Incorporation of Text Content Similarity into Token-Based Source Code Plagiarism Detection

Bachelor's Thesis of

Moritz Rimpf

At the KIT Department of Informatics
KASTEL – Institute of Information Security and Dependability

First examiner:	Prof. Dr. Ralf Reussner
Second examiner:	Prof. Dr.-Ing. Anne Koziolek
First advisor:	Robin Maisch, M. Sc.
Second advisor:	Dr.-Ing. Timur Sağlam

12. May 2025 – 12. September 2025

Abstract

In an academic context, plagiarism is a serious threat to the integrity of exercises given to students, especially in introductory programming courses. For various reasons, students may, intentionally or unintentionally, attempt to plagiarize solutions for given exercises. Since introductory programming courses often have a large number of students and the complexity of solutions is not negligible, instructors often employ plagiarism detection systems to be able to spot possible cases of plagiarism.

Current state-of-the-art systems often disregard textual content in the source code during the detection of plagiarism. However, recent developments have shown that possible cases of plagiarism often do contain similar or identical textual constructs, such as inline or documentation comments. These comments could be used to find cases of plagiarism more easily, or further aid instructors during the manual review of suspicious submissions. Therefore, in this thesis, we enhance plagiarism detection engines by introducing the matching of textual components in the form of source code comments into the plagiarism detection.

To process comments during the plagiarism detection, we introduce a three-step comment processing pipeline, which extracts comments from the source-code, compares them among the submissions, and merges the results into the result of the plagiarism detection. Furthermore, we compare three different matching algorithms for matching comments and explore the impact of natural language preprocessing of the comments inside this pipeline.

The pipeline with all matching algorithms was evaluated and compared on both datasets containing real student submissions from a first-year programming course and datasets containing submissions fully generated by artificial intelligence. These results demonstrate the effectiveness of the comment processing pipeline in real applications.

Zusammenfassung

Im akademischen Kontext ist Plagiarismus eine ernstzunehmende Gefahr für die Integrität von Aufgaben für Studierende, insbesondere in einführenden Programmierkursen. Aus verschiedensten Gründen können Studenten versuchen, absichtlich oder unabsichtlich, Plagiate für die Lösungen von Aufgaben anzufertigen. Da Einsteigerkurse häufig von vielen Studierenden besucht werden und die Abgaben eine gewisse Komplexität enthalten, nutzen Kursbetreuer häufig Plagiatserkennungssysteme, um mögliche Plagiate zu erkennen.

Moderne Plagiatserkennungssysteme ignorieren aktuell meistens Textinhalte des Quelltexts während der Plagiatserkennung. Allerdings hat sich gezeigt, dass mögliche Plagiate oft ähnliche oder identische Textinhalte beinhalten, häufig in Form von Zeilen- oder Dokumentationskommentaren. Diese Kommentare könnten genutzt werden, um mögliche Plagiatsfälle einfacher zu finden oder um Kursbetreuer während der manuellen Inspektion von auffälligen Abgaben zu unterstützen. Daher präsentiert diese Thesis eine Erweiterung für Plagiatserkennungssysteme, um den Vergleich von Textinhalten in Form von Quelltextkommentaren in die Plagiatserkennung einzubinden.

Um Kommentare während der Plagiatserkennung zu berücksichtigen, wird eine dreistufige Kommentarverarbeitungs-Pipeline eingeführt, welche Kommentare aus den Quelltextdateien extrahiert, diese untereinander vergleicht, und die Ergebnisse in die Ergebnisse der Plagiatserkennung einfügt. Darüber hinaus werden drei verschiedene Vergleichsalgorithmen für Kommentare verglichen und der Einfluss von Vorverarbeitung der Kommentare durch natürliche Sprachverarbeitung untersucht.

Die Pipeline mit allen Vergleichsalgorithmen wurde sowohl auf Datensätzen bestehend aus studentischen Abgaben als auch auf Datensätzen bestehend aus Abgaben, welche von künstlicher Intelligenz generiert wurden, evaluiert und verglichen. Die Ergebnisse demonstrieren die Effektivität der Kommentarverarbeitungs-Pipeline in realen Applikationen.

Contents

List of Figures

List of Tables

1 Introduction

In most computer science education courses, students are taught at least one programming language within the first couple of years, which is usually assessed by the instructors giving the students tasks to solve by implementing their solution in the selected programming language. Students may attempt to plagiarize in these assignments by using and (slightly) altering the solutions of other students due to their own incapability to solve the task or to simply minimize the work needed to finish the assignment [10]. With the recent developments in generative artificial intelligence, large language models (LLMs) may also be able to assist with obfuscating plagiarism or solving exercises. Possible use cases of LLMs are for example prompting an LLM to rewrite the input code without any changes to its functionality and therefore hiding a case of plagiarism. Alternatively, an LLM may simply be prompted to solve the assignment for the students, which may be a case of plagiarism depending on the sources used by the LLM, if not a violation of the rules in the first place.

To mitigate this issue, course instructors often employ automated *source code plagiarism detection systems* to find potential cases of plagiarism, which then have to be manually reviewed. These tools are able to efficiently compare many submissions at once, which is often needed due to the large number of students in a single course. These tools do not return which submissions contain plagiarism, but rather calculate a similarity score between source code documents [20], which then requires the course instructors to review the results manually, investigating submissions with a high similarity score to identify suspected cases of plagiarism in accordance with their local rules and regulations.

There are many different methods used by these tools, but the current state-of-the-art method is *token-based plagiarism detection*, which will be the focus of this work. In token-based plagiarism detection, each submission is first transformed into a sequence of tokens, which are then compared to each other. These token sequences are a simplification of the actual program source code, which keeps structural information intact, while removing information deemed unnecessary to the plagiarism detection. A lot of tools, such as JPlag [22], one of the most popular open-source token-based plagiarism detection systems, discard all forms of formatting information, like whitespace and line breaks, but also textual content, like comments and names of identifiers, since they are considered “easy to change” [22] and could simply be changed in an attempt to obfuscate cases of plagiarism.

However, in recent developments, it has been noted that some likely plagiarized submissions by students of our first-year programming course have showcased similar comments and comment structures across multiple submissions. These comments were not only similar across multiple submissions, but some of these comments had the style of some large language models, since they do usually generate comments in a similar fashion across

multiple prompts. Therefore, we will investigate whether the detection of source code plagiarism can be improved by incorporating textual constructs which were previously removed from the tokens, specifically source code comments. This could have the benefit of an improved detection of cases of plagiarism by the detection systems, as more content of the source documents are compared, but also usage of large language models might be more visible, as their similar comments are also detected in the comparison. In addition to the improvements to the detection engine, the incorporation of comments into the plagiarism detection may also benefit the instructors in their manual review: They receive a list of similar comments in addition to the similar code fragments and plagiarism in natural language, like comments, may be easier for humans to recognize than plagiarism in source code. However, it is an important requirement that the incorporation of comments is unable to decrease the similarity between submissions, as that would increase the attack surface on plagiarism detection systems.

This thesis investigates the effects of incorporating textual content into token-based source code plagiarism detection. To do so, we introduce a pipeline which extracts and tokenizes comments separately from the source code itself, compares the comments among each other using one of three different matching algorithms, and adds the result to the final result of the plagiarism detection.

We will first go over the necessary foundations of token-based source code plagiarism detection as well as text similarity metrics in Chapter 2. Afterward, the concept of incorporating textual content in the form of a comment processing pipeline will be detailed in Chapter 3. In Chapter 4 we will look at how other tools and authors handle comment processing in their code and tools. The results of our pipelines will be evaluated afterward on different datasets, both created by humans and by generative artificial intelligence, in Chapter 5. Lastly, we will discuss limitations and future work in Chapter 6, before concluding the work in Chapter 7.

2 Foundations

In this chapter, we want to explain some necessary foundations for this thesis. Since we are building upon the concept of *token-based source code plagiarism detection*, we will first introduce the concept of *Source Code Plagiarism* itself in Section 2.1 and then explain the basics of token-based source code plagiarism detection in Section 2.2. Since we are investigating textual content similarities in this thesis, we need to be able to compare text using different methods. One method which will be used are *string distance metrics*, which are explained in Section 2.3. The individual matching algorithms used in the thesis are detailed in Section 2.4. Lastly, since textual content in source code is usually made up of natural language text, we will look at the basics of *natural language processing* in Section 2.5.

2.1 Source Code Plagiarism

Whenever submissions for an exercise or something similar are collected, plagiarism becomes a serious concern and threat to the integrity of the exercise. This is especially true for programming assignments such as exercises for university students, since plagiarism in programming source code may be easier to obfuscate and harder to detect and prove.

There is no universally agreed upon definition for "*Source Code Plagiarism*" [20], but multiple different authors have used different definitions in their works. This could also be due to different institutions applying different rules on the allowed or disallowed reuse of programming source code, and therefore having different definitions of what classifies as plagiarism and how it should be handled. Prechelt et al. define program plagiarism as "[...] programs [that] are more or less copies of programs supplied by someone else [...]" [22]. Cosma and Joy suggest a new definition for source code plagiarism in an academic context which we will be using in this thesis: "Source-code plagiarism in programming assignments can occur when a student reuses [...] source-code authored by someone else and, intentionally or unintentionally, fails to acknowledge it adequately [...], thus submitting it as his/her own work." [5].

2.2 Token-based Source Code Plagiarism Detection

There are many different approaches to detecting plagiarism inside source code. Most source code plagiarism detection systems first transform the source code documents to some

intermediate representation, which is easier to work with. This intermediate representation is then subsequently compared with each other, to compute a similarity score in the end. In this thesis, however, we will focus on the current state-of-the-art approach of *Token-based Source Code Plagiarism Detection*.

In token-based source code plagiarism detection, the source code is first *tokenized* and *linearized*. During this step, a lot of information that is independent of the functionality of the source code is removed. This mostly includes constructs to make source code more readable for humans, which includes comments, names of identifiers or variables and formatting, like whitespace and line breaks. All these elements can be altered without influencing the functionality of the resulting program, and therefore are ignored in plagiarism detection [22], since it is an easily accessible method of obfuscating plagiarism.

Using the token sequence returned from the tokenization step, the token sequences from different source code submissions (in the following just *submissions*) are compared to find common matches. Since single token matches are insufficient to suggest a case of plagiarism, most plagiarism detection systems use a hyperparameter defining the minimal match length, requiring matches to consist of at least that number of tokens, discarding any matches that are not of the required length. This parameter can be fine-tuned to change the sensitivity of the plagiarism detection. For the actual matching algorithms, two common techniques are *greedy string tiling* [31] employed by JPlag [11] and *fingerprinting* [29] used by Moss [18].

After common matches have been found, the plagiarism detection computes a similarity score based on the amount of matched tokens, and the total token counts of either submission. There are different similarity scores which can be returned, for example average similarity, maximum similarity or minimum similarity. These similarity scores quantify how similar two submissions are. However, the similarity scores do not represent a binary classification of *case of plagiarism* and *no plagiarism*, therefore, a manual inspection of the results is required.

We will continue to focus on the two major parts of token-based source code plagiarism detection in this section, beginning with the tokenization of source code in Section 2.2.1 and afterwards discussing the results of the plagiarism detection, the similarity scores, in Section 2.2.2.

2.2.1 Tokenization of Source Code

Tokenization is defined as the process of parsing the source code and transforming it into a stream of tokens. This stream of tokens, the *token sequence*, represents the raw structure of a program. Individual tokens might represent the start and end of methods or initialization and assignment of variables, as seen in Figure 2.1.

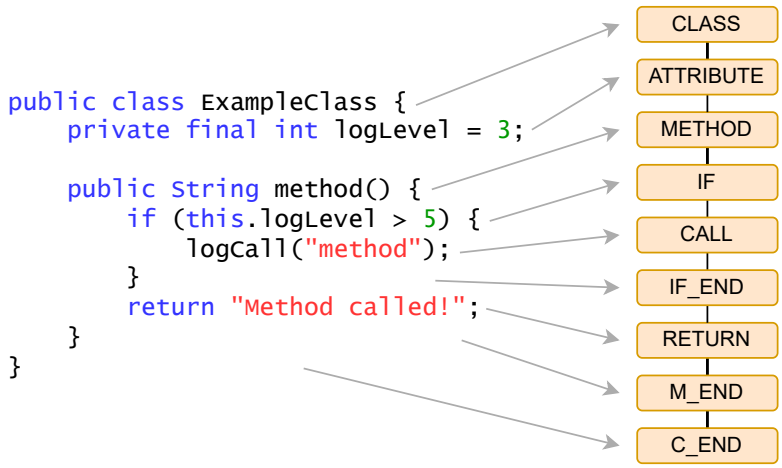


Figure 2.1: Example for Tokenization of Source Code.

The whole process of tokenization is programming language-specific and requires implementing some form of parser, scanner or grammar for the programming language to be tokenized. However, the resulting token stream can then be analyzed and compared language-agonistically.

2.2.2 Similarity Scores

Source code plagiarism detection systems compute a similarity score between pairs of submissions, based on the matching tokens found by the matching algorithms, as well as the total token count of the submissions. These similarity scores are then returned to the user, who then has the option to manually inspect the most similar submission pairs to find cases of plagiarism. It is important to note however, that, despite the name, these plagiarism detection systems actually do not detect cases of plagiarism, returning a true binary classification of *case of plagiarism* and *no plagiarism*, but rather just compute a similarity score between pairs of source code documents [20]. Whether a match between two submissions is just a coincidence, a result of allowed source code reuse or an actual case of plagiarism cannot be determined by these tools, and depends on the local rules and regulations. Therefore, a review of the results of these tools by instructors or other officials is always necessary, to find the actual cases of plagiarism.

2.3 String Distance Metrics

When comparing text, *string distance metrics*, sometimes also called *string similarity metrics* or just *string metrics*, can be used to quantify the *similarity* or *distance* between two strings of text. How exactly the distance between two strings is calculated depends on the exact metric used.

There are many different string distance metrics, each with their own benefits and disadvantages, which makes them applicable in different scenarios. Cohen et al. [4] categorizes string metrics into

- *edit-distance like functions*, which assign a cost to each *edit operation* and sum the costs for the best sequence of *edit operations* to turn one string into another one,
- *token-based distance functions*, which considers strings to be multisets of tokens consisting of individual words, which can then be compared for example using vector similarity functions, and
- *hybrid distance functions*, which combine both approaches.

By transforming tokens into a textual representation and concatenating them together, string distance metrics can also be applied to token sequences, getting the distance of their textual representations.

