Evaluation of Code Similarity Search Strategies in Large-Scale Codebases

Jorge Martinez-Gil[1], Shaoyi Yin[2]

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

[2]Paul Sabatier University, IRIT Laboratory
118 route de Narbonne, Toulouse, France
shaoyi.yin@irit.fr

Abstract. The ability to automatically identify similar code fragments within huge code repositories is crucial for software development and maintenance tasks such as code reuse and debugging. Although several solutions already exist to face this challenge, not many comparisons have yet been established. For this reason, this study presents a comparative analysis of existing and emerging techniques for code similarity search. We benchmark these methods across diverse codebases, examining metrics such as indexing time, search speed, and the semantic relevance of retrieved code fragments. Our research aims to provide software developers with practical information for performing efficient code similarity searches, addressing the challenges associated with the increasing size of codebases.

Keywords: Code Clone Detection, Code Similarity Search, Codebase Management, Code Reuse, Similarity Search

1 Introduction

Managing large-scale software projects presents several challenges regarding the efficacy and efficiency of the software development process. One significant issue is the potential for duplicated effort [5]. Developers might discover similar code used for comparable tasks as development teams expand and projects increase in complexity. This redundancy wastes valuable time and resources, complicating code maintenance [22]. Additionally, finding relevant code within a large codebase can take much work. Software developers often spend considerable time discovering the specific fragments to modify or extend. These challenges show a need for solutions that maintain consistent code across large-scale projects [21].

Traditional code search methods, such as those using regular expressions, have long been used for searching code fragments within repositories. However, these methods primarily focus on literal string matching, often needing to catch up in capturing the semantic similarity between code fragments. This limitation becomes particularly clear in large codebases where different implementations of similar functionalities may not share exact textual features, so the community widely agrees on the need for more advanced solutions [20].

In recent times, new approaches that generate vector embeddings of code fragments are more appropriate for matching code based on its underlying functionality and not just its textual resemblance. These new approaches allow for more accurate results and faster processing or retrieval of results from large code repositories. Therefore, these techniques provide a novel capability to improve efficiency and decrease redundancy. However, existing solutions still need to be systematically compared. The present study fills this gap, and therefore, the main contributions of this work are as follows:

– An overview of classical and emerging techniques for performing similarity searches in large codebases.
– An empirical evaluation of these techniques with special interest in those relying on semantic code representations rather than literal matching.

The rest of this work is structured fully to ensure a thorough understanding of our research. Section 2 provides a detailed review of existing literature regarding code similarity search. Section 3 presents the methodology usually employed in code similarity search and how we will adapt it to perform our empirical evaluation. In Section 4, we perform the experimental setup and present the results regarding the effectiveness and efficiency of existing approaches concerning several benchmarks. Finally, we offer a summary of key findings and potential future directions of this research, providing a reliable roadmap for further exploration.

2 State-of-the-Art

Code similarity search is highly beneficial in several practical scenarios. For example, code reuse involves identifying already-written components, allowing developers to improve efficiency by integrating existing solutions into new projects [14]. In debugging, these techniques can accelerate the process by finding code that has addressed similar bugs or issues in the past, providing a practical reference for developers [11]. Moreover, such techniques facilitate the exploration of different implementations and variations for code understanding, helping developers to identify diverse coding approaches and optimize their solutions [13].

The techniques for code similarity search within large codebases have evolved significantly in recent years by integrating novel techniques [23]. Advanced approaches now go beyond traditional text-based searches to understand the semantic content of code [1]. Techniques like summarization [10] and embedding source code into high-dimensional vector spaces allow systems to assess similarity based on the functionality rather than just textual similarity. Approaches like CodeBERT [4] and GraphCodeBERT [7] are examples of solutions that learn contextual relationships within code, improving the ability to detect similar patterns across diverse codebases. These strategies improve the search accuracy in large repositories, enabling developers to find functionally similar code fragments with different syntactic presentations [11].

Furthermore, the scalability of code similarity search systems has been a critical focus, given the exponential growth of source code in recent years. Systems are now designed to handle vast repositories efficiently, using technologies such as distributed computing, efficient indexing mechanisms, and other tools [12]. One of the most widely used techniques is FAISS [3], which uses a quantization-based approach to compress vectors and speed up the similarity search process without significant losses in precision.

