Kitsune: Structurally-Aware and Adaptable Plagiarism Detection

by

Zachary Monroe

A Thesis Presented in Partial Fulfillment
of the Requirement for the Degree
Master of Science

Approved April 2020 by the
Graduate Supervisory Committee:

Ajay Bansal, Chair
Timothy Lindquist
Ruben Acuña

ARIZONA STATE UNIVERSITY

May 2020

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. There are a number of plagiarism detection tools that attempt to encode knowledge about the programming languages they support in order to better detect obscured duplicates. Many such tools do not support a large number of languages because doing so requires too much code and therefore too much maintenance. It is also difficult to add support for new languages because each language is 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.

Kitsune attempts to remedy these issues by tying itself to Antlr, a pre-existing language recognition tool with over 200 currently supported languages. In addition, it provides an interface through which generic manipulations can be applied to the parse tree generated by Antlr. As Kitsune relies on language-agnostic structure modifications, it can be adapted with minimal effort to provide plagiarism detection for new languages. Kitsune has been evaluated for 10 of the languages in the Antlr grammar repository with success and could easily be extended to support all of the grammars currently developed by Antlr or future grammars which are developed as new languages are written.

ACKNOWLEDGEMENTS

I would like to thank Jesse Kimble for his help in proofreading and being a general second opinion whenever I needed it. I would like to acknowledge Vatricia Edgar for listening so intently whenever I needed to work through something. And finally, I would like to thank Professor Ajay Bansal, for his assistance throughout the entire project, for the many emails he exchanged with me, and for all of the time he spent to get me to where I am today.

TABLE OF CONTENTS

Page

LIST OF TABLES v

LIST OF FIGURES vi

CHAPTER

1 INTRODUCTION 1

1.1 An Introduction to the Process of Source File Plagiarism Detection . 1

1.2 Applications of Code Similarity Detection 1

1.2.1 Academic Plagiarism Detection 1

1.2.2 Commercial Plagiarism Detection 2

1.2.3 Code Clone Detection and Bug Detection 4

2 RELATED WORKS 6

2.1 An Overview of Modern Plagiarism Detection 6

2.2 Highlighted Approaches 10

2.2.1 MOSS's Approach 10

2.2.2 JPLAG's Approach 13

2.2.3 SIM's Approach 14

2.2.4 Sherlock's Approach 14

3 KITSUNE: STRUCTURE AWARE ANALYSIS 17

3.1 Parse Tree Generation using Antlr 17

3.2 Code Block Identification 19

4 KITSUNE: LANGUAGE ADAPTABILITY 26

4.1 Parse Tree Modifications using Cypher and Neo4j 26

4.2 Swapable Configurations 27

5 KITSUNE: ACCURATE COMPARISONS 29

6 METHODOLOGY 32

CHAPTER Page

6.1 Quality in Software Systems 32

6.2 Quality in Plagiarism Tools 32

6.3 Defining a Rubric 35

7 RESULTS 46

7.1 Overall Results 46

7.2 Maintainability 46

7.3 Efficiency 49

7.4 Reliability 49

7.5 Usability 53

7.6 Portability 61

7.7 Functionality 62

8 CONCLUSION 67

8.1 Achievements 67

8.2 Issues and Future Work 69

REFERENCES 73

LIST OF TABLES

Table | Page

2.1 Overview of Notable Tools . . . 7

2.2 Occurrences of Plagiarism Tools in Other Tools' Introductory Papers (Obtainable Tools in Gray) . . . 8

2.3 Approximate Classification of Employed Approaches by Surveyed Plagiarism Tools (Obtainable Tools Shown in Gray) . . . 9

6.1 Agreed Upon Characteristics of Quality Software Systems Adapted from "Quality Models in Software Engineering" by Rafa E. Al-Qutaish, PhD, 2010, Journal of American Science, 6.3, p. 175 . . . 33

6.2 A Quality Rubric for Plagiarism Detection Tools . . . 45

7.1 Runtimes Across Different Sized Test Sets . . . 50

7.2 Rubric Scores for the Five Evaluated Tools . . . 65

LIST OF FIGURES

Figure Page

2.1 Shingling Approach in MOSS's Winnowing Algorithm (MinHash) 11

2.2 Sherlock's Similarity Metric 16

2.3 Sherlock's Algorithm Converted to Set Theory is a Jaccard Approximation Like MinHash 16

3.1 Hello World Python Parse Tree 18

3.2 Kitsune's Block Detection Shown in "A3_6.py" (left) and "A3_57.py" (right) 21

3.3 JPlag's Results for "A3_6.py" (left) and "A3_57.py" (right) 22

3.4 Kitsune's Matching Skips the Second Function of "A3_6.py" (left) in Favor of the More Correct Third Function 23

3.5 MOSS's Results for "A3_6.py" (right) and "A3_57.py" (left) 24

3.6 SIM's Results for "A3_6.py" (left) and "A3_57.py" (right) 25

4.1 Cypher Query to Generically Replace all Identifiers in a Batch of Programs 27

4.2 Kitsune Configuration File for Python 3 28

7.1 Jury Presence Distributions for Chosen Problem Sets 52

7.2 JPlag's CLI Interface 54

7.3 JPlag's Comparison Interface 54

7.4 MOSS's Comparison Interface 55

7.5 Sherlock's Output 56

7.6 SIM's Output 58

7.7 Kitsune's Interface 59

7.8 Kitsune's Filtering Interface 60

7.9 Kitsune's Installer 60

Figure Page

7.10 Rubric Scores for the Five Evaluated Tools (Chart) 66

8.1 CSS for Python3 Syntax Highlighting 68

8.2 Example of Hierarchical Clustering For Gene Sequencing (Chua *et al.*, 2003) 71

Chapter 1

INTRODUCTION

1.1 An Introduction to the Process of Source File Plagiarism Detection

Plagiarism detection is a daunting task. When done among large files sets, it can be extremely difficult to maintain both efficient and accurate results (Mozgovoy *et al.*, 2005). 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. Detecting this easily becomes a language-specific problem and can be difficult to generically represent as can be seen by the lack of success of major plagiarism tools which support a large number of languages. 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 bench-marked 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.

1.2 Applications of Code Similarity Detection

1.2.1 Academic Plagiarism Detection

As the main aim of this thesis, it is critical to understand the needs of an academic environment. The main setting for plagiarism detection tools in academic environments is for alleviating the difficulty teachers face in identifying students working

together. There are a number of reasons teachers might look to a tool for this, most stemming from the difficulty of manually sifting through students submissions and finding matches. Large programs can be difficult to remember and thus identify copied segments, especially when there are a large number of students (Hage *et al.*, 2010). On top of this, teachers often grade with the help of a TA and are not able to look over every assignment themselves. Finally, many times plagiarism is not visually apparent and can be obscured by simple changes which make code appear visually different but semantically identical. Coupled with the problem that in the end, the goal of the students is to produce a program which often has identical aims, a fourth problem arises in that students are to some degree *expected* to have similar responses to homework assignments (Schleimer *et al.*, 2003) (Hage *et al.*, 2010). These responses should, however, demonstrate that the student has come up with their own, generally unique solution (Hage *et al.*, 2010) (Clough, 2000). More importantly, the student should not have collaborated with any other student working on the same solution when prohibited from doing so (Clough, 2000). Detecting this requires balancing some level of expected similarity on assignments with an expectation that not everything should be the same between two students. There is, however, an expectation that file sets submitted to such tools will not be generally massive in size (students are not, in fact, able to produce industry-sized-applications in the short time span that most university classes provide). This allows for certain assumptions which may not be applicable in other environments that afford additional accuracy.

1.2.2 Commercial Plagiarism Detection

In comparison to an academic environment, commercial plagiarism detection tools have a much different set of problems. Usually, there are far fewer entities (students/-companies) which need to have code compared, but the size of the code sets is usually

much larger at a commercial level (Ragkhitwetsagul, 2018). There is also a much different expectation for the degree of expected similarity (Zeidman, 2010) (Ragkhitwetsagul, 2018). Usually, companies are not expected to produce the exact same result and thus any resemblance should be considered more of an issue than general structural similarities a student answering a strict prompt might incur (Ragkhitwetsagul, 2018). Depending on the situation, commercial-level plagiarism detection tools may also face a number of other limitations including: a lack of available source code (only the binaries may be available) (Luo *et al.*, 2014), an expectation of *highly* accurate results as in the case of law suits between companies (Zeidman, 2010), interweaving of open source code or frameworks (Ragkhitwetsagul, 2018), and many more limitations specific to each case. Often, companies who are searching for such plagiarism will do so either when they believe another party has wrongfully utilized their code against licensing restrictions and wishes to prove it (Zeidman, 2010) (Ragkhitwetsagul, 2018). In some cases, companies will also search through their own code in an attempt to prevent such cases from arising accidentally (Luo *et al.*, 2014).

Commercial plagiarism cases can also often be extremely high profile. This further stresses the need for accurate results. Famous court cases such as Google v. Oracle which began in 2010 (Ragkhitwetsagul, 2018) and is currently still active and awaiting trial in the Supreme Court (Oracle, 2018) have pointed out the major ramification that plagiarism can have on companies. Oracle's lawsuit for over 9 billion US dollars hinges on evidence of plagiarism from 12 source files and 37 Java specifications (Ragkhitwetsagul, 2018). In more recent times, cases like Sonos v. Google have further evidenced the impact of plagiarism yet again, with Sonos claiming Google stole their code while working with them to implement the Google Assistant feature before Google had begun manufacturing smart speakers of their own (Sonos, 2020). Cases such as these have driven companies to great lengths to both prove and prevent

such lawsuits. In pursuit of such tools, the Computing and Communication Foundations Division of the National Science Foundation awarded CoP, one such commercial plagiarism tool, a $500,000 grant to pursue research into the detection of such cases (their paper cites IBM's $400 million settlement Compuware as a driving factor in their research) (Luo *et al.*, 2014).

1.2.3 Code Clone Detection and Bug Detection

A third application of source file similarity detection is in code clone detection and bug detection. Rather than looking for similarities across files or across related projects, many tools aim to reduce code clutter by targeting duplicate code. This is inherently a very similar problem, but minor differences change the overarching goals enough that certain aspects are different. For one thing, many code clone detection tools are aimed at finding clones in extremely large projects. Multiple of the duplication tools that were surveyed have been tested on major projects such as GCC and the Linux Kernel ranging up to 4 million lines of code (Ducasse *et al.*, 1999) (Kamiya *et al.*, 2002) (Li *et al.*, 2004). Not only is the scale of duplication programs often different, the aim sometimes differs as well. There is a widely accepted notion of different "levels" of similarity in duplicated code (Ragkhitwetsagul, 2018). A type 1 duplication is identical (Ragkhitwetsagul, 2018). A type 2 duplication changed identifiers (Ragkhitwetsagul, 2018). A type 4 duplication is actually different enough that the code might have a completely different algorithm that achieves the same purpose (Ragkhitwetsagul, 2018). This level of similarity is useful for abstraction in projects. Think for example, if a tool identified two places where a sorting algorithm was implemented. One might be able to simply keep only the better sorting algorithm and remove the other code (Ragkhitwetsagul, 2018). In an academic environment however, if two students implement the same goal (i.e. complete the assignment

according to instruction) this ”duplication” can be safely ignored (in fact it should often be expected!).

Chapter 2

RELATED WORKS

2.1 An Overview of Modern Plagiarism Detection

Almost every English class uses some form of plagiarism detection to detect students copying from other students' papers. While these tools exist for coding classes, perhaps due to the difficulty of the problem, there are few practical solutions. In 2015, a survey of major plagiarism tools mentioned six tools intended for program analysis including MOSS, JPlag, GPlag, Marble, Plaggie, and SIM (Ahmed, 2015). Another recent survey of software analysis tools published in 2016 listed the major tools considered for their study as NICAD, Plague, YAP, CoP, Sherlock, Sim, JPlag, CCFinder, CPMiner, iClones, and Moss (Ragkhitwetsagul *et al.*, 2016). A more recent thesis on the topic from 2018 selected JPlag, Sherlock, SIM, and Plaggie as their primary tools to investigate, but included mentions of 12 other tools: YAP, CCFinder, MOSS, NICAD, CoP, GPlag, SID, Accuse, Duploc, Simian, CPD, and iClones (Ragkhitwetsagul, 2018). Of these three surveys, only one tool was published in the last twelve years and of the 24 tools investigated, only six were known to have been updated within the last five years, three of which are intended for single project duplicate code detection, not cross-project comparisons (ref. table 2.1). Of the tools analyzed, this leaves three tools which are active and which can be used by teachers to detect plagiarism: JPlag, MOSS, and SIM. These tools (especially MOSS and JPlag, ref. table 2.2) are also often commonly cited by other tools as the "state of the art" in plagiarism detection.