2.4 Matching Algorithms

After extracting token sequences from source code, as explained above in Section 2.2.1, we want to compute a similarity score for pairs of submissions. There are different algorithms for matching token sequences, three different algorithms were selected for this thesis, and explained below. Section 2.4.1 explains *Greedy String Tiling*, an algorithm designed to find common subsequences inside token streams. Afterward, a string distance metric and its extension will be explained, being the *Levenshtein Distance* in Section 2.4.2 and the *Damerau-Levenshtein Distance* in Section 2.4.3. Lastly, *Bag of Words*, a model transforming tokens into a vector space, is explained, in Section 2.4.4.

2.4.1 Greedy String Tiling

Greedy String Tiling, as proposed by Wise in [31], is an algorithm for finding similarities between two token sequences by finding the longest common subsequences. It was originally proposed with the intended purpose of finding possible sources of plagiarism inside programs or other text files.

Greedy String Tiling works by iteratively finding all non-overlapping common subsequences within two token sequences above a certain threshold, called the *minimum-match-length*. Iteratively, unmarked tokens of two sequences are compared to find common subsequences,

while the longest common subsequences are stored and marked, so they will be ignored in following iterations. This process is repeated until no unmarked match longer than the minimum-match-length is found.

Wise also introduces an optimization to Greedy String Tiling by including and extending the *Karp-Rabin string-match algorithm*, which introduces initial matching based on hash-values of token subsequences instead of having to iterate through both string multiple times. This improvement greatly reduces the time complexity of the algorithm.

2.4.2 Levenshtein Distance

The *Levenshtein distance* is a simple string metric based on an *edit-distance like function*, as explained in Section 2.3. For the Levenshtein distance, the edit operations are defined as follows:

- The addition of one character to a string
- The removal of one character from a string
- The substitution of one character with another within a string

Each of these edit operations has the same cost. The Levenshtein distance between string A and string B is defined as the sum of the costs of the optimal sequence of edit operations to transform string A into string B (or vice versa). We can see an example for this in Figure 2.2, where “WORD” is transformed into “CAR”, requiring three operations, therefore these words have a Levenshtein distance of 3.

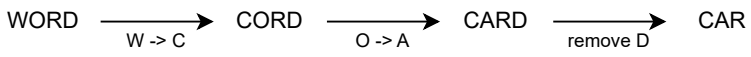


Figure 2.2: Example of the Levenshtein Distance, transforming "WORD" into "CAR" in three steps.

2.4.3 Damerau-Levenshtein Distance

The *Damerau-Levenshtein distance* is, as the name suggests, an extension to the Levenshtein distance detailed in Section 2.4.2. The Damerau-Levenshtein distance extends the regular Levenshtein distance by another edit operation:

- The transposition of two adjacent characters

This new operation also has the same cost as the original operations. Otherwise, the Damerau-Levenshtein is unchanged from the regular Levenshtein distance. This is demonstrated in Figure 2.3, where “WORD” is transformed into “ROD”, requiring only two operations this time, and therefore these words have a Damerau-Levenshtein distance of 2.

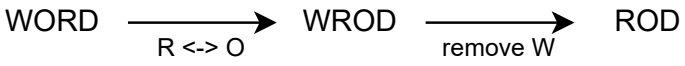


Figure 2.3: Example of the Damerau-Levenshtein distance, transforming "WORD" into "ROD" in two steps.

It is worth noting that, while the examples for the Levenshtein distance and Damerau-Levenshtein distance have been presented on single words, these distance metrics are also applicable to entire sentences.

2.4.4 Bag of Words

The *Bag of Words* model can be used to estimate the similarity of two strings of text containing natural language. The Bag of Words model falls into the category of *token-based distance functions*, since it is not based on an edit-distance but instead works by breaking up the text into individual words. The frequencies of each word across a string of text is counted and stored inside an unordered set, called a *bag*.

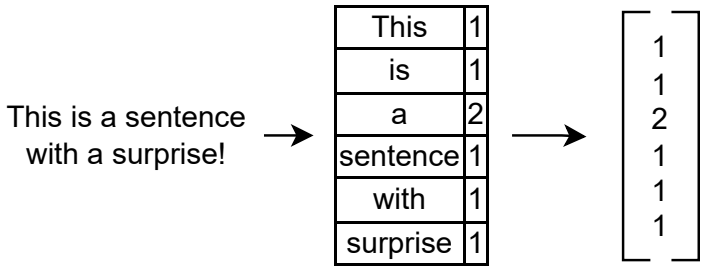


Figure 2.4: Example of the Bag of Words model, encoding a sentence to a 1-dimensional word frequency vector.

This bag can then be represented as a 1-dimensional vector, as seen in Figure 2.4, which can be compared using a vector similarity metric, most commonly the *cosine similarity*. The cosine similarity quantifies the similarity between two vectors by computing the cosine of the angle between two vectors. The angle is often used as a measure of divergence between the vectors, and the cosine is used since it has the helpful property of computing to 1.0 for identical vectors and 0.0 for orthogonal vectors [30].

2.5 Natural Language Processing

Natural Language Processing is the "area of research and application that explores how computers can be used to understand [...] natural language text or speech [...]" [9]. For this thesis, we will specifically focus on the early stages of *Natural Language Understanding*, namely *Tokenization*, explained in Section 2.5.1, and *Lemmatization*, explained in Section 2.5.2. Later steps, such as syntactic or semantic analysis, which are required for advanced applications of natural language processing, such as text generation, go beyond the scope of this thesis.

2.5.1 Tokenization of Natural Language

Tokenization of natural text is usually one of the earliest steps in natural language processing. During the *tokenization* of natural language, a longer stream of characters is subdivided into individual tokens, usually representing single words [1], similar to how tokenization for source code transforms the code into smaller, individual tokens.

During, or even before this step, typesetting dimensions, such as font size and font style, are usually removed or ignored, since they carry little meaning to the actual content of the text. This typesetting information may also be kept as *markup* information, which may be used in later stages of tokenization or natural language processing, for example to combine hyphenated words generated by line breaks, or to later detect multiple tokens belonging together, such as proper names, which may consist of multiple words [8]. The result of the tokenization is usually a stream of tokens, where each token represents one word without any punctuation, white spaces or line breaks.

In some cases, *multi-word tokenization* may also be favorable, where multiple words are joint into a single token, since individual words are joint into a habitual word combination, like noun phrases (such as *hot dog*) or verb-object constructions (such as *give up*) [16].

2.5.2 Lemmatization

Lemmatization is a technique used to improve the finding of similar or identical words to user input in natural language processing applications. Lemmatization uses vocabulary and morphological analysis of a word, and tries to return words to their dictionary form [2]. Since lemmatization returns words in their dictionary form, it can also be used to find similar words, so that for example a search for the word "warm" may also match the word "hot".

One alternative technique to lemmatization is *stemming*, which removes word suffixes in an attempt to transform words into their grammatical base [2].

These two methods may also be used simultaneously, stemming the raw words first, and afterward applying lemmatization to them. However, in this thesis, we will only use lemmatization as a first step of improving natural language processing.

3 Concept

In the past, token-based source code plagiarism detection systems have been removing textual content like variable names, documentation, and comments from the final token structure used for similarity computation. However, in recent developments, we have noticed that some likely plagiarized source code documents do contain similar comments, which may aid the plagiarism detection. Therefore, we now want to reintroduce textual content into the plagiarism detection, focusing on comments in this instance. To achieve this, we propose a three-step pipeline, consisting of comment extraction, comment matching, and result merging, with an optional step of natural language preprocessing. We will refer to this entire process described in this chapter as the *comment processing pipeline*. To not influence the results of the plagiarism detection on the source code itself; this pipeline will run independent of the rest of the plagiarism detection, and only merge the results at the end in a way that ensures the similarity scores will not decrease as a result of comment processing.

We will sequentially detail the individual steps of the pipeline in this chapter, starting with the comment extraction in Section 3.1. Afterward, the optional step of preprocessing comments using natural language processing techniques is introduced in Section 3.2. Following the extraction and optional preprocessing, we will explain the different methods of matching similar comments in Section 3.3. Lastly, we need to merge the results from the comment processing into the source code plagiarism detection, which is detailed in Section 3.4.

3.1 Extraction of Source Code Comments

To find similarities between textual constructs of source code, we first need to extract them from the source code documents. There are many different forms of textual content, such as variable names, class and function identifiers, textual input or output constants, and also source code comments.

In this thesis, we will only consider at source code comments, either in the form of documentation comments, such as JavaDoc for Java code, or inline comments within the code. Comments are better suited for this approach than variable names, since variable names often times consist of a single word or an abbreviation, which do not contain a lot of information. Comments, on the other hand, usually contain short sentences or bullet points, and are less likely to be similar to other submissions comments by pure accident, because of their larger size. Therefore, matching comments can be a strong indicator of plagiarism.

Instead of extracting the comments into the source code token sequence and creating a shared token sequence, we will extract the comments separately from the code, since otherwise existing matches in the source code could be easily broken up by simply adding lots of possibly unnecessary comments inside the reused source code. Since these comments do not alter the functionality of the code, the resulting program still works the same, but may result in a different shared token sequence for plagiarism detection systems.

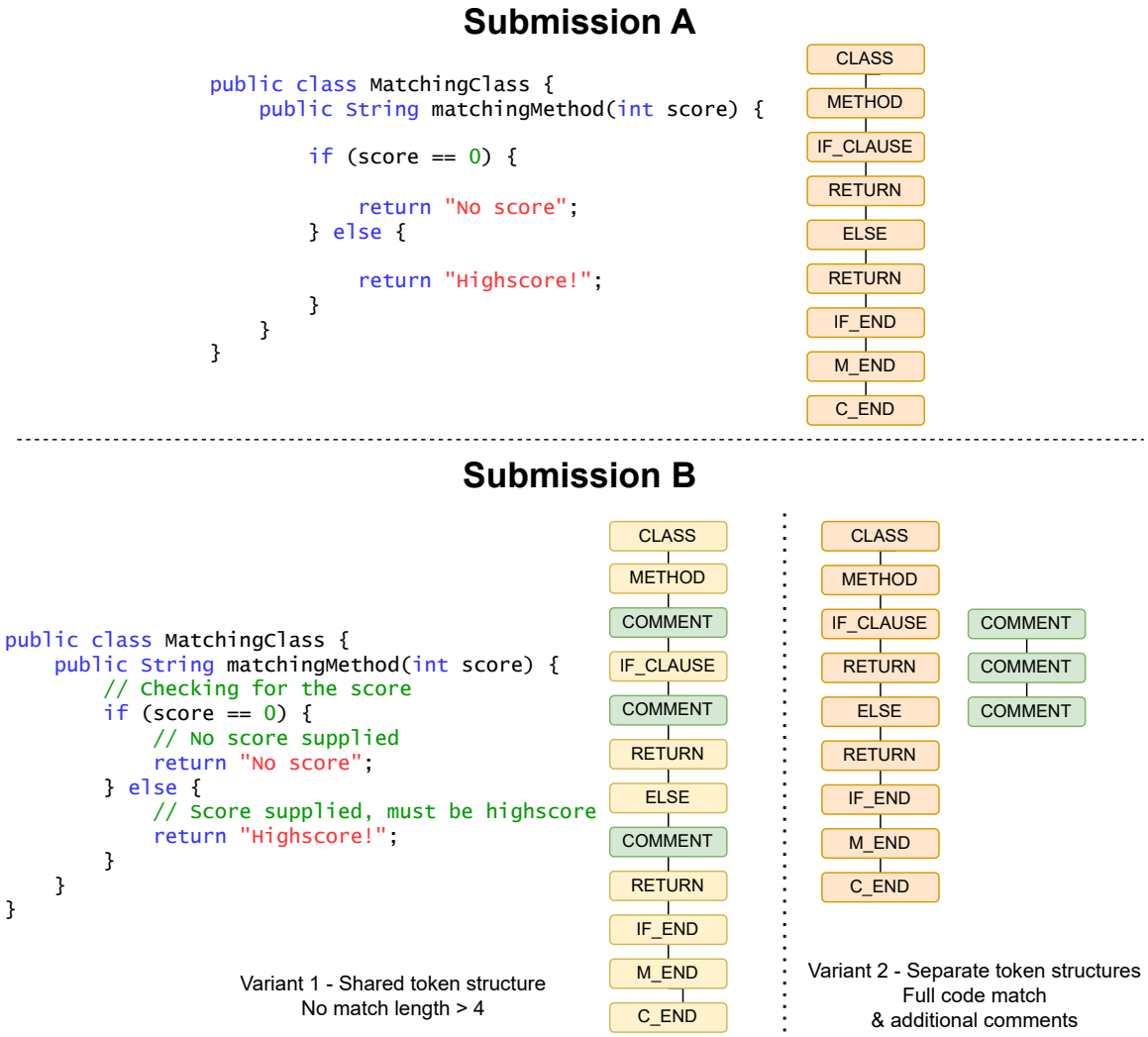


Figure 3.1: Example for breaking up matches with comments. In variant 1, the long match is broken up by comments, in variant 2 it is still present.

In Figure 3.1 we can see two submissions with identical source code; Submission B however features additional source code comments. If these comments were to be stored in the same token structure as the source code comments, like in variant 1, we would have no large match between the token structures, since the comment tokens are placed within the larger match. This could lead to plagiarism detection systems no longer recognizing this as a

possible case of plagiarism, even though the code is identical. If the comments are stored separately from the source code tokens however, as in variant 2, we still keep the large match from having identical source code, but also have the comments in addition to the code, to match them separately. This would not impact the existing source code matching, but may add additional matches if there are matching comments inside the submissions.

The extraction of comments is programming language-dependent, since different programming languages use different characters to denote the start and end of comments in source code. Therefore, a custom parser or extractor has to be implemented for each programming language used by the submissions. The extraction of comments may be done in parallel to tokenizing the source code, or separately, since both require parsing the entire source code, but neither depend on another.

3.2 Applying Natural Language Processing (NLP) Techniques

Since comments are usually written in natural language, we can use techniques from natural language processing to preprocess them before handling them further. The most important technique used here is *tokenization*, which turns the raw comment into a stream of tokens, which can then be used for matching. This will also be used to *clean* the comments, removing a lot of the formatting, like unnecessary whitespace, or special characters, to normalize the input comments. To further improve the quality of the plagiarism detection, more advanced natural language processing techniques can be applied, to normalize the input comments further. This thesis focuses on *lemmatization*, which converts individual words into their base form. However, other NLP techniques could also be applied and may result in a better detection.

