Plagiarism Detection Using Language Recognition Tools

Colleen Vu and Abiel Kim

CMPT 473 D100, Simon Fraser University

1 Introduction

1.1 Background

Plagiarism detection in source code has been a significant focus of attention in computer science education. However, most existing tools are based on specific algorithms and parsing techniques, often lacking flexibility for educators to adapt the tools to different use cases or languages. By utilizing different language specification tools such as ANTLR, Tree-sitter, JavaCC, and J-Flex + CUP, we can explore how these tools impact the design and performance of plagiarism detection systems.

1.2 Problem Statement

Despite the availability of various plagiarism detection tools, there is limited research evaluating the effectiveness of different language specification tools in building plagiarism detection systems. This project seeks to create three custom plagiarism detection systems, each based on a different language specification tool, and compare their effectiveness in detecting various types of plagiarism.

1.3 Objectives

1. Develop a (or multiple) custom plagiarism detection systems, each built using a different language specification tool.
2. Implement variants using ANTLR and Tree-sitter.
3. Evaluate the effectiveness of each system in detecting code plagiarism, focusing on different types of code manipulation (e.g., direct copying, obfuscation, formattin.
4. Analyze the strengths and limitations of each language specification tool in terms of precision, recall, false positive rate, and false negative rate.

1.4 Issues and Significance

Plagiarism in computer science-related education has posed a major challenge for educators in recent years due to the increased access to information and software tools online that have made it easier to copy code. As a result, this practice has compromised the integrity of academic assessments and does not accurately reflect students' understanding and comprehension of the material taught.

2 Methodology

This project implements three versions of a plagiarism detection tool for source code. The first version uses the winnowing algorithm for comparing files; This serves as a control group. The second version integrates ANTLR (ANother Tool for Language Recognition) with the winnowing algorithm for more structured code analysis. The third version leverages the Tree-Sitter framework to parse and abstract source files much like ANTLR, but with some key differences that shall be elaborated upon later.

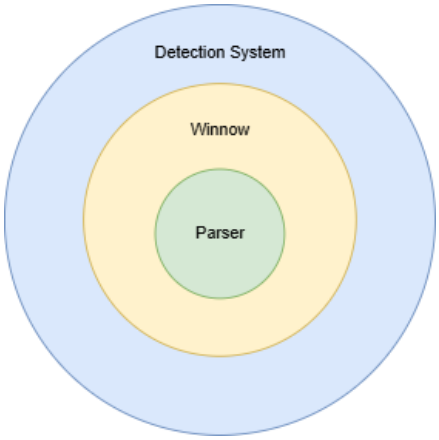


Figure 1: An abstraction of the plagiarism detection system

At a high level, variants of the detection system will be developed by reconfiguring the parser layer, which can also be interpreted as the layer that feeds into the winnow algorithm. Note that the winnowing algorithm is the central algorithm of all configurations, including the *null* parser (no parser nor syntax tree; winnowing over raw text only).

2.1 *Winnowing*

Winnowing is an algorithm that is often used in plagiarism detection to find similarities between files by identifying common substrings. The algorithm identifies common substrings by generating "fingerprints" from a sequence of hashes derived from k-grams, which are contiguous sequences of characters [1]. The main idea behind the winnowing algorithm is to hash substrings of text (k-grams) and then select a subset of hashes (fingerprints) that provides a concise representation of the text. These fingerprints are then compared throughout to detect potential plagiarism between files.

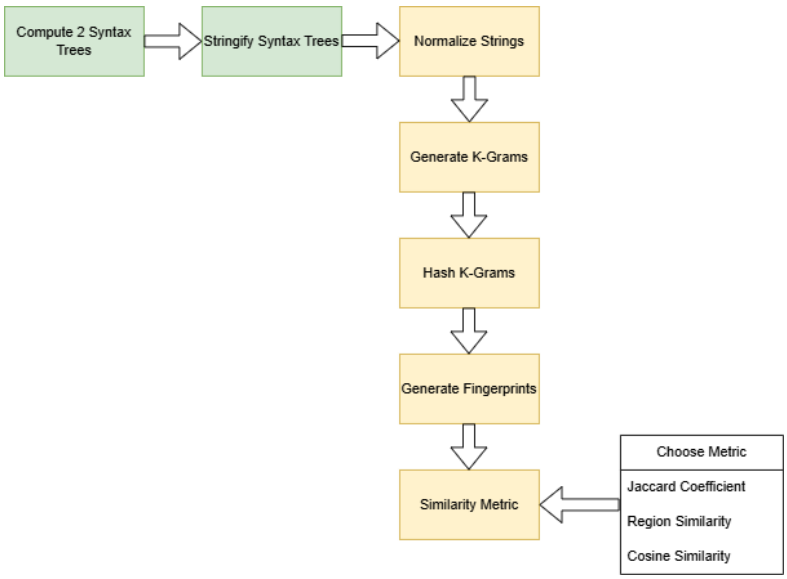


Figure 2: A visualization of the winnowing pipeline

2.1.1 Key Components

1. Normalization

   Before we use the winnowing algorithm to generate k-grams, the input source code undergoes a normalization process. This step removes comments, unnecessary whitespaces, and converts the code to lowercase. Doing so eliminates irrelevant differences that could interfere during the detection

process for potential similarities and ensures a more accurate comparison of the logic and structure of the code.