Tool	**Public**	**Launch**	**Last Update**	**# Lang.**	**Purpose**
Accuse	No	1981	Unknown	2	Plagiarism
CCFinder	Binary	2002	2009	7	Duplicated Code
CodeMatch	Binary	2004	Unknown	Text	Plagiarism
Cogger	No	1993	Unknown	1	Plagiarism
CoP	No	2014	Unknown	Binary	Comm. Plagiarism
CPD	Yes	2002	2020	15	Refactoring/Bugs
CPMiner	No	2004	Unknown	2	Duplicated Code
Duploc	No	1999	Website Down	Text	Duplicated Code
FPDS	No	2005	Unknown	1	Plagiarism
Gplag	No	2006	Unknown	3	Comm. Plagiarism
Jones	No	2001	Unknown	1	Plagiarism
Jplag	Yes	2000	2019	6	Plagiarism
Marble	No	2002	Website Down	4	Plagiarism
Moss	Service	1994	2018	25	Plagiarism
NICAD	Yes	2008	2015	4	Duplicated Code
Pdetect	Yes	2004	Website Down	1	Plagiarism
Plaggie	Binary	2006	2006	1	Plagiarism
PlagioGuard	No	2008	Unknown	2	Plagiarism
Plague	No	1990	Website Down	Unknown	Plagiarism
Sherlock	Yes	1994	Website Down	Text	Plagiarism
SID	Yes	2004	Website Down	2	Plagiarism
SIM	Yes	1989	2017	8	Plagiarism
Simian	Binary	2003	2018	Text	Refactoring/Bugs
Yap3	No	1996	Website Down	Unknown	Plagiarism

Table 2.1: Overview of Notable Tools

Tool

Paper	Accuse	CCFinder	CodeMatch	Cogger	CoP	CPD	CPMiner	Duploc	FPDS	Gplag	Jones	Jplag	Marble	Moss	NICAD	Pdetect	Plaggie	PlagioGuard	Plague	Sherlock	SID	SIM	Simian	Yap3
Accuse (Grier, 1981)	X																							
CCFinder (Kamiya *et al.*, 2002)		X						✓																
CodeMatch (Zeidman, 2010)			X																					
Cogger (Cunningham, 1993)				X																				
CoP (Luo *et al.*, 2014)					X							✓		✓										
CPD (No Paper)						X																		
CPMiner (Li *et al.*, 2004)		✓					X	✓				✓		✓										
Duploc (Ducasse *et al.*, 1999)								X																
FPDS (Mozgovoy *et al.*, 2005)									X			✓		✓						✓				
GPlag (Liu *et al.*, 2006)										X		✓		✓										
Jones (Jones, 2001)											X											✓		✓
Jplag (Prechelt *et al.*, 2002)												X		✓										✓
Marble (Hage, 2006)													X											
Moss (Schleimer *et al.*, 2003)														X										
NICAD (Roy and Cordy, 2008)		✓						✓							X									
Pdetect (Moussiades and Vakali, 2005)												✓		✓		X								✓
Plaggie (Ahtiainen *et al.*, 2006)												✓		✓			X			✓		✓		
PlagioGuard (Goel *et al.*, 2008)			✓									✓		✓				X	✓	✓				
Plague (Whale, 1990)																			X					
Sherlock (Joy and Luck, 1999)																			✓	X				✓
SID (Chen *et al.*, 2004)												✓		✓			✓				X			✓
SIM (Grune and Huntjens, 1989)																						X		
Simian (No Paper)																							X	
Yap3 (Wise, 1996)	✓																		✓					X
Totals	**1**	**2**	**1**	**0**	**0**	**0**	**0**	**3**	**0**	**0**	**0**	**8**	**0**	**9**	**0**	**0**	**1**	**0**	**3**	**3**	**0**	**2**	**0**	**5**

Table 2.2: Occurrences of Plagiarism Tools in Other Tools' Introductory Papers (Obtainable Tools in Gray)

Tool	**Text**	**Token**	**Graph**	**Metrics**	**Other**
Accuse	✓			✓	
CCFinder		✓			
CodeMatch	✓				
Cogger				✓	✓(“AI”/Statistical)
CoP			✓(CFG)	✓	✓(Binary)
CPD					
CPMiner		✓			✓(F.P. Mining)
Duploc	✓	✓			
FPDS					
Gplag			✓(PDG)		
Jones				✓	
Jplag		✓			
Marble	✓				
Moss	✓	✓			
NICAD	✓				
Pdetect	✓		✓(Clustering)		
Plaggie		✓			
PlagioGuard	✓				
Plague				✓	
Sherlock	✓				
SID		✓			
SIM		✓			
Simian					Undisclosed
Yap3	✓	✓			

Table 2.3: Approximate Classification of Employed Approaches by Surveyed Plagiarism Tools (Obtainable Tools Shown in Gray)

2.2 Highlighted Approaches

While many of the tools surveyed took a wide variety of approaches (ref. table 2.3), it is possible to classify these approaches under a number of generic categories. Many contemporary tools, for example, rely on a parsing engine generating tokens which either represent language level tokens or more abstract groupings of terms (Prechelt *et al.*, 2002) (Grune and Huntjens, 1989). Other tools rely on language agnostic text-based comparisons. Some rely on graph-based analysis such as control flow, program dependency, or graph clustering. There were a number of (mostly older) tools which relied on metric based analysis such as counts of identifiers or method complexities. Often this was done due to computational constraints of more detailed analysis (Grier, 1981). There were also a number of approaches that lie outside of common methods such as analysis of the binaries produced, and AI/data science techniques like frequent pattern mining. All of the available tools which can be tested fall into the categories of text and token-based comparisons. The three most up-to-date obtainable tools, MOSS, JPlag, and SIM, all rely heavily on token based comparisons. Out of the text-focused tools, Sherlock seems to be the most well known (ref. Table 2.2 and cited tool which also focuses on plagiarism detection (ref. Table 2.1).

2.2.1 MOSS's Approach

Moss's approach is more heavily intertwined with modern similarity analysis research. Their approach, "Winnowing", is a code-focused improvement upon a major text similarity algorithm called MinHash. The basis for this approach was first proposed in 1997 by Andrei Broder (p. 2), a computer scientist working for Google in 1997 in a paper entitled "Syntactic Clustering of the Web." The intent of this paper

was to provide “[A] mechanism for discovering when two documents are ‘roughly the same’; that is, for discovering when they have the same content except for modifications such as formatting, minor corrections, webmaster signature, or logo” in order to deal with “[t]he proliferation of documents that are identical or almost identical” on the web. Broder’s algorithm worked by breaking the documents into an overlapping windows called “shingles” of 10 words. For example, the sentence “The quick brown fox jumps over the lazy dog” with a shingle size of 3 would contain the shingles “The quick brown”, “quick brown fox”, “brown fox jumps”, etc. as in Figure 2.1.

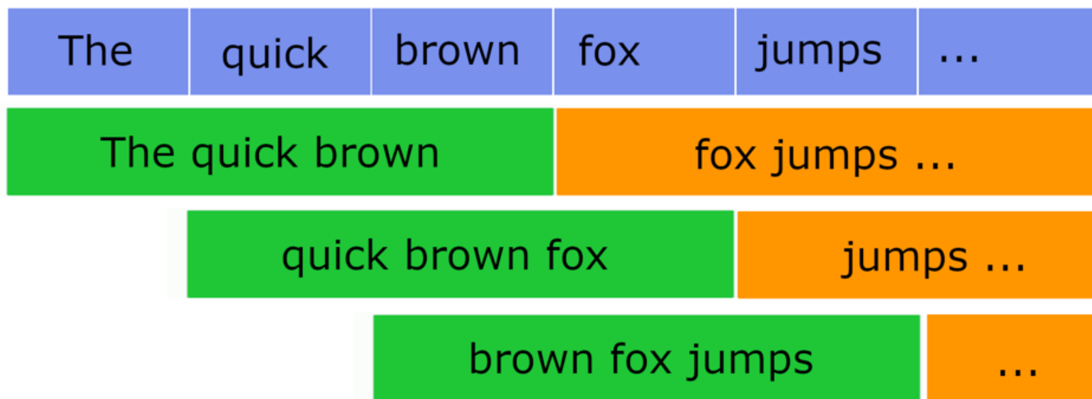

Figure 2.1: Shingling Approach in MOSS’s Winnowing Algorithm (MinHash)

Each of the shingles is then passed through a hash function. This then allows samples to be randomly selected based on the hash output value such that the random choice would be consistent across documents. For example, by choosing all samples divisible by 25, as Broder did, or by taking the smallest N samples as is typically done in modern implementations of MinHash (such as MOSS’s). This “sampling” is useful because it works as an accurate estimate for the similarity of the document with a much lower cost; Once the tokens are selected, the time to compare any 2 documents is the same, independent of size.

There are a number of issues with directly applying MinHash to programs which MOSS attempted to solve. One issue, noted in Broder's own paper, is the difference between "containment" and "resemblance" of documents. A plagiarism detection tool needs to be able to find sections of the code which are contained as opposed to detect the resemblance of an entire program. Students could easily copy only a section (like a single function) and the "resemblance" of a large program might remain low (Schleimer *et al.*, 2003). MOSS attempts to remedy this issue by pulling samples from a second sliding window such that at least one sample comes from each window. MOSS also attempts to rate programs on the number of concurrent lines which contain similar text at a later post processing step. This allows users to view sections of contained code as opposed to merely the resemblance of overall documents. MOSS's source code is not public, but the authors have made a number of statements about MOSS's behaviour outside of their 2003 paper on the general approach.

> Moss is quite conservative about what it considers to be matching passages of code. If Moss says that two passages look alike, then they almost certainly look quite alike. Moss also excludes all code that appears in too many of the submitted programs. Thus, all matches reported by Moss fairly accurately approximate the signature of plagiarized code: a passage of similar code in two programs that does not also appear in very many other programs. (Aiken, 2018)

These comments lend towards the idea that there is a further modification of the similarity percentages such that minor similarities are excluded beyond what an implementation of the Winnowing algorithm presented in their 2003 paper would generate.

MOSS also applies a number of other techniques such as removing unneeded characters like whitespace, renaming of volatile but meaningless tokens like variable names (students can easily change these with no effect on the outcome of a program to hide plagiarism), etc. Though MOSS attempts to provide some language intelligence, all steps taken to do so occur at the stage of token preparation before the similarity comparison occurs (Schleimer *et al.*, 2003). All comparisons are done text-wise as if on any piece of text.

2.2.2 JPLAG's Approach

JPlag's approach performs a higher level of abstraction upon the programs it compares. Whereas MOSS relied on any language intelligence primarily for token identification and renaming (i.e. variables), JPLAG's frontend is connected to ANTLR4, a language recognition tool primarily intended for compiler development. JPLAG uses custom grammars written for ANTLR4 along with a small amount of frontend code (around 1000 lines each for the 5 currently supported languages) to generate a token stream (Prechelt *et al.*, 2002). These token streams are not tokens as represented by the programming language, but logical groups of language level tokens that form constructs (i.e. "public class Name {" in Java is not 4 tokens, but a single token: "BEGINCLASS") (Prechelt *et al.*, 2002).

JPlag's method for comparing the token streams, much like MOSS, employs previously researched string comparison algorithms. JPlag uses a common fuzzy string comparison method, "greedy string tilling" introduced by Michael Wise in 1993 (Prechelt *et al.*, 2002). Greedy String Tilling attempts to match sections of length N starting at an upper bound and iterating downwards (Prechelt *et al.*, 2002). Once a match is found, those characters are bound and removed from possible characters

remaining to match (Prechelt *et al.*, 2002). In JPlag, instead of characters, the tokens are compared.

2.2.3 SIM's Approach

Dick Grune's SIM has a much less formalized approach. There is a small report which details its behaviour, but Grune himself has referred to the report as a "(probably obsolete) terse technical report" (Grune and Huntjens, 1989). Looking at the code itself, and using the old technical report as a supplemental reference, it seems that the general description is approximately correct. The algorithm consists of what is essentially five steps. These steps are: tokenizing the files, create a forward reference table, finding "runs", finding the corresponding newlines, and generating the similarity report.

SIM generates tokens similarly to JPlag. While JPlag uses a more established parser engine, it should be noted that SIM was originally published 11 years before JPlag. SIM's use of the parsing engine is much more direct – whereas JPlag reduces token sequences to more abstract blocks, SIM leaves tokens to their language equivalent and performs a direct comparison. The next step generates a lookup table for sequence positions. This allows the third step, "finding runs", to perform an optimized form of longest common subsequence matching. The token indexes which were identified as the start and ends of "runs" (sequences) are then converted into line numbers. Finally, the report can be generated in a readable "diff" style format.

