Kitsune: Structurally Aware and Adaptable Plagiarism Detection

Zachary Monroe, Ajay Bansal
School of Computing, Informatics, and Decision Systems Engineering
Arizona State University
Mesa, AZ, USA
{zlmonroe, ajay.bansal}@asu.edu

Abstract: Plagiarism is a huge problem in a learning environment. In programming classes especially, plagiarism can be hard to detect as source codes' appearance can be easily modified without changing the intent through simple formatting changes or refactoring. Many source code plagiarism tools do not support a high number of languages because doing so requires maintaining too large of a codebase. It is also difficult to add support for new languages because each language can be vastly different syntactically. Tools that are more extensible often do so by reducing the features of a language that are encoded and end up closer to text comparison tools than structurally aware program analysis tools [27]. This paper introduces a new tool called Kitsune, a plagiarism detection tool, focused on syntactically and structurally aware yet adaptable plagiarism detection. Kitsune has been evaluated for 10 of the languages in the Antlr4 grammar repository with success and could easily be extended to support all the grammars currently developed by Antlr4 or future grammars which are developed as new languages are written.

Keywords—plagiarism detection tools, structurally aware, adaptable

I. Introduction

Plagiarism detection is a daunting task. When done among large files sets, it can be extremely difficult to maintain both efficient and accurate results [24]. Source file plagiarism detection presents a whole new set of challenges in that many programs can be easily modified to look completely different with little to no work [1, 6, 11]. Detecting this easily becomes a language-specific problem and can be difficult to generically represent as can be seen by the small number of languages supported by the majority of surveyed plagiarism tools freely available to professors for grading as shown in Table 1. The table shows the names of the tool, their availability (in source code format, binary format, or as a service), number of languages supported, and their primary purpose. There are also a wide number of reasons to detect plagiarism, or more generically source code duplication, which adds to the difficulty of finding tools which support a given need. This paper will introduce a new tool, Kitsune, which is intended as a more adaptable solution to contemporary open source tools available to professors. In this effort, Kitsune will be benchmarked against a number of major tools to measure its success in a variety of factors which may be of importance in discerning its practicality as a contender to currently existing solutions. Evaluation of Kitsune for 10 of the languages in the Antlr4 grammar repository [5] will be presented and it could easily be extended to support all the grammars currently developed by Antlr4 or future grammars that are developed as new languages are written.

TABLE I. Language Support from Major Tools

Tool	*Available for use*	*Languages*	*Purpose*
Accuse [22]	**No**	**2**	**Plagiarism**
CCFinder [17]	Binary	7	Duplicated Code
CodeMatch [36]	**Binary**	**Text**	**Plagiarism**
Cogger [8]	**No**	**1**	**Plagiarism**
CoP [20]	No	Binary	Comm. Plagiarism
CPD	Yes	15	Refactoring/Bugs
CPMiner [18]	No	2	Duplicated Code
Duploc [12]	No	Text	Duplicated Code
FPDS [24]	**No**	**1**	**Plagiarism**
Gplag [19]	No	3	Comm. Plagiarism
Jones [15]	**No**	**1**	**Plagiarism**
Jplag [21, 26]	**Yes**	**6**	**Plagiarism**
Marble [13]	**No**	**4**	**Plagiarism**
Moss [3, 29]	**Service**	**25**	**Plagiarism**
NICAD [28]	Yes	4	Duplicated Code
Pdetect [23]	**Yes**	**1**	**Plagiarism**
Plaggie [2]	**Binary**	**1**	**Plagiarism**
PlagioGuard [10]	**No**	**2**	**Plagiarism**
Plague [34]	**No**	**Unknown**	**Plagiarism**
Sherlock [16]	**Yes**	**Text**	**Plagiarism**
SID [4]	**Yes**	**2**	**Plagiarism**
SIM [11]	**Yes**	**8**	**Plagiarism**
Simian	Binary	Text	Refactoring/Bugs
Yap3 [35]	**No**	**Unknown**	**Plagiarism**

In rest of the paper, section II presents the approach for building a plagiarism detection tool that is syntactically and structurally aware and adaptable. Section III presents the methodology followed by an evaluation of Kitsune in Section IV.

II. Approach

Antlr4 (ANother Tool for Language Recognition) [5] is a powerful parser generator for reading, processing, executing, or translating structured text or binary files. A basic read-through of most Antlr4 grammars used was required to identify major block type units of code such as classes, functions, structs, procedures, etc., especially when the language was known. This section describes the proposed approach.

