Source Code Clone Detection Using Unsupervised Similarity Measures

Jorge Martinez-Gil[iD]

Software Competence Center Hagenberg GmbH
Softwarepark 32a, 4232 Hagenberg, Austria,
`jorge.martinez-gil@scch.at`

Abstract. Assessing similarity in source code has gained significant attention in recent years due to its importance in software engineering tasks such as clone detection and code search and recommendation. This work presents a comparative analysis of unsupervised similarity measures for identifying source code clone detection. The goal is to overview the current state-of-the-art techniques, their strengths, and weaknesses. To do that, we compile the existing unsupervised strategies and evaluate their performance on a benchmark dataset to guide software engineers in selecting appropriate methods for their specific use cases. The source code of this study is available at `https://github.com/jorge-martinez-gil/codesim`

Key words: Software Engineering, Clone Detection, Similarity Measures, Code Similarity

1 Introduction

Source code clone detection holds increasing importance in the current software engineering landscape, and its significance is likely to grow even further [1]. The reason is that this approach is crucial in software development since it can help address various problems during software maintenance [18]. Clones are duplicate or similar pieces of code within a software project. Therefore, consider the chaotic situation that would happen if a bug is fixed or a change is made to a piece of code but not to its duplicates. To avoid such situations, developers should have tools to automatically evaluate the likeness between code fragments based on various aspects of their form and functionality [37, 36].

In this work, we address this challenge from the point of view of using similarity measures. When working with general and source code similarity, it is necessary to distinguish between supervised and unsupervised approaches [26]. On the one hand, supervised approaches require a training set of pairs of code fragments labeled as similar or dissimilar, which is often difficult to get, at least in terms of the necessary volume [?]. On the other hand, unsupervised approaches do not require a training set, and they can be used to measure the similarity of any two code fragments with no prior knowledge and a low consumption of computational resources [30].

This work evaluates at least one representative implementation of unsupervised similarity measures. This analysis is based on their underlying principles. In this regard, we explore measures ranging from trivial strategies for token comparison to the more advanced comparison of embeddings [34]. To facilitate a thorough assessment, we use a benchmark dataset comprising diverse code fragments with varying degrees of similarity and check the performance of each similarity measure across the dataset.

Our analysis focuses on shedding light on practical applicability and scalability. The rationale behind summarizing the existing body of knowledge and identifying research gaps is to offer a resource for software engineers interested in unsupervised measures for detecting source code clones. Furthermore, in contrast to recent works, which address the challenge from a purely qualitative perspective, our work aims at a quantitative analysis.

Therefore, this work's primary and overall contribution aims to guide the choice of appropriate unsupervised similarity measures for clone detection. Additionally, it identifies promising directions for future research in source code similarity assessment. The following specific contributions achieve this:

– We present the fundamental challenge regarding clone detection and the possibility of building solutions to cope with the absence of labeled data and different coding styles.
– We compile an extensive collection of unsupervised semantic similarity measures being able to compare textual information to elucidate the most promising measures in this context.
– We evaluate empirically this collection of unsupervised measures with focus on accuracy, time consumption, and practical feasibility. Our results indicate that several measures could be valid tools for source code clone detection.

The remainder of this paper is structured as follows: Section 2 introduces the background of this critical challenge. Section 3 technically explains the measures to face this challenge and shows several examples. Section 4 evaluates all the measures reviewed in the previous version using a complete benchmark dataset. Section 5 discusses the results of our experiments. Finally, the paper concludes with lessons learned and lines of future work.

2 Background

This section presents the information necessary to understand the challenge. First, we define code similarity assessment; second, we explain why this challenge is so significant nowadays; and third, we describe the implications and impact of the challenge in academia and industry.

2.1 Problem definition

It seems clear that code duplication can lead to inconsistencies, especially if a change is made in one part of the code but not in its clones [32]. In this context,

it is also important to differentiate between code similarity measurement and identification of source code clones. Code similarity measurement is a broad concept, and clone identification is one of its applications. For instance, the most similar instances can be reported as cloned instances just using a threshold value to filter out the results of code similarity measurement [4].

Although there is no strict definition for the assessment of code similarity, it is possible to describe the problem formally, such as given a set of code fragments $S = \{C_1, C_2, \ldots, C_n\}$, the goal is to find a function $f : S \times S \rightarrow [0, 1]$ that computes the similarity score between any C_i and C_j.