However, more approaches allow for the implementation of near-real-time code search systems [2]. This is because solutions of this kind are crucial for modern software development environments, where developers and even agents need to reuse existing code. These advancements reduce the duplication of effort and contribute to faster development cycles.

Unfortunately, very few empirical studies attempt to rigorously compare the performance of the different emerging solutions for code similarity search. Our work sheds some light on this and provides an overview of what can be expected from these emerging solutions.

3 Problem Statement

Let C_1 and C_2 be source code fragments in two different programming languages, L_1 and L_2, respectively. The code similarity problem involves defining a function $S(C_1, C_2)$ that quantifies their similarity.

Therefore, given $C_1 = \{c_{1,1}, c_{1,2}, \ldots, c_{1,n}\}$ and $C_2 = \{c_{2,1}, c_{2,2}, \ldots, c_{2,m}\}$, where $c_{1,i}$ and $c_{2,j}$ are the atomic elements (such as tokens, statements, abstract syntax tree nodes, etc.) of the code fragments in languages L_1 and L_2, respectively; the similarity function $S(C_1, C_2)$ is a mapping:

$$S : \mathcal{C}_{L_1} \times \mathcal{C}_{L_2} \rightarrow [0, 1]$$

where $\mathcal{C}_{L_1}$ and $\mathcal{C}_{L_2}$ are the sets of all possible code fragments L_1 and L_2, respectively, and $S(C_1, C_2) = 1$ indicates maximum similarity and $S(C_1, C_2) = 0$ indicates no similarity.

The function S may be defined based on various criteria, such as syntactic, semantic, or structural likeness between C_1 and C_2, and often involves complex algorithms or machine learning models for its computation [21].

Several effective techniques and tools have been developed to tackle this problem [19]. However, we are only interested in those able to vectorize the code in order to study its scalability with techniques that include Annoy [24], Elasticsearch [6], FAISS [3], HNSW [17], ScaNN [8], and Scikit-Learn-NN [9]. Each technique and tool has some characteristics that suit different scenarios. However, comparing their performance in objective aspects has yet to be studied. Figure 1 illustrates a top-k similarity search in a codebase, focusing on a specific query code fragment labeled *Q. Code Fragment*. The diagram uses nodes representing different code fragments and edges to denote the similarity percentage between the query fragment and these fragments.

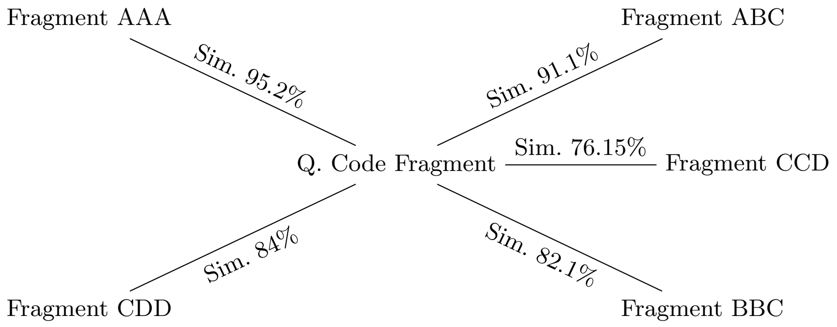

Fig. 1: Example of top-k similarity search in a codebase

To date, some of the most outstanding proposals in this area are listed below in alphabetical order:

– Annoy (Approximate Nearest Neighbors Oh Yeah) [24] that performs approximate nearest-neighbor search. It builds forests of trees to partition the space and allows for fast querying, even under very high-dimensional data. Its main advantage is its ability to use memory-mapped files and efficiently handle large-scale datasets.
– Elasticsearch [6] that, although primarily a search engine, can be used for code similarity search through its tunable vector scoring features. It supports text-based and vector-based searches, handling various forms of code and plain language queries. It is particularly advantageous when combining code similarity search with other search capabilities (e.g., text) is essential.
– FAISS (Facebook AI Similarity Search) [3] that is designed for efficient similarity search of high-dimensional vectors. It is beneficial for searching spaces with huge dimensionality, such as embeddings derived from source code. It supports several indexing strategies that optimize speed and accuracy, making it suitable for massive datasets.
– HNSW (Hierarchical Navigable Small World) [17] is an approximate nearest-neighbor search algorithm that uses hierarchical graph structures to efficiently search high-dimensional data. It is known for achieving good recall rates, even in large-scale environments, which is beneficial for code similarity searches where exact matches are not always necessary.
– ScaNN (Scalable Nearest Neighbors) [8] that improves the efficiency of nearest-neighbor computation in high-dimensional spaces using a combination of quantization and tree-based partitioning. It can be tailored to balance accuracy and speed, which is useful when dealing with large codebases.