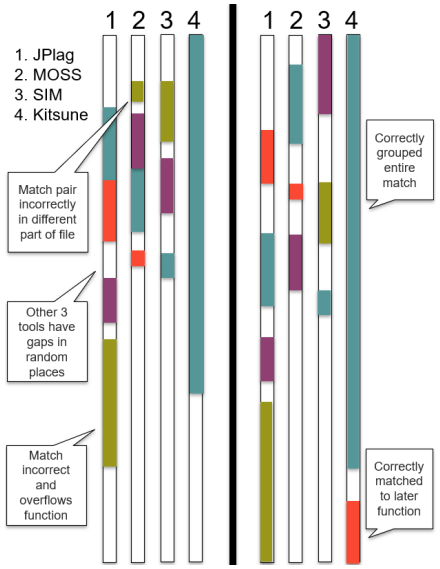
A. *Parse Tree Generation Using Antlr4*

Both JPlag, one of the best received plagiarism tools surveyed, and Kitsune rely on Antlr4 for some part of their pipeline. Despite this, the *ways* in which the Antlr4 generated parser is used is quite different. Jplag [26] focuses on using Antlr4 for providing tagging on identified tokens. It allows, for example, identifiers to be located and the beginnings of its abstract token analysis to occur. However, much of the functionality of Antlr4 is not utilized. Kitsune uses the tokens and the tagging done by Antlr4, but it also preserves the entirety of the parse tree structure that it generates. This allows it to take a much more structural view of the code as a whole. Whereas JPlag sees a program as a sequence of tokens, Kitsune views it as a tree. This presents a number of useful distinctions when analyzing source code for plagiarisms. Usually, students see programs more in blocks and in terms of structure than as an unrelated sequential list of tokens. A student would be, for example, unlikely to copy the end of one function and the start of the next from another student and much more likely to look at each function as a unit. In fact, this is an area that many of the tools suffer. SIM's internal engine works more like a scanner than an actual parser and thus loses most concept of scope. Sherlock has no concept of programming languages to begin with and views the code simply as text. MOSS portrays its results using lines and the length of token streams in its Winnowing algorithm. This unique approach allows Kitsune a more accurate view of the code. It also means that many of the problems that other tools had to overcome through careful algorithms or analysis were taken care of from the start. For example, shifting blocks of code around does less to affect Kitsune because it already compares two files at the block level instead of file to file.

In order to do this, Kitsune generates signatures for each of the major units of code. To identify these, Kitsune first takes the parse tree generated by Antlr4 and abstracts a number of things out. For example, an Antlr4 tree contains many nodes which are generated while traversing the precedence rules of a grammar. These nodes do not contain useful information for Kitsune and can be cut both for storage constraints in the resulting representation and for efficiency reasons while traversing the tree. This leaves the tree with only the grammar rules that directly matched parts of the code and the tokens which were matched. For example, a grammar which contains a generic expression rule followed by a multiplication rules, and then a match for two numeric tokens might be reduced by removing the generic expression since this has little baring for Kitsune's understanding of the code's structure. Identifiers and other volatile structures are then replaced with generic tokens. This is a common practice in plagiarism tools that use semantic aware modifications of the code to avoid being fooled by students which rename variables to appear to have different code. This can also be done for strings to avoid output modifications which have little bearing on the code's meaning.

In a small program like "Hello World!" this reduces a large amount of the state of the program. In a larger program, even of 50 lines however, this allows Kitsune to ignore small modifications to things like volatile strings which are only used for displaying content or variable names which can easily be changed. Next, a sliding hash window of three tokens is then generated. For example, in a python3 "Hello World!" program, the tokens 'print', '(', and '[[string]' would then be hashed. Currently, Kitsune shares a similar algorithm to both Sherlock and MOSS [3] where these hashes are coalesced, ordered, and a sample is chosen from them. Kitsune and MOSS both implement MinHash, in which the minimum hashes are selected to do so. The sample size is chosen such that Kitsune has an accuracy on any given block comparison of 95% using random sampling. Finally, an information node is added above the root node of the program which contains data about the program such as a unique identifier, whether it had any parse errors, the batch/directory it was added from (again a unique identifier), and any other important identifying data which might be displayed to the user.

Figure 1: File Match Pattern for Files "A3_6.py" and "A3_57.py"



B. *Code Block Identification*

Due to the way Kitsune represents source files internally using the tree that was generated by Antlr4, Kitsune has a number of advantages over many of the other tools. The following example highlights this behavior. The following is one of the file comparisons that all five tools identified as plagiarism. It is impossible to determine why Sherlock detected plagiarism definitively, but for the four other tools which generated representations to visually inspect the tools, it is clear that there is a specific function which accounts for much of the similarity. Because MOSS, JPlag, and SIM see this function as a stream of sequential tokens, however, their detection of its plagiarism identifies it as multiple separate instances of plagiarism, despite the entire encapsulating function being plagiarized and simply reordered. Kitsune on the other hand has the wherewithal to identify a single, large case of plagiarism due to its representation of the block not as a stream of sequential tokens, but as a node in a tree where all of the plagiarized sections fall inside the same branch. This distinction becomes especially apparent when looking at the similarities detected by JPlag where the match actually spans across the function and into the next function. Especially because JPlag is fairly generous with its token identification, this leads it to believe that

the next functions declaration is part of the similarity despite the next function in each program being completely different. In fact, Kitsune identified the second function for "A3_57.py" (shown in Figure 1) as being plagiarized from the *third* function of "A3_6.py" (shown on the left) a match JPlag did not catch, perhaps due to this overlap subtracting possible match tokens. "A3_57.py" and "A3_6.py" are Python program files that were compared for identical code.

C. *Parse Tree Modifications using Cypher and Neo4j*

One problem that using the parse tree presented was the difficulty of modifications to any such structure. Token streams are much easier to work with because they are stored as a flat list. Initially, the idea of continuing with an in-memory representation was floated. This however quickly became unmanageable and hard to maintain. In order to keep modifications to the representation simple, Neo4j, a graph database was used after parse tree generation. By inserting the tree generated by Antlr4 into a Neo4j database, most of the modifications to the tree became fairly simple. Figure 2 for example, shows a Cypher query to replace all variables with a set token value (a measure to avoid dissimilarity from irrelevant changes). Within this five line query, Cypher is able to identify all programs that were added in the last directory (line 1), get the unique IDs for every one of those programs (line 2), find every single terminal node that matches the specified name for identifiers in the grammar (line 3), limit those nodes to the identified programs (line 4), and finally replace the text of each with a chosen identifier token (line 5). Most interestingly, this query can be reused for every grammar thus far identified since the only requirement is that the add-on developer add the identifier's name and replacement text to a configuration file. It also is generic enough that the same query could be used to, for example, replace the text inside a string or other node depending on what was needed for a given language. This flexibility means that Kitsune does not require additional code for many of these simple tasks, nor does it require a grammar to be written in a specific way.

