A Comparative Evaluation of Unsupervised Similarity Measures for Source Code Clone Detection

Jorge Martinez-Gil

Software Competence Center Hagenberg GmbH (SCCH), Hagenberg, Austria
jorge.martinez-gil@scch.at

Abstract. Source code similarity assessment is central to many software engineering tasks, including clone detection, code search, reuse analysis, and recommendation. This paper presents a comparative study of unsupervised similarity measures for detecting source code clones. We review representative strategies, organize them according to their underlying assumptions, and evaluate their performance on a benchmark dataset. The study discusses the practical strengths and limitations of each method, with the aim of supporting software engineers and researchers in choosing suitable similarity techniques for different clone detection scenarios.

Keywords: code similarity · clone detection · software engineering

1 Introduction

Source code clone detection is a fundamental software engineering task whose importance continues to increase as software systems grow in size and complexity. Code clones, defined as duplicated or highly similar code fragments within a software system, can negatively affect maintainability, reliability, and evolution. When modifications or bug fixes are applied to one clone instance but not to its counterparts, inconsistencies may be introduced, potentially leading to software defects and increased maintenance costs. Consequently, effective mechanisms for automatically assessing the similarity between code fragments are essential for supporting software maintenance activities.

This work addresses source code clone detection from the perspective of similarity measurement. In the domain of source code analysis, similarity assessment techniques are commonly categorized as supervised or unsupervised [18]. Supervised approaches rely on labeled datasets containing pairs of code fragments annotated as similar or dissimilar. While these methods can achieve strong performance, obtaining sufficiently large and representative training datasets is often difficult and costly. In contrast, unsupervised approaches do not require labeled data and can be applied directly to previously unseen code fragments, making them attractive for many practical scenarios.

The objective of this study is to evaluate representative implementations of the principal families of unsupervised similarity measures. The analysis covers

techniques based on a wide range of underlying principles, from simple token-based comparisons to methods that exploit vector embeddings of source code. To enable a systematic evaluation, we employ a benchmark dataset containing code fragments with varying degrees of similarity and assess the effectiveness of each measure across the entire dataset.

Particular attention is given to practical applicability and scalability. In addition to providing a quantitative evaluation of existing techniques, this study seeks to identify their strengths, limitations, and areas requiring further investigation. The resulting analysis is intended to serve as a reference for software engineers and researchers interested in applying unsupervised similarity measures to source code clone detection. Furthermore, whereas many recent studies approach the topic primarily from a qualitative standpoint, our work provides an extensive quantitative comparison of representative unsupervised techniques.

The primary contribution of this work is a systematic evaluation of unsupervised similarity measures for source code clone detection, providing guidance on their suitability for practical use. A further contribution is the identification of promising directions for future research in source code similarity assessment. These contributions are realized through the following objectives:

- 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 [20]. 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.

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 $f : S \times S \to [0, 1]$ 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 [19]. 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 for facilitating human assessment.

Similarity Categories The presence of similar code fragments across a software project can increase maintenance complexity. To characterize different forms of similarity, recent studies have grouped code clones into four categories [2]. These categories describe increasing levels of variation between two code fragments:

- **Category I**: Code fragments are identical except for differences in whitespace, formatting, or comments.
- **Category II**: Code fragments share the same structure, but identifiers, data types, comments, or formatting elements differ.
- **Category III**: Code fragments remain largely similar, but statements may be added, removed, or modified.
- **Category IV**: Code fragments differ syntactically but implement equivalent or closely related functionality.

This classification provides a common basis for analyzing code similarity and supports software engineers in selecting appropriate maintenance, refactoring, and clone management strategies.

2.2 Importance of Unsupervised Measures

Code clone detection plays a key role in maintaining software quality and reducing maintenance costs [10]. Unsupervised code similarity measures are particularly attractive because they address several practical challenges commonly encountered in software development:

- They do not require labeled training data, eliminating the need for manually constructed ground-truth datasets. Labeled examples are only necessary for evaluation purposes.
- They can be applied across different programming languages, coding conventions, and development environments without requiring language-specific training.
- They can capture similarities between code fragments that exhibit different syntactic forms while preserving related functionality.
- They can reduce the influence of comments, formatting, and other non-functional elements, focusing instead on meaningful code characteristics.