2. K-Gram

   The winnowing algorithm is used to generate k-grams from the input source code. K-grams are contiguous substrings of length k that are used to break down the code into smaller, manageable parts to be processed [2]. The value of the k-gram selected highly impacts the result of the plagiarism detection process as the substring size defines the fingerprint granularity. A finer granularity (smaller k-gram value) makes the fingerprint more prone to false matches, whereas a coarser granularity (larger k-gram value) increases the fingerprint's sensitivity to changes [3]. The plagiarism detection software uses 20 as the value for k-gram size, which creates 20-character long substrings, and calculates hashes for each substring.

3. Hashing

   Each k-gram is hashed using a hash function (SHA-1), to convert the string into a fixed-size hash value. This step helps to create a compact representation of the code, which allows for efficiency during the comparison phase later on.

4. Sliding Window

   For fingerprint generating, which we will mention below, we use a fixed-size sliding window across the code and select fingerprints from the hashes of k-gram. A smaller window size increases detection sensitivity by capturing more granular patterns, but it results in more hashes, leading to higher storage requirements and reduced efficiency. On the other hand, a larger window size reduces the number of hashes and improves processing efficiency but may decrease sensitivity to smaller instances of plagiarism [4]. The plagiarism detection software uses 3 as the value for window size, which means that it examines every consecutive group of 3 k-grams and selects the minimum hash value from each group to form the file’s fingerprint.

5. Fingerprint Selection/ Generation

   Using a winnowing algorithm, only one chunk is selected as a fingerprint from each winnowing window. This involves applying a sliding window of size 3 across the hashes of the k-grams, with the window selecting the minimum hash value from each group of 3 consecutive hashes [4]. The hash value chosen will serve as the fingerprint for that particular window.

6. Fingerprint Comparison

   The final step in the winnowing process is to compare the fingerprints between files to identify similarities. By identifying common fingerprints between the two files, the algorithm can determine the percentage of similarity. Files with a higher number of common fingerprints are more likely to be plagiarized and will have a higher plagiarism percentage in the output.

Algorithm $Winnow(k, w, d1, d2)$ {

***Input**:*

k := length of a singular substring gram

w := length of sliding window during fingerprinting stage

d1 & d2 := document 1 and document 2 containing java sources

***Output**:*

m := A similarity metric between d1 & d2

d1, d2* ← Preprocess d1 & d2*

h1, h2 ← Hash K-grams of d1 & d2**

f1, f2 ← Select Fingerprints from h1 & h2 hashes

m ← Compute similarity metric for f1 & f2

Return m

}

***2.2** ANTLR + Winnowing*

ANTLR, short for ANother Tool for Language Recognition (formerly known as PCCTS), is a language processing tool designed to build recognizers, compilers, and translators. It enables developers to define grammatical descriptions that include actions written in Java, C++, or C#, providing a flexible framework for various language-based applications [5]. ANTLR's ability to generate parsers and lexers from grammar definitions simplifies the process of analyzing and transforming text and/or code. The software supports a wide range of advanced features, such as syntax tree generation, error handling, and language-specific actions, making it a powerful tool for developers in creating tailored language analyzers or translators.

2.2.1 Key Components

1. Grammar Definition

   A grammar file for ANTLR defines the syntax rules of a programming language. It outlines how the language's tokens and structures are formed. This file allows the parser to recognize and process source code written in that language by breaking it down into a structured representation [6]. By using grammar files, it becomes easier to extend clone detection tools to support multiple programming languages without having to rewrite the core detection logic. In our program, we define the syntax rules for the plagiarism detection software inside srcANTLR/PlagiarismTool.g4.

2. Normalizing and Parsing with ANTLR

   Once the grammar is defined, ANTLR's parser processes the source code by first tokenizing it with the lexer, then generating a parse tree that represents the syntactic structure of the code. The lexer breaks the source code into tokens, and the parser analyzes these tokens according to the grammar rules defined in the grammar file. In the software, the parseWithANTLR method generates this parse tree and normalizes it into a string representation, which is then used in subsequent analysis steps, such as generating k-grams and comparing fingerprints for plagiarism detection.

3. ANTLR Parse Tree

   A parse tree is an ordered structure that represents the syntactical organization of the code, generated by the ANTLR parser based on formal grammar. The method parseWithANTLR generates the parse tree using the ANTLR lexer and parser. It tokenizes the input code using PlagiarismToolLexer.java and applies grammatical rules defined in the PlagiarismToolParser.java. The parse tree is then transformed into a textual representation using toStringTree where it is then compared and analysed for similarities.