Figure 2: Cypher Querry for Variable Replacement

```
1 | MATCH (program:Program { directoryID: {directoryID}})
2 | WITH collect(DISTINCT program.programID) as programIDs
3 | MATCH (n:Node {category: "terminal", type: {type}})
4 |   WHERE n.programID in programIDs
5 | SET n.text = {replaceWith}
```

D. *Swapable Configurations*

Because of the generic nature of most of Kitsune's code, a need arose to be able to specify a minimal set of features in a language. The goal, however, was to keep this set as small as possible so as to present the developer with the least amount of work to introduce a new language to Kitsune. Currently, the average configuration comprises editing about 8 lines of code and a set export statement (7 lines), as seen in Figure 3. Thus a new language takes 8 lines of written code to introduce once an Antlr4 grammar has been written for it. Currently, the main Antlr4 grammar repository contains around 200 programming languages and is by no means a definitive source for Antlr4 grammars. If Kitsune decided to support all 200 programming languages, the language section would be only a little larger than the rest of the codebase, and leave Kitsune with less lines than many of the surveyed tools, including MOSS and JPlag.

Figure 3: Example Configuration for Python3

```
1 | const startRuleName = "file_input";
2 | const matchRules = ["if_stmt",  "while_stmt",  "for_stmt",
                        "try_stmt",  "with_stmt",  "funcdef",
                        "classdef", "decorated"];
3 | const similarityThreshold = 0.70;
4 | const replaces = [
5 |    ["STRING_LITERAL-local", "[[string]]"],
6 |    ["NAME-local", "[[identifier]]"]
7 | ];
8 | const fileExtensions = [".py"];
9 | module.exports = {
10|     startRuleName,
11|     matchRules,
12|     similarityThreshold,
13|     replaces,
14|     fileExtensions
15| };
```

E. *Accuracy*

Kitsune goes through a number of lengths to assure the accuracy of the comparisons it generates while reducing the sample size for efficiency. As mentioned earlier, Kitsune's goal is to maintain at least 95% accuracy in the comparisons it makes between any two code fragments. To explain how Kitsune achieves this, some background on MinHash needs to be given.

There are two forms of MinHash. In the first, the hash algorithm which is used to find the smallest token will be changed each time a sample is chosen [25]. In the second, the same hash function will be used for all of the tokens and rather than selecting the smallest token, the *N* smallest tokens will be chosen. The latter is a more recent development which builds on the idea of sampling without replacement. Usually, the simpler version (former) is used as an upper bound for the sample size required to estimate the similarity. This version has a required sample size of $1/E^2$ where E is the desired error margin. This version acts as a sampling-with-replacement type problem while the *N*-minimum version acts as a sampling-without-replacement. For a large document, the gains in accuracy are insignificant as the proportion of the sample to the population is rather small. As the 5% mark targeted by Kitsune is approached however, this is no longer true. As you sample more and more of the document, you approach having definitive knowledge of the entire document and thus are able to make more confident statements about the interval in which the similarity lies. The sample size required for a confidence interval of .05 (5% similarity) at 95% confidence for an infinitely sized document can be expressed by the following formula:

$$s = \frac{Z^2(p)(1-p)}{c^2}$$

where s is the required sample size, Z is the z value from the normal distribution (given that the hash function produces normally distributed hashes), p is the proportion of the documents that intersects, and c is the accuracy desired.

At 95% confidence, the Z value is 1.96. The desired confidence is simply what we hope to obtain. 5% deviation in accuracy should be enough to make a claim about whether the documents are similar. The only difficult measure to answer is p since without first comparing the documents we cannot know the overlap. However, we can solve this by simply finding the

upper bound where s is maximized. This value occurs at p = .5, a 50% overlap of the two programs. Thus we have

$$s = \frac{1.96(0.5)(1-0.5)}{0.05^2} = \lceil 384.16 \rceil = 385$$

So, for a theoretically infinitely sized program we would need 385 samples to give the similarity of the two documents within 5% with a 95% confidence. To adjust this for a finite set, we can use the equation

$$s' = \frac{s}{1 + \frac{s-1}{p}}$$

Where s' is the adjusted sample size, s is the old sample size, and P is the total population size. There is however a problem with this. Our sample comes from the population that is the Union of the two sets of tokens from each document. Because we don't know how many repeat tokens there were amongst the documents, we do not know the exact population size. We do however know that the maximum population size would occur when there is no overlap in the two programs (and thus every token in each is distinct). Thus, the new sample size will be maximized at P=|A|+|B| where |A| is the magnitude of the set of tokens from the first document and |B| is the magnitude of the set of tokens from the second. Putting it all together, we have

$$s' = \frac{s}{1 + \frac{s-1}{|A|+|B|}} = \frac{\frac{s}{1}}{\frac{|A|+|B|+s-1}{|A|+|B|}}$$

$$= \frac{s(|A|+|B|)}{|A|+|B|+s-1} = \frac{384.16(|A|+|B|)}{(|A|+|B|)+383.16}$$

So for example with 2 programs, one with 400 tokens and the other with 500, the sample size required would be

$$\frac{384.16\ (400+500)}{(400+500)+383.16} = 270$$

as opposed to the 400 tokens we would have taken before.

There are two main advantages of using MinHash for program comparison in this way. The first is that comparison across a large set of programs can be done in a very short amount of time. The second is that MinHash will be resilient to some level of difference between the two programs. By changing shingle size, MinHash balances the importance of token order against the possibility of code being rearranged slightly. For example, switching the order that two functions are defined in would only result in a similarity difference of the shingles that overlapped the two functions, which is relatively small compared to the size of the program.

III. Methodology

In order to investigate the success of Kitsune's method of plagiarism detection, a rubric to compare it to existing tools needs to be identified. To do so, it is necessary to understand what traits are needed in a quality plagiarism detection system. In that effort, a definition for quality software systems in general will first be established and later applied in the context of a plagiarism detection system.

There have been a number of well-regarded models for evaluating the quality of a software system. Though the exact terms vary, most agree that a software system which has high quality should demonstrate certain characteristics including maintainability, efficiency, reliability, usability, portability, and functionality [37]. A system which exhibits these traits can be said to be a "quality" software system. It is important to note, however that these terms are not absolute metrics, but context dependent criteria. What is considered efficient for a software comparison tool, for example, might not be what is considered efficient in a time critical system.

