S) \to \mathbb{Q}$, is given by:*

$$p(R) = \frac{1}{|T|}\left(\sum_{(t_\alpha,t_\beta)\in T} \frac{\max(\{s(r_\alpha,r_\beta) \mid (r_\alpha,r_\beta) \in R \wedge \{r_\alpha,r_\beta\} = \{t_\alpha,t_\beta\}\})}{\max(i(t_\alpha),i(t_\beta))} \right. \tag{1}$$

$$\left. - \sum_{(t_\alpha,t_\beta)\in T} \frac{\max(\{s(r_\alpha,r_\beta) \mid (r_\alpha,r_\beta) \in R \wedge \{(r_\alpha,r_\beta),(r_\beta,r_\alpha)\} \cap T = \varnothing \wedge \{r_\alpha,r_\beta\} \cap \{t_\alpha,t_\beta\} \neq \varnothing\})}{\max(i(t_\alpha),i(t_\beta))}\right)$$

The above function computes the sensitivity preservation metric for a similarity engine's result set, the set of all true-positive pairs present in the mutated sample, a function mapping a pair of files to its similarity score, and the expected similarity for each true positive in the sample assuming that no mutations were applied. This last input is known as the identical similarity map for true positives and was conceptually presented in the "identical copy" mutation procedure. The precise definitions of the match scoring function and the identical similarity function are engine-dependent, so they will be presented later in this section after the theoretical framework is solidified.

At a high level, this function produces a metric for a specific combination of similarity engine and mutation, and works by individually considering each plagiarized file that was incorporated into the input data set, prepared in Section 3.4, for the given engine mutation pair. For each

plagiarized file, the difference in the similarity scores between that of the top true-positive match, which will be known as the mutation robustness metric, and the score of the top false-positive match, which will be known as the false-positive adjustment, is taken. This difference represents the amount of similarity in the top detection involving the given plagiarized file that can be explained by a true-positive match (i.e. a match between the plagiarized file and the initial file it was replicated from in the sampling procedure, Section 3.3). This difference provides useful information in the context of the specific similarity engine it was generated for, but a normalization step is required before it can be interpreted in the context of, or compared against, scores from other similarity engines. The score is normalized by scaling the difference such that it is dimensionless and independent of the similarity engine it was computed for, which is necessary for applications such as by serving as a comparison point for the scores of other engines. Finally, these normalized scores are averaged for all plagiarized files in the given engine mutation pair, resulting in the final metric.

The sensitivity preservation metric has natural interpretations that make it inherently useful for evaluating the performance of similarity engines subjected to mutated files. For some combination of similarity engine and mutation, sample interpretations of possible sensitivity preservation metrics are provided.

- 1.0: No similarity was lost as a result of the mutations, and all reported similarly in plagiarized files can be attributed to true-positive cases of plagiarism.
- 0.5: After the mutations, only 50% of pre-mutation similarity in plagiarized pairs can still be attributed to true-positive cases of plagiarism.
- 0.0: After the mutations, none of the remaining similarity in plagiarized pairs can be attributed to true positive cases. The likelihood that the top result for a plagiarized file is a true-positive reference to a plagiarized pair is 50%.
- -0.1: plagiarized files are now, on average, more similar to typical members of the population than the files they were initially copied from. On average, false-positive matches for plagiarized files have higher similarity scores than true-positives.

The sensitivity preservation metric defined above is a novel development of this paper. As discussed previously, sensitivity preservation refers to an engine's ability to retain sensitivity to true-positive cases of plagiarism over mutation. Sensitivity preservation can be expressed as the proportion of the normalized mutation robustness of plagiarized files that can be explained by true-positive detections in the similarity engine's results. Therefore, when a similarity engine's result set contains no false positives, this paper's sensitivity preservation metric and the normalized mutation robustness metric are exactly equivalent. For the purposes of answering the research question posed by this paper, the sensitivity preservation metric is superior to a mutation robustness metric that was utilized in two related previous experiments identified in literature review, in two distinct ways.