2.3 *Tree-Sitter + Winnowing*

As per the official documentation of Tree-Sitter, the Tree-Sitter library is both a *parser generator tool* and *incremental parsing library*. In short, a parser generator is a framework that configures various parser programs for different formal grammars, typically producing variants of a syntax tree for structural analysis. Tree-Sitter is also an incremental parsing library, meaning that it can produce and edit these syntax trees in real time, thus enabling the production of efficient text editors, compilers, interpreters, and so forth. In fact, Tree-Sitter is the underlying framework that drives many text editors like Atom and GNU Emacs amongst others.

2.3.1 Key Components

1. Stringifying Tree-Sitter

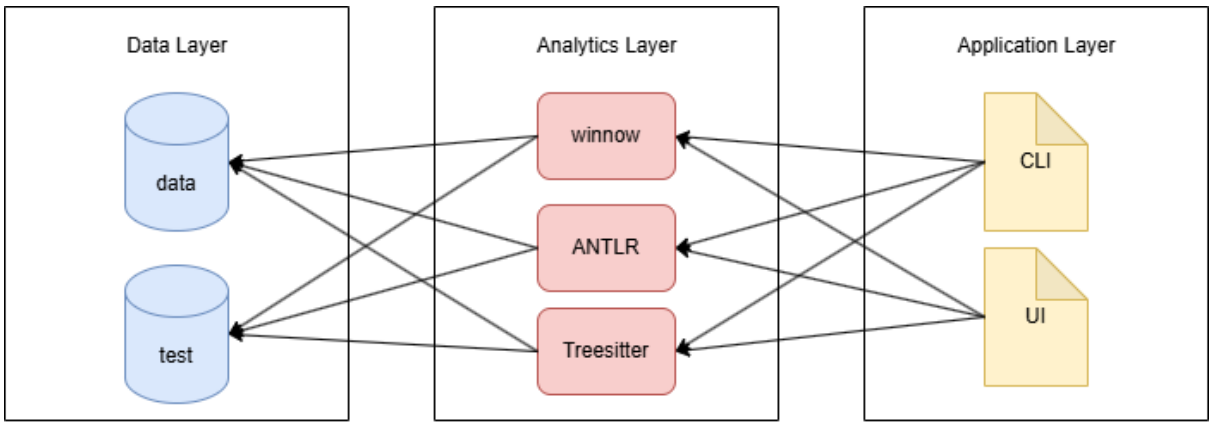
   The parser module is the interface that invokes the Tree-Sitter library. It auto generates distinct syntax trees for each test sample pair and then *stringifies* them. It should be noted that, unlike ANTLR, Tree-Sitter does not offer a method to stringify a tree; Hence, the method to stringify a given tree was manually implemented by walking through a particular syntax tree in preorder traversal.

2. A Notice on Tree-Sitter's Syntax Tree

   The syntax tree generated by Tree-Sitter's Java parser follows a fundamentally different format than ANTLR's LISP format. By inspection, Tree-Sitter appears to formulate a more compactified representation than its more verbose ANTLR counterpart. As per their official documentation, Tree-Sitter generates a DOM-style type structure that is composed of syntax nodes.

2.4 *Program Architecture*

The project utilizes the dataset available from the Zenodo repository titled "Supplementary Material for 'Detecting Automatic Software Plagiarism via Token Sequence Normalization'" [7]. This dataset contains various forms of plagiarism, showcasing different code manipulation techniques such as insertion, white spacing, reordering, and other modifications. All code samples in the dataset are written in Java, providing a consistent programming language for analysis.



2.5 *Similarity Metrics*

2.5.1 Cosine Similarity

The cosine similarity coefficient is a measure of similarity that computes the dot product between 2 vectors in an n-dimensional space. As per the formula, the vector dot product is divided by the product of their norms. We leveraged the cosine similarity coefficient by formulating 2 vectors - A and B - as the vector of the generated fingerprints from the winnow pipeline.

$$\frac{A \cdot B}{||A||\,||B||}$$

2.5.2 Jaccard Coefficient

The jaccard coefficient is another measure of similarity that compares the ratio of cardinalities between the intersection over union between 2 sets. Similarly, the jaccard coefficient was implemented by formulating vectors A and B as the sets of fingerprints generated from the winnow pipeline

$$\frac{|A \cap B|}{|A \cup B|}$$

2.5.3 Dice-Sorensen Coefficient

The Dice-Sorensen coefficient is another powerful similarity metric with a similar idea to the jaccard coefficient. Its formulation is as below; Again, we formulate sets A and B as the sets of fingerprints generated from the winnow pipeline.