A. Quality In Context

A rubric which attempts to gauge the quality of different plagiarism detection tools should exemplify measurable characteristics that demonstrate the software possesses each of these traits. Each of these traits should be defined in the context of a plagiarism detection system, and from this, a measurement should be created to determine which tools successfully demonstrate these characteristics and to what degree. ISO 9126 goes on to define each of these traits further. Maintainability is the capability of the software product to be modified. Modifications may include corrections, improvements or adaptation of the software to changes in environment, and in requirements and functional specifications [14]. To this extent, a maintainable plagiarism detection tool should demonstrate a number of properties, including: a small general codebase (excluding additional language support), an adaptable language with good support for new operating environments, and a small amount of required code for each additional language.

Efficiency under ISO 9126 is defined as the capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions [14]. This includes reasonable consumption of both time and system resources. In a plagiarism detection system, this means that the software should be able to run on a normal laptop or desktop that the average professor or grader might have. It also means that for a sufficient classroom size, a teacher can compare the entire set and expect to get results in time to grade the assignment.

Reliability is the capability of the software product to maintain a specified level of performance when used under specified conditions. A "reliable" plagiarism tool should identify all sections of sufficient size which are likely plagiarized -- or at least merit manual inspection -- in the file set and should not return false positives. As this is hard to demonstrate for most real world test sets, a reliable plagiarism detection tool can be seen as a tool which detects at least the cases that other tools find and which does not return cases where no other tool sees plagiarism or where, upon manual analysis, the tool detects cases where no other tool does due to extended capabilities of the tool.

Usability is the capability of a software to be understood, learned, and attractive to the user, when used under specified conditions. This can be more subjective than other traits and hard to define, however, some likely necessary characteristics include: a clean UI which is easy to use and clearly displays matches, and a simplistic system to obtain and run the tool.

Portability is the capability of the software product to be transferred from one environment to another. To this extent, a plagiarism detection tool should support common Operating Systems including Windows, Linux, and MacOS. Another capability some tools displayed which might be favorable is the ease of deploying it as a web service for increased access.

Functionality is the capability of the software product to provide functions which meet stated and implied needs when the

software is used under specified conditions. A plagiarism tool which shows “functionality" is one which demonstrates a broad feature set -- in this case support for a large array of languages and which can be tuned for things different teachers would desire such as exclusion of template code, threshold modifications, etc.

There are obvious trade-offs between many of these traits. A tool which aims to provide functionality through support for as many languages as possible with high reliability might become un-maintainable given that support for each additional language requires extensive additions to the codebase. Likewise, a tool which aims to support all languages through text only comparison might find its reliability falter. Each of the tools surveyed presented cases for a variety of different approaches, each valuing a different combination of goals to different degrees. Through a holistic rating system, it will be possible to determine which tool(s) utilize the most balanced approach.

B. Deriving a Rubric

It would be difficult to remain objective while rating a tool on a numeric scale such as 1-10 as other authors have done in the past [9][12]. For this reason, a more generalized scale from “Inadequate" to “Exemplary" with two mid-range levels for “Acceptable" and “Excellent" will be used. In addition, every attempt will be made to create a full rubric with hard rules as to where a tool falls to avoid subjective or non-replicable scoring. Using this level of granularity, many of the aforementioned traits are quite easy to grade with quantitative evidence. Looking at each of the traits we analyzed previously, our final goal is to attempt to categorize each into these 4 bins.

Under maintainability, we identified 3 criterion: the size of the base codebase, the size of additional language add-ons, and the suitability/adaptability of the language to new environments. Many of the tools analyzed vary greatly in code size. There is also not always a clear break between language dependent code and base code size. For this reason, the size estimate must be able to represent this range across the 4 bins. In order to avoid arbitrary coalition, an order of magnitude type comparison seems apt. It likely also makes sense for a program which is less than 1 KLOC to be seen as exemplary in terms of maintainability while a program doing the same actions of over 100 KLOC would be seen as “Inadequate" in terms of maintainability. Keeping with this scale for the additional language support code size, an exemplary program would be one that does not add any code or for which a user could easily enter information. This would mean that any language could easily be supported with zero work on the side of developers and with no maintenance. Going from there, small code changes are less desirable as some issues requiring maintenance might occur, but still excellent. A program requiring 100-1K LOC seems typical of most of the programs analyzed and thus is likely adequate in terms of maintenance. Finally, a program going beyond this would require extensive updates for language additions and likely would not see many languages introduced at a fast rate. In terms of language adaptability, it would be difficult to use a fully quantitative measure. The easiest way to measure whether the language supports systems well is perhaps how often it receives updates. If a language has been depreciated for a long time or is too new, then it likely would be inadequate for a stable system. Likewise a language which is stable but still receiving regular updates would be much stronger as a choice. Out of the tools analyzed, it is unlikely that any will have trouble with this category and thus a stronger metric is likely not needed. For other tools, especially older, less supported tools, a more accurate measure would likely be needed.

Under efficiency, the main identified criteria was simply that the software could be run by the professor in a feasible time. This can be further broken apart to talk about what size set can be run feasibly. In order to test this, 3 sample sets of varying size in terms of both lines of code and number of students will be executed on the same computer and timed. In order to avoid discrepancies in the hardware two measures will be taken. First, the times will be an average of 3 runs on each set. Stored results (including automatic caching as in a database) will be cleared to prevent unfair speedups. Second, as in the case of lines of code measurements, the rubric shall be on a scale of magnitude to illustrate only large deviations. Thus we have 3 sets, running the average execution time, and binning across that time on a scale from less than 1 min, to less than 1 hour, to less than 1 day. Any times exceeding 1 day will be cut off after 1 hour past and it will be assumed that this 1 hour is sufficient to say that future attempts will not be less than the 1 day mark.