3.3 Analyzing Source Code Comments and Computing Similarity Scores

When all source code comments are extracted, they can be matched in multiple different ways to compute a similarity score between two submissions. We can categorize the matching algorithms into two categories: *token-based matching algorithms*, which find matches in comments based on the underlying token structure, and *text-metric-based matching algorithms*, which work on the textual comment strings and compare them pairwise across submissions. We will present three different matching algorithms below. Section 3.3.1 will focus on token-based matching algorithms, while Section 3.3.2 will focus on text-metric-based matching algorithms.

3.3.1 Token-based Matching Algorithms

As part of this thesis, we will only consider a single token-based matching algorithm, *greedy string tiling*, the usage of which is detailed further in Section 3.3.1.2, after mentioning the general application of these matching algorithms in Section 3.3.1.1.

3.3.1.1 Application

Since the token-based matching algorithms work on a token sequence, they can be immediately applied to the tokenized comments received from the NLP preprocessing. These algorithms find matching subsequences inside the token structure; therefore, any matching subsequence of a predefined minimum length can be seen as a matching comment fragment, and taken into the result.

3.3.1.2 Greedy String Tiling

Some state-of-the-art token-based plagiarism detection systems, such as *JPlag* [11], already use Greedy String Tiling with Running Karp-Rabin Matching [31] as a token matching algorithm to find common subsequences within the source code token sequences, as explained in Section 2.4.1. This approach has already proven effective for matching source code tokens. Since comments can be tokenized like source code, an attempt at using the same approach for comments will be made.

3.3.2 Text-metric-based Matching Algorithms

Like with token-based matching algorithms, we will first explain the application of text-metric-based matching algorithms in Section 3.3.2.1, before detailing the two text-metric-based matching algorithms used, being the *Damerau-Levenshtein distance* in Section 3.3.2.2 and the *Bag of Words model* in Section 3.3.2.3.

3.3.2.1 Application

Text-metric-based matching algorithms work on the textual content of a comment instead of the token structure. Therefore, the token structure built in the NLP preprocessing step has to be converted into text strings for the individual comments again. The NLP preprocessing step is still important nonetheless, since it also removes a lot of unnecessary formatting information, like whitespace or special characters, from the comments.

These text metrics do not return complete matches, but only give us a *distance* between two text strings. From the returned distance, a similarity score can be computed. Since these metrics only return distances on pairs of comments, we need to compare all comments pairwise across all submissions to find matches. Additionally, since we only get a similarity

score per comment pair, we need to define a threshold at which similar comments are considered a match. In order to not define an arbitrary percentage, at which comments are taken as a match, the first approach was to use the code similarity of the submissions the comments originated from as a threshold for the comments, for example if the source code had a similarity score of 40%, the comments from those submissions needed to have a similarity score of at least 40% to count as a match.

It is worth noting that because the comments are compared sequentially, it is possible that a comment A_1 is matched with comment B_1, even though comment B_1 might have a higher similarity score when compared to another comment of submission A. This is demonstrated in Figure 3.2. However, the actual similarity scores between the individual comments is irrelevant, as long as they are above the threshold, since they will be included at that point. The only difference would be how the comments are matched in the manual review, as the comment matches may not be perfect.

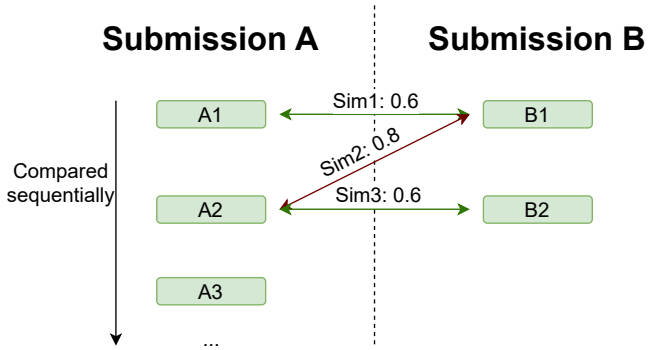


Figure 3.2: Example for nonoptimal string matching. Since the comments of submission A are matched sequentially, $A1$ will be matched with $B1$ if $Sim1$ is above the threshold, even though $A2$ and $B1$ have a higher similarity.

As we will see in the evaluation in Chapter 5, this threshold turned out to have major disadvantages for these matching algorithms, and therefore, we later added a lower bound of 50% to the threshold, resulting in a threshold of $max(50\%, similarityOfCode)$.

3.3.2.2 Damerau-Levenshtein distance

Being a simple string distance metric, the Damerau-Levenshtein distance can be used to compute the distance between two strings, in this case being the number of operations needed to transform one string into another one. To turn this distance into a similarity score, the distance will be divided by the length of the longer string, to get a difference score of the two strings, which we can turn into a similarity score by subtracting it from 1. So, given two strings with lengths $|S_1|$ and $|S_2|$, the similarity score $Sim\%$ is calculated from the Damerau-Levenshtein distance ΔDL like follows:

$$Sim\% = 1 - \frac{\Delta DL}{max(|S_1|, |S_2|)}$$

To find matching comments across two submissions, for each comment of submission A, the comment with the highest similarity score from submission B will be determined. If the similarity score is higher than the threshold, they will be considered a match. This repeats for each remaining comment of the submissions, until all remaining comments have been compared.

3.3.2.3 Bag of Words

Despite being a text metric, the Bag of Words approach does not require combining the comment tokens into a full string again, since it works by counting words. Therefore, the token sequence only needs to be split for each comment and a Bag of Words vector can be built for each comment based on the token sequence for that comment.

After building the Bag of Words-vector for each comment, they are again compared pairwise, similar to how they are compared with the Damerau-Levenshtein distance. In this instance, the two vectors Bag_1 and Bag_2 will be compared using the *cosine similarity*-formula:

$$cosine\ similarity = \frac{Bag_1 \cdot Bag_2}{||Bag_1|| * ||Bag_2||}$$

Since word frequencies can never become negative, the components of either vector are all non-negative, and the result is bounded between $[0, 1]$, which allows us to use this result as a similarity score. Again, to find the matching comments across two submissions, for each comment of submission A, the comment with the highest similarity score of submission B is determined, and if the similarity is higher than the threshold, it will be considered a match.

3.4 Merging Comment Matches into Result

After both the source code and the comments have been analyzed and their matches and similarity scores have been computed, the matches and matching tokens in the comments and the matches and matching tokens of the source code are merged to obtain a combined token sequence, from which a total similarity score can be computed, while non-matching comments are discarded. This is done to further ensure that the similarity between two submissions may only increase, but not decrease, since comments are easily changed, added, or removed without any effect on the functionality of the source code. Otherwise, the weight of existing source code matches could be diminished if the total token count is increased by adding a lot of comment tokens.

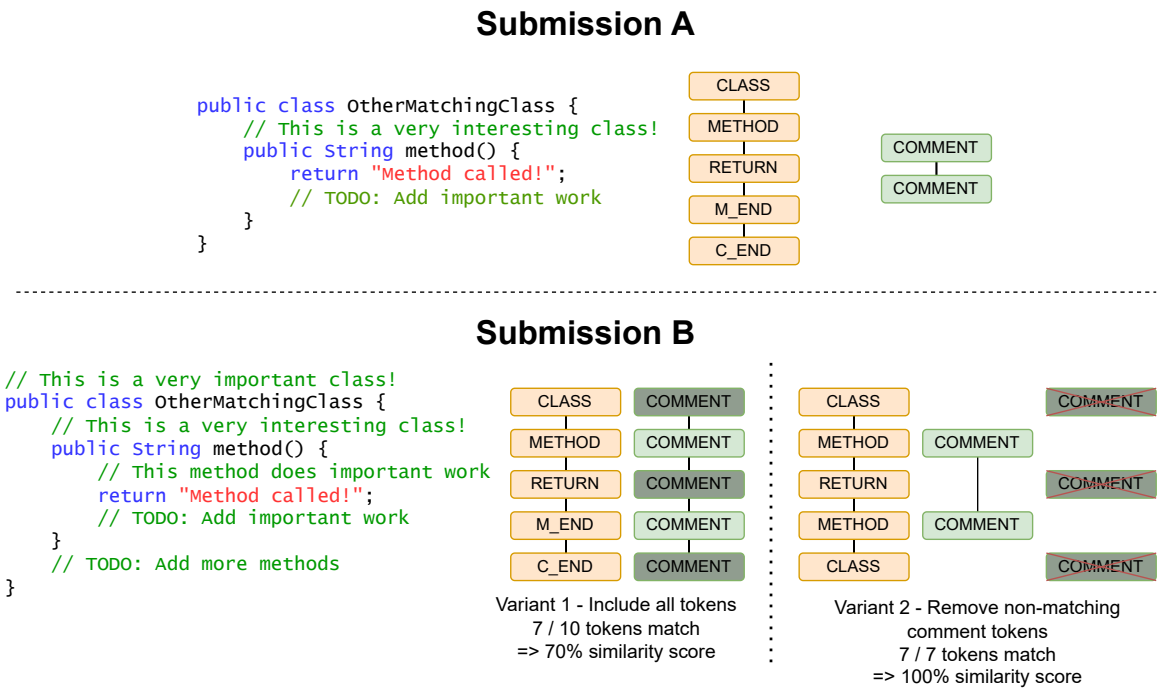


Figure 3.3: Example for diminishing weight of matches with comments. In variant 1, all tokens are included, which results in a lower similarity score than variant 2, where only matching tokens are included. As an example, the amount of matching tokens in relation to the bigger token count was used as a similarity metric.

This is demonstrated in Figure 3.3. We again have two submissions with identical code, and two identical comments. In addition to the two identical comments, Submission B contains three additional comments which are not present in Submission A. In this example, we will be using minimum similarity as our similarity metric, where the similarity score is the amount of matching tokens in relation to the bigger token count. If the three non-matching comments from submission B are included in the final token sequence, the similarity score of these two submissions would decrease to 70%, as demonstrated in variant 1, since only 7 out of 10 tokens match, but three of those tokens are unnecessarily added comments. This effect is even larger in the actual plagiarism detection, since comments do not consist of a single token, but rather one token for each word. If only matching comments are included, like in variant 2, 7 out of 7 tokens are matching, which would result in a 100% similarity score, which is more accurate, since the entire code, including some comments, are matching.

This is done on a per-comparison basis, which means that submissions may have a different total-token count based on the matching comments in this exact comparison.

4 Related Work

In recent years, a lot of research has been conducted in the area of source code plagiarism detection, with the general goal of improving plagiarism detection or defending against specific types of attacks. Sağlam et al. introduce token sequence normalization, which uses program dependence graphs to mitigate automated obfuscation attacks [27]. Maisch follows a similar goal, using graph-based structural normalization to defend against refactoring attacks [13]. A method called match merging is introduced by Niehues, where neighboring matches are merged to prevent obfuscation attacks, in which existing matches are broken up by reordering or inserting code [19]. In another paper, Sağlam et al. propose a similar technique called subsequence merging, in which sequences of tokens are iteratively merged together, to further increase resilience against obfuscation attacks [28].

Aside from other attacks and general improvements to source code plagiarism detection, research has also been done by other authors in the area of comment handling within plagiarism detection.

Various authors claim that comments may improve plagiarism detection, like Cosma and Joy: "Comments within source-code can be plagiarized and may contribute towards identifying source-code plagiarism" [5]. Zeidman also notes that comments, variable names, function names and other identifiers may be useful information for the detection of plagiarism, which are being thrown out by many approaches [32]. They may be used to pinpoint copied code, even in cases of intentional copying, and common misspellings between both submissions may be used as an additional indicator. Their proposed solution compares comments between submissions line by line, while converting sequences of whitespace into a single whitespace and ignoring case. They do not note, however, how exactly they are computing the similarity. They also value quality over efficiency, requiring "the program to run for hours, or in some cases days, to find plagiarism among large sets of files" [32], which makes it impractical for usage in an academic scenario with large courses including hundreds of submissions, which need to be checked for plagiarisms in a relatively short time frame.

In their work, Ramírez-de-la-Cruz et al. use comment similarity as one of five high-level features to detect cross-language plagiarism by computing n-grams across all comments and using a cosine similarity equation [23]. In another one of their works, Ramírez-de-la-Cruz et al. propose a simpler approach to plagiarism detection, in which one of their features is the cosine similarity of character n-grams between both source documents, with all whitespace, line breaks and reserved words of the programming languages removed [24]. This leaves only identifiers and comments in the document to compare, but in both of these approaches they are matching all documents together instead of matching individual comments to

others. They also focus a lot on cross-language plagiarism instead of plagiarism within one programming language.

Karnalim and Kurniawati use a different approach to handling comments within source code, by extracting features based on frequency of n-grams and vowels, consonants, digits, punctuation, and average word length, to compute a similarity of the comments between two submissions [12]. However, they do not consider the entirety of comments.

In contrast to this thesis, most of these works do not specifically investigate the effects of comments, but rather mention or implement one way of handling comments, instead of comparing multiple ways to analyze comments. Many of these works also revolve around their own tool, with little comparison to other established tools.

Outside of plagiarism detection, other authors have also developed methods to find similarities among source code comments or between source code comments and the source code itself.

In their work, McBurney and McMillan [15] investigate the textual similarity between source code summaries, sometimes in the form of comments, and the source code itself. For this purpose, they use several different metrics to calculate Short Text Semantic Similarity (STSS). These metrics include word overlap between text, either including or not including common stop words, STASIS, which itself is composed of word semantic similarity, sentence semantic similarity and word order similarity, and a metric called Lightweight Semantic Similarity, which calculates a text similarity for text without sentence structure. Some of these metrics are also based on a knowledge based organized as a hierarchical structure called WordNet [17], which organizes words based on meaning, grouping similar words more closely together than unrelated ones. This structure can then be used to compute a similarity between words, by comparing their distance and the depth of the lowest common ancestor. However, instead of finding cases of plagiarisms, they are investigating methods to determine how "accurate" written summaries of code are.

Folea and Slusanschi investigate in their study whether similarities between comments are an effective indicator of similar functionalities within software [7]. This goal is quite similar to finding plagiarisms within source code, as cases of plagiarism do share common functionality as well. For this, they are comparing comments using a variety of similarity functions, which includes string matching based on Levenshtein distances, as well as advanced NLP methods, such as *word2vec* embeddings. They are also not only matching individual singular comments, but also analyze up to N concatenated comments, since "[...] a logical comment may not spawn over one single line" [7]. They primarily focus on finding similar functionality by analyzing comments, while this thesis focusses on using comment similarity as a way to enhance token-based plagiarism detection, which already finds similar functionality by comparing source code token sequences.