1. Identity normalization: Conceptually, identity normalization scales the preserved sensitivity of a match containing a plagiarized file by a factor inversely proportional to the identical similarity of the plagiarized file in the match. Identical similarity of a plagiarized file for some similarity engine is obtained by the engine's similarity score of the pair containing the

initial, unmutated file and that same file with the non-substantive "identical copy" mutation applied. Consequently, the final sensitivity preservation metric is some proportion of the identical similarity for the plagiarized file involved in the match, specifically the ratio of the amount of remaining post-mutation similarity that can be explained by actual plagiarism to the identical similarity. Therefore, the identical similarity of a plagiarized file represents the maximum possible similarity between that file and any other, which could also be interpreted as the upper bound for the remaining similarity after a substantive mutation. If an engine is completely robust to a mutation, the post-mutation similarity score for a plagiarized pair will match the identical similarity of the corresponding plagiarized file. This feature of the sensitivity preservation metric serves a dual purpose.

First, it enables cross-run and cross-dataset metric consistency for similarity engines that produce an inconsistent or unbounded similarity score. Most similarity engines generate a similarity score in the form of a percentage that ranges from 0% to 100%, and for these engines, this particular identity normalization benefit is nullified. However, not all engines are so well-behaved. For example, data collected in this experiment indicate that Sherlock often produces similarity scores in excess of 200%, with no clear upper bound. This is made more confusing by the fact that Sherlock uses the "percent" unit in its reporting, but it is entirely unclear how a result of "files are 200% similar" should be interpreted.

MOSS is also ill-behaved, but in a slightly harder-to-detect way. Hages *et al.* suggested that MOSS never returns a similarity score of 100%, even for completely identical files [14]. The data collected as a result of the experiment in this paper support this conjecture. It appears that, although complete certainty is impossible since MOSS is closed-source, the maximum similarity score ever returned by MOSS is 99%. Without normalization, this scoring artifact introduces a bias that unfairly disadvantages MOSS in performance comparisons. This is because, when MOSS returns a result of 99%, it is indicating that the returned pair is maximally similar. MOSS could not possibly indicate a pair with higher similarity, since it never returns a result of 100%. Therefore, the result should be interpreted in the same way as a result from a different engine indicating maximal similarity, typically by returning 100%. Stated another way, when MOSS returns a result of 99%, it has the same meaning as a result of 100% from JPlag. It would not be appropriate to interpret MOSS similarity scores in a context where there is an implicit assumption that MOSS could have returned a 100%. Hages *et al.* identified this bias, but did not attempt to correct for it, and Cheers *et al.* did not consider MOSS for robustness comparison [14][15]. Even though this paper's focus centers around mutation robustness and associated literature, it is important to note that every score-based performance comparison featuring MOSS should include a correction for this bias. This paper's identity normalization serves as the appropriate correction.

The second benefit of the identity normalization feature is the generation of a dimensionless metric that is suitable for cross-engine comparison, regardless of the scoring technique used by the individual similarity engines. Hages *et al.* measures similarity robustness in units of absolute percent similarity, which is difficult, or perhaps impossible, to interpret in an unbiased way. This paper's sensitivity preservation metric may be freely interpreted outside the context of a specific similarity engine, as demonstrated by the sample interpretations

given earlier, or compared to the metrics of any other engines without fear of engine-specific scoring bias.

2. False-positive adjustment: Previous related experiments have not penalized mutation robustness scores for preserving similarity that cannot be explained by true-positive cases of plagiarism. Although the methods used in Cheers *et al.* are in many ways similar to those presented in this paper, Cheers' goal is to quantify only the preservation of similarity over mutations, and not to make any conclusions about the accuracy of engines' detections [15, Sec. 4.1]. However, the practical value of a mutation robustness result is minimal if it fails to consider whether the similarity was preserved due to actual mitigations of mutations, or on the contrary, if the similarity was preserved due to oversensitivity of the similarity engine in general, resulting in a high preservation metric at the cost of high similarity results for files that do not constitute actual plagiarism.

   Consider the hypothetical extreme case of a similarity engine that simply returns a similarity score of 100 for every pair of files, regardless of their contents. Under the methods used in both Cheers *et al.* and Hages *et al.*, this hypothetical engine would always produce the maximum possible mutation robustness score, convincingly outperforming every other engine in all mutation robustness tests. By contrast, under the methods presented in this paper, the false-positive adjustment term would precisely cancel with the mutation robustness score for all tests performed on the hypothetical engine, producing a sensitivity preservation score of exactly zero. This result is reasonable: none of the preserved similarity score can be attributed to the actual preservation of sensitivity to true-positive cases of plagiarism, since every false-positive match presents the same similarity as every true-positive match. Additionally, the sensitivity preservation score of zero indicates no selection bias towards either false-positive or true-positive matches, which can be interpreted as a complete lack of information about the likelihood of plagiarism. This aligns with the observation that, when all similarity scores are identical, attempting to select plagiarized pairs from similarity scores alone would be an exercise in pure chance. Zero is the only sensitivity preservation score for which this property holds. It is for these reasons that this paper quantifies only the portion of mutation robustness that is also explained by true positive cases of plagiarism, by means of the sensitivity preservation metric.