– SKLNN (Scikit-learn Nearest Neighbors) [9] that supports various algorithms for nearest-neighbor searches, each dealing with different sizes and dimensionalities. While efficient for small to medium datasets, scalability is assumed to be limited compared to specialized approaches when handling high-dimensional data.

From now on, we will determine the aspects in which these approaches are better suited, which could help give clues as to their use in software development.

4 Main Steps of the Code Similarity Search Process

Modern code similarity search involves several steps, starting with data preprocessing. This includes collecting, cleaning, and normalizing code fragments. Next, these code fragments are transformed into numerical representations through vectorization techniques like TF-IDF [15], or CodeBERT [4]. These vectors are then indexed to allow fast and accurate similarity searches in large code repositories [25]. When a user submits a query, it is also vectorized, and the index is used to retrieve the most similar code fragments efficiently. A critical factor in this search process is k, which determines the number of nearest neighbors returned. Adjusting the k value helps provide results that meet the user's specific needs.

4.1 Data Preprocessing

Effective data preprocessing is crucial for setting up a code similarity search. Initially, code fragment collection can be performed by scraping code repositories, utilizing public dataset compilations, or extracting them from internal project archives. Once collected, code fragments often require cleaning and normalization to ensure consistency and improve effectiveness. This may involve removing comments, normalizing variable names, or standardizing coding styles. Such preprocessing steps help reduce noise and improve the focus on the functional aspects of the code, which are essential for effective similarity searches.

4.2 Vectorization of Code Fragments

Although it is always possible to compare fragment-by-fragment similarity [23], this usually scales poorly. Therefore, vector representations of the fragments are used. Several techniques cover this phase, and the accuracy of the search system depends on the choice. However, since this work focuses on performance rather than accuracy, we will only focus on two techniques: a classical one (TF-IDF) and an emerging one (CodeBERT).

– TF-IDF for representing code fragments presents the ability to assess the importance of terms uniquely relevant to a particular fragment while penalizing standard terms [15]. When working with source code, terms (which

could be keywords, function names, or API calls) that appear frequently in a fragment but are rare across other fragments are weighted higher, thus capturing the uniqueness of the fragment. This feature makes TF-IDF particularly suited for distinguishing code fragments that implement specific functionalities, making it a helpful approach for code similarity search [16].
– CodeBERT [4] is a language model designed to process source code automatically. It is trained on a massive dataset of natural language and programming language text, enabling it to perform very well at tasks like code search, completion, and summarization. CodeBERT's bimodal architecture bridges the gap between natural language and code, making it useful for developers who can work with abstract representations of code that are very helpful in a wide range of programming-related tasks.

4.3 Indexing

The need for efficient and scalable similarity searches in large datasets drives the choice of a good indexing strategy. This strategy must be optimized for fast similarity computations over large sets of high-dimensional vectors, making it ideal for handling the vectorized form of code fragments. Then, some kind of distance (e.g., euclidean, cosine, etc.) is used to calculate the similarity between vectors, providing an efficient way to assess the likeness of code fragments based on their vector representations. This setup is particularly effective in environments where fast query responses are crucial, and the dataset size can be huge.

4.4 Similarity Search

During the similarity search process, query code fragments are first vectorized using the same scheme applied during the preprocessing stage. These vectorized forms are fed into an index to find similar code fragments. In this context, the parameter k refers to the number of nearest neighbors considered in the search results. Adjusting k can impact the outcome; a larger k might include more potentially relevant results but also increase the noise, whereas a smaller k focuses on the most similar fragments but may miss some relevant matches. The choice of k thus needs to be balanced based on the specific needs of the search process.

5 Evaluation

In our code similarity search experiments, we have investigated the performance of various techniques and tools under different scenarios. We measured indexing time to assess the efficiency of creating searchable representations of codebases. Accuracy has been evaluated using traditional TF-IDF and the more recent CodeBERT model based on deep learning.