Reliability is a harder trait to quantify. It would not be difficult to manufacture a plagiarized program and introduce it into an otherwise clean set as done by many other authors [4, 19, 28]. However, doing so brings into question whether the intentional modifications applied by a knowledgeable programmer successfully mimic those of a student under actual duress and time pressure. For this reason, an alternative approach presented by the author of one of the tools not being analyzed, FPDS, will be used. In this system, it is assumed that the majority of plagiarism tools which are widely used are successful in general and that their failures can primarily be attributed to the extremely apparent discrepancies between them [24]. Put another way, the tools can act as a “jury" for the programs that are analyzed. The larger the consensus of the tools in the jury, the worse it is for a tool to fall outside this jury. For example, if 4 tools agreed that a case was plagiarism and a single tool did not, this tool is likely incorrect. The frequency with which these deviations occur can be seen as the reliability of this tool. The same can be said for the reverse case when 4 tools say that a case is not plagiarism and a single believes that it is. To avoid issues with certain tools failing only on certain languages or certain problems more often, the tools will be graded on the number of test sets where they significantly (seen as 1 stdev above the average fail count) deviate from the results of the other tools. Each tool will be tested on 5 test sets of various language and size. Four languages are supported by all 5 tools - Python3, C, C++, and Java. In order to thoroughly test the capabilities of each tool, all 4 languages will be tested.

For the data sets used to test both reliability and efficiency, the goal will be to maintain as realistic of sets as possible. Generating sets manually, either plagiarized or otherwise, will be avoided in the hopes of keeping the data as realistic to a real world setting as possible. To this goal, there will be 2 main sources for the programs used in these tests. Most of the sets will be gathered from CodeChef 2020 [7], a nonprofit coding competition website. CodeChef provides the solutions to already completed competitions to users doing them for practice can see these answers. This provides 2 main advantages as a test set. First, while users are encouraged to not use these other

programs, because they are available, a number of the submissions have been “plagiarized" from the displayed solutions. Second, because the solutions are public, downloading them to run with the tools is possible. There is also strong evidence for using plagiarism detectors on these sets as after completion of competitions, the site actually uses MOSS to check that the winning submissions have not been plagiarized. In addition to the sets gathered from CodeChef, one of the sets used will be a larger program set taken from a graduate level Software Engineering course which uses Python. This will provide both a benchmark for a larger set, and will also provide a good test for code actually used in classroom at a higher level.

A full investigation into the usability of each tool would likely require extensive usability studies. As this is currently out of the scope of this investigation, the usability of each tool will instead by classified by the presence of certain absolute features. There are two main stages to using any plagiarism tool: setting it up, and running the detection. Running it involves user interactions with an interface, either command line or UI based. Because of the wide variety of users of a plagiarism detection tool, it should not be assumed that every user is technically capable of running a command line tool. For this reason a UI based tool presents an advantage in terms of userbase size. This can be further broken down into which parts of the tool support graphical usage. Many of the tools expect command line input but generate views afterwards which are more user friendly. On the note of obtaining the tools, the actual process of finding the website, etc. will be ignored. From there the difficulty of actually using the tool will be rated. A tool which requires compiling source code, for example, would be much worse in terms of usability than a tool with a traditional installer.

The portability of the system was broken down into two characteristics, its ability to support major operating systems and the ability to be web deployed for easy outreach to other users. An inadequate system can be seen as one that does not support a single common operating system (seen for this study to be Windows and MacOS as these operating systems constitute an estimated 98% of the personal computer market among general users [31, 32] and 74.5% of all developers [30]. An adequate tool would be one that supports at least one major platform since it could be used by a large number of users. Preferred would be a tool that supports both of the major platform. An exceptional tool would support Linux as well giving the tool support for at least 99% of personal computers professors or graders would have access to. As web deploy-ability was not the intent of many of the tools, it would be unfair to rate whether they currently support online usage only. Thus the level of modifications to support such an important feature should be compared instead. An inadequate tool would be one for which it is completely impractical to web-deploy, either due to language restrictions or architecture choices. An adequate tool would be one that could be deployed with some effort (including even a complete rewrite of the UI component). An excellent tool would be one that could easily be deployed with minor modifications, especially one that would not require a UI rewrite. Even better would be a tool that already supports web deployment or has supported it in the past (and thus it could easily be revived).

Finally, the functionality of the system was earlier broken down into two traits: the support of languages, and the support of additional tuning features, namely template code, removal of common code, and threshold modification. These three traits have been singled out by a large majority of authors as intrinsically necessary for a plagiarism tool to support. A tool which does not support template code often fails to work for assignments where the professor allows either using code from the textbook or other sources or provides a starting place for students to work from. A tool which does not remove common code often produces false, often random similarities which might be tied to the language or the problem. For example, a student who's code contains the java code “*public static void main(String args[])*" should not be penalized for this. Finally, tuning of the threshold allows professors and graders to more easily view only the code they need to and access what level of similarity is plagiarism themselves. A tool which supports none of these would likely be inadequate for most use cases. Each of these traits likely add to its usability. The language support of a tool can be seen it two lights: commonly used languages and less common languages. In this study, common programming languages will be defined as the top 20 most used programming languages according to the 2019 StackOverflow survey. Any other languages will be grouped separately. This prioritizes tools which have functionality which is desired by the majority of users as opposed to those which support more esoteric purposes which, though absolutely provide useful functionality, likely do not benefit the majority of users. From the investigation of major tools (as seen in Table 1) it appears as though most tools fluctuate between the range of 1 language to 25 languages. Thus the most “exemplary" of these tools should support all or almost all (75%) of the top 20 languages. An excellent tool would support at least half (50%) of the most common languages. An adequate tool should support at least 5 (25%). An inadequate tool would support less than 5 (less than 25%). For less common languages, there is no clear bound to the number of languages. Some of the tools which supported less language dependent methods can support any language. While this is certainly exceptional, it likely falls outside of expectations of most tools. As most tools focus on common languages, a tool which supports more than 20 other languages (thus supporting more uncommon languages than common) would certainly be exceptional even if it does not support all languages generically. From there, the same scale can be used at 75%, 50%, and 25% of this expectation.