Therefore, f should map a given pair (C_i, C_j) to a value in the continuous interval $[0, 1]$, whereby:

– $f(C_i, C_j) = 0$ indicates that C_i and C_j are completely dissimilar
– $f(C_i, C_j) = 1$ indicates that C_i and C_j are identical
– $f(C_i, C_j)$ increases as the similarity between C_i and C_j increases and vice versa

The function f should compare C_i and C_j, considering various characteristics such as variables, constants, function calls, comments, overall logic, or any other code element susceptible to being compared [38]. Then, clone detection can be implemented to discriminate between instances using, for example, a point value separating clones and non-clones. Furthermore, although it was not considered in the frame of this work, it would be desirable that the results could be accompanied by an explanation [20] for facilitating human assessment.

Similarity categories Multiple copies of similar code throughout a software project can make managing the codebase difficult. However, not all the cases are equal. In comparing pieces of code, some recent literature has categorized the code similarities into four categories [3]. These categories help us understand the degree of resemblance between two code fragments so that each category represents a different level of likeness:

– Category I: The code fragments are identical, with just minor variations in white spaces and annotations.
– Category II: The code fragments have the same structure, but there are differences in the names of the identifiers, data types, spaces, and comments.
– Category III: Additionally, parts of the code might be removed or altered, or new parts could be incorporated.
– Category IV: The code fragments may appear different but implement analogous functionality.

The rationale behind this categorization is to provide insights into code comparison and help software engineers understand the cases they must face to make better-informed decisions related to code maintenance.

2.2 The importance of unsupervised measures

Detecting code clones is essential for maintaining software quality [16]. Unsupervised code similarity assessment can help address this challenge since several practical aspects are common to many software development projects:

– Unsupervised measures do not rely on labeled training data, making use of a ground truth unnecessary. Labeled examples are only needed to validate the performance of unsupervised approaches [27].
– Code can be written in various programming languages, using different coding styles, etc. Some unsupervised measures can accommodate this variety without a complex universal similarity metric.
– Understanding the meaning of code is complicated because code fragments may be functionally equivalent even if they look dissimilar, and vice versa. Some unsupervised measures can face that challenge.
– Codebases often contain comments, noise, etc. Some unsupervised measures can differentiate between meaningful code patterns and unrelated elements.

2.3 Future perspectives

Duplicate code increases the maintenance burden because changes must be replicated across all clones, which is time-consuming and error-prone [5]. Therefore, identifying and refactoring these clones can reduce the maintenance effort [11].

Nowadays, where many open-source libraries and code repositories exist, unsupervised source code similarity measurement can be helpful; it enables developers to navigate this diverse ecosystem and search for relevant code efficiently with low consumption of computational resources [28]. This importance extends to facilitating code reuse, which is crucial for reducing development time in the face of growing software complexity [9].

Furthermore, detecting code similarities can improve security by identifying vulnerabilities with known code patterns in the context of growing security troubles. It can also contribute to code maintainability and refactoring efforts, allowing developers to ensure software projects' long-term sustainability.

We can also think of applications within various industries that benefit from increased compliance and reliability in critical systems [29]. Furthermore, collaboration tools facilitate cooperation by connecting developers with similar code, and quality assurance strategies could benefit from unsupervised code similarity measurement by identifying similar cases for complete test coverage.

3 Methods

Early approaches for assessing the similarity relied on just textual analysis [25]. These techniques, while efficient, often struggle to capture the structural aspects of code, resulting in limited accuracy [12]. However, the field has evolved a lot in recent years. More sophisticated similarity measures assumed to perform better have been proposed [10].

3.1 Unsupervised methods

There are many methods (a.k.a. semantic similarity measures) to determine the similarity between textual entities. Each measure offers a unique approach based on specific characteristics or representations of the compared entities. We have identified about 21 families that could be applied in this context, briefly explained below in alphabetical order.

– **Abstract Syntax Trees (ASTs) Similarity**: ASTs are hierarchical representations of the structure of code. AST similarity measures compare the structural similarity between different AST representing code [22].
– **Bag-of-Words Similarity**: This similarity measure calculates the resemblance between texts by considering the frequency of individual words in each text without considering word order or structure [7].
– **Code Embeddings Similarity (CodeBERT)**: Code embeddings are vector representations of source code. This method measures the similarity of code based on these embeddings [2].
– **Comments Similarity**: It measures the similarity between code comments, which can be helpful for code documentation and understanding. In principle, many traditional text similarity measures can be used [29].
– **Fuzzy Matching Similarity**: Fuzzy matching compares strings for minor syntactical variations. It is often used in data matching and search applications [40], but we apply it here to measure code similarity.
– **Function Calls Similarity**: This family measures the similarity between different code fragments based on the functions and procedures in the code fragments [42].
– **Graph-based Similarity**: It calculates similarity based on a graph's relations, which could represent various data structures and dependencies [43].
– **Jaccard Similarity**: Jaccard similarity measures the similarity between sets of tokens by comparing their intersection and union. It is commonly used in text analysis, recommendation systems, and information retrieval [14].
– **Levenshtein Similarity**: This measure, also known as edit distance, calculates the similarity between two strings by measuring the number of edits needed to transform one into the other [24].
– **Longest Common Subsequence (LCS) Similarity**: LCS similarity calculates the similarity between two sequences by finding the longest common subsequence between them [6].
– **Metrics Similarity**: The idea is first to compute various metrics related to the source code and then estimate the similarity between the values obtained [33].
– **N-grams Similarity**: N-grams are contiguous sequences of 'n' items (e.g., words or characters). N-gram similarity measures the similarity between texts based on shared n-grams [8].
– **Output Analysis Similarity**: This method measures the similarity of program outputs, which can be helpful for testing and debugging. In principle, and if we assume the outputs as text, a wide range of traditional text similarity measures can be used [29].