2.3 Future Perspectives

Code duplication increases maintenance effort because modifications often need to be propagated across multiple clone instances. Identifying and refactoring clones can therefore reduce development costs and improve software consistency.

The growing availability of open-source repositories and software libraries further increases the value of unsupervised source code similarity assessment. These techniques can assist developers in locating relevant code, promoting reuse, and reducing development time while maintaining relatively low computational requirements [22].

Code similarity analysis also has important implications for software security. Detecting code patterns associated with known vulnerabilities can support vulnerability discovery and mitigation efforts. In addition, similarity assessment can facilitate maintainability studies, refactoring activities, and long-term software evolution.

Beyond traditional software engineering tasks, similarity measures can support compliance verification and reliability assessment in safety-critical domains. They can also improve collaboration by helping developers discover related implementations and can strengthen quality assurance processes through the identification of similar code segments requiring consistent testing.

3 Methods

Early approaches to source code similarity assessment relied primarily on textual analysis techniques [17]. Although computationally efficient, these methods often fail to capture structural and semantic properties of source code, which limits their effectiveness [21]. Recent advances have introduced more sophisticated similarity measures that incorporate richer representations of code and generally achieve higher detection performance [6].

3.1 Unsupervised Methods

A wide range of semantic similarity measures has been proposed for comparing textual entities. These methods differ in the representations they employ and the aspects of similarity they capture. Our survey identified 21 families of unsupervised approaches that can be applied to source code similarity assessment. The following sections briefly describe these families in alphabetical order.

- **Abstract Syntax Tree (AST) Similarity**: AST-based approaches compare the structural representations of source code by analyzing similarities between their abstract syntax trees [14].
- **Bag-of-Words Similarity**: This measure represents code as a collection of tokens and estimates similarity from their frequencies, disregarding token order and syntactic structure [4].
- **Code Embedding Similarity**: These methods represent source code as dense vector embeddings and quantify similarity using distances or correlations between the resulting vectors [1].
- **Comments Similarity**: This family evaluates similarity between code comments and documentation. Traditional text similarity measures can be directly applied in this setting [23].
- **Fuzzy Matching Similarity**: Fuzzy matching techniques compare strings while tolerating minor syntactic variations. Although commonly used in record linkage and search applications, they can also be applied to source code comparison [27].
- **Function Call Similarity**: These methods assess similarity according to the functions and procedures invoked within code fragments [29].
- **Graph-Based Similarity**: Graph-based approaches represent source code as graphs and estimate similarity from relationships and dependencies captured in the graph structure [30].
- **Jaccard Similarity**: Jaccard similarity compares sets of tokens by measuring the ratio between their intersection and union. It is widely used in text mining and information retrieval [8].
- **Levenshtein Similarity**: Also known as edit-distance similarity, this measure quantifies resemblance according to the number of insertions, deletions, and substitutions required to transform one sequence into another [16].
- **Longest Common Subsequence (LCS) Similarity**: LCS-based methods estimate similarity by identifying the longest subsequence shared by two sequences while preserving element order [3].
- **Metrics Similarity**: These approaches compute software metrics for code fragments and estimate similarity from the resulting metric values [24].
- **N-Gram Similarity**: N-gram methods compare contiguous sequences of n tokens or characters and measure similarity according to the degree of overlap between the extracted n-grams [5].
- **Output Analysis Similarity**: These methods compare the outputs generated by programs. When outputs are textual, conventional text similarity techniques can be employed [23].