5 Evaluation

To evaluate the proposed concepts, they have been implemented into the popular open-source token-based plagiarism detection system *JPlag* for the Java programming language. JPlag was chosen since it is one of the most popular open-source plagiarism detection systems [20], which is based on a state-of-the-art token-based approach. Java was chosen since it is a popular choice for the first programming language being taught in entry level computer science courses [6]. A variety of datasets from university courses were chosen for this evaluation, alongside several AI-generated datasets, to evaluate both the effect on human-written code and generated code.

For the implementation of this thesis, two primary stages were implemented into JPlag. First, all comments were extracted from the Java source files using a custom file parser. These extracted comments were then tokenized using JPlag's built-in `text` language module, which utilizes the `CoreNLP` library to tokenize natural language. In the second stage, the comments of all submission-pairs were compared using one of three approaches. One approach uses JPlags existing *greedy string tiling* implementation, while the other two approaches use custom implementations for the *Damerau-Levenshtein distance* or *Bag of Words model*. In one approach, the effects of further natural language preprocessing on a *Bag of Words* based approach were evaluated, by further utilizing the `CoreNLP` library to also apply lemmatization to the text.

To structure the evaluation, a Goal-Question-Metric Plan will be detailed in Section 5.1. Several different variants of the concept, which are explained in Section 5.2, were evaluated on multiple different datasets, which are listed in Section 5.3. The results of these tests can be found in Section 5.4, which will be discussed in Section 5.5. Possible threats to validity will be listed and discussed in Section 5.6.

5.1 GQM Plan

To structure the evaluation and quantify the results, we present the following Goal-Question-Metric plan, as introduced by Basili [3]:

G1: Enhance plagiarism detection by incorporating comments

- **Q1.1:** How are the similarity scores impacted by incorporating comments?
- **Q1.2:** How well does Greedy String Tiling perform for matching comments?

Q1.3: How well does the Damerau-Levenshtein distance perform for matching comments?

Q1.4: How well does Bag of Words perform for matching comments?

Q1.5: How are the similarity scores impacted by advanced NLP-preprocessing?

Due to the similarity of these questions, for all of them, the same metrics can be used:

M1.1: Similarity scores between plagiarism submission pairs and non-plagiarism submission pairs (including code and comments)

M1.2: Change of similarity scores without comment processing and individual comment processing techniques

G2: Maintaining performance of JPlag

Q2.1: How is the performance of JPlag impacted by incorporating comments?

M2.1.1: Runtime of JPlag

Goal **G1** is the primary goal of this thesis. Since these concepts are not a defense against a special attack, any kind of improvement compared to the baseline JPlag results without comment processing can be considered a success. Small, precise improvements may even be preferable over big general improvements, since those could help instructors while reviewing the results, by highlighting comments with a high similarity, to assist in finding possible cases of plagiarism.

Question **Q1.1** points at whether this goal was achieved at all, while questions **Q1.2** through **Q1.5** evaluates the individual approaches separately. Metrics **M1.1** and **M1.2** are basic metrics to quantify the results of the evaluation, to further help to answer the previous questions.

Goal **G2** is a secondary goal of this thesis. While the performance implications are not related to the effectiveness of the concept, and are highly dependent on the implementation itself, to be able to apply the concept in real applications, the performance of the program should not worsen too much. Question **Q2.1** deals with the feasibility of applying the concepts on a large scale, since a lot more comparisons are required when incorporating comments. As this is not a difficult question to answer, metric **M2.1.1** is a simple metric which makes it easy to answer this question at a glance.

5.2 Concept Variants

In total, the evaluation will be run on eight different variants of the implementation, each implementing a different variant of the concept:

CV1: Unmodified JPlag

This concept variant consists of the current, unmodified variant of JPlag, with no comment processing. This is to obtain a baseline for all datasets, which we can compare the other implementations to.

CV2: Greedy String Tiling-based comment processing

This concept variant compares all comments by utilizing JPlags existing *Greedy String Tiling* implementation, to obtain matching token subsequences, which are added to the result set and the similarity score.

The following three variants each split up into two identical variants again, with the only difference being the applied threshold. The *old threshold* is just using the similarity of the code as the threshold for the comments (for example if the code has a similarity score of 20%, the comments need a similarity score of at least 20% to count as a match), while the *new threshold* applies a lower bound of 50% to that, if the code similarity was below 50%, the comment similarity threshold would still be 50%.

CV3: Damerau-Levenshtein-based comment processing (old threshold)

CV4: Damerau-Levenshtein-based comment processing (new threshold)

These concept variants compare all comments by computing their *Damerau-Levenshtein distance* pairwise, comparing the distance to the longer comment length, and using that as a similarity metric between two comments. If the similarity is greater than the threshold, the comment was taken as a match.

CV5: Bag of Words-based comment processing (old threshold)

CV6: Bag of Words-based comment processing (new threshold)

These concept variants compare all comments by computing their *Bag of Words* in a vector space and comparing them using the cosine-similarity formula. If the similarity score resulting from the cosine similarity is greater than the threshold, the comment was taken as a match.

CV7: Bag of Words-based comment processing with advanced natural language preprocessing (old threshold)

CV8: Bag of Words-based comment processing with advanced natural language preprocessing (new threshold)

These concept variants are a modification to the previous variant, where lemmatization was applied to the comments before calculating the *Bag of Words*.

5.3 Datasets

To evaluate the proposed concepts in a wide variety of scenarios, eight datasets were selected. Four of these are in-house built datasets containing student submissions from our first-year programming lecture, each containing a different exercise. While public datasets would be preferable to ensure that the results are reproducible and comparable across studies, however, they are mostly either too small, poorly labelled, or, in the case of this thesis, do not contain enough comments for this approach to be viable. Therefore, for the majority of the evaluation, internal datasets were used. Two of those four datasets contain the submissions of final exercises which constitute the exam in this course, while the other two are submissions from late exercise sheets, where code documentation became a requirement. These datasets were already previously analyzed with an earlier version of JPlag and the results were manually reviewed by experienced instructors, to identify cases of plagiarism. This classification will also be used in this evaluation, dividing all comparisons into the three classes *Unlikely plagiarism*, *Possibly plagiarism* and *Cases of plagiarism*. It is worth noting that the `alphazeta` dataset only contained one instance of plagiarism, while the `trafficsimulation` dataset did not contain any cases of plagiarism. While this labelling is likely not perfect, it is good enough to be used in this evaluation, as it should provide a general overview of the effects of the concept variants on the three classes.

To protect the privacy of the students submitting for those exercises, the submissions have been anonymized before using them for this evaluation, and all personal information, such as names of the submitting student or other identification tags have been removed.

Name	Exercise-Sheet	Exercise summary	Number of submissions	Average submission size
`tictactoe`	Sheet 3 of 5	Simulator for a tic-tac-toe game, including a computer-opponent.	738	217 lines
`perseverance`	Sheet 4 of 5	Simulator a mars-rover on a 2d grid, including a pathfinding algorithm.	562	359 lines
`alphazeta`	Exam	Turn-based strategy game for 2 players on a 2d grid.	155	1543 lines
`trafficsimulation`	Exam	Simulator for road networks.	436	762 lines

Table 5.1: Human-Written Datasets.

The other four datasets consist of programs generated by Large Language Models. Different models with different temperature values (quantifying the *creativity* of the model) were prompted to solve four different programming exercises, and the results of each exercise were aggregated to build one dataset. These exercises were taken from both exercise sheets made for students of our local first-year programming lecture, and the PROGpedia collection

[21]. These datasets will be used to evaluate the effectiveness of these concepts on generated code, in similar fashion to how a student might employ Large Language Models to assist or simply solve entire exercises. The generated submissions were provided by Maisch et al., who generated them for their research on plagiarism detection systems [14], and contained about 500 submissions per AI model and exercise. Since excessively large datasets become very hard to use, because of increased runtimes and resource requirements, 200 submissions were randomly selected from each model for each exercise, to build a 600 submission dataset for each exercise.

Name	Exercise summary	Models used	Number of submissions	Average submission size
`cs`	Booking system for a carsharing provider.	GPT-4.1, GPT-4o, Deepseek v3	600	371 lines
`pp19`	Finding clusters in a graph. [21]	GPT-4.1, GPT-4o, Deepseek v3	600	57 lines
`pp56`	Optimizing binary matching in a graph. [21]	GPT-4.1, GPT-4o, Deepseek v3	600	69 lines
`ttt`	Simulator for a tic-tac-toe game, including a computer-opponent. [27]	GPT-4.1, GPT-4o, Deepseek v3	600	156 lines

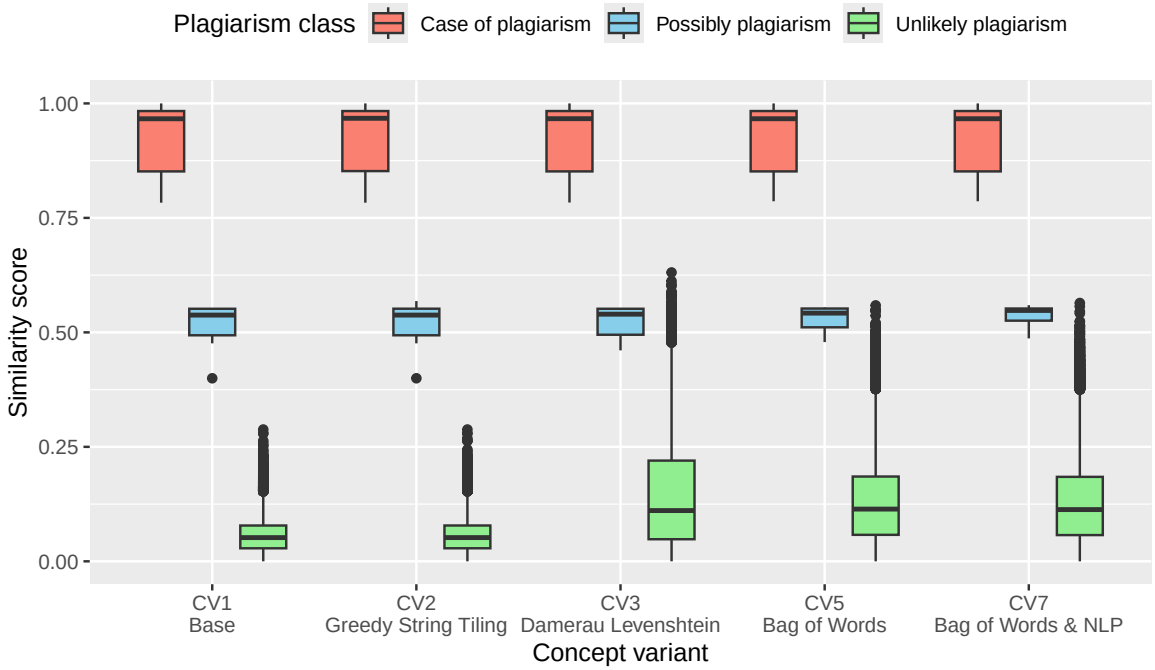
Table 5.2: AI-Generated Datasets.

5.4 Results

We will consider the two groups of datasets individually, starting with the human written datasets in Section 5.4.1 and following up with the generated datasets in Section 5.4.2. Afterward, we will shortly investigate the performance implications in Section 5.4.3.

5.4.1 Human-Written Datasets

To get a basic overview of the individual concept variants on the datasets, we use metric M1.1 (similarity score), as seen for the `perseverance` dataset in Figure 5.1.



Concept Variant	Unlikely plagiarism IQR	Unlikely plagiarism Median	Possibly plagiarism IQR	Possibly plagiarism Median	Case of plagiarism IQR	Case of plagiarism Median
CV1	0.0497	0.0518	0.0579	0.5378	0.1318	0.9664
CV2	0.0498	0.0518	0.0580	0.5378	0.1311	0.9675
CV3	0.1717	0.1108	0.0568	0.5398	0.1318	0.9667
CV5	0.1273	0.1141	0.0410	0.5421	0.1318	0.9666
CV7	0.1271	0.1130	0.0266	0.5478	0.1318	0.9666

Figure 5.1: Similarity scores of CV1, CV2, CV3, CV5, CV7 for dataset `perseverance`.

It is immediately apparent that the interquartile range of the *Unlikely plagiarism* class for CV3, CV5 and CV7 has significantly increased, and the lower quartile has climbed, while the classes of *Possibly plagiarism* and *Cases of plagiarism* have barely changed along all concept variants. This is not optimal, as this indicates that while actual cases of plagiarism, or even likely ones are barely changing, and comparisons that previously had a low similarity might have overly increased, introducing a lot of noise. Ideally, we would see the reverse result, where comparisons from the *Cases of plagiarism* and *Possibly plagiarism* class would increase in similarity, and likely reduce the interquartile range, as it gets closer to 100%.

To investigate this further, we can look at how each of the comparisons changed when processing comments compared to the base results without comment processing.

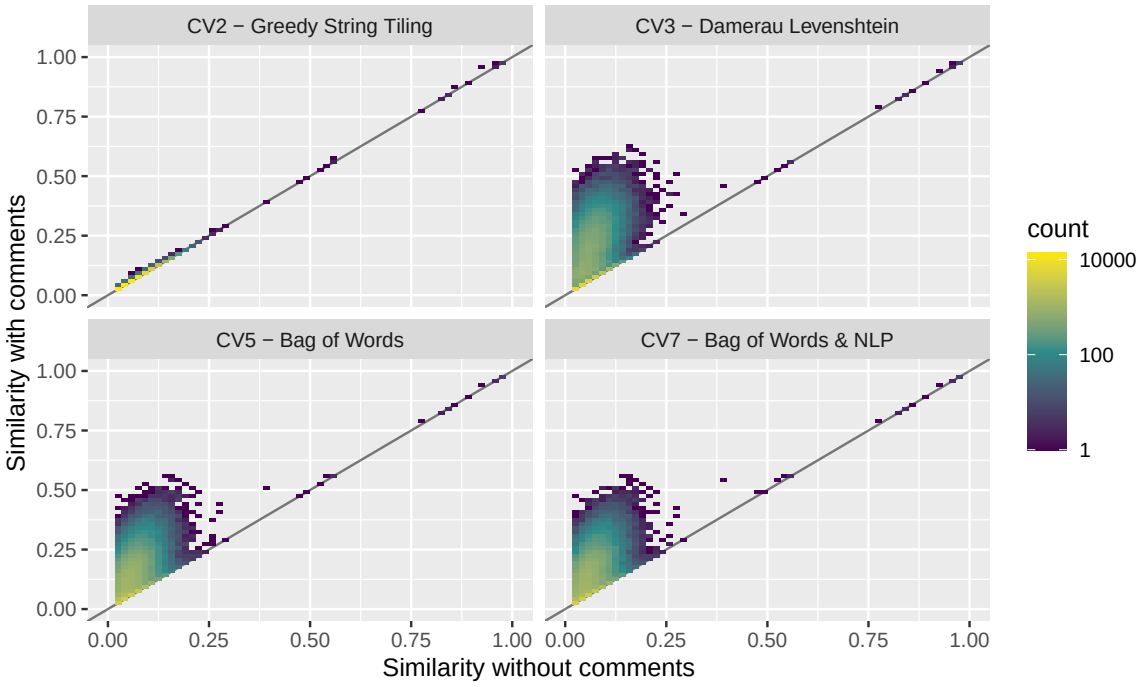


Figure 5.2: Change of similarity scores for CV2, CV3, CV5, CV7 of dataset `perseverance`. The histograms have 60 bins, with the color of each bin representing the amount of comparisons in it. Note that the color scale is logarithmic instead of linear.