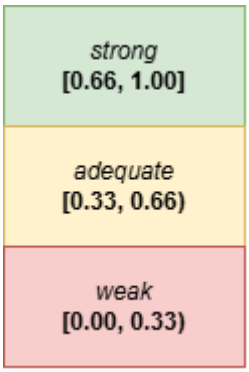
$$\frac{2\,|A \cap B|}{|A| + |B|}$$

3 Data

The following 9 tables report a non-exhaustive snippet of similarity metric calculations between data files from our referenced data repository. Each data table is a configuration of 2 architectural parameters: *detection algorithm* (Winnow, ANTLR + Winnow, Tree-Sitter + Winnow) and *similarity metric* (cosine, jaccard, dice).

The columns *Insert, Insert after Reorder, and Reorder* correspond to different modifications to a shared java file which serves as a control group. “Insert-only” modifications correspond to changes to the original source characterized by adding new lines of code. “Reorder-only” modifications correspond to changes to the original source where a number of lines are swapped in place. “Insert after Reorder” changes simply represent those modified files that swap lines in the original source followed by insertions of new java code. For instance, using the pure winnow algorithm (with no parser), we found that our system detected a 40.61% cosine similarity between the first data file and its corresponding modified “Insert-Only” mock; This corresponds to cell 1.

The computed similarity metrics are colour-coded as a function of detection strength to ease reader digestion. Since all similarity metrics map to a value in the range $[0, 1]$, we define 3 detection strength categories below:



{Winnowing} && {Cosine Coefficient}			
File Index	Insert	Insert after Reorder	Reorder
1 pa.java	40.61%	34.84%	90.77%
2 Social.java	48.52%	43.47%	83.44%
3 socio.java	53.14%	49.76%	83.47%
4 Sociologia.java	50.35%	47.86%	83.27%
5 Socialogia.java	52.57%	50.55%	88.03%
6 ProblemA.java	36.36%	32.23%	73.98%
7 Sociologia.java	51.61%	43.85%	84.74%
8 Main.java	36.89%	29.23%	72.06%
9 ProblemaF4.java	34.86%	29.68%	73.59%
10 Sociologia.java	52.76%	50.00%	88.94%

Figure 3.1

{Winnowing} && {Jaccard Coefficient}			
File Index	Insert	Insert after Reorder	Reorder
1 pa.java	24.82%	20.56%	83.10%
2 Social.java	39.10%	36.86%	84.99%
3 socio.java	34.06%	27.52%	73.51%
4 Sociologia.java	37.72%	32.97%	83.03%
5 Socialogia.java	28.80%	24.72%	83.03%
6 ProblemA.java	29.68%	28.99%	79.24%
7 Sociologia.java	33.05%	28.89%	69.62%
8 Main.java	37.30%	36.57%	87.41%
9 ProblemaF4.java	29.75%	24.73%	59.88%
10 Sociologia.java	29.67%	25.86%	69.18%

Figure 3.2

{Winnowing} && {Dice Coefficient}			
File Index	Insert	Insert after Reorder	Reorder
1 pa.java	39.77%	34.11%	90.77%
2 Social.java	47.64%	42.82%	83.44%
3 socio.java	52.52%	49.23%	83.47%
4 Sociologia.java	49.62%	47.24%	83.27%
5 Socialogia.java	51.82%	49.92%	88.03%
6 ProblemA.java	35.36%	31.28%	73.98%
7 Sociologia.java	50.82%	43.16%	84.73%
8 Main.java	35.86%	28.52%	72.06%
9 ProblemaF4.java	34.10%	29.11%	73.59%
10 Sociologia.java	52.09%	49.34%	88.94%

Figure 3.3

{ANTLR} && {Cosine Similarity}			
File Index	Insert	Insert after Reorder	Reorder
1 pa.java	51.14%	45.07%	93.24%
2 Social.java	55.44%	52.34%	86.89%
3 socio.java	57.84%	54.00%	84.87%
4 Sociologia.java	56.64%	54.58%	86.61%
5 Socialogia.java	57.45%	55.89%	89.68%
6 ProblemA.java	43.48%	39.83%	78.05%
7 Sociologia.java	57.75%	52.13%	86.32%
8 Main.java	46.19%	40.04%	76.22%
9 ProblemaF4.java	41.89%	37.82%	77.23%
10 Sociologia.java	59.13%	56.92%	91.18%

Figure 3.4

{ANTLR} && {Jaccard Coefficient}			
File Index	Insert	Insert after Reorder	Reorder
1 pa.java	33.66%	28.45%	87.33%
2 Social.java	45.16%	42.60%	87.78%
3 socio.java	39.78%	34.59%	75.94%
4 Sociologia.java	38.42%	35.96%	88.33%
5 Socialogia.java	37.72%	32.97%	86.55%
6 ProblemA.java	36.11%	35.33%	81.28%
7 Sociologia.java	38.59%	36.16%	74.40%
8 Main.java	41.21%	41.93%	97.05%
9 ProblemaF4.java	34.97%	29.68%	63.70%
10 Sociologia.java	36.61%	34.00%	74.94%