The last prerequisite necessary for the computation of the sensitivity preservation metric is the precise definition of the match scoring function. Some similarity engines require a specialized form of the match scoring function, while the remaining share a generalized definition. The specific functions used for each tested engine are given as follows.

1. MOSS: The match scoring function for MOSS requires special consideration because MOSS is the only similarity engine considered by this paper that produces a different similarity score for each of the two files in a similarity detection. The match scoring function $s : S \times S \to \mathbb{Q}$ is given by the maximum of the two scores presented for $(s_0, s_1) \in S \times S$, or 0 if it is not present on the match list.

2. AC2: Similarity scores in AC2 represent a conceptual distance between files, and therefore decrease as similarity increases. Since this procedure is designed for scoring functions that monotonically increase as similarity increases with a minimum value of zero, a nonlinear

transformation is necessary. A reciprocal was selected for this purpose. Therefore, the match scoring function $s : S \times S \to \mathbb{Q}$ is given by one divided by the numeric similarity score displayed by AC2 for $(s_0, s_1) \in S \times S$, or 0 if it is not present on the list.

3. All others: Assume the match scoring function $s : S \times S \to \mathbb{Q}$ is given by the numeric score shown for pair $(s_0, s_1) \in S \times S$ on the similarity engine's "match list", or 0 if it is not present on the list.

An abstract overview of the method used to compute the value of the sensitivity preservation function will be presented, followed by the exact procedure that was used in this paper to compute the sensitivity preservation metrics from the result sets generated in the previous section. The steps in this procedure are logically equivalent to the definition given at the beginning of this section.

To begin the overview, several variables will be defined for enhanced readability.

$p \in \mathbb{Q}$: The mutation robustness score, a rational number.
$f \in \mathbb{Q}$: The false-positive adjustment, a rational number.
$i \in \mathbb{Q}$: The identical similarity score, a rational number.

The following quotient shall be known as the sensitivity preservation subscore:

$$\frac{p - f}{i}.$$

The complete sensitivity preservation score for some combination of similarity engine and mutation is the average of all sensitivity preservation subscores when computed for each plagiarized file in the input dataset that corresponds to the aforementioned similarity engine mutation combination as described in Section 3.4.

Finally, all the tools necessary for carrying out the computation of a similarity engine's sensitivity preservation metric with respect to each mutation have been established. The identical similarity function must be defined in terms of mutation robustness metrics for the identical copy mutation, so it cannot be defined until the computation is underway. The following steps were repeated for each set of data collected in the previous section, corresponding to a unique combination of similarity engine and mutation. It was imperative that, for each engine, this procedure was carried out first for the identical copy mutation, as that would be used during each subsequent computation for the remaining mutations.

If the next mutation to evaluate was "identical copy", for each plagiarized file, p and i were set to the output of the appropriate match scoring function applied to the top true-positive result including the current plagiarized file in the collected data (the plagiarized file is associated with the file it was initially derived from in a match), and f was set to the output of the appropriate match scoring function applied to the top false-positive result including the current plagiarized file in the collected data (the plagiarized file is associated with a file other than the one it was initially derived from in a match), or zero if none exists. Then, the sensitivity preservation subscore as defined above was computed for each plagiarized file and averaged to find the sensitivity preservation score. Additionally, the identical similarity function was defined as the map from a plagiarized file to its associated i value for the remainder of the batch.

If the next mutation to evaluate was not "identical copy", for each plagiarized file, p was set to the output of the appropriate match scoring function applied to the top true-positive result in the collected data, or zero if it did not exist, and f was set to the output of the appropriate match scoring function applied to the top false-positive result including the current plagiarized file in the collected data, or zero if it did not exist, and i was set to the result of the identity similarity function defined in a previous iteration applied to the current plagiarized file. Then, the sensitivity preservation subscore, as defined above for each plagiarized file, was computed and averaged to find the sensitivity preservation score.