– **Perceptual Hashing Similarity**: Perceptual hashing, often used in image similarity, aims to generate a fixed-length hash code from images. In our context, this method measures similarity based on hashes from visual representation of the code [35].
– **Program Dependence Graph Similarity**: This measure assesses the similarity between code by analyzing the program dependence graph, which represents the dependencies between program elements [23].
– **Rolling Hash Similarity**: A rolling hash is a hash function that can be updated efficiently as new data is processed. Rolling hash similarity can compare substrings (hashes) in large texts [15]. We use here for comparing code.
– **Running-Karp-Rabin Greedy-String-Tiling (RKR-GST) Similarity**: It is often used in the context of detecting plagiarism by identifying maximal sequences of contiguous matching tokens (tiles) [41].
– **Semdiff Similarity**: Semdiff is a method for detecting semantic differences between program versions. Semdiff similarity measures how code changes affect the program's semantics [17].
– **Semantic Clone Similarity**: This method family tries to measure the similarity of code fragments based on the semantic meaning of the names of the program elements (variables, methods, etc.) [13].
– **TF-IDF Similarity**: Term Frequency-Inverse Document Frequency (TF-IDF) is used in text analysis to measure the importance of words in a text compared to a larger corpus. TF-IDF similarity compares texts based on these weighted terms [19].
– **Winnow Similarity**: It is a text comparison algorithm that identifies similar texts by hashing them and comparing their fingerprints [39].

Next, we will look at some Java code examples, representing some interesting cases of source code cloning, illustrating how all these similarity measures quantify code similarity in practice.

3.2 Examples

In the examples below, *T1* and *T01* are two Java classes that produce the same output but with different approaches: *T1* prints the statement *Welcome to Java* five times using five separate print statements. *T01* achieves the same output using a for loop that iterates five times, printing the statement on each iteration. From the perspective of code clone detection, these two classes are Category IV clones. The reason is that both are pieces of code that perform the same operations but are implemented through different syntactic variations.

Even though the actual text of the code differs, the for loop versus repeated print statements, the meaning, and the output are the same. However, detecting such code clones can be challenging because it is not just a matter of matching text strings but requires a deep understanding of the code's logic and output, which can be challenging to address. However, it is common to find similar cases in real settings.

```
1  public class T1 {
2      public static void main(String[] args) {
3          System.out.println("Welcome to Java");
4          System.out.println("Welcome to Java");
5          System.out.println("Welcome to Java");
6          System.out.println("Welcome to Java");
7          System.out.println("Welcome to Java");
8      }
9  }
```

```
1  public class T01 {
2      public static void main(String[] args){
3
4          for(int i = 0; i < 5; i++){
5              System.out.println("Welcome To Java");
6          }
7
8      }
9  }
```

On the contrary, the classes *TemperatureConverter* and *CurrencyConverter* are similar in form. However, an experienced developer would quickly realize that they calculate different things (temperature vs currencies), so they should not be considered clones. However, their high similarity in form might make many unsupervised measures consider them Category II clones.

```
1  public class TemperatureConverter {
2      public static double celsiusToFahrenheit(double cels) {
3          return cels * 9 / 5 + 32;
4      }
5  }
```

```
1  public class CurrencyConverter {
2      public static double usdToEur(double usd) {
3          return usd * 85 / 100;
4      }
5  }
```

Table 1 compares various unsupervised similarity measures for code analysis. Some of these measures are based on textual similarity, while others are based on the structure of the code. Other measures might analyze the code's functionality beyond just the text or structure. In principle, there is no accurate or inaccurate result in this context. However, intuition tells us that some measures may better serve our purposes. The ideal result would be 1.00 in the first column and 0.00 in the second. Nevertheless, any result that can discern clones (giving them a high similarity value) from non-clones (giving them a low similarity value) would be good.