To assess search performance, we have conducted queries on codebases of increasing size: 10,000, 100,000, and 1,000,000 code fragments. We have measured the time taken to retrieve relevant results for each query, providing insights into the scalability of different methods and assessing the efficiency of the different approaches under different workloads.

5.1 Assessment Methodology

A qualitative assessment can provide examples of results showcasing the system's ability to find conceptually similar codes. For instance, a search query for a sorting algorithm could return various implementations of quicksort or mergesort, demonstrating the system's capability to recognize a range of related algorithms. However, this assessment is currently not possible due to the lack of a dataset that has such information annotated, so for the time being, only a quantitative comparison is possible.

Traditionally, code similarity search strategies are compared against simple baselines like keyword searches or basic code representations. However, modern techniques often use vector representations of code, which capture semantic meaning better than keywords. This study compares different vector-based strategies to determine the most effective and efficient way to improve code representation.

5.2 Experimental Setup

In this study, we use the dataset of the BigCloneBench[1] to perform code similarity search. This dataset is a valuable resource for addressing the challenge of identifying duplicate code. Comprising diverse programming languages sourced from real-world software and student projects, it provides a realistic testing ground for evaluating code similarity search strategies. Due to its nature, BigCloneBench is frequently used in solutions to improve code maintainability.

We will always look for the three most similar code fragments. Furthermore, for the parameters of the different strategies, we rely on their default settings, i.e.:

– *Annoy*: Angular distance, 10 trees
– *Elasticsearch*: SVD components 4096, Cosine Similarity
– *FAISS*: FlatL2 index
– *HNSW*: L2 distance, ef_construction: 200, M: 16
– *ScaNN*: L2 normalization, Number of Leaves: 10, Anisotropic Quantization Threshold: 0.2
– *Scikit-Learn NN*: Algorithm K-D Tree

Moreover, the experiments have been performed in an isolated machine on Windows 11 running over an 11th Gen Intel(R) Core(TM) i7-1185G7 at 3.00GHz. In any case, the reported results are always the result of a relative comparison performed on the same hardware.

[1] https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/Clone-detection-BigCloneBench

5.3 Performance Comparison

We proceed now to show the experiments performed to establish the performance comparison. These experiments have been designed to evaluate and benchmark the accuracy, and scalability of the different search strategies under various conditions regarding the volume of data to be handled.

Indexing Time Figure 2 presents a comparative summary of the time required to index code fragments using various approaches for different dataset sizes: 10k, 100k, and 1M code fragments. The table provides an overview of the performance of the six different indexing approaches under consideration in this study.

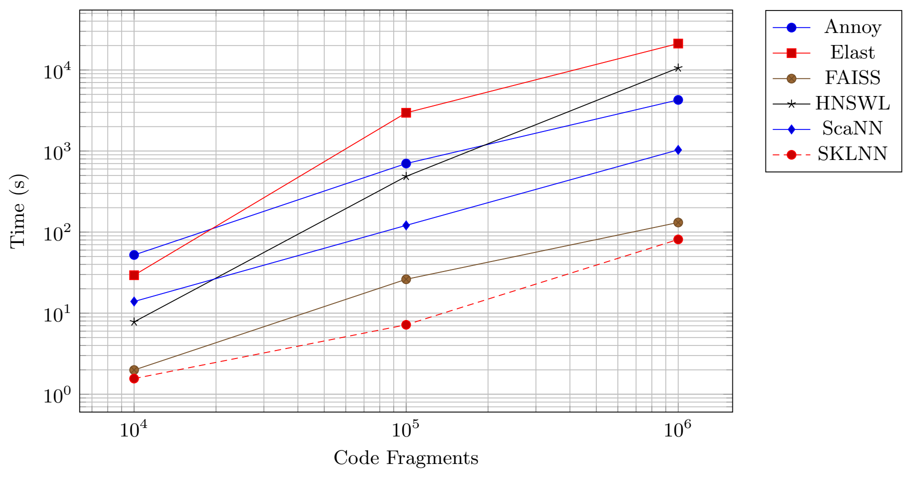

Fig. 2: Performance comparison of different methods for 10k, 100k, and 1M code fragments

All techniques show an expected increase in indexing time as the dataset size grows. The values for each technique suggest varying degrees of scalability. However, FAISS, which shows a relatively smaller increase in time, might indicate better scalability than others in the long run. This is deduced from the lower slope inclination, which will cause it to increase at a lower rate than the rest.