- **Perceptual Hashing Similarity**: Originally developed for image comparison, perceptual hashing generates compact representations that preserve visual characteristics. In this context, it is applied to visual representations of source code [25].
- **Program Dependence Graph Similarity**: These techniques compare program dependence graphs that capture data and control dependencies among program elements [15].
- **Rolling Hash Similarity**: Rolling hash functions allow efficient computation of hash values over sliding windows of data. Similarity is estimated through the comparison of hashed code fragments [9].
- **Running-Karp-Rabin Greedy-String-Tiling (RKR-GST) Similarity**: This approach identifies maximal contiguous matching token sequences, known as tiles, and is widely used in plagiarism detection [28].
- **Semdiff Similarity**: Semdiff-based methods evaluate similarity according to the semantic impact of code changes across program versions [11].
- **Semantic Clone Similarity**: These methods exploit the semantic meaning of identifiers, method names, and other program elements to estimate similarity between code fragments [7].
- **TF-IDF Similarity**: Term Frequency-Inverse Document Frequency (TF-IDF) weights tokens according to their importance within a corpus and compares code fragments using the resulting weighted representations [12].
- **Winnowing Similarity**: Winnowing generates fingerprints from token sequences and compares these fingerprints to identify similar code fragments efficiently [26].

Next, we illustrate the behavior of these similarity measures using representative Java code examples that capture challenging clone-detection scenarios.

3.2 Examples

In the examples presented below, *T1* and *T01* are two Java classes that generate identical outputs through different implementations. Class *T1* prints the message *Welcome to Java* five times using five individual *print* statements, whereas class *T01* produces the same output through a *for* loop that executes five iterations. According to the clone taxonomy discussed earlier, these classes correspond to Category IV clones because they implement the same functionality despite substantial syntactic differences.

Although the source code structures differ considerably, with one implementation relying on repeated statements and the other on an iterative construct, both classes exhibit identical observable behavior. Detecting this type of clone is particularly difficult because textual and structural similarities are limited. Effective identification therefore requires methods capable of capturing semantic equivalence and program behavior. Such situations are common in practice, where developers frequently implement the same functionality using different programming constructs, coding styles, and design decisions.

```java
public class T1 {
    public static void main(String[] args) {
        System.out.println("Welcome to Java");
        System.out.println("Welcome to Java");
        System.out.println("Welcome to Java");
        System.out.println("Welcome to Java");
        System.out.println("Welcome to Java");
    }
}
```

```java
public class T01 {
    public static void main(String[] args){

        for(int i = 0; i < 5; i++){
            System.out.println("Welcome To Java");
        }

    }
}
```

On the contrary, the classes *TemperatureConverter* and *CurrencyConverter* are similar in form. However, an experienced developer would quickly realize that they calculate very 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.

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

```java
public class CurrencyConverter {
    public static double usdToEur(double usd) {
        return usd * 85 / 100;
    }
}
```

Table 1 presents a comparison of the unsupervised similarity measures evaluated in this study. These measures rely on different principles, including textual similarity, structural characteristics of the source code, and behavioral or semantic properties that extend beyond textual and syntactic representations. Since the measures produce similarity scores rather than direct classifications, there is no inherently correct or incorrect result. Instead, their usefulness depends on their ability to distinguish clone pairs from non-clone pairs.

Ideally, a similarity measure would assign values close to 1.00 to clone pairs and values close to 0.00 to non-clone pairs, resulting in complete separation between the two groups. In practice, perfect separation is rarely achievable. There-

fore, the quality of a measure should be assessed according to its discriminative power, namely, its capacity to consistently assign high similarity scores to clones and low similarity scores to non-clones.

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 [13]. 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 it does not classify clones.

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.