2.2.4 Sherlock's Approach

Unlike the other four approaches, Sherlock does not employ any language-specific mechanism for generating tokens or renaming values. Most papers about Sherlock are rather unspecific about the approach that Sherlock employs beyond this. Looking

at the comments left by the authors in Sherlock's open source source code, Sherlock's approach seems to be an approximation of what MinHash attempts to do. First, Sherlock reads in "words" by delimiting on common punctuation symbols and whitespace, generating a list. Second, it takes 'N' consecutive "words" and hashes these. It then randomly discards a number of those hashes. The "set size consecutive sequences" Sherlock collects are known in MinHash as shingles. Furthermore, one of the comments (ref. Figure 2.2) describes their metric for similarity as a percentage of the similar/shared words divided by the sum of the count of words in each file minus the shared words. This value can be seen as the intersection of the two sets of tokens from each of the programs, "the shared", divided by the union of the two sets (ref. Figure 2.3). Adding the size of two sets and subtracting the intersection produces the unions by removing the duplicates. This metric, known as the Jaccard Index, is the same value that MinHash attempts to produce. Instead of selecting the smallest 'N' hashes, Sherlock discards in a more arbitrary fashion.

Sherlock is very similar to MOSS, but rather than adding language sense to detect tokens, it relies on common traits of programming languages such as using whitespace, parenthesis, etc. for separating tokens. This allows it more flexibility than any of the other tools by trading away some of the features such as variable renaming.

```
...
        return 100 * nsimilar / (s0->nval + s1->nval - nsimilar);
}

/*
 *  Let f1 == filesize(file1) == A+B
 *  and f2 == filesize(file2) == A+C
 *  where A is the similar section and B or C are dissimilar
 *
 *  Similarity = 100 * A / (f1 + f2 - A)
```

Figure 2.2: Sherlock's Similarity Metric

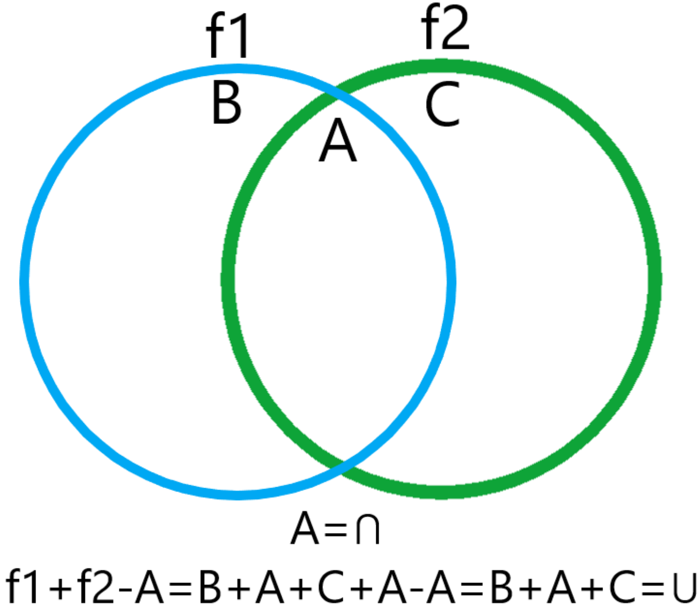

Figure 2.3: Sherlock's Algorithm Converted to Set Theory is a Jaccard Approximation Like MinHash

Chapter 3

KITSUNE: STRUCTURE AWARE ANALYSIS

3.1 Parse Tree Generation using Antlr

Both JPlag and Kitsune rely on Antlr for some part of their pipeline. Despite this, the *ways* in which the parser is used is quite different. JPlag focuses on using Antlr 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 Antlr is not utilized. Kitsune uses the tokens and the tagging done by Antlr, 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.

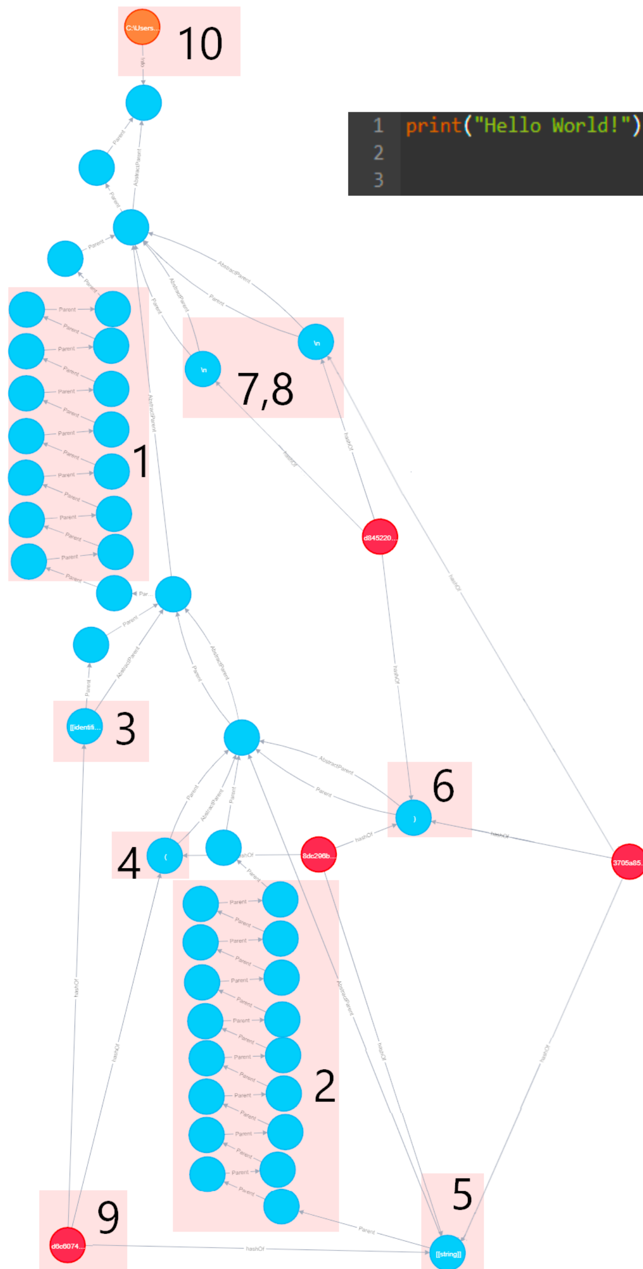

Figure 3.1: Hello World Python Parse Tree

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 Antlr and abstracts a number of things out. In Figure 3.1, for example, the tree contains many redundant nodes created while descending through the grammar rules (see 3.1, numbers 1 and 2). These nodes do not contain useful information for Kitsune as they are generated by traversing the precedence rules from the grammar. After this is done, the relationships labeled "Parent" are no longer used and queries are performed using the more minimal twelve node tree between "AbstractParent" relationships. The other nodes are abstracted out 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: print (3), '(' (4), "Hello World!" (5), ')' (6), and the two newlines (7 and 8). For the current configuration of python3, both the 'print' token and "Hello World!" token are then reduced to a generic identifier and string token respectively.

In a small program like hello world, this reduces a large amount of the state of the program. In a larger program 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. A sliding hash window of three tokens is then generated. For example, the tokens 'print', '(', and '[[string]' are then hashed (see the red node number 9). Currently, Kitsune shares a similar algorithm to both Sherlock and MOSS where these hashes are coalesced, sorted, and the minimum hashes are selected. The sample size is chosen such that Kitsune has an accuracy on any given comparison of 95%. 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.

3.2 Code Block Identification

Due to the way Kitsune represents source files internally using the tree that was generated by Antlr, Kitsune has a number of advantages over many of the other tools. The following example highlights this behaviour. 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 (seen in 3.2, 3.3, 3.5,

and 3.6), 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 (see 3.3) 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 on the right in 3.3 and 3.2) as being plagiarised 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 (the second match from Kitsune can be seen in 3.4).

Kitsune

File Edit View Window Help

LANGUAGES TEMPLATES UI DATABASE

Menu Theme: Material Black Editor Theme: dark

```
29         def get_next(point1, point2):
30                 list_temp = [(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1),
31                      (1, 0), (1, 1)]
32          list_temp = list(filter
33                           (lambda i: 0 <= (i[0] + point1) < m and (i[1] +
34                            point2) >= 0 and n > (i[1] + point2), list_temp))
35          return [pos(c=(point1 + k[0]),
36                     d=(point2 + k[1])) for k in list_temp]
37
38       for i in range(m):
39                 for j in range(n):
40                         Graph.update({pos(c=i, d=j): temp(a=False, b=G[i][j])})
41
42        all_items = Graph.items()
43      for i, j in all_items:
44                 if not j.a and j.b == '#':
45                         traverseAhead(i)
46              counter = counter + 1
47
48        return counter
49
50 # ======================= Longest Ordered Subsequence =======================
51
52
53
54 def longest_ordered_subsequence(L):
55         n = len(L)
56     if n == 0:
57                 return 0
58      A = [0] * n
59     longest = 1
60     A[0] = 1
61
62     for x in range(1, n):
63                 for y in range(x):
64                         temp = 1 + A[y]
65             if L[y] < L[x] and temp > A[x]:
66                                 A[x] = temp
67                 if longest < A[x]:
68                                         longest = temp
69         return longest
70
71 # =============================== Supermarket ================================
72
```

```
66         moves = list(filter(lambda move: (x + move[0]) >= 0 and
67                                           (x + move[0]) < m and
68                                           (y + move[1]) >= 0 and
69                                           (y + move[1]) < n, moves))
70         finalmoves = [index(x=(x + move[0]),
71                             y=(y + move[1])) for move in moves]
72         # print(finalmoves)
73         return finalmoves
74
75      def pondExplore(key):
76                 nonlocal TEMP
77         TEMP[key] = check(v=True, e=TEMP[key].e)
78         moves = adjacentGen(key.x, key.y)
79         for move in moves:
80                         if not TEMP[move].v and TEMP[move].e == WATER:
81                                 pondExplore(move)
82
83        TEMP = {}
84
85     for i in range(m):
86                 for j in range(n):
87                         TEMP.update({index(x=i, y=j): check(v=False, e=G[i][j])})
88             # print(TEMP)
89
90       pondcount = 0
91     for key, value in TEMP.items():
92                 if not value.v and value.e == WATER:
93                         pondExplore(key)
94             # print(pondExplore)
95             pondcount += 1
96
97       return pondcount
98
99 # =============================== Supermarket ================================
100
101
102
103 def supermarket(Items):
104         n = len(Items)
105
106     # base case check
107     if n == 0:
108                 return 0
109
110      if n == 1:
```

Figure 3.2: Kitsune's Block Detection Shown in "A3_6.py" (left) and "A3_57.py" (right)

Matches for A3_6.py & A3_57.py

48.3%

INDEX - HELP

	A3_6.py (46.81818%)	A3_57.py (50.0%)	Tokens
	A3_6.py(21-27)	A3_57.py(74-80)	22
	A3_6.py(29-34)	A3_57.py(59-67)	18
	A3_6.py(38-42)	A3_57.py(84-89)	18
	A3_6.py(44-56)	A3_57.py(91-106)	15

```
    def get_next(point1, point2):
        list_temp = [(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1),
                     (1, 0), (1, 1)]
        list_temp = list(filter
                         (lambda i: 0 <= (i[0] + point1) < m and (i[1] +
                          point2) >= 0 and n > (i[1] + point2), list_temp))
        return [pos(c=(point1 + k[0]),
                    d=(point2 + k[1])) for k in list_temp]

    for i in range(m):
        for j in range(n):
            Graph.update({pos(c=i, d=j): temp(a=False, b=G[i][j])})

    all_items = Graph.items()
    for i, j in all_items:
        if not j.a and j.b == '#':
            traverseAhead(i)
            counter = counter + 1

    return counter

# ======================= Longest Ordered Subsequence =======================


def longest_ordered_subsequence(L):
    n = len(L)
    if n == 0:
        return 0
    A = [0] * n
    longest = 1
    A[0] = 1

    for x in range(1, n):
        for y in range(x):
            temp = 1 + A[y]
            if L[y] < L[x] and temp > A[x]:
                A[x] = temp
                if longest < A[x]:
                    longest = temp
    return longest

# =============================== Supermarket ================================


def supermarket(Items):
    n = len(Items)
    if n == 0:
        return n
    if n == 1:
        return Items[0][0]

    optimal_profit = 0
    Items = sorted(Items, reverse=True)
    consumed = [False] * (max([i[1] for i in Items]) + 1)
```

```
        moves = list(filter(lambda move: (x + move[0]) >= 0 and
                                         (x + move[0]) < m and
                                         (y + move[1]) >= 0 and
                                         (y + move[1]) < n, moves))
        finalmoves = [index(x=(x + move[0]),
                            y=(y + move[1])) for move in moves]
        # print(finalmoves)
        return finalmoves

    def pondExplore(key):
        nonlocal TEMP
        TEMP[key] = check(v=True, e=TEMP[key].e)
        moves = adjacentGen(key.x, key.y)
        for move in moves:
            if not TEMP[move].v and TEMP[move].e == WATER:
                pondExplore(move)

    TEMP = {}

    for i in range(m):
        for j in range(n):
            TEMP.update({index(x=i, y=j): check(v=False, e=G[i][j])})
            # print(TEMP)

    pondcount = 0
    for key, value in TEMP.items():
        if not value.v and value.e == WATER:
            pondExplore(key)
            # print(pondExplore)
            pondcount += 1

    return pondcount

# ============================== Supermarket ======================


def supermarket(Items):
    n = len(Items)

    # base case check
    if n == 0:
        return 0

    if n == 1:
        return Items[0][0]

    Items = sorted(Items, reverse=True)
    # print(Items)
    optimumSell = 0
    lenItems = max(x[1] for x in Items)
    # print(lenItems)
    used = [False] * (lenItems + 1)
    # print(used)
    for item in Items:
        if not used[item[1]]:
```