It is also necessary to note that all indexes (except Elasticsearch) are kept in the main memory. Only Elasticsearch, as a complete DBMS, works directly on optimized disk files and is consolidated in secondary memory.

Accuracy Code similarity search aims to find duplicate or similar code fragments but involves a trade-off between accuracy and efficiency. The chosen algorithm and parameters can affect the results. Exact matching ensures high precision but may overlook functionally similar code with different structures. In contrast, approximate methods like token-based or semantic analysis cover more ground but can produce false positives. The quality and relevance of training data also affect the accuracy of machine learning-based approaches.

Figure 3 illustrates the relationship between the number of code fragments and accuracy using TF-IDF over FAISS. The x-axis represents the number of code fragments on a logarithmic scale, and the y-axis represents the accuracy. The figure shows that the accuracy tends to decrease as the number of code fragments increases. This downward trend suggests a negative correlation between the number of code fragments and the accuracy in this context.

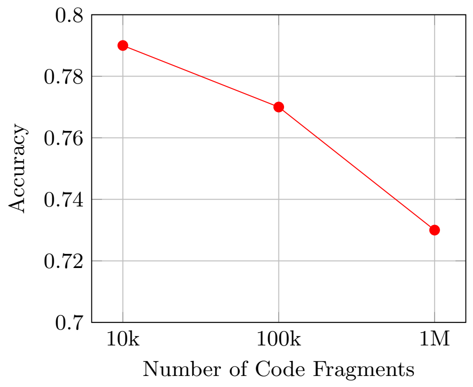

Fig. 3: Search accuracy using TF-IDF

Figure 4 illustrates the relationship between the number of code fragments and accuracy using CodeBERT over FAISS. The x-axis represents the number of code fragments on a logarithmic scale, and the y-axis represents the accuracy. The figure shows that the accuracy remains relatively stable as the number of code fragments increases. There is a slight decrease in accuracy when the number of code fragments reaches 1,000,000, but overall, the accuracy stays high, indicating a good performance across different scales.

However, this work does not seek to compare semantic similarity models. Previous studies already exist and even show that some variants of CodeBERT [7] and an ensemble of simple models for assessing semantic similarity tend to perform well [18].

Search Performance Search performance refers to how fast a given approach can compare source code fragments. Faster performance is crucial for real-time or large-scale applications. We have studied this performance for 10k, 100k, and 1M code fragments. Times were measured five times, and the median value was reported.

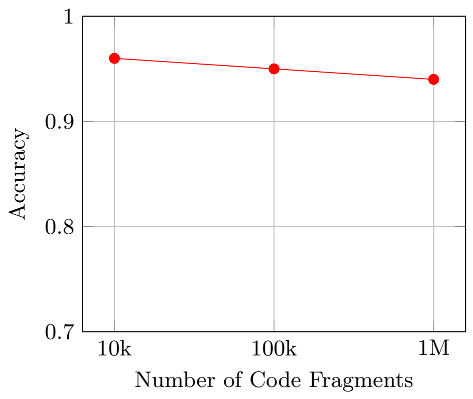

Fig. 4: Search accuracy using CodeBERT

Experiment 1: 10,000 items. Figure 5 shows us that the query performance of the different approaches varies significantly. The comparison shows how some approaches maintain lower search times consistently, which is critical for applications requiring efficient code retrieval. This is important to avoid potential bottlenecks in systems where fast code retrieval is essential, such as in integrated development environments or real-time code analysis tools.