Figure 3.5

{ANTLR} && {Dice Coefficient}			
File Index	Insert	Insert after Reorder	Reorder
1 pa.java	50.37%	44.30%	93.24%
2 Social.java	54.74%	51.82%	86.89%
3 socio.java	57.31%	53.54%	84.87%
4 Sociologia.java	55.94%	53.97%	86.61%
5 Socialogia.java	56.68%	55.22%	89.68%
6 ProblemA.java	42.47%	38.95%	78.05%
7 Sociologia.java	56.92%	51.40%	86.32%
8 Main.java	45.31%	39.28%	76.22%
9 ProblemaF4.java	41.19%	37.30%	77.23%
10 Sociologia.java	58.53%	56.33%	91.18%

Figure 3.6

{Tree-Sitter} && {Cosine Coefficient}			
File Index	Insert	Insert after Reorder	Reorder
1 pa.java	81.46%	75.52%	96.22%
2 Social.java	77.46%	80.43%	93.10%
3 socio.java	84.59%	74.15%	91.73%
4 Sociologia.java	79.44%	79.11%	88.23%
5 Socialogia.java	79.44%	77.30%	94.38%
6 ProblemA.java	80.49%	80.00%	90.00%
7 Sociologia.java	83.34%	74.72%	93.26%
8 Main.java	78.91%	72.42%	89.20%
9 ProblemaF4.java	78.19%	84.68%	87.91%
10 Sociologia.java	87.38%	83.49%	95.65%

Figure 3.7

{Tree-Sitter} && {Jaccard Coefficient}			
File Index	Insert	Insert after Reorder	Reorder
1 pa.java	68.34%	63.12%	92.70%
2 Social.java	62.59%	60.01%	87.07%
3 socio.java	73.23%	67.19%	84.70%
4 Sociologia.java	65.60%	58.66%	78.90%
5 Socialogia.java	65.29%	65.01%	89.33%
6 ProblemA.java	67.30%	62.93%	81.79%
7 Sociologia.java	71.13%	66.35%	87.37%
8 Main.java	64.89%	59.38%	80.47%
9 ProblemaF4.java	64.10%	56.65%	78.42%
10 Sociologia.java	77.45%	73.20%	91.66%

Figure 3.8

{Tree-Sitter} && {Dice Coefficient}			
File Index	Insert	Insert after Reorder	Reorder
1 pa.java	81.19%	77.39%	96.21%
2 Social.java	76.99%	75.01%	93.08%
3 socio.java	84.55%	80.37%	91.71%
4 Sociologia.java	79.23%	73.94%	88.21%
5 Socialogia.java	79.00%	78.80%	94.36%
6 ProblemA.java	80.45%	77.25%	89.98%
7 Sociologia.java	83.13%	79.77%	93.26%
8 Main.java	78.71%	74.51%	89.18%
9 ProblemaF4.java	78.12%	72.32%	87.90%
10 Sociologia.java	87.29%	84.53%	95.65%