This procedure was carried out for each collected dataset, and the final results were obtained in the form of a sensitivity preservation metric for each combination of similarity engine and mutation. Finally, these results were arranged in a tabular visualization.

3.6 Precision Analysis

The second and less complex method of analysis performed on the collected data quantified the precision, or the rate at which mutated true-positive matches appeared before false-positive matches of either plagiarized or non-plagiarized files in similarity results, for each similarity engine. This fills an important gap left by the first analysis.

From the results of the experiment, it appears that sensitivity preservation is highly correlated with detection precision, however, it would not be impossible for an engine to produce high sensitivity preservation scores by placing all true-positive instances of plagiarism above all false positives that involved any of the plagiarized files, while still placing many unrelated false-positives above the true-positives. False positives that do not involve a plagiarized file cannot affect the sensitivity preservation score, and that is why additional analysis is necessary.

First, the two metrics relevant to this analysis will be defined. The first metric transforms a result set into a set of indices with cardinality less than or equal to the number of true positives incorporated into the sample. Each index represents the position of a true positive match in the well-ordered set of results produced by the similarity engine.

Definition 2. (Positive Indexing Function). *The positive indexing function on a similarity engine result set, $z : \mathscr{P}(S \times S) \to \mathbb{Q}$, is given by:*

$$z(r_\alpha, r_\beta) = \{\textit{first index of } (r_\alpha, r_\beta) \textit{ or } (r_\beta, r_\alpha) \textit{ in engine match list} \mid (r_\alpha, r_\beta) \in T \vee (r_\beta, r_\alpha) \in T. \tag{2}$$

Additionally, this analysis requires information about the completeness of each result set. A result set is complete if and only if, for all mutated files, a detection associating the mutated file to the file it was originally derived from is present in the set. Formally, this metric is given by the result incompleteness function, which is a function that maps a result set to 1 if the set is incomplete or 0 otherwise, and is defined as follows:

Definition 3. (Result Incompleteness Function). *The result incompleteness function $c : \mathscr{P}(S \times S) \to \{0, 1\}$ is given by:*

$$c(R) = \min(|T| - |\{(r_\alpha, r_\beta) \in R : (r_\alpha, r_\beta) \in T \vee (r_\beta, r_\alpha) \in T\}|, 1). \tag{3}$$

The results for this analysis were generated by reviewing the result set for each combination of similarity engine and mutation generated in the data collection step. The collected data for the Codequiry similarity engine was excluded from this analysis because Codequiry does not present a natural ordering for its returned matches. For each mutation of every other similarity engine, the indices of each true-positive match, matches that included a plagiarized file, and the value of the result incompleteness function, a boolean value that is 1 if and only if one or more of the true-positive pairs in the sample is entirely missing from the result set, were recorded.

4 Results

The results in this section should be interpreted only in the context of the study: short Python programs. This paper presents no evidence that these results can be extrapolated to longer programs, or to programs written in different languages. Future work is needed to analyze these cases.

4.1 Sensitivity Preservation Results

The sensitivity preservation metrics computed for each combination of similarity engine and mutation are visualized in a highlight table, shown in Figure 1. The following includes noteworthy observations and interpretations for the sensitivity preservation results of each similarity engine:

	Engine					
Mutation	SPPD	MOSS	JPlag	Sherlock	codequiry	AC2
Identical Copy	0.86	0.69	0.70	1.00	0.76	0.76
Identifier Modification	0.45	0.69	0.70	0.00*	0.76	0.76
Block Reorder	0.54	-0.05*	0.24*	0.37*	-0.08*	0.22
Operator Reorder	0.30	0.06*	0.70	0.33*	-0.16*	0.08
MOSSAD (Code Insertion)	0.47	0.13	0.03*	0.57	-0.14*	0.30