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 important to note that the dataset is highly imbalanced, with clones representing approximately 77% of all instances. Consequently, a trivial classifier that labels every comparison as a clone would achieve an accuracy of 0.77. Such a result would provide little practical value, as it would fail to distinguish between clone and non-clone pairs. As shown in Figure **??**, only three measures achieve

[1] `https://github.com/oscarkarnalim/sourcecodeplagiarismdataset`

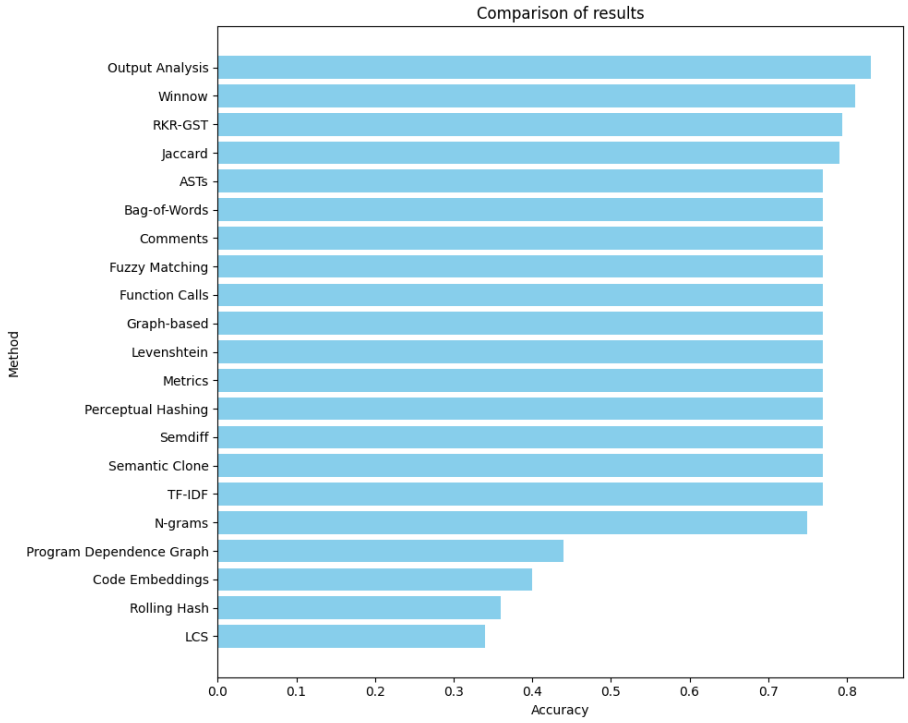

Fig. 1: Accuracy of the unsupervised semantic similarity measures when performing clone detection

a substantial improvement over this baseline, demonstrating a genuine ability to discriminate between the two classes.

Figure **??** presents a comparison of the execution times associated with the evaluated similarity measures. The horizontal axis represents execution time, whereas the vertical axis lists the corresponding unsupervised measures. Among all approaches, *Output Analysis* exhibits the highest computational cost, with execution times considerably greater than those of alternatives such as *N-grams* and *Code Embeddings*. Most of the remaining measures require substantially less processing time, indicating better computational efficiency and greater suitability for large-scale applications.

Two key observations can be drawn from these results:

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.