Figure 3.3: JPlag's Results for "A3_6.py" (left) and "A3_57.py" (right)

```
50  # ====================== Longest Ordered Subsequence ======================
51
52
53  def longest_ordered_subsequence(L):
55          n = len(L)
56      if n == 0:
57                  return 0
58      A = [0] * n
59      longest = 1
60      A[0] = 1
61
62      for x in range(1, n):
63                  for y in range(x):
64                          temp = 1 + A[y]
65              if L[y] < L[x] and temp > A[x]:
66                                  A[x] = temp
67                  if longest < A[x]:
68                                          longest = temp
69      return longest
70
71  # ============================== Supermarket ==============================
72
73
74  def supermarket(Items):
76          n = len(Items)
77      if n == 0:
78                  return n
79      if n == 1:
80                  return Items[0][0]
81
82      optimal_profit = 0
83      Items = sorted(Items, reverse=True)
84      consumed = [False] * (max([i[1] for i in Items]) + 1)
85
86      for i in Items:
87                  if not consumed[i[1]]:
88                          consumed[i[1]] = True
89              optimal_profit = optimal_profit + i[0]
90          else:
91                          for i in range(i[1], 0, -1):
92                                  if not consumed[i]:
93                                          consumed[i[1]] = True
94                      optimal_profit = optimal_profit + i[0]
95                      break
96      return optimal_profit
97
```

```
85      for i in range(m):
86                  for j in range(n):
87                          TEMP.update({index(x=i, y=j): check(v=False, e=G[i][j])})
88              # print(TEMP)
89
90      pondcount = 0
91      for key, value in TEMP.items():
92                  if not value.v and value.e == WATER:
93                          pondExplore(key)
94              # print(pondExplore)
95              pondcount += 1
96
97      return pondcount
98
99  # ============================== Supermarket ==============================
100
101
102 def supermarket(Items):
104         n = len(Items)
105
106     # base case check
107     if n == 0:
108                 return 0
109
110     if n == 1:
111                 return Items[0][0]
112
113     Items = sorted(Items, reverse=True)
114     # print(Items)
115     optimumSell = 0
116     lenItems = max(x[1] for x in Items)
117     # print(lenItems)
118     used = [False] * (lenItems + 1)
119     # print(used)
120     for item in Items:
121                 if not used[item[1]]:
122                         optimumSell += item[0]
123             used[item[1]] = True
124         else:
125                         for i in range(item[1], 0, -1):
126                                 if not used[i]:
127                                         optimumSell += item[0]
128                     used[item[1]] = True
129                     break
130     return optimumSell
```

Figure 3.4: Kitsune's Matching Skips the Second Function of "A3_6.py" (left) in Favor of the More Correct Third Function

./PYTH_3.6/A3/A3_57.py (19%)		./PYTH_3.6/A3/A3_6.py (18%)	
54-60		26-30	
69-74		18-23	
79-81		35-37	

```
def count_ponds(G):

    m = len(G)
    n = len(G[0])

    def adjacentGen(x, y):
        moves = [(-1, -1), (-1, 0), (-1, 1), (0, -1),
                 (0, 1), (1, -1), (1, 0), (1, 1)]

        # list of allowed moves

        moves = list(filter(lambda move: (x + move[0]) >= 0 and
                                         (x + move[0]) < m and
                                         (y + move[1]) >= 0 and
                                         (y + move[1]) < n, moves))

        finalmoves = [index(x=(x + move[0]),
                            y=(y + move[1])) for move in moves]
        # print(finalmoves)
        return finalmoves

    def pondExplore(key):
        nonlocal TEMP
        TEMP[key] = check(v=True, e=TEMP[key].e)
        moves = adjacentGen(key.x, key.y)
        for move in moves:

            if not TEMP[move].v and TEMP[move].e == WATER:
                pondExplore(move)

    TEMP = {}

    for i in range(m):
        for j in range(n):
            TEMP.update({index(x=i, y=j): check(v=False, e=G[i][j])})
            # print(TEMP)

    pondcount = 0
    for key, value in TEMP.items():
        if not value.v and value.e == WATER:
            pondExplore(key)
            # print(pondExplore)
            pondcount += 1

    return pondcount

# ============================== Supermarket ===============================
```

```
def count_ponds(G):
    m = len(G)
    n = len(G[0])
    Graph = {}
    counter = 0

    temp = nt('temp', ['a', 'b'])
    pos = nt('pos', ['c', 'd'])

    def traverseAhead(key):
        nonlocal Graph
        Graph[key] = temp(a=True, b=Graph[key].b)
        moves = get_next(key.c, key.d)
        for move in moves:

            if not Graph[move].a and Graph[move].b == '#':
                traverseAhead(move)

    def get_next(point1, point2):
        list_temp = [(-1, -1), (-1, 0), (-1, 1), (0, -1), (0, 1), (1, -1),
                     (1, 0), (1, 1)]
        list_temp = list(filter
                         (lambda i: 0 <= (i[0] + point1) < m and (i[1] +
                          point2) >= 0 and n > (i[1] + point2), list_temp))

        return [pos(c=(point1 + k[0]),
                    d=(point2 + k[1])) for k in list_temp]

    for i in range(m):
        for j in range(n):
            Graph.update({pos(c=i, d=j): temp(a=False, b=G[i][j])})

    all_items = Graph.items()
    for i, j in all_items:
        if not j.a and j.b == '#':
            traverseAhead(i)
            counter = counter + 1

    return counter

# ====================== Longest Ordered Subsequence ======================

def longest_ordered_subsequence(L):
    n = len(L)
    if n == 0:
        return 0
    A = [0] * n
    longest = 1
    A[0] = 1
```

Figure 3.5: MOSS's Results for "A3_6.py" (right) and "A3_57.py" (left)

```
./PYTH_3.6/A3/A3_6.py: line 27-33      |./PYTH_3.6/A3/A3_57.py: line 57-65 [75]
  traverseAhead(move)                  |  n = len(G[0])
                                       |
  def get_next(point1, point2):        |  def adjacentGen(x, y):
  list_temp = [(-1, -1), (-1, 0), (-1, |  moves = [(-1, -1), (-1, 0), (-1, 1),
  (1, 0), (1, 1)]                      |  (0, 1), (1, -1), (1, 0), (1, 1)]
  list_temp = list(filter              |
  (lambda i: 0 <= (i[0] + point1) < m a|  # list of allowed moves
                                       |
                                       |  moves = list(filter(lambda move: (x +

./PYTH_3.6/A3/A3_6.py: line 21-26      |./PYTH_3.6/A3/A3_57.py: line 74-79 [62]
  def traverseAhead(key):              |  def pondExplore(key):
  nonlocal Graph                       |  nonlocal TEMP
  Graph[key] = temp(a=True, b=Graph[key|  TEMP[key] = check(v=True, e=TEMP[key]
  moves = get_next(key.c, key.d)       |  moves = adjacentGen(key.x, key.y)
  for move in moves:                   |  for move in moves:
  if not Graph[move].a and Graph[move].|  if not TEMP[move].v and TEMP[move].e

./PYTH_3.6/A3/A3_6.py: line 38-42      |./PYTH_3.6/A3/A3_57.py: line 84-89 [52]
  for i in range(m):                   |  for i in range(m):
  for j in range(n):                   |  for j in range(n):
  Graph.update({pos(c=i, d=j): temp(a=F|  TEMP.update({index(x=i, y=j): check(v
                                       |  # print(TEMP)
  all_items = Graph.items()            |
                                       |  pondcount = 0

./PYTH_3.6/A3/A3_6.py: line 35-36      |./PYTH_3.6/A3/A3_57.py: line 69-70 [30]
  return [pos(c=(point1 + k[0]),       |  finalmoves = [index(x=(x + move[0]),
  d=(point2 + k[1])) for k in list_temp|  y=(y + move[1])) for move in moves]

./PYTH_3.6/A3/A3_6.py: line 139-148    |./PYTH_3.6/A3/A3_57.py: line 135-144[27]
  if count_ponds(["-----###----",      |  if count_ponds(["#--------##-",
  "-----###----",                      |  "-###-----###",
  "------------",                      |  "----##---##-",
  "------------",                      |  "---------##-",
  "------------",                      |  "---------#--",
  "-###--------",                      |  "--#------#--",
  "-###--------",                      |  "-#-#-----##-",
  "------------",                      |  "#-#-#-----#-",
  "------------",                      |  "-#-#------#-",
  "------------"]) == 2:               |  "--#-------#-"]) == 3:

./PYTH_3.6/A3/A3_6.py: line 88-90      |./PYTH_3.6/A3/A3_57.py: line 122-124[25]
  else:                                |  else:
  for i in range(i[1], 0, -1):         |  for i in range(item[1], 0, -1):
  if not consumed[i]:                  |  if not used[i]:

./PYTH_3.6/A3/A3_6.py: line 18-19      |./PYTH_3.6/A3/A3_57.py: line 49-50 [24]
  temp = nt('temp', ['a', 'b'])        |check = namedtuple('check', ['v', 'e'])
  pos = nt('pos', ['c', 'd'])          |index = namedtuple('index', ['x', 'y'])
```

Figure 3.6: SIM's Results for "A3_6.py" (left) and "A3_57.py" (right)

Chapter 4

KITSUNE: LANGUAGE ADAPTABILITY

4.1 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 Antlr into a Neo4j database, most of the modifications to the tree became fairly simple. Figure 4.2, 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.

```
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}
```

Figure 4.1: Cypher Query to Generically Replace all Identifiers in a Batch of Programs

4.2 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 contains about 8 lines of code that change and a set export statement that takes 7 lines (see 4.2). Thus a new language takes 8 lines of 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.

```
 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 | };
```

Figure 4.2: Kitsune Configuration File for Python 3

Chapter 5

KITSUNE: ACCURATE COMPARISONS

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 (Plank, 2019). 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 (Plank, 2019). The latter is a more recent development which builds on the idea of sampling without replacement (Plank, 2019). Usually, the simpler version (former) is used as an upper bound for the sample size required to estimate the similarity (Plank, 2019). This version has a required sample size of $1/E^2$ where E is the desired error margin (Plank, 2019). This version acts as a sampling-with-replacement type problem while the N-minimum version acts as a sampling-without-replacement (Plank, 2019). 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 is approached however, this is no longer true (Plank, 2019). 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 (Plank, 2019). 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 can not 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

$$s = \frac{1.96^2(.5)(1-.5)}{.05^2} = \lceil 384.16 \rceil = 385$$

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

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

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

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

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

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.

Chapter 6

METHODOLOGY

6.1 Quality in Software Systems

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 (ref. Table 6.1). 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.

6.2 Quality in Plagiarism Tools

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

Attributes	McCall	Boehm	Dromey	FURPS	ISO 9126
Maintainability	✓		✓		✓
Efficiency	✓	✓	✓		✓
Reliability	✓	✓	✓	✓	✓
Usability	✓		✓	✓	✓
Portability	✓	✓	✓		✓
Functionality			✓	✓	✓

Table 6.1: Agreed Upon Characteristics of Quality Software Systems Adapted from "Quality Models in Software Engineering" by Rafa E. Al-Qutaish, PhD, 2010, Journal of American Science, 6.3, p. 175

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. (ISO, 2016)" 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 ISO9126 is defined as "the capability of the software product to provide appropriate performance, relative to the amount of resources used, under stated conditions. (ISO, 2016)" This includes reasonable consumption of both time and system resources (ISO, 2016). 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 "[t]he capability of the software product to maintain a specified level of performance when used under specified conditions (ISO, 2016)" 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 the software product to be understood, learned, used, and attractive to the user, when used under specified conditions. (ISO, 2016)" 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. (ISO, 2016)" 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. (ISO, 2016)" 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 it's 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.