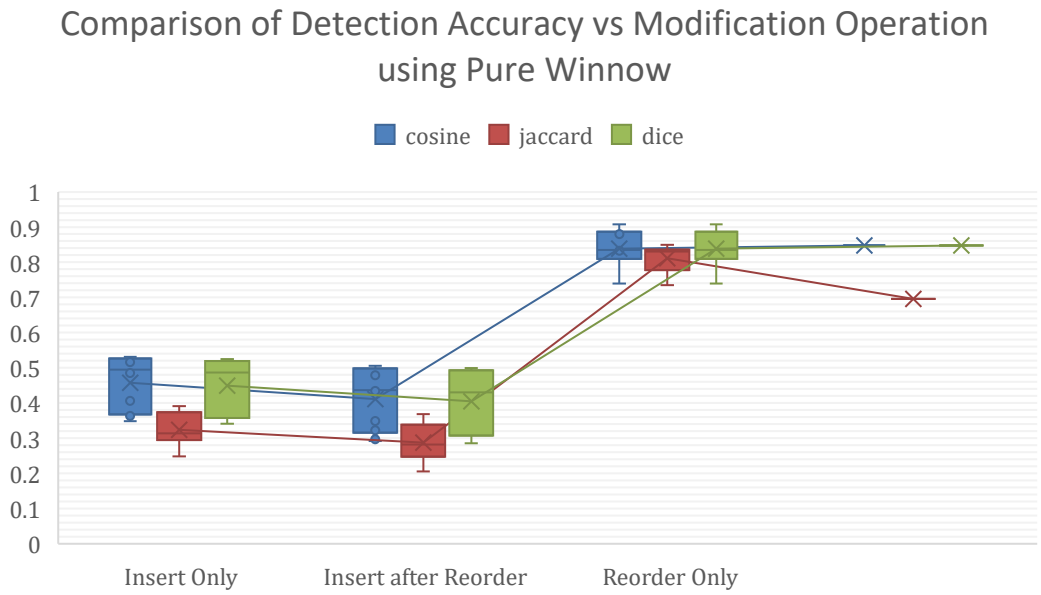
Figure 3.9

4 Discussion

4.1 *Key Findings*

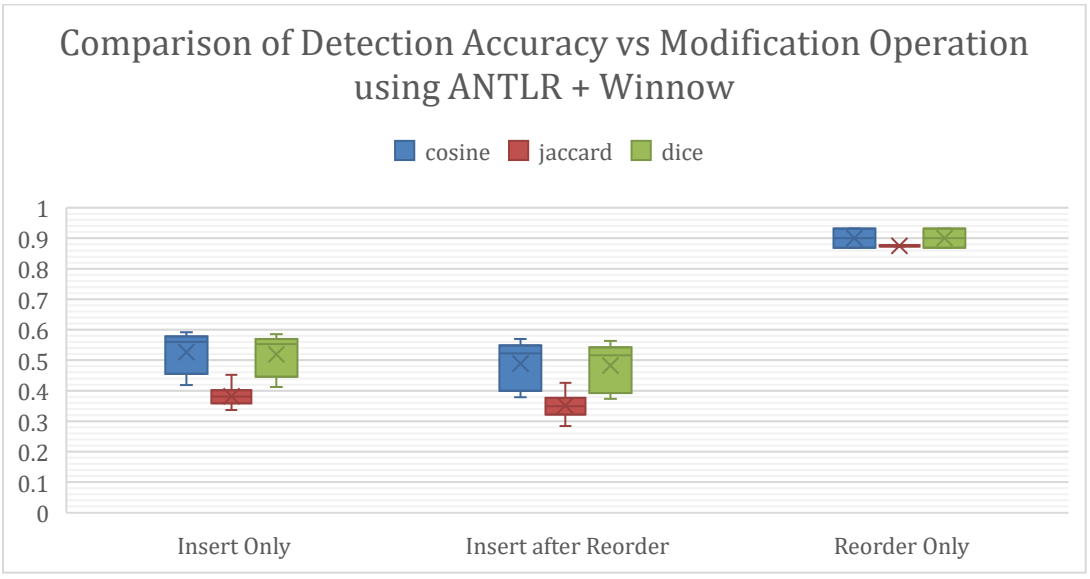
4.1.1 Distribution of Similarity Metrics

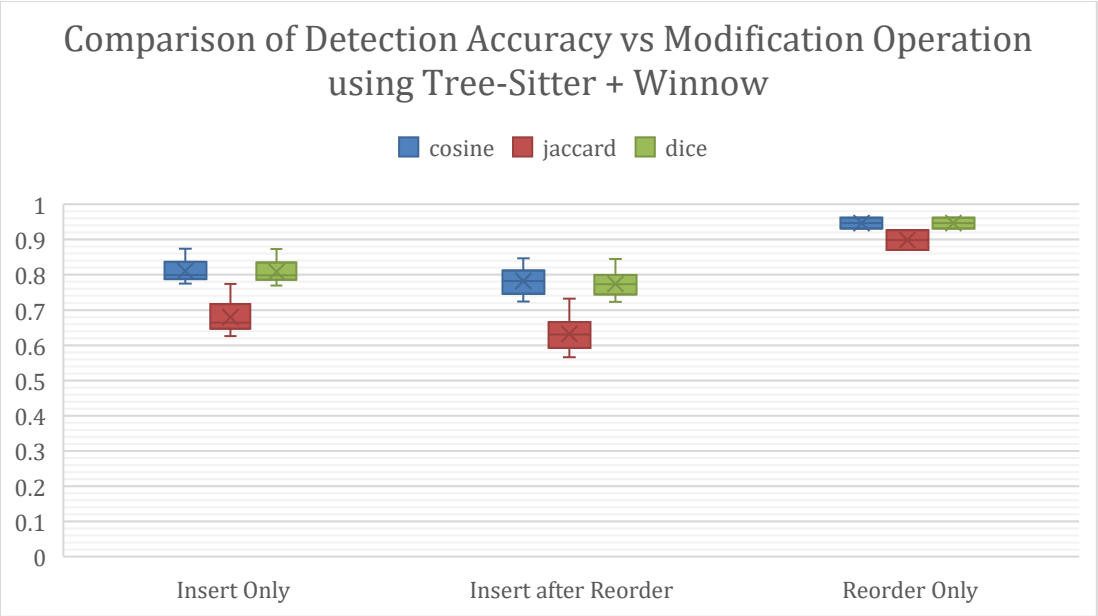
Across all data configurations we found that detecting plagiarism from Reorders-Only were superior in detection accuracy compared to Insert-Only and Insertions after Reorders. Observe the following boxplot that graphs the detection accuracies of the previous data tables for winnow algorithm only.