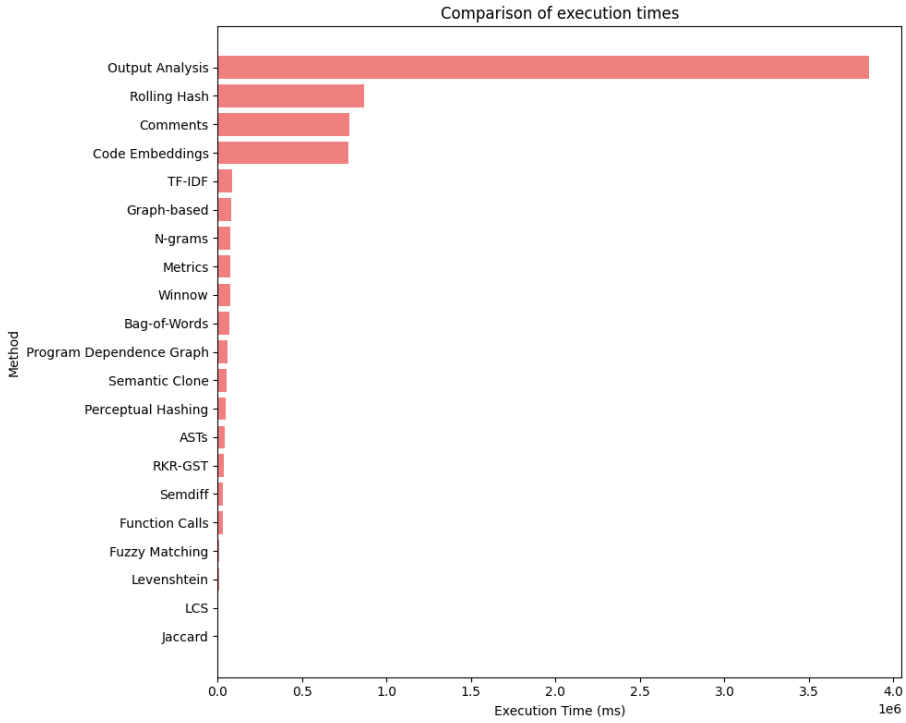

Fig. 2: Execution time of the unsupervised semantic similarity measures when performing clone detection

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 whole process is very computationally expensive.

Other computationally demanding measures include *Rolling Hash*, which requires a large number of mathematical operations; *Comments Similarity*, which relies on regular-expression processing to identify and analyze comments; and *Code Embeddings*, which involves generating, retrieving, and comparing vector representations of source code. Given these trade-offs, it would be useful to define a feasibility index that jointly considers detection effectiveness and execution time. Such an index could provide a more realistic assessment of a measure's suitability for practical deployment. One possible formulation would assign a configurable weight to accuracy relative to execution time and normalize the resulting score by the total execution cost.

Figure ?? presents the feasibility index used in this study. We assign a relative importance of 10:1 to accuracy over execution time. Under this criterion, *Jaccard*, *RKR-GST*, and *Winnow* emerge as the most promising candidates for real-world adoption due to their favorable balance between effectiveness and computational efficiency. Nevertheless, their performance gains over the baseline remain modest, suggesting that they are best suited as recommendation mechanisms that support human decision-making rather than fully autonomous clone detection solutions.

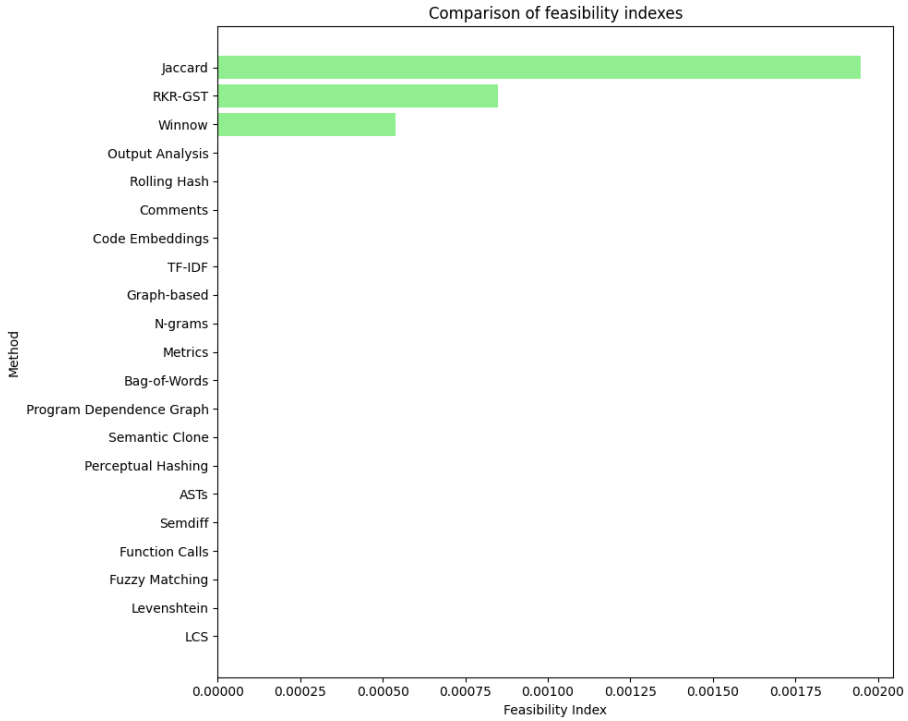

Fig. 3: Comparison of the feasibility index of the unsupervised methods

5 Discussion

Our experiments indicate that a relatively small set of unsupervised source code similarity measures can effectively support source code clone detection. These measures have the potential to benefit several software engineering activities. For example, they can help identify duplicated code fragments, creating opportunities for refactoring and the extraction of reusable components. This, in

turn, can promote code reuse, which remains a fundamental practice in software development.