6.3 Defining 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 (Hage *et al.*, 2010) (Đurić and Gasevic, 2013). 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, to times exceeding 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 plagiarised program and introduce it into an otherwise clean set as done by many other authors (Roy and Cordy, 2008) (Liu *et al.*, 2006) (Chen *et al.*, 2004). 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 (Mozgovoy *et al.*, 2005). 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 (Mozgovoy *et al.*, 2005). 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 (Mozgovoy *et al.*, 2005). 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 6 test sets of various language and size. three of these sets (one Python, one Java, and one C) are from actual classroom submissions at the sophomore, junior, and graduate level. The other 3 sets will be gatherd online from publicly posted code competitions which host the results. 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 plagiarised 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 (CodeChef, 2020), 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 "plagiarised" 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 plagiarised. 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 a 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 (StatsCounter, 2020) (Steam, 2020) and 74.5% of all developers (StackOverflow, 2019)). 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% (StackOverflow, 2019) (StatsCounter, 2020) (Steam, 2020) 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 liekly 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 (StackOverflow, 2019). 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 (ref. Table 2.1) 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 (Hage *et al.*, 2010). The survey included "Supported Languages", "Extendability", "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 (Hage *et al.*, 2010). 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" (Hage *et al.*, 2010, p. 4). 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" among their analysis which were all included in the developed rubric (Durić and Gasevic, 2013).

		Inadequate (0)	Acceptable (1)	Excellent (2)	Exemplary (3)
Maintainability	Base code size (LOC)	Unmanagably large base code (100K+)	Manageable code base (10K-100K)	Small codebase (1K-10K)	Tiny codebase (0-1k)
	Additional language code size (LOC)	Additional languages require extensive code updates (>1K per language)	Additional languages require some code updates (100-1K per language)	Additional languages require minor code updates or configuration files (0-100 per language)	Additional languages require no changes or changes which could easily be entered by non-technical users
	Adaptable development language / environment	The language used is not supported anymore or is under heavy development	The language used has infrequent or major updates for new environments	The language used has periodic small updates for new environments	The language used has regular but transparent updates
Efficiency	Time to run for small sample (50 files of 100 LOC)	Execution could not be performed or takes longer than a day	Execution takes between 1 day and 1 hour	Execution takes between 1 hour and 1 min	Execution is less than 1 min
	Medium sample (100 files 200 LOC)	Same as above	Same as above	Same as above	Same as above
	Large sample (200 files 500 LOC)	Same as above	Same as above	Same as above	Same as above

Reliability	Likely matches (False Negatives)	The tool missed likely plagiarism cases where 4 tools identified plagiarism by more than 1 stdev above average in 3 of the test sets	The tool missed likely plagiarism cases where 4 tools identified plagiarism by more than 1 stdev above average in 2 of the test sets	The tool missed likely plagiarism cases where 4 tools identified plagiarism by more than 1 stdev above average in 1 of the test sets	The tool missed likely plagiarism cases where 4 tools identified plagiarism by more than 1 stdev above average in 0 of the test sets
	Unlikely matches (False Positives)	The tool found unlikely plagiarism cases where only 1 tool identified plagiarism more often than 1 stdev above average in 3 of the test sets	The tool found unlikely plagiarism cases where only 1 tool identified plagiarism more often than 1 stdev above average in 2 of the test sets	The tool found unlikely plagiarism cases where only 1 tool identified plagiarism more often than 1 stdev above average in 1 of the test sets	The tool found unlikely plagiarism cases where only 1 tool identified plagiarism more often than 1 stdev above average in 0 of the test sets
Usability	Clean, easy to use UI	The tool has a command line execution and does not support comparing matches	The tool has a command line execution but generates a readable report to compare matches	The tool supports UI based input and viewing of matches	The tool supports UI based input and viewing of matches and is interactive beyond plain text displays of info
	Simple to obtain and use	The tool requires difficult or many steps to obtain or requires extensive developer knowledge to begin using	The tool requires some developer knowledge that most users are likely to have to begin using	The tool requires no developer knowledge to begin using	The tool has guided installation that most non-programming users could follow

Portability	Support for OS	The system cannot be run directly on any major operating system (Windows/-MacOS) or requires portability tools to operate on common environments.	There is support for at least 1 major operating system (Windows/-MacOS).	There is support for both Windows and MacOS.	There is support for both Windows, MacOS, and Linux.
	Web Deployable	It would be difficult to deploy such a system to the web without recreating large sections.	The system could be deployed to the web with some effort.	The system would need no major modifications to deploy to the web.	The system is currently hosted or has previously been hosted on the web.
Functionality	Support for common languages	The system supports less than 5 of the top 20 languages.	The system supports 5 of the top 20 languages.	The system supports 10 of the top 20 languages.	The system supports almost all of the top 20 languages (15 or more).
	Support for other languages	The system supports at least 5 less common languages.	The system supports 10 other languages	The system supports 15 other languages	The system supports 20 or more other languages.
	Support for tuning features	The system does not support template code, common code removal, or threshold modification.	The system supports 1 of the following: template code, common code removal, or threshold modification.	The system supports 2 of the following: template code, common code removal, or threshold modification.	The system supports all of the following: template code, common code removal, and threshold modification.

Table 6.2: A Quality Rubric for Plagiarism Detection Tools

Chapter 7

RESULTS

7.1 Overall 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. In the further sections, the result of each test/investigation will be detailed and any notes beyond what the rubric allowed will be noted.

7.2 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. After deleting all language specific code (they are separated neatly into packages) and running a lines of code count on only the remaining java code (i.e. ignoring html, pom.xml files from maven, etc.) JPlag's current code base came up to 28,379 lines (Malpohl, 2019). This places it in the "Acceptable" range from 10K to 100K lines. Counting the language specific module code for each of the supported languages, the average came up to 1,340 lines per language (Malpohl, 2019). Attempts were made to remove any code that is believed to have been generated by either Antlr or JavaCC. The grammars themselves were included based on notes from the authors that the grammars are not the same as language grammars intended for compiler generation and should instead be written with plagiarism detection in mind to yield much merit

Prechelt *et al.* (2002) Malpohl (2019). Keeping this in mind, 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. It should also be noted that maintainability is also a key focus of MOSS as can be seen in notes from the author inside the tool (as in the following quote on the results sharing system). This quote also highlights points about the author's statement on the portability of MOSS's system which will be developed upon later.

> Q. Couldn't you mail the HTML results back to me instead of putting them on the Web?
> A. While this would be an inherently more secure design, it is impractical for maintenance reasons. At least 90% of the time in maintaining Moss is spent dealing with problems users have with the submission script in various mutually incompatible environments. Widening the interface to include results returned from the server can only exacerbate this problem. Because Moss is not a commercial service, the time spent maintaining it must be kept low. (Aiken, 2018)

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.

SIM's codebase falls in the middle of the "Excellent" range with 4252 lines of C code. It has its own parser syntax based on Flex and for the existing languages averages 254 lines of code per language. Out of both curiosity and in order to facilitate testing on available data sets, a specification for python 3 using SIM was developed based on the existing languages and documentation from the author. This will be discussed further in the reliability section, however from this experience it should be noted that 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.

7.3 Efficiency

Kitsune's definitely had a slower execution time on average than the other tools analyzed, likely due to the fact that the other tools primarily deal with in-memory analysis of the programs whereas Kitsune has to read a representation of each program into the database. While this is certainly unfortunate, even in the worst test case with nearly 200 students and 500 lines of code, Kitsune performed within a reasonable amount of time for teachers to be able to use it.

JPlag and Moss had a large increase in time from the first set to the third as well and thus their execution times passed the threshold from "Exemplary" to "Excellent" as well. Though Kitsune did not behave as well as these two tool in terms of execution speeds for inserting new sets, this demonstrates for for very large scale submissions, JPlag and Moss will start degrading as well. Future work will be in improving the efficiency of Kitsune, including looking into other similarity algorithms.

SIM behaved oddly for the second, middle sized test set. It's time actually decreased from the small set. There are two possible explanations for this. The first is that the execution time for SIM was small enough in general that such fluctuation is rather insignificant. The second is that the smaller set was in C while the middle set was in Python. It is possible that the transition effected MOSS's execution time by enough if it's Python Parser was more efficient.

7.4 Reliability

Kitsune has demonstrated a great deal of reliability in finding the issues that other tools identified. The only tool that consistently varied from the results of other tools by a significant amount was Sherlock. In fact, the only times where Sherlock's deviations were within a single standard deviation of the average were when the tools

Iteration	Tool	Small Set Time (s)	Medium Set Time (s)	Large Set Time (s)
Execution 1	JPlag	0.166	20.691	29.483
	MOSS	10.11	17.704	69.509
	Sherlock	0.241	1.427	2.141
	SIM	0.679	0.44	0.628
	Kitsune	29.81	1067.60	1136.25
Execution 2	JPlag	0.159	19.688	43.019
	MOSS	10.644	17.341	19.436
	Sherlock	0.252	1.413	1.467
	SIM	0.72	0.43	0.654
	Kitsune	26.514	946.36	1262.11
Execution 3	JPlag	0.17	14.07	33.554
	MOSS	12.342	16.58	20.408
	Sherlock	0.253	1.333	1.50
	SIM	0.696	0.455	0.635
	Kitsune	27.54	974.78	1274.61
Avg. Execution Time	JPlag	0.165	18.15	35.352
	MOSS	11.032	17.208	36.451
	Sherlock	0.249	1.391	1.703
	SIM	0.698	0.442	0.639
	Kitsune	27.955	996.247	1224.323

Table 7.1: Runtimes Across Different Sized Test Sets

were vastly disagreeing across the board (which usually occurred when there was very little cheating or on extremely small programs. SIM did not behave favorably on the smaller C set, and ended up failing both at identifying likely matches and in avoiding identifying unlikely matches.

In Figure 7.1, the distributions for each of the tools were plotted in a stacked bar chart (i.e. times when the only that tool found plagiarism are in blue at the bottom, then 2 tools, 3 tools, 4 tools, all 5 tools) for 4 of the sets. The other set was much larger and did not process well in excel. These charts allow easy visualization of strong deviation. Preferably, the bottom section should be as small as possible. Some disagreement is expected, however when a tool falls outside that amount it can be seen as an unexpected failure of that tool. For example, in the Python A1 set, Sherlock clearly stands out with 78.21% of its guessed plagiarisms not being guessed by any of the other 4 tools.

Since only Sherlock and SIM had a single case where they fell far outside the bounds of acceptability, all 3 of the other tools fall into the "Exemplary" category. Manually looking at many of the cases where Sherlock was the only tool to find plagiarism shows a general trend - often the files bear some degree of general similarity in the tokens or names used, but this is often arbitrary and spread throughout the file. It is difficult to say exactly where Sherlock fails because of the lack of representation with which to explain its results, but based on inferences from the general algorithm, Sherlock appears to have trouble distinguishing between 2 files that share many tokens and 2 files that share many sequential chunks of tokens.

It was also interesting to cross compare each pair of tools to determine which tools each agreed with the largest majority of time. Across all 6 of the tested sets, Kitsune most frequently agreed with JPlag, followed by MOSS, then SIM, then Sherlock. In fact, in every single one of the test sets, JPlag was the most similar and in 4 of the 5

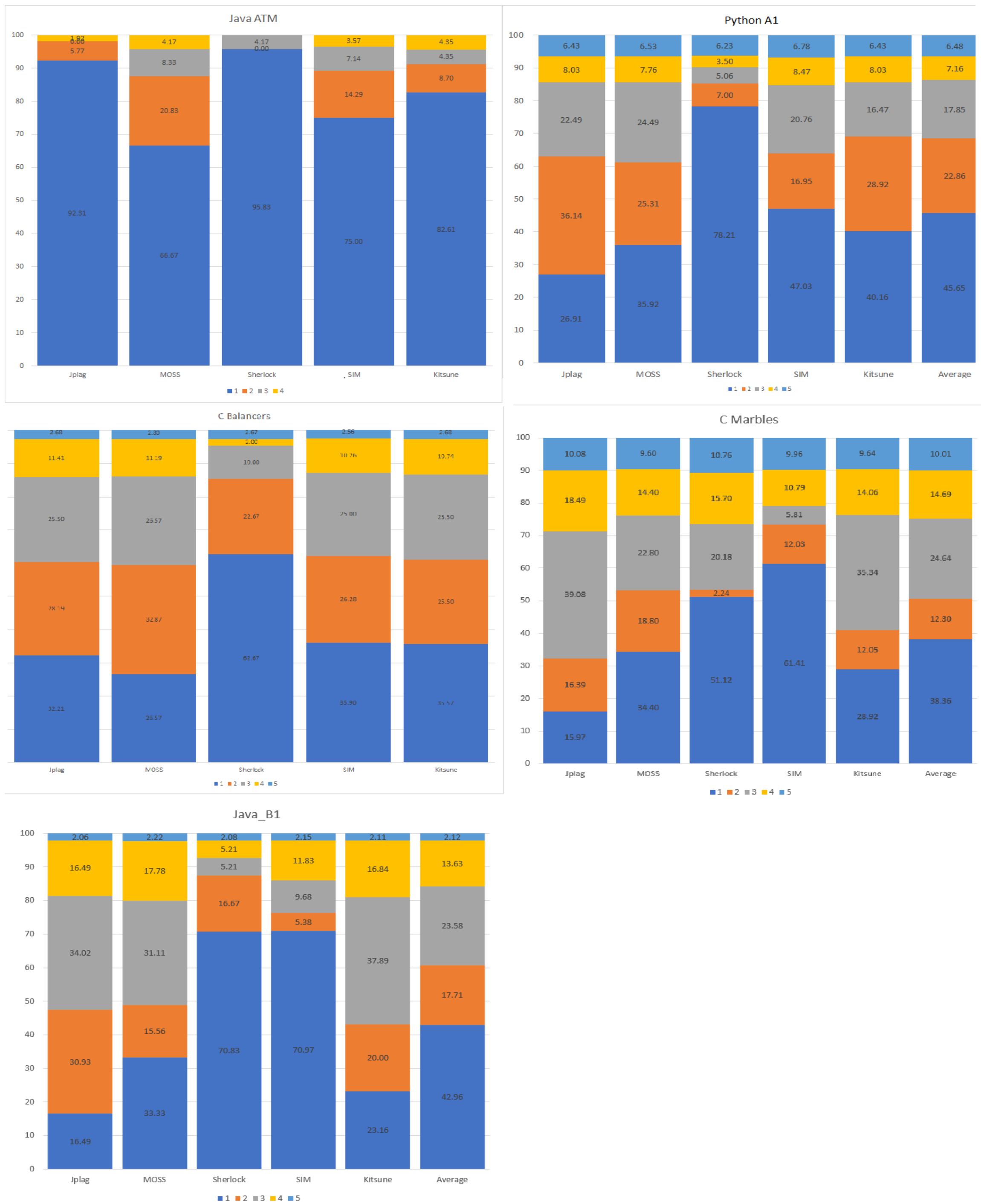

Figure 7.1: Jury Presence Distributions for Chosen Problem Sets

sets, MOSS was the second most similar (and was 3rd in the other 2 sets). This does well to demonstrate the success of Kitsune at producing results which are comparable to more widely accepted and state of the art of current plagiarism detection systems.