Measure	Score-Ex1.	Score-Ex2.
Abstract Syntax Trees (ASTs) Similarity	0.50	0.81
Bag-of-Words Similarity	0.72	0.65
Code Embeddings Similarity	0.99	1.00
Comments Similarity	1.00	1.00
Fuzzy Matching Similarity	0.54	0.64
Function Calls similarity	1.00	0.00
Graph-based Similarity	0.38	0.34
Jaccard Similarity	0.27	0.35
Levenshtein Similarity	0.51	0.69
Longest Common Subsequence (LCS) Similarity	0.19	0.29
Metrics Similarity	0.98	1.00
N-grams Similarity	0.26	0.14
Output Analysis Similarity	1.00	0.00
Perceptual Hashing Similarity	0.69	0.88
Program Dependence Graph Similarity	1.00	1.00
R.-Karp-Rabin G.-Str.-Til. (RKR-GST) Similarity	0.96	0.83
Rolling Hash Similarity	1.00	0.55
Semdiff Similarity	0.22	0.40
Semantic Clone Similarity	0.54	0.79
TF-IDF Similarity	0.67	0.48
Winnow Similarity	1.00	0.60

Table 1: Comparison of various unsupervised similarity measures for code similarity measurement

Something special happens with the *Comments Similarity* result. Since none of the displayed code fragments have comments, the measure thinks they are similar. This is just an example of why caution is necessary when considering the results.

4 Evaluation

Several aspects come into play when evaluating and comparing unsupervised similarity measures for clone detection. To effectively evaluate these techniques, it is essential to consider the dataset's nature, the clone categories to face, and the task's requirements.

In this way, some measures excel in comparing textual content, making them suitable for detecting cloned text. Other techniques are more apt for identifying similar functionality. In contrast, other measures can assist in uncovering structural similarities between code and text. The choice depends on the nature of the data in the benchmark dataset.

4.1 Dataset

We are using here the IR-Plag dataset[1] which is designed to serve as a benchmark for evaluating and comparing the performance of different strategies [21]. This dataset includes plagiarized code files deliberately crafted to mimic academic plagiarism behaviors. Although the dataset is compiled to detect plagiarism, it is valid for our purposes since the practical result of plagiarism and cloning is the same in practice, even if their original intentionality might differ (intention to deceive in the first case, no intentionality in the second). Moreover, this dataset does not merely focus on simplistic plagiarism attacks but encompasses a complete range of complexities. Although this dataset does not classify clones, it can be useful in detecting suitable semantic similarity measures for mitigating code redundancy and duplication within complex software projects.

In analyzing a dataset of code files, we observe the following metrics: The dataset contains seven original code files. A high number of files, 355 (77%), are identified as plagiarized, suggesting a considerable prevalence of duplication. There are 105 non-plagiarized files, which might represent modified or derivative works. The total count of code files in the dataset is 467. Within these files are 59,201 tokens, with 540 distinct tokens, indicating the variety of programming language elements used. The size of the files varies significantly, with the largest file containing 286 tokens and the smallest comprising 40 tokens. On average, a code file in this dataset includes around 126 tokens. These insights show the dataset's composition, reflecting a great diversity in programming syntax.

4.2 Results

In the following, we show the results obtained from the experiments on the IR-Plag dataset. We look primarily at the accuracy and the execution time required as we believe these are two of the most important aspects to consider when considering putting a measure into operation. These results can be reproduced with the provided source code[2].

On the one hand, Figure 1 compares the different measures. The horizontal axis quantifies the accuracy of each measure, while the vertical axis lists the unsupervised measures. *Output Analysis* has the highest accuracy score, which could imply that it is most effective at detecting code that performs the same function despite differences in implementation. Contrariwise, *LCS* has the lowest accuracy score, indicating that it might not be as effective in this comparison.

It is essential to note that the dataset contains 77% clones. Therefore, any method could say that all comparisons give rise to a clone and thus obtain a 0.77 accuracy. This would not be a good result. Figure 1 shows that only using three measures produces a real gain over that base result.