We can see this highlighted by looking at Figure 5.2, which plots metric M1.2 (change of similarity score) as a histogram. Especially in concept variant 3, 5 and 7, comparisons that had low similarity without comment comparison increase very greatly, while higher end similarities do not increase as much. This introduces a lot of noise in the dataset, as an increase in low similarity comparisons is not necessarily bad in itself, but without an equivalent increase in higher similarity comparisons is not the desired result. This issue is not confined to this dataset only, but can also be found in other datasets, as seen in Figure 5.3 and Figure 5.4.

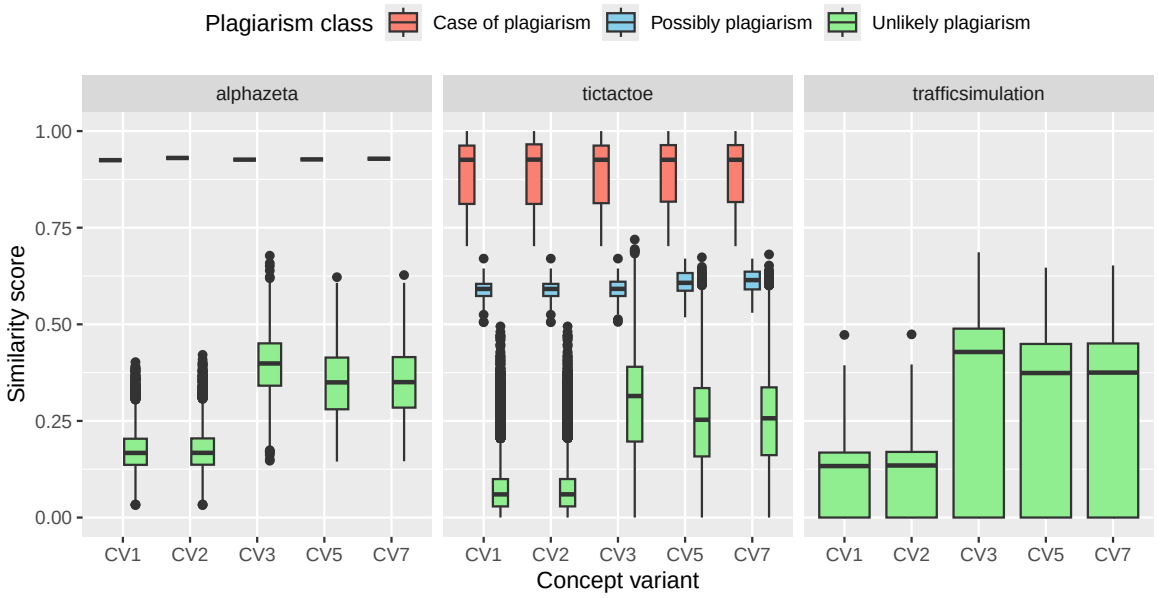


Figure 5.3: Similarity scores for CV1, CV2, CV3, CV5, CV7 of datasets `tictactoe`, `alphazeta`, `trafficsimulation`.

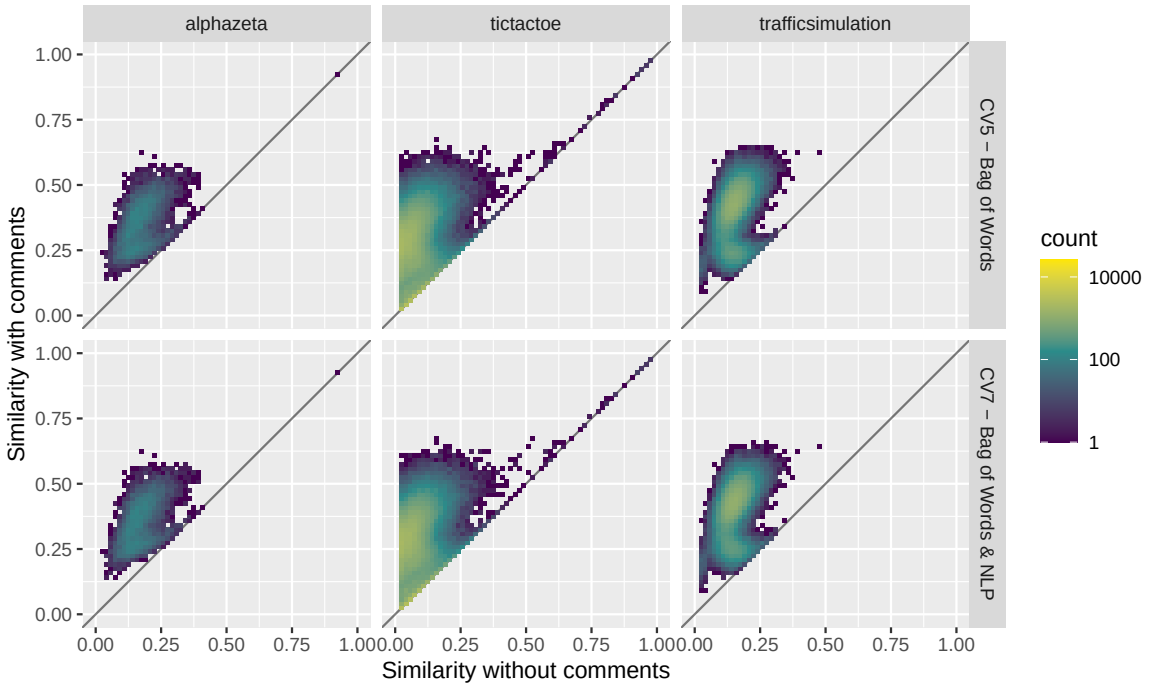
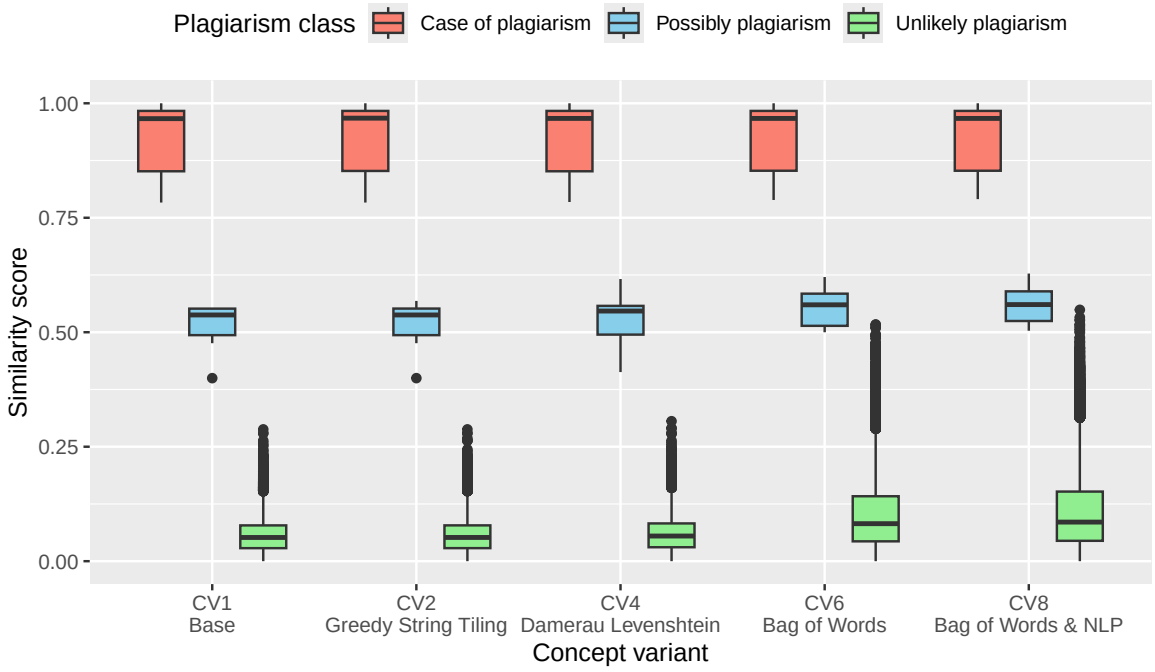


Figure 5.4: Change of similarity scores for CV5 & CV7 of datasets `tictactoe`, `alphazeta`, `trafficsimulation`. The histograms have 60 bins, with the color of each bin representing the amount of comparisons in it. Note that the color scale is logarithmic instead of linear.

In Figure 5.3, especially in the box plot of the `tictactoe` dataset we can see the introduced noise, as the interquartile ranges of CV1 and CV2 are quite low, with a lot of outliers above, while the interquartile ranges of CV3, CV5 and CV7 are quite high, with much fewer outliers, as their medians are a lot higher. The source of this issue can be traced back to the old threshold. Due to the lower end comparisons having very low similarity scores, in some cases even 0%, the comment similarity threshold for those comparisons would also become 0%. Since a single shared letter is enough to result in a Damerau-Levenshtein distance greater than 0, and a single shared word is enough to result in a Bag-of-Words similarity greater than 0, completely unrelated comments were detected as matches. Following this observation, the lower bound and new threshold for CV4, CV6 and CV8 were introduced.



Concept Variant	Unlikely plagiarism IQR	Unlikely plagiarism Median	Possibly plagiarism IQR	Possibly plagiarism Median	Case of plagiarism IQR	Case of plagiarism Median
CV1	0.0497	0.0518	0.0579	0.5378	0.1318	0.9664
CV2	0.0498	0.0518	0.0580	0.5378	0.1311	0.9675
CV4	0.0521	0.0550	0.0628	0.5463	0.1318	0.9669
CV6	0.0985	0.0820	0.0703	0.5599	0.1307	0.9669
CV8	0.1075	0.0854	0.0648	0.5604	0.1307	0.9669

Figure 5.5: Similarity scores for CV1, CV2, CV4, CV6, CV8 of dataset `perseverance`.

As we can see in Figure 5.5, with the new threshold, the interquartile range of CV4, CV6 and CV8 in the *Unlikely plagiarism* class are a lot smaller compared to Figure 5.1. Here, we can see that CV1, CV2 and CV4 seem to all perform similar to each other, with CV4 having the greatest range of values. The interquartile ranges of CV6 and CV8 in the *Unlikely*

plagiarism class are still slightly bigger than of the other three variants, and they also have the biggest span of outliers. This, again, can also be observed over multiple datasets, as seen in Figure 5.6.

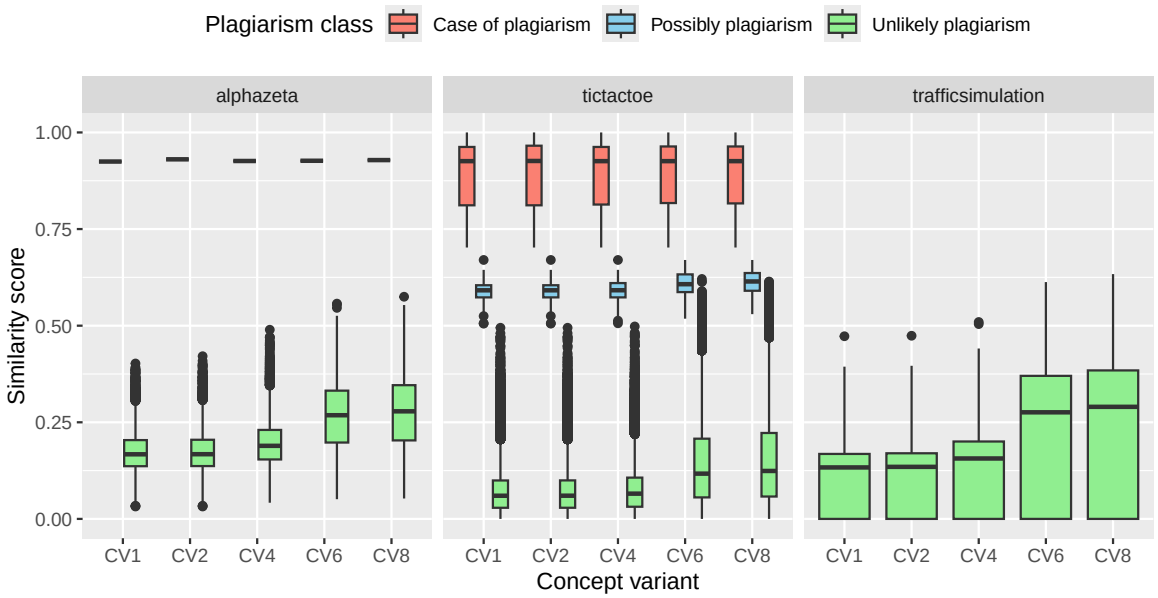


Figure 5.6: Similarity scores for CV1, CV2, CV4, CV6, CV8 of datasets `tictactoe`, `alphazeta`, `trafficsimulation`.

As before, the outliers are the most visible in the `tictactoe` dataset. However, in that dataset, because the exercise which the dataset is based on only allowed for a smaller solution space, the amount of outliers is already high for CV1 as well. The amount of outliers, especially in the *Unlikely plagiarism* class, are similar between CV1 and CV2, with CV4 only having a small variation from them, and CV6 and CV8 having the biggest jump in outliers. This big span of outliers can also be observed if we look at metric M1.2 (change of similarity scores) as a box plot, as seen in Figure 5.7.