7.5 Usability

JPlag ships as an executable jar file which can be run from the command line (see 7.2. When run, it generates a simplistic webpage based UI which can be used to navigate the results (see 7.3. 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 (see 7.4 the interface for which was actually written by the author of JPlag) and responds with the link. JPlag used to have a similar web based submission, but released the source code and became an open source tool to decrease workload in maintaining it. MOSS has the same score in this category because of the requirement of executing the Perl script from command-line. There are a number of GUIs floating around for MOSS that have been developed, but none of them are

```
JPlag (Version 2.12.1-SNAPSHOT), Copyright (c) 2004-2017 KIT - IPD Tichy, Guido Malpohl, and others.
Usage: JPlag [ options ] <root-dir> [-c file1 file2 ...]
 <root-dir>        The root-directory that contains all submissions.

options are:
 -v[qlpd]        (Verbose)
                 q: (Quiet) no output
                 l: (Long) detailed output
                 p: print all (p)arser messages
                 d: print (d)etails about each submission
 -d              (Debug) parser. Non-parsable files will be stored.
 -S <dir>        Look in directories <root-dir>/*/<dir> for programs.
                 (default: <root-dir>/*)
 -s              (Subdirs) Look at files in subdirs too (default: deactivated)

 -p <suffixes>   <suffixes> is a comma-separated list of all filename suffixes
                 that are included. ("-p ?" for defaults)

 -o <file>       (Output) The Parserlog will be saved to <file>
 -x <file>       (eXclude) All files named in <file> will be ignored
 -t <n>          (Token) Tune the sensitivity of the comparison. A smaller
                 <n> increases the sensitivity.
 -m <n>          (Matches) Number of matches that will be saved (default:20)
 -m <p>%         All matches with more than <p>% similarity will be saved.
 -r <dir>        (Result) Name of directory in which the web pages will be
                 stored (default: result)
 -bc <dir>       Name of the directory which contains the basecode (common framework)
 -c [files]      Compare a list of files. Should be the last one.
 -l <language>   (Language) Supported Languages:
                 java19 (default), java17, java15, java15dm, java12, java11, python3, c/c++, c#-1.2, char, text, scheme
```

Figure 7.2: JPlag's CLI Interface

Matches for A3_55.py & A3_16.py

61.7%

INDEX - HELP

A3_55.py (58.598724%)	A3_16.py (65.24823%)	Tokens
A3_55.py(18-28)	A3_16.py(35-48)	21
A3_55.py(31-46)	A3_16.py(12-32)	27
A3_55.py(47-52)	A3_16.py(54-64)	18
A3_55.py(92-120)	A3_16.py(107-139)	26

Figure 7.3: JPlag's Comparison Interface

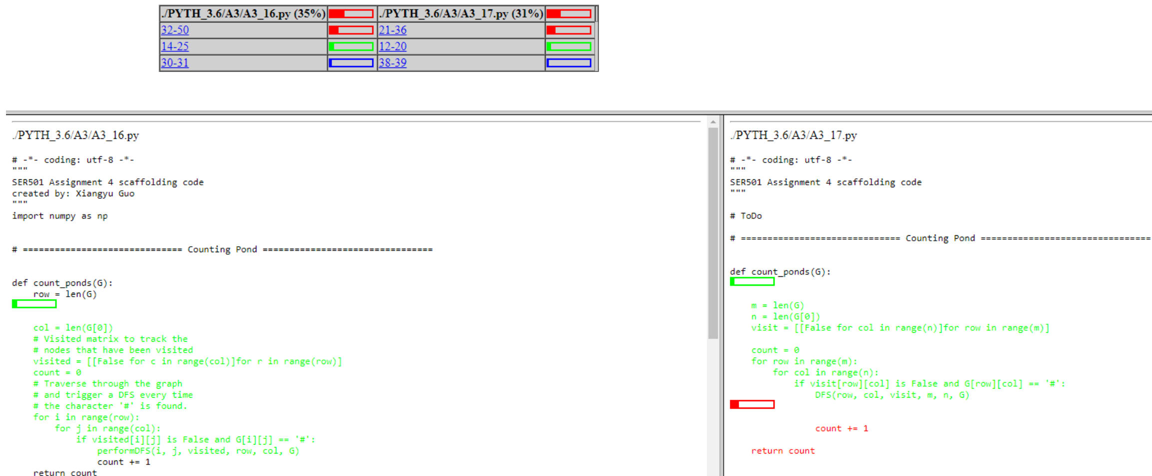

Figure 7.4: MOSS's Comparison Interface

official and were not included because of this. The three mentioned under related community works on MOSS's website all look fairly inactive as well, the most active of the three having a total of 35 commits from between 2011 and 2017 and no recent activity. Unfortunately, obtaining MOSS is actually quite a process. In order to "sign up" for the MOSS service, the user needs to email an automated service a register command which contains the email you are signing up with. This email is extremely sensitive – even misplaced spaces (or a hyperlink automatically added by Outlook) can cause the email to respond "Could not find email address in registration request." Upon successful registration, MOSS replies with a unique Perl script which can be executed to submit the files after this. Given the atypical and finicky nature of MOSS's sign up, MOSS fell under the "Inadequate" bin because to "[t]he tool requires difficult or many steps to obtain."

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 (see 7.6). 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 C Makefile and generate the binary, a feet likely out of reach for non-technically oriented users. Keeping this in mind, Sherlock received an rating of “Inadequate” for both measures under usability.

```
$ ./sherlock -e py ./PYTH_3.6/A3/
./PYTH_3.6/A3//A3_1.py;./PYTH_3.6/A3//A3_10.py;10%
./PYTH_3.6/A3//A3_1.py;./PYTH_3.6/A3//A3_100.py;8%
./PYTH_3.6/A3//A3_1.py;./PYTH_3.6/A3//A3_101.py;16%
./PYTH_3.6/A3//A3_1.py;./PYTH_3.6/A3//A3_102.py;23%
./PYTH_3.6/A3//A3_1.py;./PYTH_3.6/A3//A3_103.py;12%
./PYTH_3.6/A3//A3_1.py;./PYTH_3.6/A3//A3_104.py;26%
./PYTH_3.6/A3//A3_1.py;./PYTH_3.6/A3//A3_105.py;18%
./PYTH_3.6/A3//A3_1.py;./PYTH_3.6/A3//A3_106.py;19%
./PYTH_3.6/A3//A3_1.py;./PYTH_3.6/A3//A3_107.py;21%
./PYTH_3.6/A3//A3_1.py;./PYTH_3.6/A3//A3_108.py;15%
./PYTH_3.6/A3//A3_1.py;./PYTH_3.6/A3//A3_109.py;11%
./PYTH_3.6/A3//A3_1.py;./PYTH_3.6/A3//A3_11.py;23%
./PYTH_3.6/A3//A3_1.py;./PYTH_3.6/A3//A3_110.py;22%
./PYTH_3.6/A3//A3_1.py;./PYTH_3.6/A3//A3_111.py;12%
./PYTH_3.6/A3//A3_1.py;./PYTH_3.6/A3//A3_112.py;25%
```

Figure 7.5: Sherlock’s Output

SIM, like Sherlock does not provide a graphical mechanism for comparing files. Instead, SIM outputs every comparison it finds in a diff like format. This is slightly better than Sherlock as it is at least explainable and provides users with a way of identifying and manually looking over identified sections, but can be quite unruly for large sets. The console – or even a piped-to file becomes difficult to manage for large sets with many similar sections. The largest of the sets in the efficiency tests produced a 75,000 line text file of output full of diff-like comparisons. Combing through this would be as impractical as simply finding the similarities while grading. Because of

this SIM also scored under "Inadequate" for user interface. Unlike Sherlock, however, SIM provided an executable format to run it which simplified the process of obtaining and using it. Besides command-line execution, SIM requires very little developer knowledge to begin using and thus, along with JPlag, is the second tool to fit the "Excellent" category for obtaining and beginning usage.

```
.\PYTH_3.6\A3\A3_30.py: line 142-184   |.\PYTH_3.6\A3\A3_54.py: line 90-132[225]
  if count_ponds(["#--------##-",      |  if count_ponds(["#--------##-",
  "-###-----###",                      |  "-###-----###",
  "----##---##-",                      |  "----##---##-",
  "---------##-",                      |  "---------##-",
  "---------#--",                      |  "---------#--",
  "--#------#--",                      |  "--#------#--",
  "-#-#------#-",                      |  "-#-#-----##-",
  "#-#-#-----#-",                      |  "#-#-#-----#-",
  "-#-#------#-",                      |  "-#-#------#-",
  "--#-------#-"]) == 3:               |  "--#-------#-"]) == 3:
  print('passed')                      |  print('passed')
  else:                                |  else:
  print('failed')                      |  print('failed')
                                       |
  if count_ponds(["#--------##-",      |  if count_ponds(["#--------##-",
  "-###-----###",                      |  "-###-----###",
  "----##---##-",                      |  "----##---##-",
  "---------##-",                      |  "#--------##-",
  "---------#--",                      |  "---------#--",
  "--#---------",                      |  "--#------#--",
  "-#-#-----##-",                      |  "-#-#-----##-",
  "#-#-#-----#-",                      |  "#-#-#-----#-",
  "-#-#------#-",                      |  "-#-#------#-",
  "--#-------#-"]) == 4:               |  "--#-------#-"]) == 4:
  print('passed')                      |  print('passed')
  else:                                |  else:
  print('failed')                      |  print('failed')
                                       |
  if longest_ordered_subsequence([1, 7,|  if longest_ordered_subsequence([1, 7,
  print('passed')                      |  print('passed')
  else:                                |  else:
  print('failed')                      |  print('failed')
                                       |
  if supermarket([(50, 2), (10, 1), (20|  if supermarket([(50, 2), (10, 1), (20
  print('passed')                      |  print('passed')
  else:                                |  else:
  print('failed')                      |  print('failed')
                                       |
  if supermarket([(20, 1), (2, 1), (10,|  if supermarket([(20, 1), (2, 1), (10,
  (50, 10)]) == 185: # noqa            |  (8, 2), (5, 20), (50, 10)]) == 185:
  print('passed')                      |  print('passed')
  else:                                |  else:
```

Figure 7.6: SIM's Output

Unlike the other 4 tools, Kitsune provides a graphical interface for both inputting files, tuning parameters, and comparing the results (see 7.7). 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 simi-

larity comparisons are removed, or such that only files the user chooses are included (as in 7.8). The tool also has a guided installer (see 7.9) which can set up the application on the system as a full application that can be used normally. Because of these features, Kitsune scores “Examplary” under both its UI and the setup process.