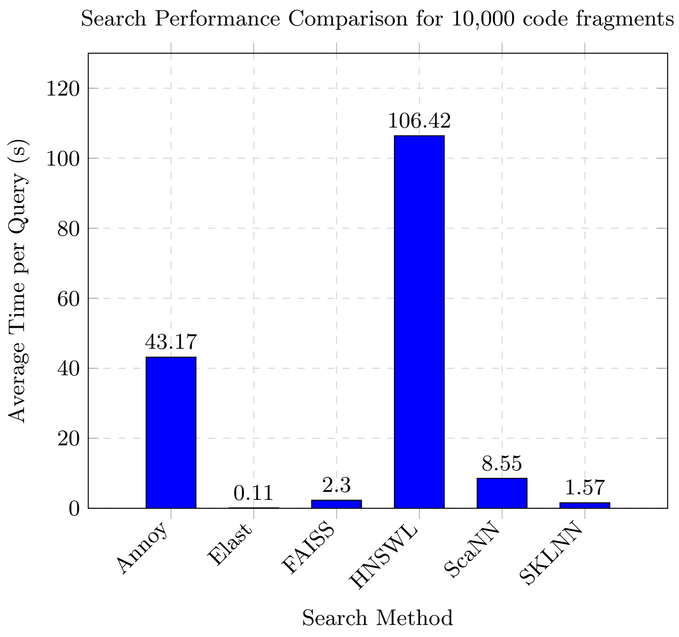

Fig. 5: Search performance for a codebase of 10,000 items

Experiment 2: 100,000 items. Figure 6 shows us that the query performance of different approaches varies again. The performance trend observed with 10,000 items continues here, with specific approaches (Elasticsearch, SKLNN, FAISS) demonstrating better scalability as the dataset size increases. So, they could be good candidates when choosing the right strategy for large-scale data, assuring that query performance remains efficient even as data volume grows. Therefore, we can see which methods could be suitable for handling expansive datasets to guarantee reliable performance in large-scale applications.

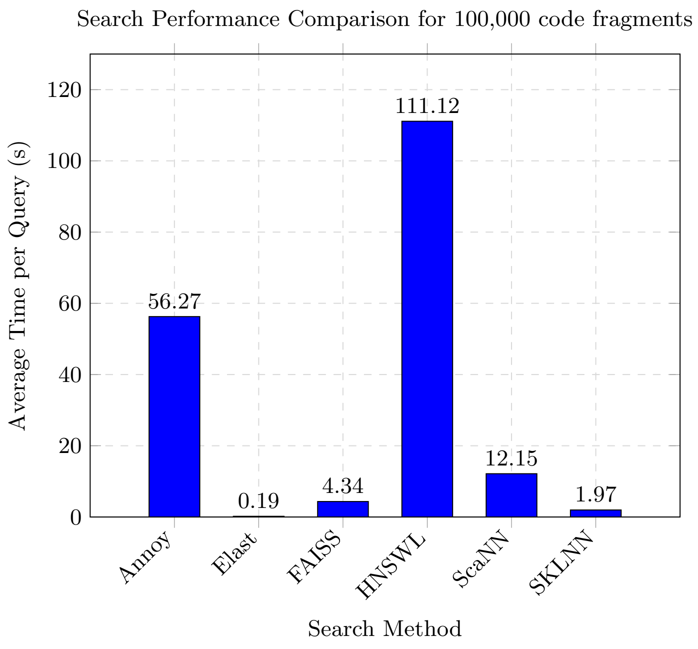

Fig. 6: Search performance for a codebase of 100,000 items

Experiment 3: 1,000,000 items. Figure 7 shows us again that the query performance of the different approaches varies a lot. This figure shows us the importance of choosing the right approach for large-scale applications, as the difference in search times can be substantial. Therefore, it seems clear that selecting an efficient method is crucial for maintaining stable performance in systems with large datasets. The rationale behind opting for the right solution is to improve system responsiveness and user satisfaction even as the code volume grows.