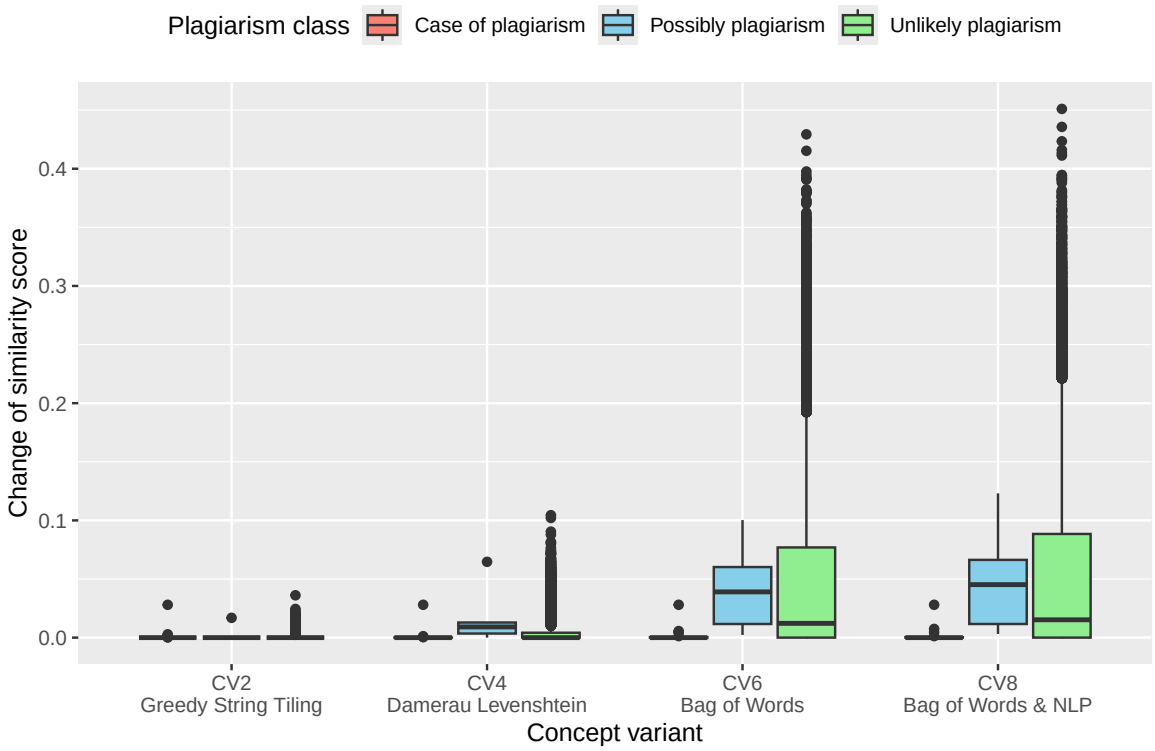


Figure 5.7: Increase of similarity scores for CV2, CV4, CV6, CV8 of dataset `perseverance`.

We can see that comparisons in the *Unlikely plagiarism* class make the biggest jumps in CV6 and CV8, some raising up by as much as 45 percentage points. The other classes make much smaller jumps, especially in CV2 and CV4.

Using these results, we can answer the first questions from the GQM Plan defined in Section 5.1:

Q1.1: How are the similarity scores impacted by incorporating comments?
A1.1: Overall, the similarity scores do increase when incorporating comments. The amount by which they increase does depend on the concept variant used, as well as the dataset. In many datasets, the comparisons that had a lower similarity without comment processing increased the most, which is not directly in line with the primary goal of this thesis.

Q1.2: How well does Greedy String Tiling perform for matching comments?
A1.2: Greedy String Tiling increased the similarity scores the least on average. The increase in the *Case of plagiarism* class was comparable to other algorithms, but the increase in the *Unlikely plagiarism* class is the smallest compared to other algorithms.

Q1.3: How well does the Damerau-Levenshtein distance perform for matching comments?

A1.3: With the old threshold, the Damerau-Levenshtein distance introduced the most noise from any algorithm, with the biggest increase in the *Unlikely plagiarism* class. With the new threshold, however, the Damerau-Levenshtein distance produces results comparable to the Greedy String Tiling approach, while the average increase in the *Unlikely plagiarism* class is still a bit higher, than when using Greedy String Tiling.

Q1.4: How well does Bag of Words perform for matching comments?

A1.4: The Bag of Words variants introduced the most amount of noise in the *Unlikely plagiarism* class with the new threshold and the second most with the old threshold. The increase in the *Case of plagiarism* class is still similar to both other approaches, but the increase in the *Possibly plagiarism* class is also bigger than with the other approaches, possibly hinting at missed cases of plagiarism during labelling.

Q1.5: How are the similarity scores impacted by advanced NLP-preprocessing?

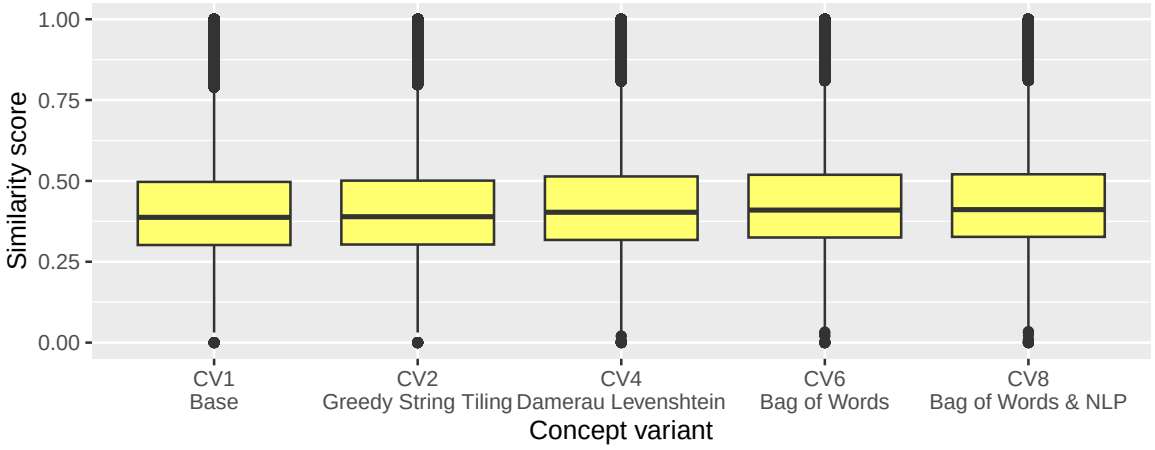
A1.5: The NLP-preprocessing applied in CV7 and CV8 did increase the similarity scores further. The increase is not large on average, but it does seem like even the small NLP-preprocessing introduced in CV7 and CV8 already impacted the results. It is possible that the similarity scores would increase even higher, if even more NLP-preprocessing techniques are introduced. However, NLP-preprocessing was only used with a Bag of Words-based approach, it is possible that it would work better or worse when combined with other algorithms.

Judging by these observations, it appears as if CV6 and CV8, both using a Bag of Words-based approach, are not suitable for comparing comments between submissions, since they greatly increase the number of outliers for submissions that had a lower similarity score beforehand and therefore introduce a lot of noise into the plagiarism detection. However, in this thesis, we have only looked at enhanced natural language preprocessing in combination with a Bad of Words-based approach. The effects of enhanced natural language preprocessing on other techniques, as well as more preprocessing steps, are out of scope for this thesis, but may be subject to future research.

Both Greedy String Tiling-based and Damerau-Levenshtein Distance-based approaches seem suitable for comment processing with Damerau-Levenshtein Distance-based approaches resulting in higher average changes of similarity in the *Unlikely plagiarism* class, but both approaches seem to be within acceptable boundaries. Bag of Words-based approaches should not be immediately discarded however, since further investigation is required to see whether the increase in similarity scores of comparisons that had a low similarity score without comments is due to the matching of unrelated comments or if the increase is actually due to similar comments across many submissions.

5.4.2 AI-generated Datasets

We also want to investigate how incorporation of comments impacts the detection of submissions fully generated by generative artificial intelligence. The metrics themselves are identical to the ones used in Section 5.4.1, but the individual submissions are not grouped into further subclasses, since no such distinction exists for these datasets. Figure 5.8 plots metric M1.1 (similarity scores) as a box plot for the `ttt` dataset.



	CV1	CV2	CV4	CV6	CV8
IQR	0.1952	0.1976	0.1963	0.1940	0.1935
Median	0.3873	0.3893	0.4031	0.4098	0.4112

Figure 5.8: Similarity scores of CV1, CV2, CV4, CV6, CV8 for AI-generated dataset `ttt`.

As we can see in Figure 5.8, the graphs barely change between concept variants for this dataset. This observation is not confined to this dataset itself, but persists across all generated datasets, as seen in Figure 5.9.

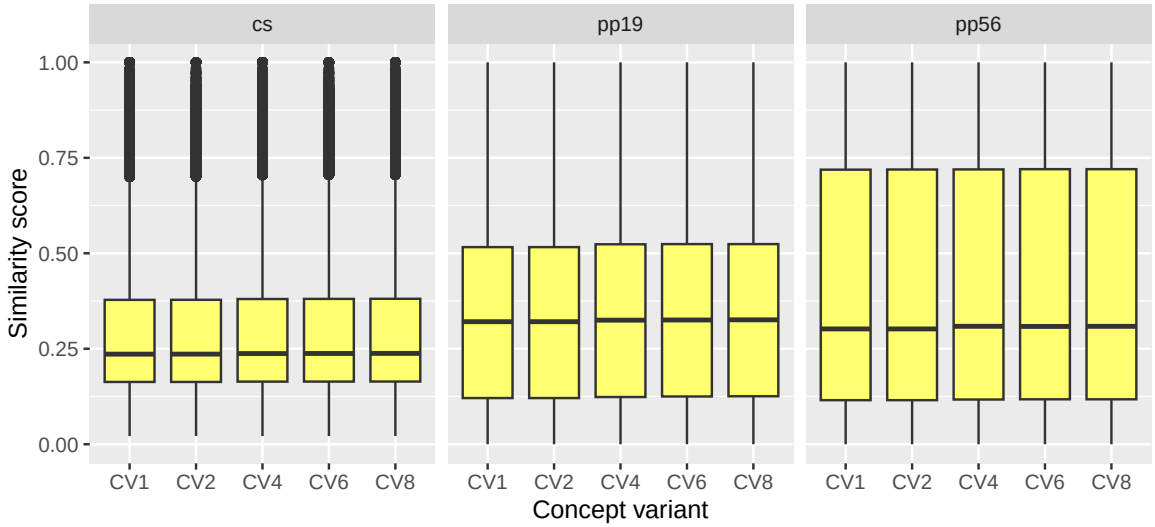


Figure 5.9: Similarity scores of CV1, CV2, CV4, CV6, CV8 for AI-generated datasets `cs`, `pp19`, `pp56`.

This is due to the incorporation of comments only changing the similarities of a small subset of submissions by a small amount, while most of the similarities stay the same. This is more visible if we look at metric M1.2 (average change of similarities) across each concept variant, as seen as a box plot in Figure 5.10.

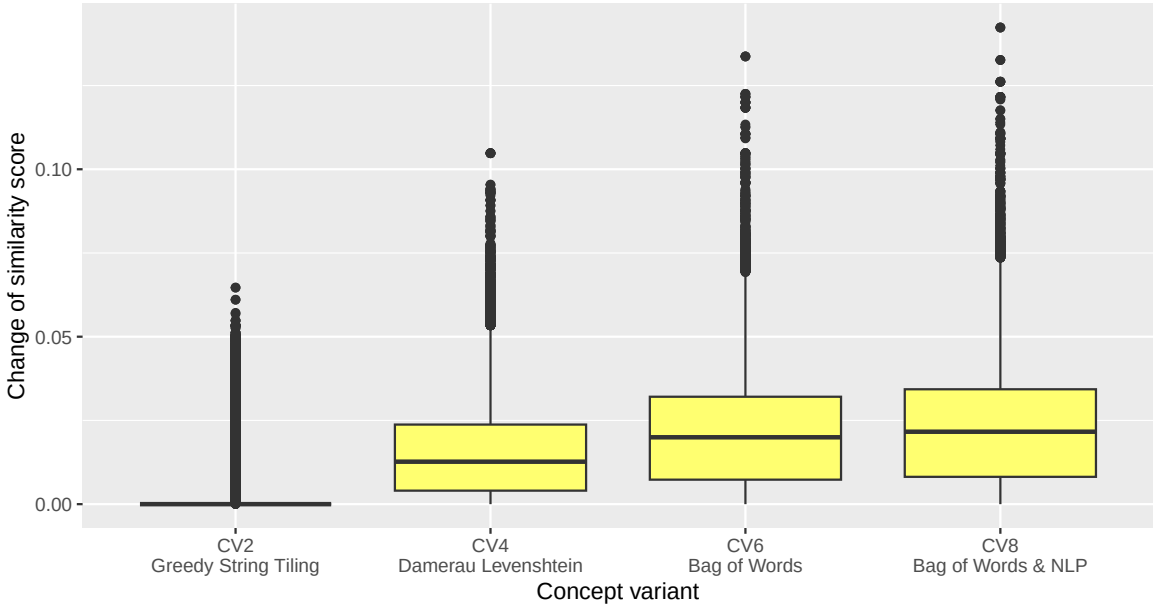


Figure 5.10: Increase of similarity scores of CV2, CV4, CV6, CV8 of AI-generated dataset `ttt`.

However, even though the increase in the similarity score is quite small on average, being below three percentage points, it is existent, which does indicate that the concept did find similar comments between fully generated submissions. This does indicate that even though the submissions for these datasets were generated with entirely separated prompts across multiple LLMs, the results did show similar comments, which were found and matched by the implementations.

When selecting a concept variant with the goal of increasing the similarities of artificially generated submissions, the biggest average increase in similarity, done by CV8, might seem like the optimal choice. However, in Section 5.4.1, we observed that the CV6 and CV8 both introduced a lot of noise, raising the similarity of likely unrelated submissions by a substantial amount, which may be undesirable. Therefore, the next best choice, CV2 and CV4, also seem like the optimal choice here, with CV4 increasing the similarity scores of generated submissions more than CV2.

5.4.3 Performance

As detailed in Section 5.1 as a secondary goal G2, we want to investigate the performance and runtime of the implementation, to ensure it is practically applicable in real applications. For this, the entire runtime of JPlag using each concept variant across each dataset has been recorded five times and the average runtime was calculated. However, since CV3, CV5 and CV7 only differ from CV4, CV6 and CV8 in the selection of the threshold, which should not impact runtime, CV3, CV5 and CV7 have been ignored during performance analysis, and CV4, CV6 and CV8 have been used instead.

It should be noted that the runtime greatly depends on the number of submissions as well as the average submission size. The number of submissions is denoted by N, the average submission size by $avgSize$, in lines of code. For demonstrative purposes, three datasets have been selected with varying numbers and sizes of submissions. The runtimes for all datasets can be found in Table 5.3 at the end of the section. Figure 5.11 shows the average runtime of the concept variants for each of those datasets in seconds.