On the other hand, Figure 2 presents a comparative analysis of various measures used to execute code, measured by their execution time. The horizontal

---

[1] https://github.com/oscarkarnalim/sourcecodeplagiarismdataset
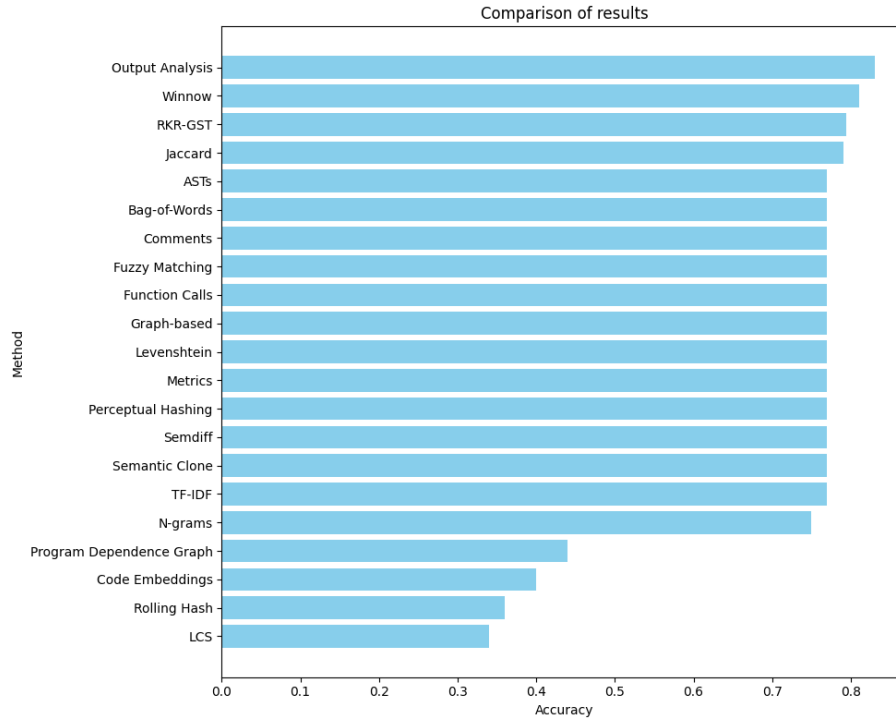[2] https://github.com/jorge-martinez-gil/codesim

Fig. 1: Accuracy of the unsupervised semantic similarity measures when performing clone detection

axis quantifies the execution time, while the vertical axis lists the unsupervised measures. The *Output Analysis* shows the longest execution time, significantly outpacing other methods such as *N-grams* and *Code Embeddings*. The remaining measures show lower execution times, suggesting a more efficient performance. Two facts can be immediately deduced from these experiments:

1. First, only four of the measures studied (i.e., *Output Analysis*, *Winnow*, *RKR-GST*, and *Jaccard*) help identify clones effectively. This suggests that most unsupervised semantic similarity measures are not helpful in the current form. Therefore, more research on innovative approaches to clone detection is needed.

2. Despite being excellent in accuracy (e.g., *Output Analysis*), some techniques incur such a high computational cost that incorporating them into a practical, real-world tool for programmers becomes unrealistic. The reason *Output Analysis* takes so much execution time is that it must take the two pieces of code, encapsulate them for compilation, pass some random parameters to them (if necessary), and compare the outputs produced. This entire process is very computationally expensive.
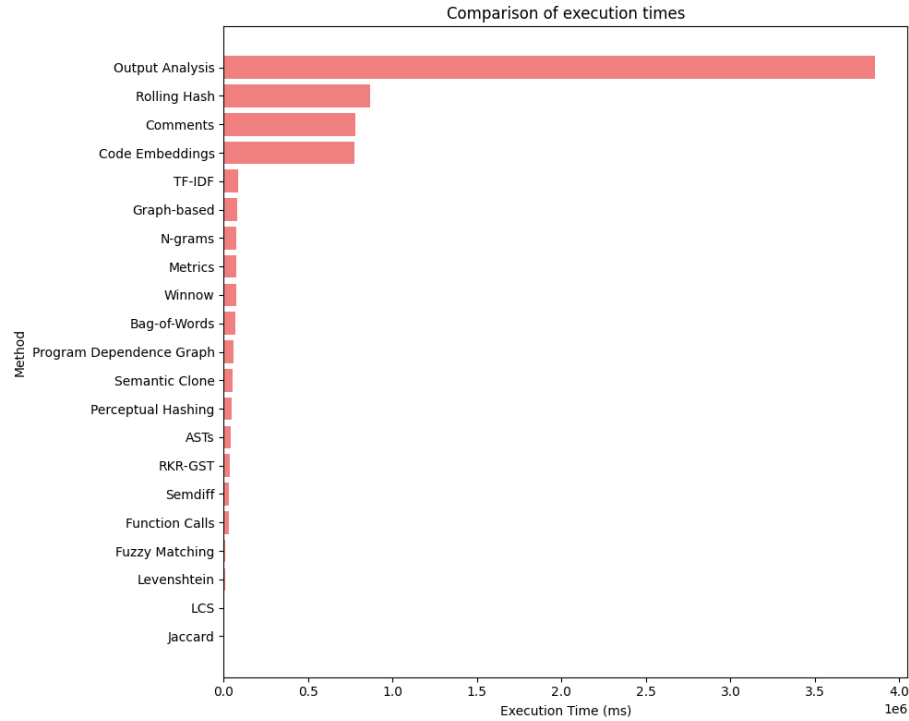
Fig. 2: Execution time of the unsupervised semantic similarity measures when performing clone detection