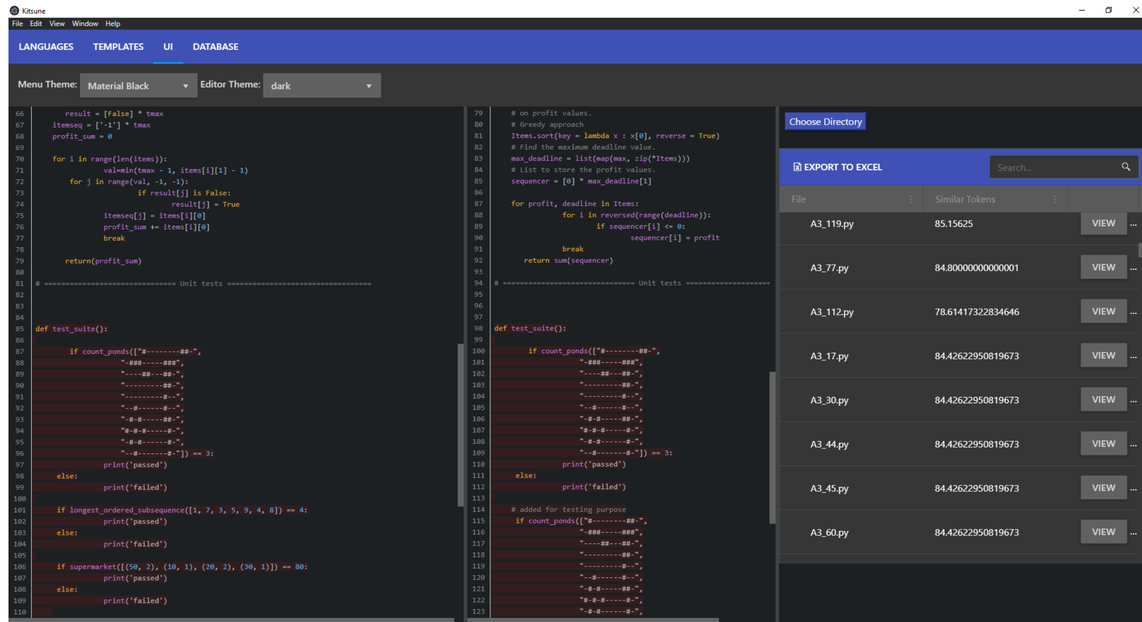

Figure 7.7: Kitsune’s Interface

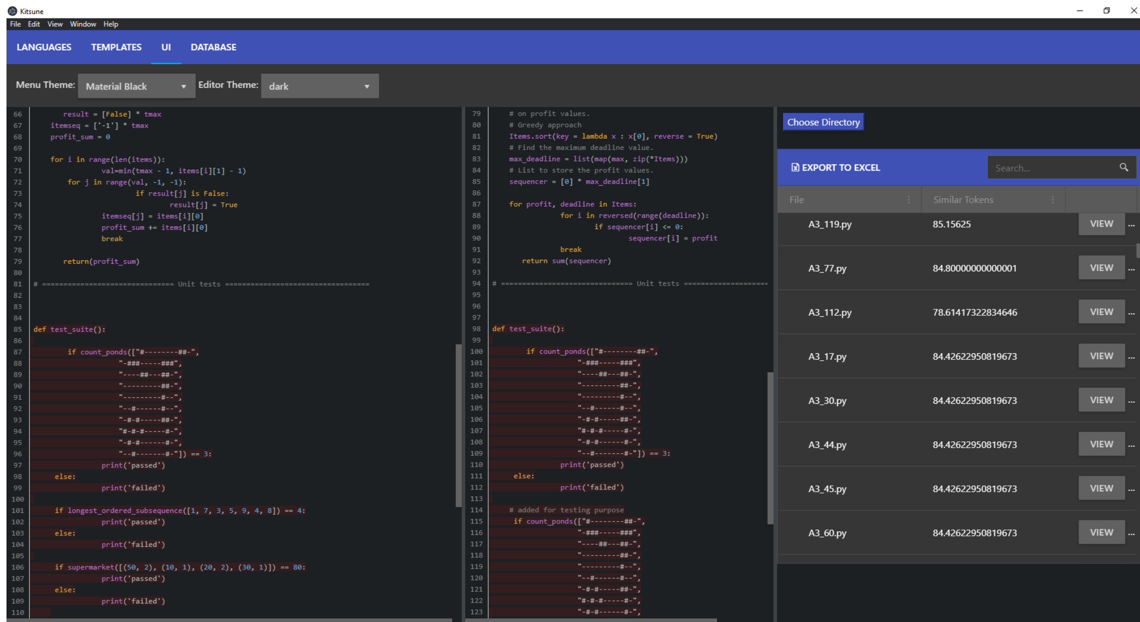

Figure 7.8: Kitsune's Filtering Interface

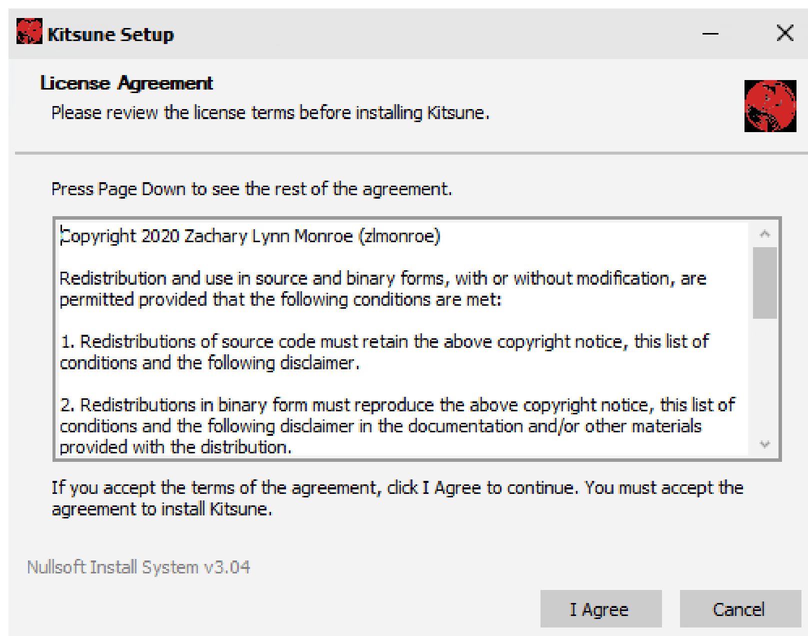

Figure 7.9: Kitsune's Installer

7.6 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. MOSS's website states that "[t]he 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. Beause of this MOSS does not meet the minimum criteria and received an "Inadequate" rating 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. 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".

7.7 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. Ignoring these 5 languages, JPlag supports one other language, Scheme. This gives it a score of 'Inadequate" in terms of support for languages outside of the top 20. JPlag currently supports threshold adjustments, and is able to take in a template to exclude from results. It does not currently support exclusion of commonly repeating code, meaning it scores "Excellent" in the support for commonly requested features.

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. Eleven of the languages MOSS has listed on their website are included in the top 20 languages from the 2019 survey: C, C++, Java, C#, Python, Visual Basic, Javascript, Lisp, MIPS assembly, a8086 assembly, and a8086 assembly. The three assembly languages supported by MOSS are all grouped under "assembly" on the survey, but leaving them there, MOSS scores "Adequate" in the common language support area. Especially given the many years MOSS has been running and the slow rate of additional language support to keep up with modern language trends (the oldest WayBack Machine Internet Archive of the website from October 11, 2006 still has MOSS with the exact same languages supported, meaning there has been no added languages in at least 14 years), this is a rather remarkable feat. On the side of support for less common languages, MOSS supports FORTRAN, ML, Haskell, Scheme, Pascal, Modula2, Ada, Perl, TCL, Matlab, VHDL, Verilog, Spice, and HCL2, putting it just shy of the 15 language mark and thus giving it a score of "Adequate". 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. This means that MOSS supported two of the tuning features giving it a score of "Excellent" for commonly requested features.

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 (StackOverflow, 2019). It also supports three other languages: Pascal, Modula-2, and Miranda. This gives SIM a score of "Adequate" for the common languages and 'Inadequate" for support for less common languages. SIM does not support common code exclusion or template code. It does however support threshold adjustments for it's output, again giving it a score of "Adequate" for commonly requested features.

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. Through a quick search on GitHub, grammars for the top 20 languages which both compile and are open source can be found through minimal effort. Though the accuracy of most of these languages has not been tested, Kitsune

was able to detect visually accurate matches in 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.

		JPlag	Moss	Sherlock	SIM	Kitsune
Maintainability	Base code size (LOC)	1	2	3	2	3
	Additional language code size (LOC)	0	2	3	1	2
	Adaptable development language / environment	3	3	3	3	3
	Total	4	7	9	6	8
Efficiency	Time to run for small sample (50 files of 100 LOC)	3	3	3	3	3
	Time to run for medium sample (130 files 150 LOC)	3	3	3	3	2
	Time to run for large sample (130 files 300 LOC)	2	2	3	3	2
	Total	8	8	9	9	7
Reliability	Likely matches	3	3	0	2	3
	Unlikely matches	3	3	0	1	3
	Total	6	6	0	3	6

Usability	Clean, easy to use UI	1	1	0	0	3
	Simple to obtain and use	2	0	0	2	3
	Total	3	1	0	2	6
Portability	Support for OS	3	0	3	1	3
	Web Deployable	3	3	1	1	3
	Total	6	3	4	2	6
Functionality	Support for common languages	1	2	3	1	3
	Support for other languages	0	1	0	0	3
	Support for tuning features	2	2	1	1	1
	Total	3	5	4	2	7
Total		30	30	26	24	40

Table 7.2: Rubric Scores for the Five Evaluated Tools

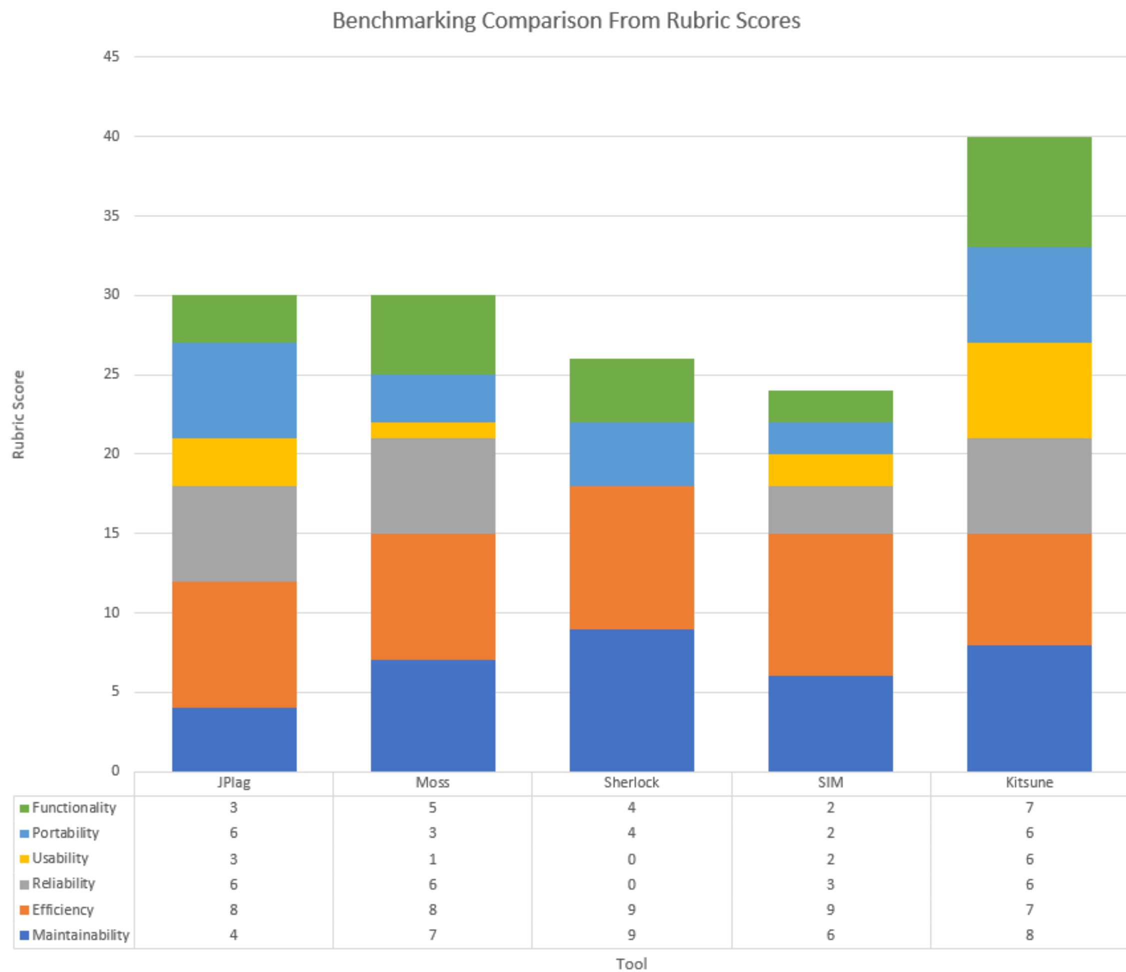

Figure 7.10: Rubric Scores for the Five Evaluated Tools (Chart)