Figure 1: Match sensitivity preservation

*Failed to identify all true-positive cases

1. SPPD: SPPD has worse sensitivity preservation when compared to JPlag over operator reorder, or Sherlock over code insertion, however, SPPD has the highest sensitivity preservation overall, indicating that, in the average case, it should preserve sensitivity more effectively than any one of the other engines.
2. MOSS: Although it is the most commonly used similarity engine, MOSS takes a clear second to last place ahead of Codequiry. MOSS was the only engine other than Codequiry that managed to obtain a negative sensitivity preservation score for any mutation. For the block reorder mutation, the top result involving a plagiarized file was more likely than not to be associated with a file other than the one it was initially derived from.
3. JPlag: JPlag's sensitivity preservation was satisfactory in general, and exceptional in regards to operator reorder, however, its consistency suffers from comparatively poor performance over code insertion, even to MOSS.
4. Sherlock-Warwick: If Sherlock had been able to capture identifier insensitivity, it would have been the clear winner of the tested engines. Unfortunately, it appears to be completely sensitive to identifier mutation when analyzing Python code. Perhaps it is able to preserve sensitivity over identifier mutation for the languages for which it provides built-in support, such as Java or C++. The likely interpretation for the score of 0 is that Sherlock simply returned no matches.
5. Codequiry: Codequiry fully preserved sensitivity in the case of identifier modification, as did MOSS, JPlag, and AC2, however, its scores over the other three substantive mutations were exceptional in their substandard quality. For block recorder, operator reorder, and code insertion, Codequiry, on average, returned a substantially higher score for a false-positive match involving a plagiarized than that file's corresponding true-positive match. In all three cases, at least one true-positive match was completely missing.
6. AC2: AC2's sensitivity preservation is not exceptional, however, it takes a clear second place in terms of consistency. This makes sense when considering AC2's implementation. Internally, it uses only one metric and that is the compressibility of a given pair of files. Therefore, it is unsurprising that different types of mutations cause reduced variation in sensitivity preservation scores when compared to other engines.

An interesting observation applicable to four of the five engines is the absence of a perfect sensitivity preservation score for the "identical copy" mutation. These non-optimal results have a simple, but useful, interpretation. An identical copy score below 1.0 indicates that, even when a pair in the sample matches identically and scores the maximum possible similarity robustness metric, not all of that similarity can be attributed to plagiarism. This is possible due to a similarity engine's inclusion of false-positive results involving plagiarized files, even when those plagiarized files match identically with a different file. For example, if an engine has a sensitivity preservation score of 0.7 for identical copy, it's safe to assume that the result set included a false-positive match for that same plagiarized file with a similarity score 0.3 times the score of the identical pair.

This effect is a concrete example of the operation of the false-positive correction. Since, in that example, at least one pair had a similarity of 0.3 times the true-positive result, that portion of the measured similarity isn't useful for identifying plagiarism, since it was able to identify a

false-positive by definition.

4.2 Detection Precision Results

The data collected in Section 3.6 are visualized in a horizontal bar chart, shown in Figure 2. This chart displays the recorded metrics for each combination of similarity engine and mutation, grouped by engine. Each index is represented as a bar on the chart, with bars for the same engine and mutation overlaying each other. The result incompleteness metric was visualized by including a red highlight across the entire row of an engine mutation combination if the incompleteness result was equal to one for that combination.

The interpretation of this visualization is straightforward. The optimal result, known as perfect precision, will identify all five true-positive results in the top five returned result pairs. This can be seen as a stack of five bars of length one in a row. For every bar with a length greater than one, a false-positive result was included before the true-positive result associated with that bar. Finally, incomplete results can be identified by the red highlight and missing bars. These results have limited room for interpretation, other than observing that the results were incomplete, and the similarity engine was simply unable to detect all true-positive matches.