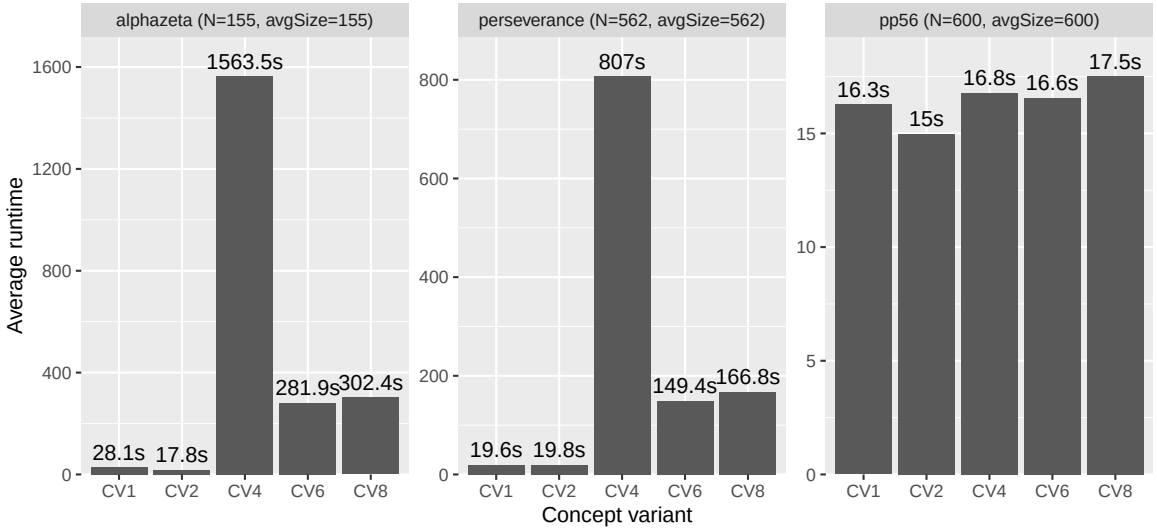


Figure 5.11: Average runtimes of CV1, CV2, CV4, CV6, CV8 on datasets `alphazeta`, `perseverance`, `pp56` in seconds.

We can immediately see that the runtime of CV4 increases dramatically with increasing submission size, going from 16 seconds for small submissions all the way to 25 minutes for larger submissions. If the submission sizes are relatively small and contain few comments, such as in dataset `pp56`, all concept variants perform similarly, some even better than the base variant, possibly due to small other optimizations in the code base. However, with increasing submission size, most concept variants naturally increase the runtime, due to more comparisons needing to be done. We can also see the proportional increase in runtime in Figure 5.12.

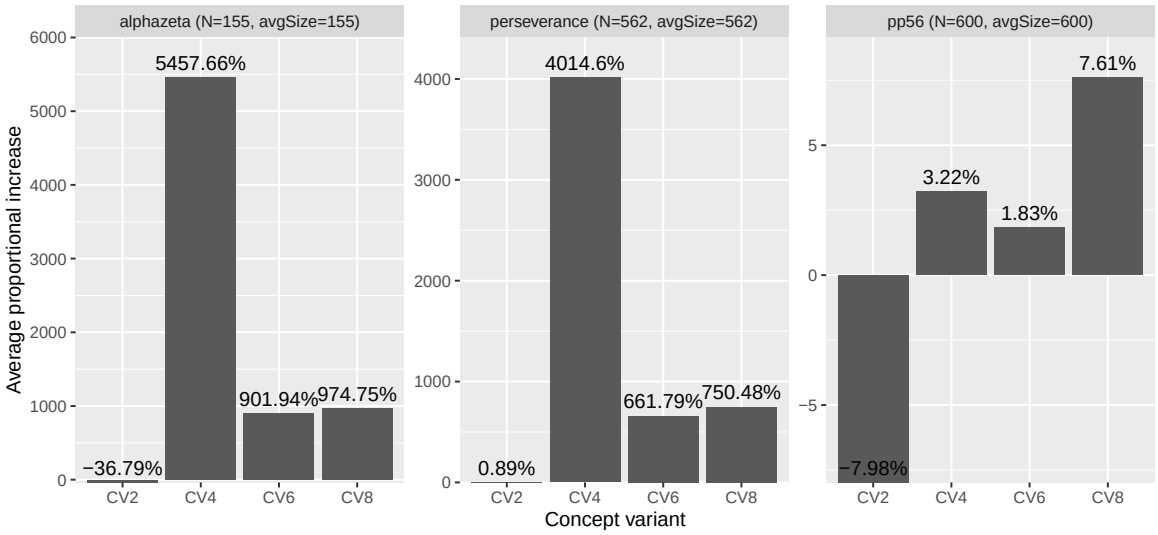


Figure 5.12: Proportional increase in runtimes of CV1, CV2, CV4, CV6, CV8 on datasets `alphazeta`, `perseverance`, `pp56`.

It is quite obvious that if runtime is a concern, CV4 is not realistically usable on large submissions, due to its over 5000% increase in runtime. However, this could also be due to an inefficient implementation of the proposed algorithms or other oversights. It is not impossible that the runtime of CV4 could be reduced a lot by improving the underlying implementation. From all the variants, CV2 seems to be running the most efficiently, possibly due to the large amounts of optimization already done in the code base and the greedy algorithm itself. The runtime of CV6 and CV8 increases by 600% or more as well, but that decrease in runtime may be acceptable, should the quality of the results improve enough.

tictactoe (N: 738, avgLen: 217)

Variant	Mean	Min	Q1	Median	Q3	Max	Sum
CV1	22.53	21.10	21.24	21.31	23.96	25.06	112.67
CV2	23.91	22.85	23.04	23.97	24.71	25.00	119.57
CV4	660.55	646.77	651.96	662.88	668.51	672.61	3302.73
CV6	141.62	138.33	139.69	140.75	143.68	145.65	708.10
CV8	162.14	152.23	156.12	160.01	167.09	174.18	486.42

ttt (N: 600, avgLen: 156)

Variant	Mean	Min	Q1	Median	Q3	Max	Sum
CV1	16.52	16.23	16.34	16.55	16.7""4	16.74	66.08
CV2	16.69	16.05	16.22	16.75	16.98	17.45	83.45
CV4	40.14	38.12	38.84	39.34	42.07	42.32	200.69
CV6	33.13	31.24	32.23	32.63	34.62	34.94	165.66
CV8	34.45	33.84	33.89	33.93	34.75	35.58	103.35

perseverance (N: 562, avgLen: 359)

Variant	Mean	Min	Q1	Median	Q3	Max	Sum
CV1	19.61	17.45	19.10	19.34	19.35	22.83	98.07
CV2	19.79	19.20	19.57	19.73	19.91	20.53	98.94
CV4	807.04	785.15	793.81	807.08	824.46	824.69	4035.19
CV6	149.42	142.01	150.17	151.22	151.25	152.44	747.09
CV8	166.81	157.45	159.93	162.41	171.50	180.58	500.44

pp19 (N: 600, avgLen: 57)

Variant	Mean	Min	Q1	Median	Q3	Max	Sum
CV1	16.00	15.51	15.89	16.14	16.14	16.33	80.01
CV2	15.27	14.34	14.40	14.43	15.29	17.89	76.35
CV4	18.37	17.95	18.06	18.22	18.40	19.21	91.84
CV6	17.63	17.16	17.34	17.54	17.71	18.40	88.15
CV8	19.53	18.37	18.76	19.14	20.11	21.08	58.59

alphazeta (N: 155, avgLen: 1543)

Variant	Mean	Min	Q1	Median	Q3	Max	Sum
CV1	28.13	24.42	27.89	28.42	29.48	30.45	140.66
CV2	17.78	16.94	17.19	17.38	18.00	19.40	88.91
CV4	1563.48	1541.21	1554.49	1560.97	1571.86	1588.88	7817.41
CV6	281.87	273.00	275.05	282.63	284.65	294.00	1409.33
CV8	302.35	290.48	294.94	299.40	308.28	317.17	907.05

pp56 (N: 600, avgLen: 69)

Variant	Mean	Min	Q1	Median	Q3	Max	Sum
CV1	16.27	15.79	15.91	16.44	16.58	16.64	81.36
CV2	14.97	14.26	14.72	14.81	14.94	16.14	74.87
CV4	16.80	16.04	16.12	16.85	17.12	17.85	83.98
CV6	16.57	15.39	16.20	16.58	16.60	18.08	82.85
CV8	17.51	17.16	17.18	17.20	17.69	18.17	52.53

trafficsimulation (N: 436, avgLen: 762)

Variant	Mean	Min	Q1	Median	Q3	Max	Sum
CV1	49.27	46.61	48.15	48.22	48.26	55.10	246.34
CV2	30.89	29.17	29.83	31.15	31.20	33.12	154.47
CV4	4844.11	4750.37	4789.09	4878.65	4883.18	4919.26	24220.55
CV6	636.19	597.17	598.68	636.37	661.48	687.24	3180.94
CV8	631.59	615.46	625.00	634.54	639.65	644.76	1894.76

cs (N: 600, avgLen: 371)

Variant	Mean	Min	Q1	Median	Q3	Max	Sum
CV1	19.08	18.60	18.81	18.81	19.45	19.73	95.40
CV2	23.75	23.05	23.63	23.71	24.00	24.36	118.75
CV4	50.32	49.22	49.28	50.02	50.05	53.04	251.61
CV6	39.66	38.34	39.75	39.83	39.88	40.51	198.31
CV8	40.43	39.40	39.62	39.85	40.94	42.03	121.28

Table 5.3: Runtime statistics for all datasets across all concept variants. *N* denotes the amount of submissions in each dataset, *avgLen* is the average length of each submission, in lines of code.

5.5 Discussion

As expected, the similarity scores were impacted by the incorporation of comments in the similarity score computation, with the different algorithms impacting the similarity scores with different intensities. This does indicate that the plagiarism detection may be improved by incorporating comments. With many algorithms, the biggest increase in similarity scores was found with comparisons that had a quite low similarity score without comments. Additionally, we did also see an increase in similarity scores when comparing submissions generated by different Large Language Models.

Increase in comparisons with low similarity scores. A lot of the increases in similarity scores appeared in comparisons that had a low similarity score without comment processing. This could be an indicator of unrelated comments being matched in those instances, but it could also be a legitimate increase, if those submissions share similar comments. More research into this area, possibly with other means of evaluation, is needed to find out if this increase in similarity score is desired or unhelpful.

Comparison of Concept Variants. All concept variants increased the similarity scores by a different amount. CV3, CV5 and CV7 were discarded quite early, since their usage of the old threshold resulted in the matching of completely unrelated comments, if the submissions had a low similarity score without comments already. This was due to a single character (in case of CV3) or a single shared word (in case of CV5 and CV7) sufficing to result in a similarity score greater than 0, which may be enough to overcome the threshold needed to include the comments in the result.

CV2, based on Greedy String Tiling, increased the similarity scores the least. This is mostly due to Greedy String Tiling finding common subsequences within the comment token sequence. However, if two words across two submissions are not spelt exactly the same, they will not match with this algorithm. This can lead to small word replacements or typing errors breaking up matches, if one word within an identical sentence is replaced or mistyped. Since the token order is also important for Greedy String Tiling, even swapping two words around is enough to trick the matching algorithm to no longer include said match. If looking to increase similarity scores the least overall, but highlight very similar comments across submissions, this concept variant is likely the best choice, since it increased the similarity scores the least on average, indicating that only very similar comments were seen as a match.

CV4, based on the Damerau-Levenshtein distance, matched slightly more comments than Greedy String Tiling. Overall, with the new threshold, the performance was comparable to Greedy String Tiling. However, we did see that the runtime of the Damerau-Levenshtein distance was by far the longest of any concept variant with longer submissions, due to the large amounts of comparisons needing to be done. This may be solvable with a more optimized implementation of the algorithm, but in the evaluated form, it did increase the runtime by up to 5000%.

CV6, based on the Bag of Words model, matched the most comments. Most of the increase in similarity scores were with submissions that had a low similarity score beforehand already. This may be due to a large amount of common filler words in comments, or keywords such as for documentation comments, which increased the similarity of comments. When looking to increase the similarity scores of submissions overall, this algorithm may be the optimal choice, since on average it increased the similarity scores the most.

CV8, which added natural language preprocessing to the Bag of Words model, did also increase the similarity scores further, but only very slightly. However, this is an indication that NLP preprocessing may be helpful in improving the comment processing even further, possibly even more if applied to another base algorithm.

Matching identical comments. Despite their differences, all the concept variants above are capable of finding identical comments across multiple submissions. This alone may already be beneficial during the manual review of the results, to identify cases of plagiarism. In those manual reviews, identical comments may be a helpful or deciding factor, as identical, large comments can be a strong indicator of plagiarism. The only exception to that could be documentation comments for common program functionality, like getter- and setter-methods, since those are often similar across different programs. However, those shared comment constructs are usually quite small and easy to spot during inspection. If matching identical comments is the primary goal, Greedy String Tiling is likely the best approach, since it otherwise skews the results the least, and only finds near identical comments with long sections of identical words.

Increasing similarity scores of generated submissions. It is also very worth noting that we did see an increase in similarity scores between fully generated submissions with all concept variants. This is the result of all algorithms being able to find similar comments in the generated submissions. This indicates that there are similar comments among the submissions, even though they were generated separately from each other, using different LLMs and different temperature values. This may improve the plagiarism detection in scenarios where LLMs are at least partially used within the submissions.

5.6 Threats to Validity

There are some threats, which may compromise the validity of the results of the evaluation. This section lists them, and detailing how they have been dealt with. Some of these threats may be inherent or even unavoidable to this type of research.

5.6.1 Threats to Internal Validity

Assumption: Existence of comments. The idea of increasing the similarity score by incorporating comments is based on the assumption that the source code which is being compared

contains comments. This assumption is valid, since it is common coding practice to comment the source code, especially in areas where the code might be difficult to understand otherwise. This assumption may not hold true in cases of exercises submitted by students, as especially beginners may not understand the importance of comments within source code. However, most programming courses do encourage or require commenting the source code at later stages during the courses.

Selection of human-made datasets. The internally provided human-made datasets were selected randomly across the students submissions from exercises and exams in recent years. However, only exercises and exams with a high submission count were considered, and earlier exercises were ignored as well, since comments within the source code only became required at a later stage in the course.

Selection of generated datasets. The generated datasets, `ttt`, `pp56`, `pp19`, `cs`, were built from a larger internal collection of generated submissions, from which a subset of 200 submissions per exercise and large language model was randomly chosen. The random choice ensures no bias is present in the datasets, as all submissions had equal probability to be chosen.