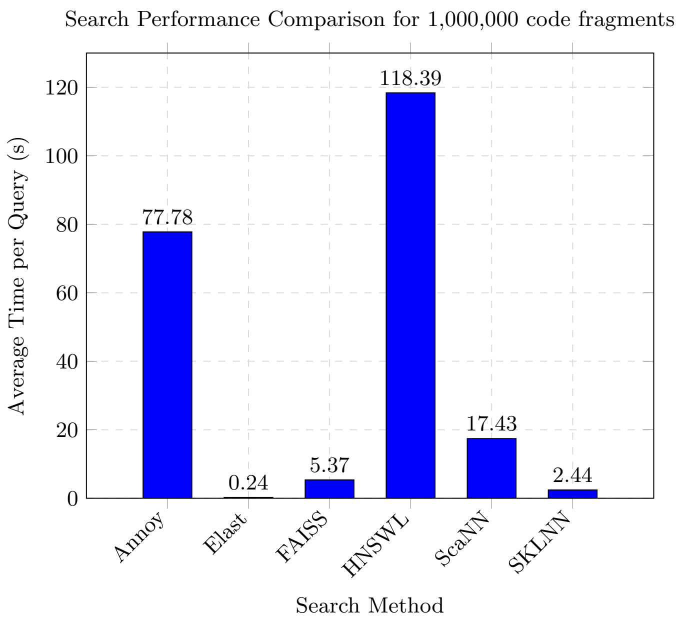

Fig. 7: Search performance for a codebase of 1,000,000 items

5.4 Summary of Results

The results of our experiments reveal some significant findings that could be interesting when guiding developers and researchers in selecting the most suitable strategies for their specific use cases and requirements regarding code similarity search. The following facts can be extracted from these results:

- CodeBERT is the best strategy to maximize accuracy. These results confirm the findings that have also been obtained in the frame of several previous research studies [7, 19].
- Elasticsearch produces the best results in terms of search performance but at the cost of having the worst indexing time.
- The approach with the best scalability prospects, in the long run, is FAISS, and in the short term, is SKLNN.
- Most approaches present good performance, making them viable candidates for integration into production systems, although Elasticsearch is recommended to consolidate the index in secondary memory.
- The results of SKLNN are positively surprising since it is part of the general-purpose Scikit-learn library, which is very popular in the software development world. In fact, for small samples, it has very good indexing and search speed times.

6 Discussion

Code similarity search helps developers identify algorithm variations, aiding in code understanding and optimization. Identifying similar code fragments across different projects or within the same codebase facilitates developers' efficient search for duplicated code fragments, which is crucial for improving their projects' maintainability.

The process of vectorizing code enables code similarity search to effectively capture the essence of the code, regardless of varying structures and styles. Vectorization transforms code into a numerical representation, allowing the different strategies for code similarity search to analyze and compare code fragments more effectively.

However, despite their advantages, code similarity search strategies face challenges, such as complex syntax and dependencies in programming languages. Effective vectorization must capture both syntactic and semantic aspects of code, which can be difficult due to varying structures and styles. Furthermore, vocabulary mismatch is another issue. Different naming conventions and synonyms can lead to inaccurate search results.

We have seen that as codebases grow, search performance can degrade. Efficient indexing and retrieval mechanisms are necessary to maintain high performance in large-scale applications. Addressing these limitations is crucial to improving code similarity search. Advances in this context offer promise in overcoming some challenges by developing models that better understand code context and semantics. Standardized naming conventions and coding practices could also reduce vocabulary mismatches and improve search accuracy.

7 Conclusion

This study has evaluated existing code similarity search techniques, focusing on their effectiveness and scalability in real-world scenarios. Our benchmark results demonstrate that the studied methods are suitable for performing code similarity searches, although with some objections. The key idea is capturing the functional essence of the code beyond mere textual similarity, improving the accuracy of search results within large codebases. This approach has proven invaluable for developers, enabling them to efficiently identify functionally similar but syntactically diverse code fragments.

We have established a comparative analysis to identify the best existing approaches to working in a context where codebases grow rapidly. Our experiments show that most approaches offer good average performance. However, some are much better than the rest in some aspects related to the results indexation, search speed, and semantic relevance.

Furthermore, we have identified several areas for future research. Developing more efficient indexing techniques could improve the scalability of code similarity search, particularly for large codebases. Exploring novel approaches to capture the semantic meaning of code could improve the relevance of retrieved code fragments and optimize the search process.

Acknowledgments

The authors thank the anonymous reviewers for their help in improving the work. The research reported in this paper has been funded by the Federal Ministry for Climate Action, Environment, Energy, Mobility, Innovation, and Technology (BMK), the Federal Ministry for Digital and Economic Affairs (BMDW), and the State of Upper Austria in the frame of SCCH, a center in the COMET - Competence Centers for Excellent Technologies Programme managed by Austrian Research Promotion Agency FFG.