Other time-consuming similarity measures are *Rolling Hash* (very intensive in the use of mathematical operations), *Comments Similarity* (identifying comments involves the use of regular expressions, which is computationally expensive), and *Code Embeddings* (which needs to search and identify embeddings as well as perform operations on them). Therefore, it would be possible to define a feasibility index that calculates a combination of accuracy and execution time to elucidate which measures could work well in real environments. This could be done by weighing the importance of accuracy about time and dividing the result by the total execution time.

Figure 3 shows us the calculation of the feasibility index. We consider the accuracy importance over the execution time as 10:1. Therefore, just *Jaccard*, *RKR-GST*, and *Winnow* (in that order) would be good candidates for use in real environments due to a reasonable combination of accuracy and execution time. However, these measures should be used just for an automatic recommendation since the gain in accuracy over the base result only allows us to operate them with supervision.
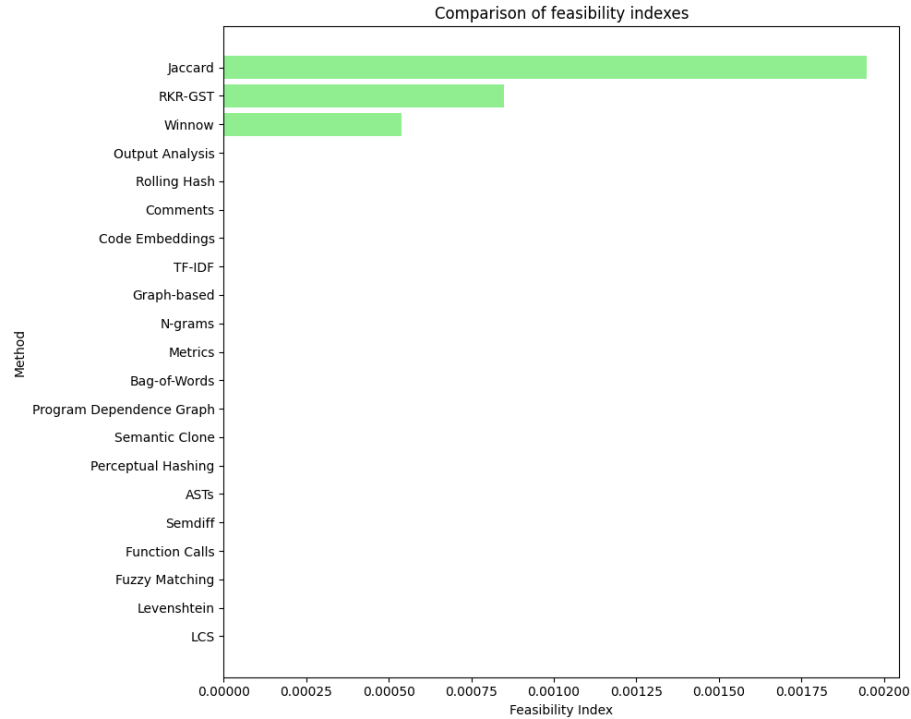
Fig. 3: Comparison of the feasibility index of the unsupervised methods

5 Discussion

Our experiments show that a reduced group of unsupervised source code similarity measurements could serve for source code clone detection. These methods could improve various aspects of software engineering. For example, they could suggest the existence of clones and, therefore, present an opportunity to refactor the code into reusable parts. This might facilitate code reuse, which is vital in software engineering.

It is also necessary to remark that noisy and unstructured code environments characterize real-world scenarios. We have identified several unsupervised measures that have shown promise in managing this noise and variability, making them valuable when labeled data is limited or impractical.

Despite some progress, our research still needs to solve several challenges. These include achieving cross-language similarity measurement and ensuring scalability for large codebases. These challenges present compelling opportunities for future research. This means that our research results, although slightly applicable in their current form, need further research to be useful as software development advances into a more automated future.

6 Conclusion

The challenge of source code clone detection represents a crucial aspect of software engineering that impacts several applications. In this work, we have evaluated the existing unsupervised similarity measures to address the challenges of the absence of labeled data and diverse coding styles. Our research will illustrate how unsupervised source code similarity measurement can facilitate clone detection.

As codebases grow, developing accurate and efficient unsupervised similarity measures remains an essential area of exploration for the community. Furthermore, the need for effective unsupervised techniques will likely expand as the software industry evolves. Although studying supervised techniques may promise good results, unsupervised techniques will always be an option due to their more realistic requirements, adaptability, interpretability, and efficiency.