While the methodology that they arrived at their rubrics was not the same, it should be noted that other surveys which attempted to compare tools holistically have used similar features to analyze them. One survey, for example, rated 5 tools on 10 features that they deemed important for any tool which was to be used by professors for student submissions [12]. The survey included “Supported Languages", “Extendibility", “Presentation of Results", “Usability", “Exclusion of Template Code", “Local or Web Based" and “Open Source" among their list of traits. Of the 10 traits, 7 were represented in the rubric that was developed. Of the 10, only “Historical Comparisons", “Submission or File-based Rating" and “Exclusion of Small Files" were not noted. This is because these 3 traits seemed to be heavily based on the specific usage that the survey was done for and not true for plagiarism detection in general, or because every tool surveyed seemed to almost unanimously support this trait . Their definition of “Historic Comparisons' for example was defined as “there must be a way to distinguish older

submissions from newer ones. Either by indicating which are the new or old submissions when starting the tool, or by putting different incarnations in different directories" [12]. Of the available tools, not a single tool was unable to separate submissions, so this seemed unnecessary as a criterion. Another survey additionally included measures including “Platform Independence", “Local or Remote", “Usability", “Number of Supported Languages", and “Algorithm Extensibility" in their analysis were all included in the developed rubric [9].

IV. Results

Kitsune scored extremely well in comparison to the current state of the art in plagiarism detection tools. While scoring each tool on the rubric, a number of deficiencies and places where each tools excels both became apparent (as shown in Figure 4). In this section, the result of each investigation will be detailed and any notes beyond what the rubric allowed will be noted.

A. Maintainability

JPlag's source code is publicly available on GitHub. Because of this, it was fairly easy to see how much code goes into both its base work and each additional language. JPlag exceeds the 1000 LOC mark per language putting it in the “Inadequate" section for language additions. Though MOSS's source code is not public, a lot of insight into its maintainability was gained through direct correspondence with the Author. Moss is primarily written in C and is about 5000 lines of code, ignoring language specific additions. Aiken has developed a domain-specific language for adding new languages and estimated that additions were just shy of 100 LOC per additional language. This puts MOSS's language support maintainability at “Excellent" falling short only of having no additions required or additions that do not require programming. Sherlock is extremely small in size (~400 lines of C code) and thus easily falls under “Exemplary" in this aspect. It does not have a “language specific" section as it is intended to do text-based analysis that is fairly independent of the language itself. This again means that Sherlock scores “Exemplary" since it does not require language specific add-ons. Because of this, Sherlock was the most maintainable of all of the tools that were surveyed. Additions to SIM were not particularly hard, but certainly require a good understanding of the language that is being added and is not a trivial task.

Finally, Kitsune's base code size including the GUI (which did not exist in any of the other tools) falls just shy of the 1000 lines of code mark including both JS code for the UI, database queries, HTML UI code, and installer code which was not present in other tools (in fact pom.xml, html files for the generated UI, etc. were strictly ignored for other tools). This puts Kitsune safely in the “Exemplary" range. Additional languages require configuration code (which currently would not be able to be done by the average user, though there are plans for this) which have not as of yet exceeded 15 lines of code. This places Kitsune under “Excellent" falling short only of Sherlock's non language specific implementation and tying with MOSS's domain-specific language.

The five tools that were surveyed all used strongly established programming languages for their main codebase and thus do not suffer in this respect (though some suffer under portability and OS support for other, less language-related choices). JPlag, MOSS, Sherlock, SIM, and Kitsune use Java, C, C, C, and NodeJS respectively. All tools were scored as “Exemplary" because of this.

B. Usability

JPlag ships as an executable jar file which can be run from the command line. When run, it generates a simplistic webpage based UI which can be used to navigate the results. Though simple, JPlag's UI provides some degree of navigability and allows prioritization based on higher percentage similarity (i.e. the user can easily choose to skip looking at a 5% similarity document). Staying objective in line with the rubric, JPlag's has a CLI based input process but a GUI based results section giving it a score of “Acceptable". In terms of obtaining and using the tool, JPlag is fairly simple. It requires having the correct version of Java installed and command line execution which may be considered “developer knowledge", but being lenient on the basis that most users are comfortable with Java versions and can run “java -jar jplag.jar" Jplag has an “Excellent" rating for obtaining and starting, falling short of “Exemplary" due to the lack of a typical installer type installation or guided process. MOSS is very similar to JPlag in terms of usage. MOSS comes shipped as a Perl script which submits the files to a server running the MOSS web platform. MOSS then generates a webpage the interface for which was actually written by the author of JPlag and responds with the link. Sherlock was perhaps one of the least usable of the three tools. Sherlock does not provide a graphical interface for either viewing results or inputting them. It also does not provide any visualization of the results -- even at a command-line level. The only output from Sherlock is a list of file pairs with a percentage. This makes it extremely difficult, especially in large files, to determine why the files were deemed similar. There is also no readily available distribution from Sherlock, so the only way to run it is to run the makefile and generate the binary, a feat likely out of reach for non-technically oriented users. SIM, like Sherlock does not provide a graphical mechanism for comparing files.