In particular, the winnow-only distribution yields an arithmetic average of 0.41, 0.37, and 0.81 detection strengths for Insert-Only, Insert-after-Reorder, and Reorder-Only operations across all similarity metrics. With respect to our detection strength classifications, this corresponds to adequate detection strengths for all but Reorder-Only operations.

It should be noted that detection distributions follow a similar trend for detection configurations that leverage parsers and syntax trees. The box plot representations for ANTLR and Tree-Sitter detection-strength distributions are given below in their respective order:





Notice by inspection, that the best overall detection configuration appears to be Tree-Sitter + Winnow. We've arrived at the conclusion that the reasons for this are likely multi-fold, with *implementation details* playing a key factor in outcome success. More specifically, detection strength is almost certainly dependent on the how a particular parser is generated with respect to what grammar file is used, whether a language binding was utilized to interface the specification tool, and so forth.

We investigated the detection performance of the Tree-Sitter + Winnow system configuration by implementing 2 different versions of it. One implementation leveraged a community-developed JavaScript binder, and another implementation scripted the Tree-Sitter implementation in the framework's native Rust. *All data results corresponding to the Tree-Sitter implementation thus far report the performance of our Rust implementation*.

To our surprise, we found that the JavaScript implementation of Tree-Sitter + Winnow performed similarly to the ANTLR implementation whilst taking orders of magnitude longer to run. In particular, the Rust implementation took <3000ms to process all data files whereas the JavaScript version took an average of 7 minutes on our local machine. Such a vast discrepancy likely stems from differences in native optimizations, considering that Tree-Sitter was constructed in Rust.

4.1.2 False Negative Rate

Surprisingly, or rather not surprisingly, the Tree-Sitter + Winnow configuration implemented in Rust never dropped below a 50% detection threshold across all metrics and modification operations. On the other hand, the detection accuracies of pure Winnow and ANTLR + Winnow comparatively fell behind. Observe the summary statistics below:

System	Minimum Accuracy	Maximum Accuracy	Average Accuracy	False Neg. Rate	Favourable Metric	Unfavourable Metric
Winnow	20.56%	90.77%	52.81%	21.11%	Dice	Jaccard
ANTLR + Winnow	28.45%	97.05%	58.54%	3.33%	Cosine/Dice	Jaccard
Tree-Sitter + Winnow	56.65%	96.22%	79.77%	0%	Cosine/Dice	Jaccard

The false negative rate is defined as the equiprobability across all metrics and modification operations of a given detection threshold to be classified as weak. Then note that Tree-Sitter was the strongest in terms of its false negative rate with ANTLR trailing closely behind. Tree-Sitter also reigned superior in its average accuracy over Winnow and ANTLR. Interestingly, ANTLR did not fair significantly better than the pure Winnow algorithm which upon investigation we hypothesize is due to the grammar file used to construct the ANTLR parser. Moving forward, a reinvestigation of ANTLR's performance should be done by using a community-generated or official Java grammar file to construct the parser instead of writing it from scratch.

One final insight to note is that across all system configurations, the Jaccard similarity metric was the most optimistic in terms of failing to report plagiarism. On the other hand, the Dice metric appears to be the most pessimistic in terms of reporting plagiarism. However, this is an incomplete analysis as we are unable to test false positives with the given data repository. Thus, an improvement that should be considered for future analyses is to include datasets that are not plagiarized in order to test false positive rates.

4.1.3 Detection Speed

The winnowing and ANTLR systems had extremely fast processing and detection speeds (under 3000ms). However, we noticed that the tree-sitter system took much longer compared to the other two systems (5000ms–10000ms). We attributed this to the stringifyTree function, which recursively traverses the entire syntax tree to generate a string representation.

As a result, this could significantly slow down the process, as it is potentially resource intensive. Additionally, a limitation exists in our research due to the size and number of files used, meaning the performance may not fully reflect the system's capabilities when dealing with larger datasets or more complex documents, which could potentially increase processing time and detection speed.

4.2 *Strength and Weaknesses of Each System Configuration*

4.2.1 Winnowing

The strength in the winnowing system lies in its simple, yet effective approach in detecting plagiarism. Its computational efficiency and ease of implementation, makes it an ideal system for basic plagiarism detection tasks. Additionally, the system does not rely on additional libraries, which makes it easier to implement, as well as more lightweight in nature.

Alternatively, the system has several weaknesses that can limit its effectiveness. For instance, its lack of understanding for deeper context and logic may result in the system overlooking more complex forms of plagiarism. Since it solely focuses on raw strings or tokens, it does not consider structural aspects, such as syntactic analysis.