Therefore, the future of source code clone detection using unsupervised measures holds notable promise. Future efforts could focus on hybrid approaches that integrate the strengths of different methods (a.k.a. ensembles), leading to more robust and accurate similarity assessments. Exploring transfer learning techniques could also be a way to improve performance. The goal should be to improve strategies for code analysis with minimal human intervention.

Acknowledgments

References

1. Qurat Ul Ain, Wasi Haider Butt, Muhammad Waseem Anwar, Farooque Azam, and Bilal Maqbool. A systematic review on code clone detection. *IEEE access*, 7:86121–86144, 2019.
2. Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.
3. Rodrigo C Aniceto, Maristela Holanda, Carla Castanho, and Dilma Da Silva. Source code plagiarism detection in an educational context: A literature mapping. In *2021 IEEE Frontiers in Education Conference (FIE)*, pages 1–9. IEEE, 2021.

4. Ira D. Baxter, Andrew Yahin, Leonardo Mendonça de Moura, Marcelo Sant'Anna, and Lorraine Bier. Clone detection using abstract syntax trees. In *1998 International Conference on Software Maintenance, ICSM 1998, Bethesda, Maryland, USA, November 16-19, 1998*, pages 368–377. IEEE Computer Society, 1998.

5. Stefan Bellon, Rainer Koschke, Giulio Antoniol, Jens Krinke, and Ettore Merlo. Comparison and evaluation of clone detection tools. *IEEE Transactions on software engineering*, 33(9):577–591, 2007.

6. Lasse Bergroth, Harri Hakonen, and Timo Raita. A survey of longest common subsequence algorithms. In *Proceedings Seventh International Symposium on String Processing and Information Retrieval. SPIRE 2000*, pages 39–48. IEEE, 2000.

7. Courtney D Corley and Rada Mihalcea. Measuring the semantic similarity of texts. In *Proceedings of the ACL workshop on empirical modeling of semantic equivalence and entailment*, pages 13–18, 2005.

8. Marc Damashek. Gauging similarity with n-grams: Language-independent categorization of text. *Science*, 267(5199):843–848, 1995.

9. Yingnong Dang, Song Ge, Ray Huang, and Dongmei Zhang. Code clone detection experience at microsoft. In *Proceedings of the 5th International Workshop on Software Clones*, pages 63–64, 2011.

10. Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: pre-training of deep bidirectional transformers for language understanding. In Jill Burstein, Christy Doran, and Thamar Solorio, editors, *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, pages 4171–4186. Association for Computational Linguistics, 2019.

11. Shihan Dou, Junjie Shan, Haoxiang Jia, Wenhao Deng, Zhiheng Xi, Wei He, Yueming Wu, Tao Gui, Yang Liu, and Xuanjing Huang. Towards understanding the capability of large language models on code clone detection: a survey. *arXiv preprint arXiv:2308.01191*, 2023.

12. Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(3):319–349, 1987.

13. Mark Gabel, Lingxiao Jiang, and Zhendong Su. Scalable detection of semantic clones. In *Proceedings of the 30th international conference on Software engineering*, pages 321–330, 2008.

14. Sakib Haque, Zachary Eberhart, Aakash Bansal, and Collin McMillan. Semantic similarity metrics for evaluating source code summarization. In *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, pages 36–47, 2022.

15. Anggit Dwi Hartanto, Andy Syaputra, and Yoga Pristyanto. Best parameter selection of rabin-karp algorithm in detecting document similarity. In *2019 International Conference on Information and Communications Technology (ICOIACT)*, pages 457–461. IEEE, 2019.

16. Yoshiki Higo, Yasushi Ueda, Toshihro Kamiya, Shinji Kusumoto, and Katsuro Inoue. On software maintenance process improvement based on code clone analysis. In *Product Focused Software Process Improvement: 4th International Conference, PROFES 2002 Rovaniemi, Finland, December 9–11, 2002 Proceedings 4*, pages 185–197. Springer, 2002.

17. Susan Horwitz. Identifying the semantic and textual differences between two versions of a program. In *Proceedings of the ACM SIGPLAN 1990 conference on Programming language design and implementation*, pages 234–245, 1990.