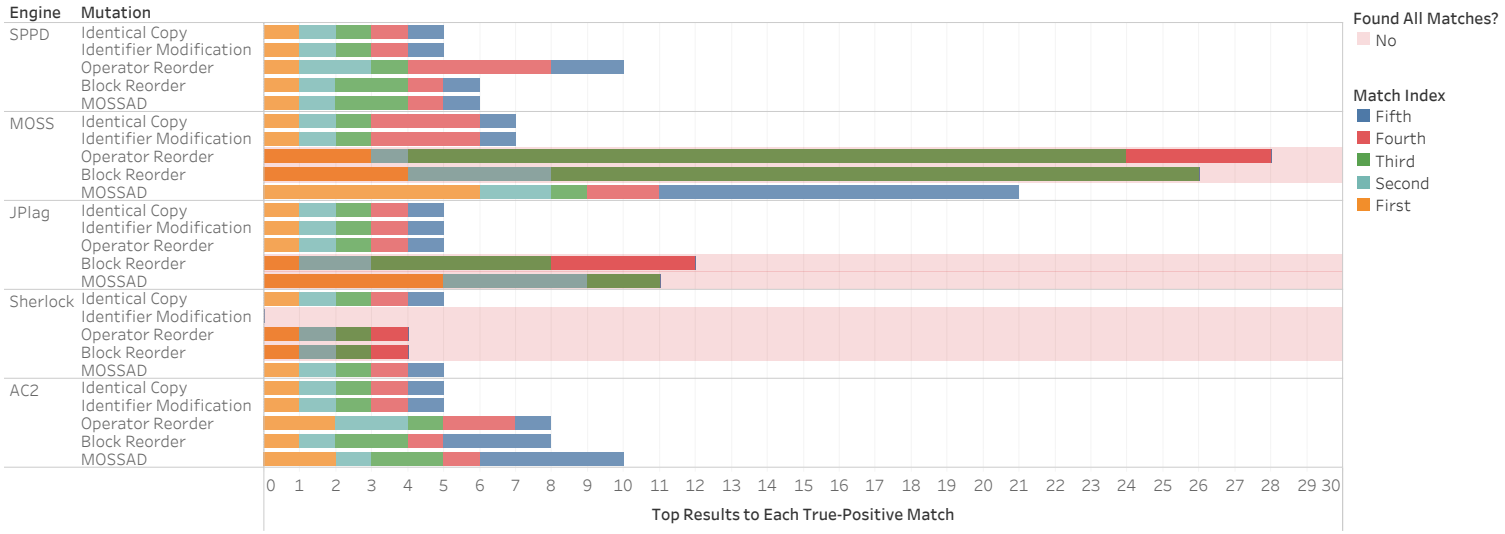

Figure 2: Top match precision results

1. SPPD: SPPD performed well, featuring perfect precision under identifier modification, and one false-positive between the second and third results for both block reorder and code insertion. It was most sensitive to the operator reorder mutation. Interestingly, SPPD has precisely the opposite performance characteristics of JPlag when considering only the last three mutations.

2. MOSS: As foreshadowed by the sensitivity preservation results, MOSS's overall performance was subpar, and MOSS was unable to maintain perfect precision under any mutation. Even in the case of identical similarity in the true-positive pairs, a false-positive appears between the third and fourth results. Under operator reordering, the mutation to

which JPlag was completely insensitive, MOSS placed the fourth true-positive result after a combined 25 false positives. Additionally, the fifth result was not detected.

3. JPlag: The performance of JPlag was perfect for three of the five mutations, however, its sensitivity to block reorder and code insertion generate numerous false positives prior to true positives in its results list, and it was unable to identify all plagiarized pairs. For each mutation, JPlag's performance was higher than at least one other engine.
4. Sherlock-Warwick: The result for Sherlock is unique in the sense that it was the only result with perfect precision for every mutation. In all cases, Sherlock never returned a true positive match after a false positive match. The problem, however, is that Sherlock failed to identify all true-positive cases of plagiarism for three of the five mutations. One possible explanation for this is Sherlock aggressively discarding high similarity pairs to reduce the chance of false positives to the greatest extent possible. The identifier sensitivity issue, as described above, presents here as the only empty bar. This result confirms the hypothesized cause of Sherlock's identifier modification sensitivity preservation score of zero: it returned zero true-positive and zero false-positive results for the plagiarized files.
5. AC2: The results for AC2 are promising. Although AC2's sensitivity preservation scores were clearly behind Sherlock, Sherlock's failure to identify all true positives in most cases clearly indicates AC2 to be the runner-up in this test.

4.3 Research Question

This paper's research question sought the similarity engine best equipped for the analysis of short Python programs with respect to sensitivity preservation and detection precision of mutated source code submissions. The first observation is that, unfortunately, no single source code similarity engine is superior to all others in terms of general mutations, even for the limited number of engines included in this experiment. Rather, some engines are superior in their robustness to one or more specific types of mutations, while performing worse than other engines for the remaining mutations. This result is not unexpected, and a detailed interpretation is provided in Section 5.3.

In general, the two similarity engines with a clear lead were SPPD and Sherlock. SPPD falls short in sensitivity preservation over code insertion and produces detections with lower precision than Sherlock. On the other hand, Sherlock is highly sensitive to trivial identifier modification and has slightly worse sensitivity preservation than SPPD over both block reorder and operator reorder. Additionally, it fails to detect all true positive matches in three out of five cases. SPPD appears to have the upper hand overall, but the lead is slight, and both tools have clear advantages and disadvantages.