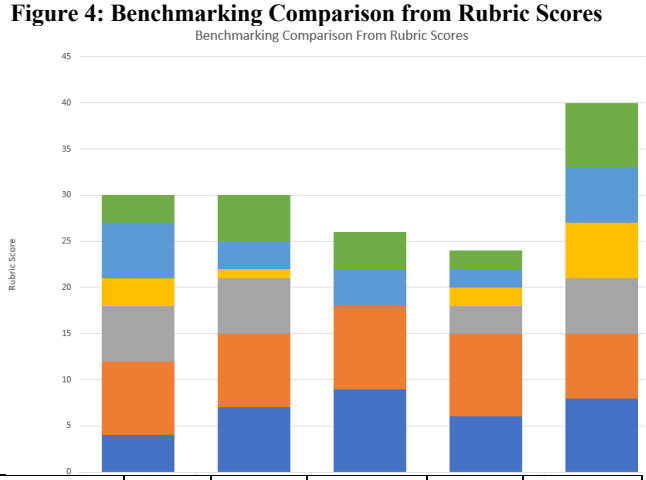
Unlike the other 4 tools, Kitsune provides a graphical interface for both inputting files, tuning parameters, and comparing the results. Kitsune also allows for regeneration of results with different parameters, allows navigating results including filtering out results based on the viewers discretion (for example such that low similarity comparisons are removed, or such that only files the user chooses are included. The tool also has a guided installer which can set up the application on the system as a full application that can be used normally. Because of these features, Kitsune scores well under both its UI and the setup process.

C. Portability

JPlag, Sherlock, and Kitsune all support both MacOS, Windows, and Linux with no issues. Only two tool presented issues with portability in terms of operating system support -- MOSS and SIM. SIM provided binaries only for “MSDOS" (which run fine on Windows 10 today). It is theoretically possible to recompile for other systems, however the Makefile is not set up for this and would require major revisions. There also may be some minor amounts of code which would need rewrites. Because of this the average user would be unable to use SIM on either Linux or MacOS. current Moss submission script is for Linux and does not provide support for different

systems. MOSS's script can be run through Cygwin, WSL, etc. but is not indented to natively run on any system besides Linux. Because of this MOSS does not meet the minimum criteria for portability. JPlag has run a web service in the past and in fact still includes the code to do so in the open source repository. MOSS is available only as an web submission service and thus is also web deployable.

Figure 4: Benchmarking Comparison from Rubric Scores



	JPlag	Moss	Sherlock	SIM	Kitsune
Functionality	3	5	4	2	7
Portability	6	3	4	2	6
Usability	3	1	0	2	6
Reliability	6	6	0	3	6
Efficiency	8	8	9	9	7
Maintainability	4	7	9	6	8

Kitsune's main UI is currently based on electron and can be ported to a web deployable service using webpack fairly easily (this has been tested). Many of the comparison operations and overhead are also run through Neo4j Graph Database. Because of this, Kitsune actually acts like a client to this remote host which could quite easily be installed on a central server should the need arise. Sherlock and SIM are both C based application. It would certainly be possible to route execution with a thin wrapper or CGI type approach, but would still require adding a decent amount of new code. Because there is no native support for doing so, both were scored under “Adequate".

D. Functionality

JPlag currently supports 5 of the top 20 languages: C, C++, Java, Python, and C#. This gives it a score of “Adequate" in terms of language support. JPlag currently supports threshold adjustments, and is able to take in a template to exclude from results. MOSS was the tool that supported the most languages out of every tool surveyed which did not have a general purpose algorithm such as Sherlock. MOSS supports template code exclusion, and automatically removes common code across the sets (the number required to be `common" can also be changed). There is currently no way to adjust the threshold for the degree of similarity. SIM supports C, C++, Java, Lisp, and 8086 assembler code giving it support for exactly five of the common languages from the 2019 StackOverflow survey. SIM does not support common code exclusion or template code. It does however support threshold adjustments for it's output. Because of Sherlock's completely language agnostic text-based comparison, Sherlock has no trouble supporting any language you throw at it. Because of this, Sherlock scores “Exemplary" in both common and uncommon language support. Sherlock, like SIM, only supports changing the threshold for output, giving it a score of “Adequate" for commonly requested features.

Because Kitsune does not require much in the way of a configuration for a given language, Kitsune can be configured to support a new language within around 5 minutes given any antlr4 grammar already exists. This means that Kitsune has no difficulty supporting the already massive repository of grammars Antlr4's grammar repository contains, as well as the many grammars that can be found across GitHub which are not contained in that repository. Kitsune's accuracy does not seem to depend too highly on the level that the grammar is tuned for plagiarism task either. That is to say, Kitsune does about the same using JPlag's grammars which are tuned with plagiarism detection in mind as it does with any of the compiler oriented grammars in the repository. Kitsune was able to detect visually accurate matches in a wide variety of languages including Python, Java, Javascript, Lua, CSS, HTML, C++, C, bash, and VBA. These languages present a wide array of languages with many different syntax and code styles. Because of this, Kitsune scored an "Exemplary" in both of the language categories. Currently, Kitsune supports only threshold tuning and does not support template code exclusion or common code detection, though common code detection has begun development. Because of this, Kitsune score “Adequate" in common requested feature support. Figure 4 shows a benchmarking comparison.

V. Conclusion

Kitsune has performed extremely well when compared with popular available solutions. It has been benchmarked against other tools in four programming languages and been comparable in all four. Additionally, it has been tested for 6 other languages and had no issues producing visibly accurate results among test sets. Kitsune requires minimal effort (usually 5 to 10 minutes) to add additional language support from start to finish and requires very little code. Only a basic read through of most Antlr4 grammars used was required to identify major block type units of code such as classes, functions, structs, procedures, etc., especially when the language was known.

Since the initial two languages, no adjustments have been required to Kitsune's algorithms to adapt it to new languages. Especially when considering the versatility in the syntax of the chosen languages (Powershell and CSS for example when compared to Python or C) this shows promise that Kitsune could quickly be adapted to include the entire Antlr4 grammar repository in its language set. This would mean support for around 170 languages not supported any of the tools analyzed which showed consistently reliable results. For these languages, Kitsune would be the only tool teachers could use to prevent plagiarism in a class not using one of the small list of languages commonly supported by existing plagiarism tools. In addition to the work done to provide sophisticated analysis of languages, Kitsune provides a robust system for visual syntax highlighting also performed using the Antlr4 grammar. Currently, 4 of the tested languages have syntax highlighting schemes written for them. While not necessary, these make interpreting results from the tool much easier and make the tool appear much more professional than a plain text display.