Chapter 8

CONCLUSION

8.1 Achievements

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. Currently it takes around 20 lines of CSS code to implement a syntax highlighter for a given language (see figure 8.1).

```css
/* Matches all tokens which were not read by the Antlr4 parser.
   This includes comments, non-compilable code, etc. */
.no-match {
    color: rgb(128, 128, 128) !important;
}

/* Matches all identifiers */
.NAME-local {
    color: #BF74D5 !important;
}

/* Matches all of the python number types */
.DECIMAL_INTEGER-local, .OCT_INTEGER-local, .HEX_INTEGER-local,
.BIN_INTEGER-local, .FLOAT_NUMBER-local, .IMAG_NUMBER-local {
    color: #5FAAE8 !important;
}

/* Matches strings */
.str-local {
    color: #88AF6E !important;
}

/* Keyword list - Taken from Antlr4 Tokens file */
.FALSE-local, .NONE-local, .TRUE-local, .AND-local, .AS-local,
.ASSERT-local, .ASYNC-local, .AWAIT-local, .BREAK-local,
.CLASS-local, .CONTINUE-local, .DEF-local, .DEL-local,
.ELIF-local, .ELSE-local, .EXCEPT-local, .FINALLY-local,
.FOR-local, .FROM-local, .GLOBAL-local, .IF-local, .IMPORT-local,
.IN-local, .IS-local, .LAMBDA-local, .NONLOCAL-local, .NOT-local,
.OR-local, .PASS-local, .RAISE-local, .RETURN-local, .TRY-local,
.WHILE-local, .WITH-local, .YIELD-local {
    color: #E1A437 !important;
}
```

Figure 8.1: CSS for Python3 Syntax Highlighting

8.2 Issues and Future Work

One of the areas where Kitsune did not score as highly as other surveyed tools was efficiency. There are two main reasons that Kitsune suffers in this area. The first is that Kitsune has to insert the files into a database in the first place. This is generally slower and would be difficult to remedy. However, this process is generally a linear increase on the size of the set and does not present a pressing issue. More of concern is the time to compare files after inserting. Currently, Kitsune uses a technique similar to MOSS and to Sherlock based on MinHash. This process requires Kitsune to identify the minimum hashes for nodes under major subtrees. Because this has to be done for every subtree each time Kitsune runs a comparison (it would also be prohibitively expensive to store a list of minimum nodes for every block of code for every program), this process does not scale well. Initial research into the idea of using an alternative comparison algorithm, SimHash, has been performed. SimHash was the main competitor to MinHash in Google's research into removing duplicate webpage results from searches Manku *et al.* (2007). In the end SimHash won out due to memory contraints and efficiency improvements (Manku *et al.*, 2007). Unlike MinHash, SimHash creates a minimal representation for documents (in this case blocks of code). Because of this, SimHash could be pre-computed a single time upon insertion leaving the comparison process a matter of calculating the differences between the two generate hashes. This not only speeds up the initial runtime, but saves a major section of the work done for the comparison so that should a comparison need to be rerun (i.e. with different parameters) the comparison can skip large amounts of work. In many of the larger sets, Kitsune's comparison time has been similar to many of the other tools after the database insertion.

Another possible goal would be to aim Kitsune at being integrated with submission systems so that preprocessing could be done upon submission (similarly to the way many universities use systems like Turnitin (Turnitin, 2020) to check for plagiarism in papers). This would allow Kitsune to do pre-processing up front, leaving only the comparison when the professor needs to compile results. In order to scale to the size of such a system as Turnitin, Kitsune could index the hashes generated by SimHash using a hierarchical clustering scheme. Preprocessing the code block hashes in this way, it is possible that Kitsune could be further optimized such that similar blocks of code could be found in extremely large sets, making it possible for Kitsune to be scaled to a more commercial sized plagiarism detection system, or at the very least, such that a teacher could compare years worth of their own submissions to verify students did not find past submissions online. Hierarchical clustering is popular for use in gene sequencing tasks (ref. 8.2). Essentially, it groups sequences of numbers across two variables into "clusters", or groups, which are similar. In gene sequencing, this is commonly used to show correlations. For example in the example in 8.2, the heat map displays a cross section of patients (columns) and genes (rows) and shows a correlation between the genes of patients with either "Acute Rejection", "Normal" reaction, or "Chronic Allograft Nephropathy" to kidney transplants. In Kitsune, this technique would be useful for grouping similar blocks of code together such that only closely packed hashes would need to be compared. The "Hamming" distance (defined as the number of bits in the hashes that differ) could be used as a distance function to create the dendrograph (the ordering of clusters). In this way, only the closest hashes would need to be compared to a given hash, meaning that after indexing the clustering, the time to process new values would be $O(1)$.

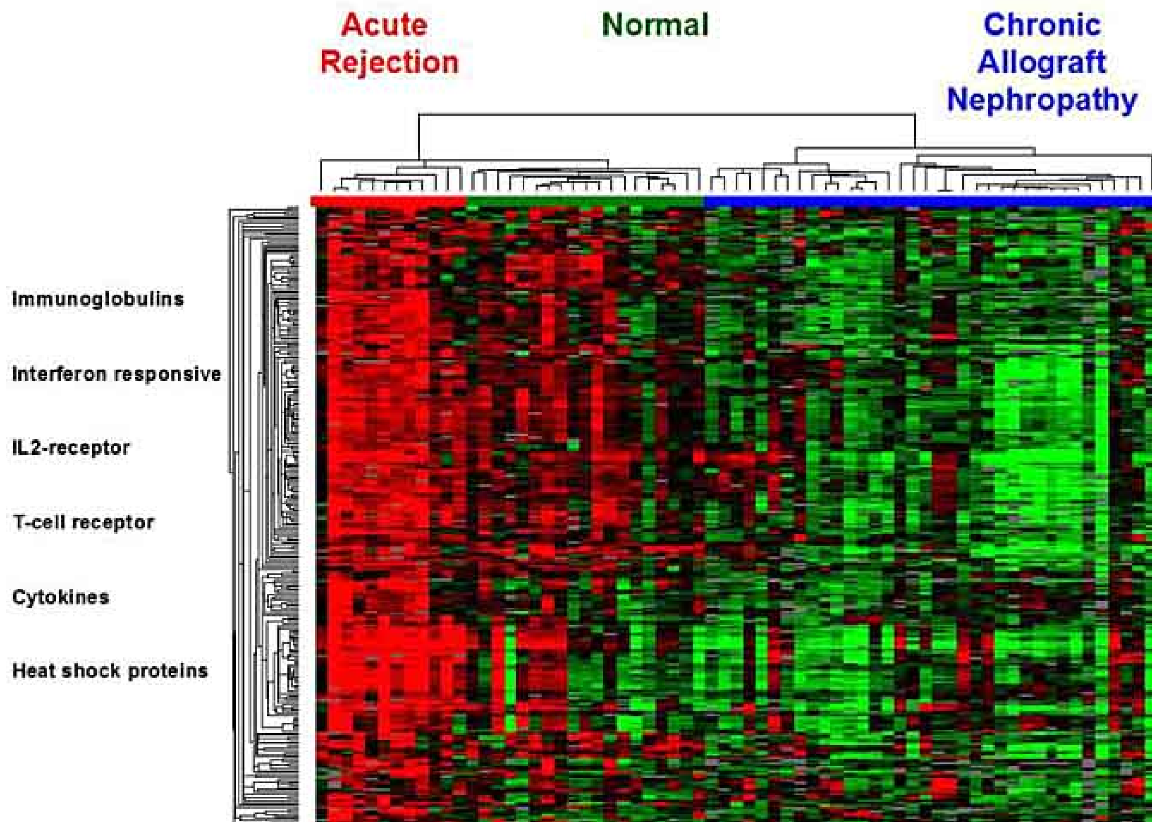

Figure 8.2: Example of Hierarchical Clustering For Gene Sequencing (Chua *et al.*, 2003)

While Kitsune's language configuration is already extremely small and simple, it would be preferred to allow regular users who may not be familiar with Antlr to add new grammars as they are developed. This would decrease the work required for developers to maintain Kitsune, and would also allow users to compare languages that Kitsune does not support. One method this could be accomplished is by utilizing Antlr4's grammar in order to parse itself. This would allow analysis of the grammar to be performed using Antlr itself and then provide the user a graphical UI to configure a new language. It would also be possible to provide suggestions for the language about what grammar rules might match blocks in the language for languages that follow

common conventions for naming units. I.E. a languages that utilized ”procedures” and has a grammar rule that contains this word would be easy to suggest to the user. Many of the parameters such as similarity threshold did not change often between programs and could easily have default values. Through this process, a user could easily choose a grammar file for Antlr4 from online and add new support themselves.

REFERENCES

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

Ahtiainen, A., S. Surakka and M. Rahikainen, "Plaggie: Gnu-licensed source code plagiarism detection engine for java exercises", in "Proceedings of the 6th Baltic Sea conference on Computing education research: Koli Calling 2006", pp. 141–142 (2006).

Aiken, A., "The moss faq", URL `http://moss.stanford.edu/general/faq.html` (2018).

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).

Chua, M.-S., E. Mansfield and M. Sarwal, "Applications of microarrays to renal transplantation: progress and possibilities", Front Biosci **8**, s913–s923 (2003).

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

CodeChef, "Codechef: Programming competition,programming contest,online computer programming", URL `https://www.codechef.com/` (2020).

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

Ducasse, S., M. Rieger and S. Demeyer, "A language independent approach for detecting duplicated code", in "Proceedings IEEE International Conference on Software Maintenance-1999 (ICSM'99).'Software Maintenance for Business Change'(Cat. No. 99CB36360)", pp. 109–118 (IEEE, 1999).

Durić, Z. and D. Gasevic, "A source code similarity system for plagiarism detection", The Computer Journal **56**, 1, 70–86 (2013).

Goel, S., D. Rao *et al.*, "Plagiarism and its detection in programming languages", TechnicalReport, Department of Computer Science and Information Technology, JIITU (2008).

Grier, S., "A tool that detects plagiarism in pascal programs", ACM SIGCSE Bulletin **13**, 1, 15–20 (1981).

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

Hage, J., "Programmeerplagiaatdetectie met marble", (2006).

Hage, J., P. Rademaker and N. van Vugt, "A comparison of plagiarism detection tools", Utrecht University. Utrecht, The Netherlands **28**, 1 (2010).

ISO, "Iso/iec tr 9126-4:2004", URL `https://www.iso.org/standard/39752.html` (2016).

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).

Joy, M. and M. Luck, "Plagiarism in programming assignments", IEEE Transactions on education **42**, 2, 129–133 (1999).

Kamiya, T., S. Kusumoto and K. Inoue, "Ccfinder: a multilinguistic token-based code clone detection system for large scale source code", IEEE Transactions on Software Engineering **28**, 7, 654–670 (2002).

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).

Liu, C., C. Chen, J. Han and P. S. Yu, "Gplag: detection of software plagiarism by program dependence graph analysis", in "Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining", pp. 872–881 (2006).

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 "Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering", pp. 389–400 (2014).

Malpohl, G., "jplag/jplag", URL `https://github.com/jplag/jplag` (2019).

Manku, G. S., A. Jain and A. Das Sarma, "Detecting near-duplicates for web crawling", in "Proceedings of the 16th international conference on World Wide Web", pp. 141–150 (2007).

Moussiades, L. and A. Vakali, "Pdetect: A clustering approach for detecting plagiarism in source code datasets", The computer journal **48**, 6, 651–661 (2005).

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).

Oracle, "Oracle america, inc. v. google llc", (2018).

Plank, J. S., "Cs494 lecture notes - minhash", URL `http://web.eecs.utk.edu/~jplank/plank/classes/cs494/494/notes/Min-Hash/index.html` (2019).

Prechelt, L., G. Malpohl, M. Philippsen *et al.*, "Finding plagiarisms among a set of programs with jplag", J. UCS **8**, 11, 1016 (2002).

Ragkhitwetsagul, C., *Code similarity and clone search in large-scale source code data*, Ph.D. thesis, UCL (University College London) (2018).

Ragkhitwetsagul, C., J. Krinke and D. Clark, "Similarity of source code in the presence of pervasive modifications", in "2016 IEEE 16th international working conference on source code analysis and manipulation (SCAM)", pp. 117–126 (IEEE, 2016).

Roy, C. K. and J. R. Cordy, "Nicad: Accurate detection of near-miss intentional clones using flexible pretty-printing and code normalization", in "2008 16th iEEE international conference on program comprehension", pp. 172–181 (IEEE, 2008).

Schleimer, S., 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", pp. 76–85 (2003).

Sonos, "Sonos, inc. v. google llc", (2020).

StackOverflow, "Stack overflow developer survey 2019", URL `https://insights.stackoverflow.com/survey/2019` (2019).

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

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

Turnitin, "About us", URL `https://www.turnitin.com/about` (2020).

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

Wise, M. J., "Yap3: Improved detection of similarities in computer program and other texts", in "Proceedings of the twenty-seventh SIGCSE technical symposium on Computer science education", pp. 130–134 (1996).

Zeidman, B., "Software v. software", IEEE Spectrum **47**, 10, 32–53 (2010).