It is important to note that real-world software projects are often characterized by noisy and heterogeneous code environments. Our evaluation identified several unsupervised measures that perform well under such conditions, demonstrating resilience to variability in coding styles and implementation patterns. These characteristics make them particularly valuable when labeled data are scarce, unavailable, or costly to obtain.

Despite these encouraging results, several challenges remain. These include improving cross-language similarity assessment and achieving scalability for increasingly large codebases. Addressing these issues represents an important direction for future research. Although the measures investigated in this study show practical potential, further advances are required before they can fully support the demands of increasingly automated software development processes.

6 Conclusion

Source code clone detection is a longstanding software engineering problem with implications for software maintenance, quality assurance, refactoring, and code reuse. In this study, we evaluated existing unsupervised similarity measures as a means of addressing the limitations associated with labeled datasets and the diversity of coding practices. Our results demonstrate that unsupervised similarity measurement can provide an effective foundation for clone detection. To provide a broad and balanced assessment, we evaluated representative variants of the major families of unsupervised approaches, offering practical guidance for software engineers and researchers.

As software systems continue to increase in size and complexity, the development of accurate and efficient unsupervised similarity measures remains an important research objective. The relevance of these techniques is likely to grow alongside the software industry's increasing reliance on automation and large-scale code generation. Although supervised approaches often achieve strong predictive performance, unsupervised methods remain attractive due to their reduced dependence on labeled data, adaptability across domains, interpretability, and lower deployment costs.

The future of unsupervised source code clone detection is therefore promising. One potential direction involves hybrid methods that combine complementary similarity measures into ensemble-based solutions capable of producing more reliable assessments. Another avenue is the incorporation of transfer learning techniques to improve generalization across programming languages and application domains. Ultimately, future research should aim to develop code analysis techniques that deliver high accuracy while requiring minimal human intervention.

References

1. 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.
2. 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.
3. 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.
4. 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.
5. Marc Damashek. Gauging similarity with n-grams: Language-independent categorization of text. *Science*, 267(5199):843–848, 1995.
6. 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.
7. 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.
8. 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.
9. 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.
10. 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.
11. 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.
12. Oscar Karnalim. Tf-idf inspired detection for cross-language source code plagiarism and collusion. *Computer Science*, 21, 2020.
13. 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.
14. 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.

15. Jens Krinke. Identifying similar code with program dependence graphs. In *Proceedings Eighth Working Conference on Reverse Engineering*, pages 301–309. IEEE, 2001.
16. Vladimir I Levenshtein. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, volume 10, pages 707–710, 1966.
17. 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.
18. Jorge Martinez-Gil. A comprehensive review of stacking methods for semantic similarity measurement. *Machine Learning with Applications*, 10:100423, 2022.
19. 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.
20. Jorge Martinez-Gil. Augmenting the interpretability of graphcodebert for code similarity tasks. *arXiv preprint arXiv:2410.05275*, 2024.
21. Jorge Martinez-Gil. Source code clone detection using unsupervised similarity measures. In *International Conference on Software Quality*, pages 21–37. Springer, 2024.
22. 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.
23. 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.
24. 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.
25. 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.
26. 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.
27. 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.
28. Michael J Wise. String similarity via greedy string tiling and running karp-rabin matching. *Online Preprint, Dec*, 119(1):1–17, 1993.
29. 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.
30. Laura A Zager and George C Verghese. Graph similarity scoring and matching. *Applied mathematics letters*, 21(1):86–94, 2008.