References

[1] Ahmed, R. A., "Overview of different plagiarism detection tools", International Journal of Futuristic Trends in Engineering and Technology 2, 10, 1–3 (2015).

[2] Ahtiainen, A., S. Surakka, et al, "Plaggie: Gnu-licensed source code plagiarism detection engine for java exercises", in BalticSea Conference on Computing Education research", pp. 141–142 (2006).

[3] Aiken, A., MOSS http://moss.stanford.edu/general/faq.html (2018).

[4] Chen, X., B. Francia, M. Li, B. Mckinnon and A. Seker, "Shared information and program plagiarism detection", IEEE Transactions on Information Theory 50, 7,1545–1551 (2004).

[5] Parr T. The definitive ANTLR 4 reference. Pragmatic Bookshelf; (2013).

[6] Clough, P., "Plagiarism in natural and programming languages: An overview of current tools and technologies", (2000).

[7] "Codechef: Programming competition, programming contest, online computer programming", https://www.codechef.com/ (2020).

[8] Cunningham, P., "Using cbr techniques to detect plagiarism in computing assignments", Tech. rep., Citeseer (1993).

[9] Duri´c, Z. and D. Gasevic, "A source code similarity system for plagiarism detection", The Computer Journal 56, 1, 70–86 (2013).

[10] Goel, S., D. Rao et al., "Plagiarism & its detection in programming languages", Technical Report, Dept. of CS & IT, JIITU (2008).

[11] Grune, D. and M. Huntjens, "Detecting copied submissions in computer science workshops", Informatica Faculteit Wiskunde & Informatica, Vrije Universiteit 9 (1989).

[12] Ducasse, S., M. Rieger and S. Demeyer, "A language independent approach for detecting duplicated code", in IEEE International Conference on Software Maintenance (1999).

[13] Hage, J., P. Rademaker and N. van Vugt, "A comparison of plagiarism detection tools", Utrecht University, Netherlands (2010).

[14] ISO 9126, URL https://www.iso.org/standard/39752.html (2016).

[15] Jones, E. L., "Metrics based plagarism monitoring", in "Journal of Computing Sciences in Colleges", vol. 16, pp. 253–261 Consortium for Computing Sciences in Colleges, 2001.

[16] Joy, M. and M. Luck, "Plagiarism in programming assignments", IEEE Transactions on Education 42, 2, 129–133 (1999).

[17] Kamiya, T., S. Kusumoto and K. Inoue, "Ccfinder: a multilinguistic token-based codeclone detection system for large scale source code", IEEE Transactions on SoftwareEngineering 28, 7, 654–670 (2002).

[18] Li, Z., S. Lu, S. Myagmar and Y. Zhou, "Cp-miner: A tool for finding copy-paste and related bugs in operating system code.", in "OSDI", vol. 4, pp. 289–302, (2004).

[19] Liu, C., C. Chen, J. Han and P. S. Yu, "Gplag: detection of software plagiarism byprogram dependence graph analysis", in "Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining", pp. 872–881 (2006).

[20] Luo, L., J. Ming, D. Wu, P. Liu and S. Zhu, "Semantics-based obfuscation resilient binary code similarity comparison with applications to software plagiarism detection", in "ACM SIGSOFT Intl. Symposium on Foundations of Software Engineering", pp. 389–400 (2014).

[21] Malpohl, G., "jplag/jplag", URLhttps://github.com/jplag/jplag (2019).

[22] Grier, S., "A tool that detects plagiarism in pascal programs", ACM SIGCSE Bulletin 13, 1, 15-20 (1981).

[23] Moussiades, L. and A. Vakali, "Pdetect: A clustering approach for detecting plagiarism in source code datasets", The Computer Journal 48, 6, 651–661 (2005).

[24] Mozgovoy, M., K. Fredriksson, D. White, M. Joy and E. Sutinen, "Fast plagiarism detection system", in International Symposium on String Processing and Information Retrieval", pp. 267–270 (Springer, 2005).

[25] Oracle, "Oracle america, inc. v. google llc". Plank, J.S., http://web.eecs.utk.edu/~jplank/plank/classes/cs494/494/notes/Min-Hash/index.html (2019).

[26] Prechelt, L., G. Malpohl, M. Philippsenet al., "Finding plagiarisms among a set ofprograms with jplag", J. UCS8, 11, 1016 (2002).

[27] Ragkhitwetsagul, C., J. Krinke, D. Clark, "Similarity of source code in the presence of pervasive modifications", in IEEE Intl. conference on source code analysis and manipulation (SCAM)", pp. 117–126 (2016).

[28] Roy, C. K. and J. R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization", in 16th IEEE International Conference on Program Comprehension", pp. 172–181 (2008).

[29] Schleimer, S., D. S. Wilkerson, and A. Aiken, "Winnowing: local algorithms for document fingerprinting", in Proc. of ACM SIGMOD Conference on Management of Data", pp. 76–85 (2003).

[30] Sonos, "Sonos, inc. v. Google LLC". StackOverflow Survey 2019, https://insights.stackoverflow.com/survey/2019 (2019).

[31] StatsCounter, "Desktop operating system market share worldwide", https://gs.statcounter.com/os-market-share/desktop/worldwide (2020).

[32] Steam, "Steam hardware and software usage survey", https://store.steampowered.com/hwsurvey?platform=combined (2020).

[33] Turnitin, "About us", https://www.turnitin.com/about (2020).

[34] Whale, G., "Identification of program similarity in large populations", The Computer Journal 33, 2, 140–146 (1990).

[35] Wise, M. J., "Yap3: Improved detection of similarities in computer program andother texts", in Proceedings of the twenty-seventh SIGCSE Technical Symposium on Computer Science Education", pp. 130–134 (1996).

[36] Zeidman, B., "Software v. software", IEEE Spectrum 47, 32-53 (2010)

[37] Al-Qutaish, R. "Quality Models in Software Engineering", In Journal of American Science, 6.3, p. 175 (2010).