5 Future Work

5.1 Analyze Leading Technology

What makes JPlag completely insensitive to operator reordering? Why is Sherlock least affected by code insertion? Answers to specific questions about the leading similarity engines could be

invaluable to the development of novel engines like SPPD, but these questions will need to be answered in future work.

5.2 Automated Testing Framework

Both the sample sizes and the number of variations used in this experiment were smaller than what is typically found in published works. Unfortunately, the effort involved in the manual data preparation and collection for this experiment was high. The addition of additional engines and mutations would make the scale of the experiment infeasible for a manual process. A potential avenue for future work involves the design and implementation of automated software tools designed for carrying out some or all of the procedures given in this paper. Specifically helpful would be automatic dataset generation, sampling, mutation, and an automated process for both running and collecting results from similarity engines. Implementation of these tools would enable the results in this study to be replicated on a much larger scale or in new contexts, including different programming languages or program lengths.

5.3 Similarity Factor Compromise

This paper presented the result that no single similarity engine is generally more robust to all forms of code mutations than all others. This result has an interpretation in the context of the internal operation of similarity engines. In some abstract sense, a similarity engine has to choose the source code attributes that it will consider when evaluating similarity. The specific selection made for any given engine entails some necessary sensitivity preservation compromises, since for each abstract factor included in the scoring process, mutations that affect that specific factor will have a negative effect on sensitivity preservation, and conversely, that factor would have a positive effect on sensitivity preservation for every other mutation. This compromise is inherent: if a similarity engine did not reduce its similarity score when a considered factor differed between the compared files, there would be no "room" for it to increase the score when the files did, in fact, match with respect to that factor. Note that an unbounded sensitivity measure is not a solution to this problem, but rather obscures the problem by increasing the relative scores of pairs that do not constitute plagiarism as an indirect result of the potential for the absolute similarity reduction implicated by the compromise. This paper's sensitivity preservation metric remains unaffected by this obfuscation as a result of the identity normalization step discussed in Section 3.5.

A clear example of this compromise effect is seen in SPPD's identifier modification sensitivity preservation score, shown in Figure 1. SPPD's design included the strategic decision to consider identifiers as a small but still relevant factor in overall similarity [13]. As a result, SPPD's sensitivity preservation with respect to the identifier modification mutation was substantially lower than the corresponding results for most other similarity engines. According to the results above, four of the six engines tested were completely insensitive to identifier modification. These results illustrate the disadvantages of SPPD's sensitivity to identifier modification, however, they do not provide any insight into the relative advantages of this inclusion. For example, it is unclear what portion of SPPD's sensitivity preservation for the other three substantive mutations can be explained by its inclusion of identifier modification sensitivity.

What are the compromises that must be made for the addition of any given factor, and in what

environments do the benefits of the added factor outweigh these compromises? The answer to this question could have profound implications for similarity engine research, but both the concrete definition of a factor as used here and the proposed analysis must be left to future work.

6 Conclusion

This paper has developed novel metrics for the measurement of a plagiarism detection tool's robustness to student code modifications and conducted an experiment that applies these metrics in the specific context of typical first-year engineering coursework. The results and their associated interpretations presented in this paper have immediate applicability to engineering assessment and are a satisfactory first step towards contextualizing the performance of plagiarism detection tools in the setting of engineering, rather than computer science, education. This paper prompts several avenues for future work, including the replication of these results with different datasets, engines, or mutations, the automation of some or all of the test procedures for enhanced throughput of the experiment, and analysis of the similarity factor compromise.

References

[1] M. Srikanth and R. Asmatulu, "Modern cheating techniques, their adverse effects on engineering education and preventions," *International Journal of Mechanical Engineering Education*, vol. 42, no. 2, pp. 129–140, 2014. [Online]. Available: https://doi.org/10.7227/IJMEE.0005

[2] M. O'Malley and T. S. Roberts, "Plagiarism on the rise? combating contract cheating in science courses," *International Journal of Innovation in Science and Mathematics Education*, vol. 20, no. 4, 2012.