5.6.2 Threats to External Validity

Dependence on the Programming Language. For this evaluation, all datasets only contained submissions written in Java, therefore, we cannot claim with certainty that the results transfer to another programming language. In theory, the comments are not bound to a programming language, as all major programming languages feature comments, albeit with different formats or other changes. In practice however, documentation comments, which may make up the majority of comments within source code documents, can be formatted differently and contain other keywords, which may skew the results of this approach. Additionally, some supporting software like spell-checkers or formatters may be configured via source code comments in other languages, which then follow a strict schema again, which was not present in this evaluation.

Dependence on the Natural Language. The vast majority of comments in the used datasets were written in the English language. Even though the evaluated approach should work for other natural languages as well, the results may still differ. Some of the approaches above are based on word frequencies or word order within comments, which may be very different in other languages, making some words or word combinations much more frequent than others, which may alter the results in an unpredictable manner.

Selected Plagiarism Detection System. Even though the implementation for the evaluated comment processing pipeline was designed around and integrated within JPlag, the concept of the pipeline itself is not confined to a single tool only. With minor modifications to integrate with another API, any token-based source code plagiarism detection system should be able to integrate this approach.

5.6.3 Threats to Construct Validity

Planning of the evaluation. The evaluation was planned beforehand, and structured according to the Goal Question Metric approach [3]. All the goals, questions and metrics were selected before the evaluation was conducted. This ensures that the goals, questions or metrics were not altered after the results of the evaluation were analyzed, to manipulate the results in a favorable manner to the author.

Selected metrics. The usage of similarity scores, optionally classified as *Case of plagiarism*, *Possibly plagiarism* or *Unlikely plagiarism* is a common metric, as used in similar fashion in other research on source code plagiarism detection, for example, by Maisch [13], Sağlam [26] and Niehues [19]. The change of similarity scores was selected as a support metric, whose results are directly calculated from the similarity scores metric.

5.6.4 Threats to Reliability

Reproducible results. A reproduction package [25] was published, which allows reproduction of the results from this evaluation. The package includes:

- The datasets `ttt`, `pp19`, `pp56` and `cs` used for the evaluation
- All concept variants CV1-CV8, implemented in JPlag, with the source code and runnable JAR files
- The results of all concept variants & dataset combinations, with similarity scores and runtimes, including results for unreleased datasets
- The R scripts used to generate the graphs used in this thesis
- A readme file containing the exact steps & program arguments used to generate the result data

Using this reproduction package, the results of this evaluation should be generally reproducible. Due to data protection restrictions, the datasets `alphazeta`, `trafficsimulation`, `perseverance` and `tictactoe` could not be published, since they contain exam and exercise submissions from real students.

6 Limitations and Future Work

Due to various constraints, there are some limits to the effectiveness of the evaluated concept. This chapter lists potential enhancements to the methods described previously, which may be subject to future research.

Supported Programming Languages. So far, the comment processing pipeline has only been implemented and evaluated for the Java programming language. In theory, large parts of the processing pipeline should still be applicable to submissions written in other programming languages, only requiring a new extractor to extract comments from source files of other languages. In practice, other programming languages might have different standards for comments within source code, especially regarding the formatting of documentation comments, akin to JavaDoc for Java programs. Different formats for such documentation comments could lead to certain keywords or phrases being used more often in comments, which could skew the results of this approach. With different formatting, keywords or other standards used, the effectiveness of this approach might differ for other programming languages.

Support for multiple natural languages. In the datasets used, most of the comments were written in the English language, which made comparing them to each other easier. However, in a multilingual environment, different submissions within the same submission set may have comments written in different languages. Since the current approach simply compares the comments word by word or character by character, a valid attack would be simply translating comments to a different language. That would lead to largely changed words and characters with a very low similarity between them, which would then lead to a worsened detection of similar comments. To circumvent this, either some form of cross-language matching algorithm is needed, or all comments have to first be translated to a common language.

Advanced Natural Language Preprocessing. As a very basic first step with Natural Language Processing, lemmatization was the only preprocessing explored in this thesis. This approach could be expanded with other natural language processing techniques, such as stemming, token expansion or multi-word tokenization. Tying in to the previous point, some of these techniques, including lemmatization which was already used, require knowing the language of the natural text processed, which would in this case either have to be defined, detected during the preprocessing, or translated to a known language beforehand. However, advanced natural language preprocessing may greatly improve the matching of similar comments, since more words can be transformed into a shared base form, and possible

attacks like changing the tense or other grammatical structures within comments can be prevented.

Different Matching Algorithms. In this thesis, we have only evaluated a small number of matching algorithms. While the evaluated matching algorithms did improve the plagiarism detection, the plagiarism detection may even be improved with different matching algorithms, or a better threshold for the distance-based matching algorithms.

Weighting of Comment Tokens. So far, when computing the similarity score, one word in a comment was represented by one token in the token structure. This means that one matching comment token is worth about as much as one matching code token, which is roughly equivalent to a token from an abstract syntax tree. This implicitly already weighs the comment tokens, as it defines by how much comment tokens influence the similarity scores. This weighting could be changed, by for example only taking half the comment tokens, or double the comment tokens in account when calculating the similarity score. It is important that this weighting only happens during the computation of the similarity score, and not during the matching of comments, as all comment tokens are important during that step. However, the resulting change in similarity scores may be positively or negatively impacted by changing the weighting of the comment tokens, which may be subject to future research.

7 Conclusion

In the past, most token-based source code plagiarism detection systems have ignored textual content such as names of identifiers or variables or comments during the plagiarism detection. In this thesis, we have investigated a concept to introduce the comparison of textual content in form of comments into the plagiarism detection.

This comment processing pipeline extracts the comments separately from the source code files, tokenizes them, optionally preprocesses them with natural language processing techniques, matches them based on a matching algorithm, and increases the similarity of two submissions if matching comments were found. The comments were extracted into a token structure separate from the source code tokens, to avoid breaking up existing code matches with arbitrary comments. Additionally, comment tokens were only added to the similarity computation if they matched, to avoid reducing the similarity of submissions if either of them featured a lot of non-matching comments.

Three different matching algorithms, Greedy String Tiling, the Damerau-Levenshtein distance and the Bag of Words model utilizing the cosine similarity were compared for their usage in this concept. In one instance, advanced natural language preprocessing in the form of lemmatization was additionally implemented.

This concept with each matching algorithm was evaluated on datasets containing submissions from first-year programming lectures, which includes real cases of plagiarism, as well as data sets generated by large language models. Using these, the improvement of the plagiarism detection on both real submissions with plagiarisms and generated submissions could be evaluated.

We have observed that all approaches increased the similarity scores of submissions by different amounts. While most of the submissions with increased similarity scores had a low similarity score without comments, we did see small improvements across the board. This improvement may aid the manual review process, as similar comments may be highlighted, as well as improving the plagiarism detection overall. Further research is necessary to determine whether the increase in similarity score for previously low similarity submissions is due to unrelated comments being matched, or similar comments being found among the submissions. In addition to that, we have also seen that submissions generated by artificial intelligence contain similar comments, which are detected by these approaches. The concept itself, extracting comments separately from source code and only including them if they match, has worked well so far, and could potentially be further improved with different matching algorithms in future research.

Bibliography

[1] Lincoln A. Mullen et al. "Fast, Consistent Tokenization of Natural Language Text". In: *Journal of Open Source Software* 3.23 (Mar. 2018), p. 655. ISSN: 2475-9066. DOI: 10.21105/joss.00655.

[2] Vimala Balakrishnan and Ethel Lloyd-Yemoh. "Stemming and lemmatization: A comparison of retrieval performances". In: *Proceedings of SCEI Seoul Conferences* (2014). URL: http://eprints.um.edu.my/id/eprint/13423 (visited on 09/10/2025).

[3] Victor R. Basili. *Software modeling and measurement: the Goal/Question/Metric paradigm.* Tech. rep. USA: University of Maryland at College Park, 1992.

[4] William W Cohen et al. "A Comparison of String Distance Metrics for Name-Matching Tasks". en. In: *IIWeb*. Vol. 3. 2003.

[5] Georgina Cosma and Mike Joy. "Towards a Definition of Source-Code Plagiarism". en. In: *IEEE Transactions on Education* 51.2 (May 2008), pp. 195–200. ISSN: 0018-9359, 1557-9638. DOI: 10.1109/TE.2007.906776.

[6] Michael De Raadt et al. "Language Trends in Introductory Programming Courses". en. In: 2002. DOI: 10.28945/2464.

[7] Rares Folea and Emil Slusanschi. "Code Comments: A Way of Identifying Similarities in the Source Code". en. In: *Mathematics* 12.7 (Apr. 2024), p. 1073. ISSN: 2227-7390. DOI: 10.3390/math12071073.

[8] Gregory Grefenstette. "Tokenization". In: *Syntactic Wordclass Tagging*. Ed. by Nancy Ide et al. Vol. 9. Series Title: Text, Speech and Language Technology. Dordrecht: Springer Netherlands, 1999, pp. 117–133. ISBN: 978-90-481-5296-4. DOI: 10.1007/978-94-015-9273-4_9.

[9] Sethunya R Joseph et al. "Natural language processing: A review". In: *International Journal of Research in Engineering and Applied Sciences* 6.3 (2016), pp. 207–210.

[10] M. Joy and M. Luck. "Plagiarism in programming assignments". en. In: *IEEE Transactions on Education* 42.2 (May 1999), pp. 129–133. ISSN: 00189359. DOI: 10.1109/13.762946.

[11] *JPlag*. URL: https://github.com/jplag/JPlag.

[12] Oscar Karnalim and Gisela Kurniawati. "Programming style on source code plagiarism and collusion detection". en. In: *International Journal of Computing* (Mar. 2020), pp. 27–38. ISSN: 2312-5381, 1727-6209. DOI: 10.47839/ijc.19.1.1690.

[13] Robin Maisch. *Preventing Refactoring Attacks on Software Plagiarism Detection through Graph-Based Structural Normalization.* en. 2024. DOI: 10.5445/IR/1000172813.

[14] Robin Maisch et al. “Too Hot To Handle: The Effects of Temperature on AI-Generated Code Used to Cheat in Programming Assignments”. In: *2026 IEEE/ACM 48th International Conference on Software Engineering (ICSE-Companion ’26)*. ICSE-Companion ’26. New York, NY, USA: ACM, 2026. DOI: 10.1145/3774748.3795678.

[15] Paul W. McBurney and Collin McMillan. “An empirical study of the textual similarity between source code and source code summaries”. en. In: *Empirical Software Engineering* 21.1 (Feb. 2016), pp. 17–42. ISSN: 1382-3256, 1573-7616. DOI: 10.1007/s10664-014-9344-6.

[16] Lukas Michelbacher. *Multi-word tokenization for natural language processing*. en. 2013. DOI: 10.18419/OPUS-3208.

[17] George A. Miller. “WordNet: a lexical database for English”. en. In: *Communications of the ACM* 38.11 (Nov. 1995), pp. 39–41. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/219717.219748.

[18] *MOSS*. URL: https://theory.stanford.edu/~aiken/moss/.

[19] Nils Niehues. *Intelligent Match Merging to Prevent Obfuscation Attacks on Software Plagiarism Detectors*. en. 2023. DOI: 10.5445/IR/1000167446. URL: https://publikationen.bibliothek.kit.edu/1000167446 (visited on 09/10/2025).

[20] Matija Novak et al. “Source-code Similarity Detection and Detection Tools Used in Academia: A Systematic Review”. en. In: *ACM Transactions on Computing Education* 19.3 (Sept. 2019), pp. 1–37. ISSN: 1946-6226. DOI: 10.1145/3313290.

[21] José Carlos Paiva et al. “PROGpedia: Collection of source-code submitted to introductory programming assignments”. In: *Data in Brief* 46 (Feb. 2023), p. 108887. ISSN: 2352-3409. DOI: https://doi.org/10.1016/j.dib.2023.108887.

[22] Lutz Prechelt et al. “Finding Plagiarisms among a Set of Programs with JPlag”. en. In: *J. Univers. Comput. Sci.* 8.11 (2002). DOI: 10.3217/JUCS-008-11-1016.

[23] A Ramírez-de-la-Cruz et al. “High level features for detecting source code plagiarism across programming languages”. en. In: *FIRE Workshops*. 2015. URL: https://downloads.webis.de/pan/publications/papers/ramirezdelacruz_2015.pdf (visited on 09/10/2025).

[24] Aarón Ramírez-de-la-Cruz et al. “On the Importance of Lexicon, Structure and Style for Identifying Source Code Plagiarism”. en. In: *Proceedings of the Forum for Information Retrieval Evaluation on - FIRE ’14*. Bangalore, India: ACM Press, 2015, pp. 31–38. ISBN: 978-1-4503-3755-7. DOI: 10.1145/2824864.2824879.

[25] Moritz Rimpf. *Replication Package for "Incorporation of Text Content Similarity into Token-Based Source Code Plagiarism Detection"*. en. Sept. 2025. DOI: 10.5281/ZENODO.17092770.

[26] Timur Sağlam. “Mitigating Automated Obfuscation Attacks on Software Plagiarism Detection Systems”. en. Medium: PDF. PhD thesis. Karlsruher Institut für Technologie (KIT), 2025. URL: https://publikationen.bibliothek.kit.edu/1000179018 (visited on 04/10/2025).

[27] Timur Sağlam et al. “Detecting Automatic Software Plagiarism via Token Sequence Normalization”. In: *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. ICSE ’24. Lisbon, Portugal: Association for Computing Machinery, Apr. 2024, 113:1–113:13. ISBN: 9798400702174. DOI: 10.1145/3597503.3639192. URL: https://doi.org/10.1145/3597503.3639192.

[28] Timur Sağlam et al. *Mitigating Obfuscation Attacks on Software Plagiarism Detectors via Subsequence Merging*. en. Tech. rep. IEEE/ACM, 2025. DOI: 10.5445/IR/1000179016.

[29] Saul Schleimer et al. “Winnowing: Local Algorithms for Document Fingerprinting”. en. In: *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. 2003.

[30] Amit Singhal et al. “Modern information retrieval: A brief overview”. In: *IEEE Data Eng. Bull.* 24.4 (2001), pp. 35–43.

[31] Michael J Wise. “String Similarity via Greedy String Tiling and Running Karp-Rabin Matching”. en. In: *Online Preprint, Dec* 119.1 (1993).

[32] Bob Zeidman. “Detecting source-code plagiarism”. en. In: *Dr. Dobb’s Journal* 29.7 (2004).