18. Elmar Juergens, Florian Deissenboeck, Benjamin Hummel, and Stefan Wagner. Do code clones matter? In *2009 IEEE 31st International Conference on Software Engineering*, pages 485–495. IEEE, 2009.
19. Oscar Karnalim. Tf-idf inspired detection for cross-language source code plagiarism and collusion. *Computer Science*, 21, 2020.
20. Oscar Karnalim. Explanation in code similarity investigation. *IEEE Access*, 9:59935–59948, 2021.
21. Oscar Karnalim, Setia Budi, Hapnes Toba, and Mike Joy. Source code plagiarism detection in academia with information retrieval: Dataset and the observation. *Informatics in Education*, 18(2):321–344, 2019.
22. Oscar Karnalim and Simon. Syntax trees and information retrieval to improve code similarity detection. In *Proceedings of the Twenty-Second Australasian Computing Education Conference*, pages 48–55, 2020.
23. Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings Eighth Working Conference on Reverse Engineering*, pages 301–309. IEEE, 2001.
24. Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.
25. Jorge Martinez-Gil. Semantic similarity aggregators for very short textual expressions: a case study on landmarks and points of interest. *J. Intell. Inf. Syst.*, 53(2):361–380, 2019.
26. Jorge Martinez-Gil. A comprehensive review of stacking methods for semantic similarity measurement. *Machine Learning with Applications*, 10:100423, 2022.
27. Jorge Martinez-Gil. A comparative study of ensemble techniques based on genetic programming: A case study in semantic similarity assessment. *Int. J. Softw. Eng. Knowl. Eng.*, 33(2):289–312, 2023.
28. Jorge Martinez-Gil and Jose M. Chaves-Gonzalez. Automatic design of semantic similarity controllers based on fuzzy logics. *Expert Syst. Appl.*, 131:45–59, 2019.
29. Jorge Martinez-Gil and Jose M. Chaves-Gonzalez. Semantic similarity controllers: On the trade-off between accuracy and interpretability. *Knowl. Based Syst.*, 234:107609, 2021.
30. Jorge Martinez-Gil and Jose Manuel Chaves-Gonzalez. A novel method based on symbolic regression for interpretable semantic similarity measurement. *Expert Syst. Appl.*, 160:113663, 2020.
31. Jorge Martinez-Gil and Jose Manuel Chaves-Gonzalez. Sustainable semantic similarity assessment. *Journal of Intelligent & Fuzzy Systems*, 43(5):6163–6174, 2022.
32. Matija Novak, Mike Joy, and Dragutin Kermek. Source-code similarity detection and detection tools used in academia: a systematic review. *ACM Transactions on Computing Education (TOCE)*, 19(3):1–37, 2019.
33. Alberto S Nuñez-Varela, Héctor G Pérez-Gonzalez, Francisco E Martínez-Perez, and Carlos Soubervielle-Montalvo. Source code metrics: A systematic mapping study. *Journal of Systems and Software*, 128:164–197, 2017.
34. Matthew E. Peters, Mark Neumann, Mohit Iyyer, Matt Gardner, Christopher Clark, Kenton Lee, and Luke Zettlemoyer. Deep contextualized word representations. In Marilyn A. Walker, Heng Ji, and Amanda Stent, editors, *Proceedings of the 2018 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2018, New Orleans, Louisiana, USA, June 1-6, 2018, Volume 1 (Long Papers)*, pages 2227–2237. Association for Computational Linguistics, 2018.

35. Chaiyong Ragkhitwetsagul, Jens Krinke, and Bruno Marnette. A picture is worth a thousand words: Code clone detection based on image similarity. In *12th IEEE International Workshop on Software Clones, IWSC 2018, Campobasso, Italy, March 20, 2018*, pages 44–50. IEEE Computer Society, 2018.
36. Chanchal K Roy, James R Cordy, and Rainer Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, 74(7):470–495, 2009.
37. Chanchal Kumar Roy and James R Cordy. A survey on software clone detection research. *Queen's School of computing TR*, 541(115):64–68, 2007.
38. Neha Saini, Sukhdip Singh, et al. Code clones: Detection and management. *Procedia computer science*, 132:718–727, 2018.
39. Saul Schleimer, Daniel S Wilkerson, and Alex Aiken. Winnowing: local algorithms for document fingerprinting. In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 76–85, 2003.
40. Nimisha Singla and Deepak Garg. String matching algorithms and their applicability in various applications. *International journal of soft computing and engineering*, 1(6):218–222, 2012.
41. Michael J Wise. String similarity via greedy string tiling and running karp-rabin matching. *Online Preprint, Dec*, 119(1):1–17, 1993.
42. Ming Xu, Lingfei Wu, Shuhui Qi, Jian Xu, Haiping Zhang, Yizhi Ren, and Ning Zheng. A similarity metric method of obfuscated malware using function-call graph. *Journal of Computer Virology and Hacking Techniques*, 9:35–47, 2013.
43. Laura A Zager and George C Verghese. Graph similarity scoring and matching. *Applied mathematics letters*, 21(1):86–94, 2008.