References

1. Q. U. Ain, W. H. Butt, M. W. Anwar, F. Azam, and B. Maqbool. A systematic review on code clone detection. *IEEE access*, 7:86121–86144, 2019.
2. M. Aumüller, E. Bernhardsson, and A. Faithfull. Ann-benchmarks: A benchmarking tool for approximate nearest neighbor algorithms. *Information Systems*, 87:101374, 2020.
3. M. Douze, A. Guzhva, C. Deng, J. Johnson, G. Szilvasy, P.-E. Mazaré, M. Lomeli, L. Hosseini, and H. Jégou. The faiss library. 2024.
4. Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, and M. Zhou. Codebert: A pre-trained model for programming and natural languages. In T. Cohn, Y. He, and Y. Liu, editors, *Findings of the Association for Computational Linguistics: EMNLP 2020, Online Event, 16-20 November 2020*, volume EMNLP 2020 of *Findings of ACL*, pages 1536–1547. Association for Computational Linguistics, 2020.
5. M. Gabel, L. Jiang, and Z. Su. Scalable detection of semantic clones. In *Proceedings of the 30th international conference on Software engineering*, pages 321–330, 2008.
6. C. Gormley and Z. Tong. *Elasticsearch: the definitive guide: a distributed real-time search and analytics engine.* " O'Reilly Media, Inc.", 2015.
7. D. Guo, S. Ren, S. Lu, Z. Feng, D. Tang, S. Liu, L. Zhou, N. Duan, A. Svyatkovskiy, S. Fu, et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.
8. R. Guo, P. Sun, E. Lindgren, Q. Geng, D. Simcha, F. Chern, and S. Kumar. Accelerating large-scale inference with anisotropic vector quantization. In *International Conference on Machine Learning*, pages 3887–3896. PMLR, 2020.
9. G. Hackeling. *Mastering Machine Learning with scikit-learn.* Packt Publishing Ltd, 2017.
10. S. Haque, Z. Eberhart, A. Bansal, and C. 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.
11. Y. Higo, Y. Ueda, T. Kamiya, S. Kusumoto, and K. 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.
12. K. Inoue, Y. Miyamoto, D. M. Germán, and T. Ishio. Finding code-clone snippets in large source-code collection by ccgrep. In D. Taibi, V. Lenarduzzi, T. Kilamo, and S. Zacchiroli, editors, *Open Source Systems - 17th IFIP WG 2.13 International*

Conference, OSS 2021, Virtual Event, May 12-13, 2021, Proceedings, volume 624 of *IFIP Advances in Information and Communication Technology*, pages 28–41. Springer, 2021.

13. E. Juergens, F. Deissenboeck, B. Hummel, and S. Wagner. Do code clones matter? In *2009 IEEE 31st International Conference on Software Engineering*, pages 485–495. IEEE, 2009.
14. A. Karmakar and R. Robbes. What do pre-trained code models know about code? In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1332–1336. IEEE, 2021.
15. O. Karnalim. Tf-idf inspired detection for cross-language source code plagiarism and collusion. *Computer Science*, 21, 2020.
16. O. Karnalim et al. Explanation in code similarity investigation. *IEEE Access*, 9:59935–59948, 2021.
17. Y. A. Malkov and D. A. Yashunin. Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs. *IEEE transactions on pattern analysis and machine intelligence*, 42(4):824–836, 2018.
18. J. Martinez-Gil. A comprehensive review of stacking methods for semantic similarity measurement. *Machine Learning with Applications*, 10:100423, 2022.
19. J. Martinez-Gil. Source code clone detection using unsupervised similarity measures. In P. Bludau, R. Ramler, D. Winkler, and J. Bergsmann, editors, *Software Quality as a Foundation for Security - 16th International Conference on Software Quality, SWQD 2024, Vienna, Austria, April 23-25, 2024, Proceedings*, volume 505 of *Lecture Notes in Business Information Processing*, pages 21–37. Springer, 2024.
20. M. Novak, M. Joy, and D. 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.
21. C. K. Roy, J. R. Cordy, and R. Koschke. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of computer programming*, 74(7):470–495, 2009.
22. N. Saini, S. Singh, et al. Code clones: Detection and management. *Procedia computer science*, 132:718–727, 2018.
23. A. Satter and K. Sakib. A similarity-based method retrieval technique to improve effectiveness in code search. In *Companion Proceedings of the 1st International Conference on the Art, Science, and Engineering of Programming*, pages 1–3, 2017.
24. Spotify. Annoy. https://github.com/spotify/annoy, 05 2023. Approximate Nearest Neighbors in C++/Python optimized for memory usage and loading/saving to disk.
25. Z. Tronícek. Indexing source code and clone detection. *Inf. Softw. Technol.*, 144:106805, 2022.