4.2.2 Winnowing + ANTLR

Combining Winnowing with ANTLR addresses the limitation mentioned previously. By incorporating ANTLR, the system is able to perform syntactic analysis using the lexer and parser generated from the grammatical rules. ANTLR's ability to parse code into a structured format allows the system to detect complex plagiarism, such as code restructuring or renaming, which Winnowing alone might miss. This deeper analysis improves the accuracy in identifying similarities in code logic, rather than just surface-level text. However, the limitations of using ANTLR mainly lies in its heavy dependency on grammar accuracy. If the grammar is not well-defined, it may miss syntactic elements, which can lead to false negatives. Additionally, the current grammar (PlagiarsmTool.g4) may have limited coverage due to its simplicity in the current version. The current grammar focuses on basic constructs, and does not cover more complex language constructs such as functions, loops, classes, conditionals, etc. As a result these limitations may hinder the system's ability to handle more complex codebases.

4.2.3 Winnowing + Tree-Sitter

By utilizing the Tree-Sitter framework to generate parse trees, and then feeding those parse trees into the winnowing pipeline, we find that plagiarism detection accuracy improved drastically as opposed to pure winnowing.

5 Conclusion

In our analysis, we ultimately found that leveraging language specification tools to generate syntax trees as input to the winnowing pipeline generally increased detection accuracy. We observed this phenomenon by hosting a control group of test samples using the pure winnowing framework against detection system configurations that leveraged parsers.

By every metric of similarity, those configurations that leveraged parsers outperformed the pure winnow algorithm. It should be noted however, that although leveraging Tree-Sitter's parser offered a significant advantage -26.96% increase in expected detection accuracy – over the control dataset, the ANTLR configuration only mildly outperformed the pure winnow system – a 5.73% increase in expected accuracy.

Although both the ANTLR and Tree-Sitter frameworks leverage syntax trees to generate logical representations of our source files, we do not necessarily believe that the Tree-Sitter framework is inherently more efficient or accurate than ANTLR with respect to detection strength and false negative rating despite their statistical disparities. Further investigation must be done in order to claim such a conclusion, as we noted several potential confounding variables in our limited analysis.

Such confounding variables include differences in implementation, language, and grammar files. A future report that aims to unify such variables would fare better in analysing the differences between any plagiarism detection systems that leverage lexical specification tools. But for our purposes, it appears to be the case that leveraging AST structures in a plagiarism detection system generally outperforms those that do not.

One would hypothesize that applying the winnow algorithm over stringified source files would lead to more accurate outcomes over winnowing raw strings of text. Intuitively, this makes sense, as leveraging such AST structures would allow the tokenization and fingerprinting process within the winnow algorithm to focus more on relationships between nodal structures and not just local regions. That is not to say that winnowing does not provide context on relationships between local regions in the text, because it does particularly during the fingerprint computations and similarity metric calculations, but incorporating a layer of explicit connections via an AST structure between local structures in the text provides the winnow algorithm even greater context between regions in the document. Fascinatingly, our findings provide confirming evidence for this hypothesis.

References

[1] Schleimer, S., Wilkerson, D. S., & Aiken, A. (2003). Winnowing: Local algorithms for document fingerprinting. *In Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* (pp. 76–85). Association for Computing Machinery. https://doi.org/10.1145/872757.872768

[2] R. Sutoyo, I. Ramadhani, A. D. Ardiatma, S. C. Bavana, H. L. H. S. Warnars, and A. Trisetyarso, "Detecting documents plagiarism using winnowing algorithm and k-gram method," *2017 IEEE International Conference on Cybernetics and Computational Intelligence (CyberneticsCom)*, Phuket, Thailand, Nov. 20-22, 2017, pp. 1-5, doi: 10.1109/CYBERNETICSCOM.2017.8311686.

[3] L. Lulu, B. Belkhouche and S. Harous, "Overview of fingerprinting methods for local text reuse detection," *2016 12th International Conference on Innovations in Information Technology (IIT)*, Al Ain, United Arab Emirates, 2016, pp. 1-6, doi: 10.1109/INNOVATIONS.2016.7880050.

[4] X. Duan, M. Wang, and J. Mu, "A Plagiarism Detection Algorithm based on Extended Winnowing," *ATEC Web of Conferences*, vol. 128, p. 02019, 2017, DOI: 10.1051/matecconf/201712802019.

[5] T. Parr, *ANTLR Reference Manual*, Version 2.7.3, University of San Francisco, Mar. 22, 2004. [Online]. Available: http://www.antlr.org/doc/index.html. Accessed: Dec. 1, 2024.

[6] Y. Semura, N. Yoshida, E. Choi, and K. Inoue, "Multilingual Detection of Code Clones Using ANTLR Grammar Definitions," *2018 25th Asia-Pacific Software Engineering Conference (APSEC)*, Nara, Japan, 2018, pp. 673-677, doi: 10.1109/APSEC.2018.00088.

[7] T. Sağlam, M. Brödel, L. Schmid, and S. Hahner, "Supplementary Material for 'Detecting Automatic Software Plagiarism via Token Sequence Normalization'," *46th International Conference on Software Engineering (ICSE 2024)*, Lisbon, Portugal, Apr. 14-20, 2024. DOI: 10.5281/zenodo.10430322.