[3] D. Starovoytova and S. S. Namango, "Viewpoint of undergraduate engineering students on plagiarism," *Journal of Education and Practice*, vol. 7, pp. 48–65, 2016.

[4] S. Manoharan and U. Speidel, "Contract cheating in computer science: A case study," in *2020 IEEE International Conference on Teaching, Assessment, and Learning for Engineering (TALE)*, 2020, pp. 91–98.

[5] H. Cheers, Y. Lin, and S. P. Smith, "Academic source code plagiarism detection by measuring program behavioral similarity," *IEEE Access*, vol. 9, pp. 50 391–50 412, 2021.

[6] H. Cheers and Y. Lin, "A novel graph-based program representation for java code plagiarism detection," in *Proceedings of the 3rd International Conference on Software Engineering and Information Management*, ser. ICSIM '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 115–122. [Online]. Available: https://doi.org/10.1145/3378936.3378960

[7] M. Wende, T. Giese, S. Bulut, and R. Anderl, "Framework of an active learning python curriculum for first year mechanical engineering students," in *2020 IEEE Global Engineering Education Conference (EDUCON)*, 2020, pp. 1193–1200.

[8] M. G. Ellis and C. W. Anderson, "Plagiarism detection in computer code," 2005.

[9] M. Novak, M. Joy, and D. Kermek, "Source-code similarity detection and detection tools used in academia: A systematic review," *ACM Trans. Comput. Educ.*, vol. 19, no. 3, may 2019. [Online]. Available: https://doi.org/10.1145/3313290

[10] Z. Durić and D. Gašević, "A Source Code Similarity System for Plagiarism Detection," *The Computer Journal*, vol. 56, no. 1, pp. 70–86, 03 2012. [Online]. Available: https://doi.org/10.1093/comjnl/bxs018

[11] D. Heres and J. Hage, "A quantitative comparison of program plagiarism detection tools," in *Proceedings of the 6th Computer Science Education Research Conference*, ser. CSERC '17. New York, NY, USA: Association for Computing Machinery, 2017, pp. 73–82. [Online]. Available: https://doi.org/10.1145/3162087.3162101

[12] V. Juričić, T. Jurić, and M. Tkalec, "Performance evaluation of plagiarism detection method based on the intermediate language," 2011.

[13] D. Ryman, P. Imbrie, and J. Kastner, "Application of source code plagiarism detection and grouping techniques for short programs," in *2021 IEEE Frontiers in Education Conference (FIE)*, 2021, pp. 1–7.

[14] J. Hage, P. Rademaker, and N. van Vugt, "Plagiarism detection for java: A tool comparison," in *Computer Science Education Research Conference*, ser. CSERC '11. Heerlen, NLD: Open Universiteit, Heerlen, 2011, p. 33–46.

[15] H. Cheers, Y. Lin, and S. P. Smith, "Evaluating the robustness of source code plagiarism detection tools to pervasive plagiarism-hiding modifications," 2021.

[16] S. Schleimer, D. S. Wilkerson, and A. Aiken, "Winnowing: Local algorithms for document fingerprinting," in *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, ser. SIGMOD '03. New York, NY, USA: Association for Computing Machinery, 2003, p. 76–85. [Online]. Available: https://doi.org/10.1145/872757.872770

[17] L. Prechelt and G. Malpohl, "Finding plagiarisms among a set of programs with jplag," *Journal of Universal Computer Science*, vol. 8, 03 2003.

[18] M. S. Joy and M. Luck, "Plagiarism in programming assignments," *IEEE Transactions on Education*, vol. 42, pp. 129–133, 1999.

[19] "Codequiry," https://codequiry.com/ [Accessed: 5 February 2022].

[20] A. Ahtiainen, S. Surakka, and M. Rahikainen, "Plaggie: Gnu-licensed source code plagiarism detection engine for java exercises," *ACM International Conference Proceeding Series*, vol. 276, 02 2006.

[21] D. Gitchell and N. Tran, "Sim: A utility for detecting similarity in computer programs," *SIGCSE Bull.*, vol. 31, no. 1, p. 266–270, mar 1999. [Online]. Available: https://doi.org/10.1145/384266.299783

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

[23] B. Devore-McDonald and E. D. Berger, "Mossad: Defeating software plagiarism detection," *Proc. ACM Program. Lang.*, vol. 4, no. OOPSLA, nov 2020. [Online]. Available: https://doi.org/10.1145/3428206